证明

现在我们就来看看推导过程：

如果每一次对数组的操作花费 1 个单位时间，则时间复杂度为：

$$O\left(\sum_{k=1}^{\pi(n)}\frac{n}{p_k}\right)=O\left(n\sum_{k=1}^{\pi(n)}\frac{1}{p_k}\right)$$

其中 p_k 表示第 k 小的素数，$\pi(n)$ 表示 $\le n$ 的素数个数. $\sum_{k=1}^{\pi(n)}$ 表示第一层 for 循环，其中累加上界 $\pi(n)$ 为 `if (prime[i])` 进入 true 分支的次数；$\frac{n}{p_k}$ 表示第二层 for 循环的执行次数.

根据 Mertens 第二定理，存在常数 B_1 使得：

$$\sum_{k=1}^{\pi(n)}\frac{1}{p_k}=\log\log n+B_1+O\left(\frac{1}{\log n}\right)$$

所以 **Eratosthenes 筛法** 的时间复杂度为 $O(n\log\log n)$. 接下来我们证明 Mertens 第二定理的弱化版本 $\sum_{k\le\pi(n)}1/p_k=O(\log\log n)$：

根据 $\pi(n)=\Theta(n/\log n)$，可知第 n 个素数的大小为 $\Theta(n\log n)$. 于是就有

$$\begin{aligned}\sum_{k=1}^{\pi(n)}\frac{1}{p_k}&=O\left(\sum_{k=2}^{\pi(n)}\frac{1}{k\log k}\right)\\&=O\left(\int_2^{\pi(n)}\frac{\mathrm{d}x}{x\log x}\right)\\&=O(\log\log\pi(n))=O(\log\log n)\end{aligned}$$

当然，上面的做法效率仍然不够高效，应用下面几种方法可以稍微提高算法的执行效率.

筛至平方根

显然，要找到直到 n 为止的所有素数，仅对不超过 $\sqrt{n}$ 的素数进行筛选就足够了.

- C++

```cpp
vector<int> prime;
bool is_prime[N];

void Eratosthenes(int n) {
  is_prime[0] = is_prime[1] = false;
  for (int i = 2; i <= n; ++i) is_prime[i] = true;
  // i * i <= n 说明 i <= sqrt(n)
  for (int i = 2; i * i <= n; ++i) {
    if (is_prime[i])
      for (int j = i * i; j <= n; j += i) is_prime[j] = false;
  }
  for (int i = 2; i <= n; ++i)
    if (is_prime[i]) prime.push_back(i);
}
```

- Python

```python
prime = []
is_prime = [False] * N


def Eratosthenes(n):
```

```
    is_prime[0] = is_prime[1] = False
    for i in range(2, n + 1):
        is_prime[i] = True
    # 让 i 循环到 <= sqrt(n)
    for i in range(2, isqrt(n) + 1):  # `isqrt` 是 Python 3.8 新增的函数
        if is_prime[i]:
            for j in range(i * i, n + 1, i):
                is_prime[j] = False
    for i in range(2, n + 1):
        if is_prime[i]:
            prime.append(i)
```

这种优化不会影响渐近时间复杂度，实际上重复以上证明，我们将得到 $n \ln \ln \sqrt{n} + o(n)$，根据对数的性质，它们的渐近相同，但操作次数会明显减少.

只筛奇数

因为除 2 以外的偶数都是合数，所以我们可以直接跳过它们，只用关心奇数就好.

首先，这样做能让我们内存需求减半；其次，所需的操作大约也减半.

减少内存的占用

我们注意到筛选时只需要 `bool` 类型的数组. `bool` 数组的一个元素一般占用 1 字节（即 8 比特），但是存储一个布尔值只需要 1 个比特就足够了.

我们可以使用 位操作→9.4@p1099 的相关知识，将每个布尔值压到一个比特位中，这样我们仅需使用 n 比特（即 $\frac{n}{8}$ 字节）而非 n 字节，可以显著减少内存占用. 这种方式被称为「位级压缩」.

值得一提的是，存在自动执行位级压缩的数据结构，如 C++ 中的 `vector<bool>` 和 `bitset<>`.

另外，`vector<bool>` 和 `bitset<>` 对程序有常数优化，时间复杂度 $O(n \log \log n)$ 的埃氏筛在使用 `bitset<>` 或 `vector<bool>` 优化后，性能甚至超过时间复杂度 $O(n)$ 的欧拉筛.

参见 bitset: 与埃氏筛结合→4.3.4@p389.

分块筛选

由优化「筛至平方根」可知，不需要一直保留整个 `is_prime[1...n]` 数组. 为了进行筛选，只保留到 $\sqrt{n}$ 的素数就足够了，即 `prime[1...sqrt(n)]`. 并将整个范围分成块，每个块分别进行筛选. 这样，我们就不必同时在内存中保留多个块，而且 CPU 可以更好地处理缓存.

设 s 是一个常数，它决定了块的大小，那么我们就有了 $\lceil \frac{n}{s} \rceil$ 个块，而块 $k (k = 0 \ldots \lfloor \frac{n}{s} \rfloor)$ 包含了区间 $[ks, ks + s - 1]$ 中的数字. 我们可以依次处理块，也就是说，对于每个块 k，我们将遍历所有质数（从 1 到 $\sqrt{n}$）并使用它们进行筛选.

值得注意的是，我们在处理第一个数字时需要稍微修改一下策略：首先，应保留 $[1, \sqrt{n}]$ 中的所有的质数；第二，数字 0 和 1 应该标记为非素数. 在处理最后一个块时，不应该忘记最后一个数字 n 并不一定位于块的末尾.

以下实现使用块筛选来计算小于等于 n 的质数数量.

实现

```
int count_primes(int n) {
  constexpr static int S = 10000;
  vector<int> primes;
  int nsqrt = sqrt(n);
```

```
  vector<char> is_prime(nsqrt + 1, true);
  for (int i = 2; i <= nsqrt; i++) {
    if (is_prime[i]) {
      primes.push_back(i);
      for (int j = i * i; j <= nsqrt; j += i) is_prime[j] = false;
    }
  }
  int result = 0;
  vector<char> block(S);
  for (int k = 0; k * S <= n; k++) {
    fill(block.begin(), block.end(), true);
    int start = k * S;
    for (int p : primes) {
      int start_idx = (start + p - 1) / p;
      int j = max(start_idx, p) * p - start;
      for (; j < S; j += p) block[j] = false;
    }
    if (k == 0) block[0] = block[1] = false;
    for (int i = 0; i < S && start + i <= n; i++) {
      if (block[i]) result++;
    }
  }
  return result;
}
```

分块筛法的渐近时间复杂度与埃氏筛法是一样的(除非块非常小),但是所需的内存将缩小为 $O(\sqrt{n}+S)$,并且有更好的缓存结果. 另一方面,对于每一对块和区间 $[1,\sqrt{n}]$ 中的素数都要进行除法,而对于较小的块来说,这种情况要糟糕得多. 因此,在选择常数 S 时要保持平衡.

块大小 S 取 10^4 到 10^5 之间,可以获得最佳的速度.

线性筛法

埃氏筛法仍有优化空间,它会将一个合数重复多次标记. 有没有什么办法省掉无意义的步骤呢? 答案是肯定的.

如果能让每个合数都只被标记一次,那么时间复杂度就可以降到 $O(n)$ 了.

实现

- C++

```
vector<int> pri;
bool not_prime[N];

void pre(int n) {
  for (int i = 2; i <= n; ++i) {
    if (!not_prime[i]) {
      pri.push_back(i);
    }
    for (int pri_j : pri) {
      if (i * pri_j > n) break;
      not_prime[i * pri_j] = true;
      if (i % pri_j == 0) {
```

```
        // i % pri_j == 0
        // 换言之，i 之前被 pri_j 筛过了
        // 由于 pri 里面质数是从小到大的，所以 i 乘上其他的质数的结果一定会被
        // pri_j 的倍数筛掉，就不需要在这里先筛一次，所以这里直接 break
        // 掉就好了
        break;
      }
    }
  }
}
```

- Python

```
pri = []
not_prime = [False] * N


def pre(n):
    for i in range(2, n + 1):
        if not not_prime[i]:
            pri.append(i)
        for pri_j in pri:
            if i * pri_j > n:
                break
            not_prime[i * pri_j] = True
            if i % pri_j == 0:
                """
                i % pri_j == 0
                换言之，i 之前被 pri_j 筛过了
                由于 pri 里面质数是从小到大的，所以 i 乘上其他的质数的结果一定会被
                pri_j 的倍数筛掉，就不需要在这里先筛一次，所以这里直接 break
                掉就好了
                """
                break
```

上面的这种 **线性筛法** 也称为 **Euler 筛法**（欧拉筛法）.

Note

注意到筛法求素数的同时也得到了每个数的最小质因子.

筛法求欧拉函数

注意到在线性筛中，每一个合数都是被最小的质因子筛掉. 比如设 p_1 是 n 的最小质因子，$n' = \frac{n}{p_1}$，那么线性筛的过程中 n 通过 $n' \times p_1$ 筛掉.

观察线性筛的过程，我们还需要处理两个部分，下面对 $n' \bmod p_1$ 分情况讨论.

如果 $n' \bmod p_1 = 0$，那么 n' 包含了 n 的所有质因子.

$$\varphi(n) = n \times \prod_{i=1}^{s} \frac{p_i - 1}{p_i}$$

$$= p_1 \times n' \times \prod_{i=1}^{s} \frac{p_i - 1}{p_i}$$

$$= p_1 \times \varphi(n')$$

那如果 $n' \bmod p_1 \neq 0$ 呢，这时 n' 和 p_1 是互质的，根据欧拉函数性质，我们有：

$$\varphi(n) = \varphi(p_1) \times \varphi(n')$$

$$= (p_1 - 1) \times \varphi(n')$$

实现

- C++

```cpp
vector<int> pri;
bool not_prime[N];
int phi[N];

void pre(int n) {
  phi[1] = 1;
  for (int i = 2; i <= n; i++) {
    if (!not_prime[i]) {
      pri.push_back(i);
      phi[i] = i - 1;
    }
    for (int pri_j : pri) {
      if (i * pri_j > n) break;
      not_prime[i * pri_j] = true;
      if (i % pri_j == 0) {
        phi[i * pri_j] = phi[i] * pri_j;
        break;
      }
      phi[i * pri_j] = phi[i] * phi[pri_j];
    }
  }
}
```

- Python

```python
pri = []
not_prime = [False] * N
phi = [0] * N


def pre(n):
    phi[1] = 1
    for i in range(2, n + 1):
        if not not_prime[i]:
            pri.append(i)
            phi[i] = i - 1
        for pri_j in pri:
            if i * pri_j > n:
                break
```

```
            not_prime[i * pri_j] = True
            if i % pri_j == 0:
                phi[i * pri_j] = phi[i] * pri_j
                break
            phi[i * pri_j] = phi[i] * phi[pri_j]
```

筛法求莫比乌斯函数

定义

根据莫比乌斯函数的定义，设 n 是一个合数，p_1 是 n 的最小质因子，$n' = \frac{n}{p_1}$，有：

$$\mu(n) = \begin{cases} 0 & n' \bmod p_1 = 0 \\ -\mu(n') & \text{otherwise} \end{cases}$$

若 n 是质数，有 $\mu(n) = -1$.

实现

- C++

```cpp
vector<int> pri;
bool not_prime[N];
int mu[N];

void pre(int n) {
  mu[1] = 1;
  for (int i = 2; i <= n; ++i) {
    if (!not_prime[i]) {
      mu[i] = -1;
      pri.push_back(i);
    }
    for (int pri_j : pri) {
      if (i * pri_j > n) break;
      not_prime[i * pri_j] = true;
      if (i % pri_j == 0) {
        mu[i * pri_j] = 0;
        break;
      }
      mu[i * pri_j] = -mu[i];
    }
  }
}
```

- Python

```python
pri = []
not_prime = [False] * N
mu = [0] * N


def pre(n):
    mu[1] = 1
    for i in range(2, n + 1):
        if not not_prime[i]:
            pri.append(i)
            mu[i] = -1
```

```
        for pri_j in pri:
            if i * pri_j > n:
                break
            not_prime[i * pri_j] = True
            if i % pri_j == 0:
                mu[i * pri_j] = 0
                break
            mu[i * pri_j] = -mu[i]
```

筛法求约数个数

用 d_i 表示 i 的约数个数，num_i 表示 i 的最小质因子出现次数.

约数个数定理

定理：若 $n=\prod_{i=1}^{m} p_i^{c_i}$ 则 $d_i=\prod_{i=1}^{m}(c_i+1)$.

证明：我们知道 $p_i^{c_i}$ 的约数有 $p_i^0, p_i^1, ..., p_i^{c_i}$ 共 c_i+1 个，根据乘法原理，n 的约数个数就是 $\prod_{i=1}^{m}(c_i+1)$.

实现

因为 d_i 是积性函数，所以可以使用线性筛.

在这里简单介绍一下线性筛实现原理.

1. 当 i 为质数时，$num_i \leftarrow 1, d_i \leftarrow 2$，同时设 $q=\left\lfloor \frac{i}{p} \right\rfloor$，其中 p 为 i 的最小质因子.
2. 当 p 为 q 的质因子时，$num_i \leftarrow num_q+1, d_i \leftarrow \frac{d_q}{num_i} \times (num_i+1)$.
3. 当 p, q 互质时，$num_i \leftarrow 1, d_i \leftarrow d_q \times (num_i+1)$.

- C++

```
vector<int> pri;
bool not_prime[N];
int d[N], num[N];

void pre(int n) {
  d[1] = 1;
  for (int i = 2; i <= n; ++i) {
    if (!not_prime[i]) {
      pri.push_back(i);
      d[i] = 2;
      num[i] = 1;
    }
    for (int pri_j : pri) {
      if (i * pri_j > n) break;
      not_prime[i * pri_j] = true;
      if (i % pri_j == 0) {
        num[i * pri_j] = num[i] + 1;
        d[i * pri_j] = d[i] / num[i * pri_j] * (num[i * pri_j] + 1);
        break;
      }
      num[i * pri_j] = 1;
      d[i * pri_j] = d[i] * 2;
    }
  }
}
```

- Python

```python
pri = []
not_prime = [False] * N
d = [0] * N
num = [0] * N


def pre(n):
    d[1] = 1
    for i in range(2, n + 1):
        if not not_prime[i]:
            pri.append(i)
            d[i] = 2
            num[i] = 1
        for pri_j in pri:
            if i * pri_j > n:
                break
            not_prime[i * pri_j] = True
            if i % pri_j == 0:
                num[i * pri_j] = num[i] + 1
                d[i * pri_j] = d[i] // num[i * pri_j] * (num[i * pri_j] + 1)
                break
            num[i * pri_j] = 1
            d[i * pri_j] = d[i] * 2
```

筛法求约数和

f_i 表示 i 的约数和，g_i 表示 i 的最小质因子的 $p^0 + p^1 + p^2 + ...p^k$.

实现

- C++

```cpp
vector<int> pri;
bool not_prime[N];
int g[N], f[N];

void pre(int n) {
  g[1] = f[1] = 1;
  for (int i = 2; i <= n; ++i) {
    if (!not_prime[i]) {
      pri.push_back(i);
      g[i] = i + 1;
      f[i] = i + 1;
    }
    for (int pri_j : pri) {
      if (i * pri_j > n) break;
      not_prime[i * pri_j] = true;
      if (i % pri_j == 0) {
        g[i * pri_j] = g[i] * pri_j + 1;
        f[i * pri_j] = f[i] / g[i] * g[i * pri_j];
        break;
      }
      f[i * pri_j] = f[i] * f[pri_j];
```

```
      g[i * pri_j] = 1 + pri_j;
    }
  }
}
```

- Python

```python
pri = []
not_prime = [False] * N
f = [0] * N
g = [0] * N


def pre(n):
    g[1] = f[1] = 1
    for i in range(2, n + 1):
        if not not_prime[i]:
            pri.append(i)
            g[i] = i + 1
            f[i] = i + 1
        for pri_j in pri:
            if i * pri_j > n:
                break
            not_prime[i * pri_j] = True
            if i % pri_j == 0:
                g[i * pri_j] = g[i] * pri_j + 1
                f[i * pri_j] = f[i] // g[i] * g[i * pri_j]
                break
            f[i * pri_j] = f[i] * f[pri_j]
            g[i * pri_j] = 1 + pri_j
```

一般的积性函数

假如一个 积性函数→9.11.1@p1160 f 满足：对于任意质数 p 和正整数 k，可以在关于 k 的低次多项式时间内计算 $f(p^k)$，那么可以在 $O(n)$ 时间内筛出 $f(1), f(2), ..., f(n)$ 的值.

设合数 n 的质因子分解是 $\prod_{i=1}^{k} p_i^{\alpha_i}$，其中 $p_1 < p_2 < ... < p_k$ 为质数，我们在线性筛中记录 $g_n = p_1^{\alpha_1}$，假如 n 被 $x \cdot p$ 筛掉（p 是质数），那么 g 满足如下递推式：

$$g_n = \begin{cases} g_x \cdot px \bmod p = 0 \\ p \qquad \text{otherwise} \end{cases}$$

假如 $n = g_n$，说明 n 就是某个质数的次幂，可以 $O(1)$ 计算 $f(n)$；否则，$f(n) = f(\frac{n}{g_n}) \cdot f(g_n)$.

本节部分内容译自博文 Решето Эратосфена[1] 与其英文翻译版 Sieve of Eratosthenes[2]．其中俄文版版权协议为 Public Domain + Leave a Link；英文版版权协议为 CC-BY-SA 4.0．

外部参考资料

[1] Решето Эратосфена  [2] Sieve of Eratosthenes 

9.11.7 分解质因数

引入

给定一个正整数 $N \in \mathbf{N}_+$，试快速找到它的一个 非平凡因数→9.11.1@p1160.

考虑朴素算法，因数是成对分布的，N 的所有因数可以被分成两块，即 $[2, \sqrt{N}]$ 和 $[\sqrt{N}+1, N)$. 只需要把 $[2, \sqrt{N}]$ 里的数遍历一遍，再根据除法就可以找出至少两个因数了．这个方法的时间复杂度为 $O(\sqrt{N})$.

当 $N \geq 10^{18}$ 时，这个算法的运行时间我们是无法接受的，希望有更优秀的算法．一种想法是通过随机的方法，猜测一个数是不是 N 的因数，如果运气好可以在 $O(1)$ 的时间复杂度下求解答案，但是对于 $N \geq 10^{18}$ 的数据，成功猜测的概率是 $\frac{1}{10^{18}}$，期望猜测的次数是 10^{18}．如果是在 $[2, \sqrt{N}]$ 里进行猜测，成功率会大一些．我们希望有方法来优化猜测.

朴素算法

最简单的算法即为从 $[2, \sqrt{N}]$ 进行遍历.

- C++

```cpp
vector<int> breakdown(int N) {
  vector<int> result;
  for (int i = 2; i * i <= N; i++) {
    if (N % i == 0) {  // 如果 i 能够整除 N，说明 i 为 N 的一个质因子.
      while (N % i == 0) N /= i;
      result.push_back(i);
    }
  }
  if (N != 1) {  // 说明再经过操作之后 N 留下了一个素数
    result.push_back(N);
  }
  return result;
}
```

- Python

```python
def breakdown(N):
    result = []
    for i in range(2, int(sqrt(N)) + 1):
        if N % i == 0:  # 如果 i 能够整除 N，说明 i 为 N 的一个质因子.
            while N % i == 0:
                N //= i
            result.append(i)
    if N != 1:  # 说明再经过操作之后 N 留下了一个素数
        result.append(N)
    return result
```

我们能够证明 `result` 中的所有元素即为 `N` 的全体素因数.

证明 `result` 中的元素即为 N 的全体素因数

首先考察 `N` 的变化．当循环进行到 `i` 结束时，由于刚执行结束 `while(N % i == 0) N /= i` 部分，`i` 不再整除 `N`．而且，每次除去一个因子，都能够保证 `N` 仍整除 N．这两点保证了，当循环进行到 `i` 开始时，`N` 是 N 的一个因子，且不被任何小于 `i` 的整数整除.

其次证明 `result` 中的元素均为 N 的因子．当循环进行到 `i` 时，能够在 `result` 中存入 `i` 的条件是 `N % i == 0`，这说明 `i` 整除 `N`，且已经说明 `N` 是 N 的因子，故而有 `i` 是 N 的因子．当对 `i` 的循环结束时，若 `N` 不为一，也会存入 `result`．此时它根据前文，也必然是 N 的一个因子.

其次证明 `result` 中均为素数．我们假设存在一个在 `result` 中的合数 K，则必然存在大于一且不超过 $\sqrt{K}$ 的 `i`，满足 `i` 是 K 的一个因子．这样的 K 不可能作为循环中的某个 `i` 存入 `result`，因为第一段已经说明，当循环到 K 时，`N` 不被任何小于 K 的 `i` 整除．这样的 K 也不可能在循环结束后

> 加入，因为循环退出的条件是 `i * i > N`，故而已经遍历完了所有不超过 $\sqrt{K}$ 的 `i`，而且据上文所说，这些 `i` 绝不能整除目前的 `N`，亦即 K.
>
> 最后证明，所有 N 的素因子必然出现在 `result` 中. 不妨假设 p 是 N 的一个素因子，但并没有出现在 `result` 中. 根据上文的讨论，p 不可能是循环中出现过的 `i`. 设 `i` 是退出循环前最后的 `i`，则 `i` 严格小于 p，而退出循环后的 `N` 不被之前的 `i` 整除，故而 p 整除 `N`. 所以最后的 `N` 大于一，则根据前文所述，它必然是素数，则 `N` 就等于 p，必会在最后加入 `result`，与假设矛盾.

值得指出的是，如果开始已经打了一个素数表的话，时间复杂度将从 $O(\sqrt{N})$ 下降到 $O(\frac{\sqrt{N}}{\ln N})$. 去 筛法→9.11.6@p1199 处查阅更多打表的信息.

例题：CF 1445C[1]

Pollard Rho 算法

引入

利用暴力算法获得一个非平凡因子的复杂度为 $O(p)=O(\sqrt{N})$，这里，p 是 N 的最小素因子. 而下面要介绍的 Pollard Rho 算法是一种随机化算法，可以在 $O(\sqrt{p})=O(N^{1/4})$ 的期望复杂度获得一个非平凡因子（**注意**！非平凡因子不一定是素因子）.

它的核心想法是，对于一个随机自映射 $f:\mathbf{Z}_p\to\mathbf{Z}_p$，从任何一点 x_1 出发，迭代计算 $x_n=f(x_{n-1})$，将在 $O(\sqrt{p})$ 期望时间内进入循环. 如果能够找到 $x_i\equiv x_j\pmod p$，则 p 整除 $\gcd(|x_i-x_j|,N)$，这一最大公约数就是 N 的一个非平凡因子.

要理解进入循环的期望时间为 $O(\sqrt{p})$，可以从生日悖论中获得启发.

生日悖论

不考虑出生年份（假设每年都是 365 天），问：一个房间中至少多少人，才能使其中两个人生日相同的概率达到 50%?

解：假设一年有 n 天，房间中有 k 人，用整数 $1,2,\ldots,k$ 对这些人进行编号. 假定每个人的生日均匀分布于 n 天之中，且两个人的生日相互独立.

设 k 个人生日互不相同为事件 A，则事件 A 的概率为

$$P(A)=\prod_{i=0}^{k-1}\frac{n-i}{n}$$

至少有两个人生日相同的概率为 $P(\overline{A})=1-P(A)$. 根据题意可知 $P(\overline{A})\geq\frac{1}{2}$，那么就有

$$P(A)=\prod_{i=0}^{k-1}\frac{n-i}{n}\leq\frac{1}{2}$$

由不等式 $1+x\leq \mathrm{e}^x$ 可得

$$P(A)\leq\prod_{i=1}^{k-1}\exp\left(-\frac{i}{n}\right)=\exp\left(-\frac{k(k-1)}{2n}\right)$$

因此

$$\exp\left(-\frac{k(k-1)}{2n}\right)\leq\frac{1}{2}\Longrightarrow P(A)\leq\frac{1}{2}$$

将 $n=365$ 代入，解得 $k\geq 23$. 所以一个房间中至少 23 人，使其中两个人生日相同的概率达到 50%，但这个数学事实十分反直觉，故称之为一个悖论.

[18] https://en.wikipedia.org/wiki/Birthday_problem#Reverse_problem[2]

当 $k > 56$，$n = 365$ 时，出现两个人同一天生日的概率将大于 99%[18]. 那么在一年有 n 天的情况下，当房间中有 $\frac{1}{2}(\sqrt{8n\ln 2+1}+1) \approx \sqrt{2n\ln 2}$ 个人时，至少有两个人的生日相同的概率约为 50%.

类似地可以计算，随机均匀地选取一列生日，首次获得重复生日需要的人数的期望也是 $O(\sqrt{n})$. 设这一人数为 X，则

$$E(X) = \sum_{x=0}^{n} P(X \geq x+1) = \sum_{x=0}^{n} \frac{n!}{(n-x)!n^x} = \sqrt{\frac{\pi n}{2}} + \frac{2}{3} + o(1).$$

这启发我们，如果可以随机选取一列数字，出现重复数字需要的抽样规模的期望也是 $O(\sqrt{n})$ 的.

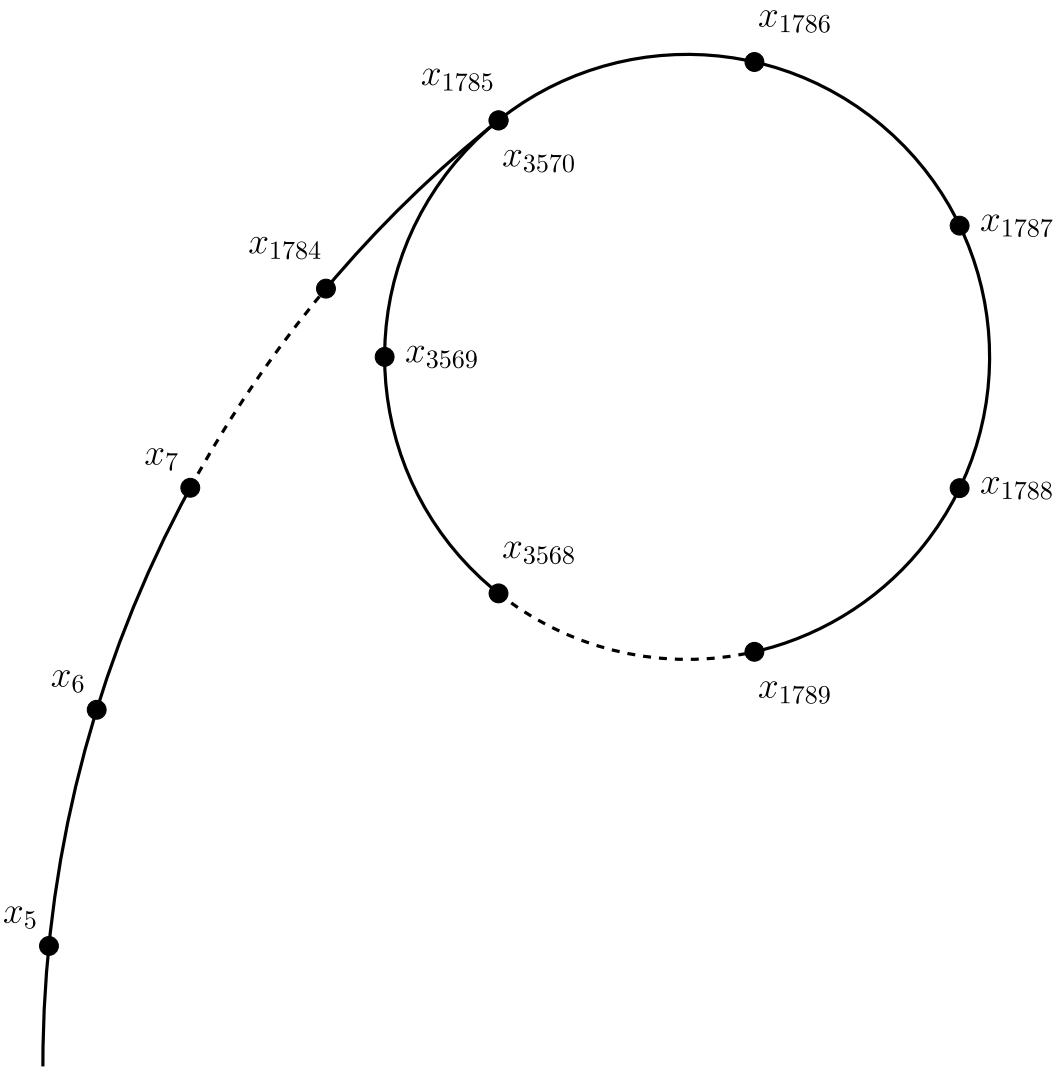
利用最大公约数求出一个约数

实际构建一列模 p 的随机数列并不现实，因为 p 正是需要求的. 所以，我们通过 $f(x) = (x^2 + c) \bmod N$ 来生成一个伪随机数序列 $\{x_i\}$：随机取一个 x_1，令 $x_2 = f(x_1)$, $x_3 = f(x_2)$, ..., $x_i = f(x_{i-1})$，其中 $c \in [1, N)$ 是一个随机选取的常数.

这里选取的函数容易计算，且往往可以生成相当随机的序列. 但它并不是完全随机的. 举个例子，设 $N = 50$, $c = 6$, $x_1 = 1$，$f(x)$ 生成的数据为

$$1, 7, 5, 31, 17, 45, 31, 17, 45, 31, ...$$

可以发现数据在 x_4 以后都在 $31, 17, 45$ 之间循环. 如果将这些数如下图一样排列起来，会发现这个图像酷似一个 ρ，算法也因此得名 rho.



更重要的是，这样的函数确实提供了 $\mathbf{Z}_p$ 上一个自映射. 也就是说，它满足性质：如果 $x \equiv y \pmod{p}$，则 $f(x) \equiv f(y) \pmod{p}$.

证明

若 $x \equiv y \pmod{p}$，则 $x^2 + c \equiv y^2 + c \pmod{p}$. 注意到，$f(x) = x^2 + c - k_x N$，这里 k_x 是一个依赖于 x 的整数，且 $p \mid N$，所以有 $f(x) \equiv x^2 + c \pmod{p}$，因而 $f(x) \equiv f(y) \pmod{p}$.

作为 $\mathbf{Z}_p$ 上的伪随机自映射反复迭代得到的序列，$\{x_n \bmod p\}$ 在 $O(\sqrt{p})$ 的期望时间内就会出现重复. 只要我们观察到这样的重复 $x_i \equiv x_j \pmod{p}$，就可以根据 $\gcd(|x_i - x_j|, N)$ 求出一个 N 的非平凡因子. 注意到，由于 p 未知，我们并没有办法直接判断重复的发生，一个简单的判断方法正是 $\gcd(|x_i - x_j|, N)$ 严格大于一.

这一算法并不是总能成功的，因为 $\gcd(|x_i - x_j|, N)$ 可能等于 N. 也就是说，$x_i \equiv x_j \pmod{N}$. 此时，$\{x_n \bmod p\}$ 首次发生重复时，恰好 $\{x_n\}$ 也发生重复了. 我们没有得到一个非平凡因子. 而且，$\{x_n\}$ 开始循环后，再继续迭代也没有意义了，因为之后只会重复这一循环. 该算法应输出分解失败，需要更换 $f(x)$ 中选取的 c 重新分解.

根据上文分析，理论上，任何满足 $\forall x \equiv y \pmod{p}, f(x) \equiv f(y) \pmod{p}$，且能够保证一定伪随机性的函数 $f(x)$（例如某些多项式函数）都可以用在此处. 实践中，主要使用 $f(x) = x^2 + c$ $(c \neq 0, -2)$.[19]

实现

我们需要实现的算法，能够在迭代过程中快速判断 $\{x_n \bmod p\}$ 是否已经出现重复. 将 f 看成以 $\mathbf{Z}_p$ 为顶点的有向图上的边，我们实际要实现的是一个判环算法. 只是将判等改为了判断 $\gcd(|x_i - x_j|, N)$ 是否大于一.

Floyd 判环

假设两个人在赛跑，A 的速度快，B 的速度慢，经过一定时间后，A 一定会和 B 相遇，且相遇时 A 跑过的总距离减去 B 跑过的总距离一定是圈长的倍数.

设 $a = f(0), b = f(f(0))$，每一次更新 $a = f(a), b = f(f(b))$，只要检查在更新过程中 a 和 b 是否相等，如果相等了，那么就出现了环.

我们每次令 $d = \gcd(|x_i - x_j|, N)$，判断 d 是否满足 $1 < d < N$，若满足则可直接返回 d. 如果 $d = N$，则说明 $\{x_i\}$ 已经形成环，在形成环时就不能再继续操作了，直接返回 N 本身，并且在后续操作里调整随机常数 c，重新分解.

基于 Floyd 判环的 Pollard Rho 算法

- C++

```cpp
ll Pollard_Rho(ll N) {
  if (N == 4) return 2;  // 因为一开始跳了两步，所以需要特判一下 4
  ll c = rand() % (N - 1) + 1;
  ll t = f(0, c, N);
  ll r = f(f(0, c, N), c, N);
  while (t != r) {
    ll d = gcd(abs(t - r), N);
    if (d > 1) return d;
    t = f(t, c, N);
    r = f(f(r, c, N), c, N);
  }
```

[19] Menezes, Alfred J.; van Oorschot, Paul C.; Vanstone, Scott A. (2001). Handbook of Applied Cryptography. Section 3.11 and 3.12.

```
  return N;
}
```

- Python

```python
import random


def Pollard_Rho(N):
    if N == 4:
        return 2  # 因为一开始跳了两步，所以需要特判一下 4
    c = random.randint(1, N - 1)
    t = f(0, c, N)
    r = f(f(0, c, N), c, N)
    while t != r:
        d = gcd(abs(t - r), N)
        if d > 1:
            return d
        t = f(t, c, N)
        r = f(f(r, c, N), c, N)
    return N
```

Brent 判环

实际上，Floyd 判环算法可以有常数上的改进．Brent 判环从 $k=1$ 开始递增 k，在第 k 轮，让 A 等在原地，B 向前移动 2^k 步，如果在过程中 B 遇到了 A，则说明已经得到环，否则让 A 瞬移到 B 的位置，然后继续下一轮．

可以证明[20]，这样得到环之前需要调用 f 的次数永远不大于 Floyd 判环算法．原论文中的测试表明，Brent 判环需要的平均时间相较于 Floyd 判环减少了 24%．

倍增优化

无论是 Floyd 判环还是 Brent 判环，迭代次数都是 $O(\sqrt{p})$ 的．但是每次迭代都用 gcd 判断是否成环会拖慢算法运行速度．可以通过乘法累积来减少求 gcd 的次数．

简单来说，如果 $\gcd(a,N)>1$，那么 $\gcd(ab \bmod N, N)=\gcd(ab,N)>1$ 对于任意 $b\in\mathbf{N}_+$ 都成立．也就是说，如果计算得到 $\gcd(\prod|x_i-x_j| \bmod N, N)>1$，那么必然有其中一对 (x_i,x_j) 满足 $\gcd(|x_i-x_j|,N)>1$．如果该乘积在某一时刻得到零，则分解失败，退出并返回 N 本身．

如果每 k 对计算一次 gcd，则算法复杂度降低到 $O(\sqrt{p}+k^{-1}\sqrt{p}\log N)$，这里，$\log N$ 为单次计算 gcd 的开销．注意到 k 和 $\log N$ 大致同阶时，可以得到 $O(\sqrt{p})$ 的期望复杂度．具体实现中，大多选取 $k=127$．

这里提供 Brent 判环且加上倍增优化的 Pollard Rho 算法实现．

实现

- C++

```
ll Pollard_Rho(ll x) {
  ll t = 0;
  ll c = rand() % (x - 1) + 1;
  ll s = t;
```

[20] Brent, R. P. (1980), An improved Monte Carlo factorization algorithm, BIT Numerical Mathematics, 20(2): 176–184, doi:10.1007/BF01933190

```
  int step = 0, goal = 1;
  ll val = 1;
  for (goal = 1;; goal <<= 1, s = t, val = 1) {
    for (step = 1; step <= goal; ++step) {
      t = f(t, c, x);
      val = val * abs(t - s) % x;
      // 如果 val 为 0，退出重新分解
      if (!val) return x;
      if (step % 127 == 0) {
        ll d = gcd(val, x);
        if (d > 1) return d;
      }
    }
    ll d = gcd(val, x);
    if (d > 1) return d;
  }
}
```

- Python

```
from random import randint
from math import gcd


def Pollard_Rho(x):
    c = randint(1, x - 1)
    s = t = f(0, c, x)
    goal = val = 1
    while True:
        for step in range(1, goal + 1):
            t = f(t, c, x)
            val = val * abs(t - s) % x
            if val == 0:
                return x  # 如果 val 为 0，退出重新分解
            if step % 127 == 0:
                d = gcd(val, x)
                if d > 1:
                    return d
        d = gcd(val, x)
        if d > 1:
            return d
        s = t
        goal <<= 1
        val = 1
```

复杂度

Pollard Rho 算法中的期望迭代次数为 $O(\sqrt{p})$，这里 p 是 N 的最小素因子．具体实现无论是采用 Floyd 判环还是 Brent 判环，如果不使用倍增优化，期望复杂度都是 $O(\sqrt{p}\log N)$；在加上倍增优化后，可以近似得到 $O(\sqrt{p})$ 的期望复杂度．

值得一提的是，前文分析基于的是完全随机的自映射函数，但 Pollard Rho 算法实际使用的是伪随机函数，所以该算法并没有严格的复杂度分析，实践中通常跑得较快．

例题：求一个数的最大素因子

例题：P4718【模板】Pollard-Rho 算法[3]

对于一个数 n，用 Miller–Rabin 算法→9.11.3@p1179 判断是否为素数，如果是就可以直接返回了，否则用 Pollard Rho 算法找一个因子 p，将 n 除去因子 p．再递归分解 n 和 p，用 Miller–Rabin 判断是否出现质因子，并用 max_factor 更新就可以求出最大质因子了．由于这个题目的数据过于庞大，用 Floyd 判环的方法是不够的，这里采用倍增优化的方法．

实现

```cpp
#include <algorithm>
#include <cstdlib>
#include <ctime>
#include <iostream>

using namespace std;
using ll = long long;
using ull = unsigned long long;

int t;
ll max_factor, n;

ll gcd(ll a, ll b) {
  if (b == 0) return a;
  return gcd(b, a % b);
}

ll bmul(ll a, ll b, ll m) {  // 快速乘
  ull c = (ull)a * (ull)b - (ull)((long double)a / m * b + 0.5L) * (ull)m;
  if (c < (ull)m) return c;
  return c + m;
}

ll qpow(ll x, ll p, ll mod) {  // 快速幂
  ll ans = 1;
  while (p) {
    if (p & 1) ans = bmul(ans, x, mod);
    x = bmul(x, x, mod);
    p >>= 1;
  }
  return ans;
}

bool Miller_Rabin(ll p) {  // 判断素数
  if (p < 2) return false;
  if (p == 2) return true;
  if (p == 3) return true;
  ll d = p - 1, r = 0;
  while (!(d & 1)) ++r, d >>= 1;  // 将d处理为奇数
  for (ll k = 0; k < 10; ++k) {
    ll a = rand() % (p - 2) + 2;
    ll x = qpow(a, d, p);
    if (x == 1 || x == p - 1) continue;
```

```
    for (int i = 0; i < r - 1; ++i) {
      x = bmul(x, x, p);
      if (x == p - 1) break;
    }
    if (x != p - 1) return false;
  }
  return true;
}

ll Pollard_Rho(ll x) {
  ll s = 0, t = 0;
  ll c = (ll)rand() % (x - 1) + 1;
  int step = 0, goal = 1;
  ll val = 1;
  for (goal = 1;; goal *= 2, s = t, val = 1) {  // 倍增优化
    for (step = 1; step <= goal; ++step) {
      t = (bmul(t, t, x) + c) % x;
      val = bmul(val, abs(t - s), x);
      if ((step % 127) == 0) {
        ll d = gcd(val, x);
        if (d > 1) return d;
      }
    }
    ll d = gcd(val, x);
    if (d > 1) return d;
  }
}

void fac(ll x) {
  if (x <= max_factor || x < 2) return;
  if (Miller_Rabin(x)) {              // 如果x为质数
    max_factor = max(max_factor, x);  // 更新答案
    return;
  }
  ll p = x;
  while (p >= x) p = Pollard_Rho(x);  // 使用该算法
  while ((x % p) == 0) x /= p;
  fac(x), fac(p);  // 继续向下分解x和p
}

int main() {
  cin >> t;
  while (t--) {
    srand((unsigned)time(NULL));
    max_factor = 0;
    cin >> n;
    fac(n);
    if (max_factor == n)  // 最大的质因数即自己
      cout << "Prime\n";
    else
      cout << max_factor << '\n';
  }
```

```
96   return 0;
97 }
98
```

参考资料与链接

[1] CF 1445C

[2] https://en.wikipedia.org/wiki/Birthday_problem#Reverse_problem

[3] P4718【模板】Pollard-Rho 算法

9.11.8 裴蜀定理 & 一次不定方程

裴蜀定理揭示了最大公约数与整数线性组合之间的深刻联系，是数论中最基础也最重要的结论之一. 基于此，本文进一步讨论了一次不定方程的求解方法.

裴蜀定理

裴蜀定理（Bézout's lemma），也译作贝祖定理，或称作贝祖等式（Bézout's identity），给出了一个整数能够表示为两个整数的整系数线性组合的充分必要条件.

裴蜀定理

设 a, b 是不全为零的整数. 那么，对于任意整数 x, y，都有 $\gcd(a,b) \mid ax+by$ 成立；而且，存在整数 x, y，使得 $ax+by=\gcd(a,b)$ 成立.

证明

记 $d=\gcd(a,b)$. 因为 $d \mid a, b$，所以，存在整数 u, v 使得 $a=du,\ b=dv$ 成立. 因此，总有

$$ax+by=d(ux+vy).$$

这就说明 $d \mid ax+by$.

反过来，需要说明存在 x, y 使得等式成立. 由于 $\gcd(a,b)=\gcd(-a,b)=\gcd(a,-b)$，所以不妨设 a, b 都是非负数. 如果 a, b 之一是 0，不妨设 $b=0$，那么它们的最大公约数为 $d=a$，显然有 $(x,y)=(1,0)$ 使得等式成立. 接下来，考虑 a, b 均不为零的情形，此时 a, b 都是正数.

考虑辗转相除法的过程，有

$$\begin{aligned}
a &= q_1 b + r_1, && 0 \le r_1 < b,\\
b &= q_2 r_1 + r_2, && 0 \le r_2 < r_1,\\
r_1 &= q_3 r_2 + r_3, && 0 \le r_3 < r_2,\\
&\cdots\\
r_{n-3} &= q_{n-1} r_{n-2} + r_{n-1}, && 0 \le r_{n-1} < r_{n-2},\\
r_{n-2} &= q_n r_{n-1} + r_n, && 0 \le r_n < r_{n-1},\\
r_{n-1} &= q_{n+1} r_n.
\end{aligned}$$

由于最大公约数是 d，最后一步辗转相除时，一定有 $r_n=d$. 所以，倒数第二个等式可以写作

$$d=r_n=r_{n-2}-q_n r_{n-1}.$$

从倒数第三个等式中解出

$$r_{n-1}=r_{n-3}-q_{n-1}r_{n-2}$$

再代入上式，就可以消去 r_{n-1}：

$$\begin{aligned}
d &= r_{n-2}-q_n(r_{n-3}-q_{n-1}r_{n-2})\\
&= (1+q_n q_{n-1})r_{n-2}-q_n r_{n-3}.
\end{aligned}$$

类似地，可以逐步地消去所有 $r_{n-2}, r_{n-3}, \cdots, r_2, r_1$，最终得到

$$d = xa + yb.$$

这就证明了存在 x, y 使得 $ax + by = d$ 成立．由前文分析可知，这也证明了原命题．

此处，关于存在性的证明是构造性的，它同时给出了该系数的一种计算方法．这一计算方法就是 扩展欧几里得算法→9.11.4@p1188．

考虑裴蜀定理在 $\gcd(a, b) = 1$ 时的特殊情形，可以得到如下推论：

推论

整数 a, b 互素，当且仅当存在整数 x, y，使得 $ax + by = 1$ 成立．

多个整数的情形

裴蜀定理可以推广到多个整数的情形．

定理

设 $a_1, a_2, \cdots, a_n$ 是不全为零的整数．那么，对于任意整数 $x_1, x_2, \cdots, x_n$，都有 $\gcd(a_1, a_2, \cdots, a_n) \mid a_1x_1 + a_2x_2 + \cdots + a_nx_n$ 成立；而且，存在整数 $x_1, x_2, \cdots, x_n$，使得 $\gcd(a_1, a_2, \cdots, a_n) = a_1x_1 + a_2x_2 + \cdots + a_nx_n$ 成立．

证明

利用 $\gcd(a_1, a_2, \cdots, a_n) = \gcd(\gcd(a_1, a_2, \cdots, a_{n-1}), a_n)$ 这一点，对 n 进行归纳即可．

例题

Codeforces 510 D. Fox And Jumping[1]

给出 $n \le 300$ 张卡片，分别有 l_i 和 c_i．在一条无限长的纸带上，你可以选择花 c_i 的钱来购买卡片 i，从此以后可以向左或向右跳 l_i 个单位任意次．问你至少花多少元钱才能够跳到纸带上全部位置．若不行，输出 -1．

解答

分析该问题，发现想要跳到每一个格子上，必须使得所选数 $l_{i_1}, \cdots, l_{i_k}$ 通过数次相加或相减得出的绝对值为 1．也就是说，存在整数 $x_1, \cdots, x_k$ 使得 $l_{i_1}x_1 + \cdots + l_{i_k}x_k = 1$．由多个整数的裴蜀定理，这相当于从数组 $l_1, \cdots, l_n$ 中选择若干个数，满足它们的最大公约数为 1，同时要求代价和最小．

解法 1：将最小代价和看作是最短路径问题，可以用 Dijkstra 算法求解．图的顶点处存储了当前的最大公约数的取值．图的起点是 0，要到达的目标点是 1．每走一步，就从当前顶点 x 出发，沿着长度为 c_i 的边走到顶点 $\gcd(x, l_i)$．这一算法的时间复杂度为 $O(n^2 \log n)$．

解法 2：从数组 $l_1, \cdots, l_n$ 选择若干个数，满足它们的最大公约数为 1，且代价和最小，由此可以想到 0-1 背包问题．

设 $f_{i,j}$ 表示考虑前 i 个数且最大公约数为 j 的最小代价，则有转移方程：

$$f_{i,j} = \min\left\{f_{i-1,j},\ \min_{\gcd(k,l_i)=j}\{f_{i-1,k} + c_i\}\right\}.$$

DP 后最终的总代价即为 $f_{n,1}$．

> 如同一般的 0-1 背包问题，可以用滚动数组优化，去掉第一维．而这里 300 个数可以组成的最大公约数 j 是很稀疏的，可以用哈希表储存．
>
> 实际上，这里解法 1 建出的图便是解法 2 中动态规划的状态转移图，解法 2 相当于用动态规划求有向无环图的最短路，因此解法 1 和解法 2 是等价的．但解法 2 无需储存全图，同时 DP 的时间复杂度为 $O(n+m)$（其中 m 为状态转移图的边数），相比 Dijkstra 算法更低，因此解法 2 在时间和空间上更优．

一次不定方程

一次不定方程（linear Diophantine equation）是形如

$$a_1x_1 + a_2x_2 + \cdots + a_nx_n = b$$

的不定方程，其中，$a_1, a_2, \cdots, a_n$ 都是整数．本节的目标是寻找它的全体整数解．

两个变量的情形

首先考虑二元一次不定方程：

$$a_1x_1 + a_2x_2 = b.$$

裴蜀定理指出，该方程有解，当且仅当

$$d = \gcd(a_1, a_2) \mid b.$$

接下来，假设这一条件成立．利用扩展欧几里得算法可以求出方程 $a_1x_1 + a_2x_2 = d$ 的一组整数解 (x_1^*, x_2^*)．由此，可以得到原方程的一组特解

$$(x_1^\circ, x_2^\circ) = \left(\frac{b}{d}x_1^*, \frac{b}{d}x_2^*\right).$$

要得到全部解，可以考虑将原方程与恒等式 $a_1x_1^\circ + a_2x_2^\circ = b$ 相减，就有

$$a_1(x_1 - x_1^\circ) + a_2(x_2 - x_2^\circ) = 0.$$

这是一个关于 $(x_1 - x_1^\circ, x_2 - x_2^\circ)$ 的齐次一次不定方程，它有通解

$$(x_1 - x_1^\circ, x_2 - x_2^\circ) = \left(t\frac{a_2}{d}, -t\frac{a_1}{d}\right). \quad (t \in \mathbf{Z})$$

因此，原方程的通解就是

$$(x_1, x_2) = \left(x_1^\circ + t\frac{a_2}{d}, x_2^\circ - t\frac{a_1}{d}\right). \quad (t \in \mathbf{Z})$$

这是直线 $a_1x_1 + a_2x_2 = b$ 上一系列等间隔分布的整点．

多个变量的情形

解决了二元的情形，多元的情形也就容易解决了．对于 n 元一次不定方程

$$a_1x_1 + a_2x_2 + \cdots + a_nx_n = b, \quad (n > 2)$$

由裴蜀定理可知，方程有解当且仅当

$$\gcd(a_1, a_2, \cdots, a_n) \mid b.$$

和二元的情形类似，多元一次不定方程的通解同样可以写作

$$(x_1^\circ, x_2^\circ, \cdots, x_n^\circ) + \sum_{k=1}^{n-1} t_k(x_1^{(k)}, x_2^{(k)}, \cdots, x_n^{(k)})$$

的形式，其中，x° 为一个特解，$x^{(k)}$ 为相应的齐次方程的 $(n-1)$ 个线性无关的解．

要求出通解的具体形式，可以通过将 n 元方程转化为 $(n-1)$ 元方程来完成．记 $d_1 = \gcd(a_1, a_2)$，那么，根据裴蜀定理，$a_1x_1 + a_2x_2$ 的全体恰为 d_1 的所有倍数．因此，可以首先求解 $(n-1)$ 元一次不定方程：

$$d_1y_1 + a_3x_3 + a_4x_4 + \cdots + a_nx_n = b.$$

设得到的它的通解为

$$y_1 = y_1^\circ + \sum_{k=2}^{n-1} t_k y_1^{(k)},$$

$$x_i = x_i^\circ + \sum_{k=2}^{n-1} t_k x_i^{(k)}, \quad i = 3, \cdots, n.$$

设 $a_1x_1 + a_2x_2 = d_1$ 的一组特解为 (x_1^*, x_2^*)，那么，根据前一节的讨论可知，关于 x_1, x_2 的二元一次不定方程 $a_1x_1 + a_2x_2 = d_1y_1$ 的通解就是

$$x_1 = x_1^* y_1 + t_1 \frac{a_2}{d_1},\ x_2 = x_2^* y_1 - t_1 \frac{a_1}{d_1}.$$

代入 y_1 的表达式，就得到原方程的通解

$$x_1 = x_1^* y_1^\circ + t_1 \frac{a_2}{d_1} + \sum_{k=2}^{n-1} t_k x_1^* y_1^{(k)},$$

$$x_2 = x_2^* y_1^\circ - t_1 \frac{a_1}{d_1} + \sum_{k=2}^{n-1} t_k x_2^* y_1^{(k)},$$

$$x_i = x_i^\circ + \sum_{k=2}^{n-1} t_k x_i^{(k)}, \quad i = 3, \cdots, n.$$

Frobenius 硬币问题

裴蜀定理给出了一个整数可以由若干个整数线性表出的充分必要条件．与此紧密相关的是 **Frobenius 硬币问题**（Frobenius coin problem）：

- 如果硬币共有 $a_1, a_2, \cdots, a_n$ 等若干种整数面值，且 $\gcd(a_1, a_2, \cdots, a_n) = 1$，那么，不能够由这些硬币组成的最大整数是多少?

同样是在考察整数 k 什么时候可以表示为 $a_1x_1 + a_2x_2 + \cdots + a_nx_n$ 的形式，裴蜀定理中 x_i 可以是任意整数，而 Frobenius 硬币问题中 x_i 只能是自然数.

只有一种硬币的情形是平凡的，因为只能有 $a_1 = 1$，所有自然数都可以由它表示．而 $n > 2$ 的情形又太过复杂，所以，本节仅讨论 $n = 2$ 的情形.

Sylvester 定理

在 1882 年，Sylvester 完全解决了 $n = 2$ 时的 Frobenius 硬币问题：

定理（Sylvester）

对于互素的正整数 a_1, a_2，不能够写作 $a_1x_1 + a_2x_2$ $(x_1, x_2 \in \mathbf{N})$ 的最大整数是 $C = a_1a_2 - a_1 - a_2$．而且，对于所有 $k \in \mathbf{Z}$，整数 k 和 $C - k$ 中有且只有一个可以写作该形式.

为表述方便，称可以写作 $a_1x_1 + a_2x_2$ $(x_1, x_2 \in \mathbf{N})$ 形式的整数为 **可表示的**.

证明一

由于 a_1, a_2 互素，对于任意整数 k，方程 $a_1x_1 + a_2x_2 = k$ 一定有解，且通解为

$$(x_1, x_2) = (x_1^\circ + ta_2, x_2^\circ - ta_1). \quad (t \in \mathbf{Z})$$

取 t 为 x_2° 对 a_1 作带余除法得到的商，那么，余数 $x_2 = x_2^\circ - ta_1$ 位于 0 和 $a_1 - 1$ 之间．考察此时得到的一组解 (x_1, x_2)．因为 x_2 是它能够取到的最小非负整数值，所以 k 可表示当且仅当 $x_1 \geq 0$.

第一步：证明大于 C 的整数都是可表示的.

当 $k > C$ 时，有

$$a_1x_1 = k - a_2x_2 > C - a_2(a_1 - 1) = -a_1.$$

所以，$x_1 > -1$，也就是说，$x_1 \geq 0$．这说明，(x_1, x_2) 是一组自然数解．此时，k 可以写作所求形式．

第二步：证明 C 不可表示．进而，C 是最大的不可表示的整数，且 k 和 $C-k$ 并非都可表示．

反证法．假设 C 可以表示，即存在 $x_1, x_2 \in \mathbf{N}$ 使得 $a_1x_1 + a_2x_2 = C$ 成立．代入 C 的表达式，可知

$$a_1a_2 = a_1(x_1+1) + a_2(x_2+1).$$

因此，$a_2 \mid (x_1+1)$ 且 $a_1 \mid (x_2+1)$．又因为 x_1+1, x_2+1 都是正数，所以，有

$$a_1a_2 \geq a_1a_2 + a_2a_1 = 2a_1a_2.$$

矛盾．这就说明 C 不可表示．结合第一步，它也就是不可表示的最大整数．

如果 k 和 $C-k$ 都可以表示，那么，将 k 和 $C-k$ 的表示中的系数相加就得到 C 的表示中的系数，这与 C 不可表示矛盾，故而 k 和 $C-k$ 至多只有一个可以表示．

第三步：证明如果 k 不可表示，那么 $C-k$ 一定是可表示的．

设 (x_1, x_2) 是前文所设的方程 $a_1x_1 + a_2x_2 = k$ 的整数解．那么，前文已经说明 k 不可表示，就等价于 $x_1 < 0$．因此，有

$$C - k = a_1a_2 - a_1 - a_2 - a_1x_1 - a_2x_2 = a_1(-1-x_1) + a_2(a_1 - 1 - x_2).$$

其中，$-1-x_1$ 和 $a_1 - 1 - x_2$ 都是非负整数，所以，$C-k$ 可以表示．

证明二

此处仅证明 $C = a_1a_2 - a_1 - a_2$ 是最大的不可表示的整数，其余部分的证明类似证明一．

考虑模 a_2 意义下，每个剩余类中最小的可表示的自然数．因为同一个剩余类中的不同自然数可以通过加减若干个 a_2 互相转化，所以，在讨论最小可表示数时，只需要考虑加减 a_1 的可能性就可以了．由于 a_1 和 a_2 互素，所以，每个剩余类中最小的可表示的自然数恰好就是 a_1 的倍数

$$0,\ a_1,\ 2a_1,\ \cdots,\ (a_2-1)a_1.$$

因此，最大的不可表示数为

$$\max_{0 \leq i < a_2} ia_1 - a_2 = (a_2-1)a_1 - a_2 = C.$$

几何意义

将方程 $a_1x_1 + a_2x_2 = k$ 看作是一条直线．那么，k 可表示，当且仅当这条直线在第一象限（包括坐标轴）内通过至少一个整点．当 $k < a_1a_2$ 时，这条直线在第一象限至多只能通过一个整点．因此，对于 $0 \leq k < a_1a_2$，整数 k 可以表示，当且仅当这条直线在第一象限恰好通过一个整点．

因此，对于 $k < a_1a_2$，小于等于 k 且可以表示的自然数的数量，恰好等于第一象限内直线 $a_1x_1 + a_2x_2 = k$ 下的整点个数（包含边界上的点）．这一数量就等于

$$\sum_{i=0}^{\lfloor k/a_1 \rfloor} \left(\left\lfloor \frac{k - ia_1}{a_2} \right\rfloor + 1 \right).$$

这是经典的直线下整点问题，可以用 类欧几里得算法→9.11.29@p1410 在 $O(\log \min\{a_1, a_2, k\})$ 时间求解．

习题

- Luogu P3951 NOIP2017 提高组 小凯的疑惑/蓝桥杯 2013 省 买不到的数目[2]

外部参考资料

[1] Codeforces 510 D. Fox And Jumping

[2] Luogu P3951 NOIP2017 提高组 小凯的疑惑/蓝桥杯 2013 省 买不到的数目

9.11.9 费马小定理 & 欧拉定理

Authors: PeterlitsZo, Tiphereth-A

本文讨论费马小定理、欧拉定理及其扩展．这些定理解决了任意模数下任意大指数的幂的计算问题．

费马小定理

费马小定理（Fermat's little theorem）是数论中最基础的定理之一．它也是 Fermat 素性测试→9.11.3@p1179 的理论基础．

费马小定理

设 p 是素数．对于任意整数 a 且 $p \nmid a$，都成立 $a^{p-1} \equiv 1 \pmod p$.

定理

设 p 是素数．对于任意整数 a，都成立 $a^p \equiv a \pmod p$.

这两个同余关系在 $p \nmid a$ 时是等价的；而在 $p \mid a$ 时，$a^p \equiv 0 \equiv a \pmod p$ 平凡地成立．因此，这两个命题是等价的．这两个命题常常都称作费马小定理．

证明一

设 p 是素数，且 $p \nmid a$．首先证明：对于 $i = 1, 2, \cdots, p-1$，余数 $ia \bmod p$ 各不相同．反证法．如果有 $1 \le i < j < p$ 使得

$$ia \bmod p = ja \bmod p \iff (j-i)a \equiv 0. \pmod p$$

但是，$(j-i)$ 和 a 都不是 p 的倍数，这显然矛盾．

换句话说，这些余数是 $\{1, 2, \cdots, p-1\}$ 的一个排列．因此，有

$$\prod_{i=1}^{p-1} i = \prod_{i=1}^{p-1} (ia \bmod p) \equiv \prod_{i=1}^{p-1} ia = a^{p-1} \prod_{i=1}^{p-1} i. \pmod p$$

这说明

$$(a^{p-1} - 1) \prod_{i=1}^{p-1} i \equiv 0. \pmod p$$

也就是说，等式左侧是 p 的倍数，但是 $i = 1, 2, \cdots, p-1$ 都不是 p 的倍数，所以，只能有 $p \mid (a^{p-1} - 1)$，亦即费马小定理成立．

证明二

注意到费马小定理的第二种表述对于所有 $a \in \mathbf{N}$ 都成立，因此，可以考虑使用数学归纳法．负整数的情形容易转化为非负整数的情形．

归纳起点为 $0^p \equiv 0 \pmod p$，显然成立．假设它对于 $a \in \mathbf{N}$ 成立，需要证明的是，它对于 $a+1$ 也成立．由二项式定理可知

$$(a+1)^p = a^p + \binom{p}{1} a^{p-1} + \binom{p}{2} a^{p-2} + \cdots + \binom{p}{p-1} a + 1.$$

除了首尾两项，组合数的表达式 $\binom{p}{k}=\dfrac{p!}{k!(p-k)!}$ 中，p 都能整除分子，而不能整除分母，因此，这些系数对于 $k\neq 0,p$ 都是 p 的倍数．因此，有

$$(a+1)^p\equiv a^p+1\equiv a+1.\pmod p$$

其中，第二步应用了归纳假设．因此，利用数学归纳法可知，费马小定理成立．

费马小定理的逆命题并不成立．即使对于所有与 n 互素的 a，都有 $a^{n-1}\equiv 1\pmod n$，n 也未必是素数．相关讨论详见 Fermat 素性测试→9.11.3@p1179 一节．

欧拉定理

欧拉定理（Euler's theorem）将费马小定理推广到了一般模数的情形，但仍然要求底数与模数互素．

欧拉定理

对于整数 $m>0$ 和整数 a，且 $\gcd(a,m)=1$，有 $a^{\varphi(m)}\equiv 1\pmod m$，其中，$\varphi(\cdot)$ 为 欧拉函数→9.11.5@p1196．

证明

与费马小定理的证明一类似，仍然是取一个与 m 互质的数列，再进行操作．考虑集合

$$R=\{r\in\mathbf{N}:0<r<m,\ \gcd(r,m)=1\}.$$

这是模 m 的 既约剩余系→9.11.1@p1160．根据欧拉函数的定义可知，$|R|=\varphi(m)$．类似上文，将它们乘以 a 相当于对该集合重新排列：

$$R=\{ar\bmod m:r\in R\}.$$

这是因为，容易验证 $\gcd(ar,m)=1$ 且不同的 $r_1,r_2\in R$ 对应的 $ar_1\bmod m$ 和 $ar_2\bmod m$ 也一定不同．因此，有

$$\prod_{r\in R}r\equiv\prod_{r\in R}ar=a^{\varphi(m)}\prod_{r\in R}r.\pmod m$$

再次重复之前的论证，消去 $\prod_{r\in R}r$，就得到 $a^{\varphi(m)}\equiv 1\pmod m$．

对于素数 p，有 $\varphi(p)=p-1$，因此，费马小定理是欧拉定理的一个特例．另外，欧拉定理中的指数 $\varphi(m)$ 在一般情形下并非使得该式成立的最小指数．它可以改进到 $\lambda(m)$，其中，$\lambda(\cdot)$ 是 Carmichael 函数→9.11.18@p1275．关于相关结论的代数背景，可以参考 整数同余类的乘法群→9.16.3@p1830 一节．

扩展欧拉定理

扩展欧拉定理[21]进一步将结论推广到了底数与指数不互素的情形．由此，它彻底解决了任意模数下任意底数的幂次计算问题，将它们转化为指数小于 $2\varphi(m)$ 的情形，从而可以通过 快速幂→9.7@p1131 在 $O(\log\varphi(m))$ 时间内计算．

扩展欧拉定理

对于任意正整数 m、整数 a 和非负整数 k，有

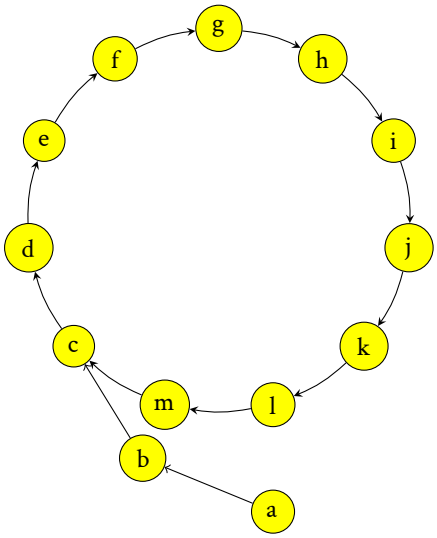
$$a^k\equiv\begin{cases}a^{k\bmod\varphi(m)}, & \gcd(a,m)=1,\\ a^k, & \gcd(a,m)\neq 1,k<\varphi(m),\\ a^{(k\bmod\varphi(m))+\varphi(m)}, & \gcd(a,m)\neq 1,k\geq\varphi(m).\end{cases}\pmod m$$

[21]这一名字主要出现在算法竞赛圈中，而并非该结论的通用名称．

第二种情形是在说，如果 $k < \varphi(m)$，那么，就无需继续降幂，直接应用快速幂即可；而第三种和第一种情形的最大区别是，通过取余降幂之后，是否需要加上一项 $\varphi(m)$．当然，将第一种情形合并进入第二、三种情形也是正确的．

直观理解

在严格证明定理之前，可以首先直观理解定理的含义．



考虑余数 $a^k \bmod m$ 随着 k 增大而变化的情况．由于余数的取值一定在区间 $[0, m)$ 内，而 k 有无限多个．将 $a^k \bmod m \mapsto a^{k+1} \bmod m$ 看作这些余数结点之间的有向边．那么，一定可以构成如图所示的循环．

扩展欧拉定理说明，这些循环可能是纯循环（第一种情形）或者混循环（第二、三种情形）．纯循环中，没有结点存在两个前驱，而混循环中就会出现这样的情形．因此，对于一般的情况，只需要能够求出循环节的长度和进入循环节之前的长度，就可以利用这个性质进行降幂．

严格证明

本节给出扩展欧拉定理的严格证明．

证明

首先说明，存在 $k_0 \in \mathbf{N}$，使得整数 a 和 $m' := \dfrac{m}{\gcd(a^{k_0}, m)}$ 互素．为此，设 $\nu_p(n)$ 是整数 n 的质因数分解中素数 p 的幂次，那么，不妨取

$$k_0 = \max\left\{\left\lceil \frac{\nu_p(m)}{\nu_p(a)} \right\rceil : \nu_p(a) > 0\right\}.$$

因为 m 中所有和 a 的公共素因子的幂次都已经包含在 a^{k_0} 中，所以，a 就与 m 中剩下的因子 $m' = \dfrac{m}{\gcd(a^{k_0}, m)}$ 互素．

进而，对 $k \geq k_0$ 考察同余关系

$$b \equiv a^k. \pmod m$$

由于 $\gcd(a^{k_0}, m) = \gcd(a^k, m) \mid b$，所以，将等式两侧（包括模数）同时除以 $\gcd(a^{k_0}, m)$，就有

$$\frac{b}{\gcd(a^{k_0}, m)} \equiv \frac{a^{k_0}}{\gcd(a^{k_0}, m)} \cdot a^{k-k_0}. \pmod{m'}$$

此时，因为 a 与模数 m' 互素，可以直接应用欧拉定理，得到

$$\frac{b}{\gcd(a^{k_0}, m)} \equiv \frac{a^{k_0}}{\gcd(a^{k_0}, m)} \cdot a^{(k-k_0) \bmod \varphi(m')}. \pmod{m'}$$

因此，再将因子 $\gcd(a^{k_0}, m)$ 乘回去，就得到

$$b \equiv a^{k_0} \cdot a^{(k-k_0) \bmod \varphi(m')} = a^{k_0+(k-k_0) \bmod \varphi(m')}. \pmod m$$

这就得到了扩展欧拉定理的形式．式子说明，循环节的长度是 $\varphi(m')$，而进入循环节之前的长度为 k_0．

此处得到的参数比扩展欧拉定理中的更紧，但是相对来说，这些参数的计算并不容易．可以说明，这些参数可以放宽到扩展欧拉定理中的情形．首先，利用 欧拉函数的表达式→9.11.5@p1196 可知，因为 $m' \mid m$，所以 $\varphi(m') \mid \varphi(m)$．也就是说，$\varphi(m)$ 也是它的循环节．其次，k_0 也可以放宽到 $\varphi(m)$．这是因为对于所有 $m \in \mathbf{N}_+$ 和任意 $p \mid m$，都有

$$\begin{aligned}\varphi(m) \geq \varphi(p^{\nu_p(m)}) &= (p-1)p^{\nu_p(m)-1} \geq p^{\nu_p(m)-1} \\ &= (1+(p-1))^{\nu_p(m)-1} \geq 1+(p-1)(\nu_p(m)-1) \\ &\geq 1+(\nu_p(m)-1) = \nu_p(m).\end{aligned}$$

其中，第二行的不等式利用了二项式展开，并只保留常数项和一次项．因此，有

$$k_0 \leq \max\{\nu_p(m) : p \in \mathbf{P}\} \leq \varphi(m).$$

这就完全证明了所述结论．

例题

本节通过一道例题展示扩展欧拉定理的一个经典应用——计算任意模数下的幂塔．**幂塔**（power tower）指形如 $A \uparrow (B \uparrow (C \uparrow (D \uparrow \cdots)))$ 的式子，其中，$\uparrow$ 是 Knuth 箭头记号，而 $A, B, C, D, \cdots$ 是一系列非负整数．

Library Checker - Tetration Mod[1]

T 组测试．每组测试中，给定 A, B, M，求 $(A \uparrow\uparrow B) \bmod M$．其中，$A \uparrow\uparrow B$ 表示由 B 个 A 组成的幂塔．或者，形式化地，定义

$$A \uparrow\uparrow B = \begin{cases} 1, & B = 0, \\ A \uparrow (A \uparrow\uparrow (B-1)), & B > 0. \end{cases}$$

规定 $0^0 = 1$．

解答

利用 $A \uparrow\uparrow B$ 的定义，递归计算即可．要计算 $(A \uparrow\uparrow B) \bmod M$，只需要应用扩展欧拉定理，计算 $(A \uparrow\uparrow (B-1)) \bmod \varphi(M)$．由于 $\varphi(\varphi(n)) \leq n/2$ 对所有 $n \geq 2$ 都成立，所以，递归过程一定在 $O(\log M)$ 步内完成．由于需要应用扩展欧拉定理，所以需要区分当前的计算结果是否严格小于当前模数．为此，只需要在取余的时候多判断一步即可．另外，需要注意边界情况的处理．

参考代码

```cpp
#include <iostream>

// Calculate Euler's totient for n.
int phi(int n) {
  int res = n;
  for (int i = 2; i * i <= n; ++i) {
    if (n % i == 0) {
      res = res / i * (i - 1);
```

```cpp
      while (n % i == 0) n /= i;
    }
  }
  if (n > 1) res = res / n * (n - 1);
  return res;
}

// Find remainder as in the exponent of extended Euler theorem.
int mod(long long v, int m) { return v >= m ? v % m + m : v; }

// Modular power.
int pow(int a, int b, int m) {
  long long res = 1, po = a;
  for (; b; b >>= 1) {
    if (b & 1) res = mod(res * po, m);
    po = mod(po * po, m);
  }
  return res;
}

// Modular tetration.
int tetra(int a, int b, int m) {
  if (a == 0) return !(b & 1);
  if (b == 0 || m == 1) return 1;
  if (b == 1) return mod(a, m);
  return pow(a, tetra(a, b - 1, phi(m)), m);
}

int main() {
  int t;
  std::cin >> t;
  for (; t; --t) {
    int a, b, m;
    std::cin >> a >> b >> m;
    std::cout << (tetra(a, b, m) % m) << std::endl;
  }
}

```

⚠ 取模技巧正确性证明

本实现利用

$$r_m(v) = \begin{cases} v \bmod m, & \text{if } v < m, \\ v \bmod m + m, & \text{if } v \geq m \end{cases}$$

代替取模，相当于同时记录了余数信息 $v \bmod m$ 和越界信息 $v \geq m$. 扩展欧拉定理只能说明，当指数的余数信息和越界信息都正确时，取幂能够得到正确的余数信息；但是，越界信息的正确性无法保证. 例如，在计算 $r_6(2^4)$ 时，指数需要对 $\varphi(6) = 2$ 取模，而 $r_2(4) = 2$（即余数为 0 且越界），进行取幂后将得到 $r_6(2^2) = 4$；与正确答案 $r_6(2^4) = 10$ 相比，模 6 的余数正确，但是越界信息丢失. 但可以证明，尽管越界信息可能丢失，余数信息却总是正确的. 因此，基于这一取模技巧的取幂算法也总是正确的.

考虑算法的迭代过程．假设从最内层开始向外层函数回溯，第一次越界信息错误发生在计算 $r_m(a^k)$ 时．$k<\varphi(m)$ 时指数精确，不会出错，故设 $k\geq\varphi(m)$；此时 $k\geq k \bmod \varphi(m)+\varphi(m)$，即取模只会低估指数，越界信息不会凭空产生，出错的唯一可能是本应记录越界而实际未记录，即

$$a^{k \bmod \varphi(m)+\varphi(m)}<m\leq a^k.$$

该不等式只能在 $a\geq 2$ 时成立，所以左侧表达式不小于 $2^{\varphi(m)}$．但是，$2^{\varphi(m)}<m$ 只会发生在 $m=6$ 时．[22]代入上式，就得到 $a^{k \bmod 2+2}<6\leq a^k$．左侧表达式中，若 k 是偶数，则指数是 2，有 $a=2$ 且 $k\geq 4$；若 k 是奇数，则指数是 3，无解．综上，丢失越界信息只发生在 $m=6$，$a=2$ 且 $k\geq 4$ 是偶数时．

下面说明，这一情形下丢失越界信息并不影响后续计算．只考虑 $a_i\geq 2$ 的非平凡情形（$a_i\leq 1$ 时幂值不超过 1，不会越界）．内层模数为 6，故回溯一层后的模数 m_1 满足 $\varphi(m_1)=6$，只能是 $7,9,14,18$ 之一．回溯到该层时，需要计算 $r_{m_1}(a_1^{k_1})$，而 k_1 本应为 10，实际为 4．但对所有 a_1 和这些 m_1，都有 $a_1^4\equiv a_1^{10} \pmod{m_1}$．为说明这一点，注意到前文扩展欧拉定理的 证明→9.11@p1160 中已经得到：对任何 a 和 m，幂次余数 $a^k \bmod m$ 进入循环节之前的长度 k_0 都不超过 $\nu_+(m):=\max\{\nu_p(m):p\mid m\}$，且循环节长度整除 $\varphi(m)$．而 $\nu_+(m_1)\leq 2$，故 $k_1=4$ 已进入循环节，且它与 10 相差 $\varphi(m_1)=6$，亦即上式总成立．所以余数信息正确．再看越界信息．由于内层真实值为 $2^k\geq 16>10$，故 $a_1^{2^k}\geq a_1^{10}\geq m_1$，本层总应该标记越界．但本层只会返回 a_1^4 是否越界；唯一出错的可能是 $a_1^4<m_1$，即只能有 $a_1=2$，$m_1=18$．其余情形的越界信息已正确标记，继续回溯自然正确．而 $a_1=2$，$m_1=18$ 时，本应返回 34，实际返回 16，需再回溯一层，此时 m_2 只能是 $19,27,38,54$ 之一．此时总有 $\nu_+(m_2)\leq 3<16$ 且 $34-16=\varphi(m_2)$，余数信息正确；且 $a_2^{16}\geq m_2$ 恒成立，越界信息也正确．所以此时结果正确，后续回溯也必然正确．

由于各层模数严格递减，$m=6$ 至多出现一次．综上，利用该取模技巧计算多层幂次时，每一层的余数信息都是正确的，至多只有两层的越界信息是错误的．注意，证明中没有假定各层取幂的底数相同，故结论对不同底数的情形也成立．

习题

- Luogu P5091【模板】扩展欧拉定理[2]
- Codeforces 906 D. Power Tower[3]
- Luogu P3747 [六省联考 2017] 相逢是问候[4]
- Luogu P4139 上帝与集合的正确用法[5]
- Luogu P3934 [Ynoi Easy Round 2016] 炸脖龙 I[6]
- Luogu P6736「Wdsr-2」白泽教育[7]

参考资料与注释

- Fermat's little theorem - Wikipedia[8]
- Euler's theorem - Wikipedia[9]
- Hardy, Godfrey Harold, and Edward Maitland Wright. An introduction to the theory of numbers. Oxford university press, 1979.

[1] Library Checker - Tetration Mod

[2] Luogu P5091【模板】扩展欧拉定理

[3] Codeforces 906 D. Power Tower

[4] Luogu P3747 [六省联考 2017] 相逢是问候

[5] Luogu P4139 上帝与集合的正确用法

[6] Luogu P3934 [Ynoi Easy Round 2016] 炸脖龙 I

[22]根据 Kendall, R. and Osborn, R. (1965) Two Simple Lower Bounds for the Euler ϕ-Function. Texas Journal of Science, 17, 324-326 一文，对于 $m>6$ 总有 $\varphi(m)\geq\sqrt{m}$．直接计算可知，对于 $m\geq 16$，总有 $\sqrt{m}\geq\log_2 m$．因此，不等式 $\varphi(m)\geq\log_2 m$ 对 $m\geq 16$ 都成立．其余可逐一验证．

[7] Luogu P6736「Wdsr-2」白泽教育  [8] Fermat's little theorem - Wikipedia  [9] Euler's theorem - Wikipedia 

9.11.10 模逆元

本文介绍模意义下乘法运算的逆元，并讨论它的常见求解方法.

基本概念

非零实数 $a \in \mathbf{R}$ 的乘法逆元就是它的倒数 a^{-1}. 类似地，数论中也可以定义一个整数 a 在模 m 意义下的逆元 $a^{-1} \bmod m$，或简单地记作 a^{-1}. 这就是 **模逆元**（modular multiplicative inverse），也称作 **数论倒数**.

逆元

对于整数 a 和正整数 m，如果存在 b 使得 $ab \equiv 1 \pmod m$，就称 b 是 a 在模 m 意义下的 **逆元**（inverse）.

这相当于说，b 是线性同余方程 $ax \equiv 1 \pmod m$ 的解. 根据 线性同余方程→9.11.11@p1233 的性质可知，当且仅当 $\gcd(a, m) = 1$，即 a, m 互素时，逆元 $a^{-1} \bmod m$ 存在，且在模 m 的意义下是唯一的.

单个逆元的求法

利用扩展欧几里得算法或快速幂法，可以在 $O(\log m)$ 时间内求出单个整数的逆元.

扩展欧几里得算法

求解逆元，就相当于求解线性同余方程. 因此，可以使用 扩展欧几里得算法→9.11.4@p1188 在 $O(\log \min\{a, m\})$ 时间内求解逆元. 同时，由于逆元对应的线性方程比较特殊，可以适当地简化相应的步骤.

参考实现

- C++

```cpp
// Extended Euclidean algorithm.
void ex_gcd(int a, int b, int& x, int& y) {
  if (!b) {
    x = 1;
    y = 0;
  } else {
    ex_gcd(b, a % b, y, x);
    y -= a / b * x;
  }
}

// Returns the modular inverse of a modulo m.
// Assumes that gcd(a, m) = 1, so the inverse exists.
int inverse(int a, int m) {
  int x, y;
  ex_gcd(a, m, x, y);
  return (x % m + m) % m;
}

```

- Python

```python
# Extended Euclidean algorithm.
```

```python
def ex_gcd(a, b):
    if b == 0:
        return 1, 0
    else:
        x1, y1 = ex_gcd(b, a % b)
        x = y1
        y = x1 - (a // b) * y1
        return x, y


# Returns the modular inverse of a modulo m.
# Assumes that gcd(a, m) = 1, so the inverse exists.
def inverse(a, m):
    x, y = ex_gcd(a, m)
    return (x % m + m) % m

```

这一算法适用于所有逆元存在的情形.

快速幂法

这一方法主要适用于模数是素数 p 的情形. 此时, 由 费马小定理→9.11.9@p1223 可知对于任意 $a \perp p$ 都有

$$a \cdot a^{p-2} = a^{p-1} \equiv 1 \pmod{p}.$$

根据逆元的唯一性可知, 逆元 $a^{-1} \bmod p$ 就等于 $a^{p-2} \bmod p$, 因此可以直接使用 快速幂→9.7@p1131 在 $O(\log p)$ 时间内计算:

参考实现

- C++

```cpp
// Binary exponentiation.
int pow(int a, int b, int m) {
  long long res = 1, po = a;
  for (; b; b >>= 1) {
    if (b & 1) res = res * po % m;
    po = po * po % m;
  }
  return res;
}

// Returns the modular inverse of a prime modulo p.
int inverse(int a, int p) { return pow(a, p - 2, p); }

```

- Python

```python
# Returns the modular inverse of a prime modulo p.
# Use built-in pow function.
def inverse(a, p):
    return pow(a, p - 2, p)

```

当然，理论上，这一方法可以利用 欧拉定理→9.11.9@p1223 推广到一般的模数 m 的情形，即利用 $a^{\varphi(m)-1} \bmod m$ 计算逆元．但是，单次求解 欧拉函数→9.11.5@p1196 $\varphi(m)$ 并不容易，因此该算法在一般情况下效率不高．

多个逆元的求法

有些场景下，需要快速处理出多个整数 $a_1, a_2, \cdots, a_n$ 在模 m 意义下的逆元．此时，逐个求解逆元，总共需要 $O(n \log m)$ 的时间．实际上，如果将它们统一处理，就可以在 $O(n + \log m)$ 的时间内求出所有整数的逆元．

考虑序列 $\{a_i\}$ 的前缀积：

$$S_0 = 1,\ S_i = a_i S_{i-1},\ i = 1, 2, \cdots, n.$$

只要每个 a_i 都与 m 互素，它们的乘积 S_n 就与 m 互素．因此，可以通过前文所述算法求出 $S_n^{-1} \bmod m$ 的值．因为乘积的逆元就是逆元的乘积，所以，从 S_n^{-1} 出发，反向遍历序列就能求出每个 S_i 的逆元：

$$S_{i-1}^{-1} = a_i S_i^{-1} \bmod m,\ i = n, n-1, \cdots, 1.$$

由此，单个 a_i 的逆元可以通过下式计算：

$$a_i^{-1} = S_{i-1} S_i^{-1} \bmod m,\ i = 1, 2, \cdots, n.$$

参考实现如下：

参考实现

- C++

```cpp
// Returns the modular inverses for each x in a modulo m.
// Assume x mod m exists for each x in a.
std::vector<int> batch_inverse(const std::vector<int>& a, int m) {
  int n = a.size();
  std::vector<int> prod(n);
  long long s = 1;
  for (int i = 0; i < n; ++i) {
    // prod[i] = product of a[0...i-1]; prod[0] = 1.
    prod[i] = s;
    s = s * a[i] % m;
  }
  // s = product of all elements in a.
  s = inverse(s, m);
  std::vector<int> res(n);
  for (int i = n - 1; i >= 0; --i) {
    res[i] = s * prod[i] % m;
    s = s * a[i] % m;
  }
  return res;
}

```

- Python

```python
# Returns the modular inverses for each x in a modulo m.
# Assume x mod m exists for each x in a.
def batch_inverse(a, m):
    n = len(a)
    prod = [0] * n
    s = 1
```

```
    for i in range(n):
        # prod[i] = product of a[0...i-1]; prod[0] = 1.
        prod[i] = s
        s = s * a[i] % m
    # s = product of all elements in a.
    s = inverse(s, m)
    res = [0] * n
    for i in reversed(range(n)):
        res[i] = s * prod[i] % m
        s = s * a[i] % m
    return res


```

算法中，只求了一次单个元素的逆元，因此总的时间复杂度是 $O(n+\log m)$ 的.

线性时间预处理逆元

如果要预处理前 n 个正整数在素数模 p 下的逆元，还可以通过本节将要讨论的递推关系在 $O(n)$ 时间内计算．这一方法常用于组合数计算中前 n 个正整数阶乘逆元的预处理.

对于 $1<i<p$ 的正整数 i，考察带余除法:

$$p=\left\lfloor\frac{p}{i}\right\rfloor i+(p \bmod i).$$

将该等式对素数 p 取模，就得到

$$0\equiv\left\lfloor\frac{p}{i}\right\rfloor i+(p \bmod i) \pmod p.$$

将等式两边同时乘以 $i^{-1}(p \bmod i)^{-1}$ 就得到

$$i^{-1}\equiv-\left\lfloor\frac{p}{i}\right\rfloor(p \bmod i)^{-1} \pmod p.$$

这就是用于线性时间递推求逆元的公式．由于 $p \bmod i<i$，这一公式将求解 $i^{-1} \bmod p$ 的问题转化为规模更小的问题 $(p \bmod i)^{-1} \bmod p$．因此，从 $1^{-1} \bmod p=1$ 开始，对每个 i 顺次应用该公式，就可以在 $O(n)$ 时间内获得前 n 个整数的逆元.

参考实现如下:

参考实现

- C++

```
// Precomputes modular inverses of all integers from 1 to n modulo
prime p.
std::vector<int> precompute_inverses(int n, int p) {
  std::vector<int> res(n + 1);
  res[1] = 1;
  for (int i = 2; i <= n; ++i) {
    res[i] = (long long)(p - p / i) * res[p % i] % p;
  }
  return res;
}

```

- Python

```python
# Precomputes modular inverses of all integers from 1 to n modulo prime p.
def precompute_inverses(n, p):
    res = [0] * (n + 1)
    res[1] = 1
    for i in range(2, n + 1):
        res[i] = (p - p // i) * res[p % i] % p
    return res


```

这一算法只适用于模数是素数的情形．对于模数 m 不是素数的情形，无法保证递推公式中得到的 $m \bmod i$ 仍然与 m 互素，因而递推所需要的 $(m \bmod i)^{-1}$ 可能并不存在．一个这样的例子是 $m = 8, i = 3$．此时，$m \bmod i = 2$，不存在模 m 的逆元．

另外，得到该递推公式后，一种自然的想法是直接递归求解任意一个数 a 的逆元．每次递归时，都利用递推公式将它转化为更小的余数 $p \bmod a$ 的逆元，直到余数变为 1 时停止．目前尚不清楚这样做的复杂度[23]，因此，推荐使用前文所述的常规方法求解．

习题

- LOJ 110 乘法逆元[2]
- LOJ 161 乘法逆元 2[3]
- LOJ 2605「NOIP2012」同余方程[4]
- Luogu P2054「AHOI2005」洗牌[5]
- LOJ 2034「SDOI2016」排列计数[6]

参考资料与注释

- Modular multiplicative inverse - Wikipedia[7]

[1] riteme 在知乎上的回答

[2] LOJ 110 乘法逆元

[3] LOJ 161 乘法逆元 2

[4] LOJ 2605「NOIP2012」同余方程

[5] Luogu P2054「AHOI2005」洗牌

[6] LOJ 2034「SDOI2016」排列计数

[7] Modular multiplicative inverse - Wikipedia

9.11.11 线性同余方程

本文讨论线性同余方程的求解．

基本概念

设 a, b, n 为整数，x 为未知数，那么，形如

$$ax \equiv b \pmod{n}$$

的方程称为 **线性同余方程**（linear congruence equation）．

求解线性同余方程，需要找到区间 $[0, n-1]$ 中 x 的全部解．当然，将它们加减 n 的任意倍数，依然是方程的解．在模 n 的意义下，这些就是该方程的全部解．

[23] riteme 在知乎上的回答[1] 中指出，这样做理论上已知的复杂度的上界是 $O(p^{1/3+\varepsilon})$，而在实际随机数据中的表现接近于 $O(\log p)$．

本文接下来介绍两种求解线性同余方程的思路，分别利用了逆元和不定方程．对于一般的情形，逆元和不定方程的求解都需要用到 扩展欧几里得算法→9.11.4@p1188，因此，这两种思路其实是一致的．

用逆元求解

首先，考虑 a 和 n 互素的情形，即 $\gcd(a,n)=1$ 的情形．此时，可以计算 a 的 逆元→9.11.10@p1229 a^{-1}，并将方程两边同乘以 a^{-1}，这就得到方程的唯一解：

$$x \equiv ba^{-1} \pmod{n}.$$

紧接着，考虑 a 和 n 不互素的情形，即 $\gcd(a,n)=d>1$ 的情形．此时，原方程不一定有解．例如，$2x \equiv 1 \pmod 4$ 就没有解．因此，需要考虑两种情形：

- 当 d 不能整除 b 时，方程无解．对于任意的 x，方程左侧 ax 都是 d 的倍数，但是方程右侧 b 不是 d 的倍数．因此，它们不可能相差 n 的倍数，因为 n 的倍数也一定是 d 的倍数．所以，方程无解．
- 当 d 可以整除 b 时，可以将方程的参数 a,b,n 都同除以 d，得到一个新的方程：

  $a'x \equiv b' \pmod{n'}$.

  其中，$\gcd(a',n')=1$，也就是说，a' 和 n' 互素．这种情形已经在前文解决，所以，可以通过求解逆元得到方程的最小非负整数解 x'.

  显然，x' 也是原方程的一个解．但这并非原方程唯一的解．由于转化后的方程的全体解为 $\{x'+kn' : k \in \mathbf{Z}\}$.

  这些解中落在区间 $[0,n-1]$ 的那些，就是原方程在区间 $[0,n-1]$ 中的全部解：

  $x \equiv (x'+kn') \pmod{n}, \quad k=0,1,\cdots,d-1.$

总结这两种情形，线性同余方程的 **解的数量** 等于 $d=\gcd(a,n)$ 或 0.

用不定方程求解

线性同余方程等价于关于 x,y 的 二元一次不定方程→9.11.8@p1218：

$$ax+ny=b.$$

利用所引页面的讨论，方程有解当且仅当 $\gcd(a,n) \mid b$，而且该方程的通解是

$$x = x_0 + t\frac{n}{d},$$

$$y = y_0 - t\frac{a}{d},$$

其中，$d=\gcd(a,n)$ 是它们的最大公约数，t 是任意整数．

进而，线性同余方程的通解就是

$$x \equiv \left(x_0 + t\frac{n}{d}\right) \pmod{n}, \quad t \in \mathbf{Z}.$$

将 x_0 对 n/d 取模就得到同余方程的最小（非负）整数解，也就是上文的 x'.

参考实现

本节提供的参考实现可以得到同余方程的最小非负整数解．如果解不存在，则输出 -1.

参考实现

- C++

```cpp
// Extended Euclidean Algorithm.
// Finds integers x, y such that a*x + b*y = gcd(a, b),
// and returns gcd(a, b).
int ex_gcd(int a, int b, int& x, int& y) {
  if (!b) {
```

```
    x = 1;
    y = 0;
    return a;
  } else {
    int d = ex_gcd(b, a % b, y, x);
    y -= a / b * x;
    return d;
  }
}

// Solves the linear congruence equation:
//     a * x ≡ b (mod n), where n > 0.
// Returns the smallest non-negative solution x,
// or -1 if there is no solution.
int solve_linear_congruence_equation(int a, int b, int n) {
  int x, y;
  int d = ex_gcd(a, n, x, y);
  if (b % d) return -1;
  n /= d;
  return ((long long)x * (b / d) % n + n) % n;
}

```

- Python

```
def ex_gcd(a, b):
    """
    Extended Euclidean Algorithm.
    Finds integers x, y such that a*x + b*y = gcd(a, b),
    and returns (gcd, x, y).
    """
    if b == 0:
        return a, 1, 0
    d, x1, y1 = ex_gcd(b, a % b)
    x = y1
    y = x1 - (a // b) * y1
    return d, x, y


def solve_linear_congruence_equation(a, b, n):
    """
    Solves the linear congruence equation:
        a * x ≡ b (mod n), where n > 0.
    Returns the smallest non-negative solution x,
    or -1 if there is no solution.
    """
    d, x, y = ex_gcd(a, n)
    if b % d != 0:
        return -1
    n //= d
    return (x * (b // d) % n + n) % n


```

习题

- 「NOIP2012」同余方程[1]

本页面主要译自博文 Модульное линейное уравнение первого порядка[2] 与其英文翻译版 Linear Congruence Equation[3]．其中俄文版版权协议为 Public Domain + Leave a Link；英文版版权协议为 CC-BY-SA 4.0．内容有改动．

外部参考资料

[1] 「NOIP2012」同余方程 [2] Модульное линейное уравнение первого порядка [3] Linear Congruence Equation

9.11.12 中国剩余定理

引入

> 「物不知数」问题：有物不知其数，三三数之剩二，五五数之剩三，七七数之剩二．问物几何?

即求满足以下条件的整数：除以 3 余 2，除以 5 余 3，除以 7 余 2.

该问题最早见于《孙子算经》中，并有该问题的具体解法．宋朝数学家秦九韶于 1247 年《数书九章》卷一、二《大衍类》对「物不知数」问题做出了完整系统的解答．上面具体问题的解答口诀由明朝数学家程大位在《算法统宗》中给出：

> 三人同行七十希，五树梅花廿一支，七子团圆正半月，除百零五便得知.

$2\times 70+3\times 21+2\times 15=233=2\times 105+23$，故答案为 23.

定义

中国剩余定理 (Chinese Remainder Theorem, CRT) 可求解如下形式的一元线性同余方程组（其中 $n_1,n_2,\cdots,n_k$ 两两互质）：

$$\begin{cases} x\equiv a_1 \pmod{n_1} \\ x\equiv a_2 \pmod{n_2} \\ \vdots \\ x\equiv a_k \pmod{n_k} \end{cases}$$

上面的「物不知数」问题就是一元线性同余方程组的一个实例.

过程

1. 计算所有模数的积 n;
2. 对于第 i 个方程:
    1. 计算 $m_i=\frac{n}{n_i}$;
    2. 计算 m_i 在模 n_i 意义下的 逆元→9.11.10@p1229 m_i^{-1};
    3. 计算 $c_i=m_i m_i^{-1}$（**不要对 n_i 取模**）.
3. 方程组在模 n 意义下的唯一解为：$x=\sum_{i=1}^{k} a_i c_i \pmod{n}$.

实现

- C++

```cpp
LL CRT(int k, LL* a, LL* r) {
  LL n = 1, ans = 0;
  for (int i = 1; i <= k; i++) n = n * r[i];
  for (int i = 1; i <= k; i++) {
    LL m = n / r[i], b, y;
```

```
    exgcd(m, r[i], b, y);  // b * m mod r[i] = 1
    ans = (ans + a[i] * m * b % n) % n;
  }
  return (ans % n + n) % n;
}
```

- Python

```
def CRT(k, a, r):
    n = 1
    ans = 0
    for i in range(1, k + 1):
        n = n * r[i]
    for i in range(1, k + 1):
        m = n // r[i]
        b = y = 0
        exgcd(m, r[i], b, y)  # b * m mod r[i] = 1
        ans = (ans + a[i] * m * b % n) % n
    return (ans % n + n) % n
```

证明

我们需要证明上面算法计算所得的 x 对于任意 $i = 1, 2, \cdots, k$ 满足 $x \equiv a_i \pmod{n_i}$.

当 $i \neq j$ 时，有 $m_j \equiv 0 \pmod{n_i}$，故 $c_j \equiv m_j \equiv 0 \pmod{n_i}$. 又有 $c_i \equiv m_i \cdot (m_i^{-1} \bmod n_i) \equiv 1 \pmod{n_i}$，所以我们有：

$$
\begin{aligned}
x &\equiv \sum_{j=1}^{k} a_j c_j && \pmod{n_i} \\
&\equiv a_i c_i && \pmod{n_i} \\
&\equiv a_i \cdot m_i \cdot (m_i^{-1} \bmod n_i) && \pmod{n_i} \\
&\equiv a_i && \pmod{n_i}
\end{aligned}
$$

即对于任意 $i = 1, 2, \cdots, k$，上面算法得到的 x 总是满足 $x \equiv a_i \pmod{n_i}$，即证明了解同余方程组的算法的正确性.

因为我们没有对输入的 a_i 作特殊限制，所以任何一组输入 $\{a_i\}$ 都对应一个解 x. 另外，若 $x \neq y$，则总存在 i 使得 x 和 y 在模 n_i 下不同余. 故系数列表 $\{a_i\}$ 与解 x 之间是一一映射关系，方程组总是有唯一解.

解释

下面演示 CRT 如何解「物不知数」问题.

1. $n = 3 \times 5 \times 7 = 105$;
2. 三人同行 **七十** 希：$n_1 = 3, m_1 = n/n_1 = 35, m_1^{-1} \equiv 2 \pmod{3}$，故 $c_1 = 35 \times 2 = 70$;
3. 五树梅花 **廿一** 支：$n_2 = 5, m_2 = n/n_2 = 21, m_2^{-1} \equiv 1 \pmod{5}$，故 $c_2 = 21 \times 1 = 21$;
4. 七子团圆正 **半月**：$n_3 = 7, m_3 = n/n_3 = 15, m_3^{-1} \equiv 1 \pmod{7}$，故 $c_3 = 15 \times 1 = 15$;
5. 所以方程组的唯一解为 $x \equiv 2 \times 70 + 3 \times 21 + 2 \times 15 \equiv 233 \equiv 23 \pmod{105}$.（除 **百零五** 便得知）

Garner 算法

CRT 的另一个用途是用一组比较小的质数表示一个大的整数.

例如，若 a 满足如下线性方程组，且 $a < \prod_{i=1}^{k} p_i$（其中 p_i 为质数）：

$$\begin{cases} a \equiv a_1 \pmod{p_1} \\ a \equiv a_2 \pmod{p_2} \\ \quad \vdots \\ a \equiv a_k \pmod{p_k} \end{cases}$$

我们可以用以下形式的式子（称作 a 的混合基数表示）表示 a：

$$a = x_1 + x_2 p_1 + x_3 p_1 p_2 + \ldots + x_k p_1 \ldots p_{k-1}$$

Garner 算法 将用来计算系数 $x_1, \ldots, x_k$.

令 r_{ij} 为 p_i 在模 p_j 意义下的 逆→9.11.10@p1229：

$$p_i \cdot r_{i,j} \equiv 1 \pmod{p_j}$$

把 a 代入我们得到的第一个方程：

$$a_1 \equiv x_1 \pmod{p_1}$$

代入第二个方程得出：

$$a_2 \equiv x_1 + x_2 p_1 \pmod{p_2}$$

方程两边减 x_1，除 p_1 后得

$$\begin{aligned} a_2 - x_1 &\equiv x_2 p_1 && \pmod{p_2} \\ (a_2 - x_1) r_{1,2} &\equiv x_2 && \pmod{p_2} \\ x_2 &\equiv (a_2 - x_1) r_{1,2} && \pmod{p_2} \end{aligned}$$

类似地，我们可以得到：

$$x_k = (\ldots((a_k - x_1) r_{1,k} - x_2) r_{2,k}) - \ldots) r_{k-1,k} \bmod p_k$$

实现

- C++

```cpp
for (int i = 0; i < k; ++i) {
  x[i] = a[i];
  for (int j = 0; j < i; ++j) {
    x[i] = r[j][i] * (x[i] - x[j]);
    x[i] = x[i] % p[i];
    if (x[i] < 0) x[i] += p[i];
  }
}
```

- Python

```python
for i in range(0, k):
    x[i] = a[i]
    for j in range(0, i):
        x[i] = r[j][i] * (x[i] - x[j])
        x[i] = x[i] % p[i]
        if x[i] < 0:
            x[i] = x[i] + p[i]
```

该算法的时间复杂度为 $O(k^2)$. 实际上 Garner 算法并不要求模数为质数，只要求模数两两互质，我们有如下伪代码：

Chinese Remainder Algorithm $\operatorname{cra}(\mathbf{v},\mathbf{m})$:
Input:$\mathbf{m}=(m_0,m_1,...,m_{n-1}),m_i\in\mathbb{Z}^+\wedge\gcd(m_i,m_j)=1$ for all $i\neq j$,
$\mathbf{v}=(v_0,...,v_{n-1})$ where $v_i=x \bmod m_i$.
Output:$x \bmod \prod_{i=0}^{n-1} m_i$.

```
1  for i from 1 to (n-1) do
2      C_i ← (∏_{j=0}^{i-1} m_j)^{-1} mod m_i
3  x ← v_0
4  for i from 1 to (n-1) do
5      u ← (v_i - x) · C_i mod m_i
6      x ← x + u ∏_{j=0}^{i-1} m_j
7  return (x)
```

可以发现在第六行中的计算过程对应上述混合基数的表示.

应用

某些计数问题或数论问题出于加长代码、增加难度、或者是一些其他原因，给出的模数：**不是质数**！

但是对其质因数分解会发现它没有平方因子，也就是该模数是由一些不重复的质数相乘得到.

那么我们可以分别对这些模数进行计算，最后用 CRT 合并答案.

下面这道题就是一个不错的例子.

洛谷 P2480 [SDOI2010] 古代猪文[1]

给出 G,n（$1\le G,n\le 10^9$），求：

$$G^{\sum_{k|n}\binom{n}{k}} \bmod 999\ 911\ 659$$

首先，当 $G=999\ 911\ 659$ 时，所求显然为 0.

否则，根据 欧拉定理→9.11.9@p1223，可知所求为：

$$G^{\sum_{k|n}\binom{n}{k} \bmod 999\ 911\ 658} \bmod 999\ 911\ 659$$

现在考虑如何计算：

$$\sum_{k|n}\binom{n}{k} \bmod 999\ 911\ 658$$

因为 999 911 658 不是质数，无法保证 $\forall x\in[1,999\ 911\ 657]$，$x$ 都有逆元存在，上面这个式子我们无法直接计算.

注意到 $999\ 911\ 658=2\times 3\times 4679\times 35617$，其中每个质因子的最高次数均为一，我们可以考虑分别求出 $\sum_{k|n}\binom{n}{k}$ 在模 2，3，4679，35617 这几个质数下的结果，最后用中国剩余定理来合并答案.

也就是说，我们实际上要求下面一个线性方程组的解：

$$\begin{cases} x\equiv a_1 \pmod 2\\ x\equiv a_2 \pmod 3\\ x\equiv a_3 \pmod{4679}\\ x\equiv a_4 \pmod{35617}\end{cases}$$

而计算一个组合数对较小的质数取模后的结果，可以利用 卢卡斯定理→9.11.15@p1251.

扩展：模数不互质的情况

两个方程

设两个方程分别是 $x\equiv a_1 \pmod{m_1}$、$x\equiv a_2 \pmod{m_2}$；

将它们转化为不定方程：$x=m_1p+a_1=m_2q+a_2$，其中 p,q 是整数，则有 $m_1p-m_2q=a_2-a_1$.

由 裴蜀定理→9.11.8@p1218，当 a_2-a_1 不能被 $\gcd(m_1,m_2)$ 整除时，无解；

其他情况下，可以通过 扩展欧几里得算法→9.11.4@p1188 解出来一组可行解 (p,q)；

则原来的两方程组成的模方程组的解为 $x\equiv b \pmod M$，其中 $b=m_1p+a_1$，$M=\operatorname{lcm}(m_1,m_2)$.

多个方程

用上面的方法两两合并即可.

习题

- 【模板】中国剩余定理（CRT）/曹冲养猪[2]
- 【模板】扩展中国剩余定理[3]
- 「NOI2018」屠龙勇士[4]
- 「TJOI2009」猜数字[5]

本页面部分内容译自博文 Китайская теорема об остатках[6] 与其英文翻译版 Chinese Remainder Theorem[7]. 其中俄文版版权协议为 Public Domain + Leave a Link；英文版版权协议为 CC-BY-SA 4.0.

外部参考资料

[1] 洛谷 P2480 [SDOI2010] 古代猪文

[2] 【模板】中国剩余定理（CRT）/曹冲养猪

[3]【模板】扩展中国剩余定理

[4] 「NOI2018」屠龙勇士

[5] 「TJOI2009」猜数字

[6] Китайская теорема об остатках

[7] Chinese Remainder Theorem

9.11.13 升幂引理

内容

升幂（Lift the Exponent，LTE）引理是初等数论中比较常用的一个定理.

定义 $\nu_p(n)$ 为整数 n 的标准分解中素因子 p 的幂次，即 $\nu_p(n)$ 满足 $p^{\nu_p(n)}\mid n$ 且 $p^{\nu_p(n)+1}\nmid n$.

由于升幂引理内容较长，我们将其分为三部分介绍：

以下内容设 p 为素数，x,y 为满足 $p\nmid x$ 且 $p\nmid y$ 的整数，n 为正整数.

第一部分

对所有的素数 p 和满足 $(n,p)=1$ 的整数 n,

1. 若 $p\mid x-y$，则：
   $\nu_p(x^n-y^n)=\nu_p(x-y)$
2. 若 $p\mid x+y$，则对奇数 n 有：
   $\nu_p(x^n+y^n)=\nu_p(x+y)$

证明

若 $p\mid x-y$，则不难发现 $p\mid x-y\iff x\equiv y \pmod p$，则显然有：

$$\sum_{i=0}^{n-1}x^iy^{n-1-i}\equiv nx^{n-1}\not\equiv 0 \pmod p$$

进而由 $x^n-y^n=(x-y)\sum_{i=0}^{n-1}x^iy^{n-1-i}$ 可知命题得证.

对 $p\mid x+y$ 的情况证明方法类似.

第二部分

若 p 是奇素数，

1. 若 $p \mid x-y$，则：
   $\nu_p(x^n-y^n)=\nu_p(x-y)+\nu_p(n)$
2. 若 $p \mid x+y$，则对奇数 n 有：
   $\nu_p(x^n+y^n)=\nu_p(x+y)+\nu_p(n)$

证明

若 $p \mid x-y$，令 $y=x+kp$，我们只需证明 $p \mid n$ 的情况.

- 若 $n=p$，则由二项式定理：
$$\begin{aligned}\sum_{i=0}^{p-1} x^{p-1-i}y^i &= \sum_{i=0}^{p-1} x^{p-1-i}\sum_{j=0}^{i}\binom{i}{j}x^j(kp)^{i-j}\\ &\equiv px^{p-1} \pmod{p^2}\end{aligned}$$
  从而
  $\nu_p(x^n-y^n)=\nu_p(x-y)+1$
- 若 $n=p^a$，则由数学归纳法可得
  $\nu_p(x^n-y^n)=\nu_p(x-y)+a$

因此命题得证.

对 $p \mid x+y$ 的情况证明方法类似.

第三部分

若 $p=2$ 且 $p \mid x-y$，

1. 对奇数 n 有（与第一部分的 1 相同）：
   $\nu_p(x^n-y^n)=\nu_p(x-y)$
2. 对偶数 n 有：
   $\nu_p(x^n-y^n)=\nu_p(x-y)+\nu_p(x+y)+\nu_p(n)-1$

另外对上述的 x,y,n，我们有：

若 $4 \mid x-y$，则：

- $\nu_2(x+y)=1$
- $\nu_2(x^n-y^n)=\nu_2(x-y)+\nu_2(n)$

证明

我们只需证明 n 为偶数的情况. 由于此时 $p \nmid \binom{p}{2}$，故我们不能用第二部分的方法证明.

令 $n=2^a b$，其中 $a=\nu_p(n)$，$2 \nmid b$，从而
$$\begin{aligned}\nu_p(x^n-y^n) &= \nu_p(x^{2^a}-y^{2^a})\\ &= \nu_p\left((x-y)(x+y)\prod_{i=1}^{a-1}\left(x^{2^i}+y^{2^i}\right)\right)\end{aligned}$$
注意到 $2 \mid x-y \Longrightarrow 4 \mid x^2-y^2$，从而 $(\forall i \geq 1)$，$x^{2^i}+y^{2^i} \equiv 2 \pmod 4$，进而上式可变为：
$$\nu_p(x^n-y^n)=\nu_p(x-y)+\nu_p(x+y)+\nu_p(n)-1$$
因此命题得证.

参考资料

1. Lifting-the-exponent lemma - Wikipedia[1]

外部参考资料

[1] Lifting-the-exponent lemma - Wikipedia 

9.11.14 阶乘取模

> **Authors:** aofall, c-forrest, CoelacanthusHex, Early0v0, Enter-tainer, Great-designer, iamtwz, Marcythm, Persdre, shuzhouliu, Tiphereth-A, wsyhb, Xeonacid

引入

本文讨论了某一模数下阶乘计算的相关结论，并提供一种时间复杂度线性相关于模数大小的计算方法，因而该方法主要适用于模数不太大（$\sim 10^6$）的情形．除了本文介绍的方法外，根据场景不同，还可以应用 多项式技术→9.12.11@p1575 进行快速计算．

根据 中国剩余定理→9.11.12@p1236，阶乘取模问题可以转化为模数为素数幂 p^α 的情形．在处理这类问题时，常常需要对于素数 p 和正整数 n，将阶乘 $n!$ 中的所有因子 p 都提取出来，进而得到分解：

$$n! = p^{\nu_p(n!)}(n!)_p.$$

其中，$\nu_p(n!)$ 表示阶乘 $n!$ 的素因数分解中 p 的幂次，$(n!)_p$ 表示在阶乘 $n!$ 的结果中去除所有 p 的幂次得到的整数．本文将讨论 $(n!)_p$ 在素数（幂）模下的余数以及幂次 $\nu_p(n!)$ 的具体计算方法．

这种分解在解决阶乘同时出现在所求表达式的分子和分母的问题时尤为有用，比如 计算某一模数下的二项式系数→9.11.15@p1251. 对于这类问题，分子和分母中 p 的幂次可以直接相减，而与 p 互素的部分 $(n!)_p$ 则可以利用 乘法逆元→9.11.10@p1229 计算．

本文还介绍了与上述问题相关的 Wilson 定理及其推广、Legendre 公式和 Kummer 定理等内容．

Wilson 定理

Wilson 定理给出了判断某个自然数是素数的一个充分必要条件．

Wilson 定理

对于自然数 $n > 1$，当且仅当 n 是素数时，$(n-1)! \equiv -1 \pmod n$.

证明

首先，证明对于素数 p 有 $(p-1)! \equiv -1 \pmod p$. 对于这一点，可以利用 同余方程→9.11.16@p1258 或 原根→9.11.18@p1275 得到两种简洁的证明，此处略去不表．下面提供前置知识较少的一种证明方法：

当 $p = 2$ 时，命题显然成立．下面设 $p \geq 3$，继而要证明 $\mathbf{Z}_p$ 中所有非零元素（即同余类）的积为 $\overline{-1}$. 因为 $\mathbf{Z}_p$ 中所有非零元素 $\bar{a}$ 都有逆元 $\bar{a}^{-1}$，于是 $\mathbf{Z}_p$ 中彼此互逆的元素乘积为 $\bar{1}$. 但是要注意 $\bar{a}$ 和 $\bar{a}^{-1}$ 可能相等：$\bar{a} = \bar{a}^{-1}$，当且仅当 $a^2 \equiv 1 \pmod p$，即

$$0 \equiv a^2 - 1 \equiv (a+1)(a-1) \pmod p,$$

从而，$a \equiv 1 \pmod p$ 或 $a \equiv -1 \pmod p$. 这说明 $\mathbf{Z}_p \setminus \{\bar{0}, \bar{1}, \overline{-1}\}$ 中所有元素的乘积为 $\bar{1}$，进而 $\mathbf{Z}_p$ 中所有非零元素的积为 $\overline{-1}$.

反过来，对于合数 n 的情形，要证明 $(n-1)! \not\equiv -1 \pmod n$. 利用反证法，不妨设 $(n-1)! \equiv -1 \pmod n$，亦即存在整数 k 使得 $(n-1)! = kn - 1$ 成立．因为 n 是合数，必然存在素数 $p < n$ 使

得 $n=pm$，所以 $(n-1)!=kpm-1\equiv -1 \pmod p$．但是，乘积 $(n-1)!$ 中必然已经出现 p，故而一定有 $(n-1)!\equiv 0 \pmod p$．这一矛盾就说明了 $(n-1)!\not\equiv -1 \pmod n$．

利用本文的记号，Wilson 定理可以写作 $(p!)_p\equiv -1 \pmod p$．

推广

Wilson 定理可以推广到一般模数的情形．

定理（Gauss）

对于自然数 $m>1$，有

$$\prod_{1\le k<m,\ k\perp m} k\equiv \pm 1 \pmod m.$$

而且，余数中的 ± 1 取值为 -1 当且仅当模 m 的 原根存在→9.11.18@p1275，即 $m=2,4,p^\alpha,2p^\alpha$ 时，其中 p 是奇素数且 α 是正整数．

证明

这个定理可以通过 模 m 整数乘法群→9.16.3@p1830 的结构简单地证明．此处给出思路相仿，但是较为初等的证明．

对于 $m=2$ 的情形，有 $1!=1\equiv -1 \pmod 2$．对于其他存在原根的情形，设原根为 g，则所有满足小于 m 且与它互素的正整数 k 都可以唯一地表示为 $g^i \bmod m$ 的形式，其中 $0\le i<\varphi(m)$ 且 $\varphi(m)$ 是 Euler 函数→9.11.5@p1196．直接验证可知，$\varphi(m)$ 一定是偶数．因为 g^i 和 $g^{\varphi(m)-i}$ 互为乘法逆元，所以在乘积中将它们两两配对，就有

$$\prod_{1\le k<m,\ k\perp m} k\equiv \prod_{i=0}^{\varphi(m)-1} g^i=g^{\varphi(m)/2}\prod_{i=1}^{\varphi(m)/2-1} g^i g^{\varphi(m)-i}\equiv g^{\varphi(m)/2} \pmod m.$$

因为 $g^{\varphi(m)/2}\bmod m$ 是唯一的不等于 $1\bmod m$ 且乘法逆元就是它自身的元素，所以它就等于 $-1\bmod m$．这就说明了此时的余数等于 -1．

对于模 m 的原根不存在的情形，要证明余数等于 1．为此，可以首先做素因数分解 $m=p_1^{e_1}p_2^{e_2}\cdots p_s^{e_s}$，然后应用 中国剩余定理→9.11.12@p1236 可知，只需要证明

$$\prod_{1\le k<m,\ k\perp m} k\equiv 1 \pmod{p_j^{e_j}}$$

对所有因子 $p_j^{e_j}$ 都成立．中国剩余定理说明，每一个可能的余数组合 $(r_1,r_2,\cdots,r_s)$，其中，$1\le r_j<p_j^{e_j}$ 且 $p_j\perp r_j$，都唯一地对应着一个 $1\le k<m$ 且 $k\perp m$ 使得 $k\equiv r_j \pmod{p_j^{e_j}}$ 成立．所以，对于每个余数 r_j，都恰好有 $\varphi(m)/\varphi(p_j^{e_j})$ 个 k 使得 $k\equiv r_j \pmod{p_j^{e_j}}$ 成立．利用这一点，可以对乘积进行分组，就有

$$\prod_{1\le k<m,\ k\perp m} k\equiv \left(\prod_{1\le r_j<p_j^{e_j},\ r_j\perp p_j} r_j\right)^{\varphi(m)/\varphi(p_j^{e_j})} \pmod{p_j^{e_j}}.$$

此处的指数 $\varphi(m)/\varphi(p_j^{e_j})=\varphi(m/p_j^{e_j})$ 要成为奇数，必然要求 $m/p_j^{e_j}=1,2$，因为 Euler 函数 $\varphi(n)$ 对于 $n\ge 3$ 都是偶数．如果 p_j 是奇素数，因为模 m 的原根不存在，必然有 $m/p_j^{e_j}\ne 1,2$；如果 $p_j^{e_j}=2,4$，因为模 m 的原根不存在，必然有 $m/p_j^{e_j}$ 含有某个奇素因子，故而大于 2：这两种情形指数 $\varphi(m)/\varphi(p_j^{e_j})$ 都是偶数．而上式中括号里的项已经证明是模 $p_j^{e_j}$ 余 -1 的，所以这个幂模 $p_j^{e_j}$ 的余数一定是 1．剩余的情形只有 $p_j=2$ 且 $e_j>2$ 时，对于这个情形，可以直接证明：

$$\prod_{1\le r_j<2^{e_j},\, r_j\perp 2} r_j \equiv 1 \pmod{2^{e_j}}.$$

仿照前文的证明思路，可以将所有 $1 \le r_j < 2^{e_j}$ 的奇数 r_j 两两配对而消去，那些无法配对的必然是方程 $x^2 \equiv 1 \pmod{2^{e_j}}$ 的解．该方程意味着 $2^{e_j} \mid (x-1)(x+1)$．令 $x = 2y+1$，就必然有 $2^{e_j-2} \mid y(y+1)$，而 y 和 $y+1$ 必然一奇一偶，所以 $y = t2^{e_j-2}$ 或 $y = t2^{e_j-2}-1$．故而，有 $x = t2^{e_j-1} \pm 1$ 且 t 是整数．模 2^{e_j} 的余数中，只有 ± 1 和 $2^{e_j-1} \pm 1$ 四个．因此，有

$$\prod_{1\le r_j<2^{e_j},\, r_j\perp 2} r_j \equiv (-1)(2^{e_j-1}-1)(2^{e_j-1}+1) \equiv 1 \pmod{2^{e_j}}.$$

这就完成了所有情形的证明．

在计算中，尤为重要的是模数为素数幂的情形：

推论

对于素数 p 和正整数 α，有

$$\prod_{1\le k<p^{\alpha},\, k\perp p} k \equiv \begin{cases} 1, & p=2 \text{ and } \alpha \ge 3, \\ -1, & \text{otherwise} \end{cases} \pmod{p^{\alpha}}.$$

注意，左侧并非 $(p^{(\alpha)!})_p$，因为后者还需要统计 p 的倍数的贡献．

阶乘余数的计算

本节讨论余数 $(n!)_p \bmod p^{\alpha}$ 的计算．

素数模的情形

算式 $(n!)_p$ 有明显的递归结构．为注意到这一点，首先考察一个具体的例子：

例子

要计算 $(32!)_5 \bmod 5$，可以做如下递归计算：

$$\begin{aligned}
(32!)_5 &= 1\times 2\times 3\times 4\times \underbrace{1}_{5}\times 6\times 7\times 8\times 9\times \underbrace{2}_{10} \\
&\quad \times 11\times 12\times 13\times 14\times \underbrace{3}_{15}\times 16\times 17\times 18\times 19\times \underbrace{4}_{20} \\
&\quad \times 21\times 22\times 23\times 24\times \underbrace{1}_{25}\times 26\times 27\times 28\times 29\times \underbrace{6}_{30}\times 31\times 32 \\
&\equiv 1\times 2\times 3\times 4\times \underbrace{1}_{5}\times 1\times 2\times 3\times 4\times \underbrace{2}_{10} \\
&\quad \times 1\times 2\times 3\times 4\times \underbrace{3}_{15}\times 1\times 2\times 3\times 4\times \underbrace{4}_{20} \\
&\quad \times 1\times 2\times 3\times 4\times \underbrace{1}_{25}\times 1\times 2\times 3\times 4\times \underbrace{1}_{30}\times 1\times 2 \\
&= (1\times 2\times 3\times 4)^6\times (1\times 2)\times (\underbrace{1}_{5}\times \underbrace{2}_{10}\times \underbrace{3}_{15}\times \underbrace{4}_{20}\times \underbrace{1}_{25}\times \underbrace{1}_{30}) \pmod 5
\end{aligned}$$

可以看出，利用模 5 余数的周期性，可以将这一乘积划分为若干个长度为 5 的块，每一块的唯一差异就是最后一个元素的余数．因为 32 除以 5 得到的商是 6 且余数是 2，所以，该乘积可以划分为 6 个完整的块和最后一段长度为 2 的不完整的块．因此，可以将前 6 个块除了最后一个元素之外的部分提取出来（这一部分恰好是 Wilson 定理能够解决的），再乘上最后一个不完整的块的乘积，最后乘上前 6 个块的最后一个元素的连乘积．每个块的最后一个元素都是 5 的倍数，去掉 5 的幂次后，它们的连乘积恰好是 $(6!)_5 \pmod 5$．这就将原来的问题转化为了规模更小的问题．

将该例子中的递归的结构一般化，就得到如下递推公式：

递推公式

对于素数 p 和正整数 n，有

$$(n!)_p \equiv (-1)^{\lfloor n/p \rfloor} \cdot (n \bmod p)! \cdot (\lfloor n/p \rfloor !)_p \pmod p.$$

证明

记 $(n)_p$ 为 n 的素因数分解中去除所有 p 的幂次的结果．于是，有

$$\begin{aligned}
(n!)_p &= \prod_{k=1}^{n} (k)_p = \left(\prod_{1 \le k \le n,\ k \perp p} (k)_p \right) \left(\prod_{1 \le k \le \lfloor n/p \rfloor} (pk)_p \right) \\
&= \left(\prod_{i=0}^{\lfloor n/p \rfloor - 1} \prod_{j=1}^{p-1} (ip+j) \right) \left(\prod_{j=1}^{n \bmod p} (\lfloor n/p \rfloor p + j) \right) \left(\prod_{1 \le k \le \lfloor n/p \rfloor} (k)_p \right) \\
&\equiv \left(\prod_{j=1}^{p-1} j \right)^{\lfloor n/p \rfloor} \left(\prod_{j=1}^{n \bmod p} j \right) (\lfloor n/p \rfloor !)_p \\
&\equiv (-1)^{\lfloor n/p \rfloor} \cdot (n \bmod p)! \cdot (\lfloor n/p \rfloor !)_p \pmod p.
\end{aligned}$$

这就完成了证明．下面对于该形式证明提供具体的解释．

要计算 $(n!)_p \bmod p$ 的值．仿照上面的例子，有

$$\begin{aligned}
(n!)_p &= 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (p-2) \cdot (p-1) \cdot \underbrace{1}_{p} \cdot (p+1) \cdot (p+2) \cdot \ldots \cdot (2p-1) \cdot \underbrace{2}_{2p} \\
&\quad \cdot (2p+1) \cdot \ldots \cdot (p^2-1) \cdot \underbrace{1}_{p^2} \cdot (p^2+1) \cdot \ldots \cdot n \\
&\equiv 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (p-2) \cdot (p-1) \cdot \underbrace{1}_{p} \cdot 1 \cdot 2 \cdot \ldots \cdot (p-1) \cdot \underbrace{2}_{2p} \cdot 1 \cdot 2 \\
&\quad \cdot \ldots \cdot (p-1) \cdot \underbrace{1}_{p^2} \cdot 1 \cdot 2 \cdot \ldots \cdot (n \bmod p) \pmod p.
\end{aligned}$$

可以清楚地看到，除了最后一个块外，阶乘被划分为几个长度相同的完整的块．

$$\begin{aligned}
(n!)_p \equiv{}& \underbrace{1 \cdot 2 \cdot 3 \cdot \ldots \cdot (p-2) \cdot (p-1) \cdot 1}_{1\text{ st}} \cdot \underbrace{1 \cdot 2 \cdot 3 \cdot \ldots \cdot (p-2) \cdot (p-1) \cdot 2}_{2\text{ nd}} \cdot \ldots \\
&\cdot \underbrace{1 \cdot 2 \cdot 3 \cdot \ldots \cdot (p-2) \cdot (p-1) \cdot 1}_{p\text{ th}} \cdot \ldots \cdot \underbrace{1 \cdot 2 \cdot \ldots \cdot (n \bmod p)}_{\text{tail}} \pmod p.
\end{aligned}$$

除了块的最后一个元素外，完整的块的主要部分 $(p-1)! \bmod p$ 很容易计算，可以应用 Wilson 定理：

$$(p-1)! \equiv -1 \pmod p.$$

总共有 $\left\lfloor \frac{n}{p} \right\rfloor$ 个完整的块，因此需要将 $\left\lfloor \frac{n}{p} \right\rfloor$ 写到 -1 的指数上．

最后一个部分块的值就是 $(n \bmod p)! \bmod p$，可以单独计算．

剩下的就是每个块的最后一个元素．如果隐藏已处理的元素，可以看到以下模式：

$$(n!)_p = \underbrace{\ldots \cdot 1}_{} \cdot \underbrace{\ldots \cdot 2}_{} \cdot \ldots \cdot \underbrace{\ldots \cdot (p-1)}_{} \cdot \underbrace{\ldots \cdot 1}_{} \cdot \underbrace{\ldots \cdot 1}_{} \cdot \underbrace{\ldots \cdot 2}_{} \cdots$$

这也是一个修正的阶乘，只是长度短得多．它是：

$$\left(\left\lfloor \frac{n}{p} \right\rfloor ! \right)_p.$$

将各部分乘起来，就得到上面的递推公式．

利用该递推式做计算，递归深度为 $O(\log_p n)$. 如果每次都重新计算中间那一项，那么每层计算的复杂度都是 $O(p)$ 的，总的时间复杂度是 $O(p \log_p n)$；如果对所有 $n = 1, 2, \cdots, p-1$ 都预先处理了 $n! \bmod p$，那么预处理的复杂度是 $O(p)$ 的，每层计算的复杂度都是 $O(1)$ 的，总的时间复杂度是 $O(p + \log_p n)$ 的.

在实现时，因为是尾递归，可以用迭代实现. 下面的实现对前 $p-1$ 个阶乘做了预计算，如果需要多次调用，可以将预计算放到函数外进行.

参考实现

```cpp
// Calculate (n!)_p mod p.
int factmod(int n, int p) {
  // Pretreatment.
  std::vector<int> f(p);
  f[0] = 1;
  for (int i = 1; i < p; ++i) {
    f[i] = (long long)f[i - 1] * i % p;
  }
  // Recursion.
  int res = 1;
  while (n > 1) {
    if ((n / p) & 1) res = p - res;
    res = (long long)res * f[n % p] % p;
    n /= p;
  }
  return res;
}

```

如果空间有限，无法存储所有阶乘，也可以将函数调用中实际用到的阶乘 $n! \bmod p$ 中的 n 都计算出来，然后对它们进行排序，从而可以在最后一次性计算出来这些阶乘的值，汇总到最终结果中，而避免存储所有阶乘的值.

素数幂模的情形

对于素数幂模的情形，可以仿照素数模的情形解决，只需要将 Wilson 定理替换成它的推广形式. 本节两个结论中的 ± 1，均特指这样的定义：当 $p = 2$ 且 $\alpha \geq 3$ 时取 1，其余情形取 -1.

递推公式

对于素数 p 和正整数 α, n，有

$$(n!)_p \equiv (\pm 1)^{\lfloor n/p^\alpha \rfloor} \cdot \left(\prod_{1 \leq j \leq (n \bmod p^\alpha),\, j \perp p} j \right) \cdot (\lfloor n/p \rfloor !)_p \pmod{p^\alpha}.$$

其中，± 1 的取值如同 Wilson 定理的推广→9.11@p1160 中规定的那样.

证明

证明思路和素数模的情形完全一致. 记 $(k)_p$ 为去除 k 的素因数分解中 p 的全部幂次的结果，则

$$
\begin{aligned}
(n!)_p &= \prod_{1\le k\le n} (k)_p = \left(\prod_{1\le k\le n,\ k\perp p} (k)_p\right)\left(\prod_{1\le k\le \lfloor n/p\rfloor} (pk)_p\right) \\
&= \left(\prod_{i=0}^{\lfloor n/p^\alpha\rfloor-1} \prod_{1\le j\le p^\alpha,\ j\perp p} (ip^\alpha+j)_p\right)\left(\prod_{1\le j\le (n \bmod p^\alpha),\ j\perp p} (\lfloor n/p^\alpha\rfloor p^\alpha+j)_p\right)\left(\prod_{1\le k\le \lfloor n/p\rfloor} (k)_p\right) \\
&\equiv \left(\prod_{1\le j\le p^\alpha,\ j\perp p} j\right)^{\lfloor n/p^\alpha\rfloor} \cdot \left(\prod_{1\le j\le (n \bmod p^\alpha),\ j\perp p} j\right)\cdot (\lfloor n/p\rfloor!)_p \\
&\equiv (\pm 1)^{\lfloor n/p^\alpha\rfloor} \cdot \left(\prod_{1\le j\le (n \bmod p^\alpha),\ j\perp p} j\right)\cdot (\lfloor n/p\rfloor!)_p \pmod{p^\alpha}.
\end{aligned}
$$

与素数模的情形不同之处，除了 -1 可能需要替换为 ± 1 之外，还需要注意预处理的数据的不同．对于素数幂模的情形，需要对所有不超过 p^α 的正整数 n 预处理自 1 至 n 但并非 p 的倍数的所有整数的乘积，即

$$\prod_{1\le k\le n,\ k\perp p} k \bmod p^\alpha.$$

在素数模的情形，当 $n<p$ 时，它退化为 $n! \bmod p$，但是该表达式在一般的素数幂的情形不再适用．

下面提供了在素数幂模的情形下计算阶乘余数的例子，以便理解上述方法：

例子

要计算 $(32!)_3 \bmod 9$，可以做如下递归计算：

$$
\begin{aligned}
(32!)_3 &= 1\times 2\times \underbrace{1}_{\overset{\smile}{3}}\times 4\times 5\times \underbrace{2}_{\overset{\smile}{6}}\times 7\times 8\times \underbrace{1}_{\overset{\smile}{9}} \\
&\quad \times 10\times 11\times \underbrace{4}_{\overset{\smile}{12}}\times 13\times 14\times \underbrace{5}_{\overset{\smile}{15}}\times 16\times 17\times \underbrace{2}_{\overset{\smile}{18}} \\
&\quad \times 19\times 20\times \underbrace{7}_{\overset{\smile}{21}}\times 22\times 23\times \underbrace{8}_{\overset{\smile}{24}}\times 25\times 26\times \underbrace{1}_{\overset{\smile}{27}} \\
&\quad \times 28\times 29\times \underbrace{10}_{30}\times 31\times 32 \\
&\equiv 1\times 2\times \underbrace{1}_{\overset{\smile}{3}}\times 4\times 5\times \underbrace{2}_{\overset{\smile}{6}}\times 7\times 8\times \underbrace{1}_{\overset{\smile}{9}} \\
&\quad \times 1\times 2\times \underbrace{4}_{\overset{\smile}{12}}\times 4\times 5\times \underbrace{5}_{\overset{\smile}{15}}\times 7\times 8\times \underbrace{2}_{\overset{\smile}{18}} \\
&\quad \times 1\times 2\times \underbrace{7}_{\overset{\smile}{21}}\times 4\times 5\times \underbrace{8}_{\overset{\smile}{24}}\times 7\times 8\times \underbrace{1}_{\overset{\smile}{27}} \\
&\quad \times 1\times 2\times \underbrace{1}_{\overset{\smile}{30}}\times 4\times 5 \\
&= (1\times 2\times 4\times 5\times 7\times 8)^3\times (1\times 2\times 4\times 5) \\
&\quad \times \left(\begin{array}{c} \underbrace{1}_{\overset{\smile}{3}}\times \underbrace{2}_{\overset{\smile}{6}}\times \underbrace{1}_{\overset{\smile}{9}}\times \underbrace{4}_{\overset{\smile}{12}}\times \underbrace{5}_{\overset{\smile}{15}}\times \underbrace{2}_{\overset{\smile}{18}} \\ \times \underbrace{7}_{\overset{\smile}{21}}\times \underbrace{8}_{\overset{\smile}{24}}\times \underbrace{1}_{\overset{\smile}{27}}\times \underbrace{1}_{\overset{\smile}{30}} \end{array}\right) \pmod 9.
\end{aligned}
$$

将 $(32!)_3 \bmod 9$ 的算式分解的结果同样可以分为三部分：

- 完整的块：由 $1\sim 9$ 之间所有不被 3 整除的整数的乘积，共 $\lfloor 32/9\rfloor = 3$ 块；
- 尾部不完整的块：所有不被 3 整除的整数从 1 一直乘到 $32 \bmod 9$；
- 所有被 3 整除的整数的乘积，对比倒数第二个等号的结果可知，这就是它的前 $\lfloor 32/3\rfloor = 10$ 项，亦即 $(\lfloor 32/3\rfloor!)_3 \bmod 9$.

最后一个括号里的部分递归求解即可，这样就将原问题转化为了更小的问题.

由此，就可以得到如下递推结果：

递推结果

对于素数 p 和正整数 α, n，有

$$(n!)_p \equiv (\pm 1)^{\sum_{j\geq\alpha}\lfloor n/p^j\rfloor} \prod_{j\geq 0} F(\lfloor n/p^j\rfloor \bmod p^\alpha) \pmod{p^\alpha},$$

其中，$F(m) = \prod_{1\leq k\leq m,\ k\perp p} k \bmod p^\alpha$ 且 ± 1 的取值与上文所述相同.

素数幂模的情形的实现和素数模的情形类似，只有一些细节上的区别. 与上文类似，同样可以将预处理放到函数外进行.

参考实现

```cpp
// Calculate (n!)_p mod pa.
int factmod(int n, int p, int pa) {
  // Pretreatment.
  std::vector<int> f(pa);
  f[0] = 1;
  for (int i = 1; i < pa; ++i) {
    f[i] = i % p ? (long long)f[i - 1] * i % pa : f[i - 1];
  }
  // Recursion.
  bool neg = p != 2 || pa <= 4;
  int res = 1;
  while (n > 1) {
    if ((n / pa) & neg) res = pa - res;
    res = (long long)res * f[n % pa] % pa;
    n /= p;
  }
  return res;
}

```

预处理的时间复杂度为 $O(p^\alpha)$，单次询问的时间复杂度为 $O(\log_p n)$.

幂次的计算

本节讨论阶乘 $n!$ 中 p 的幂次 $\nu_p(n!)$ 的计算，它可以用于计算二项式系数的余数. 因为二项式系数中，分子和分母都会出现阶乘，而分子和分母中素数 p 能否互相抵消，就成为了决定最后的余数的重要因素.

Legendre 公式

阶乘 $n!$ 中素数 p 的幂次可以通过 Legendre 公式计算，而且与 n 在 p 进制下的表示有关.

Legendre 公式

对于正整数 n，阶乘 $n!$ 中含有的素数 p 的幂次 $\nu_p(n!)$ 为

$$\nu_p(n!) = \sum_{i=1}^{\infty} \left\lfloor \frac{n}{p^i} \right\rfloor = \frac{n - S_p(n)}{p-1},$$

其中，$S_p(n)$ 为 p 进制下 n 的各个数位的和. 特别地，阶乘中 2 的幂次是 $\nu_2(n!) = n - S_2(n)$.

证明

因为

$$n! = 1 \times 2 \times \cdots \times p \times \cdots \times 2p \times \cdots \times \lfloor n/p \rfloor p \times \cdots \times n.$$

其中，p 的倍数的乘积为 $p \times 2p \times \cdots \times \lfloor n/p \rfloor p = p^{\lfloor n/p \rfloor} \lfloor n/p \rfloor!$，而 $\lfloor n/p \rfloor!$ 可能继续出现 p 的倍数. 所以，对于幂次，有递推关系：

$$\nu_p(n!) = \lfloor n/p \rfloor + \nu_p(\lfloor n/p \rfloor!).$$

将它展开就得到 Legendre 公式.

要证明第二个等号，首先将 n 展开为 p 进制，这相当于将它写作如下和式：

$$n = n_\ell p^\ell + \cdots + n_1 p + n_0 = \sum_{k=0}^{\ell} n_k p^k.$$

因此，有

$$\begin{aligned}\nu_p(n!) &= \sum_{i=1}^{\ell} \left\lfloor \frac{n}{p^i} \right\rfloor = \sum_{i=1}^{\ell} \sum_{k=i}^{\ell} n_k p^{k-i} = \sum_{k=1}^{\ell} n_k \sum_{i=1}^{k} p^{k-i} \\ &= \sum_{k=1}^{\ell} n_k \frac{p^k - 1}{p-1} = \frac{\sum_{k=0}^{\ell} n_k p^k - \sum_{k=0}^{\ell} n_k}{p-1} = \frac{n - S_p(n)}{p-1}.\end{aligned}$$

求阶乘中素数幂次的参考实现如下：

参考实现

```cpp
// Obtain multiplicity of p in n!.
int multiplicity_factorial(int n, int p) {
  int count = 0;
  do {
    n /= p;
    count += n;
  } while (n);
  return count;
}

```

它的时间复杂度为 $O(\log n)$.

Kummer 定理

组合数对一个数取模的结果，往往构成分形结构，例如谢尔宾斯基三角形就可以通过组合数模 2 得到.

如果仔细分析，p 是否整除组合数其实和上下标在 p 进制下减法是否需要借位有关. 这就有了 **Kummer 定理**.

Kummer 定理

素数 p 在组合数 $\binom{m}{n}$ 中的幂次，恰好是 p 进制下 m 减掉 n 需要借位的次数，亦即

$$\nu_p\left(\binom{m}{n}\right) = \frac{S_p(n) + S_p(m-n) - S_p(m)}{p-1}.$$

特别地，组合数中 2 的幂次是 $\nu_2\left(\binom{m}{n}\right) = S_2(n) + S_2(m-n) - S_2(m)$.

证明

首先证明下面的表达式．为此，利用 Legendre 公式，有

$$\begin{aligned}\nu_p\left(\binom{m}{n}\right) &= \nu_p(m!) - \nu_p(n!) - \nu_p((m-n)!) \\ &= \sum_{i=1}^{\infty}\left(\left\lfloor\frac{m}{p^i}\right\rfloor - \left\lfloor\frac{n}{p^i}\right\rfloor - \left\lfloor\frac{m-n}{p^i}\right\rfloor\right) \\ &= \frac{S_p(n)+S_p(m-n)-S_p(m)}{p-1}.\end{aligned}$$

该表达式可以理解为 p 进制下 m 减掉 n 需要借位的次数．因为如果在计算第 i 位（最低位下标是 1）时存在不够减需要借位的情况，那么相减的结果中第 i 位之前的数字 $\left\lfloor\frac{m-n}{p^i}\right\rfloor$，其实是 m 中第 i 位之前的数字 $\left\lfloor\frac{m}{p^i}\right\rfloor$，减去一（即借掉的一），再减去 n 中第 i 位之前的数字 $\left\lfloor\frac{n}{p^i}\right\rfloor$ 得到的差值，所以，差值

$$\left\lfloor\frac{m}{p^i}\right\rfloor - \left\lfloor\frac{n}{p^i}\right\rfloor - \left\lfloor\frac{m-n}{p^i}\right\rfloor = 1$$

当且仅当发生了一次借位；否则，该差值为 0．因此，上述表达式中的求和式就可以理解为借位发生的次数．这就得到了 Kummer 定理的文字表述．

例题

例题 HDU 2973 - YAPTCHA[1]

给定 n，计算

$$\sum_{k=1}^{n}\left\lfloor\frac{(3k+6)!+1}{3k+7} - \left\lfloor\frac{(3k+6)!}{3k+7}\right\rfloor\right\rfloor$$

解题思路

若 $3k+7$ 是素数，则

$$(3k+6)! \equiv -1 \pmod{3k+7}$$

设 $(3k+6)!+1 = m(3k+7)$

则

$$\left\lfloor\frac{(3k+6)!+1}{3k+7} - \left\lfloor\frac{(3k+6)!}{3k+7}\right\rfloor\right\rfloor = \left\lfloor m - \left\lfloor m - \frac{1}{3k+7}\right\rfloor\right\rfloor = 1$$

若 $3k+7$ 不是素数，则它是大于 4 的合数，有 $(3k+7) \mid (3k+6)!$，即

$$(3k+6)! \equiv 0 \pmod{3k+7}$$

设 $(3k+6)! = m(3k+7)$，则

$$\left\lfloor\frac{(3k+6)!+1}{3k+7} - \left\lfloor\frac{(3k+6)!}{3k+7}\right\rfloor\right\rfloor = \left\lfloor m + \frac{1}{3k+7} - m\right\rfloor = 0$$

因此

$$\sum_{k=1}^{n}\left\lfloor\frac{(3k+6)!+1}{3k+7} - \left\lfloor\frac{(3k+6)!}{3k+7}\right\rfloor\right\rfloor = \sum_{k=1}^{n}[3k+7 \text{ is prime}]$$

参考代码

```cpp
#include <iostream>

constexpr int M = 1e6 + 5, N = 3 * M + 7;

bool not_prime[N];
int sum[M];

int main() {
  for (int i = 2; i < N; ++i)
    if (!not_prime[i])
      for (int j = 2; j * i < N; ++j) not_prime[j * i] = true;
  for (int i = 1; i < M; ++i) sum[i] = sum[i - 1] + !not_prime[3 * i + 7];

  int t;
  std::cin >> t;
  while (t--) {
    int n;
    std::cin >> n;
    std::cout << sum[n] << std::endl;
  }
}
```

参考资料

- 冯克勤.《初等数论及其应用》.
- Wilson's theorem - Wikipedia[2]
- Legendre's formula - Wikipedia[3]

本页面主要译自博文 Вычисление факториала по модулю[4] 与其英文翻译版 Factorial modulo p[5]．其中俄文版版权协议为 Public Domain + Leave a Link；英文版版权协议为 CC-BY-SA 4.0．内容有改动．

外部参考资料

[1] HDU 2973 - YAPTCHA

[2] Wilson's theorem - Wikipedia

[3] Legendre's formula - Wikipedia

[4] Вычисление факториала по модулю

[5] Factorial modulo p

9.11.15 卢卡斯定理

前置知识：阶乘取模→9.11.14@p1242

引入

本文讨论大组合数取模的求解．组合数，又称二项式系数，指表达式：

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}.$$

规模不大时，组合数可以通过 递推公式→9.13.1@p1612 求解，时间复杂度为 $O(nk)$；也可以在较大的素数模数 $p > n$ 下，通过计算分子和分母的阶乘在 $O(n)$ 时间内求解．但当问题规模很大（$n \sim 10^{18}$）时，这些方法不再适用．

基于 Lucas 定理及其推广，本文讨论一种可以在模数不太大（$m \sim 10^6$）时求解组合数的方法．更准确地说，只要模数的唯一分解 $m = \prod p_i^{e_i}$ 中所有素数幂的和（即 $\sum p_i^{e_i}$）在 10^6 规模时就可以使用该方法，因为算法的预处理大致相当于这一规模．

Lucas 定理

首先讨论模数为素数 p 的情形．此时，有 Lucas 定理：

Lucas 定理

对于素数 p，有

$$\binom{n}{k} \equiv \binom{\lfloor n/p \rfloor}{\lfloor k/p \rfloor} \binom{n \bmod p}{k \bmod p} \pmod p.$$

其中，当 $n < k$ 时，二项式系数 $\binom{n}{k}$ 规定为 0.

利用生成函数证明

考虑 $\binom{p}{n} \bmod p$ 的取值．因为

$$\binom{p}{n} = \frac{p!}{n!(p-n)!},$$

所以，当 $n \neq 0, p$ 时，分母中都没有因子 p，但分子中有因子 p，所以分式一定是 p 的倍数，模 p 的余数是 0；当 $n = 0, p$ 时，分式就是 1．因此，

$$\binom{p}{n} \equiv [n = 0 \vee n = p] \pmod p.$$

记 $f(x) = ax^n + bx^m$．一般地，由 二项式展开→9.13.1@p1612 和 费马小定理→9.11.9@p1223 有

$$\begin{aligned}(f(x))^p &= (ax^n + bx^m)^p \\ &= \sum_{k=0}^{p} \binom{p}{k} (ax^n)^k (bx^m)^{p-k} \\ &\equiv a^p x^{pn} + b^p x^{pm} \\ &\equiv a(x^p)^n + b(x^p)^m \\ &= f(x^p) \pmod p.\end{aligned}$$

其中，第三行的同余利用了前文说明的结论，即只有 $k = 0, p$ 时，组合数才不是 p 的倍数．

利用这一结论，考察二项式展开：

$$\begin{aligned}(1+x)^n &= (1+x)^{p\lfloor n/p \rfloor} (1+x)^{n \bmod p} \\ &\equiv (1+x^p)^{\lfloor n/p \rfloor} (1+x)^{n \bmod p} \pmod p.\end{aligned}$$

等式左侧中，项 x^k 的系数为

$$\binom{n}{k} \bmod p.$$

转而计算等式右侧中项 x^k 的系数．第一个因子中各项的次数必然是 p 的倍数，第二个因子中各项的次数必然小于 p，而 k 分解成这样两部分的和的方式是唯一的，即带余除法：$k = p\lfloor k/p \rfloor + (k \bmod p)$．因此，第一个因子只能贡献其 $p\lfloor k/p \rfloor$ 次项，第二个因子只能贡献其 $k \bmod p$ 次项．所以，等式右侧中 x^k 的系数为两个因子各自贡献的项的系数的乘积：

$$\binom{\lfloor n/p \rfloor}{\lfloor k/p \rfloor} \binom{n \bmod p}{k \bmod p} \bmod p.$$

令两侧系数相等，就得到 Lucas 定理．

利用阶乘取模的结论证明

此处提供一种基于 阶乘取模→9.11.14@p1242 相关结论的证明方法，以方便和后文 exLucas 部分的方法建立联系．已知二项式系数

$$\binom{n}{k}=\frac{n!}{k!(n-k)!}.$$

将阶乘 $n!$ 中 p 的幂次和其他因子分离，得到分解：

$$n!=p^{\nu_p(n!)}(n!)_p.$$

就得到二项式系数的表达式：

$$\binom{n}{k}=p^{\nu_p(n!)-\nu_p(k!)-\nu_p((n-k)!)}\frac{(n!)_p}{(k!)_p((n-k)!)_p}.$$

幂次 $\nu_p(n!)$ 和阶乘余数 $(n!)_p \bmod p$ 都有递推公式：

$$\nu_p(n!)=\lfloor n/p\rfloor+\nu_p(\lfloor n/p\rfloor!),$$

$$(n!)_p\equiv(-1)^{\lfloor n/p\rfloor}\cdot(n \bmod p)!\cdot(\lfloor n/p\rfloor!)_p \pmod p.$$

前者是 Legendre 公式的推论，后者是 Wilson 定理的推论．

将递推公式代入二项式系数的表达式并整理，就得到：

$$\binom{n}{k}\equiv(-p)^{\lfloor n/p\rfloor-\lfloor k/p\rfloor-\lfloor(n-k)/p\rfloor}\cdot\frac{(n \bmod p)!}{(k \bmod p)!((n-k) \bmod p)!}$$

$$\cdot p^{\nu_p(\lfloor n/p\rfloor!)-\nu_p(\lfloor k/p\rfloor!)-\nu_p(\lfloor(n-k)/p\rfloor!)}\frac{(\lfloor n/p\rfloor!)_p}{(\lfloor k/p\rfloor!)_p(\lfloor(n-k)/p\rfloor!)_p} \pmod p.$$

现在考察 $\lfloor n/p\rfloor-\lfloor k/p\rfloor-\lfloor(n-k)/p\rfloor$ 的取值．因为有

$$n=\lfloor n/p\rfloor p+(n \bmod p),$$

$$k=\lfloor k/p\rfloor p+(k \bmod p),$$

$$n-k=\lfloor(n-k)/p\rfloor p+((n-k) \bmod p),$$

所以，利用第一式减去后两式，就得到

$$(\lfloor n/p\rfloor-\lfloor k/p\rfloor-\lfloor(n-k)/p\rfloor)p=(k \bmod p)+((n-k) \bmod p)-(n \bmod p).$$

等式右侧，前两项的和严格小于 $2p$，而第三项 $n \bmod p$ 正是前两项的和的余数，所以右侧必然非负，但小于 $2p$，又需要是 p 的倍数，就只能是 0 或 p．这说明 $\lfloor n/p\rfloor-\lfloor k/p\rfloor-\lfloor(n-k)/p\rfloor$ 只能是 0 或 1：

- 如果它是 0，那么此时也成立 $(n \bmod p)=(k \bmod p)+((n-k) \bmod p)$．因此，上式中的第一个因子的指数为 0，该因子就等于一；第二个因子就是 $\binom{n \bmod p}{k \bmod p}$；第三个因子则由前文的展开式（注意此时 $\lfloor(n-k)/p\rfloor=\lfloor n/p\rfloor-\lfloor k/p\rfloor$）可知，就等于 $\binom{\lfloor n/p\rfloor}{\lfloor k/p\rfloor}$．此时，Lucas 定理成立；
- 如果它是 1，那么第一个因子的指数为 1，该因子就等于零，所以二项式系数的余数为零．同时，Lucas 定理所要证明的等式右侧的 $\binom{n \bmod p}{k \bmod p}$ 也必然是零，因为此时必然有 $(n \bmod p)<(k \bmod p)$；否则，将有
  $((n-k) \bmod p)=p+(n \bmod p)-(k \bmod p)\geq p.$
  这显然与余数的定义矛盾．

综合两种情形，就得到了所要求证的 Lucas 定理．这一证明说明，在求解素数模下组合数时，利用 Lucas 定理和利用 exLucas 算法得到的结果是等价的．

Lucas 定理指出，模数为素数 p 时，大组合数的计算可以转化为规模更小的组合数的计算．在右式中，第一个组合数可以继续递归，直到 $k=0$ 为止；第二个组合数则可以直接计算，或者提前预处理出来．写成代码的形式就是：

示意

```
long long Lucas(long long n, long long k, long long p) {
  if (k == 0) return 1;
  return (C(n % p, k % p, p) * Lucas(n / p, k / p, p)) % p;
}
```

其中，`C(n, k, p)` 用于计算小规模的组合数.

递归至多进行 $O(\log_p n)$ 次，因而算法的复杂度为 $O(f(p) + g(p) \log_p n)$，其中，$f(p)$ 为预处理组合数的复杂度，$g(p)$ 为单次计算组合数的复杂度.

参考实现

此处给出的参考实现在 $O(p)$ 时间内预处理 p 以内的阶乘及其逆元后，可以在 $O(1)$ 时间内计算单个组合数:

参考实现

```
#include <iostream>
#include <vector>

class BinomModPrime {
  int p;
  std::vector<int> fa, ifa;

  // Calculate binom(n, k) mod p for n, k < p.
  int calc(int n, int k) {
    if (n < k) return 0;
    long long res = fa[n];
    res = (res * ifa[k]) % p;
    res = (res * ifa[n - k]) % p;
    return res;
  }

 public:
  BinomModPrime(int p) : p(p), fa(p), ifa(p) {
    // Factorials mod p till p.
    fa[0] = 1;
    for (int i = 1; i < p; ++i) {
      fa[i] = (long long)fa[i - 1] * i % p;
    }
    // Inverse of factorials mod p till p.
    ifa[p - 1] = p - 1;  // Wilson's theorem.
    for (int i = p - 1; i; --i) {
      ifa[i - 1] = (long long)ifa[i] * i % p;
    }
  }

  // Calculate binom(n, k) mod p.
  int binomial(long long n, long long k) {
    long long res = 1;
    while (n || k) {
```

```
      res = (res * calc(n % p, k % p)) % p;
      n /= p;
      k /= p;
    }
    return res;
  }
};

int main() {
  int t, p;
  std::cin >> t >> p;
  BinomModPrime bm(p);
  for (; t; --t) {
    long long n, k;
    std::cin >> n >> k;
    std::cout << bm.binomial(n, k) << '\n';
  }
  return 0;
}

```

该实现的时间复杂度为 $O(p + T \log_p n)$，其中，T 为询问次数.

exLucas 算法

Lucas 定理中对于模数 p 要求必须为素数，那么对于模数不是素数的情况，就需要用到 exLucas 算法. 虽然名字如此，该算法实际操作时并没有用到 Lucas 定理. 它的关键步骤是 计算素数幂模下的阶乘 →9.11.14@p1242. 上文的第二个证明指出了它与 Lucas 定理的联系.

素数幂模的情形

首先考虑模数为素数幂 p^α 的情形. 将阶乘 $n!$ 中的 p 的幂次和其他因子分开，可以得到分解:

$$n! = p^{\nu_p(n!)}(n!)_p.$$

其中，$\nu_p(n!)$ 为 $n!$ 的素因数分解中 p 的幂次，而 $(n!)_p$ 显然与 p 互素. 因此，组合数可以写作:

$$\binom{n}{k} = p^{\nu_p(n!)-\nu_p(k!)-\nu_p((n-k)!)} \frac{(n!)_p}{(k!)_p((n-k)!)_p}.$$

式子中的 $\nu_p(n!)$ 等可以通过 Legendre 公式→9.11.14@p1242 计算，$(n!)_p$ 等则可以通过 递推关系→9.11.14@p1242 计算. 因为后者与 p^α 互素，所以分母上的乘积的逆元可以通过 扩展欧几里得算法→9.11.10@p1229 计算. 问题就得以解决.

注意，如果幂次 $\nu_p(n!) - \nu_p(k!) - \nu_p((n-k)!) \geq \alpha$，余数一定为零，不必再做更多计算.

一般模数的情形

对于 m 是一般的合数的情形，只需要首先对它做 素因数分解→9.11.7@p1209:

$$m = p_1^{\alpha_1} p_2^{\alpha_2} \cdots p_s^{\alpha_s}.$$

然后，分别计算出模 $p_i^{\alpha_i}$ 下组合数 $\binom{n}{k}$ 的余数，就得到 s 个同余方程:

$$\begin{cases} \binom{n}{k} \equiv r_1 \pmod{p_1^{\alpha_1}}, \\ \binom{n}{k} \equiv r_2 \pmod{p_2^{\alpha_2}}, \\ \quad \cdots \\ \binom{n}{k} \equiv r_s \pmod{p_s^{\alpha_s}}. \end{cases}$$

最后，利用 中国剩余定理→9.11.12@p1236 求出模 m 的余数.

参考实现

最后，给出模板题目 二项式系数[1] 的参考实现.

参考实现

```cpp
#include <iostream>
#include <vector>

// Extended Euclid.
void ex_gcd(int a, int b, int& x, int& y) {
  if (!b) {
    x = 1;
    y = 0;
  } else {
    ex_gcd(b, a % b, y, x);
    y -= a / b * x;
  }
}

// Inverse of a mod m.
int inverse(int a, int m) {
  int x, y;
  ex_gcd(a, m, x, y);
  return (x % m + m) % m;
}

// Coefficient in CRT.
int crt_coeff(int m_i, int m) {
  long long mm = m / m_i;
  mm *= inverse(mm, m_i);
  return mm % m;
}

// Binominal Coefficient Calculator Modulo Prime Power.
class BinomModPrimePower {
  int p, a, pa;
  std::vector<int> f;

  // Obtain multiplicity of p in n!.
  long long nu(long long n) {
    long long count = 0;
    do {
      n /= p;
      count += n;
    } while (n);
```

```
41      return count;
42    }
43
44    // Calculate (n!)_p mod pa.
45    long long fact_mod(long long n) {
46      bool neg = p != 2 || pa <= 4;
47      long long res = 1;
48      while (n > 1) {
49        if ((n / pa) & neg) res = pa - res;
50        res = res * f[n % pa] % pa;
51        n /= p;
52      }
53      return res;
54    }
55
56   public:
57    BinomModPrimePower(int p, int a, int pa) : p(p), a(a), pa(pa), f(pa) {
58      // Pretreatment.
59      f[0] = 1;
60      for (int i = 1; i < pa; ++i) {
61        f[i] = i % p ? (long long)f[i - 1] * i % pa : f[i - 1];
62      }
63    }
64
65    // Calculate Binom(n, k) mod pa.
66    int binomial(long long n, long long k) {
67      long long v = nu(n) - nu(n - k) - nu(k);
68      if (v >= a) return 0;
69      auto res = fact_mod(n - k) * fact_mod(k) % pa;
70      res = fact_mod(n) * inverse(res, pa) % pa;
71      for (; v; --v) res *= p;
72      return res % pa;
73    }
74  };
75
76  // Binominal Coefficient Calculator.
77  class BinomMod {
78    int m;
79    std::vector<BinomModPrimePower> bp;
80    std::vector<long long> crt_m;
81
82   public:
83    BinomMod(int n) : m(n) {
84      // Factorize.
85      for (int p = 2; p * p <= n; ++p) {
86        if (n % p == 0) {
87          int a = 0, pa = 1;
88          for (; n % p == 0; n /= p, ++a, pa *= p);
89          bp.emplace_back(p, a, pa);
90          crt_m.emplace_back(crt_coeff(pa, m));
91        }
92      }
```

```
    if (n > 1) {
      bp.emplace_back(n, 1, n);
      crt_m.emplace_back(crt_coeff(n, m));
    }
  }

  // Calculate Binom(n, k) mod m.
  int binomial(long long n, long long k) {
    long long res = 0;
    for (size_t i = 0; i != bp.size(); ++i) {
      res = (bp[i].binomial(n, k) * crt_m[i] + res) % m;
    }
    return res;
  }
};

int main() {
  int t, m;
  std::cin >> t >> m;
  BinomMod bm(m);
  for (; t; --t) {
    long long n, k;
    std::cin >> n >> k;
    std::cout << bm.binomial(n, k) << '\n';
  }
  return 0;
}

```

该算法在预处理时将模数 m 分解为素数幂，然后对所有 p^α 预处理了自 1 至 p^α 所有非 p 倍数的自然数的乘积，以及它在中国剩余定理合并答案时对应的系数．预处理的时间复杂度为 $O(\sqrt{m}+\sum_i p_i^{\alpha_i})$．每次询问时，复杂度为 $O(\log m+\sum_i \log_{p_i} n)$，复杂度中的两项分别是计算逆元和计算幂次、阶乘余数的复杂度．

习题

- Luogu3807【模板】卢卡斯定理[2]
- SDOI2010 古代猪文 卢卡斯定理[3]
- Luogu4720【模板】扩展卢卡斯[4]
- Ceizenpok's formula[5]

外部参考资料

[1] 二项式系数

[2] Luogu3807【模板】卢卡斯定理

[3] SDOI2010 古代猪文 卢卡斯定理

[4] Luogu4720【模板】扩展卢卡斯

[5] Ceizenpok's formula

9.11.16 同余方程

> **Authors:** iamtwz, aofall, CCXXXI, CoelacanthusHex, Great-designer, Marcythm, Persdre, shuzhouliu, Tiphereth-A, Xeonacid

定义

同余方程

对正整数 m 和一元整系数多项式 $f(x)=\sum_{i=0}^{n}a_ix^i$，其中未知数 $x\in\mathbf{Z}_m$，称形如

$$f(x)\equiv 0 \pmod m \tag{1}$$

的方程为关于未知数 x 的模 m 的一元 **同余方程**（Congruence Equation）.

若 $a_n\not\equiv 0 \pmod m$，则称上式为 n 次同余方程.

类似可定义同余方程组.

关于一次同余方程与方程组的相关内容请参见 线性同余方程→9.11.11@p1233 与 中国剩余定理→9.11.12@p1236.

本文首先研究同余方程的可解性和解集结构，之后会简要介绍高次同余方程的解法.

由 中国剩余定理→9.11.12@p1236 可知，求解关于模合数 m 的同余方程可转化为求解模素数幂次的情况. 故以下只介绍素数幂模同余方程和素数模同余方程的相关理论.

素数幂模同余方程

以下假设模数 $m=p^e$ $(p\in\mathbf{P},\ e\in\mathbf{Z}_{>1})$.

注意到，若 x_0 是方程

$$f(x)\equiv 0 \pmod{p^e}$$

的解，则 x_0 是方程

$$f(x)\equiv 0 \pmod{p^{e-1}}$$

的解. 这启发我们尝试通过较低的模幂次的解去构造较高的模幂次的解. 我们有如下定理:

定理 1（Hensel 引理）

对素数 p 和整数 $e>1$，取整系数多项式 $f(x)=\sum_{i=0}^{n}a_ix^i$ $(p^e\nmid a_n)$，令 $f'(x)=\sum_{i=1}^{n}ia_ix^{i-1}$ 为其导数. 令 x_0 为方程

$$f(x)\equiv 0 \pmod{p^{e-1}} \tag{2}$$

的解，则:

1. 若 $f'(x_0)\not\equiv 0 \pmod p$，则存在整数 t 使得

   $$x=x_0+p^{e-1}t \tag{3}$$

   是方程

   $$f(x)\equiv 0 \pmod{p^e} \tag{4}$$

   的解.
2. 若 $f'(x_0)\equiv 0 \pmod p$ 且 $f(x_0)\equiv 0 \pmod{p^e}$，则对 $t=0,1,\ldots,p-1$，由式 (3) 确定的 x 均为方程 (4) 的解.
3. 若 $f'(x_0)\equiv 0 \pmod p$ 且 $f(x_0)\not\equiv 0 \pmod{p^e}$，则不能由式 (3) 构造方程 (4) 的解.

证明

我们假设式 (3) 是方程 (4) 的解，即

$$f(x_0+p^{e-1}t)\equiv 0 \pmod{p^e}$$

整理后可得

$$f(x_0)+p^{e-1}tf'(x_0)\equiv 0 \pmod{p^e}$$

于是

$$tf'(x_0) \equiv -\frac{f(x_0)}{p^{e-1}} \pmod p \tag{5}$$

1. 若 $f'(x_0) \not\equiv 0 \pmod p$, 则关于 t 的方程 (5) 有唯一解 t_0, 代入式 (3) 可验证其为方程 (4) 的解.
2. 若 $f'(x_0) \equiv 0 \pmod p$ 且 $f(x_0) \equiv 0 \pmod{p^e}$，则任意 t 均能使方程 (5) 成立，代入式 (3) 可验证其均为方程 (4) 的解.
3. 若 $f'(x_0) \equiv 0 \pmod p$ 且 $f(x_0) \not\equiv 0 \pmod{p^e}$，则方程 (5) 无解，从而不能由式 (3) 构造方程 (4) 的解.

进而我们有推论:

推论 1

对定理 1 的 p，e，$f(x)$，x_0，

1. 若 s 是方程 $f(x) \equiv 0 \pmod p$ 的解，且 $f'(s) \not\equiv 0 \pmod p$，则存在 $x_s \in \mathbf{Z}_{p^e}$，$x_s \equiv s \pmod p$ 使得 x_s 是方程 (4) 的解.
2. 若方程 $f(x) \equiv 0 \pmod p$ 与 $f'(x) \equiv 0 \pmod p$ 无公共解，则方程 (4) 和方程 $f(x) \equiv 0 \pmod p$ 的解数相同.

从而我们可以将素数幂模同余方程化归到素数模同余方程的情况.

素数模同余方程

以下令 $p \in \mathbf{P}$，整系数多项式 $f(x) = \sum_{i=0}^{n} a_i x^i$，其中 $p \nmid a_n$，$x \in \mathbf{Z}_p$.

定理 2

若方程

$$f(x) \equiv 0 \pmod p \tag{6}$$

有 k 个不同的解 $x_1, x_2, ..., x_k$ $(k \le n)$，则

$$f(x) \equiv g(x) \prod_{i=1}^{k} (x - x_i) \pmod p,$$

其中 $\deg g = n - k$ 且 $[x^{n-k}]g(x) = a_n$.

证明

对 k 应用数学归纳法.

- 当 $k = 1$ 时，做多项式带余除法，有 $f(x) = (x - x_1)g(x) + r$，其中 $r \in \mathbf{Z}$.

  由 $f(x_1) \equiv 0 \pmod p$ 可知 $r \equiv 0 \pmod p$，从而 $f(x) \equiv (x - x_1)g(x) \pmod p$.
- 假设命题对 $k - 1 (k > 1)$ 时的情况成立，现在设 $f(x)$ 有 k 个不同的解 $x_1, x_2, ..., x_k$，则 $f(x) \equiv (x - x_1)h(x) \pmod p$，进而有

  $(\forall i = 2, 3, ..., k),\ 0 \equiv f(x_i) \equiv (x_i - x_1)h(x_i) \pmod p$

  从而 $h(x)$ 有 $k - 1$ 个不同的解 $x_2, x_3, ..., x_k$，由归纳假设有

  $$h(x) \equiv g(x) \prod_{i=2}^{k} (x - x_i) \pmod p$$

  其中 $\deg g = n - k$ 且 $[x^{n-k}]g(x) = a_n$.

  因此命题得证.

推论 2

对素数 p，

- $(\forall x \in \mathbf{Z}),\quad x^{p-1}-1 \equiv \prod_{i=1}^{p-1}(x-i) \pmod p$.
- （Wilson 定理→9.11.14@p1242）$(p-1)! \equiv -1 \pmod p$.

定理 3（Lagrange 定理）

方程 (6) 至多有 n 个不同解.

证明

假设 $f(x)$ 有 $n+1$ 个不同解 $x_1, x_2, ..., x_{n+1}$，则由定理 2，对 $x_1, x_2, ..., x_n$ 有

$$f(x) \equiv a_n \prod_{i=1}^{n}(x-x_i) \pmod p$$

令 $x = x_{n+1}$，则

$$0 \equiv f(x_{n+1}) \equiv a_n \prod_{i=1}^{n}(x_{n+1}-x_i) \pmod p$$

而右侧显然不是 p 的倍数，因此假设矛盾.

推论 3

若同余方程 $\sum_{i=0}^{n} b_i x^i \equiv 0 \pmod p$ 的解数大于 n，则

$$(\forall i = 0, 1, ..., n),\quad p \mid b_i.$$

定理 4

方程 (6) 若解的个数不为 p，则必存在满足 $\deg r < p$ 的整系数多项式 $r(x)$ 使得 $f(x) \equiv 0 \pmod p$ 和 $r(x) \equiv 0 \pmod p$ 的解集相同.

证明

不妨设 $n \geq p$，对 $f(x)$ 做多项式带余除法

$$f(x) = g(x)(x^p - x) + r(x)$$

其中 $\deg r < p$.

由 Fermat 小定理→9.11.9@p1223 知对任意整数 x 有 $x^p \equiv x \pmod p$，从而

- 若 $r(x) \equiv 0 \pmod p$，则由推论 2 可知 $f(x)$ 有 p 个不同的解.
- 若 $r(x) \not\equiv 0 \pmod p$，则由 $f(x) \equiv r(x) \pmod p$ 可知 $f(x)$ 和 $r(x)$ 的解集相同.

我们可以通过这个定理对同余方程降次.

定理 5

设 $n \leq p$，则方程

$$x^n + \sum_{i=0}^{n-1} a_i x^i \equiv 0 \pmod p \tag{7}$$

有 n 个解当且仅当存在整系数多项式 $q(x)$，$r(x)$ $(\deg r < n)$ 使得

$$x^p - x = f(x)q(x) + pr(x). \tag{8}$$

证明

- 必要性：由多项式除法知存在整系数多项式 $q(x)$，$r_1(x)$ $(\deg r_1 < n)$ 使得
  $x^p - x = f(x)q(x) + r_1(x)$.
  若方程 (7) 有 n 个解，则 $r_1 \equiv 0 \pmod p$ 也有 n 个相同的解，进而由推论 3 可知存在整系数多项式 $r(x)$ 满足 $r_1(x) = pr(x)$，从而命题得证.
- 充分性：若式 (8) 成立，则由 Fermat 小定理→9.11.9@p1223 可知，对任意整数 x，
  $0 \equiv x^p - x \equiv f(x)q(x) \pmod p$.
  即方程 $f(x)q(x) \equiv 0 \pmod p$ 有 p 个解.
  设方程 (7) 的解数为 s，则由 Lagrange 定理可知 $s \le n$.
  又由于 $\deg q = p - n$，则由 Lagrange 定理可知 $q(x) \equiv 0 \pmod p$ 的解数不超过 $p - n$，而方程 $f(x)q(x) \equiv 0 \pmod p$ 的解集是 $f(x) \equiv 0 \pmod p$ 解集和 $q(x) \equiv 0 \pmod p$ 解集的并集，故 $s + (p - n) \ge p$，有 $s \ge n$.
  因此 $s = n$.

对于非首一多项式，由于 $\mathbf{Z}_p$ 是域，故可以将其化为首一多项式，从而适用该定理.

定理 6

设 $n \mid p - 1$，$p \nmid a$，则方程

$$x^n \equiv a \pmod p \tag{9}$$

有解当且仅当

$$a^{\frac{p-1}{n}} \equiv 1 \pmod p.$$

而且，若 (9) 有解，则解数为 n.

Note

方程 (9) 解集的具体结构可参见 k 次剩余→9.11.20@p1291.

证明

- 必要性：若方程 (9) 有解 x_0，则
  $a^{\frac{p-1}{n}} \equiv (x_0^n)^{\frac{p-1}{n}} \equiv 1 \pmod p$
- 充分性：若 $a^{\frac{p-1}{n}} \equiv 1 \pmod p$，则

$$\begin{aligned} x^p - x &= x(x^{p-1} - 1) \\ &= x\left((x^n)^{\frac{p-1}{n}} - a^{\frac{p-1}{n}} + a^{\frac{p-1}{n}} - 1\right) \\ &= (x^n - a)P(x) + x\left(a^{\frac{p-1}{n}} - 1\right) \end{aligned}$$

其中 $P(x)$ 是某个整系数多项式，因此由定理 5 可知方程 (9) 有 n 个解.

高次同余方程（组）的求解方法

首先我们可以借助 中国剩余定理→9.11.12@p1236 将求解 **同余方程组** 转为求解 **同余方程**，以及将求解模 **合数** m 的同余方程转化为求解模 **素数幂次** 的同余方程．之后我们借助 定理 1→9.11@p1160 将求解模 **素数幂次** 的同余方程转化为求解模 **素数** 的同余方程．

结合模素数同余方程的若干定理，我们只需考虑方程

$$x^n+\sum_{i=0}^{n-1}a_ix^i\equiv 0 \pmod p$$

的求法，其中 p 是素数，$n<p$．

我们可以通过将 x 代换为 $x-\dfrac{a_{n-1}}{n}$ 来消去 x^{n-1} 项，从而我们只需考虑方程

$$x^n+\sum_{i=0}^{n-2}a_ix^i\equiv 0 \pmod p \tag{10}$$

的求法，其中 p 是素数，$n<p$．

- 若 $n=1$，则求法参见 线性同余方程→9.11.11@p1233．
- 若 $n=2$，则求法参见 二次剩余→9.11.17@p1263．
- 若方程 (10) 可化为

  $x^n\equiv a \pmod p$,

  则求法参见 k 次剩余→9.11.20@p1291．

参考资料

1. Congruence Equation -- from Wolfram MathWorld[1]
2. Lagrange's theorem (number theory) - Wikipedia[2]
3. 潘承洞，潘承彪．初等数论．
4. 冯克勤．初等数论及其应用．
5. 闵嗣鹤，严士健．初等数论．

外部参考资料

[1] Congruence Equation -- from Wolfram MathWorld

[2] Lagrange's theorem (number theory) - Wikipedia

9.11.17 二次剩余

Authors: hly1204, ShaoChenHeng, Chrogeek, Enter-tainer, Great-designer, iamtwz, monkeysui, nanmenyangde, rgw2010, sshwy, StudyingFather, TachikakaMin, Tiphereth-A, Xeonacid, xyf007, marscheng1

引入

二次剩余可以认为是在讨论求模意义下 **开平方** 运算的可行性．对于更高次方的开方可参见 k 次剩余→9.11.20@p1291．

定义

自 二次剩余

令整数 a，p 满足 $(a,p)=1$，若存在整数 x 使得

$$x^2\equiv a \pmod p,$$

则称 a 为模 p 的二次剩余，否则称 a 为模 p 的二次非剩余．后文可能在模 p 显然的情况下简写成二次（非）剩余．

Euler 判别法

当模数为奇素数时，我们有如下定理：

自 Euler 判别法

对奇素数 p 和满足 $(a,p)=1$ 的整数 a，有

$$a^{\frac{p-1}{2}} \equiv \begin{cases} 1 \pmod p, & (\exists x \in \mathbf{Z}),\ a \equiv x^2 \pmod p, \\ -1 \pmod p, & \text{otherwise.} \end{cases}$$

即对上述的 p 和 a，

1. a 是 p 的二次剩余当且仅当 $a^{\frac{p-1}{2}} \equiv 1 \pmod p$.
2. a 是 p 的二次非剩余当且仅当 $a^{\frac{p-1}{2}} \equiv -1 \pmod p$.

证明

首先由 Fermat 小定理→9.11.9@p1223 有 $a^{p-1} \equiv 1 \pmod p$，故

$$\left(a^{\frac{p-1}{2}}+1\right)\left(a^{\frac{p-1}{2}}-1\right) \equiv 0 \pmod p,$$

从而对任意满足 $(a,p)=1$ 的 a 均有 $a^{(p-1)/2} \equiv \pm 1 \pmod p$.

另外由 p 是奇素数，我们有：

$$x^{p-1}-a^{\frac{p-1}{2}} = \left(x^2\right)^{\frac{p-1}{2}} - a^{\frac{p-1}{2}} = (x^2-a)P(x),$$

其中 $P(x)$ 是某个整系数多项式，进而：

$$\begin{aligned} x^p - x &= x\left(x^{p-1}-a^{\frac{p-1}{2}}\right) + x\left(a^{\frac{p-1}{2}}-1\right) \\ &= (x^2-a)xP(x) + \left(a^{\frac{p-1}{2}}-1\right)x. \end{aligned}$$

由 同余方程的定理 5→9.11.16@p1258 可知，a 是模 p 的二次剩余当且仅当 $a^{(p-1)/2} \equiv 1 \pmod p$. 进而 a 是模 p 的非二次剩余当且仅当 $a^{(p-1)/2} \equiv -1 \pmod p$.

基于 Euler 判别法，我们可以得到如下推论：

二次剩余的数量

对于奇素数 p，模 p 意义下二次剩余和二次非剩余均有 $\dfrac{p-1}{2}$ 个.

证明

根据 Euler 判别法，考虑 $a^{\frac{p-1}{2}} \equiv 1 \pmod p$.

注意到 $\dfrac{p-1}{2} \mid (p-1)$，由 同余方程的定理 6→9.11.16@p1258 可知 $a^{\frac{p-1}{2}} \equiv 1 \pmod p$ 有 $\dfrac{p-1}{2}$ 个解. 所以模 p 意义下二次剩余和二次非剩余均有 $\dfrac{p-1}{2}$ 个.

Legendre 符号

为了方便接下来的讨论，我们引入如下记号：

自 Legendre 符号

对 **奇素数** p 和整数 a，定义 Legendre 符号如下：

$$\left(\frac{a}{p}\right) = \begin{cases} 0, & p \mid a, \\ 1, & (p \nmid a) \wedge ((\exists x \in \mathbf{Z}),\ a \equiv x^2 \pmod p), \\ -1, & \text{otherwise.} \end{cases}$$

即对于 $(a,p)=1$ 的 a,

- a 是模 p 的二次剩余当且仅当 $\left(\frac{a}{p}\right)=1$.
- a 是模 p 的二次非剩余当且仅当 $\left(\frac{a}{p}\right)=-1$.

下表为部分 Legendre 符号的值（From Wikipedia[1]）

p \ a	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
3	1	−1	0	1	−1	0	1	−1	0	1	−1	0	1	−1	0	1	−1	0	1	−1	0	1	−1	0	1	−1	0	1	−1	0
5	1	−1	−1	1	0	1	−1	−1	1	0	1	−1	−1	1	0	1	−1	−1	1	0	1	−1	−1	1	0	1	−1	−1	1	0
7	1	1	−1	1	−1	−1	0	1	1	−1	1	−1	−1	0	1	1	−1	1	−1	−1	0	1	1	−1	1	−1	−1	0	1	1
11	1	−1	1	1	1	−1	−1	−1	1	−1	0	1	−1	1	1	1	−1	−1	−1	1	−1	0	1	−1	1	1	1	−1	−1	−1
13	1	−1	1	1	−1	−1	−1	−1	1	1	−1	1	0	1	−1	1	1	−1	−1	−1	−1	1	1	−1	1	0	1	−1	1	1
17	1	1	−1	1	−1	−1	−1	1	1	−1	−1	−1	1	−1	1	1	0	1	1	−1	1	−1	−1	−1	1	1	−1	−1	−1	1
19	1	−1	−1	1	1	1	1	−1	1	−1	1	−1	−1	−1	−1	1	1	−1	0	1	−1	−1	1	1	1	1	−1	1	−1	1
23	1	1	1	1	−1	1	−1	1	1	−1	−1	1	1	−1	−1	1	−1	1	−1	−1	−1	−1	0	1	1	1	1	−1	1	−1
29	1	−1	−1	1	1	1	1	−1	1	−1	−1	−1	1	−1	−1	1	−1	−1	−1	1	−1	1	1	1	1	−1	−1	1	0	1
31	1	1	−1	1	1	−1	1	1	1	1	−1	−1	−1	1	−1	1	−1	1	1	1	−1	−1	−1	−1	1	−1	−1	1	−1	−1
37	1	−1	1	1	−1	−1	1	−1	1	1	1	1	−1	−1	−1	1	−1	−1	−1	−1	1	−1	−1	−1	1	1	1	1	−1	1
41	1	1	−1	1	1	−1	−1	1	1	1	−1	−1	−1	−1	−1	1	−1	1	−1	1	1	−1	1	−1	1	−1	−1	−1	−1	−1
43	1	−1	−1	1	−1	1	−1	−1	1	1	1	−1	1	1	1	1	1	−1	−1	−1	1	−1	1	1	1	−1	−1	−1	−1	−1
47	1	1	1	1	−1	1	1	1	1	−1	−1	1	−1	1	−1	1	1	1	−1	−1	1	−1	−1	1	1	−1	1	1	−1	−1
53	1	−1	−1	1	−1	1	1	−1	1	1	1	−1	1	−1	1	1	1	−1	−1	−1	−1	−1	−1	1	1	−1	−1	1	1	−1
59	1	−1	1	1	1	−1	1	−1	1	−1	−1	1	−1	−1	1	1	1	−1	1	1	1	1	−1	−1	1	1	1	1	1	−1
61	1	−1	1	1	1	−1	−1	−1	1	−1	−1	1	1	1	1	1	−1	−1	1	1	−1	1	−1	−1	1	−1	1	−1	−1	−1
67	1	−1	−1	1	−1	1	−1	−1	1	1	−1	−1	−1	1	1	1	1	−1	1	−1	1	1	1	1	1	1	−1	−1	1	−1
71	1	1	1	1	1	1	−1	1	1	1	−1	1	−1	−1	1	1	−1	1	1	1	−1	−1	−1	1	1	−1	1	−1	1	1
73	1	1	1	1	−1	1	−1	1	1	−1	−1	1	−1	−1	−1	1	−1	1	1	−1	−1	−1	1	1	1	−1	1	−1	−1	−1
79	1	1	−1	1	1	−1	−1	1	1	1	1	−1	1	−1	−1	1	−1	1	1	1	1	1	1	−1	1	1	−1	−1	−1	−1
83	1	−1	1	1	−1	−1	1	−1	1	1	1	1	−1	−1	−1	1	1	−1	−1	−1	1	−1	1	−1	1	1	1	1	1	1
89	1	1	−1	1	1	−1	−1	1	1	1	1	−1	−1	−1	−1	1	1	1	−1	1	1	1	−1	−1	1	−1	−1	−1	−1	−1
97	1	1	1	1	−1	1	−1	1	1	−1	1	1	−1	−1	−1	1	−1	1	−1	−1	−1	1	−1	1	1	−1	1	−1	−1	−1
101	1	−1	−1	1	1	1	−1	−1	1	−1	−1	−1	1	1	−1	1	1	−1	1	1	1	1	1	1	1	−1	−1	−1	−1	1
103	1	1	−1	1	−1	−1	1	1	1	−1	−1	−1	1	1	1	1	1	1	1	−1	−1	−1	1	−1	1	1	−1	1	1	1
107	1	−1	1	1	−1	−1	−1	−1	1	1	1	1	1	1	−1	1	−1	−1	1	−1	−1	−1	1	−1	1	−1	1	−1	1	1
109	1	−1	1	1	1	−1	1	−1	1	−1	−1	1	−1	−1	1	1	−1	−1	−1	1	1	1	−1	−1	1	1	1	1	1	−1
113	1	1	−1	1	−1	−1	1	1	1	−1	1	−1	1	1	1	1	−1	1	−1	−1	−1	1	−1	−1	1	1	−1	1	−1	1
127	1	1	−1	1	−1	−1	−1	1	1	−1	1	−1	1	−1	1	1	1	1	1	−1	1	1	−1	−1	1	1	−1	−1	−1	1

性质

1. 对任意整数 a,

   $$a^{\frac{p-1}{2}}\equiv\left(\frac{a}{p}\right)\pmod p.$$

   进一步，我们有推论：

   - $\left(\frac{1}{p}\right)=1$.
   - $\left(\frac{-1}{p}\right)=(-1)^{\frac{p-1}{2}}=\begin{cases}1, & p\equiv 1 \pmod 4,\\ -1, & p\equiv 3 \pmod 4.\end{cases}$

2. $a_1 \equiv a_2 \pmod p \Longrightarrow \left(\frac{a_1}{p}\right) = \left(\frac{a_2}{p}\right)$.
3. （完全积性→9.11.1@p1160）对任意整数 a_1, a_2，
$\left(\frac{a_1 a_2}{p}\right) = \left(\frac{a_1}{p}\right)\left(\frac{a_2}{p}\right)$.
我们有推论：对整数 a, b，$p \nmid b$ 有
$\left(\frac{ab^2}{p}\right) = \left(\frac{a}{p}\right)$.
4. $\left(\frac{2}{p}\right) = (-1)^{\frac{p^2-1}{8}} = \begin{cases} 1, & p \equiv \pm 1 \pmod 8, \\ -1, & p \equiv \pm 3 \pmod 8. \end{cases}$

证明

1. 由 Legendre 符号的定义→9.11@p1160 和 Euler 判别法→9.11@p1160 易得.
2. 注意到
$a_1 \equiv a_2 \pmod p \Longrightarrow \left(\frac{a_1}{p}\right) \equiv \left(\frac{a_2}{p}\right) \pmod p$,
而 $\left|\left(\frac{a_1}{p}\right) - \left(\frac{a_2}{p}\right)\right| \leq 2$ 且 $p > 2$，故：
$a_1 \equiv a_2 \pmod p \Longrightarrow \left(\frac{a_1}{p}\right) = \left(\frac{a_2}{p}\right)$.
3. 由 1 得
$\left(\frac{a_1 a_2}{p}\right) \equiv a_1^{\frac{p-1}{2}} a_2^{\frac{p-1}{2}} \equiv \left(\frac{a_1}{p}\right)\left(\frac{a_2}{p}\right) \pmod p$.
而 $\left|\left(\frac{a_1 a_2}{p}\right) - \left(\frac{a_1}{p}\right)\left(\frac{a_2}{p}\right)\right| \leq 2$ 且 $p > 2$，故
$\left(\frac{a_1 a_2}{p}\right) = \left(\frac{a_1}{p}\right)\left(\frac{a_2}{p}\right)$.
4. 参见 二次互反律→9.11@p1160

基于如上性质，若对任意奇素数 p 和 q，$\left(\frac{p}{q}\right)$ 的值均可计算，则我们就可以对任意合法情况计算 Legendre 符号的值. 接下来我们有一个优美的定理，这个定理巧妙地在 $\left(\frac{p}{q}\right)$ 和 $\left(\frac{q}{p}\right)$ 之间建立起了联系，使得我们能用类似 辗转相除法→9.11.4@p1188 的思路完成计算.

二次互反律

二次互反律

设 p，q 是两个不同的奇素数，则
$$\left(\frac{p}{q}\right)\left(\frac{q}{p}\right) = (-1)^{\frac{p-1}{2}\frac{q-1}{2}}.$$

证明方式很多[24]. 一种证明方式是基于如下引理[25]:

[24] Proofs of quadratic reciprocity - Wikipedia[2]
[25] Carl Friedrich Gauss. Untersuchungen über höhere Arithmetik, 1965. Page 458-462.

Gauss 引理

设 p 是奇素数，$(n,p)=1$，对整数 k $(1\le k\le (p-1)/2)$，令 $r_k=nk \bmod p$，设 $A=\{r_k : r_k<p/2\}$，$B=\{r_k : r_k>p/2\}$，则

$$\left(\frac{n}{p}\right)=(-1)^{|B|}.$$

证明

设 $\lambda=|A|$，$\mu=|B|$，显然 $\lambda+\mu=(p-1)/2$，则

$$n^{\frac{p-1}{2}}\left(\frac{p-1}{2}\right)!=\prod_{k=1}^{\frac{p-1}{2}}nk\equiv\prod_{a\in A}a\prod_{b\in B}b\pmod p.$$

我们知道对 B 中任意元素 b，有 $\frac{p}{2}<b<p$，所以 $0<p-b<\frac{p}{2}$，进一步，对 B 中任意元素 b，我们有 $p-b\notin A$，否则若 A,B 中分别存在元素 a,b 使得 $a=p-b$，则存在整数 $0<k_1,k_2<(p-1)/2$ 使得 $a=nk_1$，$b=nk_2$ 且 $p\mid n(k_1+k_2)$，由于 $(n,p)=1$，则 $p\mid(k_1+k_2)$，注意到 $0<k_1+k_2<p$，所以产生矛盾．因此

$$n^{\frac{p-1}{2}}\left(\frac{p-1}{2}\right)!\equiv(-1)^{\mu}\prod_{a\in A}a\prod_{b\in B}(p-b)=(-1)^{\mu}\left(\frac{p-1}{2}\right)!\pmod p,$$

即

$$n^{\frac{p-1}{2}}\equiv(-1)^{\mu}\pmod p.$$

从而由 Legendre 符号的 性质 1→9.11@p1160 即得证．

推广

Gauss 引理可做如下推广[26]：

设 p 是奇素数，令 $I\subset\mathbf{Z}_p^*$ 满足 $I\cup -I=\mathbf{Z}_p^*$ 且 $I\cap -I=\emptyset$，其中 $-I:=\{-i : i\in I\}$，则对任意与 p 互质的整数 n，

$$\left(\frac{n}{p}\right)=(-1)^{|J|},$$

其中 $J=\{j\in I : nj\in -I\}$.

不难发现取 $I=\{1,2,\ldots,(p-1)/2\}$ 即可得到 Gauss 引理．证明方法和 Gauss 引理的证明基本相同，故省略．

容易得到如下推论：

推论

对奇素数 p，有

$$\left(\frac{2}{p}\right)=(-1)^{\frac{p^2-1}{8}}=\begin{cases}1, & p\equiv\pm1\pmod 8,\\ -1, & p\equiv\pm3\pmod 8.\end{cases}$$

对奇素数 p，奇数 n 满足 $(n,p)=1$，有

$$\left(\frac{n}{p}\right)=(-1)^{\sum_{i=1}^{(p-1)/2}\lfloor ni/p\rfloor}.$$

[26] Kobi Kremnizer.Lectures in number theory 2022[3]. Proposition 4.3.

证明

对 Gauss 引理中的 $n, k, r_k, A, B, \lambda, \mu$，有 $nk = p\left\lfloor \dfrac{nk}{p} \right\rfloor + r_k$，进而

$$\begin{aligned} n \cdot \frac{p^2-1}{8} = \sum_{k=1}^{\frac{p-1}{2}} nk &= p\sum_{k=1}^{\frac{p-1}{2}} \left\lfloor \frac{nk}{p} \right\rfloor + \sum_{a\in A} a + \sum_{b\in B} b \\ &= p\sum_{k=1}^{\frac{p-1}{2}} \left\lfloor \frac{nk}{p} \right\rfloor + \sum_{a\in A} a + \sum_{b\in B} (p-b) + 2\sum_{b\in B} b - p\mu \\ &= p\sum_{k=1}^{\frac{p-1}{2}} \left\lfloor \frac{nk}{p} \right\rfloor + \sum_{k=1}^{\frac{p-1}{2}} k + 2\sum_{b\in B} b - p\mu \\ &= p\sum_{k=1}^{\frac{p-1}{2}} \left\lfloor \frac{nk}{p} \right\rfloor + \frac{p^2-1}{8} + 2\sum_{b\in B} b - p\mu, \end{aligned}$$

因此

$$(n-1)\frac{p^2-1}{8} = p\sum_{k=1}^{\frac{p-1}{2}} \left\lfloor \frac{nk}{p} \right\rfloor + 2\sum_{b\in B} b - p\mu.$$

若 $n = 2$，则 $0 < \dfrac{nk}{p} \le \dfrac{p-1}{p} < 1$，从而有

$$\frac{p^2-1}{8} \equiv \mu \pmod 2.$$

若 $2 \nmid n$，则有

$$\sum_{k=1}^{\frac{p-1}{2}} \left\lfloor \frac{nk}{p} \right\rfloor \equiv \mu \pmod 2.$$

根据如上推论，证明二次互反律只需验证

$$\frac{p-1}{2}\frac{q-1}{2} = \sum_{k=1}^{\frac{p-1}{2}} \left\lfloor \frac{qk}{p} \right\rfloor + \sum_{k=1}^{\frac{q-1}{2}} \left\lfloor \frac{pk}{q} \right\rfloor.$$

考虑由点 (px, qy)，$1 \le x \le \dfrac{q-1}{2}, 1 \le y \le \dfrac{p-1}{2}$ 构成的集合 S，将其根据 px 与 qy 的大小关系分成两部分（显然 $px \ne qy$），分别验证三个集合的大小即可.

二次互反律不仅能用于判断数 n 是否是模 p 的二次剩余，还能用于确定使数 n 为二次剩余的模数的结构.

Example

- 使得 5 为二次剩余的奇素数 p 满足 $p \equiv \pm 1 \pmod 5$.
- 使得 -3 为二次剩余的奇素数 p 满足 $p \equiv 1 \pmod 3$.
- 使得 -2，3 同时为二次剩余的奇素数 p 满足 $p \equiv 11 \pmod{24}$.

另外，我们还可以证明诸如「形如 $4k+1$ 的素数有无限多个」之类的结论，这一类结论实际上是 Dirichlet 定理[4] 的简单推论.

Jacobi 符号

根据二次互反律，我们可以很自然地想到一种推广 Legendre 符号的方法：

Jacobi 符号

对 **正奇数** $m=p_1^{\alpha_1}...p_k^{\alpha_k}$ 和整数 a，其中 $p_1,...,p_k$ 是素数，$\alpha_1,...,\alpha_k$ 是正整数，定义 Jacobi 符号如下：

$$\left(\frac{a}{m}\right):=\prod_{i=1}^{k}\left(\frac{a}{p_i}\right)^{\alpha_i}.$$

其中等式右端的 $\left(\frac{a}{p_i}\right)$ 为 Legendre 符号→9.11@p1160．另外对整数 a 有 $\left(\frac{a}{1}\right)=1$.

Warning

我们一般不区分 Legendre 符号和 Jacobi 符号，因为由完全积性可知 Jacobi 符号具有和 Legendre 符号一样的性质，所以这两种符号的计算方法是一致的．但是有一点需要注意：当 m **不是奇素数** 时，$\left(\frac{a}{m}\right)$ 的值与 a 是否是模 m 的二次剩余 **无关**，但是若 $\left(\frac{a}{m}\right)=-1$，则说明 m 至少存在一个（实际上是奇数个）素因子 p 使得 a 是模 p 的二次非剩余，从而此时 a 是模 m 的二次非剩余.

我们还可以把模数进一步推广为 **整数**（只需补充 $\left(\frac{a}{-1}\right)$、$\left(\frac{a}{0}\right)$ 和 $\left(\frac{a}{2}\right)$ 的定义），这样就得到了 Kronecker 符号[5].

模意义下开平方

本节讨论模意义下开平方的算法．特别地，本节主要介绍素数模的情形．对于一般模数的情形，可以参考 模意义下开高次方→9.11.20@p1291 的讨论.

特殊情况时的算法

对于同余方程 $x^2\equiv a\pmod p$，其中 p 为奇素数且 a 为二次剩余在 $p\bmod 4=3$ 时有更简单的解法，考虑

$$\begin{aligned}\left(a^{(p+1)/4}\right)^2&\equiv a^{(p+1)/2}&&\pmod p\\&\equiv x^{p+1}&&\pmod p\\&\equiv (x^2)(x^{p-1})&&\pmod p\\&\equiv x^2&&\pmod p\quad(\because\text{Fermat's little theorem})\end{aligned}$$

那么 $a^{(p+1)/4}\bmod p$ 为一个解.

Atkin 算法

仍然考虑上述同余方程，此时 $p\bmod 8=5$，那么令 $b\equiv(2a)^{(p-5)/8}\pmod p$ 和 $\mathrm{i}\equiv 2ab^2\pmod p$ 那么此时 $\mathrm{i}^2\equiv -1\pmod p$ 且 $ab(\mathrm{i}-1)\bmod p$ 为一个解.

证明

$$\begin{aligned}\mathrm{i}^2&\equiv\left(2ab^2\right)^2&&\pmod p\\&\equiv\left(2a\cdot(2a)^{(p-5)/4}\right)^2&&\pmod p\\&\equiv\left((2a)^{(p-1)/4}\right)^2&&\pmod p\\&\equiv(2a)^{\frac{p-1}{2}}&&\pmod p\\&\equiv -1&&\pmod p\end{aligned}$$

那么

$$\begin{aligned}(ab(\mathrm{i}-1))^2 &\equiv a^2\cdot(2a)^{(p-5)/4}\cdot(-2\mathrm{i}) &\pmod p\\ &\equiv a\cdot(-\mathrm{i})\cdot(2a)^{(p-1)/4} &\pmod p\\ &\equiv a &\pmod p\end{aligned}$$

Cipolla 算法

Cipolla 算法用于求解同余方程 $y^2\equiv a\pmod p$，其中 p 为奇素数且 a 为二次剩余.

本节考虑 $\mathbf{F}_p[x]/(x^2-g)$ 中的运算，其中 $g\in\mathbf{F}_p$.

计算方法

不熟悉 多项式环→9.16.3@p1830 的读者，可以简单理解为该集合的元素都具有形式 a_0+a_1x 且 $a_0,a_1\in\mathbf{F}_p$，且遵循如下运算法则：

$$(a_0+a_1x)+(b_0+b_1x)\equiv(a_0+b_0)+(a_1+b_1)x \pmod{(x^2-g)}$$

$$(a_0+a_1x)(b_0+b_1x)\equiv(a_0b_0+a_1b_1g)+(a_1b_0+a_0b_1)x \pmod{(x^2-g)}$$

需要注意的是，此处的 x 并不是一个具体的数，而是表示多项式中的形式记号. 运算中一个关键的点在于利用 $x^2\equiv g\pmod{(x^2-g)}$ 将二次项转化为一次项和常数项. 另外，所有整数运算都需要对 p 取模.

关于该结构的更多内容，请参见 多项式→9.12.1@p1519 和 域论→9.16.4@p1849 等页面.

该算法的第一步为找到一个 r 使得 r^2-a 为二次非剩余. 当然对于 $a\equiv0\pmod p$ 不可能找到这样的 r，需要进行特判. 下文只讨论 $a\not\equiv0\pmod p$ 的情况. 此时可随机一个 r 然后判断，期望可以 2 步找到. 于是，$(r-x)^{\frac{p+1}{2}}\bmod(x^2-(r^2-a))$ 为一个解，可以通过快速幂求值.

为什么期望只需要两步

考虑 r^2-a 为二次剩余的情况，则存在一个 x 使得 $r^2-a\equiv x^2\pmod p$，移项可得 $(r+x)(r-x)\equiv a\pmod p$，不难发现每一个 $(r+x)\in[1,p-1]$，都一一对应于一组 (r,x) 的解，所以使原方程成立的解一共有 $p-1$ 组. 我们分类讨论 $x\equiv0$ 和 $x\not\equiv0$ 两种情况. 对于 $x\equiv0$，由于 a 是二次剩余，对应了 2 种 r 的取值；对于 $x\not\equiv0$，有 $p-1-2$ 种情况，每一个 r 对应其中两种，一共有 $\dfrac{p-3}{2}$ 种 r 的取值. 综上，一共有 $2+\dfrac{p-3}{2}=\dfrac{p+1}{2}$ 种情况使得 r^2-a 为二次剩余，所以每随机一次得到二次非剩余的概率就是 $\dfrac{p-1}{2p}$，期望步数为 $\dfrac{2p}{p-1}\approx2$.

证明

为了方便，首先令 $f(x)=x^2-(r^2-a)\in\mathbf{F}_p[x]$.

需要证明的是，$(r-x)^{\frac{p+1}{2}}\bmod f(x)$ 是原式的解，并且它属于 $\mathbf{F}_p$. 首先考虑证明前者，即证明 $(r-x)^{p+1}\equiv a\pmod{f(x)}$. 为此，我们需要先证明两个引理：

引理 1： $x^p\equiv-x\pmod{f(x)}$

证明：

$$\begin{aligned}x^p &= x(x^2)^{\frac{p-1}{2}}\\ &\equiv x(r^2-a)^{\frac{p-1}{2}} &\pmod{f(x)} &\quad(\because x^2\equiv r^2-a\pmod{f(x)})\\ &\equiv -x &\pmod{f(x)} &\quad(\because r^2-a\text{ is quadratic non-residue})\end{aligned}$$

引理 2： $(a+b)^p\equiv a^p+b^p\pmod p$

使用二项式定理容易发现，除了第一项和最后一项，分子上的 p 无法消掉，于是只剩下 $a^p + b^p$.

$$\begin{aligned}(a+b)^p &= \sum_{i=0}^{p} \binom{p}{i} a^i b^{p-i} \\ &= \sum_{i=0}^{p} \frac{p!}{i!(p-i)!} a^i b^{p-i} \\ &\equiv a^p + b^p \pmod p\end{aligned}$$

有了这两个引理，我们再来考虑证明原式：

$$\begin{aligned}(r-x)^{p+1} &= (r-x)^p (r-x) \\ &\equiv (r^p - x^p)(r-x) \pmod{f(x)} \\ &\equiv (r+x)(r-x) \pmod{f(x)} \\ &= r^2 - x^2 \\ &\equiv r^2 - (r^2 - a) \pmod{f(x)} \\ &= a\end{aligned}$$

下面通过反证法证明我们求出的解属于 $\mathbf{F}_p$，即其 x 的系数为 0.

假设存在一个 $(a_0 + a_1 x)^2 \equiv a \pmod{f(x)}$ 满足 $a_1 \not\equiv 0 \pmod p$，即 $a_0^2 + 2a_0 a_1 x + a_1^2 x^2 \equiv a \pmod{f(x)}$，移项并化简可得：

$$a_0^2 + a_1^2 (r^2 - a) - a \equiv -2a_0 a_1 x \pmod{f(x)}$$

式子左边的 x 的系数为 0，所以右边 x 的系数也为 0，即 $a_0 a_1 \equiv 0 \pmod p$，由于我们令 $a_1 \not\equiv 0 \pmod p$，所以一定有 $a_0 \equiv 0 \pmod p$，于是 $(a_1 x)^2 \equiv a \pmod{f(x)}$ 即 $r^2 - a \equiv a a_1^{-2} \pmod p$.

由于 a 和 a_1^{-2} 都是二次剩余，由 Legendre 符号的积性可知 $a a_1^{-2}$ 也是二次剩余，这与 $r^2 - a$ 是二次非剩余矛盾. 于是原式不存在一个解使得 x 的系数非 0，我们求出的解的 x 的系数也必定为 0.

模板题 洛谷 P5491【模板】二次剩余[6]

```cpp
#include <iostream>
#include <random>

long long p, v;  // 分别是模数和 r^2 - a 的值

struct Poly {
  long long a, b;

  Poly(long long _a = 0, long long _b = 0) : a(_a), b(_b) {}
};

Poly operator*(const Poly& x, const Poly& y) {
  // 重载乘法，可以参考上面有关运算性质的说明
  return Poly((x.a * y.a + v * (x.b * y.b % p)) % p,
              (x.a * y.b + x.b * y.a) % p);
}

// 多项式的快速幂，用于计算答案
Poly modpow(Poly a, long long b) {
  Poly res(1, 0);
```

```
  while (b) {
    if (b & 1) res = res * a;
    a = a * a;
    b >>= 1;
  }
  return res;
}

// 普通的快速幂，用于判断二次非剩余
long long modpow(long long a, long long b) {
  long long res = 1;
  while (b) {
    if (b & 1) res = res * a % p;
    a = a * a % p;
    b >>= 1;
  }
  return res;
}

// 用于生成随机数
std::mt19937 rng(std::random_device{}());

long long cipolla(long long a, long long _p) {
  p = _p;
  if (a == 0)
    return 0;  // 特判一下 0 的情况
  else if (modpow(a, (p - 1) / 2) == p - 1)
    return -1;  // 判断二次非剩余，此时无解
  else {
    // 随机 r，使得 r^2 - a 是一个二次非剩余
    long long r;
    for (r = rng() % p;; r = rng() % p) {
      if (modpow((r * r - a + p) % p, (p - 1) / 2) == p - 1) break;
    }
    v = (r * r - a + p) % p;
    return modpow(Poly(r, 1), (p + 1) / 2).a;  // 根据结论式计算结果
  }
}

int main() {
  int t, a, p;
  std::cin >> t;
  while (t--) {
    std::cin >> a >> p;
    int ans = cipolla(a, p);
    if (ans == -1)
      std::cout << "Hola!" << std::endl;
    else if (ans == 0)
      std::cout << 0 << std::endl;
    else {
      // 相反数是另一个解
      int ans2 = (p - ans) % p;
```

```
73        if (ans2 < ans) std::swap(ans, ans2);
74        std::cout << ans << " " << ans2 << std::endl;
75      }
76    }
77    return 0;
78  }
79
```

Bostan–Mori 算法

该算法基于 Cipolla 算法，我们将问题转换为 常系数齐次线性递推→9.12.10@p1572 再应用 Bostan–Mori 算法. 考虑另一种常见的 Cipolla 算法的描述: $b = x^{(p+1)/2} \bmod (x^2 - tx + a)$ 为满足 $b^2 \equiv a \pmod p$ 的一个解 [27]，其中 $x^2 - tx + a \in \mathbf{F}_p[x]$ 为不可约多项式. 系数 t 的选取同样使用随机方法. 证明过程略. 参考 Bostan 和 Mori 论文 [28] 中的算法我们可以发现问题可转化为求解形式幂级数的乘法逆元的某一项系数:

$$b = [x^{(p+1)/2}] \frac{1}{1 - tx + ax^2}$$

且

$$[x^n] \frac{k_0 + k_1 x}{1 + k_2 x + k_3 x^2} = \begin{cases} [x^{(n-1)/2}] \dfrac{k_1 - k_0 k_2 + k_1 k_3 x}{1 + (2k_3 - k_2^2)x + k_3^2 x^2}, & \text{if } n \bmod 2 = 1 \\ [x^{n/2}] \dfrac{k_0 + (k_0 k_3 - k_1 k_2)x}{1 + (2k_3 - k_2^2)x + k_3^2 x^2}, & \text{else if } n \neq 0 \end{cases}$$

而 $n = 0$ 时显然有 $[x^0] \dfrac{k_0 + k_1 x}{1 + k_2 x + k_3 x^2} = k_0$，该算法乘法次数相较于 Cipolla 算法更少. 其他相关乘法次数较少的算法可见 Müller 的文章 [29].

Legendre 算法

对于同余方程 $x^2 \equiv a \pmod p$，其中 p 为奇素数且 a 为二次剩余. Legendre 算法可描述为找到 r 满足 $r^2 - a$ 为二次非剩余，令 $a_0 + a_1 x = (r - x)^{\frac{p-1}{2}} \bmod (x^2 - a)$，那么 $a_0 \equiv 0 \pmod p$ 且 $a_1^{-2} \equiv a \pmod p$.

证明

考虑选择一个 b 满足 $b^2 \equiv a \pmod p$，那么 $(r - b)(r + b) = r^2 - a$ 为二次非剩余，所以

$$(r - b)^{\frac{p-1}{2}} (r + b)^{\frac{p-1}{2}} \equiv -1 \pmod p$$

存在环态射

$$\begin{aligned} \phi : \mathbf{F}_p[x]/(x^2 - a) &\to \mathbf{F}_p \times \mathbf{F}_p \\ x &\mapsto (b, -b) \end{aligned}$$

那么

$$\begin{aligned} (a_0 + a_1 b, a_0 - a_1 b) &= \phi(a_0 + a_1 x) \\ &= \phi(r - x)^{\frac{p-1}{2}} \\ &= ((r - b)^{\frac{p-1}{2}}, (r + b)^{\frac{p-1}{2}}) \\ &= (\pm 1, \mp 1) \end{aligned}$$

所以 $2a_0 = (\pm 1) + (\mp 1) = 0$ 而 $2a_1 b = (\pm 1) - (\mp 1) = \pm 2$.

---

[27] A. Menezes, P. van Oorschot and S. Vanstone. Handbook of Applied Cryptography, 1996.

[28] Alin Bostan, Ryuhei Mori. A Simple and Fast Algorithm for Computing the N-th Term of a Linearly Recurrent Sequence. Available at https://arxiv.org/abs/2008.08822[7].

[29] S. Müller, On the computation of square roots in finite fields, Design, Codes and Cryptography, Vol.31, pp. 301-312, 2004.

Tonelli–Shanks 算法

Tonelli–Shanks 算法是基于离散对数求解同余方程 $x^2 \equiv a \pmod p$ 的算法[30]，其中 p 为奇素数且 a 为模 p 的二次剩余.

令 $p-1=m2^n$，其中 m 为奇数．仍然使用随机方法寻找 $r \in \mathbf{F}_p$ 满足 r 为二次非剩余．令 $g \equiv r^m \pmod p$ 且 $b \equiv a^{(m-1)/2} \pmod p$，那么存在整数 $e \in \{0,1,2,...,2^n-1\}$ 满足 $ab^2 \equiv g^e \pmod p$．若 a 为二次剩余，那么 e 为偶数且 $\left(abg^{-e/2}\right)^2 \equiv a \pmod p$.

证明

根据费马小定理可知

$$g^{2^n} \equiv r^{m2^n} = r^{p-1} \equiv 1 \pmod p.$$

又由于 r 是二次非剩余，有

$$g^{2^{n-1}} \equiv r^{m2^{n-1}} = r^{\frac{p-1}{2}} \equiv -1 \pmod p.$$

所以，g 模 p 的阶是 2^n．又因为 $ab^2 \equiv a^m \pmod p$ 是 $x^{2^n} \equiv 1 \pmod p$ 的解，所以 a^m 是 g 的幂次．记 $a^m \equiv g^e \pmod p$.

因为 a 是二次剩余，所以

$$g^{e2^{n-1}} \equiv a^{m2^{n-1}} = a^{\frac{p-1}{2}} \equiv 1 \pmod p.$$

由阶的性质可知，$2^n \mid e2^{n-1}$，所以，e 是偶数．因此，$abg^{-e/2} \bmod p$ 是良定义的，且

$$\left(abg^{-e/2}\right)^2 = a^2b^2g^{-e} \equiv a^{m+1}g^{-e} \equiv a \pmod p.$$

剩下的问题是如何计算 e．Tonelli 和 Shanks 提出一次确定 e 的一个二进制位．令 e 在二进制下表示为 $e=e_0+2e_1+4e_2+\cdots$，其中 $e_k \in \{0,1\}$．因为 a 是二次剩余，所以开始时 $e_0=0$，然后利用如下性质逐位确定 e_k 的取值:

$$\left(g^e g^{-(e \bmod 2^k)}\right)^{2^{n-1-k}} \equiv g^{2^{n-1} \cdot e_k} \equiv \begin{cases} 1 \pmod p, & \text{if } e_k = 0 \\ -1 \pmod p, & \text{if } e_k = 1 \end{cases}$$

其中，$g^e \equiv ab^2 \pmod p$ 已知，而 $e \bmod 2^k$ 的取值可以由之前的数位 $e_0, e_1, \cdots, e_{k-1}$ 计算得到．当然，实现算法时，只需要直接维护乘积 $g^e g^{-(e \bmod 2^k)} \bmod p$ 即可.

习题

- 洛谷 P5491【模板】二次剩余[8]
- 「Timus 1132」Square Root[9]

参考资料与注释

1. Quadratic residue - Wikipedia[10]
2. Euler's criterion - Wikipedia[11]

[1] Wikipedia

[2] Proofs of quadratic reciprocity - Wikipedia

[3] Lectures in number theory 2022

[4] Dirichlet 定理

[5] Kronecker 符号

[6] 洛谷 P5491【模板】二次剩余

[7] https://arxiv.org/abs/2008.08822

[8] 洛谷 P5491【模板】二次剩余

[9] 「Timus 1132」Square Root

[10] Quadratic residue - Wikipedia

[11] Euler's criterion - Wikipedia

[30] Daniel. J. Bernstein. Faster Square Roots in Annoying Finite Fields.

9.11.18 阶 & 原根

前置知识：费马小定理→9.11.9@p1223、欧拉定理→9.11.9@p1223、拉格朗日定理→9.11.16@p1258

阶和原根，是理解模 m 既约剩余系→9.11.1@p1160 $\mathbf{Z}_m^*$ 乘法结构的重要工具．基于此，可以定义 离散对数→9.11.19@p1286 等概念．更为一般的讨论可以参见抽象代数部分 群论→9.16.2@p1812 和 环论→9.16.3@p1830 等页面相关章节．

阶

本节中，总是假设模数 $m \in \mathbf{N}_+$ 和底数 $a \in \mathbf{Z}$ 互素，即 $(a, m) = 1$，也记作 $a \perp m$.

对于 $n \in \mathbf{Z}$，幂次 $a^n \bmod m$ 呈现一种循环结构．这个循环节的最小长度，就是 a 模 m 的阶．阶就定义为幂 $a^n \bmod m$ 第一次回到起点 $a^0 \bmod m = 1$ 时的指数：

阶

对于 $a \in \mathbf{Z}, m \in \mathbf{N}_+$ 且 $a \perp m$，满足同余式 $a^n \equiv 1 \pmod m$ 的最小正整数 n 称作 a **模** m **的阶**（the order of a modulo m），记作 $\delta_m(a)$ 或 $\operatorname{ord}_m(a)$.

注

在 抽象代数→9.16.2@p1812 中，这里的「阶」就是模 m 既约剩余系关于乘法形成的群中，元素 a 的阶．用记号 δ 表示阶只适用于这个特殊的群．下面的诸多性质可以直接推广到抽象代数中群元素的阶的性质．

另外还有「半阶」的概念，在数论中会用 δ^- 记号表示．它是满足同余式 $a^n \equiv -1 \pmod m$ 的最小正整数．半阶不是群论中的概念．阶一定存在，半阶不一定存在．

幂的循环结构

利用阶，可以刻画幂的循环结构．对于幂 $a^n \bmod m$，可以将指数 n 对阶 $\delta_m(a)$ 做带余除法：

$$n = \delta_m(a)q + r,\ 0 \le r < \delta_m(a).$$

进而，利用幂的运算律，就得到

$$a^n = a^{\delta_m(a)q+r} = (a^{\delta_m(a)})^q \cdot a^r \equiv a^r \pmod m.$$

这说明，对于任意指数的幂，可以将它平移到第一个非负的循环节．由此，可以得到一系列关于阶的性质．

性质 1

对于 $a \in \mathbf{Z}, m \in \mathbf{N}_+$ 且 $a \perp m$，幂次 $a^0(=1), a, a^2, \cdots, a^{\delta_m(a)-1}$ 模 m 两两不同余.

证明

考虑反证．假设存在两个数 $0 \le i < j < \delta_m(a)$，且 $a^i \equiv a^j \pmod m$，则有 $a^{j-i} \equiv 1 \pmod m$．但是，$0 < j - i < \delta_m(a)$．这与阶的最小性矛盾，故原命题成立.

性质 2

对于 $a, n \in \mathbf{Z}, m \in \mathbf{N}_+$ 且 $a \perp m$，同余关系 $a^n \equiv 1 \pmod m$ 成立，当且仅当 $\delta_m(a) \mid n$.

证明

如前文所述，$a^n \equiv a^{n \bmod \delta_m(a)} \pmod m$. 由性质 1 可知，$0 \le r < \delta_m(a)$ 中唯一一个使得 $a^r \equiv 1 \pmod m$ 成立的 r 就是 $r = 0$. 因此，$a^n \equiv 1 \pmod m$，当且仅当 $n \bmod \delta_m(a) = 0$，也就是 $\delta_m(a) \mid n$.

欧拉定理→9.11.9@p1223 中，同余关系 $a^{\varphi(m)} \equiv 1 \pmod m$ 对于所有 $a \perp m$ 都成立. 结合性质 2，这说明对于所有 $a \perp m$，都有 $\delta_m(a) \mid \varphi(m)$. 换句话说，$\varphi(m)$ 是所有 $a \perp m$ 的阶的一个公倍数. 对于一个正整数 m，所有 $a \perp m$ 的阶 $\delta_m(a)$ 的最小公倍数，记作 $\lambda(m)$，就是 m 的 Carmichael 函数→9.11@p1160. 后文会详细讨论它的性质.

和其他的循环结构类似，可以根据 a 的阶计算 a^k 的阶.

性质 3

对于 $k, a \in \mathbf{Z}, m \in \mathbf{N}_+$ 且 $a \perp m$，有

$$\delta_m(a^k) = \frac{\delta_m(a)}{(\delta_m(a), k)}.$$

证明

由性质 2，同余关系 $(a^k)^n = a^{kn} \equiv 1 \pmod m$ 成立，当且仅当 $\delta_m(a) \mid kn$. 这一条件就等价于

$$\frac{\delta_m(a)}{(\delta_m(a), k)} \mid n.$$

使得这一条件成立的最小正整数就是

$$\delta_m(a^k) = \frac{\delta_m(a)}{(\delta_m(a), k)}.$$

乘积的阶

设 a, b 是与 m 互素的不同整数. 如果已知阶 $\delta_m(a)$ 和 $\delta_m(b)$，那么，同样可以获得一些关于它们乘积 ab 的阶 $\delta_m(ab)$ 的信息.

性质 4

对于 $a, b \in \mathbf{Z}, m \in \mathbf{N}_+$ 且 $a, b \perp m$，有

$$\frac{[\delta_m(a), \delta_m(b)]}{(\delta_m(a), \delta_m(b))} \mid \delta_m(ab) \mid [\delta_m(a), \delta_m(b)].$$

证明

因为 $[\delta_m(a), \delta_m(b)]$ 是 $\delta_m(a)$ 和 $\delta_m(b)$ 的倍数，所以，由性质 2 可知

$$(ab)^{[\delta_m(a), \delta_m(b)]} = a^{[\delta_m(a), \delta_m(b)]} b^{[\delta_m(a), \delta_m(b)]} \equiv 1 \pmod m.$$

再次应用性质 2，就得到

$$\delta_m(ab) \mid [\delta_m(a), \delta_m(b)].$$

这就得到右侧的整除关系.

反过来，由于

$$1 \equiv (ab)^{\delta_m(ab)\delta_m(b)} \equiv a^{\delta_m(ab)\delta_m(b)} \pmod m,$$

所以，应用性质 2，就得到 $\delta_m(a) \mid \delta_m(ab)\delta_m(b)$. 两侧消去 $(\delta_m(a), \delta_m(b))$，就得到

$$\frac{\delta_m(a)}{(\delta_m(a),\delta_m(b))} \mid \delta_m(ab)\frac{\delta_m(b)}{(\delta_m(a),\delta_m(b))}.$$

消去公因子后，两个分式互素，这就得到

$$\frac{\delta_m(a)}{(\delta_m(a),\delta_m(b))} \mid \delta_m(ab).$$

同理，也有

$$\frac{\delta_m(b)}{(\delta_m(a),\delta_m(b))} \mid \delta_m(ab).$$

由于两个整除关系的左侧互素，有

$$\frac{[\delta_m(a),\delta_m(b)]}{(\delta_m(a),\delta_m(b))} = \frac{\delta_m(a)\delta_m(b)}{(\delta_m(a),\delta_m(b))^2} \mid \delta_m(ab).$$

这就得到左侧的整除关系.

对于 a 和 b 的阶互素的情形，这一结论有着更为简单的形式.

性质 4′

对于 $a,b\in\mathbf{Z},m\in\mathbf{N}_+$ 且 $a,b\perp m$，有

$$\delta_m(ab)=\delta_m(a)\delta_m(b)\iff \delta_m(a)\perp\delta_m(b).$$

证明

如果 $\delta_m(a)\perp\delta_m(b)$，那么性质 4 中所有整除关系都是等式，所以有

$$\delta_m(ab)=[\delta_m(a),\delta_m(b)]=\delta_m(a)\delta_m(b).$$

反过来，如果 $\delta_m(ab)=\delta_m(a)\delta_m(b)$，那么根据性质 4，就有

$$\delta_m(a)\delta_m(b)=\delta_m(ab)\mid[\delta_m(a),\delta_m(b)].$$

这立马说明 $(\delta_m(a),\delta_m(b))=1$，即 $\delta_m(a)\perp\delta_m(b)$.

一般情形中，性质 4 得到的界已经是紧的. 乘积的阶取得下界的情形很容易构造：例如 $(a,b,m)=(3,5,7)$ 时，$\delta_m(a)=\delta_m(b)=6$，但是它们的乘积的阶 $\delta_m(ab)=1$.

尽管一般情形中，乘积 ab 的阶未必是它们的阶的最小公倍数，但是总能找到一个元素使得它的阶等于这个最小公倍数.

性质 5

对于 $a,b\in\mathbf{Z},m\in\mathbf{N}_+$ 且 $a,b\perp m$，总是存在 $c\in\mathbf{Z}$ 且 $c\perp m$ 使得

$$\delta_m(c)=[\delta_m(a),\delta_m(b)].$$

证明

考虑素因数分解：

$$\delta_m(a)=\prod_p p^{\alpha_p},\ \delta_m(b)=\prod_p p^{\beta_p}.$$

利用 α_p 和 β_p 的大小关系，可以将所有素因子分为两类：

$$A=\{p:\alpha_p\geq\beta_p\},\ B=\{p:\alpha_p<\beta_p\}.$$

由此，分别设

$$\gamma_A=\prod_{p\in A}p^{\alpha_p},\ \gamma_B=\prod_{p\in B}p^{\alpha_p},\ \eta_A=\prod_{p\in A}p^{\beta_p},\ \eta_B=\prod_{p\in B}p^{\beta_p},$$

就有 $\delta_m(a)=\gamma_A\gamma_B$ 和 $\delta_m(b)=\eta_A\eta_B$. 根据性质 3，可知

$$\delta_m(a^{\gamma_B})=\frac{\delta_m(a)}{(\delta_m(a),\gamma_B)}=\frac{\delta_m(a)}{\gamma_B}=\gamma_A,$$

$$\delta_m(b^{\eta_A})=\frac{\delta_m(b)}{(\delta_m(b),\eta_A)}=\frac{\delta_m(b)}{\eta_A}=\eta_B.$$

因为 $\gamma_A\perp\eta_B$，由性质 4′，就有

$$\delta_m(a^{\gamma_B}b^{\eta_A})=\gamma_A\eta_B=\prod_p p^{\max\{\alpha_p,\beta_p\}}=[\delta_m(a),\delta_m(b)].$$

因此，$c=a^{\gamma_B}b^{\eta_A}$ 就是阶为 $[\delta_m(a),\delta_m(b)]$ 的元素.

这一结论常用于构造出指定阶的元素.

原根

原根是一些特殊元素——它的阶就等于模 m 既约剩余系中元素的个数.

自 原根

对于 $m\in\mathbf{N}_+$，如果存在 $g\in\mathbf{Z}$ 且 $g\perp m$ 使得 $\delta_m(g)=|\mathbf{Z}_m^*|=\varphi(m)$，就称 g 为 **模 m 的原根**（primitive root modulo m）. 其中，$\varphi(m)$ 是 欧拉函数→9.11.5@p1196.

并非所有正整数 m 都存在模 m 的原根. 由上文的 性质 1→9.11@p1160，如果模 m 的原根 g 存在，那么，$g,g^2,\cdots,g^{\varphi(m)}$ 所在的同余类互不相同，构成模 m 既约剩余系. 特别地，对于素数 p，余数 $g^i \bmod p$ 对于 $i=1,2,\cdots,p-1$ 两两不同.

注

在 抽象代数→9.16.3@p1830 中，原根就是循环群的生成元. 这个概念只在模 m 既约剩余系关于乘法形成的群中有「原根」这个名字，在一般的循环群中都称作「生成元」. 并非每个模 m 既约剩余系关于乘法形成的群都是循环群，存在原根就表明它同构于循环群，如果不存在原根就表明不同构.

模为 1 时，模 1 整数乘法群就是 $\{0\}$. 这显然是循环群，所以原根就是 0.

原根判定定理

如果已知 $\varphi(m)$ 的全体素因子，那么很容易判断给定的 g 是否为模 m 的原根.

定理

对于整数 $m\geq 3$ 和 $g\perp m$，g 是模 m 的原根，当且仅当对于 $\varphi(m)$ 的每个素因数 p，都有

$$g^{\frac{\varphi(m)}{p}}\not\equiv 1\pmod m.$$

证明

必要性显然. 为证明充分性，考虑使用反证法. 如果 g 不是模 m 的原根，那么一定有 $\delta_m(g)<\varphi(m)$. 由 性质 2→9.11@p1160 和欧拉定理可知，$\delta_m(g)\mid\varphi(m)$. 由此，设 p 是 $\dfrac{\varphi(m)}{\delta_m(g)}$ 的一个素因子，就有 $\delta_m(g)\mid\dfrac{\varphi(m)}{p}$. 再次应用性质 2 就得到

$$g^{\frac{\varphi(m)}{p}}\equiv 1\pmod m.$$

但是，p 也是 $\varphi(m)$ 的一个素因子，这就与题设条件矛盾．由此，原命题的充分性成立．

原根个数

原根如果存在，也未必唯一．一般地，对于模 m 既约剩余系中所有元素可能的阶和某个阶的元素数量，有如下结论：

定理

如果正整数 m 有原根 g，那么，当且仅当 $d \mid \varphi(m)$ 时，模 m 的 d 阶元素存在，且恰有 $\varphi(d)$ 个．特别地，模 m 的原根个数为 $\varphi(\varphi(m))$．

证明

根据原根的定义，所有模 m 的既约同余类都可以写作 $g^k \bmod m$ 的形式，且 k 是 $1, 2, \cdots, \varphi(m)$ 之一．由 性质 3→9.11@p1160，这些元素的阶等于

$$\delta_m(g^k) = \frac{\varphi(m)}{(\varphi(m), k)}.$$

因此，d 阶元素存在，当且仅当 $d \mid \varphi(m)$．而且，对于 $d \mid \varphi(m)$，令 $d' = \varphi(m)/d$，这些元素的集合就是

$$\begin{aligned} A &= \{g^k : (\varphi(m), k) = d',\ 1 \le k \le \varphi(m)\} \\ &= \{g^k : d' \mid k,\ (d, k/d') = 1,\ 1 \le k/d' \le d\}. \end{aligned}$$

这些元素对应的 $k' = k/d'$ 恰为那些不超过 d 且与 d 互素的正整数．由欧拉函数的定义，这就是 $\varphi(d)$．

原根存在定理

本节将建立如下原根存在定理：

定理

模 m 的原根存在，当且仅当 $m = 1, 2, 4, p^e, 2p^e$，其中，p 是奇素数且 $e \in \mathbf{N}_+$．

为说明这一结论，需要分别讨论如下四种情形：

1. $m = 1, 2, 4$，原根分别是 $g = 0, 1, 3$，显然存在．
2. $m = p^e$ 是奇素数的幂，其中，p 为奇素数，$e \in \mathbf{N}_+$．

引理 1

对于奇素数 p，模 p 的原根存在．

证明

证明分为两步．

第一步：对于 $d \mid (p-1)$，同余方程 $x^d \equiv 1 \pmod p$ 恰有 d 个互不相同的解．

令 $p - 1 = kd$，多项式

$$f(x) = x^{d(k-1)} + x^{d(k-2)} + \cdots + x^d + 1.$$

根据 欧拉定理→9.11.9@p1223，同余方程 $(x^d - 1)f(x) = x^{p-1} - 1 \equiv 0 \pmod p$ 恰有 $p - 1$ 个互不相同的解．这些解分别是 $x^d - 1$ 和 $f(x)$ 的零点．由 Lagrange 定理→9.11.16@p1258，它们

分别至多只能有 d 个和 $d(k-1)$ 个互不相同的零点．由于 $d+d(k-1)=p-1$，前者只能恰好有 d 个互不相同的零点．这说明同余方程 $x^d\equiv 1 \pmod p$ 恰有 d 个互不相同的解．

第二步：对于 $d\mid(p-1)$，d 阶元素恰好有 $\varphi(d)$ 个．

将 $p-1$ 的所有因子排序，然后应用归纳法．因为 1 阶元素只能是 1，只有一个，归纳起点成立．对于 $d\mid(p-1)$，根据前文的 性质 2→9.11@p1160，同余方程 $x^d\equiv 1 \pmod p$ 的解一定满足 $\delta_p(x)\mid d$．因此，其中 d 阶元素个数为

$$N(d)=d-\sum_{e\mid d,\ e\neq d}N(e)=d-\sum_{e\mid d,\ e\neq d}\varphi(e)=\varphi(d).$$

第二个等号是归纳假设，第三个等号是欧拉函数的性质．由数学归纳法，就知道对于所有 $d\mid(p-1)$，都恰有 $\varphi(d)$ 个 d 阶元素．

特别地，对于 $d=p-1$，恰有 $\varphi(p-1)$ 个 $(p-1)$ 阶元素．因此，模 p 的原根存在．

引理 2

对于奇素数 p 和 $e\in\mathbf{N}_+$，模 p^e 的原根存在．

证明

证明分为三步．

第一步：存在模 p 的原根 g，使得 $g^{p-1}\not\equiv 1 \pmod{p^2}$．

任取一个模 p 的原根 g．如果它不符合条件，即 $g^{p-1}\equiv 1 \pmod{p^2}$，那么，可以证明 $g+p$ 符合条件：$g+p$ 也是模 p 的原根，且

$$\begin{aligned}(g+p)^{p-1}&\equiv\binom{p-1}{0}g^{p-1}+\binom{p-1}{1}g^{p-2}p\\&=g^{p-1}+g^{p-2}p(p-1)\\&\equiv 1-pg^{p-2}\not\equiv 1 \pmod{p^2}.\end{aligned}$$

第二步：上文选取的 g，对于任意 $e\geq 1$，都有 $g^{\varphi(p^e)}\not\equiv 1 \pmod{p^{e+1}}$．

对 g 的选取保证了 $e=1$ 时，该式成立．假设该式对于 e 的情形成立，现要证明 $e+1$ 的情形也成立．对于任意 $e\geq 1$，由欧拉定理可知，存在整数 t 使得

$$g^{\varphi(p^e)}=1+tp^e$$

成立．由归纳假设，$t\perp p$．因为 $\varphi(p^{e+1})=p\varphi(p^e)$，所以

$$g^{\varphi(p^{e+1})}=\left(g^{\varphi(p^e)}\right)^p=(1+tp^e)^p\equiv 1+tp^{e+1} \pmod{p^{e+2}}.$$

结合 $t\perp p$ 可知，$g^{\varphi(p^{e+1})}\not\equiv 1 \pmod{p^{e+2}}$．由数学归纳法可知，命题成立．

第三步：上文选取的 g，对于任意 $e\geq 1$，都是模 p^e 的原根．

对 g 的选取保证了 $e=1$ 时，命题成立．假设命题对于 e 成立，现在要证明命题对于 $e+1$ 也成立．将 $\delta_{p^{e+1}}(g)$ 简记为 δ．由于 $g^{\delta}\equiv 1 \pmod{p^{e+1}}$，必然也有 $g^{\delta}\equiv 1 \pmod{p^e}$．由归纳假设可知，$\delta_{p^e}(g)=\varphi(p^e)$．因此，由前文阶的 性质 2→9.11@p1160，就有 $\varphi(p^e)\mid\delta$．又由欧拉定理可知，$\delta\mid\varphi(p^{e+1})$．但是，$\varphi(p^{e+1})=p\varphi(p^e)$．因此，只有两种可能：$\delta=\varphi(p^e)$ 或 $\delta=\varphi(p^{e+1})$．但是，第二步的结论说明，$g^{\varphi(p^e)}\not\equiv 1 \pmod{p^{e+1}}$．因此，可能性 $\delta=\varphi(p^e)$ 并不成立．唯一的可能性就是 $\delta=\varphi(p^{e+1})$．这就说明 g 是模 p^{e+1} 的原根．由数学归纳法，命题对于所有 $e\geq 1$ 都成立．

3. $m=2p^e$，其中，p 为奇素数，$e\in\mathbf{N}_+$．

引理 3

对于奇素数 p 和 $e \in \mathbf{N}_+$，模 $2p^e$ 的原根存在.

证明

设 g 是模 p^e 的原根, 则 $g+p^e$ 也是模 p^e 的原根. 两者之间必然有一个是奇数, 不妨设它就是 g. 显然, $(g, 2p^e)=1$. 设 $\delta=\delta_{2p^e}(g)$, 需要证明 $\delta=\varphi(2p^e)$. 由欧拉定理, $\delta \mid \varphi(2p^e)$. 同时, 根据定义 $g^\delta \equiv 1 \pmod{2p^e}$, 所以, $g^\delta \equiv 1 \pmod{p^e}$, 因此, 由阶的 性质 2→9.11@p1160 和 g 的选取可知, $\delta_{p^e}(g)=\varphi(p^e) \mid \delta$. 由欧拉函数表达式可知, $\varphi(2p^e)=\varphi(p^e)$. 所以, $\delta=\delta_{2p^e}(g)=\varphi(p^e)=\varphi(2p^e)$. 这就说明 g 是模 $2p^e$ 的原根.

4. $m \neq 1, 2, 4, p^e, 2p^e$，其中，$p$ 为奇素数，$e \in \mathbf{N}_+$.

引理 4

假设 $m \neq 1, 2, 4$ 且不存在奇素数 p 和正整数 e 使得 $m=p^e$ 或 $m=2p^e$. 那么，模 m 的原根不存在.

证明

对于 $m=2^e$ 且 $e \geq 3$，假设模 m 的原根 g 存在. 由于 $g \perp m$，它一定是奇数. 假设 $g=2k+1$ 且 $k \in \mathbf{N}$，那么，有

$$\begin{aligned} g^{2^{e-2}} &= (2k+1)^{2^{e-2}} \\ &\equiv 1+\binom{2^{e-2}}{1}(2k)+\binom{2^{e-2}}{2}(2k)^2 \\ &= 1+2^{e-1}k+2^{e-1}(2^{e-2}-1)k^2 \\ &= 1+2^{e-1}(k+(2^{e-2}-1)k^2) \\ &\equiv 1 \pmod{2^e}. \end{aligned}$$

倒数第二行中，因为 k 与 $(2^{e-2}-1)k^2$ 奇偶性相同，所以它们的和是偶数. 由阶的定义可知，$\delta_{2^e}(g) \leq 2^{e-2} < \varphi(2^e)=2^{e-1}$. 这与假设中 g 是原根矛盾. 由反证法，这样的原根并不存在.

假设 m 满足所述条件，且不是 2 的幂，那么，一定存在 $2<m_1<m_2$ 且 $m_1 \perp m_2$ 使得 $m=m_1m_2$ 成立. 假设模 m 的原根 g 存在. 因为 $g \perp m$，所以对于 $i=1,2$，都有 $g \perp m_i$. 由欧拉定理可知,

$g^{\varphi(m_i)} \equiv 1 \pmod{m_i}$.

由于 $m_i>2$，所以 $\varphi(m_i)$ 为偶数，所以，对于 $i=1,2$，有

$g^{\frac{1}{2}\varphi(m_1)\varphi(m_2)} \equiv 1 \pmod{m_i}$.

由 中国剩余定理→9.11.12@p1236 可知

$g^{\frac{1}{2}\varphi(m_1)\varphi(m_2)} \equiv 1 \pmod{m}$.

又因为 $\varphi(m)=\varphi(m_1)\varphi(m_2)$，所以由阶的定义可知

$$\delta_m(g) \leq \frac{1}{2}\varphi(m_1)\varphi(m_2)=\frac{1}{2}\varphi(m)<\varphi(m).$$

这与 g 是模 m 的原根的假设矛盾. 故而，由反证法知，模 m 的原根不存在.

综合以上四种情形，我们便给出了一个数存在原根的充要条件.

求原根的算法

对于任何存在原根的模数 m，要求得它的原根 g，只需要枚举可能的正整数，并逐个判断它是否为原根即可．枚举时，通常有两种处理方式：从小到大逐一枚举、随机生成一些正整数．这两种枚举方式的实际效率相当．

从小到大逐一枚举时，得到的是模 m 的最小原根 g_m，因此，枚举部分的复杂度取决于 g_m 的大小．对此，有如下估计：

- 上界的估计：王元[31]和 Burgess[32]证明了素数 p 的最小原根 $g_p = O(p^{0.25+\epsilon})$，其中 $\epsilon > 0$．Cohen, Odoni, and Stothers[33]和 Elliott and Murata[34]分别证明了该估计对于模数 p^2 和 $2p^2$ 也成立，其中，p 是奇素数．由于对于 $e > 2$，模 p^2（或 $2p^2$）的原根也是模 p^e（或 $2p^e$）的原根，所以，最小原根的上界 $O(p^{0.25+\epsilon})$ 对于所有情形都成立．
- 下界的估计：Fridlender[35] 和 Salié[36] 证明了存在 $C > 0$，使得对于无穷多素数 p，都有最小原根 $g_p > C \log p$ 成立．
- 平均情形的估计：Burgess and Elliott[37] 证明了平均情形下素数 p 的最小原根 $g_p = O((\log p)^2(\log\log p)^4)$．Elliott and Murata[38] 进一步猜想素数 p 的最小原根的平均值是一个常数，且通过数值验证[39] 得到它大概为 4.926．随后，Elliott and Murata[40]将这一猜想推广到模 $2p^2$ 的情形．

根据这些分析，暴力寻找最小原根时，枚举部分的复杂度 $O(g_m(\log m)^2)$ 是可以接受的．

除了从小到大枚举外，还可以通过随机生成正整数并验证的方法寻找原根．原根的密度并不低：[41]

$$\frac{\varphi(\varphi(m))}{m} = \Omega\left(\frac{1}{\log\log m}\right).$$

所以，通过随机方法寻找原根时，枚举部分的期望复杂度为 $O((\log m)^2 \log\log m)$．

需要注意的是，判定原根时需要已知 $\varphi(m)$ 的素因数分解．算法竞赛 常用素因数分解算法→9.11.7@p1209 中，复杂度最优的 Pollard Rho 算法也需要 $O(m^{1/4+\epsilon})$ 的时间．因此，只要 $\varphi(m)$ 的素因数分解是未知的，无论采用哪种枚举方式，求原根的复杂度瓶颈都在于素因数分解这一步，而非枚举验证的部分．

Carmichael 函数

相对于模 m 元素的阶这一局部概念，Carmichael 函数是一个全局概念．它是所有与 m 互素的整数的幂次的最小公共循环节．

---

[31] Wang Y. "On the least primitive root of a prime." (in Chinese). Acta Math Sinica, 1959, 4: 432–441; English transl. in *Sci. Sinica*, 1961, 10: 1–14.

[32] BURGESS, David A. "On character sums and primitive roots." Proceedings of the London Mathematical Society, 1962, 3.1: 179-192.

[33] Cohen, S. D., R. W. K. Odoni, and W. W. Stothers. "On the least primitive root modulo p^2." Bulletin of the London Mathematical Society 6, no. 1 (1974): 42-46.

[34] Elliott, P. D. T. A., and L. Murata. "The least primitive root mod $2p^2$." Mathematika 45, no. 2 (1998): 371-379.

[35] FRIDLENDER, V. R. "On the least n-th power non-residue." Dokl. Akad. Nauk SSSR. 1949. p. 351-352.

[36] SALIÉ, Hans. "Über den kleinsten positiven quadratischen Nichtrest nach einer Primzahl." Mathematische Nachrichten, 1949, 3.1: 7-8.

[37] Burgess, D. A., and P. D. T. A. Elliott. "The average of the least primitive root." Mathematika 15, no. 1 (1968): 39-50.

[38] Elliott, Peter DTA, and Leo Murata. "On the average of the least primitive root modulo p." Journal of The London Mathematical Society 56, no. 3 (1997): 435-454.

[39] 更多结果可以参考 Least prime primitive root of prime numbers[1].

[40] Elliott, P. D. T. A., and L. Murata. "The least primitive root mod $2p^2$." Mathematika 45, no. 2 (1998): 371-379.

[41] 如果模 m 的原根存在，那么，$\varphi(m) \geq \frac{1}{3}m$，且等号仅在 $m = 2 \times 3^e$ $(e \in \mathbf{N}_+)$ 处取得．进一步地，当 $m > 2$ 时，对欧拉函数 $\varphi(m)$ 有估计：$\varphi(m) > \frac{m}{e^\gamma \log\log m + \frac{3}{\log\log m}}$．将这两者结合，就得到文中的表达式．关于欧拉函数的该估计，可以参考论文 Rosser, J. Barkley, and Lowell Schoenfeld. "Approximate formulas for some functions of prime numbers." Illinois Journal of Mathematics 6, no. 1 (1962): 64-94.

Carmichael 函数

对于 $m \in \mathbf{N}_+$，定义 $\lambda(m)$ 为能够使得同余关系 $a^n \equiv 1 \pmod m$ 对于所有 $a \perp m$ 都成立的最小正整数 n．函数 $\lambda : \mathbf{N}_+ \to \mathbf{N}_+$ 就称为 **Carmichael 函数**.

根据 性质 2→9.11@p1160，能够使得 $a^n \equiv 1 \pmod m$ 对于所有 $a \perp m$ 都成立，意味着 $\delta_m(a) \mid n$ 对于所有 $a \perp m$ 都成立．也就是说，符合这一条件的正整数 n，一定是全体 $\delta_m(a)$ 的公倍数．因此，最小的这样的 n 就是它们的最小公倍数：

$$\lambda(m) = \operatorname{lcm}\{\delta_m(a) : a \perp m\}.$$

这也常用作 Carmichael 函数的等价定义.

反复应用 性质 5→9.11@p1160 可知，一定存在某个元素 $a \perp m$ 使得 $\delta_m(a) = \lambda(m)$．因此，上式也可以写作

$$\lambda(m) = \max\{\delta_m(a) : a \perp m\}.$$

取得这一最值的元素 $a \perp m$ 也称为模 m 的 **λ-原根**．它对于所有模数 m 都存在.

递推公式

Carmichael 函数是一个 数论函数→9.11.1@p1160．本节讨论它的一个递推公式，并由此给出原根存在定理的另一个证明.

虽然不是积性函数，但是计算 Carmichael 函数时，同样可以对互素的因子分别处理.

引理

对于互素的正整数 m_1, m_2，有 $\lambda(m_1 m_2) = [\lambda(m_1), \lambda(m_2)]$.

证明

设 a_1 和 a_2 分别为模 m_1 和模 m_2 的 λ-原根．令 $m = m_1 m_2$，由 中国剩余定理→9.11.12@p1236 可知，存在 $a \perp m$ 使得 $a \equiv a_i \pmod{m_i}$ 对于 $i = 1, 2$ 都成立．由于 $a^{\lambda(m)} \equiv 1 \pmod m$，所以对于 $i = 1, 2$，都有 $a_i^{\lambda(m)} \equiv 1 \pmod{m_i}$，进而由 性质 2→9.11@p1160 和 a_i 的选取可知，$\lambda(m_i) = \delta_{m_i}(a_i) \mid \lambda(m)$．这就说明 $[\lambda(m_1), \lambda(m_2)] \mid \lambda(m)$.

反过来，对于任意 $a \perp m$ 和 $i = 1, 2$，都有 $a^{[\lambda(m_1), \lambda(m_2)]} \equiv 1 \pmod{m_i}$．应用中国剩余定理，就得到 $a^{[\lambda(m_1), \lambda(m_2)]} \equiv 1 \pmod m$ 对于所有 $a \perp m$ 都成立．根据 Carmichael 函数的定义可知，$\lambda(m) \mid [\lambda(m_1), \lambda(m_2)]$.

由此，命题中的等式成立.

因此，接下来只要计算 Carmichael 函数在素数幂处的取值．首先，处理 2 的幂次的情形.

引理

对于 $m = 2^e$ 且 $e \in \mathbf{N}_+$，有 $\lambda(2) = 1$，$\lambda(4) = 2$，且对于 $e \geq 3$ 都有 $\lambda(m) = 2^{e-2}$.

证明

对于 $m = 2, 4$ 的情形，单独讨论即可．对于 $m = 2^e$ 且 $e \geq 3$ 的情形，首先重复前文 引理 4→9.11@p1160 的证明的第一部分，就得到 $\lambda(m) \leq 2^{e-2}$．进而，只需要证明存在 2^{e-2} 阶元素即可．为此，有

$$5^{2^{e-3}} = (1 + 2^2)^{2^{e-3}} \equiv 1 + 2^2 \times 2^{e-3} = 1 + 2^{e-1} \not\equiv 1 \pmod{2^e}.$$

这说明 $\delta_m(5) \nmid 2^{e-3}$，又因为 $\delta_m(5) \mid 2^{e-2}$，所以，5 只能是 2^{e-2} 阶元素. 这就说明，$\lambda(m) = 2^{e-2}$.

在这个引理的证明过程中，实际上得到了关于模 2^e 既约剩余系结构的刻画：

推论

设模数为 2^e 且 $e \geq 2$. 那么，所有奇数都同余于唯一一个 $\pm 5^k$ 形式的整数，其中，$k \in \mathbf{N}$ 且 $k < 2^{e-2}$. 也就是说，$\pm 1, \pm 5, \cdots, \pm 5^{2^{e-2}-1}$ 两两不同余，且构成一个既约剩余系.

证明

容易验证，$e = 2$ 的情形成立. 对于 $e \geq 3$ 的情形，由于前述证明中已经得到 5 模 2^e 的阶是 2^{e-2}，所以，$1, 5, \cdots, 5^{2^{e-2}-1}$ 两两不同余. 因为这些整数都模 4 余 1，它们的相反数都模 4 余 3，所以 $\pm 1, \pm 5, \cdots, \pm 5^{2^{e-2}-1}$ 模 2^e 两两不同余. 由于它们共计 2^{e-1} 个，恰为模 2^e 的既约剩余系的大小，所以，它们就构成了既约剩余系本身.

然后，处理奇素数幂的情形.

引理

对于 $m = p^e$，其中，p 是奇素数且 $e \in \mathbf{N}_+$，有 $\lambda(m) = p^{e-1}(p-1)$.

证明

首先证明命题对于 $e = 1$，即 $m = p$ 是奇素数的情形成立. 为此，由 Carmichael 函数的定义可知，与 p 互素的所有整数 a 都是同余方程 $x^{\lambda(p)} \equiv 1 \pmod{p}$ 的解. 在模 p 的意义下，该方程共有 $p-1$ 个互不相同的解. 根据 Lagrange 定理→9.11.16@p1258 可知，$p - 1 \leq \lambda(p)$. 同时，欧拉定理要求，$\lambda(p) \mid \varphi(p) = p - 1$. 因此，$\lambda(p) = p - 1$.

对于 $m = p^e$ 且 $e > 1$ 的情形，可以从证明 $1 + p$ 是 p^{e-1} 阶元开始. 为此，有

$$(1+p)^{p^{e-1}} \equiv 1, \quad (1+p)^{p^{e-2}} \equiv 1 + p^{e-1} \not\equiv 1 \pmod{p^e}.$$

所以，$\delta_m(1+p) = p^{e-1}$. 另外，设模 p 的原根为 g，那么，由于 $g^{\delta_m(g)} \equiv 1 \pmod{p}$，所以，由阶的 性质 2→9.11@p1160 可知，$p - 1 \mid \delta_m(g)$. 由 Carmichael 函数的定义和欧拉定理可知

$$p^{e-1}(p-1) \mid [\delta_m(1+p), \delta_m(g)] \mid \lambda(m) \mid \varphi(m) = p^{e-1}(p-1).$$

因此，$\lambda(m) = p^{e-1}(p-1)$.

将本节的结果简单归纳，就得到 Carmichael 函数的递推公式：

定理

对于任意正整数 m，有

$$\lambda(m) = \begin{cases} \varphi(m), & \text{if } m = 1, 2, 4, p^e \text{ for odd prime } p \text{ and } e \geq 1, \\ \frac{1}{2}\varphi(m), & \text{if } m = 2^e,\ e \geq 3, \\ \operatorname{lcm}\{\lambda(p_1^{e_1}), \lambda(p_2^{e_2}), \cdots, \lambda(p_s^{e_s})\}, & \text{if } m = p_1^{e_1} p_2^{e_2} \cdots p_s^{e_s} \text{ for distinct } p_1, p_2, \cdots, p_s. \end{cases}$$

利用该递推公式可以加强前文的结果：

推论

对于正整数 m_1, m_2，有 $\lambda([m_1, m_2]) = [\lambda(m_1), \lambda(m_2)]$.

比较原根和 Carmichael 函数的定义可知，模 m 的原根存在，当且仅当 $\lambda(m)=\varphi(m)$. 从 Carmichael 函数的递推公式中，容易归纳出如下结果：

推论

模 m 的原根存在，当且仅当 $m=1,2,4,p^e,2p^e$，其中，p 是奇素数且 $e\in\mathbf{N}_+$.

由于本节对于递推公式的证明并没有用到原根存在定理，因此，这就构成了对该定理的又一个证明.

Carmichael 数

利用 Carmichael 函数，可以讨论 Carmichael 数（卡迈克尔数，OEIS:A002997[2]）的性质与分布. 这是 Fermat 素性测试→9.11.3@p1179 一定无法正确排除的合数.

Carmichael 数

对于合数 n, 如果对于所有整数 $a\perp n$ 都有同余式 $a^{n-1}\equiv 1 \pmod n$ 成立, 就称 n 为 **Carmichael 数**.

最小的 Carmichael 数是 $561=3\times 11\times 17$.

由 Carmichael 函数的定义可知，合数 n 是 Carmichael 数当且仅当 $\lambda(n)\mid n-1$，其中 $\lambda(n)$ 为 Carmichael 函数. 进一步地，可以得到如下判断合数 n 是否为 Carmichael 数的方法：

Korselt 判别法[42]

合数 n 是 Carmichael 数当且仅当 n 无平方因子且对 n 的任意素因子 p 均有 $(p-1)\mid(n-1)$.

证明

首先证明条件的必要性. 假设 $\lambda(n)\mid(n-1)$. 检查 Carmichael 函数的递推公式可知，如果 n 有平方因子 p^2，那么，一定有 $p\mid\lambda(n)$. 但是 $p\nmid(n-1)$，矛盾. 同理，对于 n 的任意素因子 p，Carmichael 函数的递推公式说明，$(p-1)\mid\lambda(n)$，所以，也有 $(p-1)\mid(n-1)$.

然后证明条件的充分性. 因为 n 是无平方因子的合数，所以它至少有两个不同的素因子，其中一定有奇素因子 p，由 $(p-1)\mid(n-1)$ 可知 $n-1$ 是偶数，n 也就一定是奇数. 对于无平方因子的奇合数 n，由 Carmichael 函数的递推公式可知，$\lambda(n)=\operatorname{lcm}\{p-1:p\mid n\}$. 因此，只要 $(p-1)\mid(n-1)$ 对于所有素因子 p 都成立，就一定有 $\lambda(n)\mid(n-1)$.

从这一判别法出发，可以建立 Carmichael 数的一些简单性质：

推论

Carmichael 数是奇数，没有平方因子，而且至少有 3 个不同的素因子.

证明

前两条性质可以直接从 Korselt 判别法及其证明中得到. 要得到第三条性质，只需要再证明：互异素数 p_1,p_2 的乘积 $n=p_1p_2$ 一定不是 Carmichael 数. 假设 $n=p_1p_2$ 是 Carmichael 数. 由 Korselt 判别法可知，$(p_i-1)\mid(n-1)$. 但是，有

$$n-1=p_1p_2-1\equiv p_2-1 \pmod{p_1-1}.$$

[42] Korselt, A. R. (1899). "Problème chinois." L'Intermédiaire des Mathématiciens. 6: 142–143.

因此，$(p_1-1)\mid(p_2-1)$．同理，$(p_2-1)\mid(p_1-1)$．也就是说，$p_1=p_2$．这与假设矛盾．因此，Carmichael 数 n 至少有 3 个互异素因子．

利用解析数论还可以得到 Carmichael 数分布的一些性质．设 $C(n)$ 为小于等于 n 的 Carmichael 数个数．Alford, Granville, and Pomerance[43]证明，对于充分大的 n，有 $C(n)>n^{2/7}$．由此，Carmichael 数有无限多个．在这之前，Erdős[44]已经证明，$C(n)<n\exp\left(-c\dfrac{\ln n\ln\ln\ln n}{\ln\ln n}\right)$，其中 c 为常数．因此，Carmichael 数的分布（相对于素数来说）十分稀疏．实际上，有 [45] $C(10^9)=646$，$C(10^{18})=1\ 401\ 644$.

参考资料与注释

- Primitive root modulo n - Wikipedia[3]
- The order of a unit - Course Notes[4]
- The primitive root theorem - Amin Witno's notes[5]
- Carmichael function - Wikipedia[6]
- Carmichael's Lambda Function - Brilliant Math & Science Wiki[7]
- Carmichael number - Wikipedia[8]
- Carmichael Number - Wolfram MathWorld[9]

[1] Least prime primitive root of prime numbers

[2] A002997

[3] Primitive root modulo n - Wikipedia

[4] The order of a unit - Course Notes

[5] The primitive root theorem - Amin Witno's notes

[6] Carmichael function - Wikipedia

[7] Carmichael's Lambda Function - Brilliant Math & Science Wiki

[8] Carmichael number - Wikipedia

[9] Carmichael Number - Wolfram MathWorld

9.11.19 离散对数

定义

前置知识：阶与原根→9.11.18@p1275.

离散对数的定义方式和对数类似．取有原根的正整数模数 m，设其一个原根为 g. 对满足 $(a,m)=1$ 的整数 a，我们知道必存在唯一的整数 $0\le k<\varphi(m)$ 使得

$$g^k\equiv a\pmod m$$

我们称这个 k 为以 g 为底，模 m 的离散对数，记作 $k=\operatorname{ind}_g a$，在不引起混淆的情况下可记作 $\operatorname{ind} a$.

显然 $\operatorname{ind}_g 1=0$，$\operatorname{ind}_g g=1$.

性质

离散对数的性质也和对数有诸多类似之处.

性质

设 g 是模 m 的原根，$(a,m)=(b,m)=1$，则：

1. $\operatorname{ind}_g(ab)\equiv\operatorname{ind}_g a+\operatorname{ind}_g b\pmod{\varphi(m)}$

   进而 $(\forall n\in\mathbf{N}),\ \operatorname{ind}_g a^n\equiv n\operatorname{ind}_g a\pmod{\varphi(m)}$

2. 若 g_1 也是模 m 的原根，则 $\operatorname{ind}_g a\equiv\operatorname{ind}_{g_1} a\cdot\operatorname{ind}_g g_1\pmod{\varphi(m)}$

[43] W. R. Alford; Andrew Granville; Carl Pomerance (1994). "There are Infinitely Many Carmichael Numbers." Annals of Mathematics. 140 (3): 703–722.

[44] Erdős, P. (1956). "On pseudoprimes and Carmichael numbers." Publ. Math. Debrecen. 4 (3–4): 201–206.

[45] PINCH, Richard GE. The Carmichael numbers up to 10^{20}.

3. $a \equiv b \pmod m \iff \operatorname{ind}_g a = \operatorname{ind}_g b$

证明

1. $g^{\operatorname{ind}_g(ab)} \equiv ab \equiv g^{\operatorname{ind}_g a} g^{\operatorname{ind}_g b} \equiv g^{\operatorname{ind}_g a + \operatorname{ind}_g b} \pmod m$
2. 令 $x = \operatorname{ind}_{g_1} a$，则 $a \equiv g_1^x \pmod m$. 又令 $y = \operatorname{ind}_g g_1$，则 $g_1 \equiv g^y \pmod m$.
   故 $a \equiv g^{xy} \pmod m$，即 $\operatorname{ind}_g a \equiv xy \equiv \operatorname{ind}_{g_1} a \cdot \operatorname{ind}_g g_1 \pmod{\varphi(m)}$
3. 注意到
$$\begin{aligned}\operatorname{ind}_g a = \operatorname{ind}_g b &\iff \operatorname{ind}_g a \equiv \operatorname{ind}_g b \pmod{\varphi(m)} \\ &\iff g^{\operatorname{ind}_g a} \equiv g^{\operatorname{ind}_g b} \pmod m \\ &\iff a \equiv b \pmod m\end{aligned}$$

大步小步算法

目前离散对数问题仍不存在多项式时间经典算法（离散对数问题的输入规模是输入数据的位数）. 在密码学中，基于这一点人们设计了许多非对称加密算法，如 Ed25519[1].

在算法竞赛中，BSGS（baby-step giant-step，大步小步算法）常用于求解离散对数问题. 形式化地说，对 $a, b, m \in \mathbf{Z}^+$，该算法可以在 $O(\sqrt m)$ 的时间内求解

$$a^x \equiv b \pmod m$$

其中 $a \perp m$. 方程的解 x 满足 $0 \le x < m$.（注意 m 不一定是素数）

算法描述

令 $x = A\lceil\sqrt m\rceil - B$，其中 $0 \le A, B \le \lceil\sqrt m\rceil$，则有 $a^{A\lceil\sqrt m\rceil - B} \equiv b \pmod m$，稍加变换，则有 $a^{A\lceil\sqrt m\rceil} \equiv ba^B \pmod m$.

我们已知的是 a, b, 所以我们可以先算出等式右边的 ba^B 的所有取值, 枚举 B, 用 `hash`/`map` 存下来, 然后逐一计算 $a^{A\lceil\sqrt m\rceil}$，枚举 A，寻找是否有与之相等的 ba^B，从而我们可以得到所有的 x，$x = A\lceil\sqrt m\rceil - B$.

注意到 A, B 均小于 $\lceil\sqrt m\rceil$，所以时间复杂度为 $\Theta(\sqrt m)$，用 `map` 则多一个 log.

为什么要求 a 与 m 互质

注意到我们求出的是 A, B，我们需要保证从 $a^{A\lceil\sqrt m\rceil} \equiv ba^B \pmod m$ 可以推回 $a^{A\lceil\sqrt m\rceil - B} \equiv b \pmod m$，后式是前式左右两边除以 a^B 得到，所以必须有 $a^B \perp m$ 即 $a \perp m$.

扩展 BSGS 算法

对 $a, b, m \in \mathbf{Z}^+$，求解

$$a^x \equiv b \pmod m$$

其中 a, m 不一定互质.

当 $(a, m) = 1$ 时，在模 m 意义下 a 存在逆元，因此可以使用 BSGS 算法求解. 于是我们想办法让他们变得互质.

具体地，设 $d_1 = (a, m)$. 如果 $d_1 \nmid b$，则原方程无解. 否则我们把方程同时除以 d_1，得到

$$\frac{a}{d_1} \cdot a^{x-1} \equiv \frac{b}{d_1} \pmod{\frac{m}{d_1}}$$

如果 a 和 $\frac{m}{d_1}$ 仍不互质就再除，设 $d_2 = \left(a, \frac{m}{d_1}\right)$. 如果 $d_2 \nmid \frac{b}{d_1}$，则方程无解；否则同时除以 d_2 得到

$$\frac{a^2}{d_1 d_2} \cdot a^{x-2} \equiv \frac{b}{d_1 d_2} \pmod{\frac{m}{d_1 d_2}}$$

同理，这样不停的判断下去，直到 $a \perp \dfrac{m}{d_1 d_2 \cdots d_k}$.

记 $D = \prod_{i=1}^{k} d_i$，于是方程就变成了这样：

$$\frac{a^k}{D} \cdot a^{x-k} \equiv \frac{b}{D} \pmod{\frac{m}{D}}$$

由于 $a \perp \dfrac{m}{D}$，于是推出 $\dfrac{a^k}{D} \perp \dfrac{m}{D}$. 这样 $\dfrac{a^k}{D}$ 就有逆元了，于是把它丢到方程右边，这就是一个普通的 BSGS 问题了，于是求解 $x-k$ 后再加上 k 就是原方程的解啦.

注意，不排除解小于等于 k 的情况，所以在消因子之前做一下 $\Theta(k)$ 枚举，直接验证 $a^i \equiv b \pmod{m}$，这样就能避免这种情况.

基于值域预处理的快速离散对数

前文的 BSGS 算法时间复杂度为单次 $O(\sqrt{m})$，在询问量级较大的时候效率较低. 若每次求解的模数是一个固定的质数 p，我们就有一个基于值域预处理的快速算法.

我们已经知道 $\operatorname{ind}_g(ab) \equiv \operatorname{ind}_g a + \operatorname{ind}_g b \pmod{p-1}$，所以我们可以只对所有质数通过 BSGS 算法计算离散对数，合数的离散对数则可通过该式转化为若干已知的质数离散对数值之和. 此时复杂度仍然不优，我们考虑只预处理一部分的离散对数，具体来说，我们预处理 1 到 $L = \lfloor \sqrt{p} \rfloor + 1$ 的离散对数. 注意此时的 BSGS 块长 B **不能取** $O(\sqrt{L})$，因为 BSGS 预处理（插入哈希表）部分的复杂度是 $O(B)$，而查询一共需要 $O(\pi(L))$ 次，则总时间复杂度为 $O\left(B + \dfrac{\pi(L)p}{B}\right)$，此时取 $B = O(\sqrt{\pi(L)p})$ 才是最优. 由 素数定理→9.11.3@p1179 $\pi(n) \sim \dfrac{n}{\log n}$，则总的预处理时间复杂度可以平衡为 $O\left(\dfrac{p^{3/4}}{\log^{1/2} p}\right)$.

接下来是如何求答案. 假设当前要求的是 $\operatorname{ind}_g y$，若 $y \le L$ 则直接返回，否则设 $p = vy + r$，则 $v = \left\lfloor \dfrac{p}{y} \right\rfloor < L$，$r = p \bmod y$，$y = \dfrac{p-r}{v}$，从而

$$\begin{aligned}\operatorname{ind}_g y &\equiv \operatorname{ind}_g(p-r) - \operatorname{ind}_g v \\ &\equiv \operatorname{ind}_g(-r) - \operatorname{ind}_g v \\ &\equiv \operatorname{ind}_g(p-1) + \operatorname{ind}_g r - \operatorname{ind}_g v \pmod{p-1}.\end{aligned}$$

注意到 $\operatorname{ind}_g(p-1) = (p-1)/2$，因此只需递归计算 r 的离散对数即可.

我们还可以考虑 y 的另一种表达方式，注意到 $p = vy + r = (v+1)y + r - y$，则 $y = \dfrac{p-r+y}{v+1}$，从而

$$\operatorname{ind}_g y \equiv \operatorname{ind}_g(y-r) - \operatorname{ind}_g(v+1) \pmod{p-1}.$$

我们有 $v+1 \le L$，因此只需要递归计算 $y-r$ 的离散对数即可.

综合这两种计算方式，我们有 $\min\{r, y-r\} \le \dfrac{y}{2}$，所以递归计算较小的一方即可达到 $O(\log p)$ 的查询复杂度.

至此我们得到了一个时间复杂度为 $O\left(\dfrac{p^{3/4}}{\log^{1/2} p}\right) - O(\log p)$ 的算法.

Luogu11175【模板】基于值域预处理的快速离散对数[2]

```cpp
#include <cmath>
#include <iostream>
#include <unordered_map>
using namespace std;

const int N = 1'000'000;  // sqrt(P * sqrt(P) / ln(P))

unordered_map<int, int> M;
```

```cpp
int Lg[N], p[N], t;
bool vis[N];
int P, g, B, sq, LP_1, inv;

int fadd(int x, int y, int P) {
  x += y;
  if (x >= P) x -= P;
  return x;
}

int fsub(int x, int y, int P) {
  x -= y;
  if (x < 0) x += P;
  return x;
}

int fmul(int x, int y, int P) { return 1ll * x * y % P; }

int qpow(int x, int y) {
  int ans = 1;
  while (y) {
    if (y & 1) ans = fmul(ans, x, P);
    x = fmul(x, x, P);
    y >>= 1;
  }
  return ans;
}

int calc(int x) {
  int s = x;
  for (int i = 0; i <= P / B; ++i) {
    if (M.find(s) != M.end()) return i * B + M[s];
    s = fmul(s, inv, P);
  }
  return -1;
}

void init() {
  int s = 1;
  for (int i = 0; i < B; ++i) {
    if (M.find(s) != M.end()) break;
    M[s] = i, s = fmul(s, g, P);
  }
  inv = qpow(qpow(g, B), P - 2);
  for (int i = 2; i <= sq; ++i) {
    if (!vis[i]) {
      p[++t] = i;
      Lg[i] = calc(i);
    }
    for (int j = 1; j <= t && p[j] * i <= sq; ++j) {
      vis[p[j] * i] = true;
      Lg[p[j] * i] = fadd(Lg[p[j]], Lg[i], P - 1);
```

```
      if (i % p[j] == 0) break;
    }
  }
}

int solve(int y) {
  if (y <= sq) return Lg[y];
  int v = P / y, r = P % y;
  if (r < y - r) return fadd(fsub(solve(r), Lg[v], P - 1), LP_1, P - 1);
  return fsub(solve(y - r), Lg[v + 1], P - 1);
}

int main() {
  ios::sync_with_stdio(false);
  cin.tie(nullptr);
  cin >> P >> g;
  sq = sqrt(P) + 1;
  B = sqrt(1ll * P * sqrt(P) / log(P));
  init();
  LP_1 = (P - 1) / 2;  // g ^ LP_1 = P - 1 (mod P)
  int T;
  cin >> T;
  while (T--) {
    int x;
    cin >> x;
    cout << solve(x) << '\n';
  }
  return 0;
}

```

习题

- SPOJ MOD[3] 模板
- SDOI2013 随机数生成器[4]
- SGU261 Discrete Roots[5] 模板
- SDOI2011 计算器[6] 模板
- Luogu4195【模板】exBSGS/Spoj3105 Mod[7] 模板
- Codeforces - Lunar New Year and a Recursive Sequence[8]
- LOJ6542 离散对数[9] index calculus 方法，非模板

本页面部分内容以及代码译自博文 Дискретное извлечение корня[10] 与其英文翻译版 Discrete Root[11]．其中俄文版版权协议为 Public Domain + Leave a Link；英文版版权协议为 CC-BY-SA 4.0．

参考资料

1. Discrete logarithm - Wikipedia[12]
2. 潘承洞，潘承彪．初等数论．
3. 冯克勤．初等数论及其应用．

外部参考资料

[1] Ed25519

[2] Luogu11175【模板】基于值域预处理的快速离散对数

[3] SPOJ MOD

[4] SDOI2013 随机数生成器

[5] SGU261 Discrete Roots

[6] SDOI2011 计算器

[7] Luogu4195【模板】exBSGS/Spoj3105 Mod

[8] Codeforces - Lunar New Year and a Recursive Sequence

[9] LOJ6542 离散对数

[10] Дискретное извлечение корня

[11] Discrete Root

[12] Discrete logarithm - Wikipedia

9.11.20 高次剩余 & 单位根

前置知识：离散对数→9.11.19@p1286

本文讨论模意义下的高次剩余和单位根，并介绍模意义下开方运算的算法.

高次剩余

模运算下的高次剩余，可以认为是在讨论模意义下开高次方的可行性. 它是 二次剩余→9.11.17@p1263 的推广.

自 k 次剩余

令整数 $k \geq 2$，整数 a 和正整数 m 互素. 若存在整数 x 使得

$$x^k \equiv a \pmod{m},$$

则称 a 为模 m 的 k **次剩余**（k-th residue），x 为 a 模 m 的 k **次方根**（k-th root）；否则称 a 为模 m 的 k **次非剩余**（k-th nonresidue）.

也就是说，a 模 m 的 k 次方根存在，当且仅当 a 是模 m 的 k 次剩余.

性质

类似二次剩余，可以讨论 k 次剩余的判定、个数以及 k 次剩余类的个数问题. 和其他 同余方程→9.11.16@p1258 问题一样，可以通过 中国剩余定理→9.11.12@p1236 将它们转化为素数幂模的情形. 根据原根的有无，这进一步区分为奇素数幂模和模数为 2 的幂次的情形.

奇素数幂模的情形较为简单. 事实上，对于所有原根存在的情形，都有如下结论：

定理

设整数 $k \geq 2$，整数 a 和正整数 m 互素. 设模 m 的原根存在，且 g 是模 m 的一个原根. 记 $d = \gcd(k, \varphi(m))$ 且 $d' = \dfrac{\varphi(m)}{d}$，其中，$\varphi(m)$ 是 欧拉函数→9.11.5@p1196. 那么，有：

1. a 为模 m 的 k 次剩余，当且仅当
   $a^{d'} \equiv 1 \pmod{m}$.
2. 当 a 为模 m 的 k 次剩余时，同余意义下，a 模 m 恰有 d 个互不相同的 k 次方根，且它们具有形式
   $x \equiv g^{y_0 + id'} \pmod{m},\ 0 \leq y_0 < d',\ i = 0, 1, \cdots, d-1.$
3. 模 m 的 k 次剩余类的个数为 d'，且它们的全体就是
   $\{g^{di} \bmod m : 0 \leq i < d'\}$.

证明

因为 $a \perp m$，所以 $x \perp m$. 因为 g 是模 m 的原根，所以，x 和 a 均与某个 g 的幂次同余. 设 $x \equiv g^y \pmod m$，方程 $x^k \equiv a \pmod m$ 就等价于

$$g^{ky} \equiv g^{\operatorname{ind}_g a} \pmod m.$$

其中，$\operatorname{ind}_g a$ 是离散对数. 根据 阶的性质→9.11.18@p1275 和 $\delta_m(g) = \varphi(m)$，这等价于同余方程

$$ky \equiv \operatorname{ind}_g a \pmod{\varphi(m)}.$$

这是关于 y 的 线性同余方程→9.11.11@p1233. 应用该页面对其解结构的分析，就可以知道方程有解当且仅当 $d \mid \operatorname{ind}_g a$，且通解形式为

$$y \equiv y_0 + id' \pmod{\varphi(m)},\ 0 \le y_0 < d',\ i = 0, 1, \cdots, d-1.$$

由此，就几乎可以得到本定理的全部内容；唯一需要额外说明的是判别式 $a^{d'} \equiv 1 \pmod m$. 由 阶的性质 3→9.11.18@p1275 可知

$$\delta_m(a) = \delta_m(g^{\operatorname{ind}_g a}) = \frac{\varphi(m)}{\gcd(\varphi(m), \operatorname{ind}_g a)}.$$

又已知方程有解当且仅当 $d \mid \operatorname{ind}_g a$，亦即 $\delta_m(a) \mid d'$. 由 阶的性质 2→9.11.18@p1275 可知，这就等价于该判别式.

模数为 2 的幂次的情形较为特殊. 为处理这种情形，需要用到关于模 2^e 既约剩余系结构的一个 结论→9.11.18@p1275：所有奇数 a 都唯一地同余于某个 $(-1)^s 5^r \bmod 2^e$ 形式的整数，其中，$s \in \{0, 1\}$ 且 $0 \le r < 2^{e-2}$. 借助这一结果，可以得到如下结论：

定理

设整数 $k \ge 2$，奇数 a 和正整数 $m = 2^e$ 且 $e \ge 2$. 那么，当 k 是奇数时，有：

1. a 恒为模 m 的 k 次剩余.
2. a 模 m 的 k 次方根有且仅有一个.
3. 模 m 的 k 次剩余类个数为 2^{e-1}，且它们就是全体既约剩余类.

当 k 是偶数时，记 $d = \gcd(k, 2^{e-2})$ 且 $d' = \dfrac{2^{e-2}}{d}$，有：

1. a 为模 m 的 k 次剩余，当且仅当 $a \equiv 1 \pmod 4$ 且 $a^{d'} \equiv 1 \pmod m$.
2. 当 a 为模 m 的 k 次剩余时，同余意义下，a 模 m 恰有 $2d$ 个互不相同的 k 次方根，且它们具有形式

   $$x \equiv \pm 5^{y_0 + id'} \pmod{2^e},\ 0 \le y_0 < d',\ i = 0, 1, \cdots, d-1.$$

3. 模 m 的 k 次剩余类的个数为 d'，且它们的全体就是

   $$\{5^{di} \bmod m : 0 \le i < d'\}.$$

证明

因为 $a \perp m$, 所以 $x \perp m$. 因为 x 和 a 都是奇数, 由前述结论可知, 可以设 $a \equiv (-1)^s 5^r \pmod{2^e}$ 且 $x \equiv (-1)^z 5^y \pmod{2^e}$. 因为表示是唯一的, 所以同余方程 $x^k \equiv a \pmod{2^e}$ 等价于 线性同余方程→9.11.11@p1233 组

$$kz \equiv s \pmod 2,$$

$$ky \equiv r \pmod{2^{e-2}}.$$

结合该页面对于线性同余方程解的分析，就可以得到同余方程 $x^k \equiv a \pmod{2^e}$ 解的结构. 根据 k 的奇偶性不同，可以分为两种情形：

- 当 k 是奇数时，因为 $\gcd(k,2)=\gcd(k,2^{e-2})=1$，所以两个线性同余方程对于所有 s,r 都有解，故而原同余方程对于所有奇数 a 总是有解.
- 当 k 是偶数时，第一个方程有解当且仅当 $2\mid s$，第二个方程有解当且仅当 $d=\gcd(k,2^{e-2})\mid r$. 将两者结合就得到 k 次剩余类的全体形式. 直接计算可知，第一个条件等价于 $a\equiv 1 \pmod 4$；重复奇素数幂情形的分析可知，第二个条件等价于 $a^{d'}\equiv 1 \pmod m$. 将两点结合起来就得到定理中的判定方法. 两个线性同余方程的通解也是已知的：

$$z\equiv 0,1 \pmod 2,$$

$$y\equiv y_0+id' \pmod{2^{e-2}},\ 0\leq y_0<d',\ i=0,1,\cdots,d-1.$$

将两者结合就得到原方程的通解.

这就完全解决了不同模数下 k 次剩余的判定问题. 二次剩余中的 Legendre 记号和二次互反律等内容也可以推广到高次剩余的情形，但这并不容易，需要用到 分圆域→9.16.4@p1849 等概念. 在代数数论中，二次互反律最终可以推广到 Artin 互反律[1].

单位根

作为 k 次方根的特殊情形，本节讨论 k 次（本原）单位根的概念. 它可以看作是复数域 $\mathbf{C}$ 中 k 次 单位根→9.10@p1153 的概念在模 m 既约剩余系 $\mathbf{Z}_m^*$ 中的对应. 当模数 m 合适时，用模 m 的 k 次本原单位根代替复数根 ω_k 可以加速计算.

类似于复数域的情形，有如下定义：

自 模 m 的 k 次单位根

对于模数 m，元素 1 的 k 次方根称为 **模 m 的 k 次单位根**（k-th root of unity modulo m）. 特别地，如果 x 是模 m 的一个 k 次单位根，且它不是模 m 的任何 $k'<k$ 次单位根，那么，也称 x 为 **模 m 的 k 次本原单位根**（k-th primitive root of unity modulo m）.

比较 原根的定义→9.11.18@p1275 可知，原根 g 就是模 m 的 $\varphi(m)$ 次本原单位根，其中，$\varphi(m)$ 是 欧拉函数→9.11.5@p1196.

当模 m 的 k 次本原单位根存在时，它的代数性质和 k 次本原单位复根 ω_k 一致，可以代替 ω_k 进行各种计算. 例如，将它应用于 快速傅里叶变换→9.12.3@p1527 中，就得到有限域[46] 上的 快速数论变换→9.12.4@p1539.

性质

复数域中，任意次（本原）单位根都存在. 但是，数论中的（本原）单位根并非如此.

性质

对于模数 m，设 $\lambda(m)$ 为它的 Carmichael 函数→9.11.18@p1275，有：

1. 所有与 m 互素的整数 a 都是模 m 的 $\delta_m(a)$ 次本原单位根，其中，$\delta_m(a)$ 是 a 模 m 的 阶→9.11.18@p1275.
2. 元素 a 是模 m 的 k 次单位根，且 k' 是 k 的任意倍数，那么 a 也是模 m 的 k' 次单位根.
3. 元素 a 是模 m 的 k 次（本原）单位根，那么元素 a^ℓ 是模 m 的 $\dfrac{k}{\gcd(k,\ell)}$ 次（本原，相应地）单位根.

[46] 实际上，模数 m 未必是素数. 只要 a 是模 m 的 $k=2^e$ 次本原单位根，就可以用于模 m 的快速数论变换. 但是，由于通常需要处理的 2^e 比较大，这意味着模数 m 中的每个素因子都是 $c2^e+1$ 形式. 因此，单个素因子就很大，而模数 m 通常会更大，因而一般模数的情形并没有素数模的情形常用.

4. 当 k' 遍历 k 的因数，所有模 m 的 k' 次本原单位根恰构成模 m 的 k 次单位根的一个划分．而且，对于 $\ell \perp k$，映射 $x \mapsto x^\ell$ 给出 k 次单位根之间的双射，且保持上述划分不变：它将 $k' \mid k$ 次本原单位根仍然映射到 k' 次本原单位根．
5. 模 m 的 k 次本原单位根存在，当且仅当 $k \mid \lambda(m)$．特别地，模 m 的 $\lambda(m)$ 次本原单位根存在，称为 **模 m 的 λ-原根**．
6. 元素 a 是模 m 的 k 次本原单位根，当且仅当 $a^k \equiv 1 \pmod m$ 且对于任意素因子 $p \mid k$ 都有 $a^{k/p} \not\equiv 1 \pmod m$．

证明

根据阶的定义，所有与 m 互素的整数 a 都是模 m 的 $\delta_m(a)$ 次本原单位根，其中，$\delta_m(a)$ 是 a 模 m 的阶．反过来，如果 a 是模 m 的 k 次单位根，那么 $\gcd(a^k, m) = 1$，所以 $\gcd(a, m) = 1$．因此，a 是模 m 的（本原）单位根，当且仅当 a 与 m 互素．这就是性质 1.

直接验证定义可知，只要 $k \mid k'$，就可以从 $a^k \equiv 1 \pmod m$ 推出 $a^{k'} \equiv 1 \pmod m$，这就是性质 2．根据 阶的性质→9.11.18@p1275 可知

$$\delta_m(a^\ell) = \frac{\delta_m(a)}{\gcd(\delta_m(a), \ell)}.$$

如果 a 是模 m 的 k 次本原单位根，那么，$\delta_m(a) = k$，直接代入上式就得到 a^ℓ 是模 m 的 $\dfrac{k}{\gcd(k, \ell)}$ 次本原单位根．如果 a 只是模 m 的 k 次单位根，设它是 $k' \mid k$ 次本原单位根，故而 a^ℓ 是模 m 的 $\dfrac{k'}{\gcd(k', \ell)}$ 次本原单位根．由于 $k' \mid k$，有

$$\frac{k'}{\gcd(k', \ell)} \mid \frac{k}{\gcd(k, \ell)},$$

再由性质 2，就得到 a^ℓ 是模 m 的 $\dfrac{k}{\gcd(k, \ell)}$ 次单位根．这就是性质 3.

对于 $k' \mid k$，由性质 2，模 m 的 k' 次本原单位根必然是模 m 的 k 次单位根；反过来，每个模 m 的 k 次单位根 a 都是 $\delta_m(a)$ 次本原单位根，且 $\delta_m(a) \mid k$．这些集合两两不交，故而构成划分．而对于 $\ell \perp k$，总有 $\ell \perp k'$，因此对于模 m 的 k' 次本原单位根 a，总有 a^ℓ 是模 m 的 k' 次本原单位根．取 $\ell' = \ell^{-1} \bmod k$，可以验证 $x \mapsto x^\ell$ 和 $x \mapsto x^{\ell'}$ 互为逆映射，因此，$x \mapsto x^\ell$ 是双射．这就是性质 4.

根据 Carmichael 函数的性质可知，模 m 的 $\lambda(m)$ 次本原单位根总是存在的，设它为 a，且 $\delta_m(a) = \lambda(m)$．对于 $k \mid \lambda(m)$，设 $k' = \dfrac{\lambda(m)}{k}$，总有

$$\delta_m(a^{k'}) = \frac{\lambda(m)}{\gcd(\lambda(m), k')} = \frac{\lambda(m)}{k'} = k.$$

因此，$a^{k'}$ 是 k 次本原单位根．而根据 Carmichael 函数的定义，所有 $x \perp m$ 的阶都是 $\lambda(m)$ 的因子．这就得到性质 5.

几乎重复 原根判定定理→9.11.18@p1275 的证明，就可以得到性质 6．这一判别方法实际上在验证 $\delta_m(a) = k$.

从这些性质可以看出，相对于原根存在的情形，模 m 的 λ-原根起到了类似的基础作用．与原根不同的是，λ-原根的幂次并不能用于生成模 m 的全体单位根．尽管如此，由于 λ-原根的密度并不低[47]，如果确

[47]根据 原根个数相关结论→9.11.18@p1275 可知，λ-原根的数量恰为 $\varphi(\lambda(m))$，其中，$\varphi(\cdot)$ 和 $\lambda(\cdot)$ 分别是欧拉函数和 Carmichael 函数．因为对于几乎所有整数 m，都有 $\lambda(m)/m = \exp(-(1+o(1))\log\log m \log\log\log m)$，而存在 $C > 0$，使得对于整数 $m > 2$，都有 $\varphi(m)/m \geq C/\log\log m$，所以，对于几乎所有整数 m，都有 $\varphi(\lambda(m))/m = \exp(-(1+o(1))\log\log m \log\log\log m)$．其中，指数部分系数中的 $o(1)$ 吸收了因子 $\varphi(\lambda(m))/\lambda(m)$ 的贡献．故而，λ-原根可以在期望 $\exp((1+o(1))\log\log m \log\log\log m)$ 次内找到．关于欧拉函数的估计，可以参考论文 Rosser, J. Barkley, and Lowell Schoenfeld. "Approximate formulas for some functions of prime numbers." Illinois Journal of Mathematics 6, no. 1 (1962): 64-94.

实需要找到 k 次本原单位根，可以首先通过随机方法找到一个 λ-原根，再通过求幂次得到一个 k 次本原单位根.

如果已知 a 模 m 的一个 k 次方根，可以通过模 m 的全体 k 次单位根生成 a 模 m 的全体 k 次方根.

定理

设 x 是 a 模 m 的一个 k 次方根，当 r 遍历模 m 的全体 k 次单位根时，xr 遍历 a 模 m 的全体 k 次方根.

证明

对于 a 模 m 的两个 k 次方根 x, y，设 $r = x^{-1}y \bmod m$，那么 r 满足 $r^k \equiv 1 \pmod m$，是模 m 的 k 次单位根. 反过来，只要 r 是模 m 的 k 次单位根，那么，$(xr)^k = x^k r^k \equiv a \pmod m$，也就是说，$xr$ 是 a 模 m 的 k 次方根.

利用 k 次单位根生成全体 k 次方根, 就类似于利用齐次线性方程组的解生成非齐次线性方程组的通解一样.

前面讨论的是一般情形. 仅对于原根存在的情形，单位根的结构更为简单:

定理

对于模数 m，设模 m 的原根存在，且 a 是模 m 的 k 次本原单位根. 那么，b 是模 m 的 k 次单位根，当且仅当它可以表示为 a 的幂次.

证明

设 g 是模 m 的原根，那么，所有与 m 互素的元素都可以表示为 g 的幂次. 那么，a 是模 m 的 k 次本原单位根，当且仅当

$$\delta_m(a) = \delta_m(g^{\operatorname{ind}_g a}) = \frac{\varphi(m)}{\gcd(\varphi(m), \operatorname{ind}_g a)} = k.$$

类似地，b 是模 m 的 k 次单位根，当且仅当

$$\delta_m(b) = \delta_m(g^{\operatorname{ind}_g b}) = \frac{\varphi(m)}{\gcd(\varphi(m), \operatorname{ind}_g b)} = k' \mid k.$$

所以，有

$$\gcd(\varphi(m), \operatorname{ind}_g a) \mid \gcd(\varphi(m), \operatorname{ind}_g b) \mid \operatorname{ind}_g b.$$

根据对线性同余方程的 分析→9.11.11@p1233 可知，这一条件就等价于方程

$$(\operatorname{ind}_g a)x \equiv \operatorname{ind}_g b \pmod{\varphi(m)}$$

有解. 将这一条件对 g 取幂，就得到 $a^x \equiv b \pmod m$，亦即 b 可以表示为 a 的幂次.

这一定理说明，原根存在时，全体 k 次单位根呈现 循环群→9.16.2@p1812 的结构，而 k 次本原单位根则是该循环群的生成元. 稍后将会看到，Tonelli–Shanks 算法正是利用这一点，加速了开方运算中离散对数部分的计算.

模意义下开方

最后，本文讨论 k 次方根的求法. 对于 $k = 2$ 的情形，有 很多高效算法→9.11.17@p1263 可以用于模意义下开平方运算. 但是，对于一般的 k，并没有已知的多项式时间算法. 本节将介绍两种常见算法，分别可

关于 Carmichael 函数的估计，可以参考论文 Erdos, Paul, Carl Pomerance, and Eric Schmutz. "Carmichael's lambda function." Acta Arith 58, no. 4 (1991): 363-385.

以在 $O(m^{1/2})$ 和 $O(m^{1/4+\varepsilon})$ 时间内求出一个 k 次方根．利用中国剩余定理总是可以将问题转换为素数幂模的情形，因此，本节主要讨论素数幂模情形的解法．

朴素算法

前文对于 k 次剩余性质的 分析→9.11@p1160 实际上已经指出了一种求解素数幂模下 k 次方根的方法．严格来说，前文解决的情形是被开方数 a 与模数 m 互素的情形．算法过程总结如下：

- 当 $m=p^e$ 是奇素数幂时，设模 m 的一个原根是 g．那么，方程 $x^k\equiv a\pmod m$ 可以转化为线性同余方程

  $$ky\equiv \operatorname{ind}_g a\pmod{\varphi(m)}.$$

  其中，$\operatorname{ind}_g a$ 可以通过 BSGS 算法→9.11.19@p1286 求出，而 线性同余方程→9.11.11@p1233 的全体解容易求出．由此，就可以得到 a 的全部 k 次方根 $x\equiv g^y\pmod m$．

  除此之外，还有另一种相仿的思路．同样是设 $x\equiv g^y\pmod m$，还可以通过变形

  $$x^k\equiv (g^k)^y\equiv a\pmod m$$

  转化为求底数为 g^k 时 a 的离散对数．这同样可以通过 BSGS 算法找到一组特解．它的通解可以通过前文的解的表达式求出，也就是将特解与全体 k 次单位根逐一相乘得到．

  无论采用哪种思路，原根已知时，该算法求出单个解的复杂度都是 $O(m^{1/2})$．因为可以在 $o(m^{1/2})$ 时间内找到一个原根，所以，总的时间复杂度仍然是 $O(m^{1/2})$．
- 当 $m=2^e$ 且 $e\in\mathbf{N}_+$ 时，可以首先求出 $a\equiv(-1)^s5^r\pmod m$ 中的 s,r．这两个指数中，s 可以在 $O(1)$ 时间内确定：

  $$s=\begin{cases}0, a\equiv 1\pmod 4,\\ 1, a\equiv 3\pmod 4.\end{cases}$$

  而 $r=\operatorname{ind}_5((-1)^s a)$ 可以通过 BSGS 算法在 $O(m^{1/2})$ 时间内求出．接下来，只需要求解线性同余方程组：

  $$\begin{aligned}kz&\equiv s\pmod 2,\\ ky&\equiv r\pmod{2^{e-2}}.\end{aligned}$$

  这个线性同余方程组的通解 (z,y) 容易求出，而 $x=(-1)^z5^y$ 就是所求的方根．这一算法求出单个解的复杂度仍然是 $O(m^{1/2})$．

当然，对于无解的情形，其实可以通过前文叙述的判别方法在 $O(\log m)$ 时间内快速判断，而无需在求解过程中判断．

求素数模 k 次方根的参考实现如下：（代码仅作示意，由于时间复杂度过高，无法通过本题）

模板题 Library Checker - Kth Root (Mod)[2] 参考实现

```cpp
// Submission (TLE): https://judge.yosupo.jp/submission/320582
#include <algorithm>
#include <cmath>
#include <iostream>
#include <tuple>
#include <unordered_map>
#include <vector>

struct PrimePower {
  int p, e, pe;

  PrimePower(int p, int e, int pe) : p(p), e(e), pe(pe) {}
};
```

```cpp

// Factorization.
auto factorize(int n) {
  std::vector<PrimePower> ans;
  for (int x = 2; x * x <= n; ++x) {
    int e = 0, pe = 1;
    for (; n % x == 0; n /= x, ++e, pe *= x);
    if (e) ans.emplace_back(x, e, pe);
  }
  if (n > 1) ans.emplace_back(n, 1, n);
  return ans;
}

// Binary exponentiation.
int pow(int a, int b, int m = 0) {
  int res = 1;
  for (; b; b >>= 1) {
    if (b & 1) res = m ? (long long)res * a % m : res * a;
    a = m ? (long long)a * a % m : a * a;
  }
  return res;
}

// Find a primitive root modulo prime.
int primitive_root(int p) {
  std::vector<int> exp;
  for (auto factor : factorize(p - 1)) {
    exp.push_back((p - 1) / factor.p);
  }
  int ans = 0;
  bool succ = false;
  while (!succ) {
    ++ans;
    succ = true;
    for (int b : exp) {
      if (pow(ans, b, p) == 1) {
        succ = false;
        break;
      }
    }
  }
  return ans;
}

// Discrete logarithm. (BSGS Algorithm)
int log(int g, int a, int m) {
  int b = std::sqrt(m + 0.25l) + 1;
  std::unordered_map<int, int> mp;
  int po0 = a % m, po1 = 1;
  for (int i = 0; i < b; ++i) {
    mp[po0] = i;
    po0 = (long long)po0 * g % m;
```

```
  }
  po0 = pow(g, b, m);
  for (int j = 1; j <= b; ++j) {
    po1 = (long long)po1 * po0 % m;
    if (mp.count(po1)) return j * b - mp[po1];
  }
  return -1;
}

// Extended Euclidean Algorithm.
int ex_gcd(int a, int b, int& x, int& y) {
  if (!b) {
    x = 1;
    y = 0;
    return a;
  } else {
    int d = ex_gcd(b, a % b, y, x);
    y -= a / b * x;
    return d;
  }
}

// Solves the linear congruence equation: Ax = B mod N.
// Return the least nonnegative solution and the common difference.
std::pair<int, int> solve_linear(int a, int b, int n) {
  int x, y;
  int d = ex_gcd(a, n, x, y);
  if (b % d) return {-1, -1};
  n /= d;
  x = ((long long)x * (b / d) % n + n) % n;
  return {x, n};
}

// Subroutine: Find a K-th root with a primitive root G known.
int calc(int g, int k, int a, int p) {
  int ind = log(g, a, p);
  if (ind == -1) return -1;
  int y0, d;
  std::tie(y0, d) = solve_linear(k, ind, p - 1);
  if (y0 == -1) return -1;
  return pow(g, y0, p);
}

// Find a K-th root of A modulo prime P.
int kth_roots_mod_p(int k, int a, int p) {
  a %= p;
  if (k == 0) return a == 1 ? 0 : -1;
  if (a == 0) return 0;
  return calc(primitive_root(p), k, a, p);
}

void solve() {
```

```
118   int k, y, p;
119   std::cin >> k >> y >> p;
120   std::cout << kth_roots_mod_p(k, y, p) << '\n';
121 }
122
123 int main() {
124   int t;
125   std::cin >> t;
126   for (; t; --t) {
127     solve();
128   }
129 }
130
```

改良 Tonelli–Shanks 算法

将用于模意义下开平方的 Tonelli–Shanks 算法→9.11.17@p1263 做适当推广，就可以解决素数幂模下开方运算．一种较为直接的推广方式是 Adleman–Manders–Miller 算法 [48]，但是它的复杂度仍然不够优秀 [49]．本节介绍由 sugarknri、Min_25、37zigen 等人提出的改良 Tonelli–Shanks 算法．它可以在 $O(m^{1/4+\varepsilon})$ 时间内求出一个 k 次方根．

Tonelli–Shanks 算法的核心想法是，将离散对数的求解放到阶为 2^e 的群里，进而降低时间复杂度．类似地，对于任意素数幂 p^e 阶群内的离散对数，同样可以较为高效地求解，但是算法的复杂度为 $\Omega(\sqrt{p})$．Adleman–Manders–Miller 算法将 k 次方根的求解分拆为多个素数幂阶群内离散对数的计算，但是受限于 k 的最大素因子 $p_{\max}(k)$ 的大小，算法复杂度仍然为 $\Omega\left(\sqrt{p_{\max}(k)}\right)$．本节算法进一步改良了这一过程，避免了对较大的素因子计算离散对数，进而将整体复杂度控制到 $O(m^{1/4+\varepsilon})$．

过程

考虑素数幂模 m 下 a 的 k 次方根的计算，即求解同余方程：

$$x^k \equiv a \pmod{m}.$$

特别地，对于 $m = 2^e$ 的情形，还需要保证 $a \equiv 1 \pmod{4}$，进而 a 可以写成 $g = 5$ 的幂次．类似前文讨论，模 2^e 下 k 次方根的计算总是可以转化为这样的情形．处理模 2^e 的情形时，本节提到的 $\varphi(m)$ 都应换作 $\delta_m(5) = 2^{e-2}$．

首先，问题可以转化为开方次数整除 $\varphi(m)$ 的情形．设 $d = \gcd(k, \varphi(m))$．那么，由 k 次剩余的性质可知，当 a 模 m 是 k 次剩余时，a 总是模 m 的 $\dfrac{\varphi(m)}{d}$ 次单位根．根据单位根的性质，对于任意 $\ell \perp \dfrac{\varphi(m)}{d}$，映射 $x \mapsto x^\ell$ 都是 $\dfrac{\varphi(m)}{d}$ 次单位根之间的双射．因此，可以取

$$\ell = \left(\frac{k}{d}\right)^{-1} \bmod \frac{\varphi(m)}{d}.$$

将原来的同余方程两侧同时取 ℓ 次幂，就得到

$$x^d \equiv x^{k\ell} \equiv a^\ell =: b \pmod{m}.$$

---

[48] 原始论文参见 Adleman, Leonard, Kenneth Manders, and Gary Miller. "On taking roots in finite fields." In 18th Annual Symposium on Foundations of Computer Science (sfcs 1977), pp. 175-178. IEEE Computer Society, 1977．一个更易读的介绍可见于 Cao, Zhengjun, Qian Sha, and Xiao Fan. "Adleman-Manders-Miller root extraction method revisited." In International Conference on Information Security and Cryptology, pp. 77-85. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011.

[49] 由于这一算法要求 k 是素数，所以最差情形中，它需要对 $\varphi(m)$ 的最大素因子 p 求 a 模 m 的 p 次方根．这一过程中，需要对 p 次本原单位根求 a 模 m 的离散对数．即使应用 BSGS 算法，这一过程也需要 $O(\sqrt{p})$ 时间．但是，论文 Fouvry, Étienne. "Theoreme de Brun-Titchmarsh; application au theoreme de Fermat." Inventiones mathematicae 79, no. 2 (1985): 383-407 指出，存在正密度的素数 m，使得 $\varphi(m) = m - 1$ 的最大素因子 $p = \Omega(m^{2/3})$．这意味着这一算法的复杂度至少为 $\Omega(m^{1/3})$，劣于文中介绍的改良 Tonelli–Shanks 算法．

最左侧同余号利用了 欧拉定理→9.11.9@p1223 和如下同余关系：（$c \in \mathbf{Z}$）

$$k\ell = d\left(\frac{k}{d}\ell\right) = d\left(c\frac{\varphi(m)}{d} + 1\right) \equiv d \pmod{\varphi(m)}.$$

对于转化后的问题，考虑 d 的素因数分解：

$$d = \prod_{p\in\mathbf{P}} p^e.$$

可以从 $b = a^\ell$ 开始，对每个 $p^e \neq 1$，依次开 p^e 次方，最后就能得到 b 的 d 次方根，也就是 a 的 k 次方根.

最后，问题转化为如何求如下方程的解：

$$x^{p^e} \equiv b \pmod{m}.$$

不妨设 $\varphi(m) = p^s r$ 且 $p \perp r$. 设 $q \in \mathbf{N}_+$ 是方程 $qr \equiv -1 \pmod{p^e}$ 的解. 那么，因为 b 是 rp^{s-e} 次单位根，所以 b^{qr} 一定是 p^{s-e} 次单位根. 又设 ζ 是模 m 的 p^s 次本原单位根. 那么，ζ^{p^e} 是 p^{s-e} 次本原单位根，进而存在 $h \in \mathbf{N}$ 使得 $b^{qr} \equiv \zeta^{hp^e} \pmod{m}$. 所以，直接验证可知

$$x \equiv b^{(qr+1)/p^e}\zeta^{-h} \pmod{m}$$

是 b 模 m 的 p^e 次方根.

为了计算 x，需要找到模 m 的一个 p 次非剩余 η. 为此，由前文性质，只需要随机 $\eta \perp m$ 并验证 $\eta^{\varphi(m)/p} \bmod m \neq 1$ 即可. 这样的数的密度是

$$\frac{\varphi(m)}{m}\left(1 - \frac{1}{p}\right) \geq \frac{1}{4}.$$

因此，期望随机不超过 4 个整数就能找到它. 注意到，$\eta^{rp^{s-1}} \not\equiv 1 \pmod{m}$ 且 $\eta^{rp^s} \equiv 1 \pmod{m}$，所以，如果设 $\zeta = \eta^r \bmod m$ 和 $\xi = \eta^{rp^{s-1}} \bmod m$，那么它们分别是 p^s 次和 p 次本原单位根.

最后，需要计算 $h \in \mathbf{N}$. 显然，可以取 $h < p^{s-e}$. 考虑 h 的 p 进制表示：

$$h = \sum_{j=0}^{s-e-1} h_j p^j = h_0 + h_1 p + h_2 p^2 + \cdots.$$

逐位计算这些数位. 当前 j 个数位都计算完成时，必然有

$$\left(b^{qr}\zeta^{-p^e(h_0+h_1p+\cdots+h_{j-1}p^{j-1})}\right)^{p^{s-e-j-1}} \equiv \zeta^{h_j p^{s-1}} \equiv \xi^{h_j} \pmod{m}.$$

故而，h_j 可以通过计算关于 ξ 的离散对数求出. 为了获得更好的时间复杂度，需要使用 BSGS 算法. 总共需要计算 $(s-e)$ 次离散对数，设预处理 B 个 ξ 的幂次，则单次求解离散对数的时间复杂度为 $O(p/B)$，总的时间复杂度为

$$O\left(B + (s-e)\frac{p}{B}\right).$$

当 $B = \sqrt{(s-e)p}$ 时，总的时间复杂度最低，为 $O\left(\sqrt{(s-e)p}\right)$. 得到 h 之后，代入前文 x 的表达式，就可以找到一个特解.

时间复杂度

这一算法的时间复杂度为 $O(m^{1/4+\varepsilon})$. 本节讨论复杂度时，总是假设单次乘法需要 $O(1)$ 时间，且计算幂次时，总是应用欧拉定理降幂，则涉及的单个幂的计算总是可以在 $O(\log m)$ 时间内完成.

先考虑单个 p^e 次方根的计算. 找到 p 次非剩余只需要验证期望 $O(1)$ 个数，总时间复杂度为 $O(\log m)$. 计算 $s, r, \zeta, \eta, b^{qr}$ 各只需要 $O(\log m)$ 时间. 计算 h 时，单个数位需要通过 $O(\log m)$ 时间计算幂次，总共 $(s-e)$ 位，故而总的时间复杂度为 $O((s-e)\log m)$. 前文已经说明，计算离散对数的部分预处理和 $(s-e)$ 次查询的总时间为 $O\left(\sqrt{(s-e)p}\right)$. 因为 $s - e \in O(\log m)$，所以单个 p^e 次方根的计算的时间复杂度为 $O(p^{1/2+\varepsilon})$. 特别地，当 $s = e$ 时，时间复杂度可以进一步减少为 $O(\log m)$.

进而，可以考虑算法总的时间复杂度．计算 $\varphi(m), d, \ell$ 的时间复杂度均为 $O(\log m)$．紧接着需要做素因数分解 $d = \prod_p p^e$，这一步利用 Pollard Rho 算法→9.11.7@p1209 可以在 $O(m^{1/4})$ 时间内完成．最后，依次求 p^e 次方根的总时间复杂度为

$$O\left(\sum_{e<s} p^{1/2+\varepsilon}\right).$$

由于满足 $e < s$ 的素因子 p 至少在 $\varphi(m)$ 中出现 2 次，必然有 $p < m^{1/2}$．故而，总时间复杂度为 $O(m^{1/4+\varepsilon})$．

事实上，在这一情景中，无需使用 Pollard Rho 算法分解素因数，仍然可以获得 $O(m^{1/4+\varepsilon})$ 的时间复杂度．具体来说，只需要对 d 暴力试除进行分解，并只枚举到不超过 $m^{1/4}$ 的素因子．设去除这些小素因子后得到的整数为 z．那么，对于 z 的素因子 $p > m^{1/4}$，必然有 $\nu_p(\varphi(m)) < 4$，其中，$\nu_p(n)$ 表示 n 的素因数分解中 p 的次数．由于只需要考虑

$$1 \le e = \nu_p(d) < s = \nu_p(\varphi(m)) < 4$$

的情形，满足该条件的素因子 p 至多只能有一个；否则，它们在 $\varphi(m)$ 中的次数都不小于 2，总的乘积必然超过 m．要分离出这个（可能存在的）唯一的大素因子，只需要计算

$$p^\star = \gcd\left(z, \frac{\varphi(m)}{z}\right) = \prod_{p \mid z,\ \nu_p(d) < \nu_p(\varphi(m))} p^{\min\{\nu_p(d), \nu_p(\varphi(m)) - \nu_p(d)\}}.$$

枚举 $\nu_p(d), \nu_p(\varphi(m))$ 的所有可能性可知，乘积中 p 的次数一定是 1，因此这样算出来的就是唯一的大素因子 $p^\star$（如果存在的话）．至于剩余的部分 $z/p^\star$，因为其中只能包含若干满足 $e = s$ 的素因子，所以无需继续分解．

求素数模 k 次方根的参考实现如下：

模板题 Library Checker - Kth Root (Mod)[3] 参考实现

```cpp
#include <algorithm>
#include <cmath>
#include <iostream>
#include <random>
#include <tuple>
#include <unordered_map>
#include <vector>

std::mt19937 rng(std::random_device{}());

// Binary exponentiation.
int pow(int a, int b, int m = 0) {
  int res = 1;
  for (; b; b >>= 1) {
    if (b & 1) res = m ? (long long)res * a % m : res * a;
    a = m ? (long long)a * a % m : a * a;
  }
  return res;
}

// Find a P-th non-residue mod M.
int non_residue(int p, int m, int phi) {
  std::uniform_int_distribution<int> dis(1, m - 1);
  while (true) {
    int c = dis(rng);
```

```
    if (pow(c, phi / p, m) != 1) return c;
  }
  return -1;
}

// Euclidean Algorithm.
int gcd(int a, int b) { return b ? gcd(b, a % b) : a; }

// Extended Euclidean Algorithm.
int ex_gcd(int a, int b, int& x, int& y) {
  if (!b) {
    x = 1;
    y = 0;
    return a;
  } else {
    int d = ex_gcd(b, a % b, y, x);
    y -= a / b * x;
    return d;
  }
}

// Returns the modular inverse of A modulo M.
// Assumes that gcd(A, M) = 1, so the inverse exists.
int inv(int a, int m) {
  int x, y;
  ex_gcd(a, m, x, y);
  return (x % m + m) % m;
}

// Subroutine: Find a P^E-th root of A mod M.
int peth_root_mod_m(int p, int e, int a, int m, int phi) {
  int s = 0, r = phi, pe = pow(p, e);
  for (; r % p == 0; r /= p, ++s);
  int q = pe - inv(r, pe);
  int ans = pow(a, ((long long)q * r + 1) / pe % phi, m);
  int eta = non_residue(p, m, phi);
  std::unordered_map<int, int> mp;
  int zeta = pow(eta, r, m);
  int xi = pow(eta, phi / p, m);
  // Precompute powers for BSGS.
  int B = std::sqrt((s - e) * p + 0.25l) + 1;
  int pB = p / B + 1;
  int po0 = pow(xi, pB, m);
  for (int j = 1, po1 = 1; j <= B; ++j) {
    po1 = (long long)po1 * po0 % m;
    mp[po1] = j;
  }
  // Compute p-adic digits of h.
  for (int j = 0; j < s - e; ++j) {
    int err = (long long)pow(ans, pe, m) * inv(a, m) % m;
    int xi_hj = pow(err, pow(p, s - e - j - 1), m);
    long long hj = 0;
```

```
    // BSGS query.
    for (int i = 1; i <= pB; ++i) {
      xi_hj = (long long)xi_hj * xi % m;
      if (mp.count(xi_hj)) {
        hj = mp[xi_hj] * pB - i;
        break;
      }
    }
    ans = (long long)ans * pow(zeta, phi - hj * pow(p, j) % phi, m) % m;
  }
  return ans;
}

// Find a K-th root of A modulo prime P.
int kth_root_mod_p(int k, int a, int p) {
  a %= p;
  if (k == 0) return a == 1 ? 0 : -1;
  if (a == 0) return 0;
  int d = gcd(k, p - 1);
  if (pow(a, (p - 1) / d, p) != 1) return -1;
  a = pow(a, inv(k / d, (p - 1) / d), p);
  for (int dp = 2; dp * dp <= d && dp * dp * dp * dp < p; ++dp) {
    if (d % dp == 0) {
      int de = 0;
      for (; d % dp == 0; d /= dp, ++de);
      a = peth_root_mod_m(dp, de, a, p, p - 1);
    }
  }
  if (d != 1) {
    int dp = gcd(d, (p - 1) / d), de = 0;
    if (dp != 1) {
      for (; d % dp == 0; d /= dp, ++de);
      a = peth_root_mod_m(dp, de, a, p, p - 1);
    }
    if (d != 1) a = peth_root_mod_m(d, 1, a, p, p - 1);
  }
  return a;
}

void solve() {
  int k, y, p;
  std::cin >> k >> y >> p;
  std::cout << kth_root_mod_p(k, y, p) << '\n';
}

int main() {
  int t;
  std::cin >> t;
  for (; t; --t) {
    solve();
  }
}
```

```
130
```

一般情形的处理

考虑一般的情形，仍然设模数 m 是素数幂 p^e，但是 $\gcd(a,m)>1$．如果 $a\equiv 0 \pmod m$，那么

$$x\equiv p^{\lceil e/k\rceil}\ell \pmod{p^e},\ \ell=0,1,\cdots,p^{e-\lceil e/k\rceil}-1$$

都是原方程的解．接下来，考察 $a\not\equiv 0 \pmod m$ 的情形．设 $a=p^sa'$ 且 $p\perp a'$．于是，设 $x=p^zx'$ 且 $p\perp x'$，就有

$$x^k=p^{kz}(x')^k\equiv p^sa' \pmod{p^e}.$$

由于 $(x')^k\perp p$，所以该式成立当且仅当 $kz=s$ 且 $(x')^k\equiv a' \pmod{p^{e-s}}$．当且仅当 $k\mid s$ 时，第一个方程有解 $z=\dfrac{s}{k}$；而第二个方程的求解已经解决．需要注意的是，因为第二个方程的通解的模数与原方程通解的模数并不相同，所以第二个方程的每一个解 x'，都对应原方程的若干解：

$$x\equiv p^{s/k}(x'+\ell p^{e-s}) \pmod{p^e},\ \ell=0,1,\cdots,p^{s-s/k}-1.$$

求解任一模数下全体 k 次方根的参考实现如下：

模板题 Luogu P5668【模板】N 次剩余[4] 参考代码

- 朴素算法

```cpp
#include <algorithm>
#include <cmath>
#include <iostream>
#include <tuple>
#include <unordered_map>
#include <vector>

struct PrimePower {
  int p, e, pe;

  PrimePower(int p, int e, int pe) : p(p), e(e), pe(pe) {}
};

// Factorization.
auto factorize(int n) {
  std::vector<PrimePower> ans;
  for (int x = 2; x * x <= n; ++x) {
    int e = 0, pe = 1;
    for (; n % x == 0; n /= x, ++e, pe *= x);
    if (e) ans.emplace_back(x, e, pe);
  }
  if (n > 1) ans.emplace_back(n, 1, n);
  return ans;
}

// Binary exponentiation.
int pow(int a, int b, int m = 0) {
  int res = 1;
  for (; b; b >>= 1) {
    if (b & 1) res = m ? (long long)res * a % m : res * a;
    a = m ? (long long)a * a % m : a * a;
```

```
  }
  return res;
}

// Find a primitive root modulo odd prime power.
int primitive_root(PrimePower pp) {
  std::vector<int> exp;
  int phi = pp.pe / pp.p * (pp.p - 1);
  for (auto factor : factorize(pp.p - 1)) {
    exp.push_back(phi / factor.p);
  }
  if (pp.e != 1) exp.push_back(phi / pp.p);
  int ans = 0;
  bool succ = false;
  while (!succ) {
    ++ans;
    succ = true;
    for (int b : exp) {
      if (pow(ans, b, pp.pe) == 1) {
        succ = false;
        break;
      }
    }
  }
  return ans;
}

// Discrete logarithm. (BSGS Algorithm)
int log(int g, int a, int m) {
  int b = std::sqrt(m + 0.25l) + 1;
  std::unordered_map<int, int> mp;
  int po0 = a % m, po1 = 1;
  for (int i = 0; i < b; ++i) {
    mp[po0] = i;
    po0 = (long long)po0 * g % m;
  }
  po0 = pow(g, b, m);
  for (int j = 1; j <= b; ++j) {
    po1 = (long long)po1 * po0 % m;
    if (mp.count(po1)) return j * b - mp[po1];
  }
  return -1;
}

// Extended Euclidean Algorithm.
int ex_gcd(int a, int b, int& x, int& y) {
  if (!b) {
    x = 1;
    y = 0;
    return a;
  } else {
    int d = ex_gcd(b, a % b, y, x);
```

```
    y -= a / b * x;
    return d;
  }
}

// Returns the modular inverse of A modulo M.
// Assumes that gcd(A, M) = 1, so the inverse exists.
int inv(int a, int m) {
  int x, y;
  ex_gcd(a, m, x, y);
  return (x % m + m) % m;
}

// Solves the linear congruence equation: Ax = B mod N.
// Return the least nonnegative solution and the common difference.
std::pair<int, int> solve_linear(int a, int b, int n) {
  int x, y;
  int d = ex_gcd(a, n, x, y);
  if (b % d) return {-1, -1};
  n /= d;
  x = ((long long)x * (b / d) % n + n) % n;
  return {x, n};
}

// Subroutine: Find all the K-th roots with a primitive root G known.
std::vector<int> calc(int g, int k, int a, int p, int pe) {
  int ind = log(g, a, pe);
  if (ind == -1) return {};
  int mm = p == 2 ? pe / 4 : pe / p * (p - 1);
  int y0, d;
  std::tie(y0, d) = solve_linear(k, ind, mm);
  if (y0 == -1) return {};
  int ans = pow(g, y0, pe), po = pow(g, d, pe);
  std::vector<int> res(mm / d);
  for (auto& x : res) {
    x = ans;
    ans = (long long)ans * po % pe;
  }
  return res;
}

// Find all the K-th roots of A modulo prime power P^E.
std::vector<int> kth_roots_mod_pe(int k, int a, PrimePower pp) {
  int p = pp.p, e = pp.e, pe = pp.pe;
  a %= pe;
  if (a == 0) {
    int d = pow(p, (e - 1) / k + 1);
    std::vector<int> res(pe / d);
    for (int i = 0; i < pe / d; ++i) {
      res[i] = i * d;
    }
    return res;
```

```
  }
  int s = 0;
  for (; a % p == 0; a /= p, ++s);
  if (s % k) return {};
  int psk = pow(p, s / k), pss = pow(p, s - s / k), pes = pow(p, e - s);
  std::vector<int> res;
  if (p != 2) {
    int g = primitive_root(PrimePower(p, e - s, pes));
    res = calc(g, k, a, p, pes);
  } else if (pes == 2) {
    res.push_back(a);
  } else if (k & 1) {
    int z = a % 4 == 3;
    a = z ? pes - a : a;
    res = calc(5, k, a, p, pes);
    if (z) {
      for (auto& x : res) x = pes - x;
    }
  } else {
    if (a % 4 == 3) return {};
    res = calc(5, k, a, p, pes);
    int m = res.size();
    res.reserve(m * 2);
    for (int i = 0; i < m; ++i) {
      res.push_back(pes - res[i]);
    }
  }
  int m = res.size();
  res.reserve(m * pss);
  for (int j = 1; j < pss; ++j) {
    for (int i = 0; i < m; ++i) {
      res.push_back(res.end()[-m] + pes);
    }
  }
  for (auto& x : res) x *= psk;
  return res;
}

// Find all the K-th roots of A modulo positive integer M.
std::vector<int> kth_roots_mod_m(int k, int a, int m) {
  auto factors = factorize(m);
  int m0 = 0;
  std::vector<std::vector<int>> sols;
  for (const auto& pp : factors) {
    sols.push_back(kth_roots_mod_pe(k, a, pp));
    if (sols.back().empty()) return {};
  }
  std::vector<int> ans;
  for (int i = 0; i < (int)factors.size(); ++i) {
    auto pp = factors[i];
    if (!i) {
```

```
      m0 = pp.pe;
      ans = sols[i];
    } else {
      long long m1 = pp.pe * inv(pp.pe, m0);
      long long m2 = m0 * inv(m0, pp.pe);
      m0 *= pp.pe;
      std::vector<int> _ans;
      _ans.reserve(ans.size() * sols[i].size());
      for (auto x : ans) {
        for (auto y : sols[i]) {
          _ans.push_back((m1 * x + m2 * y) % m0);
        }
      }
      ans.swap(_ans);
    }
  }
  return ans;
}

void solve() {
  int n, m, k;
  std::cin >> n >> m >> k;
  auto ans = kth_roots_mod_m(n, k, m);
  if (ans.empty()) {
    std::cout << 0 << '\n';
    return;
  }
  std::cout << ans.size() << '\n';
  std::sort(ans.begin(), ans.end());
  for (auto x : ans) std::cout << x << ' ';
  std::cout << '\n';
}

int main() {
  int t;
  std::cin >> t;
  for (; t; --t) {
    solve();
  }
}

```

- 改良 Tonelli–Shanks 算法

```
#include <algorithm>
#include <cmath>
#include <iostream>
#include <random>
#include <tuple>
#include <unordered_map>
#include <vector>

std::mt19937 rng(std::random_device{}());
```

```cpp

struct PrimePower {
  int p, e, pe;

  PrimePower(int p, int e, int pe) : p(p), e(e), pe(pe) {}
};

// Factorization.
auto factorize(int n) {
  std::vector<PrimePower> ans;
  for (int x = 2; x * x <= n; ++x) {
    int e = 0, pe = 1;
    for (; n % x == 0; n /= x, ++e, pe *= x);
    if (e) ans.emplace_back(x, e, pe);
  }
  if (n > 1) ans.emplace_back(n, 1, n);
  return ans;
}

// Binary exponentiation.
int pow(int a, int b, int m = 0) {
  int res = 1;
  for (; b; b >>= 1) {
    if (b & 1) res = m ? (long long)res * a % m : res * a;
    a = m ? (long long)a * a % m : a * a;
  }
  return res;
}

// Find a primitive root modulo odd prime power.
int primitive_root(PrimePower pp) {
  std::vector<int> exp;
  int phi = pp.pe / pp.p * (pp.p - 1);
  for (auto factor : factorize(pp.p - 1)) {
    exp.push_back(phi / factor.p);
  }
  if (pp.e != 1) exp.push_back(phi / pp.p);
  int ans = 0;
  bool succ = false;
  while (!succ) {
    ++ans;
    succ = true;
    for (int b : exp) {
      if (pow(ans, b, pp.pe) == 1) {
        succ = false;
        break;
      }
    }
  }
  return ans;
}

```

```cpp
// Euclidean Algorithm.
int gcd(int a, int b) { return b ? gcd(b, a % b) : a; }

// Extended Euclidean Algorithm.
int ex_gcd(int a, int b, int& x, int& y) {
  if (!b) {
    x = 1;
    y = 0;
    return a;
  } else {
    int d = ex_gcd(b, a % b, y, x);
    y -= a / b * x;
    return d;
  }
}

// Returns the modular inverse of A modulo M.
// Assumes that gcd(A, M) = 1, so the inverse exists.
int inv(int a, int m) {
  int x, y;
  ex_gcd(a, m, x, y);
  return (x % m + m) % m;
}

// Find a P-th non-residue mod M.
int non_residue(int p, int m, int phi) {
  std::uniform_int_distribution<int> dis(1, m - 1);
  while (true) {
    int c = dis(rng);
    if (gcd(c, m) == 1 && pow(c, phi / p, m) != 1) return c;
  }
  return -1;
}

// Subroutine: Find a P^E-th root of A mod M.
int peth_root_mod_m(int p, int e, int a, int m, int phi) {
  if (m == 2) return 1;
  int s = 0, r = phi, pe = pow(p, e);
  for (; r % p == 0; r /= p, ++s);
  int q = pe - inv(r, pe);
  int ans = pow(a, ((long long)q * r + 1) / pe % phi, m);
  int eta = non_residue(p, m, phi);
  std::unordered_map<int, int> mp;
  int zeta = pow(eta, r, m);
  int xi = pow(eta, phi / p, m);
  // Precompute powers for BSGS.
  int B = std::sqrt((s - e) * p + 0.25l) + 1;
  int pB = pe / B + 1;
  int po0 = pow(xi, pB, m);
  for (int j = 1, po1 = 1; j <= B; ++j) {
    po1 = (long long)po1 * po0 % m;
    mp[po1] = j;
```

```
  }
  // Compute p-adic digits of h.
  for (int j = 0; j < s - e; ++j) {
    int err = (long long)pow(ans, pe, m) * inv(a, m) % m;
    int xi_hj = pow(err, pow(p, s - e - j - 1), m);
    long long hj = 0;
    // BSGS query.
    for (int i = 1; i <= pB; ++i) {
      xi_hj = (long long)xi_hj * xi % m;
      if (mp.count(xi_hj)) {
        hj = mp[xi_hj] * pB - i;
        break;
      }
    }
    ans = (long long)ans * pow(zeta, phi - hj * pow(p, j) % phi, m)
% m;
  }
  return ans;
}

// Find a K-th root of A modulo prime P^E.
int kth_root_mod_pe(int k, int a, int pe, int phi) {
  a %= pe;
  if (k == 0) return a == 1 ? 0 : -1;
  if (a == 0) return 0;
  int d = gcd(k, phi);
  if (pow(a, phi / d, pe) != 1) return -1;
  a = pow(a, inv(k / d, phi / d), pe);
  for (int dp = 2; dp * dp <= d && dp * dp * dp * dp < pe; ++dp) {
    if (d % dp == 0) {
      int de = 0;
      for (; d % dp == 0; d /= dp, ++de);
      a = peth_root_mod_m(dp, de, a, pe, phi);
    }
  }
  if (d != 1) {
    int dp = gcd(d, phi / d), de = 0;
    if (dp != 1) {
      for (; d % dp == 0; d /= dp, ++de);
      a = peth_root_mod_m(dp, de, a, pe, phi);
    }
    if (d != 1) a = peth_root_mod_m(d, 1, a, pe, phi);
  }
  return a;
}

// Subroutine: Find all the K-th roots with a primitive root G known.
std::vector<int> calc(int g, int k, int a, int p, int pe) {
  int mm = p == 2 ? pe / 4 : pe / p * (p - 1);
  int ans = kth_root_mod_pe(k, a, pe, mm);
  if (ans == -1) return {};
  int d = mm / gcd(k, mm);
```

```
  int po = pow(g, d, pe);
  std::vector<int> res(mm / d);
  for (auto& x : res) {
    x = ans;
    ans = (long long)ans * po % pe;
  }
  return res;
}

// Find all the K-th roots of A modulo prime power P^E.
std::vector<int> kth_roots_mod_pe(int k, int a, PrimePower pp) {
  int p = pp.p, e = pp.e, pe = pp.pe;
  a %= pe;
  if (a == 0) {
    int d = pow(p, (e - 1) / k + 1);
    std::vector<int> res(pe / d);
    for (int i = 0; i < pe / d; ++i) {
      res[i] = i * d;
    }
    return res;
  }
  int s = 0;
  for (; a % p == 0; a /= p, ++s);
  if (s % k) return {};
  int psk = pow(p, s / k), pss = pow(p, s - s / k), pes = pow(p, e
- s);
  std::vector<int> res;
  if (p != 2) {
    int g = primitive_root(PrimePower(p, e - s, pes));
    res = calc(g, k, a, p, pes);
  } else if (pes == 2) {
    res.push_back(a);
  } else if (k & 1) {
    int z = a % 4 == 3;
    a = z ? pes - a : a;
    res = calc(5, k, a, p, pes);
    if (z) {
      for (auto& x : res) x = pes - x;
    }
  } else {
    if (a % 4 == 3) return {};
    res = calc(5, k, a, p, pes);
    int m = res.size();
    res.reserve(m * 2);
    for (int i = 0; i < m; ++i) {
      res.push_back(pes - res[i]);
    }
  }
  int m = res.size();
  res.reserve(m * pss);
  for (int j = 1; j < pss; ++j) {
    for (int i = 0; i < m; ++i) {
```

```cpp
      res.push_back(res.end()[-m] + pes);
    }
  }
  for (auto& x : res) x *= psk;
  return res;
}

// Find all the K-th roots of A modulo positive integer M.
std::vector<int> kth_roots_mod_m(int k, int a, int m) {
  auto factors = factorize(m);
  int m0 = 0;
  std::vector<std::vector<int>> sols;
  for (const auto& pp : factors) {
    sols.push_back(kth_roots_mod_pe(k, a, pp));
    if (sols.back().empty()) return {};
  }
  std::vector<int> ans;
  for (int i = 0; i < (int)factors.size(); ++i) {
    auto pp = factors[i];
    if (!i) {
      m0 = pp.pe;
      ans = sols[i];
    } else {
      long long m1 = pp.pe * inv(pp.pe, m0);
      long long m2 = m0 * inv(m0, pp.pe);
      m0 *= pp.pe;
      std::vector<int> _ans;
      _ans.reserve(ans.size() * sols[i].size());
      for (auto x : ans) {
        for (auto y : sols[i]) {
          _ans.push_back((m1 * x + m2 * y) % m0);
        }
      }
      ans.swap(_ans);
    }
  }
  return ans;
}

void solve() {
  int n, m, k;
  std::cin >> n >> m >> k;
  auto ans = kth_roots_mod_m(n, k, m);
  if (ans.empty()) {
    std::cout << 0 << '\n';
    return;
  }
  std::cout << ans.size() << '\n';
  std::sort(ans.begin(), ans.end());
  for (auto x : ans) std::cout << x << ' ';
  std::cout << '\n';
}
```

```
268
269 int main() {
270   int t;
271   std::cin >> t;
272   for (; t; --t) {
273     solve();
274   }
275 }
276
```

参考资料与注释

- 冯克勤. 初等数论及其应用.
- Root of unity modulo n - Wikipedia[5]
- No.981 一般冪乗根 解説 by 37zigen[6]

[1] Artin 互反律

[2] Library Checker - Kth Root (Mod)

[3] Library Checker - Kth Root (Mod)

[4] Luogu P5668【模板】N 次剩余

[5] Root of unity modulo n - Wikipedia

[6] No.981 一般冪乗根 解説 by 37zigen

9.11.21 数论分块

数论分块可以快速计算一些形如

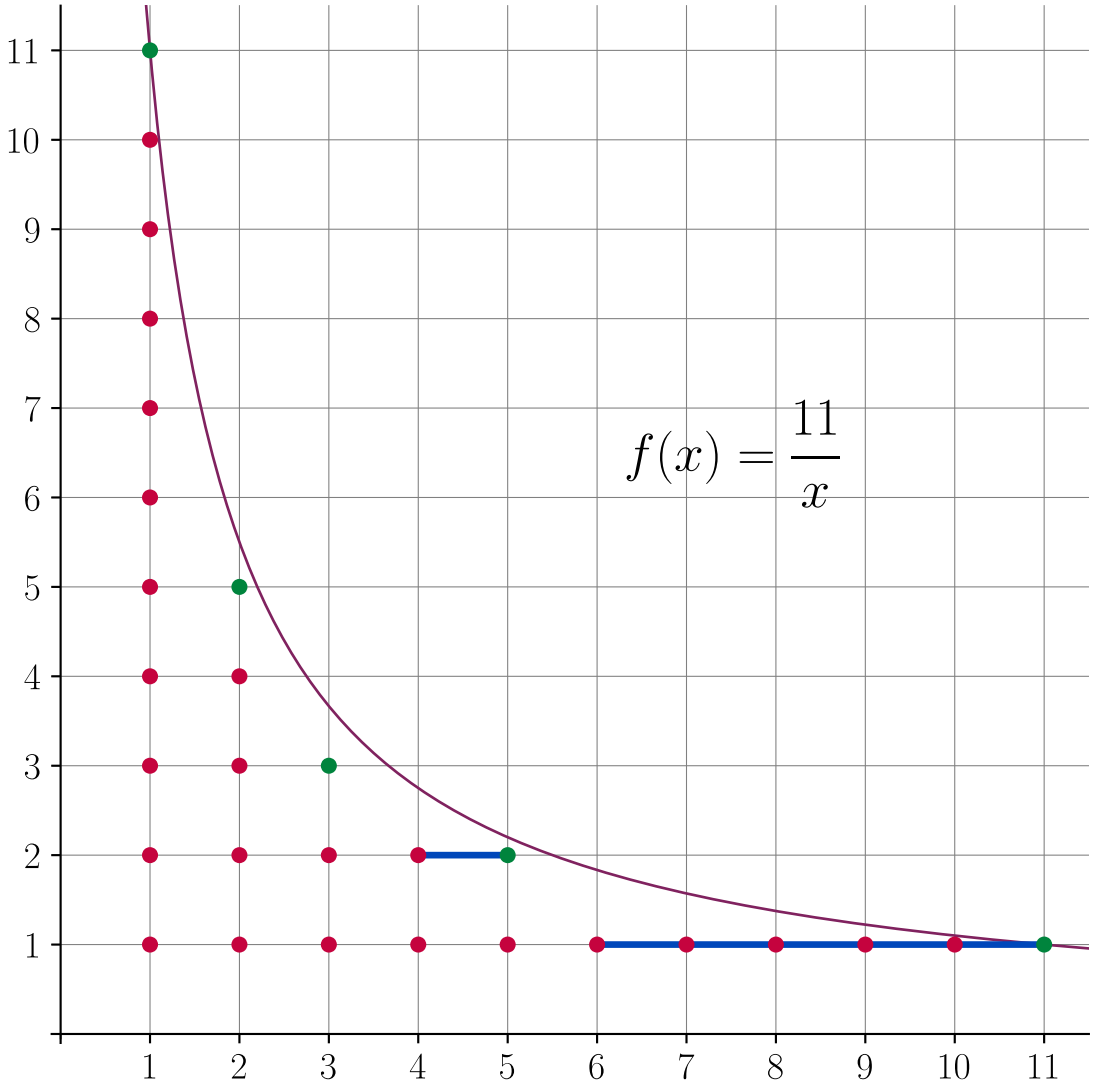
$$\sum_{i=1}^{n} f(i)g\left(\left\lfloor\frac{n}{i}\right\rfloor\right)$$

的和式. 如果可以在 $O(1)$ 时间内计算出 $\sum_{i=l}^{r} f(i)$ 或已经预处理出 f 的前缀和，数论分块就可以在 $O(\sqrt{n})$ 时间内计算出上述和式的值.

数论分块常与 莫比乌斯反演→9.11.23@p1331 等技巧结合使用.

思路

首先，本文通过一个简单的例子来说明数论分块的思路. 假设要计算如图所示的双曲线下的整点个数:



这相当于在计算和式：

$$\sum_{i=1}^{11}\left\lfloor\frac{11}{i}\right\rfloor.$$

这是前文所示和式在 $f(k)=1,\ g(k)=k$ 时的特殊情况.

最简单的做法当然是逐列计算然后求和，但是这样需要计算第 $i=1,2,\cdots,11$ 列中每列整点的个数. 观察图示可以发现，这些整点列可以分成 5 块，每块内点列的高度是一致的，形成一个矩形点阵. 所以，只要能够知道这些块的宽度，就能够通过计算这些矩形块的大小快速完成统计.

这就是数论分块的基本思路.

性质

本节讨论关于双曲线 $y=\dfrac{n}{x}$ 下整点分块的若干结论. 具体地，需要将 $1\sim n$ 之间的整数按照 $\left\lfloor\dfrac{n}{i}\right\rfloor$ 的取值分为若干块. 设

$$D(n)=\left\{\left\lfloor\frac{n}{i}\right\rfloor:1\le i\le n,\ i\in\mathbf{N}_+\right\}.$$

这就是 $\left\lfloor\dfrac{n}{i}\right\rfloor$ 所有可能的取值集合.

首先，这样的不同取值只有 $O(\sqrt{n})$ 个. 所以，数论分块得到的块的数目只有 $O(\sqrt{n})$ 个.

性质 1

$|D(n)| \le 2\sqrt{n}$.

证明

分两种情况讨论：

- 当 $i \le \sqrt{n}$ 时，i 的取值至多 $\sqrt{n}$ 个，所以 $\left\lfloor \frac{n}{i} \right\rfloor$ 的取值也至多只有 $\sqrt{n}$ 个.
- 当 $i > \sqrt{n}$ 时，$\left\lfloor \frac{n}{i} \right\rfloor \le \frac{n}{i} < \sqrt{n}$ 也至多只有 $\sqrt{n}$ 个取值.

因此，所有可能取值的总数 $|D(n)| \le 2\sqrt{n}$.

通过更细致的分析，实际上可以精确地描述集合 $D(n)$ 及其大小.

性质 2

设 $s = \lfloor \sqrt{n} \rfloor$. 集合 $D(n)$ 中的元素从小到大依次是

$$1 < 2 < \cdots < s-1 < s \le \left\lfloor \frac{n}{s} \right\rfloor < \left\lfloor \frac{n}{s-1} \right\rfloor < \cdots < \left\lfloor \frac{n}{2} \right\rfloor < \left\lfloor \frac{n}{1} \right\rfloor = n.$$

进而，$|D(n)| = \lfloor \sqrt{4n+1} \rfloor - 1$.

证明

首先，对于 $1 \le i \le s$，可以证明 $\left\lfloor \frac{n}{\lfloor n/i \rfloor} \right\rfloor = i$. 令 $d = \left\lfloor \frac{n}{i} \right\rfloor$，需要证明 $\left\lfloor \frac{n}{d} \right\rfloor = i$. 写成不等式，这相当于已知 $d \le \frac{n}{i} < d+1$，需要证明 $i \le \frac{n}{d} < i+1$. 已知条件可以写作 $i \le \frac{n}{d} < i + \frac{i}{d}$，因此，只需要证明 $\frac{i}{d} \le 1$，亦即 $i \le d = \left\lfloor \frac{n}{i} \right\rfloor$. 这等价于 $i \le \frac{n}{i}$，这对于所有 $1 \le i \le s \le \sqrt{n}$ 都成立.

这一结果实际上说明，映射 $i \mapsto \left\lfloor \frac{n}{i} \right\rfloor$ 构成集合 $\{i : 1 \le i \le s\}$ 和集合 $\left\{ \left\lfloor \frac{n}{i} \right\rfloor : 1 \le i \le s \right\}$ 之间的双射. 由于 i 各不相同，$\left\lfloor \frac{n}{i} \right\rfloor$ 也各不相同. 两个集合唯一有可能相同的元素就是 s 与 $\left\lfloor \frac{n}{s} \right\rfloor$. 所以，$|D(n)| = 2s - [s = \lfloor n/s \rfloor]$.

要得到 $|D(n)|$ 最终的表达式，需要考察何时 $s = \left\lfloor \frac{n}{s} \right\rfloor$ 成立.

- 当 $s = \left\lfloor \frac{n}{s} \right\rfloor$ 时，总有 $s \le \frac{n}{s} < s+1$，亦即 $s^2 \le n < s^2 + s$. 此时，有 $4s^2 + 1 \le 4n + 1 < 4s^2 + 4s + 1 = (2s+1)^2$. 又因为 $4n+1$ 一定是奇数，左侧可以等价地松弛到 $4s^2$，所以该条件就等价于 $2s \le \sqrt{4n+1} < 2s+1$，亦即 $\lfloor \sqrt{4n+1} \rfloor = 2s$.
- 当 $s < \left\lfloor \frac{n}{s} \right\rfloor$ 时，总有 $s+1 \le \frac{n}{s}$，亦即 $s^2 + s \le n$. 又因为 $n < (s+1)^2$，就有 $s^2 + s \le n < (s+1)^2$. 这就等价于 $(2s+1)^2 \le 4n + 1 < 4(s+1)^2 + 1$. 再次利用 $4n+1$ 是奇数这一点，右侧可以等价地收紧到 $4(s+1)^2$，所以该条件就等价于 $2s+1 \le \sqrt{4n+1} < 2s+2$，亦即 $\lfloor \sqrt{4n+1} \rfloor = 2s+1$.

总结这两种情形，就得到 $|D(n)| = \lfloor \sqrt{4n+1} \rfloor - 1$ 成立.

然后，单个块的左右端点都很容易确定.

性质 3

对于 $d \in D(n)$，所有满足 $\left\lfloor \frac{n}{i} \right\rfloor = d$ 的整数 i 的取值范围为

$$\left\lfloor \frac{n}{d+1} \right\rfloor + 1 \le i \le \left\lfloor \frac{n}{d} \right\rfloor.$$

证明

因为 $\left\lfloor \frac{n}{i} \right\rfloor = d$ 相当于不等式

$$d \le \frac{n}{i} < d+1.$$

这进一步等价于

$$\frac{n}{d+1} < i \le \frac{n}{d}.$$

利用 $i \in \mathbf{N}_+$ 这一点，可以对该不等式取整，它就等价于

$$\left\lfloor \frac{n}{d+1} \right\rfloor + 1 \le i \le \left\lfloor \frac{n}{d} \right\rfloor.$$

这一性质还体现了图像的对称性：每个块的右端点（前文图中的绿点）的集合恰为 $D(n)$．这很容易理解，因为整个图像关于直线 $y=x$ 是对称的．

除了这些性质外，集合 $D(n)$ 还具有良好的递归性质：

性质 4

对于 $m \in D(n)$，有 $D(m) \subseteq D(n)$．

证明

设 $m = \left\lfloor \frac{n}{k} \right\rfloor$，那么，因为对于所有 $i \in \mathbf{N}_+$，都有

$$\left\lfloor \frac{m}{i} \right\rfloor = \left\lfloor \frac{\lfloor n/k \rfloor}{i} \right\rfloor = \left\lfloor \frac{n}{ki} \right\rfloor \in D(n),$$

所以，$D(m) \subseteq D(n)$．其中，第二个等号用到了 取整函数→9.11.1@p1160 关于嵌套分式的性质．

前文已经提到，$D(n)$ 既是每块中 $\left\lfloor \frac{n}{i} \right\rfloor$ 的取值集合，也是每块的右端点集合．这意味着，如果要递归地应用数论分块（即函数在 n 处的值依赖于它在 $m \in D(n) \setminus \{n\}$ 处的值），那么在整个计算过程中所涉及的取值集合和右端点集合，其实都是 $D(n)$．一个典型的例子是 杜教筛→9.11.24@p1353．

过程

利用上一节叙述的结论，就得到数论分块的具体过程．

为计算和式

$$\sum_{i=1}^{n} f(i) g\left(\left\lfloor \frac{n}{i} \right\rfloor\right)$$

的取值，可以依据 $\left\lfloor \frac{n}{i} \right\rfloor$ 的取值将标号 $i = 1, 2, \cdots, n$ 分块．因为 $\left\lfloor \frac{n}{i} \right\rfloor$ 取值相同的标号是一段连续整数 $[l, r]$，所以该块中和式的取值为

$$\left(\sum_{i=l}^{r} f(i)\right) \cdot g\left(\left\lfloor \frac{n}{l} \right\rfloor\right).$$

为了快速计算该和式，通常需要能够快速计算左侧关于 f 的和式．有些时候，该和式的表达式是已知的，可以在 $O(1)$ 时间内完成单次计算；有些时候，可以预处理出它的前缀和，仍然可以在 $O(1)$ 时间内完成单次查询．

在顺次计算每块左右端点时，当前块的左端点 l 就等于前一块的右端点再加 1，而当前块的右端点就等于 $\left\lfloor \frac{n}{\lfloor n/l \rfloor} \right\rfloor$．由此，可以得到如下伪代码：

Algorithm $\mathrm{Sum}(f, g, n)$:
Input. $n,\ s(k) = \sum_{i=1}^{k} f(i),\ g(k)$.
Output. $S(n) = \sum_{i=1}^{n} f(i)g(\lfloor n/i \rfloor)$.
Method.

```
l ← 1
result ← 0
while l ≤ n do
    r ← ⌊ n / ⌊n/l⌋ ⌋
    result ← result + (s(r) − s(l − 1)) · g(⌊ n / l ⌋)
    l ← r + 1
end while
return result
```

假设单次计算 $s(\cdot)$ 的时间复杂度为 $O(1)$，则整个过程的时间复杂度为 $O(\sqrt{n})$.

扩展

前文讨论的是数论分块的最常见也最基本的形式．本节进一步讨论数论分块的扩展形式.

向上取整的数论分块

数论分块可以用于计算含有向上取整的和式:

$$\sum_{i=1}^{n} f(i)g\left(\left\lceil \frac{n}{i} \right\rceil\right).$$

因为 $\left\lceil \dfrac{n}{i} \right\rceil = \left\lfloor \dfrac{n-1}{i} \right\rfloor + 1$，该和式可以转化为向下取整的情形:

$$f(n)g(1) + \sum_{i=1}^{n-1} f(i)g\left(\left\lfloor \frac{n-1}{i} \right\rfloor + 1\right).$$

注意到求和的上限发生了变化，以及 $i = n$ 时单独的一项.

多维数论分块

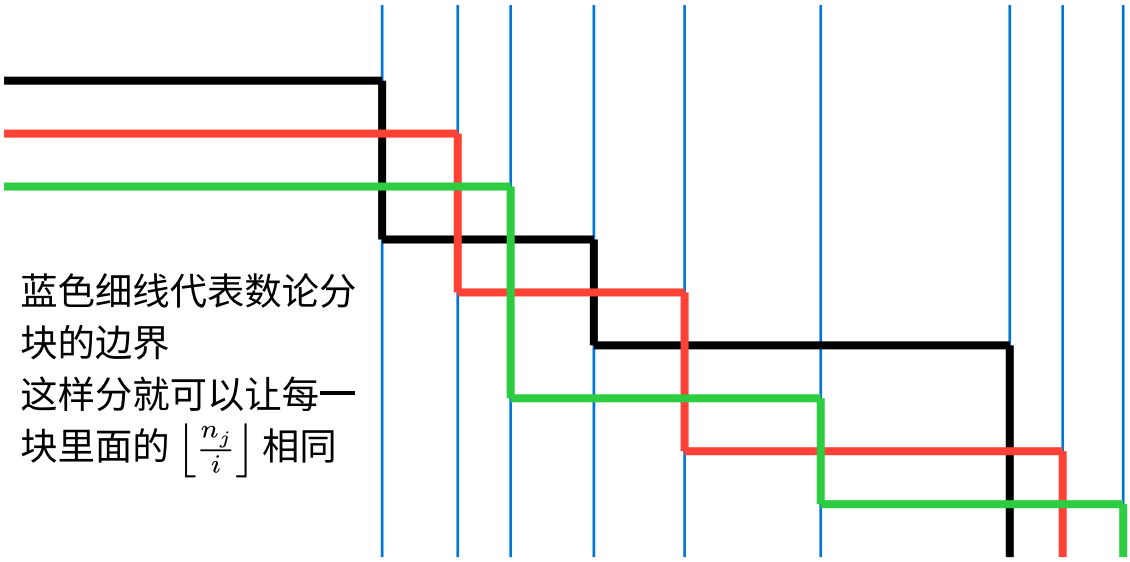
数论分块还可以用于处理包含不止一个取整式的和式:

$$\sum_{i=1}^{n} f(i)g\left(\left\lfloor \frac{n_1}{i} \right\rfloor, \left\lfloor \frac{n_2}{i} \right\rfloor, \cdots, \left\lfloor \frac{n_m}{i} \right\rfloor\right).$$

其中，通常有 $n \le \min\{n_1, n_2, \cdots, n_m\}$，否则需要单独处理某个 $\lfloor n_j/i \rfloor = 0$ 的情形．为了应用数论分块的思想，需要保证每块中所有取整式 $\left\lfloor \dfrac{n_1}{i} \right\rfloor, \left\lfloor \dfrac{n_2}{i} \right\rfloor, \cdots, \left\lfloor \dfrac{n_m}{i} \right\rfloor$ 的取值都不发生变化，也就是说，多维的块应当是所有一维的块的交集．为此，对于已知的左端点 l，相应的右端点为

$$\min\left\{\left\lfloor \frac{n_1}{\lfloor n_1/l \rfloor} \right\rfloor, \left\lfloor \frac{n_2}{\lfloor n_2/l \rfloor} \right\rfloor, \cdots, \left\lfloor \frac{n_m}{\lfloor n_m/l \rfloor} \right\rfloor\right\}.$$

也就是说，对于所有一维分块的右端点取最小值，作为多维分块的右端点．可以借助下图理解:



较为常见的是二维形式，此时可将前述伪代码中 $r \leftarrow \left\lfloor \dfrac{n}{\lfloor n/l \rfloor} \right\rfloor$ 替换成

$$r \leftarrow \min\left\{ \left\lfloor \frac{n_1}{\lfloor n_1/l \rfloor} \right\rfloor, \left\lfloor \frac{n_2}{\lfloor n_2/l \rfloor} \right\rfloor \right\}.$$

任意指数数论分块

数论分块可以用于含有任意指数的取整式的和式计算：

$$\sum_{i=1}^{\lfloor n^{\alpha/\beta} \rfloor} f(i) g\left(\left\lfloor \frac{n^\alpha}{i^\beta} \right\rfloor \right).$$

其中，α, β 为正实数．本文讨论的基本形式中，$\alpha = \beta = 1$.

性质

对于正整数 n 和正实数 α, β，设

$$D(n, \alpha, \beta) = \left\{ \left\lfloor \frac{n^\alpha}{i^\beta} \right\rfloor : i = 1, 2, \cdots, \lfloor n^{\alpha/\beta} \rfloor \right\}.$$

那么，有

1. $|D(n, \alpha, \beta)| \le 2n^{\alpha/(1+\beta)}$.
2. 对于 $d \in D(n, \alpha, \beta)$，使得 $\left\lfloor \dfrac{n^\alpha}{i^\beta} \right\rfloor = d$ 成立的 i 的取值范围为 $\left\lfloor \dfrac{n^{\alpha/\beta}}{(d+1)^{1/\beta}} \right\rfloor + 1 \le i \le \left\lfloor \dfrac{n^{\alpha/\beta}}{d^{1/\beta}} \right\rfloor$.

证明

对于第一点，分两种情况：

- 当 $i \le \dfrac{n^\alpha}{i^\beta}$ 时，有 $i \le n^{\alpha/(1+\beta)}$，所以 $\left\lfloor \dfrac{n^\alpha}{i^\beta} \right\rfloor$ 至多有 $n^{\alpha/(1+\beta)}$ 种取值.
- 当 $i > \dfrac{n^\alpha}{i^\beta}$ 时，有 $i > n^{\alpha/(1+\beta)}$，进而有 $\dfrac{n^\alpha}{i^\beta} < n^{\alpha/(1+\beta)}$，所以 $\left\lfloor \dfrac{n^\alpha}{i^\beta} \right\rfloor$ 也至多只有 $n^{\alpha/(1+\beta)}$ 种取值.

综合两种情形，就有 $|D(n, \alpha, \beta)| \le 2n^{\alpha/(1+\beta)}$.

对于第二点，$\left\lfloor \dfrac{n^\alpha}{i^\beta} \right\rfloor = d$ 就等价于

$$d \le \frac{n^\alpha}{i^\beta} < d+1 \iff \frac{n^{\alpha/\beta}}{(d+1)^{1/\beta}} < i \le \frac{n^{\alpha/\beta}}{d^{1/\beta}}.$$

对该不等式取整，就得到第二个命题.

利用这些性质，就可以在 $O(n^{\alpha/(1+\beta)})$ 的时间复杂度下实现对任意指数的数论分块.

例子

例如，对于 $\alpha=\beta=1/2$ 时的如下和式

$$\sum_{i=1}^{n} f(i)g\left(\left\lfloor\sqrt{\frac{n}{i}}\right\rfloor\right),$$

可以通过数论分块在 $O(n^{1/3})$ 时间内解决. 已知块的左端点为 l 时，可以计算右端点为 $r=\left\lfloor\frac{n}{\lfloor\sqrt{n/l}\rfloor^2}\right\rfloor$.

例题

UVa11526 H(n)[1]

T 组数据，每组一个整数 n. 对于每组数据，输出 $\sum_{i=1}^{n}\left\lfloor\frac{n}{i}\right\rfloor$.

解答

根据前文分析，可以对于每一块相同的 $\left\lfloor\frac{n}{i}\right\rfloor$ 一起计算. 时间复杂度为 $O(T\sqrt{n})$.

实现

```cpp
#include <iostream>

long long H(int n) {
  long long res = 0;  // 储存结果
  int l = 1, r;       // 块左端点与右端点
  while (l <= n) {
    r = n / (n / l);  // 计算当前块的右端点
    // 累加这一块的贡献到结果中。乘上 1LL 防止溢出
    res += 1LL * (r - l + 1) * (n / l);
    l = r + 1;  // 左端点移到下一块
  }
  return res;
}

int main() {
  std::ios::sync_with_stdio(false);
  std::cin.tie(nullptr);
  int t, n;
  std::cin >> t;
  while (t--) {
    std::cin >> n;
    std::cout << H(n) << '\n';
  }
  return 0;
}
```

```
26
```

Codeforces 1954E Chain Reaction[2]

有一排 n 只怪兽，每只怪兽初始血量为 a_i．一次攻击会使一段连续的存活的怪兽血量减 k，血量不大于 0 视作死亡．对于所有 k 求出击杀所有怪兽所需攻击次数．其中，$n, a_i \le 10^5$．

解答

令 $a_0 = 0$．假设击杀所有前 $(i-1)$ 只怪兽需要 $T(k, i-1)$ 次攻击，第 i 只怪兽的血量为 a_i．由于击杀第 $(i-1)$ 只怪兽时，需要攻击它 $\lceil a_{i-1}/k \rceil$ 次，这些攻击都可以延伸到第 i 只怪兽．因此，要击杀第 i 只怪兽，只需要再攻击 $\max\{0, \lceil a_i/k \rceil - \lceil a_{i-1}/k \rceil\}$ 次．由此，总的攻击次数为

$$T(k, n) = \sum_{i=1}^{n} \max\left\{0, \left\lceil \frac{a_i}{k} \right\rceil - \left\lceil \frac{a_{i-1}}{k} \right\rceil\right\}.$$

由于题目涉及的 n, k 都比较大，对每个 k 分别计算该和式并不可行．可以考虑对每个 $i = 1, 2, \cdots, n$，都维护数列 $\{T(k, i)\}_k$．初始时，设 $T(k, 0) \equiv 0$．假设数列 $\{T(k, i-1)\}_k$ 已知，考虑如何对它进行修改才能得到数列 $\{T(k, i)\}_k$．根据前文分析，只需要对数列的第 k 项增加 $\max\left\{0, \left\lceil \frac{a_i}{k} \right\rceil - \left\lceil \frac{a_{i-1}}{k} \right\rceil\right\}$ 即可．利用二维数论分块，可以将这一修改操作拆分成 $O(\sqrt{a_{i-1}} + \sqrt{a_i})$ 段区间修改操作，且每段区间上增加的值是固定的．最后得到的数列 $\{T(k, n)\}_k$ 就是答案．

由于题目涉及一系列区间加操作，且查询只发生在所有修改完成后．所以，可以通过维护差分序列进行区间加修改，最后通过求前缀和得到所求数列．总的时间复杂度为 $O(\max_i a_i + \sum_i \sqrt{a_i})$．本题也存在其他解法．

实现

```cpp
#include <algorithm>
#include <iostream>

constexpr int N = 100009;
int n, a[N], maxn;
long long ans[N];

int main() {
  std::ios::sync_with_stdio(false);
  std::cin.tie(nullptr);
  std::cin >> n;
  for (int i = 1; i <= n; ++i) {
    std::cin >> a[i];
    maxn = std::max(maxn, a[i]);
  }
  for (int i = 0; i < n; ++i)
    for (int l = 1, r;; l = r + 1) {
      r = std::min(l < a[i] ? (a[i] - 1) / ((a[i] - 1) / l) : N,
                   l < a[i + 1] ? (a[i + 1] - 1) / ((a[i + 1] - 1) / l)
                                : N);  // 二维数论分块
      if (r == N) break;
      int x = (a[i + 1] - 1) / l - std::max(a[i] - 1, 0) / l;
```

```
      if (x > 0) ans[l] += x, ans[r + 1] -= x;  // 累加贡献
    }
  ++ans[0];  // [a/l]=(a-1)/l+1的式子当a=0时不成立，需要修正
  for (int i = 1; i <= maxn; ++i)
    std::cout << (ans[i] += ans[i - 1]) << " \n"[i == maxn];
  return 0;
}

```

习题

- UVa11526 H(n)[3]
- Luogu P2261 CQOI2007 余数求和[4]
- Luogu P3455 POI2007 ZAP-Queries[5]

参考资料与注释

- 杜教筛的时空复杂度分析 by riteme[6]

外部参考资料

[1] UVa11526 H(n)

[2] Codeforces 1954E Chain Reaction

[3] UVa11526 H(n)

[4] Luogu P2261 CQOI2007 余数求和

[5] Luogu P3455 POI2007 ZAP-Queries

[6] 杜教筛的时空复杂度分析 by riteme

9.11.22 狄利克雷卷积

> **Authors:** billchenchina, c-forrest, CCXXXI, danielqfmai, Enter-tainer, Great-designer, HeRaNO, lychees, Menci, Nanarikom, ouuan, shuzhouliu, sshwy, Tiphereth-A

本文介绍 Dirichlet 卷积和 Dirichlet 生成函数.

Dirichlet 卷积

数论函数 $f(n)$ 和 $g(n)$ 的 **Dirichlet 卷积**（Dirichlet convolution），记作 $f*g$，定义为数论函数

$$(f*g)(n)=\sum_{k|n}f(k)g\left(\frac{n}{k}\right)=\sum_{k\ell=n}f(k)g(\ell).$$

Dirichlet 卷积是数论函数的重要运算. 数论函数的许多性质都是通过这个运算挖掘出来的.

例子

1. 单位函数 ε 是莫比乌斯函数 μ 和常数函数 1 的 Dirichlet 卷积：

   $$\varepsilon=\mu*1\iff\varepsilon(n)=\sum_{d|n}\mu(d).$$

2. 除数个数函数 τ 是常数函数 1 和它自身的 Dirichlet 卷积：

   $$\tau=1*1\iff\tau(n)=\sum_{d|n}1.$$

3. 除数和函数 σ 是恒等函数 id 和常数函数 1 的 Dirichlet 卷积：

   $$\sigma=\mathrm{id}*1\iff\sigma(n)=\sum_{d|n}d.$$

4. 欧拉函数 φ 是恒等函数 id 和莫比乌斯函数 μ 的 Dirichlet 卷积：

$$\varphi = \mathrm{id} * \mu \iff \varphi(n) = \sum_{d|n} d \cdot \mu\left(\frac{n}{d}\right).$$

莫比乌斯反演→9.11.23@p1331 就是利用 $\varepsilon = \mu * 1$ 对数论函数恒等式进行变形.

性质

Dirichlet 卷积具有一系列代数性质.

定理

设 f, g, h 都是数论函数. 那么，有：

1. **交换律**：$f * g = g * f$.
2. **结合律**：$(f * g) * h = f * (g * h)$.
3. **分配律**：$(f + g) * h = f * h + g * h$.
4. **单位元**：$f * \varepsilon = \varepsilon * f = f$，其中，$\varepsilon(n) = [n = 1]$ 是卷积单位元，$[\cdot]$ 是 Iverson 括号.
5. **逆元**：当且仅当 $f(1) \neq 0$ 时，存在 g 使得 $f * g = g * f = \varepsilon$，且 g 称为 f 的 **Dirichlet 逆元**（Dirichlet inverse），可以记作 f^{-1}. 而且，逆元 g 满足递推公式

$$g(n) = \frac{\varepsilon(n) - \sum_{k\ell=n,\ k\neq 1} f(k)g(\ell)}{f(1)}.$$

证明

为验证交换律，直接计算可知

$$(f * g)(n) = \sum_{k\ell=n} f(k)g(\ell) = (g * f)(n).$$

为验证结合律，直接计算可知

$$((f * g) * h)(n) = \sum_{k\ell m=n} f(k)g(\ell)h(m) = (f * (g * h))(n).$$

为验证分配律，直接计算可知

$$\begin{aligned}((f + g) * h)(n) &= \sum_{k\ell=n} (f(k) + g(k))h(\ell) \\ &= \sum_{k\ell=n} f(k)h(\ell) + \sum_{k\ell=n} g(k)h(\ell) = (f * h + g * h)(n).\end{aligned}$$

为验证 $\varepsilon(n)$ 是单位元，直接计算可知

$$(f * \varepsilon)(n) = \sum_{k\ell=n} f(k)\varepsilon(\ell) = f(n).$$

第二个等号是因为 $\varepsilon(\ell)$ 仅在 $\ell = 1$ 即 $k = n$ 时取得非零值.

最后，需要证明 f^{-1} 存在，当且仅当 $f(1) \neq 0$. 对于任意 f，假设存在 g 使得 $f * g = \varepsilon$. 这意味着

$$(f * g)(n) = \sum_{k\ell=n} f(k)g(\ell) = \varepsilon(n).$$

这实际上给出了一系列关于 $g(n)$ 取值的方程组，从中可以直接求出 $g(n)$. 特别地，当 $n = 1$ 时，等式变为 $f(1)g(1) = 1$，所以 g 存在，至少要求 $f(1) \neq 0$. 而只要 $f(1) \neq 0$，可以直接解出

$$g(n) = \frac{\varepsilon(n) - \sum_{k\ell=n,\ k\neq 1} f(k)g(\ell)}{f(1)}.$$

它可以用于递归计算 $g(n)$ 的取值. 因此，逆元 g 存在，当且仅当 $f(1) \neq 0$.

用抽象代数的语言说，这些代数性质说明，全体数论函数在（逐点）加法运算和 Dirichlet 卷积运算下构成 交换环→9.16.1@p1805，且它的全体可逆元就是那些在 $n=1$ 处取非零值的函数．这个环称为 **Dirichlet 环**（Dirichlet ring）．

积性函数是一类特殊的数论函数．它对于 Dirichlet 卷积和 Dirichlet 逆都是封闭的．

定理

设 f,g 是积性函数，那么，$f*g$ 也是积性函数．而且，逆元 f^{-1} 一定存在，它也是积性函数．

证明

对于第一点，设 $h=f*g$，直接验证可知，对于 $n_1\perp n_2$，都有

$$\begin{aligned}h(n_1)h(n_2)&=\left(\sum_{k_1\ell_1=n_1}f(k_1)g(\ell_1)\right)\left(\sum_{k_2\ell_2=n_2}f(k_2)g(\ell_2)\right)\\&=\sum_{k_1\ell_1=n_1,\ k_2\ell_2=n_2}f(k_1)f(k_2)g(\ell_1)g(\ell_2)\\&=\sum_{k\ell=n_1n_2}f(k)g(\ell)\\&=h(n_1n_2).\end{aligned}$$

其中，第三个等号利用了 f,g 的积性，其改变求和顺序的逻辑是：当 k 遍历 n_1n_2 的因数时，k 的素因子可以根据它是 n_1 还是 n_2 的素因子分为两类，将两类中的素因子（计重复）分别乘起来得到 k_1 和 k_2，它们将分别遍历 n_1 和 n_2 的因数；反过来，根据 n_1 和 n_2 的因数 k_1 和 k_2，总是可以得到 n_1n_2 的因数 $k=k_1k_2$．

对于第二点，设 $g=f^{-1}$，考虑应用数学归纳法．首先，$g(1)=1/f(1)=1$．此时，逆元的递推公式可以写作

$$g(n)=\varepsilon(n)-\sum_{k\ell=n,\ k\neq1}f(k)g(\ell).$$

所以，对于 $n_1\perp n_2$ 且 $n_1n_2>1$，有

$$\begin{aligned}g(n_1n_2)&=-\sum_{k\ell=n_1n_2,\ k\neq1}f(k)g(\ell)\\&=-\sum_{k_1\ell_1=n_1,\ k_2\ell_2=n_2,\ k_1k_2\neq1}f(k_1)f(k_2)g(\ell_1)g(\ell_2)\\&=f(1)f(1)g(n_1)g(n_2)-\sum_{k_1\ell_1=n_1,\ k_2\ell_2=n_2}f(k_1)f(k_2)g(\ell_1)g(\ell_2)\\&=g(n_1)g(n_2)-\left(\sum_{k_1\ell_1=n_1}f(k_1)g(\ell_1)\right)\left(\sum_{k_2\ell_2=n_2}f(k_2)g(\ell_2)\right)\\&=g(n_1)g(n_2)-\varepsilon(n_1)\varepsilon(n_2)\\&=g(n_1)g(n_2).\end{aligned}$$

其中，第二个等号用到了归纳假设，即对于 $\ell_1\ell_2<n_1n_2$ 且 $\ell_1\perp\ell_2$，条件 $g(\ell_1\ell_2)=g(\ell_1)g(\ell_2)$ 成立．

用抽象代数的语言说，全体积性函数在 Dirichlet 卷积运算下构成 Dirichlet 环乘法群的 子群→9.16.2@p1812．

更为特殊的是完全积性函数．

定理

设 α 是完全积性函数，f, g 是数论函数．那么，有：

1. 分配律：$(\alpha f) * (\alpha g) = \alpha \cdot (f * g)$.
2. 逆元：$(\alpha f)^{-1} = \alpha f^{-1}$，只要 f^{-1} 存在.
3. 积性函数 f 是完全积性函数，当且仅当 $f^{-1} = \mu f$，其中，μ 是 莫比乌斯函数→9.11.23@p1331.

证明

对于第一条，直接验证可知

$$\begin{aligned}((\alpha f) * (\alpha g))(n) &= \sum_{k\ell=n} (\alpha f)(k)(\alpha g)(\ell) \\ &= \sum_{k\ell=n} \alpha(k) f(k) \alpha(\ell) g(\ell) \\ &= \sum_{k\ell=n} \alpha(n) f(k) g(\ell) \\ &= \alpha(n)(f * g)(n).\end{aligned}$$

其中，第三个等号用到了完全积性函数的性质：$\alpha(n) = \alpha(k)\alpha(\ell)$ 对所有 $n = k\ell$ 都成立.

对于第二条，利用第一条就有

$$(\alpha f) * (\alpha f^{-1}) = \alpha(f * f^{-1}) = \alpha\varepsilon = \varepsilon.$$

其中，最后一个等号只利用了 $\alpha(1) = 1$．由逆元定义，$(\alpha f)^{-1} = \alpha f^{-1}$.

对于第三条，利用第二条和 $1^{-1} = \mu$ 可知，如果 f 是完全积性函数，那么

$$f^{-1} = (1f)^{-1} = 1^{-1} \cdot f = \mu f.$$

其中，1 是常数函数．反过来，如果 f 是积性函数且 $f^{-1} = \mu f$，那么只需要证明对于所有素数 p 和 $e \in \mathbf{N}_+$，都有 $f(p^e) = f(p)^e$ 成立，就能证明 f 是完全积性函数．为此，对 $e \in \mathbf{N}_+$ 应用数学归纳法．归纳起点 $e = 1$ 处命题显然成立．对于任意 $e > 1$，应用逆元递推公式，都有

$$\begin{aligned}f^{-1}(p^e) &= -\sum_{i=1}^{e} f(p^i) f^{-1}(p^{e-i}) \\ &= -\sum_{i=1}^{e} f(p^i) \mu(p^{e-i}) f(p^{e-i}) \\ &= -f(p^e) f(1) \mu(1) - f(p^{e-1}) \mu(p) f(p) \\ &= -f(p^e) + f(p)^e.\end{aligned}$$

其中，最后一个等号用到了归纳假设 $f(p^{e-1}) = f(p)^{e-1}$．应用 $f^{-1} = \mu f$，就得到

$$f^{-1}(p^e) = \mu(p^e) f(p^e) = 0.$$

代入前式，就得到

$$f(p^e) = f(p)^e.$$

所以，归纳步骤成立．原命题得证.

用抽象代数的语言说，如果 α 是完全积性函数，映射 $f \mapsto \alpha f$ 是 Dirichlet 环的 自同态→9.16.3@p1830.

Dirichlet 生成函数

与 Dirichlet 卷积紧密相关的是 Dirichlet 生成函数.

数论函数 $f(n)$——也就是数列 $\{f(n)\}$——对应的 **Dirichlet 生成函数**（Dirichlet series generating function，DGF）定义为形式 Dirichlet 级数（formal Dirichlet series）：

$$F(s)=\sum_{n=1}^{\infty}\frac{f(n)}{n^s}.$$

级数中的 s 是形式变元．常见的 Dirichlet 生成函数中，s 往往可以看作是复变量，进而讨论 Dirichlet 级数的解析性质，但这超出了算法竞赛的范围．

Dirichlet 生成函数的乘积对应着相应的数论函数的 Dirichlet 卷积：

定理

对于数论函数 f,g 及其 Dirichlet 生成函数 F,G，它们的 Dirichlet 卷积 $f*g$ 的生成函数等于 $F\cdot G$.

证明

直接验证：

$$\begin{aligned}F(s)G(s)&=\left(\sum_{k=1}^{\infty}\frac{f(k)}{k^s}\right)\left(\sum_{\ell=1}^{\infty}\frac{g(\ell)}{\ell^s}\right)=\sum_{k=1}^{\infty}\sum_{\ell=1}^{\infty}\frac{f(k)g(\ell)}{(k\ell)^s}\\&=\sum_{n=1}^{\infty}\frac{\sum_{k\ell=n}f(k)g(\ell)}{n^s}=\sum_{n=1}^{\infty}\frac{(f*g)(n)}{n^s}.\end{aligned}$$

利用 Dirichlet 卷积和 Dirichlet 生成函数乘积之间的对应关系，可以从 Dirichlet 生成函数的角度理解 Dirichlet 卷积的性质．由于形式 Dirichlet 级数的乘法运算满足交换律、结合律、对加法的分配律，数论函数的 Dirichlet 卷积运算满足同样的代数性质．

Euler 乘积

积性函数的特殊性同样反映在 Dirichlet 生成函数上．由于整数有 唯一分解定理→9.11.1@p1160，积性函数 $f(n)$ 的生成函数 $F(s)$ 可写成如下形式：

$$\begin{aligned}F(s)&=\sum_{n=1}^{\infty}\frac{f(n)}{n^s}=\sum_{n=1}^{\infty}\prod_{p\in\mathbf{P}}\frac{f(p^{\nu_p(n)})}{p^{\nu_p(n)s}}=\prod_{p\in\mathbf{P}}\sum_{e=0}^{\infty}\frac{f(p^e)}{p^{es}}\\&=\prod_{p\in\mathbf{P}}\left(1+\frac{f(p)}{p^s}+\frac{f(p^2)}{p^{2s}}+\frac{f(p^3)}{p^{3s}}+\cdots\right).\end{aligned}$$

其中，$\nu_p(n)$ 表示 n 的素因数分解中 p 的幂次．这意味着，$F(s)$ 可以分解为若干 $F_p(s)=\sum_{e=0}^{\infty}f(p^e)p^{-es}$ 的乘积，且每个 $F_p(s)$ 对应的数论函数都只在 p 的幂次处可能取非零值．这一无穷乘积也称为 **Euler 乘积**（Euler product）．如果 $F(s)$ 和 $G(s)$ 都能分解成类似的形式，那么它们的乘积同样如此；将这一观察对应到数论函数上，就是积性函数的 Dirichlet 卷积仍然是积性函数．

进一步地，如果 $f(n)$ 还是完全积性函数，那么 $f(p^e)=f(p)^e$，上式可以继续简化：

$$F(s)=\prod_{p\in\mathbf{P}}\sum_{e=0}^{\infty}\frac{f(p)^e}{p^{es}}=\prod_{p\in\mathbf{P}}\left(1-\frac{f(p)}{p^s}\right)^{-1}.$$

与积性函数不同，完全积性函数的 Dirichlet 生成函数的形式在乘法运算下并不具有封闭性，因此，完全积性函数的 Dirichlet 卷积和 Dirichlet 逆都未必是完全积性函数，但一定是积性函数．

例子

1. 单位函数 $\varepsilon(n)$ 是完全积性函数．它的 Dirichlet 生成函数是关于不定元 s 的常值函数
   $$E(s)=\sum_{n=1}^{\infty}\frac{\varepsilon(n)}{n^s}=1.$$
2. 常数函数 $1(n)$ 是完全积性函数．它的 Dirichlet 生成函数是 Riemann ζ 函数

$$I(s)=\sum_{n=1}^{\infty}\frac{1}{n^s}=\prod_{p\in\mathbf{P}}\frac{1}{1-p^{-s}}=\zeta(s).$$

3. 莫比乌斯函数 $\mu(n)$ 是常数函数的 Dirichlet 逆．它的 Dirichlet 生成函数是 $\zeta(s)$ 的倒数：

$$M(s)=\sum_{n=1}^{\infty}\frac{\mu(n)}{n^s}=\prod_{p\in\mathbf{P}}(1-p^{-s})=\frac{1}{\zeta(s)}.$$

4. 幂函数 $\mathrm{id}_k(n)=n^k$ 是完全积性函数．特别地，当 $k=0$ 时，它就是常数函数；当 $k=1$ 时，它就是恒等函数．它的 Dirichlet 生成函数是

$$I_k(s)=\sum_{n=1}^{\infty}\frac{n^k}{n^s}=\prod_{p\in\mathbf{P}}\frac{1}{1-p^{k-s}}=\zeta(s-k).$$

5. 欧拉函数 $\varphi(n)$ 是积性函数．它的 Dirichlet 生成函数是

$$\begin{aligned}\Phi(s)&=\prod_{p\in\mathbf{P}}\left(1+\frac{p-1}{p^s}+\frac{p(p-1)}{p^{2s}}+\frac{p^2(p-1)}{p^{3s}}+\cdots\right)\\&=\prod_{p\in\mathbf{P}}\left(\frac{1}{1-p^{1-s}}-\frac{1}{p^s}\frac{1}{1-p^{1-s}}\right)=\prod_{p\in\mathbf{P}}\frac{1-p^{-s}}{1-p^{1-s}}=\frac{\zeta(s-1)}{\zeta(s)}.\end{aligned}$$

结合幂函数的 Dirichlet 生成函数表达式，就得到 $\mathrm{id}=\varphi*1$.

6. 约数函数 $\sigma_k(n)=\sum_{d|n}d^k$ 是积性函数．它的 Dirichlet 生成函数是

$$\begin{aligned}\Sigma_k(s)&=\prod_{p\in\mathbf{P}}\left(1+\frac{1+p^k}{p^s}+\frac{1+p^k+p^{2k}}{p^{2s}}+\frac{1+p^k+p^{2k}+p^{3k}}{p^{3s}}+\cdots\right)\\&=\prod_{p\in\mathbf{P}}\frac{1}{1-p^k}\left((1-p^k)+\frac{1-p^{2k}}{p^s}+\frac{1-p^{3k}}{p^{2s}}+\frac{1-p^{4k}}{p^{3s}}+\cdots\right)\\&=\prod_{p\in\mathbf{P}}\frac{1}{1-p^k}\left(\frac{1}{1-p^{-s}}-\frac{p^k}{1-p^{k-s}}\right)\\&=\prod_{p\in\mathbf{P}}\frac{1}{(1-p^{-s})(1-p^{k-s})}=\zeta(s-k)\zeta(s).\end{aligned}$$

结合幂函数的 Dirichlet 生成函数表达式，就得到 $\sigma_k=\mathrm{id}_k*1$．这正是 σ_k 的定义式.

7. 无平方因子数的指示函数 $u(n)=|\mu(n)|=\mu^2(n)$ 是积性函数．它的 Dirichlet 生成函数是

$$U(s)=\prod_{p\in\mathbf{P}}(1+p^{-s})=\prod_{p\in\mathbf{P}}\frac{1-p^{-2s}}{1-p^{-s}}=\frac{\zeta(s)}{\zeta(2s)}.$$

Bell 级数

Euler 乘积说明，积性函数的 Dirichlet 生成函数可以分解为局部因子 $F_p(s)$ 的乘积．如果只关心单个素数处的信息，可以将 p^{-s} 整体看作一个形式变元，这就得到了 Bell 级数.

对于积性函数 f 和素数 p，定义 f 关于 p 的 **Bell 级数**（Bell series）为形式幂级数

$$f_p(x)=\sum_{e=0}^{\infty}f(p^e)x^e=1+f(p)x+f(p^2)x^2+\cdots.$$

对比 Euler 乘积的表达式可知，$f_p(x)$ 就是将 $F_p(s)$ 中的 p^{-s} 替换为 x 的结果，即 $F_p(s)=f_p(p^{-s})$，从而

$$F(s)=\prod_{p\in\mathbf{P}}f_p(p^{-s}).$$

引入这一记号后可以看出，Dirichlet 卷积在局部化之后就是普通的幂级数乘法.

将 Dirichlet 生成函数的结果转译到 Bell 级数上，就得到如下性质：

1. 对于积性函数 f,g 及任意素数 p，有 $(f*g)_p(x)=f_p(x)g_p(x)$.

2. 积性函数 f 是完全积性函数，当且仅当对每个素数 p 都有 $f_p(x)=\dfrac{1}{1-f(p)x}$.
3. 积性函数的 Dirichlet 逆总是存在，且 $(f^{-1})_p(x)=\dfrac{1}{f_p(x)}$.

因为积性函数由它在素数幂处的取值唯一确定，所以积性函数与它的全体 Bell 级数 $\{f_p\}_{p\in\mathbf{P}}$ 是一一对应的. 于是，关于积性函数的 Dirichlet 卷积的许多问题，都可以逐个素数地转化为形式幂级数的问题来讨论. 相比直接使用 Dirichlet 生成函数，其形式上要简洁一些. 例如，第三条性质说明，求 Dirichlet 逆只需逐个素数地求形式幂级数的逆；在实际计算中，因为只需要 $p^e\le n$ 的部分，每个素数至多截断到 $x^{\lfloor\log_p n\rfloor}$ 项.

例子

前文例子中的积性函数，对应的 Bell 级数如下：

f	ε	1	μ	id_k	φ	σ_k	$\lvert\mu\rvert=\mu^2$
$f_p(x)$	1	$\dfrac{1}{1-x}$	$1-x$	$\dfrac{1}{1-p^kx}$	$\dfrac{1-x}{1-px}$	$\dfrac{1}{(1-x)(1-p^kx)}$	$1+x$

利用上表，常见的 Dirichlet 卷积关系都可以直接验证. 例如 $\mu*1=\varepsilon$ 对应

$$(1-x)\cdot\frac{1}{1-x}=1,$$

而 $\varphi*1=\mathrm{id}$ 对应

$$\frac{1-x}{1-px}\cdot\frac{1}{1-x}=\frac{1}{1-px}.$$

应用

Dirichlet 生成函数可以用于将数论函数表示为 Dirichlet 卷积.

例如在杜教筛的过程中，要计算数论函数 f 的前缀和，需要找到另一个数论函数 g 使得 $f*g$ 和 g 都可以快速求前缀和. 可以利用 Dirichlet 生成函数推导这一过程.

以杜教筛一节的例题 Luogu P3768 简单的数学题→9.11.24@p1353 为例，需要对 $f(n)=n^2\varphi(n)$ 构造满足上述条件的数论函数 $g(n)$. 由于 f 是积性函数，它的 Dirichlet 生成函数为

$$F(s)=\prod_{p\in\mathbf{P}}\left(1+\sum_{k=1}^{\infty}\frac{p^{3k-1}(p-1)}{p^{ks}}\right)=\prod_{p\in\mathbf{P}}\frac{1-p^{2-s}}{1-p^{3-s}}=\frac{\zeta(s-3)}{\zeta(s-2)}.$$

对比幂函数的 Dirichlet 生成函数可知，只要取 $g=\mathrm{id}_2$，就有 $f*g=\mathrm{id}_3$. 两者都是可以快速计算前缀和的.

由于常见问题中的数论函数都是积性函数，实践中利用 Bell 级数往往更加方便. 例如，刚刚的例子中 $f(n)$ 的 Bell 级数可以直接由 $f(p^e)=p^{3e-1}(p-1)$ 推得为

$$f_p(x)=1+\frac{p-1}{p}\sum_{e=1}^{\infty}(p^3x)^e=1+\frac{(p-1)p^2x}{1-p^3x}=\frac{1-p^2x}{1-p^3x}.$$

对比幂函数的 Bell 级数立即得到 $f*\mathrm{id}_2=\mathrm{id}_3$.

Dirichlet 卷积的计算

本节讨论 Dirichlet 卷积的计算问题，即给定序列 $\{f(k)\}_{k=1}^n$ 和 $\{g(k)\}_{k=1}^n$，求解 Dirichlet 卷积 $h=f*g$ 的前若干项 $\{h(k)\}_{k=1}^n$ 的问题. 根据涉及到的函数性质，算法的复杂度也略有不同.

一般情形

如果 f,g,h 都没有特殊性质，那么 Dirichlet 卷积的计算只能利用其定义：

$$h(n)=\sum_{k\ell=n}f(k)g(\ell).$$

枚举 k 和 ℓ，将贡献 $f(k)g(\ell)$ 累加到 $h(k\ell)$ 上即可. 枚举复杂度为

$$O\left(\sum_{k=1}^{n}\frac{n}{k}\right)=O(n\log n).$$

参考实现如下：

参考实现

```
// Compute the Dirichlet convolution h = f * g.
auto dirichlet_convolute(const std::vector<int>& f, const std::vector<int>&
g) {
  int n = f.size() - 1;
  std::vector<int> h(n + 1);
  for (int k = 1; k <= n; ++k) {
    for (int d = 1; k * d <= n; ++d) {
      h[k * d] += f[k] * g[d];
    }
  }
  return h;
}

```

与积性函数卷积的情形

如果 g 是积性函数，那么可以利用 Euler 乘积加速 Dirichlet 卷积的计算．计算 h 相当于计算它的 Dirichlet 生成函数 H 中各项的系数．由于

$$H(s)=F(s)G(s)=F(s)\prod_{p\in\mathbf{P}}G_p(s).$$

其中，$G_p(s)$ 是 $G(s)$ 的 Euler 乘积分解中的因式，它只包含 p 的幂次处的系数：

$$G_p(s)=\sum_{p^k\le n}\frac{g(p^k)}{p^{ks}}=1+\frac{g(p)}{p^s}+\frac{g(p^2)}{p^{2s}}+\cdots.$$

那么，从 $F(s)$ 开始，遍历所有不超过 n 的素数 p，将 $G_p(s)$ 逐一乘上去，同样可以得到最终结果 $H(s)$．将 $G_p(s)$ 乘上去时，直接应用一般情形中的暴力枚举算法即可．总枚举次数

$$\sum_{p\in\mathbf{P},\,p\le n}\sum_{k=1}^{\infty}\left\lfloor\frac{n}{p^k}\right\rfloor\le\sum_{p\in\mathbf{P},\,p\le n}\frac{n}{p-1}\le\sum_{p\in\mathbf{P},\,p\le n}\frac{2n}{p}\in O(n\log\log n).$$

最后一步复杂度的估计与 Eratosthenes 筛法→9.11.6@p1199 复杂度的证明一致．所以，本算法的时间复杂度为 $O(n\log\log n)$．

参考实现如下：

参考实现

```
// Compute the Dirichlet convolution h = f * g.
// Assume that g is multiplicative.
auto dirichlet_convolute(const std::vector<int>& f, const std::vector<int>&
g) {
  int n = f.size() - 1;
  std::vector<int> h(f);
  std::vector<bool> vis(n + 1);
  for (int i = 2; i <= n; ++i) {
    if (vis[i]) continue;

```

```
    // Reverse the order for in-place computation.
    for (int k = n / i * i; k; k -= i) {
      vis[k] = true;
      int d = k;
      while (true) {
        d /= i;
        h[k] += h[d] * g[k / d];
        if (d % i) break;
      }
    }
  }
  return h;
}

```

特别地，当积性函数 g 是完全积性函数或其 Dirichlet 逆时，例如当 $g = 1$ 或 $g = \mu$ 时，算法可以进一步简化．此时，Dirichlet 卷积 $h = f * g$ 的计算可以采用常数更小的 Dirichlet 前缀和/差分→9.11.23@p1331 算法，但是算法时间复杂度仍为 $O(n \log \log n)$.

结果为积性函数的情形

最后，考虑 h 是积性函数的情形．特别地，当 f, g 都是积性函数时，$h = f * g$ 就是积性函数．要计算 h，只需要确定它在素数幂处的取值，就可以通过 线性筛→9.11.6@p1199 在 $O(n)$ 时间内计算．而对于素数幂 p^e 处的取值 $h(p^e)$ 直接暴力计算即可：

$$h(p^e) = \sum_{i=0}^{e} f(p^i) g(p^{e-i}).$$

这些暴力计算需要的枚举次数为

$$\begin{aligned}\sum_{p \in \mathbf{P},\, p \le n} \sum_{e=1}^{\lfloor \log_p n \rfloor} (e+1) &\le \sum_{p \in \mathbf{P},\, p \le \sqrt{n}} 2 \left\lfloor \log_p n \right\rfloor^2 + \sum_{p \in \mathbf{P},\, \sqrt{n} < p \le n} 2 \\ &\le 2\sqrt{n} (\log_2 n)^2 + 2n \in O(n).\end{aligned}$$

因此，这一算法的总时间复杂度为 $O(n)$.

参考实现如下：

参考实现

```
// Compute the Dirichlet convolution h = f * g.
// Assume that h is multiplicative.
auto dirichlet_convolute(const std::vector<int>& f, const std::vector<int>&
g) {
  int n = f.size() - 1;
  std::vector<int> h(n + 1), primes, rem(n + 1), lpf(n + 1);
  std::vector<bool> vis(n + 1);
  h[1] = 1;
  for (int x = 2; x <= n; ++x) {
    if (!vis[x]) {
      primes.push_back(x);
      rem[x] = 1;
      lpf[x] = x;
    }
```

```
    for (int p : primes) {
      if (x * p > n) break;
      vis[x * p] = true;
      rem[x * p] = x % p ? x : rem[x];
      lpf[x * p] = p;
      if (x % p == 0) break;
    }
    if (rem[x] == 1) {  // prime powers.
      for (int k = x; k; k /= lpf[x]) {
        h[x] += f[k] * g[x / k];
      }
    } else {  // other cases.
      h[x] = h[rem[x]] * h[x / rem[x]];
    }
  }
  return h;
}

```

参考资料与注释

- Dirichlet convolution - Wikipedia[1]
- Dirichlet series - Wikipedia[2]
- Euler product - Wikipedia[3]
- Bell series - Wikipedia[4]
- Dirichlet 積と、数論関数の累積和 by maspy[5]
- 杜教筛（+ 贝尔级数 + powerful number）by command_block - 洛谷[6]

外部参考资料

[1] Dirichlet convolution - Wikipedia

[2] Dirichlet series - Wikipedia

[3] Euler product - Wikipedia

[4] Bell series - Wikipedia

[5] Dirichlet 積と、数論関数の累積和 by maspy

[6] 杜教筛（+ 贝尔级数 + powerful number）by command_block - 洛谷

9.11.23 莫比乌斯反演

Authors: hydingsy, hyp1231, ranwen, 383494

前置知识：数论分块→9.11.21@p1314、Dirichlet 卷积→9.11.22@p1322

莫比乌斯反演是数论中的重要内容. 对于一些函数 $f(n)$, 如果很难直接求出它的值, 而容易求出其倍数和或约数和 $g(n)$, 那么可以通过莫比乌斯反演简化运算, 求得 $f(n)$ 的值.

莫比乌斯函数

莫比乌斯函数（Möbius 函数）定义为

$$\mu(n) = \begin{cases} 1, & n = 1, \\ 0, & n \text{ is divisible by a square} > 1, \\ (-1)^k, & n \text{ is the product of } k \text{ distinct primes.} \end{cases}$$

具体地, 假设正整数 n 有素因数分解 $n = \prod_{i=1}^{k} p_i^{e_i}$, 其中, p_i 是素数, e_i 是正整数. 那么, 三种情形分别对应:

1. $\mu(1)=1$;
2. 当存在 i 使得 $e_i>1$，即存在任何素因数出现超过一次时，$\mu(n)=0$;
3. 否则，对于所有 i 都有 $e_i=1$，即任何素因数都只出现一次时，$\mu(n)=(-1)^k$，其中，k 就是互异素因子的个数.

性质

根据定义容易验证，莫比乌斯函数 $\mu(n)$ 是积性函数，但不是完全积性函数. 除此之外，最为重要的性质是下述恒等式:

性质

对于正整数 n，有

$$\sum_{d|n}\mu(d)=[n=1]=\begin{cases}1,n=1,\\0,n\neq 1.\end{cases}$$

其中 $[\cdot]$ 是 Iverson 括号.

证明

令 $n=\prod_{i=1}^{k}p_i^{e_i}$，设 $n'=\prod_{i=1}^{k}p_i$. 根据 二项式定理→9.13.1@p1612，有

$$\sum_{d|n}\mu(d)=\sum_{d|n'}\mu(d)=\sum_{i=0}^{k}\binom{k}{i}(-1)^i=(1+(-1))^k=[k=0]=[n=1].$$

利用 Dirichlet 卷积, 该表达式可以写作 $\varepsilon=1*\mu$. 也就是说, 莫比乌斯函数是常值函数 1 的 Dirichlet 逆.

这一性质有一个很常见的应用:

$$[i\perp j]=[\gcd(i,j)=1]=\sum_{d|\gcd(i,j)}\mu(d)=\sum_{d}[d\mid i][d\mid j]\mu(d).$$

它将互素的条件转化为关于莫比乌斯函数的求和式，方便进一步推导.

求法

如果需要对单个 n 计算莫比乌斯函数 $\mu(n)$ 的值，可以利用它的 素因数分解→9.11.7@p1209. 例如，在 n 不太大时，可以在 $O(\sqrt{n})$ 时间内求出 $\mu(n)$ 的值.

参考实现

- C++

```cpp
int mu(int n) {
  int res = 1;
  for (int i = 2; i * i <= n; ++i) {
    if (n % i == 0) {
      n /= i;
      // Check if square-free.
      if (n % i == 0) return 0;
      res = -res;
    }
  }
  // The remaining factor must be prime.
  if (n > 1) res = -res;
  return res;
}
```

```
15
```

- Python

```python
def mu(n):
    res = 1
    i = 2
    while i * i <= n:
        if n % i == 0:
            n //= i
            # Check if square-free
            if n % i == 0:
                return 0
            res = -res
        i += 1
    # The remaining factor must be prime
    if n > 1:
        res = -res
    return res


```

如果需要对前 n 个正整数预处理出 $\mu(n)$ 的值，可以利用它是积性函数，通过 线性筛→9.11.6@p1199 在 $O(n)$ 时间内计算.

参考实现

- C++

```cpp
std::vector<int> get_mu(int n) {
  std::vector<int> mu(n + 1), primes;
  std::vector<bool> not_prime(n + 1);
  primes.reserve(n);
  mu[1] = 1;
  for (int x = 2; x <= n; ++x) {
    if (!not_prime[x]) {
      primes.push_back(x);
      mu[x] = -1;
    }
    for (int p : primes) {
      if (x * p > n) break;
      not_prime[x * p] = true;
      if (x % p == 0) {
        mu[x * p] = 0;
        break;
      } else {
        mu[x * p] = -mu[x];
      }
    }
  }
  return mu;
}

```

- Python

```python
def get_mu(n):
    mu = [0] * (n + 1)
    primes = []
    not_prime = [False] * (n + 1)

    mu[1] = 1
    for x in range(2, n + 1):
        if not not_prime[x]:
            primes.append(x)
            mu[x] = -1
        for p in primes:
            if x * p > n:
                break
            not_prime[x * p] = True
            if x % p == 0:
                mu[x * p] = 0
                break
            else:
                mu[x * p] = -mu[x]
    return mu


```

莫比乌斯反演

莫比乌斯函数最重要的应用就是莫比乌斯反演.

莫比乌斯反演

设 $f(n), g(n)$ 是两个数论函数. 那么，有

$$f(n) = \sum_{d|n} g(d) \iff g(n) = \sum_{d|n} \mu\left(\frac{n}{d}\right) f(d).$$

证明一

直接验证，有:

$$\begin{aligned}
\sum_{d|n} \mu\left(\frac{n}{d}\right) f(d) &= \sum_{d|n} \mu\left(\frac{n}{d}\right) \sum_{k|d} g(k) \\
&= \sum_{k|n} g(k) \sum_{d} [k \mid d \mid n] \mu\left(\frac{n}{d}\right) \\
&= \sum_{k|n} g(k) \sum_{d|n} \left[\frac{n}{d} \mid \frac{n}{k}\right] \mu\left(\frac{n}{d}\right) \\
&= \sum_{k|n} g(k) \left[\frac{n}{k} = 1\right] \\
&= g(n).
\end{aligned}$$

式子变形的关键在于交换求和次序，并注意到 $k \mid d \mid n$ 就等价于 $\frac{n}{d} \mid \frac{n}{k}$．倒数第二个等号相当于对 $\frac{n}{k}$ 的因子 $\frac{n}{d}$ 处的莫比乌斯函数求和，所以就等于 $\left[\frac{n}{k}=1\right]$．这一表达式仅在 $n=k$ 处不是 0，最后就会得到 $g(n)$.

证明二

利用 Dirichlet 卷积，命题等价于

$$f=1*g \iff g=\mu*f.$$

利用 $1*\mu=\varepsilon$，在左式的等号两侧同时对 μ 做卷积，就得到

$$f*\mu=(1*g)*\mu=(1*\mu)*g=\varepsilon*g=g.$$

在涉及各种整除关系的数论函数求和中，莫比乌斯反演是有力的变形工具.

例子

1. 欧拉函数→9.11.5@p1196 $\varphi(n)$ 满足关系式 $n=\sum_{d\mid n}\varphi(d)$，亦即 $\mathrm{id}=1*\varphi$．对它进行反演，就得到 $\varphi=\mu*\mathrm{id}$，亦即
$$\varphi(n)=\sum_{d\mid n} d\mu\left(\frac{n}{d}\right).$$
2. 除数函数 $\sigma_k(n)=\sum_{d\mid n} d^k$，亦即 $\sigma_k=1*\mathrm{id}_k$．对它进行反演，就得到 $\mathrm{id}_k=\mu*\sigma_k$，亦即
$$n^k=\sum_{d\mid n}\mu\left(\frac{n}{d}\right)\sigma_k(d).$$
3. 互异素因子数目函数 $\omega(n)=\sum_{d\mid n}[d\in\mathbf{P}]$，亦即 $\omega=1*\mathbf{1}_{\mathbf{P}}$，其中 $\mathbf{1}_{\mathbf{P}}$ 是素数集 $\mathbf{P}$ 的指示函数．对它进行反演，就得到 $\mathbf{1}_{\mathbf{P}}=\mu*\omega$，亦即
$$[n\in\mathbf{P}]=\sum_{d\mid n}\mu\left(\frac{n}{d}\right)\omega(d).$$
4. 考察满足 $\log n=\sum_{d\mid n}\Lambda(d)$ 的数论函数 $\Lambda(n)$．它就是对数函数的莫比乌斯反演，也称为 von Mangoldt 函数：
$$\Lambda(n)=\sum_{d\mid n}\mu\left(\frac{n}{d}\right)\log d=\begin{cases}\log p, n=p^e,\ p\in\mathbf{P},\ e\in\mathbf{N}_+,\\ 0, \quad \text{otherwise.}\end{cases}$$

附：$\Lambda(n)$ 表达式的证明

对于素数幂 $n=p^e$ $(e\in\mathbf{N}_+)$，有

$$\Lambda(n)=\sum_{i=0}^{e}\mu(p^{e-i})\log p^i=\log p^e-\log p^{e-1}=\log p.$$

对于 $n=1$，显然有 $\Lambda(n)=\log 1=0$．对于其他合数 n，有

$$\Lambda(n)=\sum_{d\mid n}\mu(d)(\log n-\log d)=\left(\sum_{d\mid n}\mu(d)\right)\log n-\sum_{d\mid n}\mu(d)\log d.$$

根据莫比乌斯函数的性质，$\log n$ 一项的系数为 $[n=1]=0$．对于后面的一项，可以进一步将 d 分解为素因数之积．对于任何素数 $p\mid n$，设 $n=p^e m$ 且 $p\perp m$（由于 n 不是素数幂，$m>1$），考察 $\log p$ 的系数（只有无平方因子的 d 才有贡献），都有：

$$-\sum_{p\mid d\mid n}\mu(d)=-\sum_{d'\mid m}\mu(pd')=\sum_{d'\mid m}\mu(d')=[m=1]=0.$$

由此，对于不止一个素因子的合数 n，都有 $\Lambda(n)=0$.

拓展形式

除了上述基本形式外，莫比乌斯反演还有一些常见的拓展形式．首先，可以考虑它的倍数和形式．

拓展一

设 $f(n),g(n)$ 是两个数论函数．那么，有

$$f(n)=\sum_{n|d}g(d)\iff g(n)=\sum_{n|d}\mu\left(\frac{d}{n}\right)f(d).$$

证明

直接验证，有：

$$\begin{aligned}\sum_{n|d}\mu\left(\frac{d}{n}\right)f(d)&=\sum_{n|d}\mu\left(\frac{d}{n}\right)\sum_{d|k}g(k)\\&=\sum_{n|k}g(k)\sum_{d}[n\mid d\mid k]\mu\left(\frac{d}{n}\right)\\&=\sum_{n|k}g(k)\sum_{n|d}\left[\frac{d}{n}\mid\frac{k}{n}\right]\mu\left(\frac{d}{n}\right)\\&=\sum_{n|k}g(k)\left[\frac{k}{n}=1\right]\\&=g(n).\end{aligned}$$

这和基本形式的推导完全对偶．

其次，莫比乌斯反演并不仅限于加法，它实际上对于任何 Abel 群→9.16.1@p1805 中的运算都成立．例如，它有如下的乘法形式：

拓展二

设 $f(n),g(n)$ 是两个数论函数．那么，有

$$f(n)=\prod_{d|n}g(d)\iff g(n)=\prod_{d|n}f(d)^{\mu(n/d)}.$$

证明

直接验证，有：

$$\begin{aligned}\prod_{d\mid n} f(d)^{\mu(n/d)} &= \prod_{d\mid n}\left(\prod_{k\mid d} g(k)\right)^{\mu(n/d)}\\ &= \prod_{k\mid n} g(k) \uparrow \left(\sum_{d}[k \mid d \mid n]\mu\left(\frac{n}{d}\right)\right)\\ &= \prod_{k\mid n} g(k) \uparrow \left(\sum_{d\mid n}\left[\frac{n}{d} \mid \frac{n}{k}\right]\mu\left(\frac{n}{d}\right)\right)\\ &= \prod_{k\mid n} g(k) \uparrow \left[\frac{n}{k}=1\right]\\ &= g(n).\end{aligned}$$

其中，$a\uparrow b = a^b$ 是 Knuth 箭头．对比基本形式的证明可以发现，唯一的区别就是加法换成了乘法，且乘法换成了取幂．

从 Dirichlet 卷积的角度看，莫比乌斯反演只是利用了「莫比乌斯函数是常值函数的 Dirichlet 逆」这一点．容易想象，类似莫比乌斯反演的关系对于一般的 Dirichlet 逆→9.11.22@p1322 同样成立．

拓展三

设 $f(n), g(n), \alpha(n)$ 都是数论函数，且 $\alpha^{-1}(n)$ 是 $\alpha(n)$ 的 Dirichlet 逆，即

$$[n=1] = \sum_{d\mid n}\alpha\left(\frac{n}{d}\right)\alpha^{-1}(d).$$

那么，有

$$f(n) = \sum_{d\mid n}\alpha\left(\frac{n}{d}\right)g(d) \iff g(n) = \sum_{d\mid n}\alpha^{-1}\left(\frac{n}{d}\right)f(d).$$

证明

直接验证，有：

$$\begin{aligned}\sum_{d\mid n}\alpha^{-1}\left(\frac{n}{d}\right)f(d) &= \sum_{d\mid n}\alpha^{-1}\left(\frac{n}{d}\right)\sum_{k\mid d}\alpha\left(\frac{d}{k}\right)g(k)\\ &= \sum_{k\mid n} g(k)\sum_{d}[k \mid d \mid n]\alpha\left(\frac{d}{k}\right)\alpha^{-1}\left(\frac{n}{d}\right)\\ &= \sum_{k\mid n} g(k)\sum_{d\mid n}\left[\frac{n}{d} \mid \frac{n}{k}\right]\alpha\left(\frac{d}{k}\right)\alpha^{-1}\left(\frac{n/k}{d/k}\right)\\ &= \sum_{k\mid n} g(k)\left[\frac{n}{k}=1\right]\\ &= g(n).\end{aligned}$$

和基本形式的证明相比较，只需要将倒数第二个等号替换成 Dirichlet 逆的定义式．

推论

设 $f(n), g(n)$ 是数论函数，且 $t(n)$ 是完全积性函数．那么，有

$$f(n) = \sum_{d\mid n} t\left(\frac{n}{d}\right)g(d) \iff g(n) = \sum_{d\mid n}\mu\left(\frac{n}{d}\right)t\left(\frac{n}{d}\right)f(d).$$

证明

由 Dirichlet 卷积的 性质→9.11.22@p1322 可知，对于完全积性函数 $t(n)$，它的 Dirichlet 逆就是 $\mu(n)t(n)$.

最后，莫比乌斯反演还可以推广到 $[1,+\infty)$ 上的复值函数，而不仅仅局限于数论函数．基本形式的莫比乌斯反演可以看作是复值函数在所有非整数点处均取零值的特殊情形.

拓展四

设 $F(x)$ 和 $G(x)$ 都是 $[1,+\infty)$ 上的复值函数．那么，有

$$F(x)=\sum_{n=1}^{\lfloor x\rfloor}G\left(\frac{x}{n}\right)\iff G(x)=\sum_{n=1}^{\lfloor x\rfloor}\mu(n)F\left(\frac{x}{n}\right).$$

证明

不妨对 F 和 G 补充定义，设当 $x<1$ 时，恒有 $F(x)=G(x)=0$．那么，命题就等价于：

$$F(x)=\sum_{n}G\left(\frac{x}{n}\right)\iff G(x)=\sum_{n}\mu(n)F\left(\frac{x}{n}\right).$$

这些求和式都是对 $n\in\mathbf{N}_+$ 求和.

直接验证，有：

$$\begin{aligned}\sum_{n}\mu(n)F\left(\frac{x}{n}\right)&=\sum_{n}\mu(n)\sum_{d}G\left(\frac{x/n}{d}\right)\\&=\sum_{k}G\left(\frac{x}{k}\right)\sum_{n\mid k}\mu(n)\\&=\sum_{k}G\left(\frac{x}{k}\right)[k=1]\\&=G(x).\end{aligned}$$

其中，为得到第二个等号，需要令 $k=nd$.

推论

设 $f(n),g(n)$ 是数论函数．那么，有

$$f(n)=\sum_{k=1}^{n}g\left(\left\lfloor\frac{n}{k}\right\rfloor\right)\iff g(n)=\sum_{k=1}^{n}\mu(k)f\left(\left\lfloor\frac{n}{k}\right\rfloor\right).$$

证明

只需要取 $F(x)=f(\lfloor x\rfloor)$ 和 $G(x)=g(\lfloor x\rfloor)$ 即可.

这些拓展形式之间可以互相组合，进而得到更为复杂的反演关系.

Dirichlet 前缀和

前置知识：前缀和与差分→5.8@p617

考虑基本形式的莫比乌斯反演关系：

$$f(n)=\sum_{d\mid n}g(d)\iff g(n)=\sum_{d\mid n}\mu\left(\frac{n}{d}\right)f(d).$$

左侧等式中，$f(n)$ 的值是 n 的所有因数处 g 的值之和．如果将 $a \mid b$ 理解为 a 排在 b 之前，那么 $f(n)$ 就可以理解为某种意义下 $g(n)$ 的前缀和．因此，在国内竞赛圈，由 $\{g(k)\}_{k=1}^{n}$ 求出 $\{f(k)\}_{k=1}^{n}$ 的过程也称为 **Dirichlet 前缀和**，相应的逆过程则称为 Dirichlet 差分．这些方法大多出现在需要预处理某个数论函数在前 N 个点处取值的情形．

接下来，讨论 Dirichlet 前缀和的计算．如果将每一个素数都看作一个维度，这就是一种高维前缀和．回忆高维前缀和的 逐维前缀和算法→5.8@p617：逐个遍历所有的维度，并将每个位置的值都累加到该位置在该维度上的后继位置．对于数论函数，这相当于说，从小到大遍历所有素数 p，并将 n 处的函数值累加到 np 处．这和 Eratosthenes 筛法→9.11.6@p1199 的遍历顺序是一致的．因此，这一算法可以在 $O(n \log \log n)$ 时间内计算出长度为 n 的数列的 Dirichlet 前缀和．类似地，利用逐维差分就可以在相同时间复杂度内求出数列的 Dirichlet 差分．

参考实现

- Dirichlet 前缀和

```cpp
std::vector<int> dirichlet_presum(const std::vector<int>& g) {
  int n = g.size() - 1;
  std::vector<int> f(g);
  std::vector<bool> vis(n + 1);
  for (int x = 2; x <= n; ++x) {
    if (vis[x]) continue;
    for (int y = 1, xy = x; xy <= n; ++y, xy += x) {
      vis[xy] = true;
      f[xy] += f[y];
    }
  }
  return f;
}

```

- Dirichlet 差分

```cpp
std::vector<int> dirichlet_diff(const std::vector<int>& f) {
  int n = f.size() - 1;
  std::vector<int> g(f);
  std::vector<int> vis(n + 1);
  for (int x = 2; x <= n; ++x) {
    if (vis[x]) continue;
    for (int y = n / x, xy = x * y; y; --y, xy -= x) {
      vis[xy] = true;
      g[xy] -= g[y];
    }
  }
  return g;
}

```

这一计算方法可以推广到倍数和（拓展一）、乘积形式（拓展二）、利用完全积性函数代替常值函数（拓展三的推论）等拓展形式中．

例题

本节通过例题展示莫比乌斯反演的应用方法以及一些常见变形技巧．首先，通过一道例题熟悉处理求和式中最大公因数条件的基本技巧．

Luogu P2522 [HAOI 2011] Problem b[1]

T 组数据．对每组数据，求值：

$$\sum_{i=x}^{n}\sum_{j=y}^{m}[\gcd(i,j)=k].$$

数据范围：$1\le T,x,y,n,m,k\le 5\times 10^4$.

解答

根据容斥原理，原式可以分成 4 块来处理，且每一块的式子都具有形式

$$f(n,m,k)=\sum_{i=1}^{n}\sum_{j=1}^{m}[\gcd(i,j)=k].$$

对于这类式子，接下来是一段标准的推导流程：提取公因数，应用莫比乌斯函数性质，交换求和次序．

首先，由于 i,j 都只能取 k 的倍数，可以先将这个因子提出来——这相当于代入 $i=ki'$ 和 $j=kj'$，就得到：

$$f(n,m,k)=\sum_{i=1}^{\lfloor n/k\rfloor}\sum_{j=1}^{\lfloor m/k\rfloor}[\gcd(i,j)=1].$$

再利用莫比乌斯函数的性质可知：

$$[\gcd(i,j)=1]=\sum_{d\mid\gcd(i,j)}\mu(d)=\sum_{d}[d\mid i][d\mid j]\mu(d).$$

将它代入表达式，并交换求和次序，就得到：

$$f(n,m,k)=\sum_{d}\mu(d)\left(\sum_{i=1}^{\lfloor n/k\rfloor}[d\mid i]\right)\left(\sum_{j=1}^{\lfloor m/k\rfloor}[d\mid j]\right).$$

这样一段操作的好处是，固定 d 时，求和式中关于 i 和 j 的项相互分离，可以分别求和．接下来，因为

$$\sum_{i=1}^{\lfloor n/k\rfloor}[d\mid i]=\left\lfloor\frac{\lfloor n/k\rfloor}{d}\right\rfloor,\quad\sum_{j=1}^{\lfloor m/k\rfloor}[d\mid j]=\left\lfloor\frac{\lfloor m/k\rfloor}{d}\right\rfloor,$$

所以，有

$$f(n,m,k)=\sum_{d}\mu(d)\left\lfloor\frac{\lfloor n/k\rfloor}{d}\right\rfloor\left\lfloor\frac{\lfloor m/k\rfloor}{d}\right\rfloor.$$

用线性筛预处理完 $\mu(d)$，并预处理其前缀和后，就可以通过数论分块求解．总的时间复杂度为 $O(N+T\sqrt{N})$，其中，N 为 n,m 的上界，T 为数据组数．

参考代码

```cpp
#include <algorithm>
#include <iostream>
using namespace std;
constexpr int N = 50000;
int mu[N + 5], p[N + 5];
bool flg[N + 5];

void init() {
```

```
  int tot = 0;
  mu[1] = 1;
  for (int i = 2; i <= N; ++i) {
    if (!flg[i]) {
      p[++tot] = i;
      mu[i] = -1;
    }
    for (int j = 1; j <= tot && i * p[j] <= N; ++j) {
      flg[i * p[j]] = true;
      if (i % p[j] == 0) {
        mu[i * p[j]] = 0;
        break;
      }
      mu[i * p[j]] = -mu[i];
    }
  }
  for (int i = 1; i <= N; ++i) mu[i] += mu[i - 1];
}

int solve(int n, int m) {
  int res = 0;
  for (int i = 1, j; i <= min(n, m); i = j + 1) {
    j = min(n / (n / i), m / (m / i));
    res += (mu[j] - mu[i - 1]) * (n / i) * (m / i);  // 代推出来的式子
  }
  return res;
}

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  int T, a, b, c, d, k;
  init();  // 预处理mu数组
  cin >> T;
  for (int i = 1; i <= T; i++) {
    cin >> a >> b >> c >> d >> k;
    // 根据容斥原理，1<=x<=b&&1<=y<=d范围中的答案数减去1<=x<=b&&1<=y<=c-1范围中的答案数和
    //   1<=x<=a-1&&1<=y<=d范围中的答案数再加上1<=x<=a-1&&1<=y<=c-1范围中的答案数
    //   即可得到a<=x<=b&&c<=y<=d范围中的答案数
    // 这一步如果不懂可以画坐标图进行理解
    cout << solve(b / k, d / k) - solve(b / k, (c - 1) / k) -
                solve((a - 1) / k, d / k) + solve((a - 1) / k, (c - 1) / k)
         << '\n';
  }
  return 0;
}

```

接下来的两道例题展示了枚举公因数的处理方法，并利用 筛法→9.11.6@p1199 计算一般积性函数的值.

SPOJ LCMSUM[2]

T 组数据. 对每组数据，求值：

$$\sum_{i=1}^{n}\operatorname{lcm}(i,n).$$

数据范围：$1\le T\le 3\times 10^5$，$1\le n\le 10^6$.

解答一

题目提供的是最小公倍数，但往往最大公因数更容易处理．所以，首先做变形：

$$f(n)=\sum_{i=1}^{n}\operatorname{lcm}(i,n)=\sum_{i=1}^{n}\frac{i\cdot n}{\gcd(i,n)}.$$

将 n 提出，并枚举最大公因数 k：

$$f(n)=n\sum_{k\mid n}\sum_{i=1}^{n}\frac{i}{k}[\gcd(i,n)=k].$$

对于内层的求和式，这是最常见的含有最大公因数的情形，重复标准的处理流程，就有：

$$\begin{aligned}f(n)&=n\sum_{k\mid n}\sum_{i=1}^{n/k}i\left[\gcd\left(i,\frac{n}{k}\right)=1\right]\\&=n\sum_{k\mid n}\sum_{i=1}^{n/k}i\sum_{d}\mu(d)[d\mid i]\left[d\mid\frac{n}{k}\right]\\&=n\sum_{k\mid n}\sum_{d}\mu(d)\left[d\mid\frac{n}{k}\right]\left(\sum_{i=1}^{n/k}i[d\mid i]\right).\end{aligned}$$

再次地，关于 i 的求和式与其他部分分离，可以单独处理．最后的求和式实际上是一个等差数列求和：（取 $i=di'$）

$$\sum_{i=1}^{n/k}i[d\mid i]=d\frac{1}{2}\left(\frac{n}{kd}+1\right)\frac{n}{kd}=:dG\left(\frac{n}{kd}\right).$$

由此，就得到如下表达式：

$$f(n)=n\sum_{k\mid n}\sum_{d}\mu(d)\left[d\mid\frac{n}{k}\right]dG\left(\frac{n}{kd}\right).$$

在枚举公因数之后，这样形式的二重求和式很常见．对于它，同样有固定的处理方法：将乘积设为新变量 $\ell=kd$，然后再次交换求和次序．因为 $d\mid(n/k)$ 就相当于 $d\mid\ell\mid n$，所以，原式变形为：

$$f(n)=n\sum_{\ell\mid n}G\left(\frac{n}{\ell}\right)\sum_{d\mid\ell}\mu(d)d.$$

设 $F(\ell)=\sum_{d\mid\ell}\mu(d)d$，则原式具有形式：

$$f(n)=n\sum_{\ell\mid n}G\left(\frac{n}{\ell}\right)F(\ell).$$

因为 $\mu(d)d$ 是积性函数，所以它和常值函数 1 的卷积 $F(n)$ 也是积性函数．尽管上述表达式中，求和式呈现 Dirichlet 卷积的形式，但是 $G(n)$ 并非积性函数，所以这一求和式的整体并非积性函数．但是，$G(n)$ 是多项式，所以它其实是若干完全积性函数的线性组合．所以，有

$$f(n)=\frac{1}{2}n\left(\sum_{\ell\mid n}\left(\frac{n}{\ell}\right)^2F(\ell)+\sum_{\ell\mid n}\frac{n}{\ell}F(\ell)\right).$$

这两项（不包含系数）都是积性函数，可以直接通过线性筛预处理（或者也可以线性筛出内层函数后，用 Dirichlet 前缀和在 $O(N\log\log N)$ 时间内预处理）．具体地，设

$$H_s(n)=\sum_{\ell|n}\left(\frac{n}{\ell}\right)^s F(\ell),\ s=1,2.$$

要推导它们的表达式，只需要确定它们在素数幂处的取值即可．为此，对于素数 p 和正指数 e，有

$$F(p^e)=\mu(1)+\mu(p)p+\sum_{j=2}^{e}\mu(p^j)p^j=1-p,$$

$$H_s(p^e)=(p^e)^sF(1)+\sum_{j=1}^{e}(p^{e-j})^sF(p^j)=p^{es}+(1-p)\frac{1-p^{es}}{1-p^s},\ s=1,2.$$

特别地，$H_1(p^e)=1$ 是常值函数，而

$$H_2(p^e)=p^{2e}+(1-p)\frac{1-p^{2e}}{1-p^2}=H_2(p^{e-1})+p^{2e}-p^{2e-1}.$$

这就很容易通过线性筛求解．在线性筛预处理出 $H_2(n)$ 后，单次询问可以通过表达式 $f(n)=(n/2)(H_2(n)+1)$ 在 $O(1)$ 时间内求解．总的时间复杂度为 $O(N+T)$，其中，N 为 n 的上界，T 为数据组数．

参考实现中，利用本题表达式的特殊性，对线性筛部分做了进一步推导，这并非必需的．仅利用素数幂处的取值，仍然可以在 $O(N)$ 时间内完成预处理．这些推导详见解答二．

解答二

就本题而言，有着更为灵活的处理方法．从解答一可以看出

$$f(n)=n\sum_{k|n}\sum_{i=1}^{n/k}i\left[\gcd\left(i,\frac{n}{k}\right)=1\right]=n\sum_{k|n}F\left(\frac{n}{k}\right).$$

如果在这一步不继续做莫比乌斯反演，而是观察后面的求和式实际上是不超过 $d=n/k$ 且与之互素的整数之和．对于 $d>1$，因为与 d 互素的整数成对出现，即 i 和 $d-i$ 必定同时与 d 互素，所以，有

$$F(d)=\sum_{i=1}^{d}i[i\perp d]=\sum_{i=1}^{d}(d-i)[i\perp d]=\frac{1}{2}d\sum_{i=1}^{d}[i\perp d]=\frac{1}{2}d\varphi(d).$$

对于 $d=1$，则有

$$F(d)=1=\frac{1}{2}+\frac{1}{2}d\varphi(d).$$

进而，原式可以表示为

$$f(n)=\frac{1}{2}n\left(\sum_{d|n}d\varphi(d)+1\right).$$

由于 $G(n)=\sum_{d|n}d\varphi(d)$ 是积性函数 $n\varphi(n)$ 与常值函数 1 的 Dirichlet 卷积，所以它也是积性函数，可以通过线性筛预处理．为此，只需要确定它在素数幂处的取值．对于素数 p 和正指数 e，有

$$G(p^e)=1+\sum_{i=1}^{e}p^i\cdot p^{i-1}(p-1)=G(p^{e-1})+p^{2e}-p^{2e-1}.$$

可以看出，这一表达式和解答一推导的结果是一致的．这一方法的总时间复杂度仍然是 $O(N+T)$．

最后，利用本题积性函数的表达式，可以进一步优化线性筛的计算过程．对于素数 p，有

$$G(p)=1-p+p^2.$$

线性筛的关键在于对于一般的 n，需要求出 $G(pn)$ 的取值．这进一步分为两种情形．当 $p\perp n$ 时，因为 G 是积性函数，所以

$$G(pn) = G(p)G(n).$$

否则，当 $p \mid n$ 时，设 $n = p^e m$ 且 $p \perp m$，就有

$$\begin{aligned} G(pn) &= G(p^{e+1})G(m) \\ &= G(p^e)G(m) + (p^{2e+2} - p^{2e+1})G(m) \\ &= G(n) + (p^{2e+2} - p^{2e+1})G(m). \end{aligned}$$

直接验证可知，这一表达式对于 $p \perp n$（即 $e = 0$）的情形也成立．而当 $p \mid n$ 时，还有

$$G(n) - G\left(\frac{n}{p}\right) = (p^{2e} - p^{2e-1})G(m).$$

代入上式，就得到

$$G(pn) = G(n) + p^2\left(G(n) - G\left(\frac{n}{p}\right)\right).$$

这简化了线性筛部分的计算．当然，这一推导并非必需，对于没有特殊性质的积性函数，直接利用 $G(pn) = G(p^{e+1})G(m)$ 就可以完成线性筛的计算．

参考代码

```cpp
#include <iostream>
constexpr int N = 1000000;
int tot, p[N + 5];
long long g[N + 5];
bool flg[N + 5];  // 标记数组

void solve() {
  g[1] = 1;
  for (int i = 2; i <= N; ++i) {
    if (!flg[i]) {
      p[++tot] = i;
      g[i] = (long long)1 * i * (i - 1) + 1;
    }
    for (int j = 1; j <= tot && i * p[j] <= N; ++j) {
      flg[i * p[j]] = true;
      if (i % p[j] == 0) {
        g[i * p[j]] =
            g[i] + (g[i] - g[i / p[j]]) * p[j] * p[j];  // 代入推出来的式子
        break;
      }
      g[i * p[j]] = g[i] * g[p[j]];
    }
  }
}

using std::cin;
using std::cout;

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  int T, n;
  solve();  // 预处理g数组
  cin >> T;
```

```
34   for (int i = 1; i <= T; ++i) {
35     cin >> n;
36     cout << (g[n] + 1) * n / 2 << '\n';
37   }
38   return 0;
39 }
40
```

BZOJ 2154 [国家集训队] Crash 的数字表格[3]

求值：

$$\sum_{i=1}^{n}\sum_{j=1}^{m}\operatorname{lcm}(i,j) \bmod 20101009.$$

数据范围：$1 \le n, m \le 10^7$.

解答

推导过程中忽略模数．设

$$f(n,m)=\sum_{i=1}^{n}\sum_{j=1}^{m}\operatorname{lcm}(i,j).$$

依然是将最小公倍数转换为最大公因数，枚举公因数，并应用标准的处理流程，就得到

$$\begin{aligned}
f(n,m)&=\sum_{k}\sum_{i=1}^{n}\sum_{j=1}^{m}\frac{ij}{k}[\gcd(i,j)=k]\\
&=\sum_{k}\sum_{i=1}^{\lfloor n/k\rfloor}\sum_{j=1}^{\lfloor m/k\rfloor}kij[\gcd(i,j)=1]\\
&=\sum_{k}\sum_{i=1}^{\lfloor n/k\rfloor}\sum_{j=1}^{\lfloor m/k\rfloor}kij\sum_{d}\mu(d)[d\mid i][d\mid j]\\
&=\sum_{k}k\sum_{d}\mu(d)\left(\sum_{i=1}^{\lfloor n/k\rfloor}i[d\mid i]\right)\left(\sum_{j=1}^{\lfloor m/k\rfloor}j[d\mid j]\right).
\end{aligned}$$

再次地，求和式对于 i 和 j 分离．首先计算这些内层的求和式，提取因数（即取 $i=di'$），就有

$$\sum_{i=1}^{\lfloor n/k\rfloor}i[d\mid i]=d\sum_{i=1}^{\lfloor\lfloor n/k\rfloor/d\rfloor}i=dG\left(\left\lfloor\frac{\lfloor n/k\rfloor}{d}\right\rfloor\right)=dG\left(\left\lfloor\frac{n}{kd}\right\rfloor\right).$$

其中，$G(n)=\dfrac{1}{2}n(n+1)$ 就是等差数列求和，最后一个等号利用了 下取整函数→9.11.1@p1160 的性质．对称地，对于另一个和式可以类似计算．代回前文表达式，就有

$$f(n,m)=\sum_{k}k\sum_{d}\mu(d)d^2G\left(\left\lfloor\frac{n}{kd}\right\rfloor\right)G\left(\left\lfloor\frac{m}{kd}\right\rfloor\right).$$

和前文的情形一致，对于这类枚举公因数的式子，往往都需要枚举乘积 $\ell=kd$，再次交换求和次序：

$$f(n,m)=\sum_{\ell}\left(\sum_{d\mid\ell}\mu(d)d\ell\right)G\left(\left\lfloor\frac{n}{\ell}\right\rfloor\right)G\left(\left\lfloor\frac{m}{\ell}\right\rfloor\right).$$

设

$$F(\ell) = \sum_{d|\ell} \mu(d) d\ell.$$

这是积性函数 ℓ 与积性函数 $\sum_{d|\ell} \mu(d)d$ 的乘积, 所以也是积性函数, 可以直接用线性筛预处理, 并预处理出它的前缀和. 然后, 就可以用数论分块计算 $f(n, m)$ 的值. 总的时间复杂度为 $O(\min\{n, m\})$.

参考代码

```cpp
#include <algorithm>
#include <iostream>
#include <vector>

int main() {
  constexpr int M = 20101009;
  int n, m;
  std::cin >> n >> m;
  if (n > m) std::swap(n, m);
  std::vector<int> f(n + 1), vis(n + 1), prime;
  prime.reserve(n);
  f[1] = 1;
  for (int x = 2; x <= n; ++x) {
    if (!vis[x]) {
      prime.emplace_back(x);
      f[x] = 1 - x;
    }
    for (int p : prime) {
      if (1LL * p * x > n) break;
      vis[x * p] = 1;
      f[x * p] = x % p ? (1 - p) * f[x] : f[x];
      if (x % p == 0) break;
    }
  }
  for (int x = 1; x <= n; ++x) {
    f[x] = 1LL * x * f[x] % M + M;
    f[x] = (f[x] + f[x - 1]) % M;
  }
  long long res = 0;
  for (int l = 1, r; l <= n; l = r + 1) {
    int nn = n / l, mm = m / l;
    r = std::min(n / nn, m / mm);
    int g_n = (1LL * nn * (nn + 1) / 2) % M;
    int g_m = (1LL * mm * (mm + 1) / 2) % M;
    res += 1LL * g_n * g_m % M * (f[r] - f[l - 1] + M) % M;
  }
  std::cout << (res % M) << '\n';
  return 0;
}

```

接下来的一道例题较为特殊，需要对乘积的约数个数函数进行转换.

LOJ 2185. [SDOI2015] 约数个数和[4]

T 组数据．对每组数据，求值：

$$\sum_{i=1}^{n}\sum_{j=1}^{m}\sigma_0(ij).$$

其中，$\sigma_0(n)=\sum_{d|n}1$ 表示 n 的约数个数．

数据范围：$1\le n,m,T\le 5\times 10^4$．

解答

这道题目的难点在于将 $\sigma_0(ij)$ 转换为关于最大公因数的表达式．由于 σ_0 是积性函数，可以首先考虑素数幂的情形．对于素数 p 和非负指数 e_1,e_2，设 $i=p^{e_1}$，$j=p^{e_2}$，就有

$$\sigma_0(ij)=1+e_1+e_2=\sum_{x|i}\sum_{y|j}[x\perp y].$$

对于一般情形，不妨设 $i=\prod_p i_p$ 且 $j=\prod_p j_p$，其中，i_p,j_p 分别是 i,j 的素因数分解中 p 对应的素数幂因子．进而，有

$$\sigma_0(ij)=\prod_p\sigma_0(i_pj_p)=\prod_p\sum_{x_p|i_p}\sum_{y_p|j_p}[x_p\perp y_p].$$

注意到，对于 i 的每个素数幂因子 i_p 都枚举它的因数 x_p，就相当于对 i 枚举它的因数 x 再分解出所有素数幂因子 x_p；对 j 同理．因此，利用乘法分配律，该式就有

$$\sigma_0(ij)=\sum_{x|i}\sum_{y|j}\prod_p[x_p\perp y_p]=\sum_{x|i}\sum_{y|j}[x\perp y].$$

最后一步用到了结论：$x\perp y$，当且仅当对于每个素因子 p，都有 $x_p\perp y_p$ 成立．

得到这一表达式后，就可以应用标准的处理流程：

$$\begin{aligned}\sigma_0(ij)&=\sum_{x|i}\sum_{y|j}[x\perp y]\\&=\sum_{x|i}\sum_{y|j}\sum_{d}\mu(d)[d\mid x][d\mid y]\\&=\sum_{d}\mu(d)\left(\sum_{x}[d\mid x\mid i]\right)\left(\sum_{y}[d\mid y\mid j]\right)\\&=\sum_{d}\mu(d)[d\mid i][d\mid j]\sigma_0\left(\frac{i}{d}\right)\sigma_0\left(\frac{j}{d}\right).\end{aligned}$$

最后一步推导的含义是：函数只有在 $d\mid i$ 且 $d\mid j$ 时才取非零值，且此时，枚举满足 $d\mid x\mid i$ 的 x 就相当于枚举 $\frac{i}{d}$ 的因数 $\frac{x}{d}$，枚举满足 $d\mid y\mid j$ 的 y 同理．

将这一表达式再代回原式，并交换求和次序：

$$
\begin{aligned}
f(n,m) &= \sum_{i=1}^{n}\sum_{j=1}^{m}\sigma_0(ij) \\
&= \sum_{i=1}^{n}\sum_{j=1}^{m}\sum_{d}\mu(d)[d\mid i][d\mid j]\sigma_0\left(\frac{i}{d}\right)\sigma_0\left(\frac{j}{d}\right) \\
&= \sum_{d}\mu(d)\left(\sum_{i=1}^{n}[d\mid i]\sigma_0\left(\frac{i}{d}\right)\right)\left(\sum_{j=1}^{m}[d\mid j]\sigma_0\left(\frac{j}{d}\right)\right) \\
&= \sum_{d}\mu(d)\left(\sum_{i=1}^{\lfloor n/d\rfloor}\sigma_0(i)\right)\left(\sum_{j=1}^{\lfloor m/d\rfloor}\sigma_0(j)\right).
\end{aligned}
$$

令 $G(n)=\sum_{i=1}^{n}\sigma_0(i)$，就有

$$
f(n,m)=\sum_{d}\mu(d)G\left(\left\lfloor\frac{n}{d}\right\rfloor\right)G\left(\left\lfloor\frac{m}{d}\right\rfloor\right).
$$

这可以通过数论分块求解. 只需要预处理出 $\mu(n)$ 和 $\sigma_0(n)$ 的前缀和即可. 总时间复杂度为 $O(N+T\sqrt{N})$，其中，N 为 n,m 的上界，T 为数据组数.

参考代码

```cpp
#include <algorithm>
#include <iostream>
using namespace std;
constexpr long long N = 5e4 + 5;
long long n, m, T, pr[N], mu[N], d[N], t[N],
    cnt;  // t 表示 i 的最小质因子出现的次数
bool bp[N];

void prime_work(long long k) {
  bp[0] = bp[1] = true, mu[1] = 1, d[1] = 1;
  for (long long i = 2; i <= k; i++) {  // 线性筛
    if (!bp[i]) pr[++cnt] = i, mu[i] = -1, d[i] = 2, t[i] = 1;
    for (long long j = 1; j <= cnt && i * pr[j] <= k; j++) {
      bp[i * pr[j]] = true;
      if (i % pr[j] == 0) {
        mu[i * pr[j]] = 0;
        d[i * pr[j]] = d[i] / (t[i] + 1) * (t[i] + 2);
        t[i * pr[j]] = t[i] + 1;
        break;
      } else {
        mu[i * pr[j]] = -mu[i];
        d[i * pr[j]] = d[i] << 1;
        t[i * pr[j]] = 1;
      }
    }
  }
  for (long long i = 2; i <= k; i++)
    mu[i] += mu[i - 1], d[i] += d[i - 1];  // 求前缀和
}

long long solve() {
  long long res = 0, mxi = min(n, m);
```

```cpp
  for (long long i = 1, j; i <= mxi; i = j + 1) {  // 整除分块
    j = min(n / (n / i), m / (m / i));
    res += d[n / i] * d[m / i] * (mu[j] - mu[i - 1]);
  }
  return res;
}

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  cin >> T;
  prime_work(50000);  // 预处理
  while (T--) {
    cin >> n >> m;
    cout << solve() << '\n';
  }
  return 0;
}

```

最后一道例题展示了如何应用乘法版本的莫比乌斯反演.

Luogu P5221 Product[5]

求值:

$$\prod_{i=1}^{n}\prod_{j=1}^{n}\frac{\operatorname{lcm}(i,j)}{\gcd(i,j)} \pmod{104857601}.$$

数据范围: $1 \le n \le 1 \times 10^6$.

解答一

推导过程中忽略模数. 设

$$f(n)=\prod_{i=1}^{n}\prod_{j=1}^{n}\frac{\operatorname{lcm}(i,j)}{\gcd(i,j)}.$$

依然是将最小公倍数转换为最大公因数:

$$f(n)=\prod_{i=1}^{n}\prod_{j=1}^{n}\frac{ij}{(\gcd(i,j))^2}.$$

注意, 这些因子的乘积是相互独立的, 可以分别计算. 令

$$g(n)=\prod_{i=1}^{n}\prod_{j=1}^{n}\gcd(i,j).$$

原式就等于:

$$f(n)=\frac{(n!)^{2n}}{g(n)^2}.$$

重点是解决 $g(n)$ 的计算问题. 对它的处理流程和前文描述的相仿, 但是需要换成相应的乘法版本. 首先, 枚举并提取公因数:

$$g(n)=\prod_k\prod_{i=1}^{n}\prod_{j=1}^{n}k\uparrow[\gcd(i,j)=k]$$

$$=\prod_k\prod_{i=1}^{\lfloor n/k\rfloor}\prod_{j=1}^{\lfloor n/k\rfloor}k\uparrow[\gcd(i,j)=1].$$

其中，$a\uparrow b=a^b$ 是 Knuth 箭头．然后，代入 $[\gcd(i,j)=1]=\sum_d\mu(d)[d\mid i][d\mid j]$，并将指数上的和式转换为幂的乘积式，得到：

$$g(n)=\prod_k\prod_d\prod_{i=1}^{\lfloor n/k\rfloor}\prod_{j=1}^{\lfloor n/k\rfloor}k\uparrow(\mu(d)[d\mid i][d\mid j]).$$

进一步地提取因数（即令 $i=di'$，$j=dj'$），并应用 下取整函数→9.11.1@p1160 的性质，就得到：

$$g(n)=\prod_k\prod_d\prod_{i=1}^{\lfloor n/(kd)\rfloor}\prod_{j=1}^{\lfloor n/(kd)\rfloor}k\uparrow\mu(d).$$

然后分离关于 i,j 的乘积，就发现乘式中并不含有 i,j，因此它就相当于对乘式取幂：

$$g(n)=\prod_k\prod_d k\uparrow\left(\mu(d)\left\lfloor\frac{n}{kd}\right\rfloor^2\right).$$

因为前面枚举了公因数，所以对于这个式子需要再次交换求乘积的次序．令 $\ell=kd$，有：

$$g(n)=\prod_\ell\prod_{d\mid\ell}\left(\frac{\ell}{d}\right)\uparrow\left(\mu(d)\left\lfloor\frac{n}{\ell}\right\rfloor^2\right)$$

$$=\prod_\ell\left(\prod_{d\mid\ell}\left(\frac{\ell}{d}\right)\uparrow\mu(d)\right)\uparrow\left\lfloor\frac{n}{\ell}\right\rfloor^2.$$

设

$$F(n)=\prod_{d\mid n}\left(\frac{n}{d}\right)\uparrow\mu(d).$$

容易发现这是关于 $\tilde{F}(n)=n$ 的乘积形式莫比乌斯反演．即使不知道它的表达式，也可以应用 Dirichlet 差分→9.11@p1160 方法在 $O(n\log\log n)$ 时间内预处理．当然，由于 $\tilde{F}(n)$ 的形式非常简单，$F(n)$ 的表达式可以直接求出：

$$F(n)=\begin{cases}p,n=p^e,\ p\in\mathbf{P},\ e\in\mathbf{N}_+,\\1,\text{otherwise.}\end{cases}$$

von Mangoldt 函数→9.11@p1160 就是它的自然对数．得到 $F(n)$ 的取值后，直接应用乘积版本的数论分块就可以在 $O(\sqrt{n})$ 时间内求出 $g(n)$ 的取值，进而得到 $f(n)$ 的取值．总的时间复杂度为 $O(n)$．

值得注意的是，涉及乘积的计算时，往往需要用到 欧拉定理→9.11.9@p1223，因此指数部分取模用到的模数与题目所给的模数并不相同．

解答二

乘积版本推导的难点在于对乘积和幂次的处理相对陌生，因此，对于这类问题，也可以取对数后再推导．对于本题，仅考虑 $g(n)$ 的推导．将它取对数后，有：

$$\log g(n)=\sum_{i=1}^{n}\sum_{j=1}^{n}\log\gcd(i,j).$$

对于这类含有最大公因数的式子，直接应用标准的推导流程，就得到：

$$
\begin{aligned}
\log g(n) &= \sum_k \log k \sum_{i=1}^{n} \sum_{j=1}^{n} [\gcd(i,j) = k] \\
&= \sum_k \log k \sum_{i=1}^{\lfloor n/k \rfloor} \sum_{j=1}^{\lfloor n/k \rfloor} [\gcd(i,j) = 1] \\
&= \sum_k \log k \sum_d \mu(d) \left(\sum_{i=1}^{\lfloor n/k \rfloor} [d \mid i] \right) \left(\sum_{j=1}^{\lfloor n/k \rfloor} [d \mid j] \right) \\
&= \sum_k \log k \sum_d \mu(d) \left\lfloor \frac{n}{kd} \right\rfloor^2 \\
&= \sum_\ell \left(\sum_{d \mid \ell} \mu(d) \log \frac{\ell}{d} \right) \left\lfloor \frac{n}{\ell} \right\rfloor^2 \\
&= \sum_\ell \Lambda(\ell) \left\lfloor \frac{n}{\ell} \right\rfloor^2 .
\end{aligned}
$$

其中，$\Lambda(n)$ 是 von Mangoldt 函数→9.11@p1160．将这一推导结果取幂，就得到解答一的结果．

参考代码

```cpp
#include <algorithm>
#include <iostream>
#include <vector>

int pow(int a, int b, int m) {
  int res = 1;
  for (int po = a; b; b >>= 1) {
    if (b & 1) res = 1LL * res * po % m;
    po = 1LL * po * po % m;
  }
  return res;
}

int main() {
  constexpr int M = 104857601;
  int n;
  std::cin >> n;
  // Get F(n), i.e., exp(Lambda(n)).
  std::vector<int> vis(n + 1), prime, pf(n + 1), rem(n + 1);
  prime.reserve(n);
  for (int x = 2; x <= n; ++x) {
    if (!vis[x]) {
      prime.emplace_back(x);
      pf[x] = x;
      rem[x] = 1;
    }
    for (int p : prime) {
      if (1LL * p * x > n) break;
      vis[x * p] = 1;
      pf[x * p] = p;
      rem[x * p] = x % p ? x : rem[x];
```

```
32        if (x % p == 0) break;
33      }
34    }
35    pf[1] = 1;
36    for (int x = 2; x <= n; ++x) {
37      pf[x] = rem[x] == 1 ? pf[x] : 1;
38    }
39    // Prefix products and their inverses.
40    std::vector<int> pm(n + 1), ip(n + 1);
41    pm[0] = 1;
42    for (int x = 1; x <= n; ++x) {
43      pm[x] = 1LL * pm[x - 1] * pf[x] % M;
44    }
45    int inv = pow(pm[n], M - 2, M);
46    ip[0] = 1;
47    for (int x = n; x >= 1; --x) {
48      ip[x] = inv;
49      inv = 1LL * inv * pf[x] % M;
50    }
51    // Calculate g(n).
52    int res = 1;
53    for (int l = 1, r; l <= n; l = r + 1) {
54      r = n / (n / l);
55      int a = 1LL * pm[r] * ip[l - 1] % M;
56      int b = 1LL * (n / l) * (n / l) % (M - 1);
57      res = 1LL * res * pow(a, b, M) % M;
58    }
59    // Take square and then inverse.
60    res = pow(res, M - 3, M);
61    // Get factorials.
62    int fac = 1;
63    for (int x = 1; x <= n; ++x) {
64      fac = 1LL * fac * x % M;
65    }
66    // Final answer.
67    res = 1LL * res * pow(fac, 2 * n, M) % M;
68    std::cout << res << std::endl;
69    return 0;
70  }
71
```

习题

- Luogu P3312 [SDOI2014] 数表[6]
- Luogu P3700 [CQOI2017] 小 Q 的表格[7]
- Luogu P3704 [SDOI2017] 数字表格[8]
- Luogu P3768 简单的数学题[9]
- Luogu P4464 [国家集训队] JZPKIL[10]
- Luogu P4619 [SDOI2018] 旧试题[11]
- Luogu P5518 [MtOI2019] 幽灵乐团[12]
- Luogu P6222 简单题 加强版[13]
- Luogu P6825「EZEC-4」求和[14]

- Luogu P7486「Stoi2031」彩虹[15]
- AtCoder Grand Contest 038 C - LCMs[16]
- Codeforces 1139 D. Steps to One[17]

参考文献

- Möbius function - Wikipedia[18]
- Möbius inversion formula - Wikipedia[19]
- Von Mangoldt function - Wikipedia[20]
- algocode 算法博客[21]

外部参考资料

[1] Luogu P2522 [HAOI 2011] Problem b

[2] SPOJ LCMSUM

[3] BZOJ 2154 [国家集训队] Crash 的数字表格

[4] LOJ 2185. [SDOI2015] 约数个数和

[5] Luogu P5221 Product

[6] Luogu P3312 [SDOI2014] 数表

[7] Luogu P3700 [CQOI2017] 小 Q 的表格

[8] Luogu P3704 [SDOI2017] 数字表格

[9] Luogu P3768 简单的数学题

[10] Luogu P4464 [国家集训队] JZPKIL

[11] Luogu P4619 [SDOI2018] 旧试题

[12] Luogu P5518 [MtOI2019] 幽灵乐团

[13] Luogu P6222 简单题 加强版

[14] Luogu P6825「EZEC-4」求和

[15] Luogu P7486「Stoi2031」彩虹

[16] AtCoder Grand Contest 038 C - LCMs

[17] Codeforces 1139 D. Steps to One

[18] Möbius function - Wikipedia

[19] Möbius inversion formula - Wikipedia

[20] Von Mangoldt function - Wikipedia

[21] algocode 算法博客

9.11.24 狄利克雷双曲线法 & 杜教筛

Authors: Backl1ght, c-forrest, Enter-tainer, Great-designer, Henry-ZHR, huayucaiji, hsfzLZH1, Ir1d, kenlig, ksyx, Marcythm, MegaOwIer, Menci, Nanarikom, nanmenyangde, ouuan, purple-vine, shawlleyw, sshwy, StudyingFather, Tiphereth-A, Xeonacid

前置知识：Dirichlet 卷积→9.11.22@p1322、数论分块→9.11.21@p1314

本文介绍 Dirichlet 双曲线法，它可用于在亚线性时间复杂度内计算两个数论函数的 Dirichlet 卷积的前缀和. 通过对该算法的分析，可以引申出块筛的概念；据此，本文将进一步介绍块筛卷积的快速计算方法. 最后，本文将介绍杜教筛，它可以用于计算两个数论函数在 Dirichlet 卷积意义下的商的前缀和.

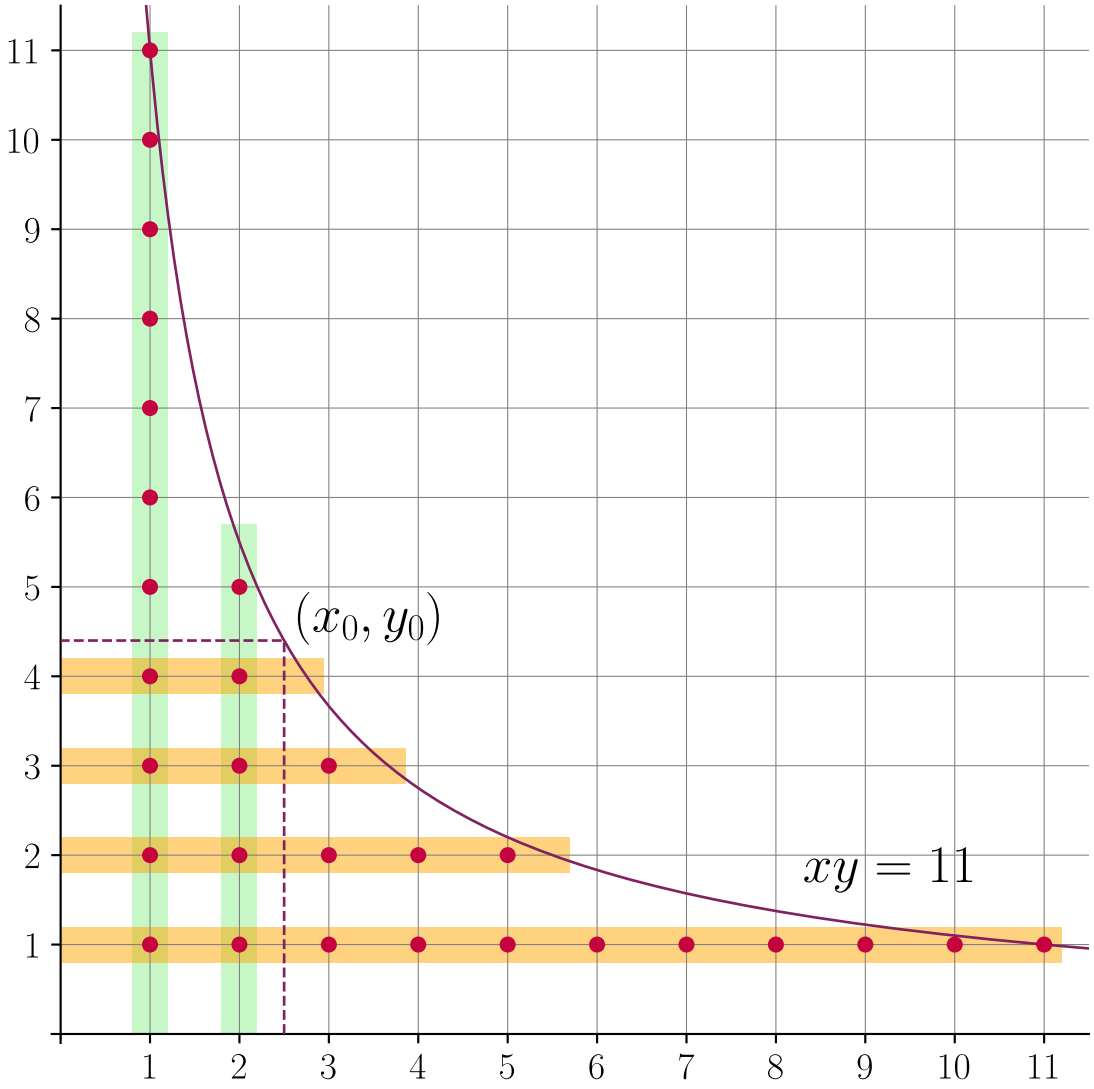
Dirichlet 双曲线法

Dirichlet 双曲线法可以用于计算两个数论函数的 Dirichlet 卷积的前缀和.

设 f, g, h 是数论函数，且 $h = f * g$. 那么，利用 Dirichlet 卷积的定义，h 的前缀和

$$H(n) = \sum_{k=1}^{n} h(k) = \sum_{k=1}^{n} \sum_{xy=k} f(x)g(y).$$

求和式遍历的点集恰为第一象限（不含坐标轴）中双曲线 $xy = n$ 下方的整点集合. 设整点 (x, y) 的权值为 $f(x)g(y)$，那么 $H(n)$ 就是这一权值的和.



如图所示，这一权值和可以通过容斥原理计算：

$$H(n) = \sum_{x=1}^{\lfloor x_0 \rfloor} f(x) G\left(\left\lfloor \frac{n}{x} \right\rfloor\right) + \sum_{y=1}^{\lfloor y_0 \rfloor} F\left(\left\lfloor \frac{n}{y} \right\rfloor\right) g(y) - F(\lfloor x_0 \rfloor) G(\lfloor y_0 \rfloor).$$

其中，F, G 分别是 f, g 的前缀和函数，(x_0, y_0) 是双曲线 $xy = n$ 上任意一个点．表达式中，第一项表示图中绿色区域的权值和，第二项表示图中橙色区域的权值和，第三项则表示两个区域重叠部分的权值和．这个表达式仅含有 $\lfloor x_0 \rfloor + \lfloor y_0 \rfloor + 1$ 项．对于合理选择的 (x_0, y_0)，它的计算复杂度显著优于暴力计算 $h(n)$ 的前缀和．这就是 **Dirichlet 双曲线法**（Dirichlet hyperbola method）．

卷积前缀和点值的计算

Dirichlet 双曲线法最基本的应用就是计算前缀和函数的点值 $H(n)$．

如果 F, G 的点值已知（或可以在 $O(1)$ 时间内计算），进而 f, g 的点值也已知，那么 Dirichlet 双曲线法的表达式中每一项都可以在 $O(1)$ 时间内计算，总时间复杂度就等于 $O(x_0 + y_0)$．因为 $x_0 y_0 = n$，所以由均值不等式可知，当 $x_0 = y_0 = \sqrt{n}$ 时，就得到最低时间复杂度 $O(\sqrt{n})$．

这并非新的结果．在 Dirichlet 双曲线法的表达式中，令 $x_0 > n$，就得到

$$H(n) = \sum_{x=1}^{n} f(x) G\left(\left\lfloor \frac{n}{x} \right\rfloor\right).$$

利用 数论分块→9.11.21@p1314 的技巧，当 F,G 的点值已知时，该式可以在 $O(\sqrt{n})$ 时间内计算．这实际上和本节得到的算法是几乎等价的：这两种方法需要的已知信息基本类似，计算的表达式也大致相同．

等价性的说明

细究数论分块的计算过程可以发现，实际计算的表达式为

$$H(n)=\sum_{y\in D(n)}\left(F\left(\left\lfloor\frac{n}{y}\right\rfloor\right)-F\left(\left\lfloor\frac{n}{y+1}\right\rfloor\right)\right)G(y).$$

其中，$D(n)=\left\{\left\lfloor\frac{n}{x}\right\rfloor:1\le x\le n,\ x\in\mathbf{N}_+\right\}$ 是数论分块中的关键点集合．

根据数论分块的性质可知，对于 $x\le\sqrt{n}$，对应分块的高度 $y=\lfloor n/x\rfloor$ 各不相同，于是这些分块的长度都是 1，亦即

$$\left\lfloor\frac{n}{y+1}\right\rfloor+1=x=\left\lfloor\frac{n}{y}\right\rfloor.$$

因此，这些分块对应的求和为

$$I_1=\sum_{x=1}^{\lfloor\sqrt{n}\rfloor}(F(x)-F(x-1))G\left(\left\lfloor\frac{n}{x}\right\rfloor\right)=\sum_{x=1}^{\lfloor\sqrt{n}\rfloor}f(x)G\left(\left\lfloor\frac{n}{x}\right\rfloor\right).$$

剩下的分块，高度 y 遍历 $1,2,\cdots,y^*=\left\lfloor\frac{n}{\lfloor\sqrt{n}\rfloor+1}\right\rfloor$．对剩余的和式应用 Abel 变换[1]（即分部积分法的求和形式），就得到

$$\begin{aligned}I_2&=\sum_{y=1}^{y^*}\left(F\left(\left\lfloor\frac{n}{y}\right\rfloor\right)-F\left(\left\lfloor\frac{n}{y+1}\right\rfloor\right)\right)G(y)\\&=\sum_{y=1}^{y^*}F\left(\left\lfloor\frac{n}{y}\right\rfloor\right)g(y)-F\left(\left\lfloor\frac{n}{y^*+1}\right\rfloor\right)G(y^*).\end{aligned}$$

数论分块的性质说明，$\lfloor n/\lfloor\sqrt{n}\rfloor\rfloor\ge\lfloor\sqrt{n}\rfloor$．由此，分两种情形：

- 当 $\lfloor n/\lfloor\sqrt{n}\rfloor\rfloor=\lfloor\sqrt{n}\rfloor$ 时，必然有 $y^*<\lfloor\sqrt{n}\rfloor$．由于它与 $x=\lfloor\sqrt{n}\rfloor$ 所在块相邻，块高只能恰好相差一，即 $y^*=\lfloor\sqrt{n}\rfloor-1$．此时，有

$$\begin{aligned}I_2&=\sum_{y=1}^{\lfloor\sqrt{n}\rfloor-1}F\left(\left\lfloor\frac{n}{y}\right\rfloor\right)g(y)-F(\lfloor\sqrt{n}\rfloor)G(\lfloor\sqrt{n}\rfloor-1)\\&=\sum_{y=1}^{\lfloor\sqrt{n}\rfloor-1}F\left(\left\lfloor\frac{n}{y}\right\rfloor\right)g(y)-F(\lfloor\sqrt{n}\rfloor)(G(\lfloor\sqrt{n}\rfloor)-g(\lfloor\sqrt{n}\rfloor))\\&=\sum_{y=1}^{\lfloor\sqrt{n}\rfloor}F\left(\left\lfloor\frac{n}{y}\right\rfloor\right)g(y)-F(\lfloor\sqrt{n}\rfloor)G(\lfloor\sqrt{n}\rfloor).\end{aligned}$$

- 当 $\lfloor n/\lfloor\sqrt{n}\rfloor\rfloor>\lfloor\sqrt{n}\rfloor$ 时，必然有 $y^*=\lfloor\sqrt{n}\rfloor$．直接代入求和式，就得到

$$I_2=\sum_{y=1}^{\lfloor\sqrt{n}\rfloor}F\left(\left\lfloor\frac{n}{y}\right\rfloor\right)g(y)-F(\lfloor\sqrt{n}\rfloor)G(\lfloor\sqrt{n}\rfloor).$$

综上，除了一个 Abel 变换，数论分块的计算过程实际上就是在计算如下算式：

$$H(n)=\sum_{x=1}^{\lfloor\sqrt{n}\rfloor}f(x)G\left(\left\lfloor\frac{n}{x}\right\rfloor\right)+\sum_{y=1}^{\lfloor\sqrt{n}\rfloor}F\left(\left\lfloor\frac{n}{y}\right\rfloor\right)g(y)-F(\lfloor\sqrt{n}\rfloor)G(\lfloor\sqrt{n}\rfloor).$$

这就是 $(x_0,y_0)=(\sqrt{n},\sqrt{n})$ 时 Dirichlet 双曲线法的表达式．因此可以说，两种算法的计算过程几乎等价．而且，由于 Dirichlet 双曲线法利用了更多数论分块的性质，避免了朴素数论分块中的不必要计算，常数更小一些．

在处理实际问题时，已知 F,G 点值这一条件可能过强．但是，Dirichlet 双曲线法（或对应的数论分块）其实并不需要 F,G 的全部点值信息．观察求和表达式可知，其实只需要 F 和 G 在数论分块关键点集合

$$D(n)=\left\{\left\lfloor\frac{n}{x}\right\rfloor:1\le x\le n,\ x\in\mathbf{N}_+\right\}$$

处的取值．它既是全体块高的集合，也是全体块的右端点集合．注意，根据数论分块的 性质→9.11.21@p1314 可知，这一集合包含所有 $1\le x\le\sqrt{n}$ 的整数值，因此，已知 F,G 在 $D(n)$ 处的取值就相当于已知 f,g 在所有 $1\le x\le\sqrt{n}$ 处的取值．这一集合的大小是 $\Theta(\sqrt{n})$ 的，所以，计算 $H(n)$ 时，只需要知道 F,G 在一个稀疏集合处的点值信息．这个观察是优化数论函数前缀和计算的关键．

块筛及其卷积

有些时候，$h=f*g$ 并非最终需要计算前缀和的函数，而只是中间步骤之一．根据前文分析，为了后续计算，需要求出前缀和函数 H 在集合 $D(n)$ 处的取值．这就称为数论函数 h 的 **块筛**：

$$\mathcal{S}_h(n)=\{H(x):x\in D(n)\}.$$

本节讨论 **块筛卷积** 问题的计算方法：已知 f,g 的块筛时，求它们的 Dirichlet 卷积 $h=f*g$ 的块筛．

朴素算法

朴素算法就是将块筛的计算看作是 $|D(n)|$ 次前缀和点值的计算．这样做的总时间复杂度为

$$\begin{aligned}O\left(\sum_{d\in D(n)}\sqrt{d}\right)&=O\left(\sum_{x=1}^{\lfloor\sqrt{n}\rfloor}\sqrt{x}+\sum_{x=1}^{\lfloor\sqrt{n}\rfloor}\sqrt{\frac{n}{x}}\right)\\&=O\left(\int_1^{\sqrt{n}}\sqrt{x}\mathrm{d}x+\int_1^{\sqrt{n}}\sqrt{\frac{n}{x}}\mathrm{d}x\right)\\&=O(n^{3/4}).\end{aligned}$$

正是因为块筛是稀疏的，所以整个块筛可以在亚线性时间内求出．

但是，这一算法显然过于暴力．集合 $D(n)$ 中较小的那些元素取值相对稠密，所以块筛中相邻两个前缀和相差并不大，完全可以直接计算卷积 h 的点值，再求它的前缀和．这比起对每个点都单独求一遍前缀和点值要更快．例如，对 $x=1,2,\cdots,\lfloor\sqrt{n}\rfloor$ 分别计算前缀和点值需要

$$O\left(\sum_{x=1}^{\lfloor\sqrt{n}\rfloor}\sqrt{x}\right)=O(n^{3/4})$$

的时间；但是，直接计算 h 在这些点处的点值再累和，却只需要 $O(n^{1/2}\log n)$ 的时间．尽管如此，如果只知道 f,g 的块筛而不知道更多信息，那么这一思路是无法继续优化该算法复杂度的．这是因为块筛中只包含 $x\le\sqrt{n}$ 处的点值信息，至多只能计算 h 在 $1\le x\le\sqrt{n}$ 处的点值，而剩余前缀和点值的计算仍然需要 $O(n^{3/4})$ 的时间．

利用点值信息优化

如果已知信息除了 f,g 的块筛外，还包含它们的更多点值，那么，确实可以改进这一算法的复杂度．实践中，这一算法通常应用于 f,g 的点值可以快速预处理时．

选择 $z\ge\sqrt{n}$．卷积 h 的块筛分为两部分：

- 计算 $f*g$ 的卷积 h 在 $1\le x\le z$ 处的点值，再直接累加求和得到 H 在 $1\le x\le z$ 处的点值．
- 对于 $x\in D(n)$ 且 $x>z$，通过 Dirichlet 双曲线法计算 H 在 x 处的点值．

对于一般情形，这样做的时间复杂度为

$$O\left(z\log z+\sum_{d\in D(n),\ d\geq z}\sqrt{d}\right)=O\left(z\log z+\sum_{x=1}^{n/z}\sqrt{\frac{n}{x}}\right)$$
$$=O\left(z\log z+\int_1^{n/z}\sqrt{\frac{n}{x}}\mathrm{d}x\right)$$
$$=O\left(z\log z+\frac{n}{\sqrt{z}}\right).$$

当 $z=\left(\dfrac{n}{\log n}\right)^{2/3}$ 时，总时间复杂度最小为 $O\left(n^{2/3}(\log n)^{1/3}\right)$.

当然，Dirichlet 卷积 点值计算→9.11.22@p1322 的复杂度与 f,g,h 的性质有关. 对于 f,g,h 有特殊性质的情形，最优分点和复杂度均略有不同：

- 如果 f 或 g 是积性的，那么当 $z=\left(\dfrac{n}{\log\log n}\right)^{2/3}$ 时，总时间复杂度最小为 $O\left(n^{2/3}(\log\log n)^{1/3}\right)$.
- 如果 h 是积性的，那么当 $z=n^{2/3}$ 时，总时间复杂度最小为 $O(n^{2/3})$.

应用这一优化并不需要 f,g 的全部点值，而只需要它们在 $1\leq x\leq z$ 处的点值. 因为这一算法也得到了 h 在 $1\leq x\leq z$ 处的点值，所以当 h 作为中间变量时，同样可以利用 h 的点值优化后续计算过程. 数论函数的块筛，再加上它这些点值，就构成了一个增强版的块筛：它们是在 $O(n^{2/3+\varepsilon})$ 时间内计算卷积前缀和的全部必要信息.

快速块筛卷积

前置知识：快速傅里叶变换→9.12.3@p1527

> **⚠ 注意**
>
> 初学者可以跳过本节内容.

本节讨论周康阳在 2024 年集训队论文提出的快速块筛卷积算法. 它可以在 $O(\sqrt{n}\log^2 n)$ 时间内根据块筛 $\mathcal{S}_f$ 和 $\mathcal{S}_g$ 计算出它们卷积 $\mathcal{S}_h$ 的取值. 这一算法不依赖于额外的点值信息和数论函数的积性，但实现较为复杂.

块筛卷积问题希望计算

$$h(z)=\sum_{xy=z}f(x)g(y)$$

在块筛 $D(n)=\{\lfloor n/t\rfloor:1\leq t\leq n\}$ 处的前缀和. 对于这一问题，单一贡献可以由 (x,y,t) 标记，即将项 $f(x)g(y)$ 累加到 $\lfloor n/t\rfloor$ 处前缀和的过程. 算法将这些贡献分成若干组处理.

首先，考虑 $x>\sqrt{n}$ 的所有点对于这一系列前缀和的贡献，那些 $y>\sqrt{n}$ 的贡献类似. 因为所有贡献必须满足 $xy\leq\lfloor n/t\rfloor$，即 $xyt\leq n$，所以只需要枚举所有可能的 t,y，利用前缀和技巧以及块筛 $\mathcal{S}_f$ 中的信息，就可以在 $O(\sum_{t,y:ty\leq\sqrt{n}}1)=O(\sqrt{n}\log n)$ 时间内计算出这部分贡献.

然后，考虑 $\lfloor n/t\rfloor\leq\sqrt{n}$ 的这部分贡献. 这一部分同样可以暴力枚举所有可能的 x,y 完成，时间复杂度仍然是 $O(\sum_{x,y:xy\leq\sqrt{n}}1)=O(\sqrt{n}\log n)$. 这一部分实际上得到了函数 h 在 $D(n)$ 的前 $\lfloor\sqrt{n}\rfloor$ 个点值.

考虑剩下的贡献，即满足 $x,y\leq\sqrt{n}$ 且 $\lfloor n/t\rfloor>\sqrt{n}$ 的贡献. 所有贡献必须满足 $xyt\leq n$，亦即 $\ln x+\ln y\leq\ln(n/t)$. 取正数 S，可以利用 $\lceil S\ln x\rceil+\lceil S\ln y\rceil\leq S\ln(n/t)$ 近似估计这一条件. 定义多项式 $\sigma_f(u)$ 和 $\sigma_g(u)$，使得它们的系数 $[u^k]\sigma_f$ 和 $[u^k]\sigma_g$ 分别等于满足 $\lceil S\ln x\rceil=k$ 时 $f(x)$ 的和以及满足 $\lceil S\ln y\rceil=k$ 时 $g(y)$ 的和（只考虑 $x,y\leq\sqrt{n}$ 的这一部分）. 利用快速傅里叶变换（FFT）可以得到它们的乘积 $\sigma_f\sigma_g$，它的系数 $[u^k](\sigma_f\sigma_g)$ 就表示 $\lceil S\ln x\rceil+\lceil S\ln y\rceil=k$ 时 $f(x)g(y)$ 的和. 由此，只需要对于每个 $D(n)$ 中剩下的每个 $\lfloor n/t\rfloor$ 都找到满足 $k\leq S\ln(n/t)$ 的最大 k 值，就可以得到这一部分贡献的估计值.

最后，对于这一部分贡献进行误差修正. 前文试图利用更强的条件 $\lceil S\ln x\rceil+\lceil S\ln y\rceil\leq S\ln(n/t)$ 去近似 $\ln x+\ln y\leq\ln(n/t)$，可能会遗漏部分贡献. 这只会发生在 $\lceil S\ln x\rceil+\lceil S\ln y\rceil>S\ln(n/t)$ 时，所以有

$$S\ln x+1+S\ln y+1\geq\lceil S\ln x\rceil+\lceil S\ln y\rceil>S\ln(n/t)\geq S\ln x+S\ln y.$$

这相当于

$$xyt\in(ne^{-2/S},n].$$

这是一个长度为 $O(n/S)$ 的区间. 枚举区间内所有可能的贡献 (x,y,t), 逐个检验是否遗漏, 就可以完成误差的修正. 为了快速枚举区间内所有贡献, 可以首先筛出不超过 $\sqrt{n}$ 的全部素数, 再利用这些素数去除区间中的整数, 剩下的因子必然是大于 $\sqrt{n}$ 的素数, 由此就可以得到区间内所有整数的素因数分解, 进而可以快速枚举所有可能的 (x,y,t).

考察最后这一部分贡献计算的时间复杂度. 估计贡献时, 需要对长度为 $S\log n$ 的多项式做乘法, 时间复杂度为 $O(S\log n\log(S\log n))$. 误差修正时, 预处理素因数分解的时间复杂度为 $O(\sqrt{n}+(n/S)\log\log n)$, 枚举区间内所有贡献的时间复杂度为 $O(\sum_{k\in(ne^{-2/S},n]}d_3(k))$, 此处, $d_3(n)$ 表示将 n 分解成三个有序整数乘积的方法数. 解析数论的结果 [50] 指出:

$$\sum_{k\leq n}d_3(k)=nP(\log n)+O(n^{43/96+\varepsilon}),$$

其中, $P(\cdot)$ 是二次多项式. 因为前两步时间复杂度已经是 $O(\sqrt{n}\log n)$, 忽略所有 $o(\sqrt{n}\log n)$ 的项, 最后这一部分贡献计算的时间复杂度为

$$O\left(S\log n\log(S\log n)+\frac{n}{S}\log^2 n\right).$$

取 $S=\sqrt{n}$, 就得到时间复杂度为 $O(\sqrt{n}\log^2 n)$. 这也是该算法的总时间复杂度.

参考实现

```cpp
struct BlockSieve {
  long long n, b;
  std::vector<int> s1, s2;

  BlockSieve(long long _n)
      : n(_n), b(std::sqrt(_n + 0.25l)), s1(b + 1), s2(b + 1) {}

  int& operator[](long long x) { return x <= b ? s1[x] : s2[n / x]; }
};

// Fast block sieve convolution by ZKY.
BlockSieve block_sieve_convolute(const BlockSieve& f, const BlockSieve& g) {
  // Special cases for n <= 3 are not implemented.
  // The upper bound is a loose bound for NTT to work for M = 998244353.
  assert(f.n == g.n && f.n > 3 && f.n < 1e11);
  long long n = f.n, b = f.b;
  BlockSieve h(n);
  std::vector<int> df(b + 1), dg(b + 1);
  for (int i = 1; i <= b; ++i) {
    df[i] = (f.s1[i] + M - f.s1[i - 1]) % M;
    dg[i] = (g.s1[i] + M - g.s1[i - 1]) % M;
  }
  // Prefix sum trick for x or y > sqrt(n).
  for (int t = 1; t <= n / (b + 1); ++t) {
    long long x;
    for (int y = 1; (x = n / (t * y)) > b; ++y) {
      (h.s2[t] += (long long)dg[y] * (f.s2[n / x] + M - f.s1[b]) % M) %= M;
```

[50] Piltz divisor problem - Divisor summatory function - Wikipedia[2]

```
      (h.s2[t] += (long long)df[y] * (g.s2[n / x] + M - g.s1[b]) % M) %= M;
    }
  }
  // Brutal force for small x*y.
  for (int x = 1; x <= b; ++x) {
    for (int y = 1; x * y <= b; ++y) {
      (h.s1[x * y] += (long long)df[x] * dg[y] % M) %= M;
    }
  }
  for (int i = 1; i <= b; ++i) {
    (h.s1[i] += h.s1[i - 1]) %= M;
  }
  // Approximation.
  std::vector<int> id1(b + 1);
  for (int i = 1; i <= b; ++i) {
    id1[i] = std::ceil(b * std::log(i));
  }
  std::vector<long double> id2(b + 1);
  for (int i = 1; i <= b; ++i) {
    id2[i] = b * std::log(1.0l * n / i);
  }
  std::vector<int> sf(id1[b] + 1), sg(id1[b] + 1);
  for (int i = 1; i <= b; ++i) {
    (sf[id1[i]] += df[i]) %= M;
    (sg[id1[i]] += dg[i]) %= M;
  }
  auto sh = ntt_mul(sf, sg);
  for (int i = 1; i < (int)sh.size(); ++i) {
    (sh[i] += sh[i - 1]) %= M;
  }
  for (int t = 1; t <= n / (b + 1); ++t) {
    int k = std::min((int)std::floor(id2[t]), (int)sh.size() - 1);
    (h.s2[t] += sh[k]) %= M;
  }
  // Correction.
  auto primes = get_primes(b);  // Obtain primes not exceeding b.
  long long l = n * std::exp(-2.0l / b);
  std::vector<long long> co(n - l + 1);
  std::iota(co.begin(), co.end(), l);
  std::vector<std::vector<std::pair<long long, int>>> fac(n - l + 1);
  for (int p : primes) {
    for (int i = (l + p - 1) / p * p - l; i <= n - l; i += p) {
      int e = 0;
      for (; co[i] % p == 0; co[i] /= p) ++e;
      fac[i].emplace_back(p, e);
    }
  }
  for (int i = 0; i <= n - l; ++i) {
    if (co[i] != 1) {
      fac[i].emplace_back(co[i], 1);
    }
  }
```

```
  long long num = 0;
  std::vector<std::pair<long long, int>> cur;
  auto dfs = [&](auto&& dfs, int k, int x, int y) {
    if (k == (int)cur.size()) {
      auto t = num / x / y;
      if (t && t <= n / (b + 1) && id1[x] + id1[y] > id2[t]) {
        (h.s2[t] += (long long)df[x] * dg[y] % M) %= M;
      }
      return;
    }
    auto p = cur[k].first;
    auto e = cur[k].second;
    long long nx = x;
    for (int e1 = 0; e1 <= e; ++e1) {
      if (e1) {
        nx *= p;
        if (nx > b) break;
      }
      long long ny = y;
      for (int e2 = 0; e1 + e2 <= e; ++e2) {
        if (e2) {
          ny *= p;
          if (ny > b) break;
        }
        dfs(dfs, k + 1, nx, ny);
      }
    }
  };
  for (int i = 0; i <= n - l; ++i) {
    num = l + i;
    cur = fac[i];
    dfs(dfs, 0, 1, 1);
  }
  return h;
}

```

杜教筛

前文讨论了如何计算数论函数 Dirichlet 卷积的前缀和. 本节考虑它的逆过程: 设 $f*g=h$, 且 f,h 已知, 计算 g 的前缀和

$$G(n)=\sum_{x=1}^{n}g(x).$$

换句话说, 本节考虑两个数论函数 Dirichlet 卷积意义的商的前缀和计算. 本节总是假设 $f(1)\neq 0$, 以保证 f 可逆.

为此, 在 Dirichlet 双曲线法表达式中令 $x_0>n$, 就得到

$$H(n)=\sum_{x=1}^{n}f(x)G\left(\left\lfloor\frac{n}{x}\right\rfloor\right).$$

直接解出 $G(n)$, 就得到

$$G(n)=\frac{1}{f(1)}\left(H(n)-\sum_{x=2}^{n}f(x)G\left(\left\lfloor\frac{n}{x}\right\rfloor\right)\right).$$

这就是杜教筛的表达式．实际上对于 $x_0\geq 1$，总有

$$G(n)=\frac{1}{f(1)}\left(H(n)-\sum_{x=2}^{\lfloor x_0\rfloor}f(x)G\left(\left\lfloor\frac{n}{x}\right\rfloor\right)-\sum_{y=1}^{\lfloor y_0\rfloor}F\left(\left\lfloor\frac{n}{y}\right\rfloor\right)g(y)+F(\lfloor x_0\rfloor)G(\lfloor y_0\rfloor)\right).$$

无论是哪种形式，它都是一个关于 $G(n)$ 的递推关系式．为计算 $G(n)$ 的取值，需要计算 G 在 $D(n)\setminus\{n\}$ 处的取值．因为 $D(n)$ 具有 递归结构→9.11.21@p1314，即对于 $m\in D(n)$，总是有 $D(m)\subseteq D(n)$，所以在整个递归计算过程中，只需要计算 G 在 $D(n)$ 中元素处的取值各一次．换句话说，计算 $G(n)$ 时，实际上得到了 g 的块筛 $\mathcal{S}_g(n)$．

具体实现时，可以采用递归的方法，并采用记忆化避免重复计算．当然，也可以采用迭代的方法，从小到大依次计算 $D(n)$ 中每个点处 G 的取值．此时，表达式中的求和式既可以采用数论分块计算，也可以采用 Dirichlet 双曲线法计算．

这些实现的复杂度是相同的．由于杜教筛总是得到块筛，所以杜教筛的复杂度其实相当于计算块筛的复杂度．如果已知信息只有 F,H 的块筛，那么杜教筛的复杂度就是 $O(n^{3/4})$；如果对于某个 $z\geq\sqrt{n}$，可以在 $T_0(z)$ 时间内预处理出 g 在 $1\leq x\leq z$ 处的点值，那么杜教筛的复杂度就是

$$O\left(T_0(z)+\frac{n}{\sqrt{z}}\right).$$

当 g 是积性函数时，可以应用线性筛，即 $T_0(z)=\Theta(z)$，所以最优需要预处理到 $z=n^{2/3}$ 处，总时间复杂度为 $O(n^{2/3})$；对于更一般的情形，总时间复杂度则为 $O(n^{2/3}(\log n)^{1/3})$．这些都和块筛部分的分析完全一致．

⚠ 递归实现时，不使用记忆化将导致复杂度错误

杜教筛的表达式中，计算 $G(n)$ 的值，需要依赖于 $D(n)\setminus\{n\}$ 中 G 的取值．此处保证复杂度的关键在于注意到集合 $D(n)$ 的递归结构：当 $m\in D(n)$ 时，$D(m)\subseteq D(n)$．利用这一性质，可以通过记忆化加速计算．利用记忆化后，如前文所示，算法的复杂度是 $O(n^{3/4})$ 的．此处要说明的是，如果不使用记忆化，算法复杂度是错误的．

设不使用记忆化时，应用递归方法计算 $G(n)$ 的复杂度是 $T(n)$．那么，有

$$\begin{aligned}T(n)&=\Theta(\sqrt{n})+\sum_{d\in D(n),\ d\neq n}T(d)\\&=\Theta(\sqrt{n})+\sum_{x=1}^{\lfloor n/\lfloor\sqrt{n}\rfloor\rfloor-1}T(x)+\sum_{x=2}^{\lfloor\sqrt{n}\rfloor}T\left(\left\lfloor\frac{n}{x}\right\rfloor\right).\end{aligned}$$

利用类似 主定理→5.2.1@p544 的证明思路，可以说明，最后一项主导了该式的增长，且 $T(n)\in\Theta(n^{\alpha})$，其中，$\alpha\approx 1.73$ 是 $\zeta(\alpha)=2$ 的根．

应用杜教筛计算数论函数 g 前缀和时，关键在于找到合适的 f,h 使得 $h=f*g$ 且 f,h 的块筛都容易计算．在有些时候，这样的 f,h 是显然的；在另一些时候，需要利用 Dirichlet 卷积的性质推导，也可以通过计算相应的 Dirichlet 生成函数或 Bell 级数（当满足积性时）来找到相应的分解．后文的例题展示了这些情形．

例题

本节介绍几道计算数论函数前缀和的题目．

AtCoder Regular Contest 116 C - Multiple Sequences[3]

给定正整数 N 和 M. 需要计算有多少长度为 N 的序列 A 满足 $1 \le A_i \le M$ 且 A_i 整除 A_{i+1}. 答案对 998244353 取模. 数据范围: $1 \le N, M \le 2 \times 10^5$.

解答

设长度为 n 且 $A_n = m$ 的序列数目为 $f_n(m)$. 那么, 最终的答案就是 $\sum_{m=1}^{M} f_N(m)$. 要计算 $f_n(m)$, 考虑应用动态规划的方法. 动态转移方程为:

$$f_n(m) = \sum_{k|m} f_{n-1}(k).$$

利用 Dirichlet 卷积的记号, 它可以记作 $f_n = f_{n-1} * 1$, 其中, 1 是常值数论函数. 注意到 $f_1 = 1$, 归纳可知, $f_n = 1^{*n}$. 也就是说, f_n 是 n 个常值函数的卷积. 最后的答案就是 f_N 的前缀和. 由于过程中只涉及积性函数, 利用前文介绍的 Dirichlet 卷积前缀和的计算方法, 单次卷积前缀和的计算只需要 $O(M^{2/3})$ 的时间; 再利用 快速幂→9.7@p1131 的方法, 只需要计算 $O(\log N)$ 次卷积前缀和即可得到 f_N 的卷积前缀和的值. 整体时间复杂度为 $O(M^{2/3} \log N)$.

参考实现

```cpp
#include <cassert>
#include <cmath>
#include <iostream>
#include <vector>

constexpr int M = 998244353;

// Compute the Dirichlet convolution h = f * g.
// Assume that h is multiplicative.
auto dirichlet_convolute(const std::vector<int>& f, const std::vector<int>&
g) {
  int n = f.size() - 1;
  std::vector<int> h(n + 1), primes, rem(n + 1), lpf(n + 1);
  std::vector<bool> vis(n + 1);
  h[1] = 1;
  for (int x = 2; x <= n; ++x) {
    if (!vis[x]) {
      primes.push_back(x);
      rem[x] = 1;
      lpf[x] = x;
    }
    for (int p : primes) {
      if (x * p > n) break;
      vis[x * p] = true;
      rem[x * p] = x % p ? x : rem[x];
      lpf[x * p] = p;
      if (x % p == 0) break;
    }
    if (rem[x] == 1) {  // prime powers.
      for (int k = x; k; k /= lpf[x]) {
        (h[x] += (long long)f[k] * g[x / k] % M) %= M;
```

```
      }
    } else {  // other cases.
      h[x] = (long long)h[rem[x]] * h[x / rem[x]] % M;
    }
  }
  return h;
}

// Block sieve convolution for multiplicative functions.
struct BlockSieve {
  int n, z;
  std::vector<int> f, F, F2;

  BlockSieve(int _n, int _z)
      : n(_n), z(_z), f(_z + 1), F(_z + 1), F2(_n / _z + 1) {}

  int sum(int x) const { return x <= z ? F[x] : F2[n / x]; }

  BlockSieve operator*(const BlockSieve& oth) const {
    assert(n == oth.n && z == oth.z);
    BlockSieve res(n, z);
    res.f = dirichlet_convolute(f, oth.f);
    for (int i = 1; i <= z; ++i) {
      res.F[i] = (res.F[i - 1] + res.f[i]) % M;
    }
    for (int i = 1; i <= n / z; ++i) {
      int k = n / i;
      int sqr = std::sqrt(k + 0.25l);
      for (int x = 1; x <= sqr; ++x) {
        (res.F2[i] += (long long)f[x] * oth.sum(k / x) % M) %= M;
        (res.F2[i] += (long long)sum(k / x) * oth.f[x] % M) %= M;
      }
      (res.F2[i] += (M - (long long)sum(sqr) * oth.sum(sqr) % M)) %= M;
    }
    return res;
  }
};

int main() {
  int n, k;
  std::cin >> k >> n;
  int z = std::pow(n, 2.0l / 3);
  // Initialization of inputs.
  BlockSieve po(n, z);
  for (int i = 1; i <= z; ++i) {
    po.f[i] = 1;
    po.F[i] = i;
  }
  for (int i = 1; i <= n / z; ++i) {
    po.F2[i] = n / i;
  }
  BlockSieve res(n, z);
```

```
  res.f[1] = 1;
  for (int i = 1; i <= z; ++i) {
    res.F[i] = 1;
  }
  for (int i = 1; i <= n / z; ++i) {
    res.F2[i] = 1;
  }
  // Binary exponentiation.
  for (; k; k >>= 1) {
    if (k & 1) res = res * po;
    po = po * po;
  }
  std::cout << res.sum(n) << std::endl;
  return 0;
}

```

P4213【模板】杜教筛（Sum）[4]

设 μ 和 φ 分别是莫比乌斯函数和欧拉函数. 求 $S_1(n)=\sum_{i=1}^{n}\mu(i)$ 和 $S_2(n)=\sum_{i=1}^{n}\varphi(i)$ 的值. 数据范围：$1\le n<2^{31}$.

解答

注意到 Dirichlet 卷积关系式：

$$\varepsilon=\mu*1,\ \ \mathrm{id}=\varphi*1.$$

其中，$\varepsilon(n)=[n=1]$ 是 Dirichlet 卷积的单位元函数，$\mathrm{id}(n)=n$ 是恒等函数，$1(n)=1$ 是常值函数. 因为这三个函数的前缀和都可以在 $O(1)$ 时间内计算且都是积性函数，所以，利用前文介绍的杜教筛方法就可以在 $O(n^{2/3})$ 时间内计算.

对于欧拉函数的前缀和的计算，另一种方法是利用 莫比乌斯反演→9.11.23@p1331. 注意到

$$\begin{aligned}S_2(n)&=\sum_{i=1}^{n}\varphi(i)=\sum_{i=1}^{n}\sum_{j=1}^{i}[i\perp j]\\&=\sum_{i=1}^{n}\sum_{j=1}^{i}\sum_{d}\mu(d)[d\mid i][d\mid j]\\&=\sum_{d}\mu(d)\frac{1}{2}\left\lfloor\frac{n}{d}\right\rfloor\left(\left\lfloor\frac{n}{d}\right\rfloor+1\right).\end{aligned}$$

在数论分块的过程中，需要 $\mu(d)$ 的前缀和，而这可以通过杜教筛预处理出来. 时间复杂度仍然是 $O(n^{2/3})$.

参考实现

```
#include <cstring>
#include <iostream>
#include <map>
using namespace std;
constexpr int MAXN = 2000010;
```

```
long long T, n, pri[MAXN], cur, mu[MAXN], sum_mu[MAXN];
bool vis[MAXN];
map<long long, long long> mp_mu;

long long S_mu(long long x) {  // 求mu的前缀和
  if (x < MAXN) return sum_mu[x];
  if (mp_mu[x]) return mp_mu[x];  // 如果map中已有该大小的mu值，则可直接返回
  long long ret = (long long)1;
  for (long long i = 2, j; i <= x; i = j + 1) {
    j = x / (x / i);
    ret -= S_mu(x / i) * (j - i + 1);
  }
  return mp_mu[x] = ret;  // 路径压缩，方便下次计算
}

long long S_phi(long long x) {  // 求phi的前缀和
  long long ret = (long long)0;
  long long j;
  for (long long i = 1; i <= x; i = j + 1) {
    j = x / (x / i);
    ret += (S_mu(j) - S_mu(i - 1)) * (x / i) * (x / i);
  }
  return (ret - 1) / 2 + 1;
}

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  cin >> T;
  mu[1] = 1;
  for (int i = 2; i < MAXN; i++) {  // 线性筛预处理mu数组
    if (!vis[i]) {
      pri[++cur] = i;
      mu[i] = -1;
    }
    for (int j = 1; j <= cur && i * pri[j] < MAXN; j++) {
      vis[i * pri[j]] = true;
      if (i % pri[j])
        mu[i * pri[j]] = -mu[i];
      else {
        mu[i * pri[j]] = 0;
        break;
      }
    }
  }
  for (int i = 1; i < MAXN; i++)
    sum_mu[i] = sum_mu[i - 1] + mu[i];  // 求mu数组前缀和
  while (T--) {
    cin >> n;
    cout << S_phi(n) << ' ' << S_mu(n) << '\n';
  }
  return 0;
}
```

```
58
```

「LuoguP3768」简单的数学题[5]

给定 p, n，计算

$$\sum_{i=1}^{n}\sum_{j=1}^{n} ij \cdot \gcd(i,j) \pmod p.$$

数据范围：$n \le 10^{10}$，$5\times 10^8 \le p \le 1.1\times 10^9$ 且 p 是质数.

解答

利用 欧拉函数→9.11.5@p1196 的性质，做如下反演：

$$\begin{aligned} T(n) &= \sum_{i=1}^{n}\sum_{j=1}^{n} ij \cdot \gcd(i,j) \\ &= \sum_{i=1}^{n}\sum_{j=1}^{n} ij \sum_{d} \varphi(d)[d \mid i][d \mid j] \\ &= \sum_{d} \varphi(d) \left(\sum_{i=1}^{\lfloor n/d \rfloor} id\right)\left(\sum_{j=1}^{\lfloor n/d \rfloor} jd\right) \\ &= \sum_{d} d^2\varphi(d) F\left(\left\lfloor \frac{n}{d} \right\rfloor\right)^2. \end{aligned}$$

其中，$F(n) = \dfrac{1}{2}n(n+1)$. 该式可以通过数论分块计算，但需要预处理出 $d^2\varphi(d)$ 的前缀和.

为此，可以利用杜教筛. 记 $f(n) = (\mathrm{id}^2 \varphi)(n)$ 和 $S(n) = \sum_{i=1}^{n} f(i)$. 应用杜教筛的关键是构造出函数 g，使得 $f * g$ 和 g 都可以快速求和. 前文已经讨论了用杜教筛预处理 φ 前缀和的方法，这只需要利用关系：$\mathrm{id} = \varphi * 1$. 相较于 φ，这里的 f 多了一个 id^2. 由于 id 是完全积性函数，利用 Dirichlet 卷积的 性质→9.11.22@p1322，只需要类似地将每一项都乘以 id^2，就得到

$$\mathrm{id}^3 = f * \mathrm{id}^2 .$$

因为 $\mathrm{id}^2(n) = n^2$ 和 $\mathrm{id}^3(n) = n^3$ 的前缀和都可以在 $O(1)$ 时间内计算，f 的前缀和就可以在 $O(n^{2/3})$ 时间内预处理得到. 再加上数论分块，整体时间复杂度仍然是 $O(n^{2/3})$.

另一种将类似积性函数表示为两函数之（在 Dirichlet 卷积意义下的）商的方法是利用 Dirichlet 生成函数→9.11.22@p1322.

参考实现

```cpp
// 不要为了省什么内存把数组开小,会卡80
#include <cmath>
#include <iostream>
#include <map>
using namespace std;
constexpr int N = 5e6, NP = 5e6, SZ = N;
long long n, P, inv2, inv6, s[N];
int phi[N], p[NP], cnt, pn;
bool bp[N];
map<long long, long long> s_map;
```

```cpp

long long ksm(long long a, long long m) {  // 求逆元用
  long long res = 1;
  while (m) {
    if (m & 1) res = res * a % P;
    a = a * a % P, m >>= 1;
  }
  return res;
}

void prime_work(int k) {  // 线性筛phi, s
  bp[0] = bp[1] = true, phi[1] = 1;
  for (int i = 2; i <= k; i++) {
    if (!bp[i]) p[++cnt] = i, phi[i] = i - 1;
    for (int j = 1; j <= cnt && i * p[j] <= k; j++) {
      bp[i * p[j]] = true;
      if (i % p[j] == 0) {
        phi[i * p[j]] = phi[i] * p[j];
        break;
      } else
        phi[i * p[j]] = phi[i] * phi[p[j]];
    }
  }
  for (int i = 1; i <= k; i++)
    s[i] = (1ll * i * i % P * phi[i] % P + s[i - 1]) % P;
}

long long s3(long long k) {  // 立方和
  return k %= P, (k * (k + 1) / 2) % P * ((k * (k + 1) / 2) % P) % P;
}

long long s2(long long k) {  // 平方和
  return k %= P, k * (k + 1) % P * (k * 2 + 1) % P * inv6 % P;
}

long long calc(long long k) {  // 计算S(k)
  if (k <= pn) return s[k];
  if (s_map[k]) return s_map[k];  // 对于超过pn的用map离散存储
  long long res = s3(k), pre = 1, cur;
  for (long long i = 2, j; i <= k; i = j + 1)
    j = k / (k / i), cur = s2(j),
    res = (res - calc(k / i) * (cur - pre) % P) % P, pre = cur;
  return s_map[k] = (res + P) % P;
}

long long solve() {
  long long res = 0, pre = 0, cur;
  for (long long i = 1, j; i <= n; i = j + 1) {
    j = n / (n / i);
    cur = calc(j);
    res = (res + (s3(n / i) * (cur - pre)) % P) % P;
    pre = cur;
```

```
  }
  return (res + P) % P;
}

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  cin >> P >> n;
  inv2 = ksm(2, P - 2), inv6 = ksm(6, P - 2);
  pn = (long long)pow(n, 0.666667);  // n^(2/3)
  prime_work(pn);
  cout << solve();
  return 0;
}

```

习题

- AtCoder Xmas Contest 2019 D - Sum of (-1)^f(n)[6]

参考资料与注释

- 任之洲. 2016.《积性函数求和的几种方法》. 2016 年信息学奥林匹克中国国家队候选队员论文.
- 周康阳. 2024.《关于积性函数求和问题的一些进展》. 2024 年信息学奥林匹克中国国家队候选队员论文.
- Dirichlet hyperbola method - Wikipedia[7]
- 杜教筛的时空复杂度分析 - riteme.site[8]
- OI 中常用数论函数求和法的简化陈述 by negiizhao - UOJ[9]
- Dirichlet 積と、数論関数の累積和 by maspy[10]
- Dirichlet convolution. Part 1: Fast prefix sum computations by adamant - Codeforces[11]

[1] Abel 变换

[2] Piltz divisor problem - Divisor summatory function - Wikipedia

[3] AtCoder Regular Contest 116 C - Multiple Sequences

[4] P4213【模板】杜教筛(Sum)

[5]「LuoguP3768」简单的数学题

[6] AtCoder Xmas Contest 2019 D - Sum of (-1)^f(n)

[7] Dirichlet hyperbola method - Wikipedia

[8] 杜教筛的时空复杂度分析 - riteme.site

[9] OI 中常用数论函数求和法的简化陈述 by negiizhao - UOJ

[10] Dirichlet 積と、数論関数の累積和 by maspy

[11] Dirichlet convolution. Part 1: Fast prefix sum computations by adamant - Codeforces

9.11.25 素数计数

Authors: Peanut-Tang, Early0v0, Vxlimo, GHLinZhengyu, 1196131597, c-forrest

本文介绍素数计数问题的亚线性算法. 素数计数问题即计算不超过 n 的素数个数 $\pi(n)$. 对于 $n \sim 10^8$, 可以通过 筛法→9.11.6@p1199 在 $\tilde{O}(n)$ 时间内计算. 然而, 当 $n \sim 10^{10}$ 乃至更大时, 筛法等基于枚举思想的算法将难以适用. 针对这一情形, 本文介绍了一系列较易实现的算法, 其时间复杂度大多介于 $\tilde{O}(n^{2/3})$ 和 $\tilde{O}(n^{3/4})$ 之间.

素数计数问题是素数幂前缀和问题的一个特例. 本文介绍的 Lucy 算法可以直接推广到素数幂前缀和, 而这一推广构成了 Min_25 筛→9.11.27@p1403 等积性函数求和算法的一部分.

> **Tip**
>
> 本文提供了许多素数计数算法．为保证可读性，都没有进行太多常数优化．这些算法在时间空间表现和实现难度上各有差异，读者可以根据自身需求选取合适算法．对于 $10^{11} \sim 10^{13}$ 范围数据，Lehmer 截断法则→9.11@p1160 一节算法效率往往已经足够且实现较简单，优化后的 Lucy 算法→9.11@p1160 实现略复杂但是有着极优秀的时空表现．对于更小数据范围，原始 Lucy 算法→9.11@p1160 实现最简单，且在积性函数求和中有着更广阔的用途．

基本概念与记号

本文将使用如下记号：

- $\mathbf{P}$ 是（正）素数的集合，且 p_a 是第 a 小的素数（下标自 1 开始）．为叙述方便，另设 $p_0 = 1$．本文中，字母 p 和 q 总是表示素数．
- $\mathrm{lpf}(n)$ 表示 n 的最小素因子．另设 $\mathrm{lpf}(1) = +\infty$．
- $D(x) = \{\lfloor x/i \rfloor : i = 1, 2, ..., \lfloor x \rfloor\}$．它的性质详见 数论分块的性质→9.11.21@p1314 一节．
- $\pi(x)$ 是不超过 x 的素数个数．
- $\varphi(x, a) = \#\{n \in \mathbf{N}_+ : n \le x,\ (p \mid n \Longrightarrow p > p_a)\}$ 是所有不超过 x 的正整数中，所有素因数都大于 p_a 的数的个数．
- $P_k(x, a) = \#\{n \in \mathbf{N}_+ : n \le x,\ n = q_1 q_2 \cdots q_k,\ \forall i (q_i > p_a)\}$ 是所有不超过 x 的正整数中，素因数个数（计重数）恰好等于 k，且所有素因数都大于 p_a 的数的个数．
- $S(x, a)$ 是 Eratosthenes 筛法中，利用前 a 个素数筛完后，剩下的大于 1 且不超过 x 的整数个数．

这些函数有着如下简单的性质：

- $\varphi(x, 0) = \lfloor x \rfloor$．
- $S(x, a) = \varphi(x, a) + \min\{\pi(x), a\} - 1$．
- $\varphi(x, a) = \sum_{k=0}^{\infty} P_k(x, a)$．
- $P_0(x, a) = 1$．
- $P_1(x, a) = \pi(x) - \min\{\pi(x), a\}$．
- 对于 $x < p_{a+1}^k$，有 $P_k(x, a) = 0$．

这些性质都很容易从它们的定义得出．

Meissel–Lehmer 算法

将 P_0 和 P_1 的表达式代入 φ 的展开式，简单整理就得到

$$\pi(x) = \varphi(x, a) + a - 1 - P_2(x, a) - P_3(x, a) - \cdots.$$

这意味着 $\pi(x)$ 可以通过计算 $\varphi(x, a)$ 的取值而得到，且误差由一系列 P_k 项给出．随着 a 取值的上升，误差项的数目也逐渐减少．特别地，对于 $a = \pi(\sqrt{x})$，有 $P_2(x, a) = P_3(x, a) = \cdots = 0$，亦即

$$\pi(x) = \varphi(x, \pi(\sqrt{x})) + \pi(\sqrt{x}) - 1.$$

对于 $\pi(x^{1/3}) \le a < \pi(\sqrt{x})$，项 $P_2(x, a)$ 不为零，但后续项仍然是零，即

$$\pi(x) = \varphi(x, a) + a - 1 - P_2(x, a).$$

类似地，对于 $\pi(x^{1/4}) \le a < \pi(x^{1/3})$，项 $P_2(x, a)$ 和 $P_3(x, a)$ 均不为零，如此类推．总之，利用这些关系式，可以将 $\pi(x)$ 的计算转化为 $\varphi(x, a)$ 的计算和 $P_k(x, a)$ 项的计算．所有依赖于这些转化关系的算法，都可以称为 Meissel–Lehmer 算法．

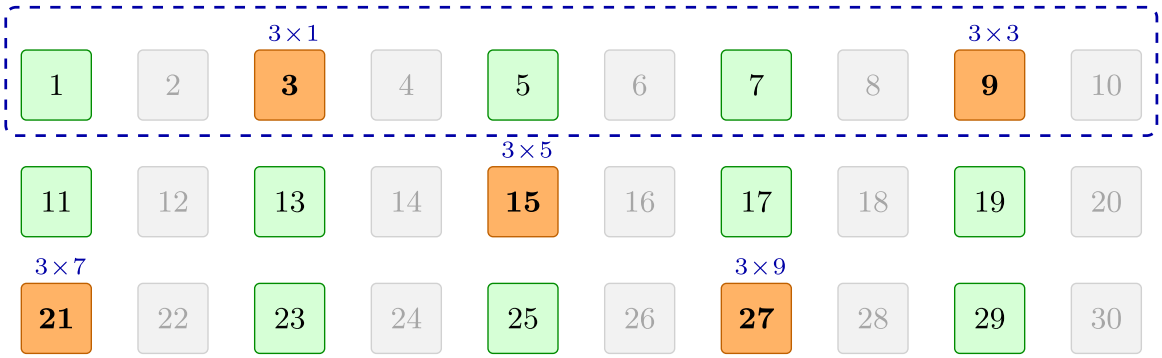
对于 $\varphi(x, a)$，有如下递推关系：

$$\varphi(x, a) = \varphi(x, a - 1) - \varphi\left(\frac{x}{p_a}, a - 1\right).$$

自然地，边界条件由 $\varphi(x, 0) = \lfloor x \rfloor$ 给出．

示例

以 $x=30, a=2$ 为例（$p_a=3$），递推的一步如下图所示：



图中绿色格子表示所有素因数都大于 $p_a=3$ 的整数（1 没有素因数，也计入其中），绿色与橙色合起来则表示所有素因数都大于 2 的整数．前者的个数是 $\varphi(x,a)=\varphi(30,2)=10$，后者的个数是 $\varphi(x,a-1)=\varphi(30,1)=15$.

两者之差恰是橙色部分：能被 3 整除，且不含有比 3 小的素因子．于是橙色的数都具有 $n=3k$ 的形式．由于 k 的每个素因数也是 n 的素因数，k 同样不含比 3 小的素因子，即 k 自身也落在绿色或橙色之中；再由 $3k\le 30$ 得 $k\le\lfloor x/p_a\rfloor=10$．反过来也一样：$n=3k$ 的素因数无非是 3 与 k 的素因数，所以只要 $k\le 10$ 且不含比 3 小的素因子，n 就落在橙色部分．二者因此一一对应．

表格特意排成 $30/3=10$ 列，所以第一行恰好就是 $1,\dots,10$．这样的 k 正是虚线框内的绿色和橙色格子，共 $\varphi(x/p_a,a-1)=\varphi(30/3,1)=5$ 个；每个橙色格子右上角的 $3\times k$ 标出了它所对应的 k．因此

$$\varphi(x,a)=\varphi(x,a-1)-\varphi\left(\left\lfloor\frac{x}{p_a}\right\rfloor,a-1\right).$$

此即，$\varphi(30,2)=\varphi(30,1)-\varphi(30/3,1)=15-5=10$.

利用该关系进行递归计算，所得到的递归树是一棵二叉树．完全展开，就得到

$$\varphi(x,a)=\sum_{n\mid p_1p_2\cdots p_a}\mu(n)\left\lfloor\frac{x}{n}\right\rfloor.$$

由此，这棵二叉树的叶子结点数目就等于所有不超过 x 的无平方因子的 p_a-光滑数（即不含有超过 p_a 素因子的整数）的数目，这一数目是 $\Theta(x)$ 的[51]．这意味着，直接递归计算，复杂度仍然是 $\Omega(x)$ 的．为了快速计算，需要对这棵二叉树适当地进行剪枝．不同 Meissel–Lehmer 算法的主要区别，就在于剪枝方法．文献中将它们称为 **截断法则**（truncation rule）．随后，本节将重点介绍几种简单的截断法则．

最后是处理 $P_k(x,a)$．以 $P_2(x,a)$ 为例，有

$$\begin{aligned}P_2(x,a)&=\sum_{i,j:\ p_a<p_i\le p_j\le x/p_i}1=\sum_{i=a+1}^{\pi(\sqrt{x})}\sum_{j=i}^{\pi(x/p_i)}1\\&=\sum_{i=a+1}^{\pi(\sqrt{x})}\left(\pi\left(\frac{x}{p_i}\right)-i+1\right)\\&=\sum_{i=a+1}^{\pi(\sqrt{x})}\pi\left(\frac{x}{p_i}\right)-\frac{1}{2}\big(\pi(\sqrt{x})+a-1\big)\big(\pi(\sqrt{x})-a\big).\end{aligned}$$

[51]参见 Smooth number - Wikipedia[1].

这意味着 $P_2(x,a)$ 的计算可以转化为若干个 $\pi(x/p)$ 的计算．对于 $k>2$，仍然存在类似的求和式，但是求和的层数会变多，所以逐渐不再实用．常见算法大多会取 $a\geq\pi(x^{1/3})$，以避免计算更多的 P_k 项．实际计算时，通常会考虑利用筛法预处理这一部分的 $\pi(x/p)$ 的值．

数学家很早就思考了不依赖枚举直接计算 $\pi(x)$ 的问题. Legendre 给出 $a=\pi(\sqrt{x})$ 时的上述表达式, 但将 $\varphi(x,a)$ 完全展开得到的项数过多，无法实际用于计算．在 1870 年，Meissel 提出，可以在 $a=\pi(x^{1/3})$ 处计算，减少 $\varphi(x,a)$ 展开的项数，且误差仍然容易计算．在 1959 年，Lehmer 进一步改进和简化了该过程．在 1985 年，Lagarias、Miller 和 Odlyzko 提出的截断规则，首次将该思路改进到了亚线性复杂度，得到了 $\tilde{O}(x^{2/3})$ 的时间复杂度和 $\tilde{O}(x^{1/3})$ 的空间复杂度．之后，Deléglise and Rivat (1996)，Gourdon (2001) 和 Staple (2015) 等沿着该方向做出更多的优化，进一步减少了复杂度中 $\log x$ 的次数．需要说明的是，这些算法为了保持良好的空间复杂度，以处理类似 $x\sim 10^{26}$ 规模的问题，通常较为繁复．本节将大幅简化其中细节，只介绍一些简单的优化思路．这样会牺牲一定的时空复杂度，但代码较容易实现．对于原文处理感兴趣的读者，可以参考文末的文献自行学习．

另外，除了枚举方法和本文介绍的组合方法外，计算 $\pi(x)$ 还可以利用解析方法，做到 $\tilde{O}(\sqrt{x})$ 的时间复杂度．但它们无法应用于算法竞赛，本文不做介绍．

Lehmer 截断法则

Lehmer (1959) 提出了一种截断法则．对于如下两种情形，不再展开 $\varphi(u,b)$：

1. 当 $u<p_b$ 时；
2. 当 $b=c$ 时，其中，c 是提前选取的小正整数．

对于第一种情形，依前文讨论，必然有 $\varphi(u,b)=1$，无需计算．对于第二种情形，则需要额外计算出 $\varphi(u,c)$ 的值．根据定义，有

$$\varphi(u,c)=\sum_{n=1}^{u}\prod_{b=1}^{c}[p_b\nmid n].$$

由于对 p_b 的整除关系具有 p_b 的周期，求和项 $\prod_{b=1}^{c}[p_b\nmid n]$ 就具有周期 $p_c\#=\prod_{b=1}^{c}p_b$．利用周期性，就有

$$\varphi(u,c)=\left\lfloor\frac{u}{p_c\#}\right\rfloor\varphi(p_c\#,c)+\varphi(u\bmod p_c\#,c).$$

因此，只要对 $n=0,1,2,\ldots,p_c\#$ 利用递推关系预处理出所有 $\varphi(n,c)$ 的取值，就能迅速查询第二种情形中 $\varphi(u,c)$ 的取值．

需要说明的是，尽管这两条截断法则确实提高了计算效率，但是算法的渐近复杂度没有显著改善．Lagarias, Miller, and Odlyzko (1985) 证明，Lehmer 算法中，递归树叶子结点数目是 $\Omega\left(\dfrac{x}{\log^4 x}\right)$ 的，因此时间复杂度仍然是 $\tilde{\Theta}(x)$ 的．

当然，Lehmer 提出的第一条截断法则可以适当改良：

$$\varphi(u,b)=\begin{cases}1, & u\leq p_b,\\ \pi(u)-b+1, & p_b<u\leq p_b^2,\\ \pi(u)-\dfrac{1}{2}(\pi(\sqrt{u})+b-2)(\pi(\sqrt{u})-b+1)+\displaystyle\sum_{i=b+1}^{\pi(\sqrt{u})}\pi\left(\frac{u}{p_i}\right), & p_b^2<u\leq p_b^3.\end{cases}$$

后面两种情形利用了前文导出的关系式．这一改良进一步削减了递归树的规模，但是引入了更多的 $\pi(u/p)$ 项需要计算．由于 u/p 最高可以达到 x/p_{c+1}，为了减少无效剪枝，可以设定一个可以触发该法则的 u 的上限 V，先预处理出 $[1,V]$ 的 $\pi(u)$ 值；而当 $u>V$ 时，仍然采取正常的递归．

下面给出改良后的 Lehmer 截断法则的实现：

参考实现

```cpp
// Modified from sgtlaugh's codes:
//   https://github.com/sgtlaugh/algovault/blob/master/code_library/fast_prime_counting.cpp
// Submission:
// * https://judge.yosupo.jp/submission/396797 (1e11)
// * https://www.luogu.com.cn/record/295291689 (1e13)
constexpr int V = 20000010;  // tuned for 1e12; at least sqrt{n} + 1.

constexpr int C = 7;
constexpr int N = 50;                              // N >= C must hold.
constexpr int Q = 2 * 3 * 5 * 7 * 11 * 13 * 17;  // = p_C#

std::vector<int> primes;
std::array<int, V> pi{};
std::array<std::array<int, Q + 1>, N> dp{};

// Find the primes and pi below V.
void sieve() {
  std::vector<bool> vis(V);
  for (int x = 2; x < V; ++x) {
    if (!vis[x]) primes.push_back(x);
    for (int p : primes) {
      if (x * p >= V) break;
      vis[x * p] = true;
      if (x % p == 0) break;
    }
  }
  for (int x = 2; x < V; ++x) {
    pi[x] = pi[x - 1] + !vis[x];
  }
}

// Initialization step.
// First sieve, then obtain the value of phi(u, C) for u <= p_C#.
void init() {
  sieve();
  for (int u = 1; u <= Q; ++u) dp[0][u] = u;
  for (int b = 1; b < N; ++b) {
    for (int u = 1; u <= Q; ++u) {
      dp[b][u] = dp[b - 1][u] - dp[b - 1][u / primes[b - 1]];
    }
  }
}

// Recursively find phi with Lehmer's truncation rule.
long long phi(long long u, int b) {
  if (u <= Q && b < N) return dp[b][u];
  if (b == C) return dp[b][u % Q] + (u / Q) * dp[b][Q];
  long long p = primes[b - 1];
  if (u < V && p * p >= u) return pi[u] - b + 1;
  if (p * p * p < u || u >= V) return phi(u, b - 1) - phi(u / p, b - 1);
  int lim = pi[(int)std::sqrt(u + 0.25l)];
```

```
  long long res = pi[u] - (lim + b - 2) * (lim - b + 1) / 2;
  for (int i = b; i < lim; ++i) {
    res += pi[u / primes[i]];
  }
  return res;
}

// Meissel-Lehmer with a = pi(n^{1/3}) and Lehmer's truncation rule.
long long lehmer_pi(long long n) {
  if (n < V) return pi[n];
  int sqr = std::sqrt(n + 0.25l);
  int a = std::cbrt(n + 0.25l);
  long long res = phi(n, pi[a]) + pi[a] - 1;
  for (int i = pi[a]; i < pi[sqr]; ++i) {
    res -= lehmer_pi(n / primes[i]) - i;
  }
  return res;
}

```

尽管该实现的理论复杂度难以证明[52], 但是优势在于常数很小且实现简单, 代码实际运行效率很高. 代码中的 c, V, N 等常数的取值可以根据实际需求进行调整.

LMO 截断法则

Lagarias, Miller, and Odlyzko (1985) 提出了另一种截断法则. 选取 y 满足 $x^{1/3} \le y \le x^{2/5}$, 取 $a = \pi(y)$. 那么, 对于如下两种情形, 不再展开 $\varphi(x/n, b)$:

1. 当 $b = c$ 且 $n \le y$ 时, 其中, c 是提前选取的小自然数;
2. 当 $n > y$ 时.

LMO 将第一种情形称为普通叶子结点, 将第二种情形称为特殊叶子结点.

这一截断法则的优势在于, 容易对叶子结点数目进行计数. 注意到, 不同的叶子结点, 必然有着不同的 n. 普通叶子结点总是满足 $n \le y$, 所以数目是 $O(y)$ 的. 特殊叶子结点总是满足 $n > y$ 且 $\dfrac{n}{\operatorname{lpf}(n)} \le y$. 由此, 可以将特殊叶子结点分为 $\operatorname{lpf}(n) < \sqrt{y}$ 和 $\operatorname{lpf}(n) \ge \sqrt{y}$ 两类. 第一类结点中, 必然有 $\operatorname{lpf}(n) < \sqrt{y}$, 所以 $\operatorname{lpf}(n)$ 的数目不超过 $\pi(\sqrt{y})$, 而 $\dfrac{n}{\operatorname{lpf}(n)} \le y$ 至多也只有 y 种选择, 所以 n 可能的数目——亦即这类结点总数——也不超过 $y\pi(\sqrt{y})$. 第二类结点中, 必然有 $n = pq$ 且 $\sqrt{y} \le p < q \le y$, 故而这样的结点数目至多是 $\dfrac{1}{2}\pi(y)^2$. 综合两种情形, 特殊叶子结点总数为 $O\left(\dfrac{y^2}{\log^2 x}\right)$.

对于普通叶子结点, 可以利用和前文所述一致的预处理方法. 对于特殊叶子结点, 要计算 $\varphi(x/n, b)$ 的取值, 可以按照定义将其理解为「不超过 x/n 的正整数中, 最小素因子严格大于 p_b 的数的个数」. (注意前文已设 $\operatorname{lpf}(1) = +\infty$.) 做这样的转化后, 可以将所有特殊叶子结点处的查询离线, 预处理出 $[1, x/y]$ 中整数 lpf 的取值并排序, 再利用树状数组更新并查询. 考虑排序离线查询和树状数组操作, 这样做的时间复杂度为

$$O\left(\frac{y^2}{\log^2 x}\log\frac{y^2}{\log^2 x} + \left(\frac{x}{y} + \frac{y^2}{\log^2 x}\right)\log\pi\left(\frac{x}{y}\right)\right) = O\left(\frac{x}{y}\log x + \frac{y^2}{\log x}\right).$$

除了 $\varphi(x, a)$ 的计算比较特殊外, 其余部分的计算与前一节类似. 只需要预处理出 $[1, x/y]$ 中的 $\pi(u)$ 值, 然后利用前文关系式计算 $P_2(x, a)$ 即可; 对于 $c > 0$ 的情形, 还需要预处理出 $\varphi(u, c)$ 的取值. 注意,

[52] 尽管原作者在代码中标注复杂度大致为 $O(x^{2/3})$, 但经过实际测量, 对于恒定的 V, 复杂度大致为 $\Theta(x)$. 即使根据问题规模适当调整 V 的大小, 复杂度也至少有 $\Omega(x^{0.8})$.

为了满足离线查询的需要，递归搜索叶子结点过程中，还需要记录每个叶子结点前面的符号，即 $\mu(n)$ 的取值，用于统计贡献．由此，就得到完整的算法．

这一算法的时空复杂度均是 $\tilde{O}(x^{2/3})$ 的．如果取 $c=0$，那么其他部分的复杂度可以忽略不计，这一算法总复杂度就在 $y=x^{1/3}\log^{2/3}x$ 处得到最小值 $O(x^{2/3}\log^{1/3}x)$．此时，该算法的空间复杂度是 $O(x^{2/3}\log^{-2/3}x)$ 的．如果 c 取一个小正整数，那么需要平衡预处理的时空成本和后续查询的常数改良，但整体复杂度不会变化．

下面给出 $c=0$ 时该算法的参考实现：

参考实现

```cpp
// Modified from Maksim1744's codes:
//   https://codeforces.com/blog/entry/91632
// Submission:
// * https://judge.yosupo.jp/submission/396868 (1e11)
// * https://www.luogu.com.cn/record/295345663 (1e13, MLE)
// Submission: (lpf with segmented sieve)
// * https://judge.yosupo.jp/submission/396984 (1e11)
// * https://www.luogu.com.cn/record/295487031 (1e13)
// Binary-indexed tree.
struct BIT {
  int n;
  std::vector<int> su;

  BIT(int _n) : n(_n), su(_n + 1) {}

  void add(int x) {
    for (; x <= n; x += (x & (-x))) ++su[x];
  }

  int get(int x) {
    int res = 0;
    for (; x; x &= x - 1) res += su[x];
    return res;
  }
};

// Meissel-Lehmer with LMO's truncation rule and offline queries.
long long lmo_pi(long long n) {
  long long y = std::pow(n, 0.36l);  // tuned for n = 1e12 or 1e13.
  long long s = n / y;
  if (n < 100) s = n;
  // Linear sieve.
  // lpf records the prime rank (0-indexed).
  std::vector<int> primes, lpf(s + 1, -1);
  for (int x = 2; x <= s; ++x) {
    if (lpf[x] == -1) {
      lpf[x] = primes.size();
      primes.push_back(x);
    }
    for (int i = 0; i < (int)primes.size(); ++i) {
      int p = primes[i];
      if (x * p > s) break;
```

```
      lpf[x * p] = i;
      if (x % p == 0) break;
    }
  }
  if (n < 100) return primes.size();
  // pi(y).
  int pi_y = std::upper_bound(primes.begin(), primes.end(), y) -
primes.begin();
  long long res = pi_y;
  // P2(n,pi(y)) with two pointers.
  int ptr = primes.size() - 1;
  for (int i = pi_y; i < (int)primes.size(); ++i) {
    while (ptr >= i && (long long)primes[i] * primes[ptr] > n) --ptr;
    if (ptr < i) break;
    res -= ptr - i + 1;
  }
  // phi(n,pi(y)).
  std::vector<std::tuple<long long, int, signed char>> queries;
  auto phi = [&](auto&& phi, long long u, int b, signed char sign = 1) ->
void {
    if (!u) return;
    if (!b) return (void)(res += u * sign);
    if (u <= s) return (void)queries.emplace_back(u, b, sign);
    phi(phi, u, b - 1, sign);
    phi(phi, u / primes[b - 1], b - 1, -sign);
  };
  phi(phi, n, pi_y);
  std::sort(queries.begin(), queries.end());
  int sz = primes.size(), idx = 2;
  BIT bit(sz);
  for (const auto& query : queries) {
    long long u;
    int b;
    signed char sign;
    std::tie(u, b, sign) = query;
    while (idx <= u) bit.add(sz - lpf[idx++]);
    res += sign * (bit.get(sz - b) + 1);
  }
  queries.clear();
  return res - 1;
}

```

这种离线做法的空间复杂度较高，难以处理 10^{13} 规模的问题．瓶颈在于，预处理时需要存储 $[1, x/y]$ 中的 lpf 信息．实际上该信息仅用于后续离线查询，只需要单次顺序访问，完全可以使用 分块筛法→9.11.6@p1199 在查询时计算．

实际上，原论文提供了一种完全基于分块筛法的实现方式．将 $P_2(x, a)$ 和 $\varphi(x, a)$ 的计算都在分块筛法中完成，从而得到了时间复杂度为 $\tilde{O}(x^{2/3})$ 且空间复杂度为 $\tilde{O}(x^{1/3})$ 的算法．这需要以某种方式直接枚举所有叶子结点，并将离线算法改造为在线算法．接下来，介绍一种简单的实现方式．

实现的核心是分块筛法：预处理完 $[1, y]$ 内的素数后，将 $[1, x/y]$ 分成长度为 y 的若干块，并对每块分别应用筛法．在处理每一块时，都需要记录该块内元素处 $\pi(u)$ 和 $\varphi(u, b)$ 的取值．由于这些数值都是计数，

很容易在分块的过程中维护．然后，需要找到哪些查询落入该块的处理范围．在本节之前描述的算法中，设 $y = \tilde{O}(x^{1/3})$ 且 $c = 0$．此时，素数个数 $\pi(x)$ 有如下表达式：

$$\pi(x) = \frac{1}{2}\big(\pi(\sqrt{x}) + a - 2\big)\big(\pi(\sqrt{x}) - a + 1\big) - \sum_{y < p \le \sqrt{x}} \pi\left(\frac{x}{p}\right) + \sum_{n \le y} \mu(n)\varphi\left(\frac{x}{n}, 0\right) + \sum_{n > y,\ n/\operatorname{lpf}(n) \le y,\ \operatorname{lpf}(n) \le y} \mu(n)\varphi\left(\frac{x}{n}, \pi(\operatorname{lpf}(n)) - 1\right).$$

第一项只需要 $\pi(\sqrt{x})$ 的取值，可以在分块处理到该元素时计算．第二项需要枚举素数 p 使得 x/p 位于当前处理的块内，解出 p 所在的区间后，可以通过分块筛法得到对应素数序列．由于 p 所在的区间不包含大于 $\sqrt{x}$ 的元素，这个内层分块筛法只需要使用 $[1, x^{1/4}]$ 以内的素数．第三项只要枚举 $[1, y]$ 内元素即可．第四项需要枚举 $[1, y]$ 内的素数作为 $\operatorname{lpf}(n)$，进而确定 $\dfrac{n}{\operatorname{lpf}(n)}$ 的取值范围，再枚举其中满足素因子不小于 $\operatorname{lpf}(n)$ 的元素[53]，就可以计算得到这一部分叶子结点的值．在计算当前块部分的贡献时，可以使用树状数组维护．这样就在保证了空间复杂度为 $\tilde{O}(x^{1/3})$ 的前提下，仍然取得了 $\tilde{O}(x^{2/3})$ 的时间复杂度．当然，它的实现相较于离线算法会繁琐一些．

参考实现

```cpp
// An implementation of LMO section 3.
// Submission:
// * https://judge.yosupo.jp/submission/397047 (1e11)
// * https://www.luogu.com.cn/record/295493092 (1e13)
// Binary-indexed tree.
struct BIT {
  int n;
  std::vector<int> su;

  void alloc(int _n) {
    n = _n;
    su.assign(_n + 1, 0);
  }

  BIT(int _n = 0) { alloc(_n); }

  void add(int x) {
    for (; x <= n; x += (x & (-x))) ++su[x];
  }

  int get(int x) {
    int res = 0;
    for (; x; x &= x - 1) res += su[x];
    return res;
  }
};

// Meissel-Lehmer with LMO's truncation rule and online queries.
long long lmo_pi(long long n) {
  long long y = std::pow(n, 0.36l);  // tuned for 1e12.
  long long s = n / y;
  long long sqr = std::sqrt(n + 0.25l);
  // Linear sieve for [1,y].
```

[53]这一步枚举的 n 的总数实际上多于特殊叶子结点数目，但是对总复杂度没有影响．具体证明参见原论文．

```cpp
  if (n < 100) y = n;
  // lpf records the prime rank (0-indexed).
  std::vector<int> primes, lpf(y + 1, -1), mu(y + 1);
  mu[1] = 1;
  for (int x = 2; x <= y; ++x) {
    if (lpf[x] == -1) {
      lpf[x] = primes.size();
      primes.push_back(x);
      mu[x] = -1;
    }
    for (int i = 0; i < (int)primes.size(); ++i) {
      int p = primes[i];
      if (x * p > y) break;
      lpf[x * p] = lpf[p];
      mu[x * p] = x % p ? -mu[x] : 0;
      if (x % p == 0) break;
    }
  }
  if (n < 100) return primes.size();
  // The ordinary leaves.
  long long res = 0;
  for (int i = 1; i <= y; ++i) {
    if (mu[i]) res += mu[i] * (n / i);
  }
  // Segmented sieve.
  // Counts from previous blocks. phi[sz] stores the prime counts.
  int sz = primes.size();
  std::vector<int> phi(sz + 1, 0);
  BIT bit;
  auto phi_at = [&](long long v, int i) -> long long {
    return phi[i] + v - bit.get(v);
  };
  for (int id = 0; id * y + 1 <= s; ++id) {
    int ll = id * y + 1;
    int rr = std::min((id + 1) * y, s);
    int len = rr - ll + 1;
    bit.alloc(len);
    std::vector<bool> vis(len);
    for (int i = 0; i < sz; ++i) {
      int p = primes[i];
      // Special leaves.
      int nl = std::max(n / (rr + 1) / p, y / p) + 1;
      int nr = std::min(n / ll / p, y);
      for (int j = nl; j <= nr; ++j) {
        if (mu[j] && lpf[j] > i) {
          res += -mu[j] * phi_at(n / p / j - ll + 1, i);
        }
      }
      // Accumulate this block's info.
      phi[i] = phi_at(len, i);
      // Sieve this block with p.
      for (int x = (ll - 1) / p * p + p; x <= rr; x += p) {
```

```
        if (!vis[x - ll]) {
          bit.add(x - ll + 1);
          vis[x - ll] = true;
        }
      }
    }
    // All the remaining in this block are primes.
    // Those prime counts in P2(x,a).
    int nl = std::max(n / (rr + 1) + 1, y + 1);
    int nr = std::min(n / ll, sqr);
    if (nr >= nl) {
      std::vector<bool> vis(nr - nl + 1);
      for (int i = 0; i < sz; ++i) {
        int p = primes[i];
        if ((long long)p * p > nr) break;
        for (int x = (nl - 1) / p * p + p; x <= nr; x += p) {
          vis[x - nl] = true;
        }
      }
      for (int x = nl; x <= nr; ++x) {
        if (!vis[x - nl]) {
          res -= phi_at(n / x - ll + 1, sz);
        }
      }
    }
    // Those constant terms.
    if (ll <= sqr && sqr <= rr) {
      auto pi_sqr = phi_at(sqr - ll + 1, sz);
      res += (pi_sqr + sz - 2) * (pi_sqr - sz + 1) / 2;
    }
    // Accumulate the prime count.
    phi[sz] = id ? phi_at(len, sz) : sz;
  }
  return res;
}

```

原论文和后续论文对于特殊叶子结点做了更多分类和讨论，进一步优化了时空复杂度．但是，本节给出的实现足以满足竞赛需求，故不再讨论 Meissel–Lehmer 算法那些更复杂的优化思路．

Lucy 算法

应用 Meissel–Lehmer 算法，要解决的核心问题之一就在于 $\varphi(x,a)$ 的递归树规模过大．这是因为 $\varphi(x,a)$ 递推关系的终止条件只会出现在 $a=0$ 处．考虑用 $S(x,a)$ 替换 $\varphi(x,a)$．它的好处在于，如果 $p_a^2>x$，那么 Eratosthenes 筛法中用素数 p_a 去筛时，不会筛掉任何合数（因为它们必然有更小的素因子），即 $S(x,a)=S(x,a-1)$．这就为递归树的提前终止提供了可能．对于 $p_a^2\le x$ 的情形，有 $a\le\pi(x)$，所以有 $S(x,a)=\varphi(x,a)+a-1$．又由 $a-1\le\pi(x/p_a)$，可以将 $\varphi(x,a)$ 递推关系中的所有项都相应替换为 $S(\cdot,\cdot)$，就可以得到 $S(x,a)$ 的递推关系．综合两种情形，$S(x,a)$ 的递推关系可以写作

$$S(x,a)=S(x,a-1)-[p_a^2\le x]\left(S\left(\frac{x}{p_a},a-1\right)-(a-1)\right).$$

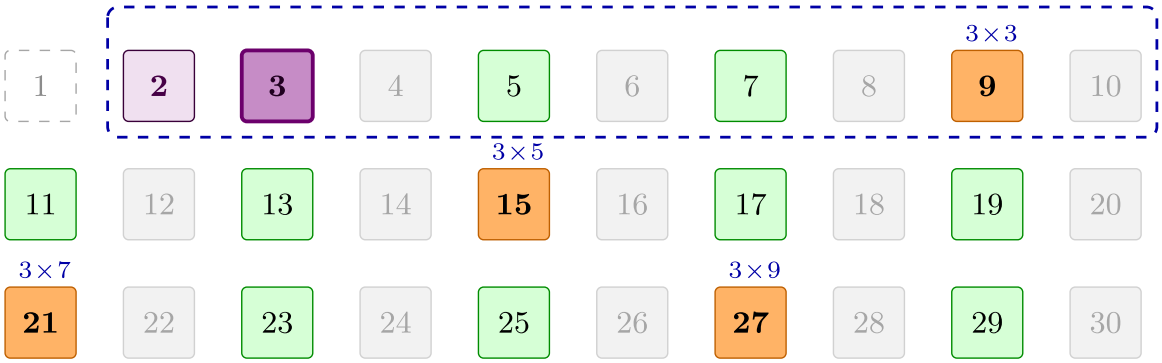
边界条件为 $S(x,0)=\lfloor x\rfloor-1$．最后，由关系式

$$\pi(x)=S(x,\pi(\sqrt{x})),$$

就可以直接得到素数计数函数 $\pi(x)$ 的取值.

示例

同样是 $x=30,a=2$ 的情形，$S(x,a)$ 的递推关系如下图所示:



Eratosthenes 筛法的初始区间为 $[2,x]=[2,30]$，故格子 1 画成白色，不参与计数. 灰色格子是之前已经筛去的合数，橙色格子是这一轮用素数 $p_a=3$ 筛去的合数；剩下的格子中，浅紫色表示之前的素数，深紫色表示当前的素数 3，绿色则表示尚未被筛去的整数（可能是素数，也可能是合数，如 25）. 于是 $S(x,a-1)=S(30,1)=15$ 是橙、紫、绿三色格子之和，$S(x,a)=S(30,2)=11$ 是紫、绿格子之和，两者的差值正是那 4 个橙色格子.

与 $\varphi(x,a)$ 的情形类似，这些数仍具有 $3k$ 的形式，且由 $3k\le 30$ 知 k 落在虚线框内的 $[2,\lfloor x/p_a\rfloor]=[2,10]$ 中（$k=1$ 不在其中，这正是 3 自身得以留下、没有变成橙色的原因）. 区别在于 k 现在只遍历 $3,5,7,9$（已标在各橙色格子的右上角）：k 必须是上一轮筛后的幸存者（否则 $3k$ 早已随 k 一同被筛去），即框内的 $2,3,5,7,9$，共 $S(x/p_a,a-1)=S(10,1)=5$ 个；其中还须去掉比 p_a 小的素数——它们恰是前 $a-1$ 个素数，此处即 $k=2$ 一个——因为 $3\times 2=6$ 早已被 2 筛去. 于是

$$S(x,a)=S(x,a-1)-\left(S\left(\left\lfloor\frac{x}{p_a}\right\rfloor,a-1\right)-(a-1)\right).$$

此即 $S(30,2)=S(30,1)-\big(S(10,1)-(2-1)\big)=15-(5-1)=11$.

这一递推关系很容易通过动态规划进行计算. 因为 $S(x,a)=S(\lfloor x\rfloor,a)$，所以递推关系中的除式都可以看作是整除. 根据 数论分块的性质→9.11.21@p1314 可知，动态规划中第一维的取值必然在集合 $D(x)$ 内，只有 $\Theta(\sqrt{x})$ 种. 而且，由于递推关系较为特殊，可以通过一个长度为 $|D(x)|$ 的数组存储 $S(\cdot,a)$ 的取值；对于每个 a，从大到小遍历 $D(x)$ 中的元素，利用递推关系更新数组，直到元素严格小于 p_a^2. 只需要 $\pi(\sqrt{x})$ 次更新就可以得到最终结果.

算法的空间复杂度明显是 $O(x^{1/2})$ 的. 至于时间复杂度，预处理部分的复杂度是 $O(x^{1/2})$ 的，动态规划部分的复杂度则由

$$I=\sum_{a=1}^{\pi(\sqrt{x})}|\{u\in D(x):u\ge p_a^2\}|$$

给出. 它可以分为两部分进行估计. 对于 $a\in[1,\pi(x^{1/4})]$，由于求和项不会超过 $|D(x)|$，所以这一部分的和不会超过 $\pi(x^{1/4})|D(x)|$，这是 $O(x^{3/4}\log^{-1}x)$ 的. 对于 $a\in(\pi(x^{1/4}),\pi(\sqrt{x})]$，有 $p_a^2>\sqrt{x}$；由集合 $D(x)$ 结构可知，这些元素数目恰为 $\lfloor x/p_a^2\rfloor$. 由此，再结合素数定理 $p_a\sim a\log a$，就得到

$$\sum_{a=\pi(x^{1/4})+1}^{\pi(\sqrt{x})}\frac{x}{p_a^2}\in O\left(\int_{\pi(x^{1/4})}^{+\infty}\frac{x}{a^2\log^2 a}da\right)=O\left(\frac{x^{3/4}}{\log x}\right).$$

将两部分相加可知，算法整体时间复杂度是 $O(x^{3/4}\log^{-1}x)$ 的.

在具体实现时，可以将预处理不超过 $\sqrt{x}$ 的素数的步骤合并到动态规划过程中. 只需要从小到大枚举区间 $[2, \sqrt{x}]$ 内所有整数 u，并检查条件 $S(u, \pi(u-1)) \neq S(u-1, \pi(u-1))$ 就可以找到区间内所有素数；条件成立时，需要进行动态转移. 原因是，枚举到 u 时，已经转移到 $S(\cdot, \pi(u-1))$，于是 $\pi(u) = S(u, \pi(u-1))$ 且 $\pi(u-1) = S(u-1, \pi(u-1))$，前述条件成立必然意味着 u 是素数. 另外，动态转移方程中 $a-1$ 的取值也可以由 $S(u-1, \pi(u-1))$ 给出，无需额外维护. 由此，就得到如下实现：

参考实现

```cpp
// Modified from griff's codes:
//   https://gbroxey.github.io/blog/2023/04/09/lucy-fenwick.html
// Submission:
// * https://judge.yosupo.jp/submission/397250 (1e11)
// * https://www.luogu.com.cn/record/295680182 (1e13, TLE)

// Lucy's Quotient DP to find pi.
long long lucy_pi(long long n) {
  int s = std::sqrt(n + 0.25l);
  // D(n) = {floor(n/x): x=1,2,...,n}
  std::vector<long long> d;
  for (long long l = 1, r; l <= n; l = r + 1) {
    r = n / (n / l);
    d.push_back(r);
  }
  int m = d.size();
  auto id = [&](long long x) -> int { return x <= s ? x - 1 : m - n / x; };
  // Quotient DP.
  std::vector<long long> dp(m);
  for (int j = 0; j < m; ++j) dp[j] = d[j] - 1;
  for (int p = 2; p <= s; ++p) {
    if (dp[p - 1] == dp[p - 2]) continue;  // Only primes can survive.
    for (int j = m - 1; d[j] >= (long long)p * p; --j) {
      dp[j] -= dp[id(d[j] / p)] - dp[p - 2];
    }
  }
  return dp.back();
}

```

这一算法实现简单，虽然时间复杂度略差，但由于常数很小，对于较小数据规模表现优秀. 更为重要的是，因为 $\pi(x) = S(x, a)$ 对于所有 $a \geq \pi(\sqrt{x})$ 都成立，所以作为副产品，算法实际上得到了集合 $D(x)$ 内所有元素处 $\pi(\cdot)$ 的取值. 这一特性在积性函数求和问题中尤为重要，后续将讨论它的推广及应用.

树状数组优化

利用树状数组，很容易将 Lucy 算法的时间复杂度从 $\tilde{O}(x^{3/4})$ 降低到 $\tilde{O}(x^{2/3})$.

为此，取实数 $y \in (\sqrt{x}, x]$. 当前状态 $S(\cdot, a)$ 分成两部分维护：大于 y 的那一部分仍按照前述递推关系递归计算；不大于 y 的那一部分则利用 Eratosthenes 筛法维护，利用树状数组更新和查询当前 $S(\cdot, a)$ 的值.

算法的空间复杂度显然是 $O(y)$ 的. 为计算算法的时间复杂度，需要考虑如下三部分：Eratosthenes 筛法部分，每筛到一个合数就需要更新一次树状数组，共计 $O(y)$ 个合数，总时间成本为 $O(y \log y)$ 的；对于 $p_a^2 \leq y$，对应轮的状态转移需要进行 $O(x/y)$ 次，每次转移时查询操作是 $O(\log y)$ 的，共计 $O(\pi(\sqrt{y}))$ 次，总时间成本是

$$O\left(\pi(\sqrt{y})\frac{x}{y}\log y\right)=O\left(\frac{x}{\sqrt{y}}\right)$$

的；最后，对于 $p_a^2>y$，对应轮的状态转移需要进行 $O(x/p_a^2)$ 次，利用上一小节的方法可知，总时间成本为

$$O\left(\sum_{a=\pi(\sqrt{y})+1}^{\pi(\sqrt{x})}\frac{x}{p_a^2}\log y\right)=O\left(\frac{x}{\sqrt{y}}\right)$$

的．将三部分相加，令 $y=x^{2/3}\log^{-2/3}x$，就得到整体时间复杂度 $O(x^{2/3}\log^{1/3}x)$．此时，空间复杂度也是 $\tilde{O}(x^{2/3})$ 的.

参考实现

```cpp
// Submission:
// * https://judge.yosupo.jp/submission/397277 (1e11)
// * https://www.luogu.com.cn/record/295680548 (1e13, TLE)
// Binary-indexed tree.
struct BIT {
  int n;
  std::vector<int> su;

  BIT(int _n) : n(_n), su(_n + 1) {}

  void add(int x) {
    for (; x <= n; x += (x & (-x))) ++su[x];
  }

  int get(int x) {
    int res = 0;
    for (; x; x &= x - 1) res += su[x];
    return res;
  }
};

// Lucy's Quotient DP to find pi, with BIT optimization.
long long lucy_pi(long long n) {
  if (n <= 1) return 0;
  int s = std::sqrt(n + 0.25l);
  int y = std::pow(n / std::log(n), 2.0l / 3);
  y = std::max(y, s + 1);
  // D(n) = {floor(n/x): x=1,2,...,n}
  std::vector<long long> d;
  for (long long l = 1, r; l <= n; l = r + 1) {
    r = n / (n / l);
    d.push_back(r);
  }
  int m = d.size();
  auto id = [&](long long x) -> int { return x <= s ? x - 1 : m - n / x; };
  // Quotient DP.
  std::vector<bool> vis(y + 1);
  BIT bit(y);
  std::vector<long long> dp(m);
  for (int j = 0; j < m; ++j) dp[j] = d[j] - 1;
```

```
  auto eval = [&](long long x) -> long long {
    return x <= y ? x - 1 - bit.get(x) : dp[id(x)];
  };
  for (long long p = 2; p * p <= n; ++p) {
    if (vis[p]) continue;
    auto a_1 = eval(p - 1);
    auto lim = n / std::max(p * p, (long long)y);
    for (int i = 1; i <= lim; ++i) {
      dp.end()[-i] -= eval(n / (p * i)) - a_1;
    }
    // Sieve.
    for (auto x = p * p; x <= y; x += p) {
      if (!vis[x]) {
        vis[x] = true;
        bit.add(x);
      }
    }
  }
  return dp.back();
}

```

需要说明的是，尽管理论时间复杂度确实降低，但是引入树状数组后带来的常数损失使得算法运行效率在较小数据规模时反而降低，而在较大数据规模时显著恶化的空间占用又限制了算法使用．所以，这一优化的实用性不高．

进一步优化

虽然树状数组成功地将 Lucy 算法优化至 $\tilde{O}(x^{2/3})$，但实际运行效率仍然不高．为了得到高效的算法，可以进一步对 Lucy 算法做出如下优化：（设 $z < w < y < x$，且都是 x 的幂次）

1. 树状数组优化的 Lucy 算法中，Eratosthenes 筛法部分的时间成本是 $O(y \log y)$ 的．实际上，最开始若干轮状态转移中，筛去的合数数量庞大，没有必要使用树状数组维护．因此，可以选取 $z < y$．当 $p_a \le z$ 时，只使用状态转移方程；当 $p_a > z$ 时，再引入树状数组维护 $S(\cdot, a)$ 中不超过 y 的部分．由于引入树状数组时，已经筛去了所有含有不大于 z 的素因子的整数，剩下的数——常称作 z-粗糙数——中不超过 y 的数只有 $O\left(\dfrac{y}{\log z}\right)$ 个[54]．由此，就可以将筛法部分时间成本降低到 $O(y)$ 的．最开始这些轮状态转移引入的时间成本是 $O(x^{1/2}z/\log x)$ 的．
2. 前文描述的 Lucy 算法最终得到的都是 $D(x)$ 中所有元素处 $\pi(\cdot)$ 的值．如果只想得到 $\pi(x)$ 的值，很多状态转移是没有必要的．例如，如果要计算 $S(x, a)$ 的取值，只需要知道 $S(x, a-1)$ 和 $S\left(\dfrac{x}{p_a}, a-1\right)$ 的取值；而要计算这两项的取值，又只需要知道

   $$S(x, a-2),\ S\left(\frac{x}{p_a}, a-2\right),\ S\left(\frac{x}{p_{a-1}}, a-2\right),\ S\left(\frac{x}{p_a p_{a-1}}, a-2\right)$$

   的取值．由此归纳可知，在进行到第 i 轮状态转移时，需要知道取值的状态 $S\left(\dfrac{x}{n}, i\right)$ 中的 n 必然是 p_i-粗糙数．由于 p_i-粗糙数对于乘法是封闭的，所以只需要在每次状态转移时，都能处理到所有 p_i-粗糙数（作为除数的结点），就能保证状态转移终止时，所有 p_i-粗糙数处结点值都是正确的．

[54] 参见 Buchstab function - Wikipedia[2].
[55] 动态维护的实现可以参考 griff 博文的优化技巧一节[3].

当然，p_i-粗糙数的集合可以在筛法过程中动态维护[55]；又或者，可以复用前文 z-粗糙数的集合，用它代替所有 p_i-粗糙数．看似第二种做法多做了一些无用功，但是它们都使得动态转移部分的复杂度减少了一个 $\log x$，只是在常数上存在差异．

3. 和筛法部分类似，状态转移部分同样存在树状数组引入额外 $\log x$ 的问题．为了减少树状数组查询，可以将树状数组中不再更新的部分存储到静态数组中．具体地，当动态转移进行到第 i 轮时，不会改变 $u < p_i^2$ 处 $S(u,i)$ 的取值，就可以将它们查询出来并存储．

   下面计算这样做带来的复杂度改进．为了引入其他优化的影响，此处设带树状数组筛法的状态转移只用于处理 $z < p_i \le w$ 的部分，并假定 $y \ge w^2$，以保证下文计数是紧的．由于需要进行状态转移的 u 必然具有 $\lfloor x/j \rfloor$ 的形式，只要对相应的 j 计数就可以了．为方便计算，忽略不等式边界处的讨论．由于 $u > y$，必然有 $j < x/y$．如果 $jp_i < x/y$，那么不会涉及树状数组；如果 $x/y < jp_i < \min\{x/p_i^2, p_i x/y\}$，只能在树状数组上查询；如果 $\min\{x/p_i^3, x/y\} < j < x/y$，可以在静态数组或树状数组中查询．第一部分的总计数为

   $$\sum_{z<p_i<w} \frac{x}{yp_i} = \frac{x}{y} \log \frac{\log w}{\log z} \in O\left(\frac{x}{y}\right).$$

   第二部分的总计数为

   $$\sum_{z<p_i<y^{1/3}} \frac{x}{y}\left(1 - \frac{1}{p_i}\right) + \sum_{y^{1/3}<p_i<w} \left(\frac{x}{p_i^3} - \frac{x}{yp_i}\right) \le \frac{x}{y}\pi(y^{1/3}) + \frac{x}{y^{2/3}\log y} \in O\left(\frac{x}{y^{2/3}\log y}\right).$$

   第三部分的总计数为

   $$\sum_{y^{1/3}<p_i<w} \left(\frac{x}{y} - \frac{x}{p_i^3}\right) \le \pi(w)\frac{x}{y} \in O\left(\frac{xw}{y\log w}\right).$$

   第一部分计数远小于另外两部分，可以忽略不计．当 $w^2 \le y \ll w^3$ 时，第三部分总计数远大于第二部分．只有此时，利用静态数组存储这一部分值才会将状态转移部分的总时间复杂度减小一个 $\log x$；否则，第二部分的查询操作会成为瓶颈．综上，引入该优化后，动态转移部分的总时间复杂度为

   $$O\left(\max\left\{\frac{x}{y^{2/3}}, \frac{xw}{y\log x}\right\}\right).$$

   注意到，因为粗糙数的分布大致是均匀的，第二条优化（即状态转移部分的粗糙数优化）的效果和这一条是独立的．如果同时应用粗糙数优化，此处得到的复杂度可以再少一个 $\log x$．

4. 状态转移可以提前终止，只更新到 $p_a \le w$ 的部分，对于剩余的部分利用 Meissel–Lehmer 算法的思想解决．与依赖于 $\varphi(x,a)$ 的 Meissel–Lehmer 算法不同，$S(x,a)$ 中保存着直到 $x \le p_a^2$ 为止所需要的 $\pi(x)$ 的取值，无需额外预处理．但是和前文优化方法结合使用时，需要注意动态规划部分粗糙数优化（即优化 2）的影响，不要用到未正确更新的值．这样做对于算法的时空复杂度都有改进，但具体改进幅度高度依赖于其余部分实现，在此不做一般分析．下文会对参考实现中的这一部分做具体分析．

将这四种小优化结合到一起．任选 $w \le x^{1/3}$，对于 $z < w$ 和 $w^2 \le y < w^3$，筛法部分和状态转移部分的总时间复杂度为

$$O\left(y + \frac{z\sqrt{x}}{\log x} + \frac{xw}{y\log x}\right).$$

于是，在 $y = x^{5/8}\log^{-1} x$ 且 $z = w^{1/2}$ 时，总时间复杂度达到 $O\left(\frac{\sqrt{xw}}{\log x}\right)$．其中，$z$ 在不增加总体复杂度的前提下，尽可能取得大，是为了减少 z-粗糙数的密度，降低算法常数．

如果算法提前终止在 $w = x^{1/3}$ 处，算法时空复杂度就都至少[56]是 $O(x^{2/3}\log^{-1} x)$ 的．下面说明，如果将算法在 $w = x^{1/4}$ 处终止，将得到更低的时空复杂度．

[56]经过简单分析可知，算法时空复杂度其实恰好是 $O(x^{2/3}\log^{-1} x)$.

设 $w=x^{1/4}$．由于提前终止在 $a=\pi(x^{1/4})$ 处，要得到 $\pi(x)$，需要利用关系：

$$\pi(x)=S(x,a)-P_2(x,a)-P_3(x,a).$$

由前文分析可知

$$P_2(x,a)=\sum_{i=a+1}^{\pi(\sqrt{x})}\pi\left(\frac{x}{p_i}\right)-\frac{1}{2}(\pi(\sqrt{x})+a-1)(\pi(\sqrt{x})-a).$$

由于已经筛到了 p_a，所以 $\pi(\sqrt{x})$ 可以从 $S(\cdot,a)$ 中获得．但是，因为 $x/p_i\in[\sqrt{x},x^{3/4})$，求和式中的 $\pi(\cdot)$ 仍然和 $S(\cdot,a)$ 不一致．于是，进一步展开，有

$$\sum_{i=a+1}^{\pi(\sqrt{x})}\pi\left(\frac{x}{p_i}\right)=\sum_{i=a+1}^{\pi(\sqrt{x})}S\left(\frac{x}{p_i},a\right)-\sum_{i=a+1}^{\pi(\sqrt{x})}P_2\left(\frac{x}{p_i},a\right).$$

代回前式，就得到

$$\begin{aligned}\pi(x)=S(x,a)&+\frac{1}{2}(\pi(\sqrt{x})+a-1)(\pi(\sqrt{x})-a)\\&-\sum_{i=a+1}^{\pi(\sqrt{x})}S\left(\frac{x}{p_i},a\right)+\underbrace{\sum_{i=a+1}^{\pi(\sqrt{x})}P_2\left(\frac{x}{p_i},a\right)-P_3(x,a)}_{\Delta(x)}.\end{aligned}$$

为了计算 $\Delta(x)$，考虑其组合意义．由 $P_2(x,a)$ 和 $P_3(x,a)$ 定义可知

$$\sum_{i=a+1}^{\pi(\sqrt{x})}P_2\left(\frac{x}{p_i},a\right)=\#\{p_ip_jp_k\le x:a<i,\ a<j\le k\},$$

$$P_3(x,a)=\#\{p_ip_jp_k\le x:a<i\le j\le k\}.$$

将两集合作差，就得到

$$\Delta(x)=\#\{p_ip_jp_k\le x:a<j,\ j<i,\ j\le k\}.$$

为了计数，仍然考虑枚举最小素因子 p_j（因为它的枚举上限最小）；对于固定的 j，其余下标必须满足 $i>j$ 且 $k\ge j$；于是，拆分 $k=j$ 的特殊情形，剩下情形就有 $i,k>j$，利用对称性可以进一步简化，就有如下表达式：

$$\Delta(x)=\underbrace{\sum_{j=a+1}^{\pi(x^{1/3})}\left(\pi\left(\frac{x}{p_j^2}\right)-j\right)}_{i>j=k}+\underbrace{2\sum_{j=a+1}^{\pi(x^{1/3})}\left(\sum_{i=j+1}^{\pi(\sqrt{x/p_j})}\left(\pi\left(\frac{x}{p_ip_j}\right)-i\right)\right)}_{i>k>j\text{ or }k>i>j}+\underbrace{\sum_{j=a+1}^{\pi(x^{1/3})}\left(\pi\left(\sqrt{\frac{x}{p_j}}\right)-j\right)}_{i=k>j}.$$

整理一下求和式，就得到

$$\Delta(x)=\sum_{j=a+1}^{\pi(x^{1/3})}\left(\pi\left(\frac{x}{p_j^2}\right)-j+2\sum_{i=j+1}^{\pi(\sqrt{x/p_j})}\pi\left(\frac{x}{p_ip_j}\right)-\left(\pi\left(\sqrt{\frac{x}{p_j}}\right)^2-j^2\right)\right).$$

利用这一表达式，计算 $\Delta(x)$ 的复杂度为（估算方法参考前文 Lucy 算法复杂度部分）

$$O\left(\sum_{x^{1/4}<p\le x^{1/3}}\pi\left(\sqrt{\frac{x}{p}}\right)\right)=O\left(\frac{x^{2/3}}{\log^2 x}\right).$$

这也正是最后一部分计算的复杂度．简单检查可以发现，计算 $\Delta(x)$ 时涉及到的所有 $\pi(x/n)$ 项都具有 $x/n\le\sqrt{x}<y$，所以它们都可以从树状数组存储的 $S(\cdot,a)$ 中直接获得（需提前存到静态数组中，以避免引入额外的 $O(\log x)$ 查询）．

再结合前文分析可知，当 $z=x^{1/8}$，$w=x^{1/4}$，$y=x^{5/8}\log^{-1}x$ 时，筛法和动态规划部分的总时间复杂度是 $O(x^{5/8}\log^{-1}x)$，所以算法整体时间复杂度是 $O(x^{2/3}\log^{-2}x)$．空间复杂度就是树状数组的长度 $O(y)=O(x^{5/8}\log^{-1}x)$．

参考实现

```cpp
// Modified from 渐变色's codes:
//   https://www.luogu.com.cn/article/q4d4jl20
// Submission:
// * https://judge.yosupo.jp/submission/398981 (1e11)
// * https://www.luogu.com.cn/record/296415705 (1e13)
long long count_pi(long long n) {
  if (n <= 1) return 0;
  // Constants.
  int v = std::sqrt(n + 0.25l);               // x^{1/2}.
  int w = std::sqrt(v + 0.25l);               // x^{1/4}. Phase II stops here.
  int z = std::min(w, (int)std::sqrt(w) * 2);  // ~x^{1/8}. Phase I stops here.
  int y = std::pow(n, 0.625l) / std::log(n) * 2;  // ~x^{5/8}/log(x). BIT size.
  y = std::min<long long>(std::max(y, v), n);
  int B = n / y;  // the largest j such that n/j >= y.
  B = std::min<long long>(n / (n / B), y);
  // Initialize.
  std::vector<long long> l(v + 1);  // l[i] = S(n/i, .)
  std::vector<int> s(y + 1);        // phase I: S(i, .); phase II: BIT.
  std::vector<bool> e(y + 1);       // e[i] == true once i has been crossed out.
  std::vector<int> pi(y + 1);       // prefix for e; then pi(.) for O(1) query.
  for (int i = 1; i <= v; ++i) l[i] = n / i - 1;  // Lucy DP initialization.
  for (int i = 1; i <= v; ++i) s[i] = i - 1;

  // ============ PHASE I: plain Lucy DP, p <= z ~ n^(1/8) ==================
  int p;
  for (p = 2; p <= z; ++p) {
    if (s[p] != s[p - 1]) {  // when p is prime.
      auto m = n / p;
      int t0 = s[p - 1];  // the number of primes < p.
      int t = v / p;      // splitting point by where n/(i*p) lands.
      // S(u, .) -= (S(u / p) - t0), case by case.
      for (int i = 1; i <= t; ++i) l[i] -= l[i * p] - t0;
      for (int i = t + 1; i <= v; ++i) l[i] -= s[m / i] - t0;
      for (int i = v, j = t; j >= p; --j)
        for (int k = j * p; i >= k; --i) s[i] -= s[j] - t0;
      // Sieve.
      for (int i = p * p; i <= y; i += p) e[i] = 1;
    }
  }
  // Obtain z-rough number list for [2,B].
  e[1] = 1;
  int id = 1;
  std::vector<int> roughs(B + 1);
  for (int i = 1; i <= B; ++i)
```

```
    if (!e[i]) roughs[id++] = i;
  roughs[id] = 0x7fffffff;  // sentinel: terminates every walk.
  // Build the BIT (i.e., Fenwick tree).
  for (int i = 1; i <= y; ++i) pi[i] = pi[i - 1] + e[i];
  for (int i = 1; i <= y; ++i) s[i] = pi[i] - pi[i & (i - 1)];
  // BIT query: obtain S(x,.).
  const auto query = [&](int x) -> int {
    int sum = x;
    for (; x; x &= x - 1) sum -= s[x];
    return sum;
  };
  // BIT modify: mark composite and add 1 in the BIT.
  const auto add = [&](int x) -> void {
    e[x] = 1;
    for (; x <= y; x += x & -x) ++s[x];
  };

  // ============ PHASE II: Lucy DP with BIT sieve, z < p <= n^(1/4)
=========
  id = 1;
  for (; p <= w; ++p) {
    if (e[p]) continue;
    auto q = (long long)p * p;
    auto m = n / p;
    int t0 = query(p - 1);  // the number of primes < p.
    // freeze finalized values for O(1) query.
    // the frozen S(i, .) is actually pi(i).
    for (; id < q; ++id) pi[id] = query(id);
    // splitting point by where n/(i*p) lands.
    int t1 = B / p;
    int t2 = std::min<long long>(B, m / q);
    // S(u, .) -= (S(u / p) - t0), case by case.
    int i = 1, j = 1;
    for (; i <= t1; i = roughs[++j]) l[i] -= l[i * p] - t0;
    for (; i <= t2; i = roughs[++j]) l[i] -= query(m / i) - t0;
    for (; i <= B; i = roughs[++j]) l[i] -= pi[m / i] - t0;
    // Sieve (with BIT).
    for (int i = q; i <= y; i += p)
      if (!e[i]) add(i);
  }
  // freeze value till v=x^{1/2}.
  for (; id <= v; ++id) pi[id] = query(id);
  // prime list till v=x^{1/2}.
  std::vector<int> primes;
  primes.push_back(1);  // dummy, p_0=1.
  for (int i = 2; i <= v; ++i)
    if (!e[i]) primes.push_back(i);

  // ============ PHASE III: stop early, correct with P2 and P3
==============
  // ---- -P2, part 1: the triangular part.
  l[1] += (pi[v] + pi[w] - 1LL) * (pi[v] - pi[w]) / 2;
```

```
  // ---- -P2, part 2: the sum of S(n/p,a).
  for (int i = pi[w] + 1; i <= pi[B]; ++i) l[1] -= l[primes[i]];
  for (int i = pi[B] + 1; i <= pi[v]; ++i) l[1] -= query(n / primes[i]);
  // ---- +Delta(n).
  for (int i = pi[w] + 1; i <= pi[v]; ++i) {
    int p = primes[i];
    auto m = n / p;
    int e = pi[m / p];  // pi(n/p^2).
    if (e <= i) break;  // i.e., p^3 > n.
    l[1] += e - i;
    long long t = 0;
    auto s = pi[(int)std::sqrt(m + 0.25l)];  // pi(sqrt(n/p)).
    for (int k = i + 1; k <= s; ++k) t += pi[m / primes[k]];
    l[1] += 2 * t - (long long)(i + s) * (s - i);
  }

  return l[1];
}

```

在算法竞赛常见数据范围 ($10^{10} \sim 10^{14}$) 内, 这一实现的时空成本都相当优秀. 需要说明的是, 尽管理论复杂度分析中, 最后一部分是复杂度瓶颈, 但是由于其复杂度中对数因子更小、常数更优, 在上述数据范围内, 算法表现的实际瓶颈仍然是筛法和动态规划部分. 这也正是对它们复杂度优化必不可少的原因.

推广：素数幂前缀和

更一般地, 考虑如下问题:

$$F_{\text{prime}}(x) = \sum_{p\in\mathbf{P},\ p\le x} f(p).$$

也就是说, 计算数论函数 $f(\cdot)$ 在不超过 x 的素数处取值的和. 素数计数函数 $\pi(x)$ 是 $f \equiv 1$ 时的特例.

假定数论函数 f 满足如下条件:

1. f 是完全积性函数, 即 $f(mn) = f(m)f(n)$ 对于所有正整数 m, n 都成立;
2. 前缀和 $F(x) = \sum_{n\le x} f(n)$ 容易计算.

满足这些条件的常见数论函数包括幂函数和 Dirichlet 特征[4] (例如 Legendre 符号→9.11.17@p1263). 当然, 如果某个函数可以表示为若干满足这些条件的函数线性组合, 它在素数处的前缀和也可以求得. 所以, 本节的方法可以处理 f 是多项式或者 f 是周期函数[57] 的情形. 另一种更为简洁的处理周期函数的方法详见后文例题.

在这些条件下, 可以推广前文的递推关系. 定义 $S(x, a)$ 为筛去前 a 个素数后, 区间 $[2, x]$ 内剩下整数处 $f(\cdot)$ 取值之和, 即

$$S(x, a) = \sum_{1<n\le x,\ (n\in\mathbf{P})\vee(\text{lpf}(n)>p_a)} f(n).$$

此时, 可以将前文关于 $S(x, a)$ 的递推关系改写如下:

$$S(x, a) = S(x, a-1) - [p_a^2 \le x] f(p_a)\left(S\left(\frac{x}{p_a}, a-1\right) - S(p_a - 1, a-1)\right).$$

[57] 所有周期为 m, 且仅在与 m 互素的整数处取非零值的数论函数, 都可以表示为若干个模 m 的 Dirichlet 特征的线性组合. 由于计算素数处的前缀和时, 只涉及有限个与 m 不互素的整数, 因此总可以对原周期函数进行适当调整, 使其在这些整数处取零, 从而将其表示为模 m 的 Dirichlet 特征的线性组合.

边界条件为 $S(x,0)=F(x)-1$. 该公式表达的内容和前文类似：Eratosthenes 筛法中，利用 p_a 筛去的合数一定是 p_a 和某个尚未被前 $a-1$ 个素数筛去的整数的乘积. 但是，全体尚未被前 $a-1$ 个素数筛去的整数中，还包括前 $a-1$ 个素数，它们需要额外剔除. 所以，为了得到这些合数处函数值的和，利用 **完全积性**，可以提取系数 $f(p_a)$，而剩下的和就是 $S(x/p_a,a-1)$ 减去前 $(a-1)$ 个素数的贡献 $S(p_a-1,a-1)$. 注意，减去的这一项也可以写作 $S(p_{a-1},a-1)$，它们是相等的.

利用这一递推关系，可以通过 Lucy 算法在 $O(x^{3/4}\log^{-1}x)$ 时间内计算 $F_{\text{prime}}(x)$ 的取值，空间复杂度仅为 $O(x^{1/2})$. 利用树状数组可以将时间复杂度降低为 $\tilde{O}(x^{2/3})$，但并不十分实用.

模板题 Luogu P5493 质数前缀统计[5] 参考实现

```cpp
#include <cmath>
#include <iostream>
#include <vector>

int solve(long long n, int k, int P) {
  int s = std::sqrt(n + 0.25l);
  // Linear sieve.
  std::vector<int> primes, vis(s + 1);
  for (int x = 2; x <= s; ++x) {
    if (!vis[x]) primes.push_back(x);
    for (int p : primes) {
      if (x * p > s) break;
      vis[x * p] = true;
      if (x % p == 0) break;
    }
  }
  // D(n) = {floor(n/x): x=1,2,...,n}.
  std::vector<long long> d;
  for (long long l = 1, r; l <= n; l = r + 1) {
    r = n / (n / l);
    d.push_back(r);
  }
  int m = d.size();
  auto id = [&](long long x) -> int { return x <= s ? x - 1 : m - n / x; };
  // binary exponentiation.
  auto pow = [&](long long a, int b) -> int {
    int res = 1, po = a % P;
    for (; b; b >>= 1) {
      if (b & 1) res = (long long)res * po % P;
      po = (long long)po * po % P;
    }
    return res;
  };
  // lagrange interpolation to initialize DP.
  std::vector<int> dp(m, P - 1), su(k + 2);
  for (int i = 1; i <= k + 1; ++i) {
    su[i] = (su[i - 1] + pow(i, k)) % P;
  }
  std::vector<int> ifa(k + 2);
  ifa[0] = ifa[1] = 1;
  for (int i = 2; i <= k + 1; ++i) {
    ifa[i] = (long long)(P - P / i) * ifa[P % i] % P;
```

```
  }
  for (int i = 2; i <= k + 1; ++i) {
    ifa[i] = (long long)ifa[i] * ifa[i - 1] % P;
  }
  for (int j = 0; j < m; ++j) {
    long long x = d[j] % P;
    std::vector<int> lp(k + 2), rp(k + 2);
    lp[0] = rp[k + 1] = 1;
    for (int i = 1; i <= k + 1; ++i) {
      lp[i] = lp[i - 1] * (x + P - (i - 1)) % P;
      rp[k + 1 - i] = rp[k + 2 - i] * (x + P - (k + 2 - i)) % P;
    }
    for (int i = 0; i <= k + 1; ++i) {
      dp[j] += ((k + 1 - i) % 2 ? P - 1LL : 1LL) * su[i] % P * lp[i] % P *
               rp[i] % P * ifa[i] % P * ifa[k + 1 - i] % P;
      if (dp[j] >= P) dp[j] -= P;
    }
  }
  // Lucy's Quotient DP.
  for (int p : primes) {
    auto pk_ = (P - 1LL) * pow(p, k) % P;
    for (int j = m - 1; d[j] >= (long long)p * p; --j) {
      dp[j] += pk_ * (dp[id(d[j] / p)] + P - dp[p - 2]) % P;
      if (dp[j] >= P) dp[j] -= P;
    }
  }
  // Output.
  long long res = 0;
  for (int i = 1; i <= s; ++i) {
    res += (long long)i * i % P * dp.end()[-i] % P;
  }
  return res % P;
}

int main() {
  long long n;
  int k, p;
  std::cin >> n >> k >> p;
  std::cout << solve(n, k, p) << std::endl;
  return 0;
}

```

原则上，对于推广后的 Lucy 算法建议预处理素数，而不是合并到动态规划中．这是因为判据 $S(u,\pi(u-1))\neq S(u-1,\pi(u-1))$ 对于一般的 f 未必成立．另外，尽管本节仅讨论了 Lucy 算法的推广，其他算法也可以做类似推广；但是 Lucy 算法可以处理出 $D(x)$ 内所有值处的 F_{prime} 值，对于后续积性函数求和更为有用，所以本节只介绍了它的推广．

例题

Codeforces 665 F. Four Divisors[6]

给定 n，求 $[1,n]$ 中恰有 4 个因数的整数个数．其中，$1 \le n \le 10^{11}$．

解答

恰有 4 个因数的整数必然具有形式 p^3 或 pq，其中，p,q 都是素数且 $p<q$．区间 $[1,n]$ 中，具有形式 p^3 的整数总计 $\pi(n^{1/3})$ 个；具有 pq 形式的整数共计

$$\sum_{p<\sqrt{n}}\left(\pi\left(\frac{n}{p}\right)-\pi(p)\right)$$

个．由于上述表达式中所涉及的素数计数函数均可通过集合 $D(n)$ 中的值计算，只需要执行一次 Lucy 算法，就可以完成求解．时间复杂度是 $O(n^{3/4}\log^{-1}n)$ 的，空间复杂度是 $O(n^{1/2})$ 的．

参考实现

```cpp
#include <cmath>
#include <iostream>
#include <vector>

long long solve(long long n) {
  int s = std::sqrt(n + 0.25l);
  // D(n) = {floor(n/x): x=1,2,...,n}
  std::vector<long long> d;
  for (long long l = 1, r; l <= n; l = r + 1) {
    r = n / (n / l);
    d.push_back(r);
  }
  int m = d.size();
  auto id = [&](long long x) -> int { return x <= s ? x - 1 : m - n / x; };
  // Lucy's Quotient DP to find pi.
  std::vector<long long> dp(m);
  for (int j = 0; j < m; ++j) dp[j] = d[j] - 1;
  for (int p = 2; p <= s; ++p) {
    if (dp[p - 1] == dp[p - 2]) continue;  // Only primes can survive.
    for (int j = m - 1; d[j] >= (long long)p * p; --j) {
      dp[j] -= dp[id(d[j] / p)] - dp[p - 2];
    }
  }
  // Count p^3.
  long long res = dp[(int)std::cbrt(n + 0.25l) - 1];
  // Count pq.
  for (int p = 2; p <= s; ++p) {
    if (dp[p - 1] == dp[p - 2]) continue;
    res += dp.end()[-p] - dp[p - 1];
  }
  return res;
}

int main() {
```

```
  long long n;
  std::cin >> n;
  std::cout << solve(n) << std::endl;
  return 0;
}

```

LOJ 6028.「from CommonAnts」质数计数 II[7]

给定 n, m，求 $[1, n]$ 中模 m 等于 $0, 1, 2, \ldots, m-1$ 的素数分别有多少个. 其中，$1 \le n \le 3 \times 10^{10}$，$1 < m \le 12$ 且 $n > m$.

解答

对于任意余数 $r = 0, 1, 2, \ldots, m-1$，函数 $[n \bmod m = r]$ 显然是周期函数. 前文说明，这类问题可以借助对 Dirichlet 特征求和进行处理，但过于繁琐且涉及复数计算. 因此，本题采用另一种方法解决.

考虑在 Lucy 算法的状态转移方程中增加一维. 令 $S(n, r, a)$ 表示 $[2, n]$ 的整数中，利用前 a 个素数筛完后，剩下的模 m 余 r 的整数个数. 和正文类似，可以得到递推关系如下：

$$S(n, rp_a \bmod m, a) = S(n, rp_a \bmod m, a-1) - [p_a^2 \le n]\left(S\left(\frac{n}{p_a}, r, a-1\right) - S(p_a - 1, r, a-1)\right).$$

递推公式中，函数第二个参数采用了乘法而非除法，是考虑到存在 $p_a \mid m$ 的可能. 边界条件为 $S(n, r, 0) = \left\lfloor \frac{n-r}{m} \right\rfloor + [r > 1]$. 据此，通过动态规划即可解决. 时间复杂度是 $O(mn^{3/4} \log^{-1} n)$ 的，空间复杂度是 $O(mn^{1/2})$ 的.

参考实现

```cpp
#include <cmath>
#include <iostream>
#include <numeric>
#include <vector>

std::vector<long long> solve(long long n, int m) {
  int s = std::sqrt(n + 0.25l);
  // Linear sieve.
  std::vector<int> primes, vis(s + 1);
  for (int x = 2; x <= s; ++x) {
    if (!vis[x]) primes.push_back(x);
    for (int p : primes) {
      if (x * p > s) break;
      vis[x * p] = true;
      if (x % p == 0) break;
    }
  }
  // D(n) = {floor(n/x): x=1,2,...,n}
  std::vector<long long> d;
  for (long long l = 1, r; l <= n; l = r + 1) {
    r = n / (n / l);
    d.push_back(r);
```

```
  }
  int sz = d.size();
  auto id = [&](long long x) -> int { return x <= s ? x - 1 : sz - n / x; };
  // Lucy DP mod MOD.
  std::vector<std::vector<long long>> dp(sz, std::vector<long long>(m));
  for (int j = 0; j < sz; ++j) {
    for (int r = 0; r < m; ++r) {
      dp[j][r] = (d[j] + m - r) / m - (r <= 1);
    }
  }
  for (int p : primes) {
    for (int j = sz - 1; d[j] >= (long long)p * p; --j) {
      for (int r = 0; r < m; ++r) {
        dp[j][r * p % m] -= dp[id(d[j] / p)][r] - dp[p - 2][r];
      }
    }
  }
  // Finalize.
  return dp.back();
}

int main() {
  long long n;
  int m;
  std::cin >> n >> m;
  auto res = solve(n, m);
  for (auto x : res) std::cout << x << '\n';
  return 0;
}

```

习题

- Counting Primes - Library Checker[8]
- P7884【模板】Meissel-Lehmer[9]
- SPOJ NTHPRIME - Nth Prime[10]
- SPOJ SUMPRIM1 - Sum of primes[11]
- SPOJ BBAD - Breaking Math[12]
- SPOJ APS2 - Amazing Prime Sequence (hard)[13]
- SPOJ DIVFACT4 - Divisors of factorial (extreme)[14]

参考文献与注释

- Meissel–Lehmer algorithm - Wikipedia[15]
- Lucy's Algorithm + Fenwick Trees - griff's math blog![16]
- Counting primes in $\tilde{O}(n^{2/3})$ by Maksim1744 - Codeforces[17]
- 题解 P7884【模板】Meissel–Lehmer 算法 by 渐变色 - 洛谷[18]
- Lehmer, Derrick H. "On the exact number of primes less than a given limit." Illinois Journal of Mathematics 3, no. 3 (1959): 381-388.
- Lagarias, Jeffrey C., Victor S. Miller, and Andrew M. Odlyzko. "Computing $\pi(x)$: the Meissel-Lehmer method." Mathematics of Computation 44.170 (1985): 537-560.

- Deléglise, Marc, and Joël Rivat. "Computing $\pi(x)$: the Meissel, Lehmer, Lagarias, Miller, Odlyzko method." Mathematics of Computation 65.213 (1996): 235-245.[19]
- Staple, Douglas B. "The combinatorial algorithm for computing $\pi(x)$." arXiv preprint arXiv:1503.01839 (2015).[20]

[1] Smooth number - Wikipedia

[2] Buchstab function - Wikipedia

[3] griff 博文的优化技巧一节

[4] Dirichlet 特征

[5] Luogu P5493 质数前缀统计

[6] Codeforces 665 F. Four Divisors

[7] LOJ 6028.「from CommonAnts」质数计数 II

[8] Counting Primes - Library Checker

[9] P7884【模板】Meissel-Lehmer

[10] SPOJ NTHPRIME - Nth Prime

[11] SPOJ SUMPRIM1 - Sum of primes

[12] SPOJ BBAD - Breaking Math

[13] SPOJ APS2 - Amazing Prime Sequence (hard)

[14] SPOJ DIVFACT4 - Divisors of factorial (extreme)

[15] Meissel–Lehmer algorithm - Wikipedia

[16] Lucy's Algorithm + Fenwick Trees - griff's math blog!

[17] Counting primes in $\tilde{O}(n^{2/3})$ by Maksim1744 - Codeforces

[18] 题解 P7884【模板】Meissel–Lehmer 算法 by 渐变色 - 洛谷

[19] Deléglise, Marc, and Joël Rivat. "Computing $\pi(x)$: the Meissel, Lehmer, Lagarias, Miller, Odlyzko method." Mathematics of Computation 65.213 (1996): 235-245.

[20] Staple, Douglas B. "The combinatorial algorithm for computing $\pi(x)$." arXiv preprint arXiv:1503.01839 (2015).

9.11.26 Powerful Number 筛

定义

Powerful Number（以下简称 PN）筛类似于杜教筛，或者说是杜教筛的一个扩展，可以拿来求一些积性函数的前缀和.

要求:

- 存在一个函数 g 满足:
  - g 是积性函数.
  - g 易求前缀和.
  - 对于质数 p，$g(p)=f(p)$.

假设现在要求积性函数 f 的前缀和 $F(n)=\sum_{i=1}^{n}f(i)$.

Powerful Number

定义：对于正整数 n，记 n 的质因数分解为 $n=\prod_{i=1}^{m}p_i^{e_i}$．$n$ 是 PN 当且仅当 $\forall 1\le i\le m, e_i>1$.

性质 1：所有 PN 都可以表示成 a^2b^3 的形式.

证明：若 e_i 是偶数，则将 $p_i^{e_i}$ 合并进 a^2 里；若 e_i 为奇数，则先将 p_i^3 合并进 b^3 里，再将 $p_i^{e_i-3}$ 合并进 a^2 里.

性质 2：n 以内的 PN 至多有 $O(\sqrt{n})$ 个.

证明：考虑枚举 a，再考虑满足条件的 b 的个数，有 PN 的个数约等于

$$\int_1^{\sqrt{n}}\sqrt[3]{\frac{n}{x^2}}\mathrm{d}x=O(\sqrt{n})$$

那么如何求出 n 以内所有的 PN 呢？线性筛找出 $\sqrt{n}$ 内的所有素数，再 DFS 搜索各素数的指数即可. 由于 n 以内的 PN 至多有 $O(\sqrt{n})$ 个，所以至多搜索 $O(\sqrt{n})$ 次.

PN 筛

首先，构造出一个易求前缀和的积性函数 g，且满足对于素数 p，$g(p)=f(p)$．记 $G(n)=\sum_{i=1}^{n}g(i)$．

然后，构造函数 $h=f/g$，这里的 / 表示狄利克雷卷积除法．根据狄利克雷卷积的性质可以得知 h 也为积性函数，因此 $h(1)=1$．$f=g*h$，这里 * 表示狄利克雷卷积．

对于素数 p，$f(p)=g(1)h(p)+g(p)h(1)=h(p)+g(p)\Longrightarrow h(p)=0$．根据 $h(p)=0$ 和 h 是积性函数可以推出对于非 PN 的数 n 有 $h(n)=0$，即 h 仅在 PN 处取有效值．

现在，根据 $f=g*h$ 有

$$\begin{aligned}F(n)&=\sum_{i=1}^{n}f(i)\\&=\sum_{i=1}^{n}\sum_{d\mid i}h(d)g\left(\frac{i}{d}\right)\\&=\sum_{d=1}^{n}\sum_{i=1}^{\lfloor\frac{n}{d}\rfloor}h(d)g(i)\\&=\sum_{d=1}^{n}h(d)\sum_{i=1}^{\lfloor\frac{n}{d}\rfloor}g(i)\\&=\sum_{d=1}^{n}h(d)G\left(\left\lfloor\frac{n}{d}\right\rfloor\right)\\&=\sum_{\substack{d=1\\d\text{ is PN}}}^{n}h(d)G\left(\left\lfloor\frac{n}{d}\right\rfloor\right)\end{aligned}$$

$O(\sqrt{n})$ 找出所有 PN，计算出所有 h 的有效值．对于 h 有效值的计算，只需要计算出所有 $h(p^c)$ 处的值，就可以根据 h 为积性函数推出 h 的所有有效值．现在对于每一个有效值 d，计算 $h(d)G\left(\left\lfloor\frac{n}{d}\right\rfloor\right)$ 并累加即可得到 $F(n)$．

下面考虑计算 $h(p^c)$，一共有两种方法：一种是直接推出 $h(p^c)$ 仅与 p,c 有关的计算公式，再根据公式计算 $h(p^c)$；另一种是根据 $f=g*h$ 有 $f(p^c)=\sum_{i=0}^{c}g(p^i)h(p^{c-i})$，移项可得 $h(p^c)=f(p^c)-\sum_{i=1}^{c}g(p^i)h(p^{c-i})$，现在就可以枚举素数 p 再枚举指数 c 求解出所有 $h(p^c)$．

过程

1. 构造 g
2. 构造快速计算 G 的方法
3. 计算 $h(p^c)$
4. 搜索 PN，过程中累加答案
5. 得到结果

对于第 3 步，可以直接根据公式计算，可以使用枚举法预处理打表，也可以搜索到了再临时推．

性质

以使用第二种方法计算 $h(p^c)$ 为例进行分析．可以分为计算 $h(p^c)$ 和搜索两部分进行分析．

对于第一部分，根据 $O(\sqrt{n})$ 内的素数个数为 $O\left(\frac{\sqrt{n}}{\log n}\right)$，每个素数 p 的指数 c 至多为 $\log n$，计算 $h(p^c)$ 需要循环 $(c-1)$ 次，由此有第一部分的时间复杂度为 $O\left(\frac{\sqrt{n}}{\log n}\cdot\log n\cdot\log n\right)=O(\sqrt{n}\log n)$，且这是一个宽松的上界．根据题目的不同还可以添加不同的优化，从而降低第一部分的时间复杂度．

对于搜索部分，由于 n 以内的 PN 至多有 $O(\sqrt{n})$ 个，所以至多搜索 $O(\sqrt{n})$ 次．对于每一个 PN，根据计算 G 的方法不同，时间复杂度也不同．例如，假设计算 $G\left(\left\lfloor\frac{n}{d}\right\rfloor\right)$ 的时间复杂度为 $O(1)$，则第二部分的复杂度为 $O(\sqrt{n})$.

特别地，若借助杜教筛计算 $G\left(\left\lfloor\frac{n}{d}\right\rfloor\right)$，则第二部分的时间复杂度为杜教筛的时间复杂度，即 $O(n^{\frac{2}{3}})$. 因为若事先计算一次 $G(n)$，并且预先使用线性筛优化和用支持快速随机访问的数据结构（如 C++ 中的 `std::map` 和 `std::unordered_map`）记录较大的值，则杜教筛过程中用到的 $G\left(\left\lfloor\frac{n}{d}\right\rfloor\right)$ 都是线性筛中记录的或者 `std::map` 中记录的，这一点可以直接用程序验证.

对于空间复杂度，其瓶颈在于存储 $h(p^c)$．若使用二维数组 a 记录，$a_{i,j}$ 表示 $h(p_i^j)$ 的值，则空间复杂度为 $O\left(\frac{\sqrt{n}}{\log n}\cdot\log n\right)=O(\sqrt{n})$.

例题

Luogu P5325【模板】Min_25 筛[1]

题意：给定积性函数 $f(p^k)=p^k(p^k-1)$，求 $\sum_{i=1}^{n}f(i)$.

易得 $f(p)=p(p-1)=\mathrm{id}(p)\varphi(p)$，构造 $g(n)=\mathrm{id}(n)\varphi(n)$.

考虑使用杜教筛求 $G(n)$，根据 $(\mathrm{id}\cdot\varphi)*\mathrm{id}=\mathrm{id}_2$ 可得 $G(n)=\sum_{i=1}^{n}i^2-\sum_{d=2}^{n}d\cdot G\left(\left\lfloor\frac{n}{d}\right\rfloor\right)$.

之后 $h(p^k)$ 的取值可以枚举计算，这种方法不再赘述.

此外，此题还可以直接求出 $h(p^k)$ 仅与 p,k 有关的公式，过程如下：

$$
\begin{aligned}
& f(p^k)=\sum_{i=0}^{k}g(p^{k-i})h(p^i)\\
\iff & p^k(p^k-1)=\sum_{i=0}^{k}p^{k-i}\varphi(p^{k-i})h(p^i)\\
\iff & p^k(p^k-1)=\sum_{i=0}^{k}p^{2k-2i-1}(p-1)h(p^i)\\
\iff & p^k(p^k-1)=h(p^k)+\sum_{i=0}^{k-1}p^{2k-2i-1}(p-1)h(p^i)\\
\iff & h(p^k)=p^k(p^k-1)-\sum_{i=0}^{k-1}p^{2k-2i-1}(p-1)h(p^i)\\
\iff & h(p^k)-p^2h(p^{k-1})=p^k(p^k-1)-p^{k+1}(p^{k-1}-1)-p(p-1)h(p^{k-1})\\
\iff & h(p^k)-ph(p^{k-1})=p^{k+1}-p^k\\
\iff & \frac{h(p^k)}{p^k}-\frac{h(p^{k-1})}{p^{k-1}}=p-1
\end{aligned}
$$

再根据 $h(p)=0$，通过累加法即可推出 $h(p^k)=(k-1)(p-1)p^k$.

参考代码

```cpp
#include <iostream>
#include <map>
using namespace std;

constexpr int MOD = 1e9 + 7;

```

```
template <typename T>
int mint(T x) {
  x %= MOD;
  if (x < 0) x += MOD;
  return x;
}

int add(int x, int y) { return x + y >= MOD ? x + y - MOD : x + y; }

int mul(int x, int y) { return (long long)1 * x * y % MOD; }

int sub(int x, int y) {
  return x < y ? x - y + MOD : x - y;  // 防止负数
}

int qp(int x, int y) {
  int r = 1;
  for (; y; y >>= 1) {
    if (y & 1) r = mul(r, x);
    x = mul(x, x);
  }
  return r;
}

int inv(int x) { return qp(x, MOD - 2); }

namespace PNS {
constexpr int N = 2e6 + 5;
constexpr int M = 35;

long long global_n;

int g[N], sg[N];

int h[N][M];
bool vis_h[N][M];

int ans;

int pcnt, prime[N], phi[N];
bool isp[N];

void sieve(int n) {
  pcnt = 0;
  for (int i = 2; i <= n; ++i) isp[i] = true;  // 判断质数数组
  phi[1] = 1;
  for (int i = 2; i <= n; ++i) {
    if (isp[i]) {
      ++pcnt;
      prime[pcnt] = i;
      phi[i] = i - 1;
    }
```

```
    for (int j = 1; j <= pcnt; ++j) {  // 筛去非质数
      long long nxt = (long long)1 * i * prime[j];
      if (nxt > n) break;
      isp[nxt] = false;
      if (i % prime[j] == 0) {  // i是非质数的情况
        phi[nxt] = phi[i] * prime[j];
        break;
      }
      phi[nxt] = phi[i] * phi[prime[j]];
    }
  }

  for (int i = 1; i <= n; ++i) g[i] = mul(i, phi[i]);

  sg[0] = 0;
  for (int i = 1; i <= n; ++i) sg[i] = add(sg[i - 1], g[i]);  // g函数的前缀和
}

int inv2, inv6;

void init() {
  sieve(N - 1);
  for (int i = 1; i <= pcnt; ++i) h[i][0] = 1, h[i][1] = 0;
  for (int i = 1; i <= pcnt; ++i) vis_h[i][0] = vis_h[i][1] = true;
  inv2 = inv(2);
  inv6 = inv(6);
}

int S1(long long n) { return mul(mul(mint(n), mint(n + 1)), inv2); }

int S2(long long n) {
  return mul(mul(mint(n), mul(mint(n + 1), mint(n * 2 + 1))), inv6);
}

map<long long, int> mp_g;

int G(long long n) {
  if (n < N) return sg[n];
  if (mp_g.count(n)) return mp_g[n];

  int ret = S2(n);
  for (long long i = 2, j; i <= n; i = j + 1) {
    j = n / (n / i);
    ret = sub(ret, mul(sub(S1(j), S1(i - 1)), G(n / i)));
  }
  mp_g[n] = ret;
  return ret;
}

void dfs(long long d, int hd, int pid) {
  ans = add(ans, mul(hd, G(global_n / d)));

```

```
  for (int i = pid; i <= pcnt; ++i) {
    if (i > 1 && d > global_n / prime[i] / prime[i]) break;  // 剪枝

    int c = 2;
    for (long long x = d * prime[i] * prime[i]; x <= global_n;
         x *= prime[i], ++c) {  // 计算f.g函数
      if (!vis_h[i][c]) {
        int f = qp(prime[i], c);
        f = mul(f, sub(f, 1));
        int g = mul(prime[i], prime[i] - 1);
        int t = mul(prime[i], prime[i]);

        for (int j = 1; j <= c; ++j) {
          f = sub(f, mul(g, h[i][c - j]));
          g = mul(g, t);
        }
        h[i][c] = f;
        vis_h[i][c] = true;
      }

      if (h[i][c]) dfs(x, mul(hd, h[i][c]), i + 1);
    }
  }
}

int solve(long long n) {
  global_n = n;
  ans = 0;
  dfs(1, 1, 1);
  return ans;
}
}  // namespace PNS

int main() {
  PNS::init();
  long long n;
  cin >> n;
  cout << PNS::solve(n) << '\n';
  return 0;
}

```

「LOJ #6053」简单的函数[2]

给定 $f(n)$:

$$f(n)=\begin{cases}1 & n=1\\ p\oplus c & n=p^c\\ f(a)f(b) & n=ab \text{ and } a\perp b\end{cases}$$

易得:

$$f(p)=\begin{cases}p+1 & p=2\\ p-1 & \text{otherwise}\end{cases}$$

构造 g 为

$$g(n)=\begin{cases}3\varphi(n) 2\mid n\\ \varphi(n)\ \text{otherwise}\end{cases}$$

易证 $g(p)=f(p)$ 且 g 为积性函数.

下面考虑求 $G(n)$.

$$\begin{aligned}G(n)&=\sum_{i=1}^{n}[i \bmod 2=1]\varphi(i)+3\sum_{i=1}^{n}[i \bmod 2=0]\varphi(i)\\&=\sum_{i=1}^{n}\varphi(i)+2\sum_{i=1}^{n}[i \bmod 2=0]\varphi(i)\\&=\sum_{i=1}^{n}\varphi(i)+2\sum_{i=1}^{\lfloor\frac{n}{2}\rfloor}\varphi(2i)\end{aligned}$$

记 $S_1(n)=\sum_{i=1}^{n}\varphi(i)$，$S_2(n)=\sum_{i=1}^{n}\varphi(2i)$，则 $G(n)=S_1(n)+2S_2\left(\left\lfloor\frac{n}{2}\right\rfloor\right)$.

当 $2\mid n$ 时，有

$$\begin{aligned}S_2(n)&=\sum_{i=1}^{n}\varphi(2i)\\&=\sum_{i=1}^{\frac{n}{2}}(\varphi(2(2i-1))+\varphi(2(2i)))\\&=\sum_{i=1}^{\frac{n}{2}}(\varphi(2i-1)+2\varphi(2i))\\&=\sum_{i=1}^{\frac{n}{2}}(\varphi(2i-1)+\varphi(2i))+\sum_{i=1}^{\frac{n}{2}}\varphi(2i)\\&=\sum_{i=1}^{n}\varphi(i)+S_2\left(\frac{n}{2}\right)\\&=S_1(n)+S_2\left(\left\lfloor\frac{n}{2}\right\rfloor\right)\end{aligned}$$

当 $2\nmid n$ 时，有

$$\begin{aligned}S_2(n)&=S_2(n-1)+\varphi(2n)\\&=S_2(n-1)+\varphi(n)\\&=\sum_{i=1}^{n-1}\varphi(i)+S_2\left(\frac{n-1}{2}\right)+\varphi(n)\\&=S_1(n)+S_2\left(\left\lfloor\frac{n}{2}\right\rfloor\right)\end{aligned}$$

综上，有 $S_2(n)=S_1(n)+S_2\left(\left\lfloor\frac{n}{2}\right\rfloor\right)$.

S_1 可以用杜教筛求，S_2 直接按照公式推，这样 G 也可以求出来了.

参考代码

```cpp
#include <iostream>
#include <map>
using namespace std;

constexpr int MOD = 1e9 + 7;
constexpr int inv2 = (MOD + 1) / 2;

```

```cpp
template <typename T>
int mint(T x) {
  x %= MOD;
  if (x < 0) x += MOD;
  return x;
}

int add(int x, int y) {
  return x + y >= MOD ? x + y - MOD : x + y;  // 防止大于模数
}

int mul(int x, int y) { return (long long)1 * x * y % MOD; }

int sub(int x, int y) {
  return x < y ? x - y + MOD : x - y;  // 防负数
}

namespace PNS {
constexpr int N = 2e6 + 5;
constexpr int M = 35;

long long global_n;

int s1[N], s2[N];

int h[N][M];
bool vis_h[N][M];

int ans;

int pcnt, prime[N], phi[N];
bool isp[N];

void sieve(int n) {
  pcnt = 0;
  for (int i = 2; i <= n; ++i) isp[i] = true;  // 判断质数数组
  phi[1] = 1;
  for (int i = 2; i <= n; ++i) {
    if (isp[i]) {
      ++pcnt;
      prime[pcnt] = i;
      phi[i] = i - 1;
    }
    for (int j = 1; j <= pcnt; ++j) {  // 筛去非质数
      long long nxt = (long long)1 * i * prime[j];
      if (nxt > n) break;
      isp[nxt] = false;
      if (i % prime[j] == 0) {  // i是非质数的情况
        phi[nxt] = phi[i] * prime[j];
        break;
      }
      phi[nxt] = phi[i] * phi[prime[j]];
```

```
60      }
61    }
62
63    s1[0] = 0;
64    for (int i = 1; i <= n; ++i) s1[i] = add(s1[i - 1], phi[i]);
65
66    s2[0] = 0;
67    for (int i = 1; i <= n / 2; ++i) {
68      s2[i] = add(s2[i - 1], phi[2 * i]);
69    }
70  }
71
72  void init() {
73    sieve(N - 1);
74    for (int i = 1; i <= pcnt; ++i) h[i][0] = 1;
75    for (int i = 1; i <= pcnt; ++i) vis_h[i][0] = true;
76  }
77
78  map<long long, int> mp_s1;
79
80  int S1(long long n) {
81    if (n < N) return s1[n];
82    if (mp_s1.count(n)) return mp_s1[n];
83
84    int ret = mul(mul(mint(n), mint(n + 1)), inv2);
85    for (long long i = 2, j; i <= n; i = j + 1) {
86      j = n / (n / i);
87      ret = sub(ret, mul(mint(j - i + 1), S1(n / i)));
88    }
89    mp_s1[n] = ret;
90    return ret;
91  }
92
93  map<long long, int> mp_s2;
94
95  int S2(long long n) {
96    if (n < N / 2) return s2[n];
97    if (mp_s2.count(n)) return mp_s2[n];
98    int ret = add(S1(n), S2(n / 2));
99    mp_s2[n] = ret;
100   return ret;
101 }
102
103 int G(long long n) { return add(S1(n), mul(2, S2(n / 2))); }
104
105 void dfs(long long d, int hd, int pid) {
106   ans = add(ans, mul(hd, G(global_n / d)));
107
108   for (int i = pid; i <= pcnt; ++i) {
109     if (i > 1 && d > global_n / prime[i] / prime[i]) break;  // 剪枝
110
111     int c = 2;
```

```
    for (long long x = d * prime[i] * prime[i]; x <= global_n;
         x *= prime[i], ++c) {
      if (!vis_h[i][c]) {
        int f = prime[i] ^ c, g = prime[i] - 1;

        // p = 2时特判一下
        if (i == 1) g = mul(g, 3);

        for (int j = 1; j <= c; ++j) {
          f = sub(f, mul(g, h[i][c - j]));
          g = mul(g, prime[i]);
        }
        h[i][c] = f;
        vis_h[i][c] = true;
      }

      if (h[i][c]) dfs(x, mul(hd, h[i][c]), i + 1);
    }
  }
}

int solve(long long n) {
  global_n = n;
  ans = 0;
  dfs(1, 1, 1);
  return ans;
}
}  // namespace PNS

int main() {
  PNS::init();  // 预处理函数
  long long n;
  cin >> n;
  cout << PNS::solve(n) << '\n';
  return 0;
}

```

习题

- PE708 Twos are all you need[3]
- PE639 Summing a multiplicative function[4]
- PE484 Arithmetic Derivative[5]

参考资料

- 破壁人五号 - Powerful number 筛略解[6]
- command_block - 杜教筛（+ 贝尔级数 + powerful number）[7]

外部参考资料

[1] Luogu P5325【模板】Min_25 筛

[2] 「LOJ #6053」简单的函数

[3] PE708 Twos are all you need

[4] PE639 Summing a multiplicative function  [5] PE484 Arithmetic Derivative  [6] 破壁人五号 - Powerful number 筛略解 

[7] command_block - 杜教筛（+贝尔级数 + powerful number） 

9.11.27 Min_25 筛

Authors: Marcythm, Xeonacid, CSPNOIP

定义

从此种筛法的思想方法来说，其又被称为「Extended Eratosthenes Sieve」.

由于其由 Min_25[1] 发明并最早开始使用，故称「Min_25 筛」.

性质

其可以在 $O\left(\frac{n^{\frac{3}{4}}}{\log n}\right)$ 或 $\Theta(n^{1-\epsilon})$ 的时间复杂度下解决一类 **积性函数** 的前缀和问题.

要求：$f(p)$ 是一个关于 p 可以快速求值的完全积性函数之和（例如多项式）；$f(p^c)$ 可以快速求值.

记号

- **如无特别说明，本节中所有记为 p 的变量的取值集合均为全体质数.**
- $x/y := \left\lfloor \frac{x}{y} \right\rfloor$
- $\operatorname{isprime}(n) := [\,|\,\{d : d \mid n\}\,| = 2]$，即 n 为质数时其值为 1，否则为 0.
- p_k：全体质数中第 k 小的质数（如：$p_1 = 2, p_2 = 3$）. 特别地，令 $p_0 = 1$.
- $\operatorname{lpf}(n) := [1 < n] \min\{p : p \mid n\} + [1 = n]$，即 n 的最小质因数. 特别地，$n = 1$ 时，其值为 1.
- $F_{\text{prime}}(n) := \sum_{2 \le p \le n} f(p)$
- $F_k(n) := \sum_{i=2}^{n} [p_k \le \operatorname{lpf}(i)] f(i)$

解释

观察 $F_k(n)$ 的定义，可以发现答案即为 $F_1(n) + f(1) = F_1(n) + 1$.

考虑如何求出 $F_k(n)$. 通过枚举每个 i 的最小质因子及其次数可以得到递推式：

$$
\begin{aligned}
F_k(n) &= \sum_{i=2}^{n} [p_k \le \operatorname{lpf}(i)] f(i) \\
&= \sum_{\substack{k \le i \\ p_i^2 \le n}} \sum_{\substack{c \ge 1 \\ p_i^c \le n}} f(p_i^c)([c > 1] + F_{i+1}(n/p_i^c)) + \sum_{\substack{k \le i \\ p_i \le n}} f(p_i) \\
&= \sum_{\substack{k \le i \\ p_i^2 \le n}} \sum_{\substack{c \ge 1 \\ p_i^c \le n}} f(p_i^c)([c > 1] + F_{i+1}(n/p_i^c)) + F_{\text{prime}}(n) - F_{\text{prime}}(p_{k-1}) \\
&= \sum_{\substack{k \le i \\ p_i^2 \le n}} \sum_{\substack{c \ge 1 \\ p_i^{c+1} \le n}} \left(f(p_i^c) F_{i+1}(n/p_i^c) + f\left(p_i^{c+1}\right)\right) + F_{\text{prime}}(n) - F_{\text{prime}}(p_{k-1})
\end{aligned}
$$

最后一步推导基于这样一个事实：对于满足 $p_i^c \le n < p_i^{c+1}$ 的 c，有 $p_i^{c+1} > n \iff n/p_i^c < p_i < p_{i+1}$，故 $F_{i+1}(n/p_i^c) = 0$.

其边界值即为 $F_k(n) = 0 (p_k > n)$.

假设现在已经求出了所有的 $F_{\text{prime}}(n)$，那么有两种方式可以求出所有的 $F_k(n)$：

1. 直接按照递推式计算.
2. 从大到小枚举 p 转移，仅当 $p^2 < n$ 时转移增加值不为零，故按照递推式后缀和优化即可.

现在考虑如何计算 $F_{\text{prime}}(n)$.

观察求 $F_k(n)$ 的过程，容易发现 F_{prime} 有且仅有 $1,2,\dots,\lfloor\sqrt{n}\rfloor,n/\sqrt{n},\dots,n/2,n$ 这 $O(\sqrt{n})$ 处的点值是有用的.

一般情况下，$f(p)$ 是一个关于 p 的低次多项式，可以表示为 $f(p)=\sum a_i p^{c_i}$.

那么对于每个 p^{c_i}，其对 $F_{\text{prime}}(n)$ 的贡献即为 $a_i\sum_{2\le p\le n}p^{c_i}$.

分开考虑每个 p^{c_i} 的贡献，问题就转变为了：给定 $n,s,g(p)=p^s$，对所有的 $m=n/i$，求 $\sum_{p\le m}g(p)$.

> **注意**
>
> $g(p)=p^s$ 是完全积性函数!

于是设 $G_k(n):=\sum_{i=2}^{n}[p_k<\text{lpf}(i)\vee\text{isprime}(i)]g(i)$，即埃筛第 k 轮筛完后剩下的数的 g 值之和.

对于一个合数 $x\le n$，必定有 $\text{lpf}(x)\le\sqrt{x}\le\sqrt{n}$. 设 $p_{\ell(n)}$ 为不大于 $\sqrt{n}$ 的最大质数，则 $\sum_{2\le p\le n}g(p)=G_{\ell(n)}(n)$，即在埃筛进行 ℓ 轮之后剩下的均为质数. 考虑 G 的边界值，显然为 $G_0(n)=\sum_{i=2}^{n}g(i)$.（还记得吗？特别约定了 $p_0=1$）

对于转移，考虑埃筛的过程，分开讨论每部分的贡献，有：

1. 对于 $n<p_k^2$ 的部分，G 值不变，即 $G_k(n)=G_{k-1}(n)$.
2. 对于 $p_k^2\le n$ 的部分，被筛掉的数必有质因子 p_k，即 $-g(p_k)G_{k-1}(n/p_k)$.
3. 对于第二部分，由于 $p_k^2\le n\iff p_k\le n/p_k$，满足 $\text{lpf}(i)<p_k$ 的 i 会被额外减去. 这部分应当加回来，即 $g(p_k)G_{k-1}(p_{k-1})$.

则有：

$$G_k(n)=G_{k-1}(n)-[p_k^2\le n]g(p_k)(G_{k-1}(n/p_k)-G_{k-1}(p_{k-1}))$$

复杂度分析

对于 $F_k(n)$ 的计算，其第一种方法的时间复杂度被证明为 $O(n^{1-\epsilon})$（见朱震霆集训队论文《一些特殊的数论函数求和问题》[2] 2.3）；

对于第二种方法，其本质即为洲阁筛的第二部分，在任之洲论文《积性函数求和的几种方法》[3] 中也有提及（6.5.4），其时间复杂度被证明为 $O\left(\frac{n^{\frac{3}{4}}}{\log n}\right)$.

对于 $F_{\text{prime}}(n)$ 的计算，事实上，其实现与洲阁筛第一部分是相同的.

考虑对于每个 $m=n/i$，只有在枚举满足 $p_k^2\le m$ 的 p_k 转移时会对时间复杂度产生贡献，则时间复杂度可估计为：

$$\begin{aligned}T(n)&=\sum_{i^2\le n}O\left(\pi\left(\sqrt{i}\right)\right)+\sum_{i^2\le n}O\left(\pi\left(\sqrt{\frac{n}{i}}\right)\right)\\&=\sum_{i^2\le n}O\left(\frac{\sqrt{i}}{\ln\sqrt{i}}\right)+\sum_{i^2\le n}O\left(\frac{\sqrt{\frac{n}{i}}}{\ln\sqrt{\frac{n}{i}}}\right)\\&=O\left(\int_1^{\sqrt{n}}\frac{\sqrt{\frac{n}{x}}}{\log\sqrt{\frac{n}{x}}}\mathrm{d}x\right)\\&=O\left(\frac{n^{\frac{3}{4}}}{\log n}\right)\end{aligned}$$

对于空间复杂度，可以发现不论是 F_k 还是 F_{prime}，其均只在 n/i 处取有效点值，共 $O(\sqrt{n})$ 个，仅记录有效值即可将空间复杂度优化至 $O(\sqrt{n})$.

首先，通过一次数论分块可以得到所有的有效值，用一个大小为 $O(\sqrt{n})$ 的数组 lis 记录. 对于有效值 v，记 $\text{id}(v)$ 为 v 在 lis 中的下标，易得：对于所有有效值 v，$\text{id}(v)\le\sqrt{n}$.

然后分开考虑小于等于 $\sqrt{n}$ 的有效值和大于 $\sqrt{n}$ 的有效值：对于小于等于 $\sqrt{n}$ 的有效值 v，用一个数组 le 记录其 $\mathrm{id}(v)$，即 $\mathrm{le}_v = \mathrm{id}(v)$；对于大于 $\sqrt{n}$ 的有效值 v，用一个数组 ge 记录 $\mathrm{id}(v)$，由于 v 过大所以借助 $v' = n/v < \sqrt{n}$ 记录 $\mathrm{id}(v)$，即 $\mathrm{ge}_{v'} = \mathrm{id}(v)$.

这样，就可以使用两个大小为 $O(\sqrt{n})$ 的数组记录所有有效值的 id 并 $O(1)$ 查询. 在计算 F_k 或 F_{prime} 时，使用有效值的 id 代替有效值作为下标，即可将空间复杂度优化至 $O(\sqrt{n})$.

过程

对于 $F_k(n)$ 的计算，我们实现时一般选择实现难度较低的第一种方法，其在数据规模较小时往往比第二种方法的表现要好；

对于 $F_{\mathrm{prime}}(n)$ 的计算，直接按递推式实现即可.

对于 $p_k^2 \le n$，可以用线性筛预处理出 $s_k := F_{\mathrm{prime}}(p_k)$ 来替代 F_k 递推式中的 $F_{\mathrm{prime}}(p_{k-1})$.

相应地，G 递推式中的 $G_{k-1}(p_{k-1}) = \sum_{i=1}^{k-1} g(p_i)$ 也可以用此方法预处理.

用 Extended Eratosthenes Sieve 求 **积性函数** f 的前缀和时，应当明确以下几点：

- 如何快速（一般是线性时间复杂度）筛出前 $\sqrt{n}$ 个 f 值；
- $f(p)$ 的多项式表示；
- 如何快速求出 $f(p^c)$.

明确上述几点之后按顺序实现以下几部分即可：

1. 筛出 $[1, \sqrt{n}]$ 内的质数与前 $\sqrt{n}$ 个 f 值；
2. 对 $f(p)$ 多项式表示中的每一项筛出对应的 G，合并得到 F_{prime} 的所有 $O(\sqrt{n})$ 个有用点值；
3. 按照 F_k 的递推式实现递归，求出 $F_1(n)$.

例题

Luogu P4213【模板】杜教筛[4]

求 $\sum_{i=1}^{n} \varphi(i)$ 和 $\sum_{i=1}^{n} \mu(i)$.

解答

对于求 $\varphi(i)$ 的前缀和，首先易知 $f(p) = p - 1$. 对于 $f(p)$ 的一次项 (p)，有 $g(p) = p, G_0(n) = \sum_{i=2}^{n} g(i) = \frac{(n+2)(n-1)}{2}$；对于 $f(p)$ 的常数项 (-1)，有 $g(p) = -1, G_0(n) = \sum_{i=2}^{n} g(i) = -n + 1$. 筛两次加起来即可得到 F_{prime} 的所有 $O(\sqrt{n})$ 个所需点值.

对于求 $\mu(i)$ 的前缀和，易知 $f(p) = -1$. 则 $g(p) = -1, G_0(n) = \sum_{i=2}^{n} g(i) = -n + 1$. 直接筛即可得到 F_{prime} 的所有 $O(\sqrt{n})$ 个所需点值.

LOJ 6053 简单的函数[5]

给定 $f(n)$：

$$f(n) = \begin{cases} 1 & n = 1 \\ p \operatorname{xor} c & n = p^c \\ f(a)f(b) & n = ab \wedge a \perp b \end{cases}$$

求 $\sum_{i=1}^{n} f(i)$.

解答

易知 $f(p)=p-1+2[p=2]$. 则按照筛 φ 的方法筛，对 2 讨论一下即可.

参考代码

```cpp
/* 「LOJ #6053」简单的函数 */
#include <cmath>
#include <iostream>

constexpr int MAXS = 200000;  // 2sqrt(n)
constexpr int mod = 1000000007;

template <typename x_t, typename y_t>
void inc(x_t &x, const y_t &y) {
  x += y;
  (mod <= x) && (x -= mod);
}

template <typename x_t, typename y_t>
void dec(x_t &x, const y_t &y) {
  x -= y;
  (x < 0) && (x += mod);
}

template <typename x_t, typename y_t>
int sum(const x_t &x, const y_t &y) {
  return x + y < mod ? x + y : (x + y - mod);
}

template <typename x_t, typename y_t>
int sub(const x_t &x, const y_t &y) {
  return x < y ? x - y + mod : (x - y);
}

template <typename _Tp>
int div2(const _Tp &x) {
  return ((x & 1) ? x + mod : x) >> 1;
}

// 以上目的均为防负数和取模
template <typename _Tp>
long long sqrll(const _Tp &x) {  // 平方函数
  return (long long)x * x;
}

int pri[MAXS / 7], lpf[MAXS + 1], spri[MAXS + 1], pcnt;

void sieve(const int &n) {
  for (int i = 2; i <= n; ++i) {
    if (lpf[i] == 0) {  // 记录质数
      lpf[i] = ++pcnt;
      pri[lpf[i]] = i;
```

```
48        spri[pcnt] = sum(spri[pcnt - 1], i);  // 前缀和
49      }
50      for (int j = 1, v; j <= lpf[i] && (v = i * pri[j]) <= n; ++j) lpf[v]
= j;
51    }
52  }
53
54  long long global_n;
55  int lim;
56  int le[MAXS + 1],  // x <= \sqrt{n}
57      ge[MAXS + 1];  // x > \sqrt{n}
58  #define idx(v) (v <= lim ? le[v] : ge[global_n / v])
59
60  int G[MAXS + 1][2], Fprime[MAXS + 1];
61  long long lis[MAXS + 1];
62  int cnt;
63
64  void init(const long long &n) {
65    for (long long i = 1, j, v; i <= n; i = n / j + 1) {
66      j = n / i;
67      v = j % mod;
68      lis[++cnt] = j;
69      (j <= lim ? le[j] : ge[global_n / j]) = cnt;
70      G[cnt][0] = sub(v, 1ll);
71      G[cnt][1] = div2((long long)(v + 2ll) * (v - 1ll) % mod);
72    }
73  }
74
75  void calcFprime() {
76    for (int k = 1; k <= pcnt; ++k) {
77      const int p = pri[k];
78      const long long sqrp = sqrll(p);
79      for (int i = 1; lis[i] >= sqrp; ++i) {
80        const long long v = lis[i] / p;
81        const int id = idx(v);
82        dec(G[i][0], sub(G[id][0], k - 1));
83        dec(G[i][1], (long long)p * sub(G[id][1], spri[k - 1]) % mod);
84      }
85    }
86    /* F_prime = G_1 - G_0 */
87    for (int i = 1; i <= cnt; ++i) Fprime[i] = sub(G[i][1], G[i][0]);
88  }
89
90  int f_p(const int &p, const int &c) {
91    /* f(p^{c}) = p xor c */
92    return p ^ c;
93  }
94
95  int F(const int &k, const long long &n) {
96    if (n < pri[k] || n <= 1) return 0;
97    const int id = idx(n);
98    long long ans = Fprime[id] - (spri[k - 1] - (k - 1));
```

```
  if (k == 1) ans += 2;
  for (int i = k; i <= pcnt && sqrll(pri[i]) <= n; ++i) {
    long long pw = pri[i], pw2 = sqrll(pw);
    for (int c = 1; pw2 <= n; ++c, pw = pw2, pw2 *= pri[i])
      ans +=
          ((long long)f_p(pri[i], c) * F(i + 1, n / pw) + f_p(pri[i], c +
1)) %
          mod;
  }
  return ans % mod;
}

using std::cin;
using std::cout;

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  cin >> global_n;
  lim = sqrt(global_n);  // 上限

  sieve(lim + 1000);  // 预处理
  init(global_n);
  calcFprime();
  cout << (F(1, global_n) + 1ll + mod) % mod << '\n';

  return 0;
}
```

外部参考资料

[1] Min_25

[2] 《一些特殊的数论函数求和问题》

[3] 《积性函数求和的几种方法》

[4] Luogu P4213【模板】杜教筛

[5] LOJ 6053 简单的函数

9.11.28 洲阁筛

Authors: Early0v0

前置知识

- 积性函数→9.11.1@p1160

定义

洲阁筛是一种能在亚线性时间复杂度内求出大多数积性函数前缀和的筛法.

下面将以求解 $\sum_{i=1}^{n} f(i)$ 为例，具体阐述洲阁筛的原理.

约定

- $\mathbb{P}$ 表示质数集，p_i 表示第 i 个质数.
- m 表示 $\sqrt{n}$ 内的质数个数.

要求

当 $p \in \mathbb{P}, c \in \mathbb{N}$ 时，$f(p^c)$ 为一个关于 p 的低阶多项式.

思想

- 对于任意 $[1,n]$ 内的整数，其至多只有一个 $> \sqrt{n}$ 的质因子.
- 利用 $\left\lfloor \frac{n}{i} \right\rfloor (i \in [1,n] \cap \mathbb{N})$ 只有 $\sqrt{n}$ 级别个取值的性质来降低时间复杂度.

过程

将 $[1,n]$ 内的所有整数按是否有 $> \sqrt{n}$ 的质因子分为两类：

$$\sum_{i=1}^{n} f(i) = \sum_{i=1}^{n} \left[\exists d \in (\sqrt{n}, n] \cap \mathbb{P}, d \mid i\right] f(i) + \sum_{i=1}^{n} \left[\forall d \in (\sqrt{n}, n] \cap \mathbb{P}, d \nmid i\right] f(i)$$

对于前半部分，枚举最大因子，根据积性函数的性质可以转换：

$$\sum_{i=1}^{n} f(i) = \sum_{i=1}^{\sqrt{n}} f(i) \cdot \left(\sum_{d=\lfloor \sqrt{n} \rfloor + 1}^{\lfloor \frac{n}{i} \rfloor} [d \in \mathbb{P}] f(d) \right) + \sum_{i=1}^{n} \left[\forall d \in (\sqrt{n}, n] \cap \mathbb{P}, d \nmid i\right] f(i)$$

前后部分可以分别计算.

Part 1

> 计算 $\sum_{i=1}^{\sqrt{n}} f(i) \cdot \left(\sum_{d=\lfloor \sqrt{n} \rfloor + 1}^{\lfloor \frac{n}{i} \rfloor} [d \in \mathbb{P}] f(d) \right)$.

考虑枚举 i，然后 $O(1)$ 计算括号内部分.

记 $g(t,l) = \sum_{i=1}^{l} [\forall j \in [1,t], \gcd(i, p_j) = 1] f(i)$，即 $[1,l]$ 中与 $p_1, p_2, ..., p_t$ 均互质的数的 f 值之和.

这样 Part 1 的计算就变成了 $\sum_{i=1}^{\sqrt{n}} f(i) \cdot g\left(m, \left\lfloor \frac{n}{i} \right\rfloor\right)$.

边界 $g(0,l) = \sum_{i=1}^{l} f(i)$，转移 $g(t,l) = g(t-1,l) - f(p_t) \cdot g\left(t-1, \left\lfloor \frac{l}{p_t} \right\rfloor\right)$.

l 共有 $\sqrt{n}$ 级别种取值，对于每种取值则需要枚举其质因子，所以复杂度为 $O\left(\frac{\sqrt{n}}{\ln \sqrt{n}} \cdot \sqrt{n}\right) = O\left(\frac{n}{\log n}\right)$，需要优化.

注意到 $p_{t+1}^2 > l$ 时符合条件的数只有 1，所以此时 $g(t,l) = f(1) = 1$.

代入递推式可得：当 $p_t^2 > l$ 时，$g(t,l) = g(t-1,l) - f(p_t)$.

所以一旦发现 $p_t^2 > l$ 就停止转移，记此时的 t 为 t_l，则 $\forall t > t_l, g(t,l) = g(t_l,l) - \sum_{i=t_l}^{t-1} f(p_i)$.

预处理质数的 f 值前缀和即可快速求出 g，时间复杂度被优化至 $O\left(\frac{n^{\frac{3}{4}}}{\log n}\right)$.

Part 2

> 计算 $\sum_{i=1}^{n} \left[\forall d \in (\sqrt{n}, n] \cap \mathbb{P}, d \nmid i\right] f(i)$.

记 $h(t,l) = \sum_{i=1}^{l} \left[i = \prod_{j=t}^{m} p_j^{c_j}, c_j \in \mathbb{N} \right] f(i)$，即 $[1,l]$ 中所有只含 $p_t, p_{t+1}, ..., p_m$ 质因子的数的 f 值之和.

Part 2 即为求 $h(0,n)$.

边界 $h(m+1,l) = 1$，转移 $h(t,l) = h(t+1,l) + \sum_{c \in \mathbb{N}^*} f(p_t^c) \cdot h\left(t+1, \left\lfloor \frac{l}{p_t^c} \right\rfloor\right)$.

l 共有 $\sqrt{n}$ 级别种取值，所以直接转移复杂度为 $O\left(\sqrt{n} \cdot \frac{\sqrt{n}}{\ln \sqrt{n}}\right) = O\left(\frac{n}{\log n}\right)$，需要优化.

与 g 的优化方式类似，注意到 $p_t > l$ 时，能用 $p_t, p_{t+1}, \ldots, p_m$ 组成的数只有 1，此时的 $h(t,l) = f(1) = 1$.

类似的，推出 $\forall p_t^2 > l, h(t,l) = h(t+1,l) + f(p_t)$.

所以一旦发现 $p_t^2 > l$ 就停止转移，记此时的 t 为 t_l，之后用到 h 时，把此时的 h 值加上 $\sum_{i=p_{t_l}}^{\min(l,\sqrt{n})} [i \in \mathbb{P}] f(i)$ 即可.

时间复杂度被优化至 $O\left(\frac{n^{\frac{3}{4}}}{\log n}\right)$.

求和

算出了 Part 1 和 Part 2 的答案，将其相加即为 $\sum_{i=1}^{n} f(i)$.

参考

积性函数线性筛/杜教筛/洲阁筛学习笔记 | Bill Yang's Blog[1]

外部参考资料

[1] 积性函数线性筛/杜教筛/洲阁筛学习笔记 | Bill Yang's Blog

9.11.29 类欧几里得算法

Authors: sshwy, FFjet, qz-cqy

引入

类欧几里得算法是洪华敦在 2016 年冬令营营员交流中提出的内容. 它常用于解决形如

$$\left\lfloor \frac{ai+b}{c} \right\rfloor$$

结构的数列（下标为 i）的求和问题. 它的主要想法是，利用分数自身的递归结构，将问题转化为更小规模的问题，递归求解. 因为分数的递归结构和 欧几里得算法→9.11.4@p1188 存在直接的 联系→9.11.30@p1431，因此，这一求和方法也称为类欧几里得算法.

因为 连分数→9.11.30@p1431 和 Stern–Brocot 树→9.11.31@p1482 等方法同样刻画了分数的递归结构，所以利用类欧几里得算法可以解决的问题，通常也可以用这些方法解决. 与这些方法相比，类欧几里得算法通常更容易理解，它的实现也更为简明.

类欧几里得算法

最简单的例子，就是求和问题：

$$f(a,b,c,n) = \sum_{i=0}^{n} \left\lfloor \frac{ai+b}{c} \right\rfloor,$$

其中，a, b, c, n 都是正整数.

代数解法

首先，将 a, b 对 c 取模，可以简化问题，将问题转化为 $0 \le a, b < c$ 的情形：

$$\begin{aligned}
f(a,b,c,n) &= \sum_{i=0}^{n}\left\lfloor\frac{ai+b}{c}\right\rfloor \\
&= \sum_{i=0}^{n}\left\lfloor\frac{(\lfloor\frac{a}{c}\rfloor c+(a \bmod c))i+(\lfloor\frac{b}{c}\rfloor c+(b \bmod c))}{c}\right\rfloor \\
&= \sum_{i=0}^{n}\left(\left\lfloor\frac{a}{c}\right\rfloor i+\left\lfloor\frac{b}{c}\right\rfloor+\left\lfloor\frac{(a \bmod c)i+(b \bmod c)}{c}\right\rfloor\right) \\
&= \frac{n(n+1)}{2}\left\lfloor\frac{a}{c}\right\rfloor+(n+1)\left\lfloor\frac{b}{c}\right\rfloor+f(a \bmod c, b \bmod c, c, n).
\end{aligned}$$

现在，考虑转化后的问题．令

$$m=\left\lfloor\frac{an+b}{c}\right\rfloor.$$

那么，原问题可以写作二次求和式：

$$\sum_{i=0}^{n}\left\lfloor\frac{ai+b}{c}\right\rfloor=\sum_{i=0}^{n}\sum_{j=0}^{m-1}\left[j<\left\lfloor\frac{ai+b}{c}\right\rfloor\right].$$

交换求和次序，这需要对于每个 j 计算满足条件的 i 的范围．为此，将条件变形：

$$\begin{aligned}
&j<\left\lfloor\frac{ai+b}{c}\right\rfloor=\left\lceil\frac{ai+b+1}{c}\right\rceil-1 \\
\iff &j+1<\left\lceil\frac{ai+b+1}{c}\right\rceil\iff j+1<\frac{ai+b+1}{c} \\
\iff &\frac{cj+c-b-1}{a}<i\iff\left\lfloor\frac{cj+c-b-1}{a}\right\rfloor<i.
\end{aligned}$$

变形过程中多次利用了 取整函数→9.11.1@p1160 的性质．代入变形后的条件，原式可以写作：

$$\begin{aligned}
f(a,b,c,n) &= \sum_{j=0}^{m-1}\sum_{i=0}^{n}\left[i>\left\lfloor\frac{cj+c-b-1}{a}\right\rfloor\right] \\
&= \sum_{j=0}^{m-1}\left(n-\left\lfloor\frac{cj+c-b-1}{a}\right\rfloor\right) \\
&= nm-f(c,c-b-1,a,m-1).
\end{aligned}$$

令 $(a',b',c',n')=(c,c-b-1,a,m-1)$，这就又回到了前面讨论过的 $a'>c'$ 的情形．

将这两步转化结合在一起，可以发现在过程中，(a,c) 不断地取模后交换位置，直到 $a=0$．这就类似于对 (a,c) 进行辗转相除，这也是类欧几里得算法的得名．它的时间复杂度是 $O(\log\min\{a,c\})$ 的．

在计算过程中，可能会出现 $m=0$ 的情形，此时内层递归会出现 $n=-1$．这并不影响最终的结果．但是，如果在出现 $m=0$ 时直接终止算法，那么算法的时间复杂度可以改良为 $O(\log\min\{a,c,n\})$ 的．

对复杂度的解释

利用该算法和欧几里得算法的相似性，很容易说明它的时间复杂度是 $O(\log\min\{a,c\})$ 的．因此，只需要说明，如果在 $m=0$ 时终止算法，那么它的时间复杂度也是 $O(\log n)$ 的．

令 $m=\lfloor(an+b)/c\rfloor$，并记 $S=mn$，$k=m/n$，它们分别相当于几何直观（见下一节）中点阵图的面积和直线的斜率．对于充分大的 n，近似有 $k\doteq a/c$．

考察 S 和 k 在算法过程中的变化．第一步取模时，n 保持不变，k 近似由 a/c 变为 $(a \bmod c)/c$，相当于斜率由 k 变为 $k-\lfloor k\rfloor$，而 S 也近似变为原来的 $(k-\lfloor k\rfloor)/k$ 倍．第二步交换横纵坐标时，S 近似保持不变，k 则变为它的倒数．因此，若设两步操作后，二元组 (k,S) 变为 (k',S')，则有 $k'=(k-\lfloor k\rfloor)^{-1}$ 且 $S'=\dfrac{k-\lfloor k\rfloor}{k}S\le(k-\lfloor k\rfloor)S$，因为从第二轮开始，每轮开始时总有 $k\ge1$．

因为 $1 \le \lfloor k' \rfloor \le k' < \lfloor k' \rfloor + 1$，所以，递归计算两轮后，$S$ 至多变为原来的

$$(k' - \lfloor k' \rfloor)(k - \lfloor k \rfloor) = 1 - \frac{\lfloor k' \rfloor}{k'} < 1 - \frac{\lfloor k' \rfloor}{\lfloor k' \rfloor + 1} = \frac{1}{\lfloor k' \rfloor + 1} \le \frac{1}{2}.$$

因此，至多 $O(\log S)$ 轮，算法必然终止．因为从第二轮开始，每轮开始时的 S 总是不超过上一轮取模结束后的 S，而后者大致为 kn^2 且 $k < 1$，因而 $O(\log S) \subseteq O(\log n)$．这就得到了上述结论．

模板题的参考实现如下：

模板题实现（Library Checker - Sum of Floor of Linear[1]）

```cpp
#include <iostream>

long long solve(long long a, long long b, long long c, long long n) {
  long long n2 = n * (n + 1) / 2;
  if (a >= c || b >= c)
    return solve(a % c, b % c, c, n) + (a / c) * n2 + (b / c) * (n + 1);
  long long m = (a * n + b) / c;
  if (!m) return 0;
  return m * n - solve(c, c - b - 1, a, m - 1);
}

int main() {
  int t;
  std::cin >> t;
  for (; t; --t) {
    int a, b, c, n;
    std::cin >> n >> c >> a >> b;
    std::cout << solve(a, b, c, n - 1) << '\n';
  }
  return 0;
}

```

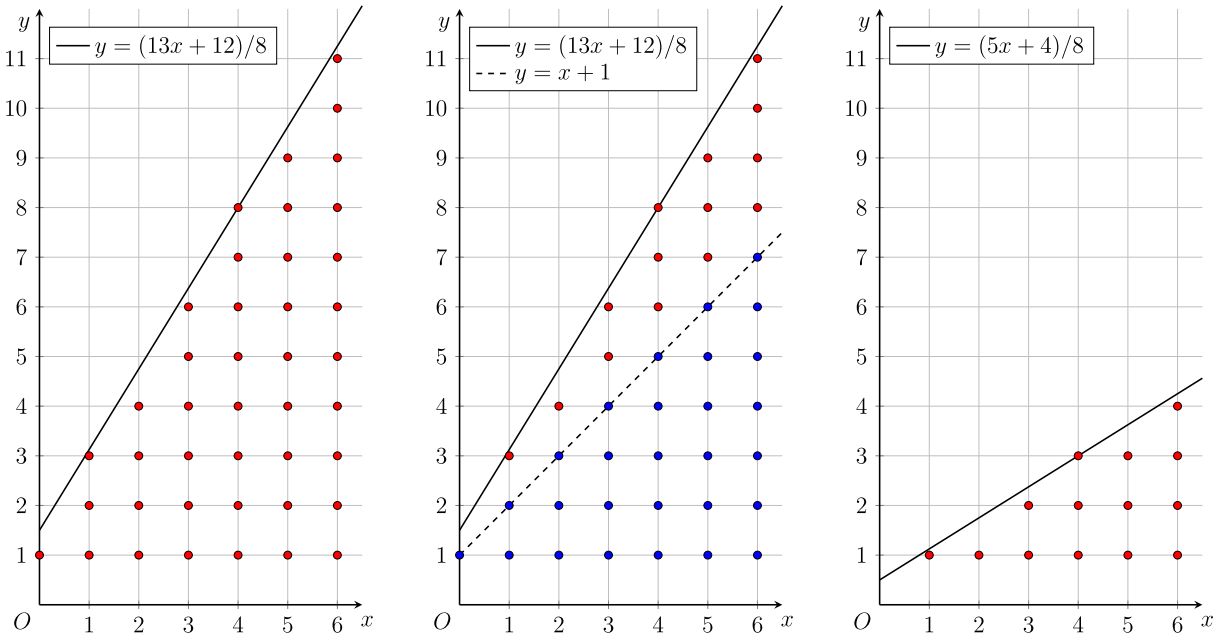
几何直观

这个算法还可以从几何的角度理解．类欧几里得算法可以解决的问题主要是直线下整点计数问题．

如下图最左部分所示，该求和式相当于求直线

$$y = \frac{ax + b}{c}$$

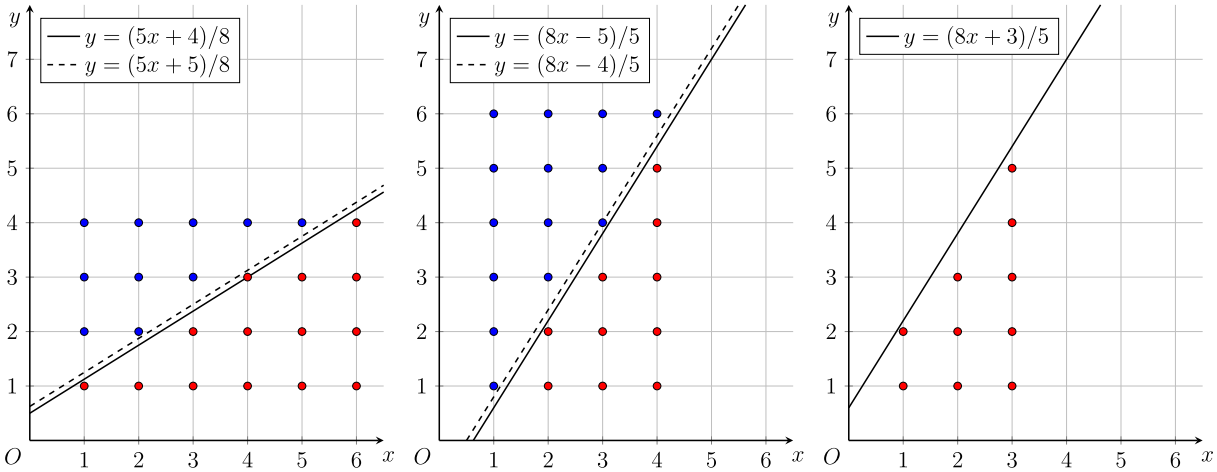
下方，x 轴上方（不包括 x 轴），且横坐标位于 $[0, n]$ 之间的格点数目．



首先，移除斜率和截距中的整数部分．这一步相当于将上图中间部分的蓝点数量单独计算出来．当斜率和截距都是整数时，蓝点一定构成一个梯形阵列，也就是说，不同纵列的格点形成等差数列，因而这些点的数量是容易计算的．将这些点移除后，剩余的格点和上图最右部分的红点数量一致．问题就转化成了斜率和截距都小于一的情形．因为梯形的高为 $n+1$ 且两个底边长度分别为 $\lfloor b/c \rfloor$ 和 $(\lfloor a/c \rfloor n + \lfloor b/c \rfloor)$，所以，利用梯形面积公式，这一步骤可以归纳为算式

$$f(a,b,c,n) = f(a \bmod c, b \bmod c, c, n) + \frac{1}{2}(n+1)\left(\left\lfloor \frac{b}{c} \right\rfloor + \left(\left\lfloor \frac{a}{c} \right\rfloor n + \left\lfloor \frac{b}{c} \right\rfloor\right)\right).$$

然后，翻转横纵坐标轴．如下图最左部分所示，图中的红点和蓝点构成了一个横向长度为 n、纵向长度为 $m = \lfloor (an+b)/c \rfloor$ 的矩形点阵．要计算红点的数量，只需要计算蓝点的数量，再用矩形点阵的数量减去蓝点的数量即可．翻转后，上图左半部分中的蓝点点阵就变成了某条直线下的红色点阵．而且，翻转后，斜率大于一，就又回到了上文已经处理过的情形．



关键在于如何计算新的红色点阵上方的直线的方程．将上图最左部分的横纵坐标轴翻转，就得到上图中间部分．翻转后的红色点阵上方的直线（中间部分的实线），并非对应翻转前的直线（最左部分的实线），而是翻转前的直线向左上平移一点点的结果（最左部分的虚线）．这是因为，如果直接将直线（最左部分的实线）翻转，将得到中间部分的虚线，但是按照定义，它下方的格点包括恰好落在直线上的格点，这就会导致直线上的格点重复计数．为了避免这一点，需要将翻转直线 $y = (ax+b)/c$ 后得到的直线 $y = (cx - b)/a$ 向下平移一点点，得到直线 $y = (cx - b - 1)/a$，这样它下方的点阵才恰为翻转前的蓝色点阵．

还有另一处细节需要处理．上图中间部分的直线的截距是负数，这意味着还没有回到之前的初始情形．要让截距恢复为非负数，只需要将直线（中间部分的实线）向左平移一个单位．这样做不会漏掉任何格点，因为翻转前的蓝色点阵中没有纵坐标为零的点，翻转后也就不存在横坐标为零的点．最后，直线方程就变为 $y=(cx+c-b-1)/a$；同时，点阵的横坐标的上界也从 m 变成了 $m-1$．这一步骤可以归纳为算式

$$f(a,b,c,n)=mn-f(c,c-b-1,a,m-1).$$

这种递归的算法行得通，主要有两个原因：

- 第一，直线的斜率不断地先取小数部分再取倒数，这等价于计算直线斜率 $k=a/c$ 的 连分数展开→9.11.30@p1431．因为有理分数的连分数展开的长度是 $O(\log\min\{a,c\})$ 的，所以这一过程一定在 $O(\log\min\{a,c\})$ 步后终止；
- 第二，因为每次翻转坐标轴的时候，直线斜率都是小于一的，因此，直觉上应该有 $m<n$，也就是说，经过这样一轮迭代后，横坐标的范围一直是在缩小的．前文的复杂度计算中通过严格的分析说明，每两轮迭代后，$S=mn$ 至多为原来的一半，故而这一过程一定在 $O(\log n)$ 步后终止．

这也是斜率为有理数时的类欧几里得算法的复杂度是 $O(\log\min\{a,c,n\})$ 的原因．

利用类似的几何直观，可以将类欧几里得算法推广到斜率为无理数的情形，具体分析请参考后文的例题．

例题

【模板】类欧几里得算法[2]

多组询问．给定正整数 a,b,c,n，求

$$f(a,b,c,n)=\sum_{i=0}^{n}\left\lfloor\frac{ai+b}{c}\right\rfloor,$$

$$g(a,b,c,n)=\sum_{i=0}^{n}i\left\lfloor\frac{ai+b}{c}\right\rfloor,$$

$$h(a,b,c,n)=\sum_{i=0}^{n}\left\lfloor\frac{ai+b}{c}\right\rfloor^2.$$

解答一

类似于 f 的推导，可以得到 g,h 的递归表达式．

首先，利用取模，将问题转化为 $0\le a,b<c$ 的情形：

$$g(a,b,c,n)=g(a\bmod c,b\bmod c,c,n)+\left\lfloor\frac{a}{c}\right\rfloor\frac{n(n+1)(2n+1)}{6}+\left\lfloor\frac{b}{c}\right\rfloor\frac{n(n+1)}{2},$$

$$\begin{aligned}h(a,b,c,n)=&\,h(a\bmod c,b\bmod c,c,n)\\&+2\left\lfloor\frac{b}{c}\right\rfloor f(a\bmod c,b\bmod c,c,n)+2\left\lfloor\frac{a}{c}\right\rfloor g(a\bmod c,b\bmod c,c,n)\\&+\left\lfloor\frac{a}{c}\right\rfloor^2\frac{n(n+1)(2n+1)}{6}+\left\lfloor\frac{b}{c}\right\rfloor^2(n+1)+\left\lfloor\frac{a}{c}\right\rfloor\left\lfloor\frac{b}{c}\right\rfloor n(n+1).\end{aligned}$$

然后，利用交换求和次序，可以进一步转化．同样地，令

$$m=\left\lfloor\frac{an+b}{c}\right\rfloor.$$

那么，对于和式 g，有

$$
\begin{aligned}
g(a,b,c,n) &= \sum_{i=0}^{n} i\left\lfloor \frac{ai+b}{c} \right\rfloor \\
&= \sum_{i=0}^{n}\sum_{j=0}^{m-1} i\left[j < \left\lfloor \frac{ai+b}{c} \right\rfloor \right] \\
&= \sum_{j=0}^{m-1}\sum_{i=0}^{n} i\left[i > \left\lfloor \frac{cj+c-b-1}{a} \right\rfloor \right] \\
&= \sum_{j=0}^{m-1} \frac{1}{2}\left(\left\lfloor \frac{cj+c-b-1}{a} \right\rfloor + n + 1 \right)\left(n - \left\lfloor \frac{cj+c-b-1}{a} \right\rfloor \right) \\
&= \frac{1}{2}mn(n+1) - \frac{1}{2}\sum_{j=0}^{m-1}\left\lfloor \frac{cj+c-b-1}{a} \right\rfloor - \frac{1}{2}\sum_{j=0}^{m-1}\left\lfloor \frac{cj+c-b-1}{a} \right\rfloor^2 \\
&= \frac{1}{2}mn(n+1) - \frac{1}{2}f(c,c-b-1,a,m-1) - \frac{1}{2}h(c,c-b-1,a,m-1).
\end{aligned}
$$

对于和式 h，有

$$
\begin{aligned}
h(a,b,c,n) &= \sum_{i=0}^{n}\left\lfloor \frac{ai+b}{c} \right\rfloor^2 \\
&= \sum_{i=0}^{n}\sum_{j=0}^{m-1}(2j+1)\left[j < \left\lfloor \frac{ai+b}{c} \right\rfloor \right] \\
&= \sum_{j=0}^{m-1}\sum_{i=0}^{n}(2j+1)\left[i > \left\lfloor \frac{cj+c-b-1}{a} \right\rfloor \right] \\
&= \sum_{j=0}^{m-1}(2j+1)\left(n - \left\lfloor \frac{cj+c-b-1}{a} \right\rfloor \right) \\
&= nm^2 - \sum_{j=0}^{m-1}\left\lfloor \frac{cj+c-b-1}{a} \right\rfloor - 2\sum_{j=0}^{m-1} j\left\lfloor \frac{cj+c-b-1}{a} \right\rfloor \\
&= nm^2 - f(c,c-b-1,a,m-1) - 2g(c,c-b-1,a,m-1).
\end{aligned}
$$

从几何直观的角度看，这些非线性的求和式相当于给区域中的每个点 (i,j) 都赋予了相应的权重 $w(i,j)$．除了这些权重之外，其余部分的计算过程是完全一致的．对于权重的选择，一般地，有

$$
\sum_{i=0}^{n} i^r\left\lfloor \frac{ai+b}{c} \right\rfloor^s = \sum_{i=0}^{n}\sum_{j=0}^{m-1} i^r\left((j+1)^s - j^s\right)\left[j < \left\lfloor \frac{ai+b}{c} \right\rfloor \right].
$$

本题的另一个特点是，g 和 h 在递归计算时，会相互交错．因此，需要将 (f,g,h) 作为三元组同时递归．

```cpp
#include <iostream>

struct Data {
  int f, g, h;
};

Data solve(long long a, long long b, long long c, long long n) {
  constexpr long long M = 998244353;
  constexpr long long i2 = (M + 1) / 2;
  constexpr long long i6 = (M + 1) / 6;
  long long n2 = (n + 1) * n % M * i2 % M;
  long long n3 = (2 * n + 1) * (n + 1) % M * n % M * i6 % M;
```

```
  Data res = {0, 0, 0};
  if (a >= c || b >= c) {
    auto tmp = solve(a % c, b % c, c, n);
    long long aa = a / c, bb = b / c;
    res.f = (tmp.f + aa * n2 + bb * (n + 1)) % M;
    res.g = (tmp.g + aa * n3 + bb * n2) % M;
    res.h = (tmp.h + 2 * bb * tmp.f % M + 2 * aa * tmp.g % M +
             aa * aa % M * n3 % M + bb * bb % M * (n + 1) % M +
             2 * aa * bb % M * n2 % M) %
            M;
    return res;
  }
  long long m = (a * n + b) / c;
  if (!m) return res;
  auto tmp = solve(c, c - b - 1, a, m - 1);
  res.f = (m * n - tmp.f + M) % M;
  res.g = (m * n2 + (M - tmp.f) * i2 + (M - tmp.h) * i2) % M;
  res.h = (n * m % M * m - tmp.f - tmp.g * 2 + 3 * M) % M;
  return res;
}

int main() {
  int t;
  std::cin >> t;
  for (; t; --t) {
    int n, a, b, c;
    std::cin >> n >> a >> b >> c;
    auto res = solve(a, b, c, n);
    std::cout << res.f << ' ' << res.h << ' ' << res.g << '\n';
  }
  return 0;
}

```

[清华集训 2014] Sum[3]

多组询问．给定正整数 n 和 r，求

$$\sum_{d=1}^{n}(-1)^{\lfloor d\sqrt{r}\rfloor}.$$

解答一

如果 r 是完全平方数，那么当 $\sqrt{r}$ 为偶数时，和式为 n；否则，和式依据 n 奇偶性不同，在 0 和 -1 之间交替变化．下面考虑 r 不是完全平方数的情形．

为了应用类欧几里得算法，首先将求和式转化为熟悉的形式：

$$\begin{aligned}\sum_{d=1}^{n}(-1)^{\lfloor d\sqrt{r}\rfloor} &= \sum_{d=1}^{n}\left(1-2\left(\lfloor d\sqrt{r}\rfloor \bmod 2\right)\right)\\ &= n-2\sum_{d=1}^{n}\left(\lfloor d\sqrt{r}\rfloor-2\left\lfloor\frac{\lfloor d\sqrt{r}\rfloor}{2}\right\rfloor\right)\\ &= n-2\sum_{d=1}^{n}\lfloor d\sqrt{r}\rfloor+4\sum_{d=1}^{n}\left\lfloor\frac{d\sqrt{r}}{2}\right\rfloor\\ &= n-2f(1,0,1,n)+4f(1,0,2,n).\end{aligned}$$

其中的函数 f 具有形式

$$f(a,b,c,n)=\sum_{i=1}^{n}\left\lfloor\frac{a\sqrt{r}+b}{c}i\right\rfloor.$$

与正文中的算法不同的是，此处的斜率不再是有理数．设斜率

$$k=\frac{a\sqrt{r}+b}{c}.$$

同样分为两种情形讨论．如果 $k\geq 1$，那么

$$\begin{aligned}f(a,b,c,n)&=\sum_{i=1}^{n}\lfloor ki\rfloor=\sum_{i=1}^{n}\lfloor(k-\lfloor k\rfloor)i\rfloor+\lfloor k\rfloor\sum_{i=1}^{n}i\\ &=\lfloor k\rfloor\frac{n(n+1)}{2}+f(a,b-c\lfloor k\rfloor,c,n).\end{aligned}$$

问题转化为斜率小于一的情形．如果 $k<1$，那么设 $m=\lfloor nk\rfloor$，有

$$\begin{aligned}f(a,b,c,n)&=\sum_{i=1}^{n}\lfloor ki\rfloor=\sum_{i=1}^{n}\sum_{j=1}^{m}[j\leq\lfloor ki\rfloor]\\ &=\sum_{j=1}^{m}\sum_{i=1}^{n}[i>\lfloor k^{-1}j\rfloor]=nm-\sum_{j=1}^{m}\sum_{i=1}^{n}[i\leq\lfloor k^{-1}j\rfloor].\end{aligned}$$

此处的推导中，交换 i 和 j 的条件比正文中的情形更为简单，是因为直线 $y=kx$ 上没有除了原点之外的格点．关键在于交换后的求和式写成 $f(a,b,c,n)$ 的形式，这相当于要求 a',b',c' 满足

$$k^{-1}=\frac{a'\sqrt{r}+b'}{c'}.$$

这并不困难，只需要将分母有理化，就能得到

$$k^{-1}=\frac{c}{a\sqrt{r}+b}=\frac{ca\sqrt{r}-cb}{a^2r-b^2}.$$

因此，有

$$a'=ca,\ b'=-cb,\ c'=a^2r-b^2.$$

这说明

$$f(a,b,c,n)=nm-f(ca,-cb,a^2r-b^2,m).$$

为了避免整数溢出，需要每次都将 a,b,c 同除以它们的最大公约数．因为这个计算过程和计算 k 的连分数的过程完全一致，所以根据 连分数理论→9.11.30@p1431，只要保证 $\gcd(a,b,c)=1$，它们在计算过程中必然在整型范围内．另外，尽管 (a,b,c,n) 不会溢出，但是在该题数据范围下，$f(a,b,c,n)$ 可能会超过 64 位整数的范围，自然溢出即可，无需额外处理，最后结果一定在 $[-n,n]$ 之间．

尽管斜率不会变为零，算法的复杂度仍然是 $O(\log n)$ 的，这一点从前文关于算法复杂度的论证容易看出．

```cpp
#include <cmath>
#include <iostream>
```

```cpp

long long r;
long double sqrt_r;

long long gcd(long long a, long long b) { return b ? gcd(b, a % b) : a; }

unsigned long long f(long long a, long long b, long long c, long long n) {
  if (!n) return 0;
  auto d = gcd(a, gcd(b, c));
  a /= d;
  b /= d;
  c /= d;
  unsigned long long k = (a * sqrt_r + b) / c;
  if (k) {
    return n * (n + 1) / 2 * k + f(a, b - c * k, c, n);
  } else {
    unsigned long long m = n * (a * sqrt_r + b) / c;
    return n * m - f(c * a, -c * b, a * a * r - b * b, m);
  }
}

unsigned long long solve(long long n, long long r) {
  long long sqr = sqrt_r = sqrtl(r);
  if (r == sqr * sqr) return r % 2 ? (n % 2 ? -1 : 0) : n;
  return n - 2 * f(1, 0, 1, n) + 4 * f(1, 0, 2, n);
}

int main() {
  int t;
  std::cin >> t;
  for (; t; --t) {
    int n;
    std::cin >> n >> r;
    long long res = solve(n, r);
    std::cout << res << '\n';
  }
  return 0;
}

```

Fraction[4]

给定正整数 a, b, c, d，求所有满足 $a/b < p/q < c/d$ 的最简分数 p/q 中 (q, p) 的字典序最小的那个.

解答

这道题目也是 Stern–Brocot 树→9.11.31@p1482 的经典应用，相关题解可以在 此处→9.11.30@p1431 找到. 因为它只依赖于分数的递归结构，所以它同样可以利用类似欧几里得算法的方法求解，故而也可以视作类欧几里得算法的一个应用.

如果 a/b 和 c/d 之间（不含端点）存在至少一个自然数，可以直接取 $(q, p) = (1, \lfloor a/b \rfloor + 1)$. 否则，必然有

$$\left\lfloor \frac{a}{b} \right\rfloor \le \frac{a}{b} < \frac{p}{q} < \frac{c}{d} \le \left\lfloor \frac{a}{b} \right\rfloor + 1.$$

从这个不等式中可以看出，p/q 的整数部分可以确定为 $\lfloor a/b \rfloor$，直接消去该整数部分，然后整体取倒数，用于确定它的小数部分．这正是确定 p/q 的连分数的 基本方法→9.11.30@p1431．若最终的答案是 p/q，那么算法的时间复杂度为 $O(\log \min\{p, q\})$．

此处，有一个细节问题，即取倒数之后得到的字典序最小的分数，是否是取倒数之前的字典序最小的分数．换句话说，满足 $a/b < p/q < c/d$ 的分数 p/q 中，字典序 (q, p) 最小的，是否也是字典序 (p, q) 最小的．假设不然，设 p/q 是字典序 (q, p) 最小的，但是 $r/s \neq p/q$ 是字典序 (r, s) 最小的．这必然有 $r < p$ 且 $q < s$．但是，这说明

$$\frac{a}{b} < \frac{r}{s} < \frac{r}{q} < \frac{p}{q} < \frac{c}{d}.$$

因此，r/q 无论按照哪个字典序都严格小于当前解. 这与所设条件矛盾. 因此，上述算法是正确的.

```cpp
#include <iostream>

void solve(int a, int b, int& p, int& q, int c, int d) {
  if ((a / b + 1) * d < c) {
    p = a / b + 1;
    q = 1;
  } else {
    solve(d, c - d * (a / b), q, p, b, a % b);
    p += q * (a / b);
  }
}

int main() {
  int a, b, c, d, p, q;
  while (std::cin >> a >> b >> c >> d) {
    solve(a, b, p, q, c, d);
    std::cout << p << '/' << q << '\n';
  }
  return 0;
}

```

万能欧几里得算法

上一节讨论的类欧几里得算法推导通常较为繁琐，而且能够解决的和式主要是可以转化为直线下（带权）整点计数问题的和式．本节讨论一种更为一般的方法，它进一步抽象了上述过程，从而可以解决更多的问题．因此，这一方法也称为万能欧几里得算法．它同样利用了分数的递归结构求解问题，但是与类欧几里得算法约化问题的思路稍有不同.

仍然考虑最经典的求和问题：

$$f(a, b, c, n) = \sum_{i=1}^{n} \left\lfloor \frac{ai + b}{c} \right\rfloor,$$

其中，a, b, c, n 都是正整数.

问题转化

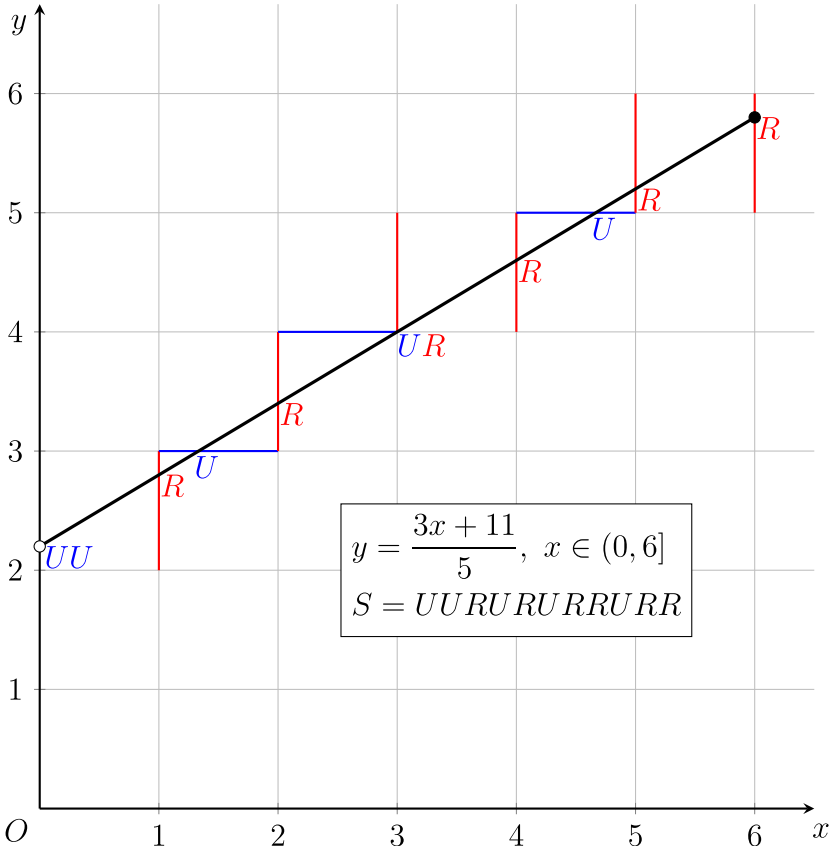
设参数为 (a, b, c, n) 的线段为

$$y = \frac{ax + b}{c},\ 0 < x \le n.$$

对于这条线段，可以按照如下方法定义一个由 U 和 R 组成的字符串 S，也称为 **操作序列**：

- 字符串恰由 n 个 R 和 $m = \lfloor (an + b)/c \rfloor$ 个 U 组成；
- 第 i 个 R 前方的 U 的数量恰等于 $\lfloor (ai + b)/c \rfloor$，其中，$i = 1, \cdots, n$.

从几何直观上看，这大致相当于从原点开始，每向右穿过一次竖向的网格线，就写下一个 R，每向上穿过一次横向的网格线，就写下一个 U. 如下图所示：



当然，这样的定义还需要考量一系列特殊情形：

- 经过整点（即同时上穿和右穿）时，需要先写 U 再写 R；
- 字符串开始时，除了在 $(0, 1]$ 区间内上穿网格线的次数外，还需要额外补充 $\lfloor b/c \rfloor$ 个 U；
- 字符串结束时，不能有额外的 U.

如果对于几何直观的描述有任何不明晰的地方，可以参考上述代数方法的定义辅助理解. 几何直观的描述，有助于理解下文的算法过程.

万能欧几里得算法的基本思路，就是将操作序列中的 U 和 R 都视作某个 幺半群→9.16.1@p1805 内的元素，将整个操作序列视为幺半群内元素的乘积，而问题最终的答案与这个乘积有关.

比如，本题中，可以定义状态向量 $v = (1, y, \sum y)$，表示自原点开始，经历了若干次上穿和右穿网格线后，当前的状态. 其中，第一个分量是常数，第二个分量是纵坐标 y，第三个分量是要求的和式. 起始时，有 $v = (1, 0, 0)$. 每向上穿过一次网格线，纵坐标就累加一，即相当于将状态向量右乘以矩阵

$$U = \begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

每向右穿过一次网格线，和式就累加一次纵坐标，即相当于将状态向量右乘以矩阵

$$R=\begin{pmatrix}1&0&0\\0&1&1\\0&0&1\end{pmatrix}.$$

因此，最终的状态就是乘积 $(1,0,0)S$，其中，S 理解为上述矩阵的乘积．所求的答案，就是最终状态的第三个分量．

除了将幺半群中的元素定义为矩阵以外，还可以将它们定义为一段操作序列对于最终结果的贡献，然后将操作的乘积定义为两段操作序列的贡献的合并．

本题中，可以定义每段操作序列的贡献为 $(x,y,\sum y)$．为了严谨地解释这些记号，可以将这些分量都看作是操作序列的函数，也就是说，对于操作序列 S，它的贡献可以写作 $(x(S),y(S),(\sum y)(S))$．其中，$x(S)$ 和 $y(S)$ 分别对应着操作序列 S 中 R 和 U 的数量，也就是该线段右穿和上穿网格线的次数．最后一项中的求和符号，一般地，有如下定义：对于操作序列上的函数 $f(S)$，可以定义 $(\sum f)(S)$，或记作 $\sum_S f$，为下面的表达式：

$$\sum_S f:=\sum\{f(S_{[1,r]}):S_r=R\}.$$

其中，S_r 是 S 中的第 r 个字符，$S_{[1,r]}$ 是 S 中前 r 个字符组成的前缀．也就是说，这个求和记号，可以看作是对于操作序列 S 中所有以 R 结尾的前缀进行的求和．比如说，有

$$\sum_S 1=x,\ \sum_S x=\frac{1}{2}x(x+1).$$

再比如说，$\sum y$ 就是操作序列中，每次右穿网格线时，之前上穿网格线的次数的累加．对于整段操作序列来说，y 在所有以 R 结尾的前缀处的值，就是在 $i=1,\cdots,n$ 处的所有 $\lfloor(ai+b)/c\rfloor$ 的值．因此，对于整段操作序列计算的 $\sum y$，就是本题最终要求的量．

两个操作分别为 $U=(0,1,0)$ 和 $R=(1,0,0)$．进一步，可以将两个元素 (x_1,y_1,s_1) 和 (x_2,y_2,s_2) 的乘积定义为

$$(x_1,y_1,s_1)\cdot(x_2,y_2,s_2)=(x_1+x_2,y_1+y_2,s_1+s_2+x_2y_1).$$

其中，最后一项贡献合并的结果可以通过如下计算得到：

$$\sum_{S_1+S_2}y=\sum_{S_1}y+\sum_{S_2}(y+y_1)=\sum_{S_1}y+\sum_{S_2}y+y_1\sum_{S_2}1=s_1+s_2+x_2y_1.$$

容易验证，这个乘法运算满足结合律，且单位元为 $(0,0,0)$，所以这些元素在该乘法运算下构成幺半群．所求的答案，就是乘积的第三个分量．

这两种方法都可以得到正确的结果．但是，因为保留了较多的冗余信息，矩阵运算的常数较大，所以第二种方法在处理实际问题时更为实用．

算法过程

与类欧几里得算法整体约化不同，万能欧几里得算法约化问题的手段是将这些操作分批次地合并．记字符串对应的操作的乘积为

$$F(a,b,c,n,U,R).$$

约化过程具体如下：

- 当 $b\geq c$ 时，操作序列的开始有 $\lfloor b/c\rfloor$ 个 U，直接计算它们的乘积，并将这些 U 从操作序列中移除．此时，第 i 个 R 前方的 U 的数量等于

  $$\left\lfloor\frac{ai+b}{c}\right\rfloor-\left\lfloor\frac{b}{c}\right\rfloor=\left\lfloor\frac{ai+(b\bmod c)}{c}\right\rfloor.$$

  因此，这相当于将线段参数由 (a,b,c,n) 变为 $(a,b\bmod c,c,n)$．所以，对于这种情形，有

  $F(a,b,c,n,U,R)=U^{\lfloor b/c\rfloor}F(a,b\bmod c,c,n,U,R)$.
- 当 $a\geq c$ 时，操作序列中每个 R 的前方都至少有 $\lfloor a/c\rfloor$ 个 U，可以将它们合并到 R 上．也就是说，可以用 $U^{\lfloor a/c\rfloor}R$ 替代 R．合并后的字符串中，第 i 个 R 前方的 U 的数量等于

$$\left\lfloor \frac{ai+b}{c} \right\rfloor - \left\lfloor \frac{a}{c} \right\rfloor i = \left\lfloor \frac{(a \bmod c)i+b}{c} \right\rfloor.$$

因此，这相当于将线段参数由 (a,b,c,n) 变为 $(a \bmod c, b, c, n)$．所以，对于这种情形，有 $F(a,b,c,n,U,R) = F(a \bmod c, b, c, n, U, U^{\lfloor a/c \rfloor} R)$.

- 对于剩下的情形，需要翻转横纵坐标，这基本是在交换 U 和 R，只是翻转后线段的参数需要仔细计算．结合操作序列的定义可知，需要确定系数 (a', b', c', n') 使得变换前的操作序列中，第 j 个 U 前方的 R 的数量恰为 $\lfloor (a'j+b')/c' \rfloor$ 且总共有 n' 个 U．根据定义可知，

  $$n' = \left\lfloor \frac{an+b}{c} \right\rfloor = m,$$

  而第 j 个 U 前方的 R 的数量，就等于最大的 i 使得

  $$\left\lfloor \frac{ai+b}{c} \right\rfloor < j \iff \frac{ai+b}{c} < j \iff i < \frac{cj-b}{a}$$

  $$\iff i < \left\lceil \frac{cj-b}{a} \right\rceil = \left\lfloor \frac{cj-b-1}{a} \right\rfloor + 1.$$

  因此，$i = \lfloor (cj-b-1)/a \rfloor$．这一推导过程与前文类欧几里得算法的推导类似，同样利用了上下取整函数的性质．

  有两处细节需要处理：

  - 截距项 $-(b+1)/a$ 为负数．注意到，如果将线段向左平移一个单位，就可以让截距项恢复为非负数，因为总有 $(c-b-1)/a \geq 0$．因此，可以将交换前的第一段 $R^{\lfloor (c-b-1)/a \rfloor} U$ 提取出来，只交换剩余操作序列中的 U 和 R；
  - 交换 U 和 R 后，结尾存在多余的 U．因此，交换 U 和 R 之前，需要首先将最后一段 R 提取出来，只交换剩余操作序列中的 U 和 R．这一段 R 的数量为 $n - \lfloor (cm-b-1)/a \rfloor$．

  去掉头尾若干个字符后，第 j 个 U 前方的 R 的数量变为：

  $$\left\lfloor \frac{c(j+1)-b-1}{a} \right\rfloor - \left\lfloor \frac{c-b-1}{a} \right\rfloor = \left\lfloor \frac{cj + (c-b-1) \bmod a}{a} \right\rfloor.$$

  回忆起，交换前的序列中 U 的数量为 $m = \lfloor (an+b)/c \rfloor$．而上述左移一个单位的操作，要求保证交换前至少存在一个 U，也就是 $m > 0$．利用这一条件，可以分为两种情形：

  - 对于 $m > 0$ 的情形，处理了上面的两点后，交换完 U 和 R 的操作序列就是对应着参数为 $(c, (c-b-1) \bmod a, a, m-1)$ 的线段的合法序列．所以，有 $F(a,b,c,n,U,R) = R^{\lfloor (c-b-1)/a \rfloor} U F(c, (c-b-1) \bmod a, a, m-1, R, U) R^{n - \lfloor (cm-b-1)/a \rfloor}$.
  - 特别地，对于 $m = 0$ 的情形，交换前的操作序列中只包含 n 个 R，无需交换，可以直接返回：$F(a,b,c,n,U,R) = R^n$.

    与类欧几里得算法不同，万能欧几里得算法的这一特殊情形需要单独处理，否则会因涉及负幂次而无法正确计算．

  利用这些讨论，就可以将问题递归地解决．

假设幺半群内元素单次相乘的时间复杂度是 $O(1)$ 的．那么，如果计算过程中这些元素的幂次计算都使用 快速幂→9.7@p1131 进行，最终的算法复杂度就是 $O(\log \max\{a, c\} + \log(b/c))$ 的 [58].

[58] 通常考虑的问题中，b 都与 a 同阶，$O(\log(b/c))$ 这一项可以忽略．而且，如果在调用万能欧几里得算法前，首先进行了一轮类欧几里得算法的取模，消除 b 的影响，这一项的快速幂的复杂度是可以规避的．这其实是因为通常的问题中，U 的初始形式较为特殊，它的幂次有着更简单的形式，不需要通过快速幂计算．比如正文的例子中，$U^{\lfloor b/c \rfloor}$ 的结果，就是将 U 中不在对角线上的那个 1 替换成 $\lfloor b/c \rfloor$，而无需用快速幂计算．

对复杂度的解释

对比(类)欧几里得算法, 万能欧几里得算法只是多了求快速幂的步骤. 其余的计算过程的复杂度和类欧几里得算法相仿, 已经说明是 $O(\log \min\{a, c, n\})$ 的. 现在, 需要计算这些快速幂的总复杂度.

除了第一轮迭代, 都有 $b < c$, 因此这些迭代每轮都涉及三次快速幂的计算, 总的复杂度是:

$$O\left(\log\left\lfloor\frac{a}{c}\right\rfloor + \log\left\lfloor\frac{c-b_1-1}{a_1}\right\rfloor + \log\left(n - \left\lfloor\frac{cm-b_1-1}{a_1}\right\rfloor\right)\right),$$

其中, $a_1 = a \bmod c$, $b_1 = b \bmod c$ 且 $m = \lfloor (a_1 n + b_1)/c \rfloor$. 后面两项, 分别有估计:

$$\frac{c-b_1-1}{a_1} \le \frac{c}{a_1},$$

$$\begin{aligned} n - \left\lfloor\frac{cm-b_1-1}{a_1}\right\rfloor &\le n - \frac{cm-b_1-1}{a_1} + 1 \\ &\le n - \frac{c((a_1 n + b_1)/c - 1) - b_1 - 1}{a_1} + 1 \\ &= \frac{c+1}{a_1} + 1. \end{aligned}$$

因此, 这两项的复杂度都是 $O(\log(c/a_1))$ 的.

每一轮迭代中, 线段的参数都由 $(a, \cdot, c, \cdot)$ 变换为 $(c, \cdot, a \bmod c, \cdot)$, 且该轮总的时间复杂度为

$$O\left(\log\frac{a}{c} + \log\frac{c}{a \bmod c}\right).$$

对于全部递归的轮次, 这些项可以裂项相消, 因此, 最后总和复杂度就是 $O(\log a + \log c) = O(\log \max\{a, c\})$ 的.

最后, 再加上第一轮迭代中快速幂 $U^{\lfloor b/c \rfloor}$ 的复杂度 $O(\log(b/c))$, 就得到总的复杂度为 $O(\log \max\{a, c\} + \log(b/c))$.

万能欧几里得算法的流程可以写成统一的模板, 处理具体问题时只需要更改模板类型 `T` 的实现即可.

参考实现

```cpp
// Class T implements the monoid.
// Assume that it provides a multiplication operator
//     and a default constructor returning the unity in the monoid.

// Binary exponentiation.
template <typename T>
T pow(T a, int b) {
  T res;
  for (; b; b >>= 1) {
    if (b & 1) res = res * a;
    a = a * a;
  }
  return res;
}

// Universal Euclidean algorithm.
template <typename T>
T euclid(int a, int b, int c, int n, T U, T R) {
  if (b >= c) return pow(U, b / c) * euclid(a, b % c, c, n, U, R);
```

```
  if (a >= c) return euclid(a % c, b, c, n, U, pow(U, a / c) * R);
  auto m = ((long long)a * n + b) / c;
  if (!m) return pow(R, n);
  return pow(R, (c - b - 1) / a) * U *
         euclid(c, (c - b - 1) % a, a, m - 1, R, U) *
         pow(R, n - (c * m - b - 1) / a);
}

```

利用万能欧几里得算法可以得到模板题的实现如下：

模板题实现（Library Checker - Sum of Floor of Linear[5]）

```
#include <array>
#include <iostream>

// Switch between matrix and info merging approaches.
#define MATRIX 1

// Class T implements the monoid.
// Assume that it provides a multiplication operator
//     and a default constructor returning the unity in the monoid.

// Binary exponentiation.
template <typename T>
T pow(T a, int b) {
  T res;
  for (; b; b >>= 1) {
    if (b & 1) res = res * a;
    a = a * a;
  }
  return res;
}

// Universal Euclidean algorithm.
template <typename T>
T euclid(int a, int b, int c, int n, T U, T R) {
  if (b >= c) return pow(U, b / c) * euclid(a, b % c, c, n, U, R);
  if (a >= c) return euclid(a % c, b, c, n, U, pow(U, a / c) * R);
  auto m = ((long long)a * n + b) / c;
  if (!m) return pow(R, n);
  return pow(R, (c - b - 1) / a) * U *
         euclid(c, (c - b - 1) % a, a, m - 1, R, U) *
         pow(R, n - (c * m - b - 1) / a);
}

#if MATRIX

template <size_t N>
struct Matrix {
  std::array<long long, N * N> mat;

```

```
  auto loc(size_t i, size_t j) const { return mat[i * N + j]; }

  auto& loc(size_t i, size_t j) { return mat[i * N + j]; }

  Matrix() : mat{} {
    for (size_t i = 0; i != N; ++i) {
      loc(i, i) = 1;
    }
  }

  Matrix operator*(const Matrix& rhs) const {
    Matrix res;
    res.mat.fill(0);
    for (size_t i = 0; i != N; ++i) {
      for (size_t k = 0; k != N; ++k) {
        for (size_t j = 0; j != N; ++j) {
          res.loc(i, j) += loc(i, k) * rhs.loc(k, j);
        }
      }
    }
    return res;
  }
};

long long solve(int a, int b, int c, int n) {
  if (!n) return 0;
  Matrix<3> U, R;
  U.loc(0, 1) = R.loc(1, 2) = 1;
  auto res = euclid(a, b, c, n, U, R);
  return res.loc(0, 2);
}

#else

struct Info {
  long long x, y, s;

  Info() : x(0), y(0), s(0) {}

  Info operator*(const Info& rhs) const {
    Info res;
    res.x = x + rhs.x;
    res.y = y + rhs.y;
    res.s = s + rhs.s + rhs.x * y;
    return res;
  }
};

long long solve(int a, int b, int c, int n) {
  if (!n) return 0;
  Info U, R;
  U.y = 1;
```

```
  R.x = 1;
  auto res = euclid(a, b, c, n, U, R);
  return res.s;
}

#endif

int main() {
  int t;
  std::cin >> t;
  for (; t; --t) {
    int a, b, c, n;
    std::cin >> n >> c >> a >> b;
    std::cout << solve(a, b, c, n - 1) << '\n';
  }
  return 0;
}

```

例题

【模板】类欧几里得算法[6]

多组询问．给定正整数 a, b, c, n，求

$$f(a,b,c,n)=\sum_{i=0}^{n}\left\lfloor\frac{ai+b}{c}\right\rfloor,$$

$$g(a,b,c,n)=\sum_{i=0}^{n}i\left\lfloor\frac{ai+b}{c}\right\rfloor,$$

$$h(a,b,c,n)=\sum_{i=0}^{n}\left\lfloor\frac{ai+b}{c}\right\rfloor^2.$$

解答二

为了应用万能欧几里得算法的模板，首先将 $i=0$ 的项提出来，单独考虑．对于剩下的部分，可以看作是对参数为 (a,b,c,n) 的线段分别计算 $\sum y, \sum xy, \sum y^2$．如正文所言，有两种将操作序列转换为幺半群元素的方式．

矩阵运算：状态向量定义为 $(1, x, y, xy, y^2, \sum y, \sum xy, \sum y^2)$．初始状态为 $(1,0,0,0,0,0,0,0)$，两个操作分别为

$$U=\begin{pmatrix}1&0&1&0&1&0&0&0\\0&1&0&1&0&0&0&0\\0&0&1&0&2&0&0&0\\0&0&0&1&0&0&0&0\\0&0&0&0&1&0&0&0\\0&0&0&0&0&1&0&0\\0&0&0&0&0&0&1&0\\0&0&0&0&0&0&0&1\end{pmatrix},\quad R=\begin{pmatrix}1&1&0&0&0&0&0&0\\0&1&0&0&0&0&0&0\\0&0&1&1&0&1&1&0\\0&0&0&1&0&0&1&0\\0&0&0&0&1&0&0&1\\0&0&0&0&0&1&0&0\\0&0&0&0&0&0&1&0\\0&0&0&0&0&0&0&1\end{pmatrix}.$$

最终答案为初始状态右乘这些操作矩阵的乘积得到的向量末尾三个分量．

这个做法的常数巨大，并不能通过本题，这里给出细节仅仅是为了辅助理解.

贡献合并：一段操作序列的贡献定义为 $(x, y, \sum y, \sum xy, \sum y^2)$. 两个操作分别为

$$U = (0,1,0,0,0),\ R = (1,0,0,0,0).$$

贡献合并时，有

$$\begin{aligned}
\sum_{S_1+S_2} y &= \sum_{S_1} y + \sum_{S_2} (y + y_1) = \sum_{S_1} y + \sum_{S_2} y + x_2 y_1, \\
\sum_{S_1+S_2} xy &= \sum_{S_1} xy + \sum_{S_2} (x + x_1)(y + y_1) \\
&= \sum_{S_1} xy + \sum_{S_2} xy + x_1 \sum_{S_2} y + y_1 \sum_{S_2} x + x_1 y_1 \sum_{S_2} 1 \\
&= \sum_{S_1} xy + \sum_{S_2} xy + x_1 \sum_{S_2} y + \frac{1}{2} x_2 (x_2 + 1) y_1 + x_1 x_2 y_1, \\
\sum_{S_1+S_2} y^2 &= \sum_{S_1} y^2 + \sum_{S_2} (y + y_1)^2 \\
&= \sum_{S_1} y^2 + \sum_{S_2} y^2 + 2y_1 \sum_{S_2} y + y_1^2 \sum_{S_2} 1 \\
&= \sum_{S_1} y^2 + \sum_{S_2} y^2 + 2y_1 \sum_{S_2} y + x_2 y_1^2.
\end{aligned}$$

这说明，应该将操作的乘法定义为

$$\begin{aligned}
&(x_1, y_1, s_1, t_1, u_1) \cdot (x_2, y_2, s_2, t_2, u_2) \\
&= (x_1 + x_2, y_1 + y_2, s_1 + s_2 + x_2 y_1, \\
&\quad t_1 + t_2 + x_1 s_2 + (1/2) x_2 (x_2 + 1) y_1 + x_1 x_2 y_1, \\
&\quad u_1 + u_2 + 2y_1 s_2 + x_2 y_1^2).
\end{aligned}$$

虽然直接验证较为繁琐，但是上述定义的贡献向量在该乘法下的确构成幺半群，单位元为 $(0,0,0,0,0)$.

对于一般的情形，有

$$\begin{aligned}
\sum_{S_1+S_2} x^r y^s &= \sum_{S_1} x^r y^s + \sum_{S_2} (x + x_1)^r (y + y_1)^s \\
&= \sum_{S_1} x^r y^s + \sum_{i=0}^{r} \sum_{j=0}^{s} \binom{r}{i} \binom{s}{j} x_1^{r-i} y_1^{s-j} \sum_{S_2} x^i y^j.
\end{aligned}$$

只要维护好所有更低幂次的贡献，就可以计算一般情形的和式.

```cpp
#include <iostream>

template <typename T>
T pow(T a, int b) {
  T res;
  for (; b; b >>= 1) {
    if (b & 1) res = res * a;
    a = a * a;
  }
  return res;
}

template <typename T>
```

```cpp
T euclid(int a, int b, int c, int n, T U, T R) {
  if (b >= c) return pow(U, b / c) * euclid(a, b % c, c, n, U, R);
  if (a >= c) return euclid(a % c, b, c, n, U, pow(U, a / c) * R);
  auto m = ((long long)a * n + b) / c;
  if (!m) return pow(R, n);
  return pow(R, (c - b - 1) / a) * U *
         euclid(c, (c - b - 1) % a, a, m - 1, R, U) *
         pow(R, n - (c * m - b - 1) / a);
}

constexpr int M = 998244353;

struct Info {
  long long x, y, s, t, u;

  Info() : x(0), y(0), s(0), t(0), u(0) {}

  Info operator*(const Info& rhs) const {
    Info res;
    res.x = (x + rhs.x) % M;
    res.y = (y + rhs.y) % M;
    res.s = (s + rhs.s + rhs.x * y) % M;
    auto tmp = (rhs.x * (rhs.x + 1) / 2 + x * rhs.x) % M;
    res.t = (t + rhs.t + x * rhs.s + tmp * y) % M;
    res.u = (u + rhs.u + 2 * y * rhs.s + rhs.x * y % M * y) % M;
    return res;
  }
};

void solve(int a, int b, int c, int n) {
  Info U, R;
  U.y = 1;
  R.x = 1;
  auto res = euclid(a, b, c, n, U, R);
  auto f = (res.s + b / c) % M;
  auto g = res.t;
  auto h = (res.u + (long long)(b / c) * (b / c)) % M;
  std::cout << f << ' ' << h << ' ' << g << '\n';
}

int main() {
  int t;
  std::cin >> t;
  for (; t; --t) {
    int a, b, c, n;
    std::cin >> n >> a >> b >> c;
    solve(a, b, c, n);
  }
  return 0;
}

```

[清华集训 2014] Sum[7]

多组询问．给定正整数 n 和 r，求

$$\sum_{d=1}^{n}(-1)^{\lfloor d\sqrt{r}\rfloor}.$$

解答二

首先，单独处理 r 为完全平方数的情形，与前文完全一致，从略．此处，仅考虑 r 不是完全平方数的情形．

本题应用万能欧几里得算法的方式有很多．比如说，可以为每个操作定义一个线性变换：

$$U(x)=-x,\ R(x)=x+1.$$

操作的乘法定义为线性变换的复合．那么，最终的答案就是操作序列对应的变换的复合得到的函数在 $x=0$ 处的值．

还可以为每段操作序列定义它的贡献．贡献可以定义为 $((-1)^y,\sum(-1)^y)$．那么，两个操作分别取

$$U=(-1,0),\ R=(1,1).$$

贡献的合并定义为

$$(u_1,v_1)\cdot(u_2,v_2)=(u_1u_2,v_1+u_1v_2).$$

容易验证，在该乘法下，所有操作构成了幺半群，且单位元为 $(1,0)$．最终的答案就是所有元素乘积的第二个分量．

这两种方法是一致的，因为如果将线性变换写作 $f(x)=ux+v$，那么线性变换的复合对应的系数的变化，恰恰就是上述操作的乘法．也就是说，这两个幺半群是同构的．

本题中，线段的参数为 (k,n)，其中，$k\in\mathbf{R}$ 为直线的斜率．设操作序列对应的乘积为 $F(k,n,U,R)$．那么，有如下递归算法：

- 如果 $k\geq 1$，那么操作序列中每个 R 前方都有至少 $\lfloor k\rfloor$ 个 U，所以，有
  $F(k,n,U,R)=F(k-\lfloor k\rfloor,n,U,U^{\lfloor k\rfloor}R)$.
- 如果 $k<1$，那么交换操作序列中的 U 和 R，并舍去末尾的 U（即交换前的 R），那么，记 $m=\lfloor kn\rfloor$，就有
  $F(k,n,U,R)=F(k^{-1},m,R,U)R^{n-\lfloor k^{-1}m\rfloor}$.

算法中，k 的迭代过程其实就是在求 $\sqrt{r}$ 的连分数展开．为此，可以应用 PQa 算法→9.11.33@p1503．求连分数的过程和万能欧几里得算法迭代的过程可以同时进行．

和类欧几里得算法的情形一致，算法的复杂度仍然是 $O(\log n)$ 的．

```cpp
#include <algorithm>
#include <cmath>
#include <iostream>

template <typename T>
T pow(T a, int b) {
  T res;
  for (; b; b >>= 1) {
    if (b & 1) res = res * a;
    a = a * a;
  }
  return res;
```

```
}

struct LinearTransform {
  int u, v;

  LinearTransform() : u(0), v(1) {}

  LinearTransform operator*(const LinearTransform& rhs) const {
    LinearTransform res;
    res.u = u + v * rhs.u;
    res.v = v * rhs.v;
    return res;
  }

  int eval(int x) const { return u + v * x; }
};

int solve(int n, int r) {
  long double sqrt_r = sqrtl(r);
  int sqr = sqrt_r;
  if (r == sqr * sqr) return sqr % 2 ? (n % 2 ? -1 : 0) : n;
  int P = 0, Q = 1, D = r, val = 0;
  LinearTransform U, R;
  U.v = -1;
  R.u = 1;
  while (n) {
    int a = (P + sqr) / Q;
    R = pow(U, a) * R;
    int m = ((P + sqrt_r) / Q - a) * n;
    P = a * Q - P;
    Q = (D - P * P) / Q;
    int rem = n - (int)(m * (P + sqrt_r) / Q);
    val = pow(R, rem).eval(val);
    std::swap(U, R);
    n = m;
  }
  return val;
}

int main() {
  int t;
  std::cin >> t;
  for (; t; --t) {
    int n, r;
    std::cin >> n >> r;
    std::cout << solve(n, r) << '\n';
  }
  return 0;
}

```

习题

模板题：

- Library Checker - Sum of Floor of Linear[8]
- Luogu P5170【模板】类欧几里得算法[9]
- Luogu P5171 Earthquake[10]
- Luogu P5172 [清华集训 2014] Sum[11]
- Luogu P4132 [BJOI2012] 算不出的等式[12]
- LOJ 138. 类欧几里得算法[13]
- LOJ 6440. 万能欧几里得[14]
- Luogu P5179 Fraction[15]
- Codeforces 1182 F. Maximum Sine[16]

应用题：

- Luogu P4433 [COCI 2009/2010 #1] ALADIN[17]
- AtCoder Beginner Contest 372 G - Ax + By < C[18]
- AtCoder Beginner Contest 313 G - Redistribution of Piles[19]
- AtCoder Beginner Contest 283 Ex - Popcount Sum[20]
- Codeforces 1098 E. Fedya the Potter[21]
- Codeforces 868 G. El Toll Caves[22]

参考资料与注释

[1] Library Checker - Sum of Floor of Linear 

[2] 【模板】类欧几里得算法 

[3] [清华集训 2014] Sum 

[4] Fraction 

[5] Library Checker - Sum of Floor of Linear 

[6] 【模板】类欧几里得算法 

[7] [清华集训 2014] Sum 

[8] Library Checker - Sum of Floor of Linear 

[9] Luogu P5170【模板】类欧几里得算法 

[10] Luogu P5171 Earthquake 

[11] Luogu P5172 [清华集训 2014] Sum 

[12] Luogu P4132 [BJOI2012] 算不出的等式 

[13] LOJ 138. 类欧几里得算法 

[14] LOJ 6440. 万能欧几里得 

[15] Luogu P5179 Fraction 

[16] Codeforces 1182 F. Maximum Sine 

[17] Luogu P4433 [COCI 2009/2010 #1] ALADIN 

[18] AtCoder Beginner Contest 372 G - Ax + By < C 

[19] AtCoder Beginner Contest 313 G - Redistribution of Piles 

[20] AtCoder Beginner Contest 283 Ex - Popcount Sum 

[21] Codeforces 1098 E. Fedya the Potter 

[22] Codeforces 868 G. El Toll Caves 

9.11.30 连分数

> **Authors:** 383494, CCXXXI, chunibyo-wly, Enter-tainer, Great-designer, megakite, Menci, shawlleyw, shuzhouliu, StudyingFather, Tiphereth-A, untitledunrevised, c-forrest, Resorie

引入

连分数可以将实数表示为一个收敛的有理数数列的极限. 这个数列中的有理数易于计算，而且提供了这个实数的最佳逼近，因而在算法竞赛中常常会用到连分数. 除此之外，连分数还和欧几里得算法密切相关，因而可以应用到一系列数论问题中.

关于连分数相关的算法实现

本文会提供一系列的连分数的算法实现，其中部分算法可能无法保证计算中间过程所涉及的整数都在 32 位或 64 位整型变量的取值范围内．对于这种情形，请参考相应的 Python 的实现，或将 C++ 实现中的整型变量替换为 高精度整数类→9.6@p1110．为突出重点，本文行文过程中的部分代码可能会调用前文实现过的函数而不再重复给出实现．

连分数

连分数（continued fraction）本身只是一种形式记号．

自 有限连分数

对于数列 $\{a_k\}_{k=0}^{n}$，连分数 $[a_0, a_1, ..., a_n]$ 表示展开式

$$x = a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{\cdots + \cfrac{1}{a_n}}}}.$$

连分数有意义，当且仅当对应的展开式有意义．这些 a_k 称为连分数的 **项**（term）或 **系数**（coefficient）．

记号

更一般的连分数允许展开式中的分子不恒为 1，相应的连分数记号也需要修改，这超出了本文的范畴．另外，有些文献中会将第一个逗号「,」写作分号「;」，这与本文的记号在含义上没有差异．

当然，连分数还可以推广到无穷数列的情形．

自 无限连分数

对于无穷数列 $\{a_k\}_{k=0}^{\infty}$，连分数 $[a_0, a_1, ...]$ 表示极限

$$x = \lim_{k\to\infty} x_k = \lim_{k\to\infty} [a_0, a_1, ..., a_k].$$

连分数有意义，当且仅当对应的极限有意义．其中，$x_k = [a_0, a_1, ..., a_k]$ 称为 x 的第 k 个 **渐近分数**（convergent）或 **收敛子**，而 $r_k = [a_k, a_{k+1}, ...]$ 称为 x 的第 k 个 **余项** 或 **完全商**（complete quotient）．相应地，项 a_k 有时也称为第 k 个 **部分商**（partial quotient）．

简单连分数

数论中，主要考虑连分数的项都是整数的情形．

自 简单连分数

对于连分数 $[a_0, a_1, ...]$，如果 a_0 是整数，$a_1, a_2, ...$ 都是正整数，则称它为 **简单连分数**（simple continued fraction），也简称 **连分数**．如果数列 $\{a_i\}$ 有限，则称为 **有限（简单）连分数**；否则称为 **无限（简单）连分数**．而且，a_0 称为它的 **整数部分**（integer part）．

除非特别说明，本文提到的连分数都指的是简单连分数．可以证明，无限的简单连分数必然是收敛的，而且简单连分数的余项也一定是正的．

连分数有如下基本性质：

性质

设实数 $x=[a_0,a_1,a_2,...]$. 那么，成立如下性质：

1. 对于任意 $k\in\mathbf{Z}$，有 $x+k=[a_0+k,a_1,a_2,...]$；
2. 对实数 $x>1$，有 $a_0>0$，且它的倒数 $x^{-1}=[0,a_0,a_1,a_2,...]$.

有限连分数对应的是有理数. 每个有理数都有且仅有两种方式可以表示成连分数，长度必然一奇一偶. 这两种方式唯一的区别在于最后一项是否为 1，即

$$x=[a_0,a_1,...,a_n]=[a_0,a_1,...,a_n-1,1].$$

这两个连分数称为有理数 x 的 **连分数表示**（continued fraction representation）. 其中，末项不为一的称为标准表示，末项为一的称为非标准表示. [59]

例子

有理数 $x=\dfrac{5}{3}$ 的连分数表示为

$$x=[1,1,1,1]=1+\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{1}}},$$

$$x=[1,1,2]=1+\cfrac{1}{1+\cfrac{1}{2}}.$$

无限连分数对应的是无理数. 而且，每个无理数仅有唯一的方式表示为连分数，称为无理数的连分数表示.

连分数表示的求法

要求某个实数 x 的连分数表示，只需要注意到它的余项 r_k 如果不是整数，就一定满足

$$r_k=[a_k,a_{k+1},...]=[a_k,r_{k+1}]=a_k+\frac{1}{r_{k+1}}.$$

而且，$r_{k+1}>1$. 因此，可以从 $r_0=x$ 开始递归地计算

$$a_k=\lfloor r_k\rfloor,\ r_{k+1}=\frac{1}{r_k-a_k}.$$

这个过程产生的数列 $\{a_k\}$ 总是唯一确定的，除非某个余项 r_k 成为整数. 如果出现了 r_k 是整数，则说明过程应当终止，可以选择输出相应的标准表示或者非标准表示.

在算法竞赛中，往往处理的都是有理数 $x=\dfrac{p}{q}$ 的情形. 此时，每个余项 r_k 都是有理数 $\dfrac{p_k}{q_k}$，而且对于 $k>0$，因为 $r_k>1$，就总有 $p_k>q_k$. 具体计算上述递推关系，可以发现

$$a_k=\left\lfloor\frac{p_k}{q_k}\right\rfloor,\ r_{k+1}=\frac{1}{r_k-a_k}=\frac{q_k}{p_k-a_kq_k}=\frac{q_k}{p_k\bmod q_k}.$$

此时的计算过程实际上是对 p 和 q 做 辗转相除法→9.11.4@p1188. 这也说明，对于有理数 $x=\dfrac{p}{q}$，连分数表示的长度是 $O(\log\min\{p,q\})$ 的. 计算有理数 $\dfrac{p}{q}$ 的复杂度也是 $O(\log\min\{p,q\})$ 的.

参考实现

给定分数的分子 p 和分母 q，输出连分数的系数序列 $[a_0,a_1,...,a_n]$.

- C++

[59] 自然数 1 只有非标准表示：$1=[1]=[0,1]$.

```cpp
// Find the continued fraction representation of P/Q.
auto fraction(int p, int q) {
  std::vector<int> a;
  while (q) {
    a.push_back(p / q);
    std::tie(p, q) = std::make_pair(q, p % q);
  }
  return a;
}

```

- Python

```python
# Find the continued fraction representation of P/Q.
def fraction(p, q):
    a = []
    while q:
        a.append(p // q)
        p, q = q, p % q
    return a


```

渐近分数

在连分数的定义中介绍了渐近分数的概念. 实数的渐近分数就是它的连分数表示的渐近分数: 在实数 x 的连分数表示中，只保留前 k 个项，得到的连分数 x_k 就称为实数 x 的第 k 个渐近分数. 实数 x 的渐近分数 x_k 都是有理数，而且序列 $\{x_k\}$ 收敛于实数 x.

例子：黄金分割比的渐近分数

连分数 $x = [1, 1, 1, 1, ...]$ 的前几个渐近分数分别是

$$x_0 = [1] = 1,$$

$$x_1 = [1, 1] = 2,$$

$$x_2 = [1, 1, 1] = \frac{3}{2},$$

$$x_3 = [1, 1, 1, 1] = \frac{5}{3},$$

$$x_4 = [1, 1, 1, 1, 1] = \frac{8}{5}.$$

可以归纳地证明

$$x_k = \frac{F_{k+2}}{F_{k+1}},$$

其中，$\{F_k\}$ 是 斐波那契数列→9.13.4@p1629. 根据它的通项公式可知,

$$x_k = \frac{\phi^{k+2} - (-\phi)^{-(k+2)}}{\phi^{k+1} - (-\phi)^{-(k+1)}},$$

其中的 $\phi = \frac{1+\sqrt{5}}{2}$ 是黄金分割比. 当 k 趋于无穷时，有

$$x = \lim_{k \to \infty} x_k = \phi.$$

因而，连分数 $x = [1, 1, 1, 1, ...]$ 表示的是黄金分割比 ϕ.

这些渐近分数趋近于相应的实数，所以可以用于逼近该实数．为此，有必要了解渐近分数的性质．

递推关系

首先，要解决这些渐近分数的计算问题．虽然渐近分数总是在连分数的后面添加一项，但是并不需要每次都重新计算它的值．其实，渐近分数有如下递推关系：

递推公式

对于连分数 $x=[a_0,a_1,a_2,...]$，设它的第 k 个渐近分数 $x_k=[a_0,a_1,...,a_k]$ 可以唯一写成既约分数 $\dfrac{p_k}{q_k}$．那么，有

$$\begin{aligned}p_k&=a_kp_{k-1}+p_{k-2},\\q_k&=a_kq_{k-1}+q_{k-2}.\end{aligned}$$

递推的起点是（形式）分数

$$x_{-1}=\frac{p_{-1}}{q_{-1}}=\frac{1}{0},\ x_{-2}=\frac{p_{-2}}{q_{-2}}=\frac{0}{1}.$$

证明

每个渐近分数都是有限连分数．前面已经提到过，求有限连分数其实相当于做辗转相除法．但是辗转相除法得到的实际上是连分数的一个后缀，而渐近分数需要的是连分数的一个前缀．为了得到这个前缀，需要考察每次加上一步时，连分数怎么变化．为此，可以使用矩阵乘法表示辗转相除过程的一步，即

$$M(a)=\begin{pmatrix}a&1\\1&0\end{pmatrix},\qquad M(a)\begin{pmatrix}u\\v\end{pmatrix}=\begin{pmatrix}au+v\\u\end{pmatrix},\qquad \frac{au+v}{u}=a+\frac{1}{u/v}.$$

向量 $(a_k,1)^{\mathrm{T}}$ 表示有限连分数 $[a_k]$；依次左乘矩阵相当于在左侧依次补上连分数的系数．将该连分数补成渐近分数 $x_k=\dfrac{p_k}{q_k}=[a_0,a_1,...,a_k]$ 的过程得到如下等式：

$$M(a_0)M(a_1)M(a_2)\cdots M(a_{k-1})\begin{pmatrix}a_k\\1\end{pmatrix}=\begin{pmatrix}p_k\\q_k\end{pmatrix}.$$

由 $\det(M(a_0)M(a_1)\cdots M(a_{k-1}))=(-1)^k$ 知，该矩阵变换将本原向量 $(a_k,1)^{\mathrm{T}}$ 变换为 $(p_k,q_k)^{\mathrm{T}}$，故 p_k 和 q_k 必互质．$M(a_k)$ 的第二列为 $(1,0)^{\mathrm{T}}$，而 $M(a_{k-1})(1,0)^{\mathrm{T}}=(a_{k-1},1)^{\mathrm{T}}$，这一列的乘积可以递推代入得到．补上这一列，便得到

$$M(a_0)M(a_1)M(a_2)\cdots M(a_k)=\begin{pmatrix}p_k&p_{k-1}\\q_k&q_{k-1}\end{pmatrix}.$$

现在只需代入矩阵乘法的前 k 项，便容易得渐近分数的递推公式：

$$\begin{pmatrix}p_{k-1}&p_{k-2}\\q_{k-1}&q_{k-2}\end{pmatrix}M(a_k)=\begin{pmatrix}p_k&p_{k-1}\\q_k&q_{k-1}\end{pmatrix}$$

$$\Rightarrow\quad p_k=a_kp_{k-1}+p_{k-2},\qquad q_k=a_kq_{k-1}+q_{k-2}.$$

记号

本文将渐近分数 x_k 记作 $\dfrac{p_k}{q_k}$ 时，总是默认分子 p_k 和 q_k 由上面的递推关系给出．前文的证明说明，这样总能得到渐近分数的既约表示．

这个递推式说明

$$x_k=\frac{a_kp_{k-1}+p_{k-2}}{a_kq_{k-1}+q_{k-2}}$$

介于 x_{k-1} 和 x_{k-2} 之间．

利用本节得到的递推关系，可以得到计算渐近分数的算法如下：

参考实现

给定连分数的系数 $a_0, a_1, ..., a_n$，求渐近分数的分子和分母序列 $(p_0, q_0), (p_1, q_1), ..., (p_n, q_n)$.

- C++

```cpp
// Find the convergents of a continued fraction A.
// Numerators and denominators stored separately in P and Q.
auto convergents(std::vector<int> a) {
  std::vector<int> p = {0, 1};
  std::vector<int> q = {1, 0};
  for (auto it : a) {
    p.push_back(p.back() * it + p.end()[-2]);
    q.push_back(q.back() * it + q.end()[-2]);
  }
  return std::make_pair(p, q);
}

```

- Python

```python
# Find the convergents of a continued fraction A.
# Numerators and denominators stored separately in P and Q.
def convergents(a):
    p = [0, 1]
    q = [1, 0]
    for it in a:
        p.append(p[-1] * it + p[-2])
        q.append(q[-1] * it + q[-2])
    return p, q


```

连项式

连项式（continuant）来源于渐近分数递推关系的中间量．[60]

自 连项式

对任意有限序列，如下递归定义连项式：

- $K(\emptyset) = 1, K(a) = a$;
- 对 $n \geq 1$，$K(a_0, ..., a_n) = a_n K(a_0, ..., a_{n-1}) + K(a_0, ..., a_{n-2})$.

根据渐近分数的递推容易发现，对 $x = [a_0, a_1, a_2, ...]$，其渐近分数 $\frac{p_k}{q_k} = [a_0, a_1, ..., a_k]$ 可以表示成连项式的形式：

$$p_k = K(a_0, a_1, ..., a_k), \quad q_k = K(a_1, ..., a_k).$$

也就是说，连项式可以用来表示有限连分数：

$$\frac{K(a_0, a_1, ..., a_k)}{K(a_1, ..., a_k)} = [a_0, a_1, ..., a_k].$$

先前的矩阵表示也容易用连项式改写．当 $k \geq 1$ 时，有

[60]译名来自张明尧、张凡翻译的《具体数学》第 6.7 节.

$$M(a_0)\cdots M(a_k)=\begin{pmatrix}K(a_0,\ldots,a_k) & K(a_0,\ldots,a_{k-1})\\ K(a_1,\ldots,a_k) & K(a_1,\ldots,a_{k-1})\end{pmatrix}.$$

连项式最重要的性质是其对称性.

连项式的对称性

对任意有限序列 $(a_0,\ldots,a_n)$，有

$$K(a_0,\ldots,a_n)=K(a_n,\ldots,a_0).$$

证明

由于任意矩阵 $M(a)$ 都是对称矩阵，将其乘积转置，得到

$$(M(a_0)\cdots M(a_n))^{\mathrm{T}}=M(a_n)^{\mathrm{T}}\cdots M(a_0)^{\mathrm{T}}=M(a_n)\cdots M(a_0).$$

比较两个乘积矩阵的左上角，立即得到结论.

由对称性，连项式的递推关系也可以反向写作

$$K(a_0,\ldots,a_n)=a_0K(a_1,\ldots,a_n)+K(a_2,\ldots,a_n).$$

连项式的行列式表示

连项式可以写成行列式的形式：

$$K(a_0,\ldots,a_k)=\det\begin{pmatrix}a_0 & 1 & 0 & \cdots & 0\\ -1 & a_1 & 1 & \ddots & \vdots\\ 0 & -1 & a_2 & \ddots & 0\\ \vdots & \ddots & \ddots & \ddots & 1\\ 0 & \cdots & 0 & -1 & a_k\end{pmatrix}.$$

这是一个 三对角矩阵[1] 的行列式，从左上角开始展开，可以得到反向递推. 反过来，从右下角开始展开，则能得到定义中的递推关系.

连项式另一个有趣的性质是其拼接性质.

连项式的拼接

对两个非空序列 $(a_1,\ldots,a_m)$ 和 $(b_1,\ldots,b_n)$，有

$$K(a_1,\ldots,a_m,b_1,\ldots,b_n)=K(a_1,\ldots,a_m)K(b_1,\ldots,b_n)+K(a_1,\ldots,a_{m-1})K(b_2,\ldots,b_n).$$

证明

仅需要将两段矩阵乘积利用结合律分开即可. 令

$$P=M(a_1)\cdots M(a_m),\quad Q=M(b_1)\cdots M(b_n),$$

那么有

$$(PQ)_{1,1}=P_{1,1}Q_{1,1}+P_{1,2}Q_{2,1}.$$

这恰对应拼接性质的恒等式.

该性质可以用来考察辗转相除法的中间量. 例如，Heilbronn 曾用它估计辗转相除法的平均步数[61]. 下面给出一个利用该性质处理相关求和问题的竞赛例题.

[61] H. Heilbronn, On the Average Length of a Class of Finite Continued Fractions, 1969.[2]

洛谷 P17229 [Math×Girl²] 英雄变奏曲[3]

给定一个正整数 N，以及两个整数值函数 $g,h:\{1,2,...,N\}\to\mathbf{Z}$ 的点值.

对于满足 $1\le y<x\le N$ 的整数 x,y，令 $z=x \bmod y$，并递归定义

$$f(x,y)=\begin{cases}0, & z=0,\\ g\left(\left\lfloor\frac{x}{y}\right\rfloor\right)h\left(\left\lfloor\frac{y}{z}\right\rfloor\right)+f(y,z), & z>0.\end{cases}$$

求

$$S(N)=\sum_{\substack{1\le b<a\le N\\ \gcd(a,b)=1}} f(a,b).$$

思路

题目中 $f(x,y)$ 的定义对应欧几里得算法的过程. 用连分数表示欧几里得算法，设 $\frac{x}{y}=[a_0,a_1,...,a_k]$ 为有限连分数的标准表示，那么有

$$f(x,y)=\sum_{i=0}^{k-1} g(a_i)h(a_{i+1}).$$

利用连项式的拼接性质，从 a_i 处切开连项式，有

$$K(a_0,a_1,...,a_k)=K(a_0,...,a_i)K(a_{i+1},...,a_k)+K(a_0,...,a_{i-1})K(a_{i+2},...,a_k).$$

令

$$X=K(a_0,...,a_i),\quad Y=K(a_0,...,a_{i-1}),$$
$$Z=K(a_{i+1},...,a_k),\quad W=K(a_{i+2},...,a_k).$$

由连项式表示有限连分数的性质得到

$$\frac{X}{Y}=[a_i,a_{i-1},...,a_0],\quad \frac{Z}{W}=[a_{i+1},a_{i+2},...,a_k].$$

需要满足的条件是 $\gcd(X,Y)=\gcd(Z,W)=1,1\le Y\le X,1\le W\le Z,XZ+YW=x\le N$. 看起来我们做出了一个 (x,y,i) 和 (X,Y,Z,W) 之间的双射. 但是注意此处由于 a_0 可能为 1，$\frac{X}{Y}$ 对应的连分数可能是非标准表示；而 $\frac{Z}{W}$ 对应的连分数则必定为标准表示（由此也必不可能有 $Z=W=1$）.

现在算一下这个变换之后，四元组 (X,Y,Z,W) 对原求和式的贡献. 对分数取整可以得到连分数首项，这样我们就得到了函数的自变量. 接下来对 (X,Y) 分情况讨论：

1. $Y>1$：$\frac{X}{Y}$ 的标准表示和非标准表示均对应一组 (x,y,i)，贡献为 $2g\left(\left\lfloor\frac{X}{Y}\right\rfloor\right)h\left(\left\lfloor\frac{Z}{W}\right\rfloor\right)$；
2. $Y=1,X\ge 2$：$\frac{X}{Y}=X$，对应 $[X]$ 和 $[X-1,1]$，注意后者的贡献不同，总贡献为 $(g(X)+g(X-1))h\left(\left\lfloor\frac{Z}{W}\right\rfloor\right)$；
3. $X=Y=1$：贡献为 $g(1)h\left(\left\lfloor\frac{Z}{W}\right\rfloor\right)$.

由此，我们将特别难处理的 $S(N)$ 变为了以下关于 X,Y,Z,W 的求和（补充定义 $g(0)=2g(1)$）：

$$S(N)=2\sum_{\substack{XZ+YW\le N\\ X>Y\ge 1,\,Z>W\ge 1\\ \gcd(X,Y)=\gcd(Z,W)=1}} g(\lfloor X/Y\rfloor)h(\lfloor Z/W\rfloor)$$
$$+\sum_{\substack{XZ+W\le N\\ X\ge 1,\,Z>W\ge 1\\ \gcd(Z,W)=1}} (g(X-1)-g(X))h(\lfloor Z/W\rfloor).$$

这个和式相对更容易用常规工具处理，在此不再赘述.

作为渐近分数的递推关系和连项式性质的推论，成立如下的反序定理和倒数定理：

反序定理

设实数 $x=[a_0,a_1,a_2,...]>0$ 的第 k 个渐近分数是 $\frac{p_k}{q_k}$，则相邻两个渐近分数的分子和分母的比值分别为

$$\frac{p_k}{p_{k-1}}=[a_k,a_{k-1},...,a_1,a_0],$$

$$\frac{q_k}{q_{k-1}}=[a_k,a_{k-1},...,a_1].$$

如果 $a_0=0$ 且 $k\geq 2$，则第一个连分数应当理解为在倒数第二项处截断，即 $[a_k,a_{k-1},...,a_2]$.

证明

根据连项式表示可知

$$\frac{q_k}{q_{k-1}}=\frac{K(a_1,...,a_{k-1},a_k)}{K(a_1,...,a_{k-1})}=\frac{K(a_k,a_{k-1},...,a_1)}{K(a_{k-1},...,a_1)}=[a_k,a_{k-1},...,a_1].$$

如果 $a_0\neq 0$，则

$$\frac{p_k}{p_{k-1}}=\frac{K(a_0,...,a_{k-1},a_k)}{K(a_0,...,a_{k-1})}=\frac{K(a_k,a_{k-1},...,a_0)}{K(a_{k-1},...,a_0)}=[a_k,a_{k-1},...,a_0].$$

如果 $a_0=0$，由连项式的递推关系有 $K(0,a_1,a_2,...,a_k)=K(a_2,...,a_k)$，故有

$$\frac{p_k}{p_{k-1}}=\frac{K(a_2,...,a_{k-1},a_k)}{K(a_2,...,a_{k-1})}=\frac{K(a_k,a_{k-1},...,a_2)}{K(a_{k-1},...,a_2)}=[a_k,a_{k-1},...,a_2].$$

倒数定理

实数 $x>0$ 的渐近分数的倒数是 x^{-1} 的渐近分数.

证明

不妨设 $x>1$ 且有连分数表示 $[a_0,a_1,a_2,...]$，则 x^{-1} 的连分数表示是 $[0,a_0,a_1,a_2,...]$. 它们的渐近分数可以从递推关系中求得. 而且，对于 x，有初值条件 $x_{-2}=\frac{0}{1}$ 和 $x_{-1}=\frac{1}{0}$；对于 $y=x^{-1}$，有初值条件 $y_{-1}=\frac{1}{0}$ 和 $y_0=\frac{0}{1}$. 因此，有 $x_{-2}=(y_{-1})^{-1}$ 和 $x_{-1}=(y_0)^{-1}$. 根据递推关系，可以得到 $x_k=y_{k+1}^{-1}$. 这就说明，x 的渐近分数的倒数是 $y=x^{-1}$ 的渐近分数. 对于 $0<x\leq 1$ 的情形，也可以做类似讨论.

误差估计

利用渐近分数的递推公式，可以估计用渐近分数逼近实数产生的误差.

首先，可以计算相邻的渐近分数的差值：

渐近分数的差分

设 $x_k=\frac{p_k}{q_k}$ 是实数 x 的第 k 个渐近分数. 那么，有

$$p_{k+1}q_k-p_kq_{k+1}=(-1)^k.$$

因此，相邻两项的渐近分数的差分是

$$x_{k+1}-x_k=\frac{(-1)^k}{q_{k+1}q_k}.$$

证明

根据递推关系，有

$$\det\begin{pmatrix}p_{k+1}&p_k\\q_{k+1}&q_k\end{pmatrix}=\det\begin{pmatrix}a_{k+1}p_k+p_{k-1}&p_k\\a_{k+1}q_k+q_{k-1}&q_k\end{pmatrix}=\det\begin{pmatrix}p_{k-1}&p_k\\q_{k-1}&q_k\end{pmatrix}$$

$$=-\det\begin{pmatrix}p_k&p_{k-1}\\q_k&q_{k-1}\end{pmatrix}=(-1)^{k+2}\det\begin{pmatrix}1&0\\0&1\end{pmatrix}=(-1)^k.$$

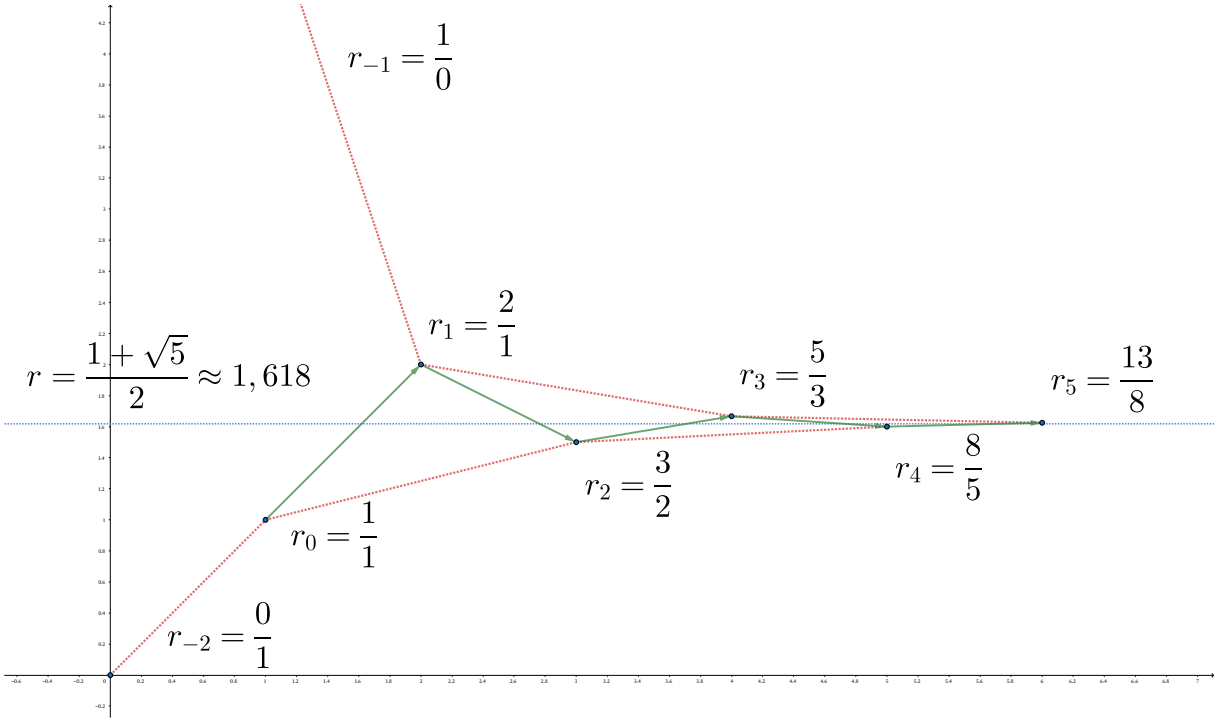
这就是 $p_{k+1}q_k-p_kq_{k+1}=(-1)^k$. 对两边同除以 $q_{k+1}q_k$，就得到关于 $x_{k+1}-x_k$ 的结论.

因而，奇数项渐近分数总是大于相邻两项，偶数项渐近分数总是小于相邻两项：渐近分数是交错变化的.

如果只看偶数项（奇数项）渐近分数，序列也是单调递增（递减）的. 这是因为

$$x_{k+2}-x_k=\frac{(-1)^{k+1}}{q_{k+2}q_{k+1}}+\frac{(-1)^k}{q_{k+1}q_k}=\frac{(-1)^k(q_{k+2}-q_k)}{q_{k+2}q_{k+1}q_k}=\frac{(-1)^ka_{k+2}}{q_{k+2}q_k}$$

当 k 为偶数（奇数）时为正（负）. 同时，因为成立递推关系 $q_k=a_kq_{k-1}+q_{k-2}$，分母 q_k 的增长速度不会慢于斐波那契数列的速度. 所以，相邻两项的差一定趋近于零. 这就说明，偶数项和奇数项渐近分数分别自下而上和自上而下地逼近同一极限. 这就证明了无限简单连分数一定收敛. 渐近分数趋近于相应实数的动态可以见下图：



自 上（下）渐近分数

对于实数 x 和它的渐近分数 x_k，如果 $x_k>x$（$x_k<x$），就称 x_k 为 x 的 **上（下）渐近分数**（upper (lower) convergent）.

前面已经说明，上渐近分数就是奇数项渐近分数，下渐近分数就是偶数项渐近分数.

利用差分公式，可以将实数 x 写成交错级数的形式：

$$x = a_0 + \sum_{k=0}^{\infty} \frac{(-1)^k}{q_{k+1}q_k}.$$

连分数定义中的渐近分数和余项就是该级数的部分和和余项.

利用差分公式，还可以直接对渐近分数逼近实数产生的误差做出估计：

误差

设 $x_k = \dfrac{p_k}{q_k} \neq x$ 是实数 x 的第 k 个渐近分数. 那么，有

$$x_k - x = \frac{(-1)^k}{q_k(r_{k+1}q_k + q_{k-1})},$$

其中，r_{k+1} 是实数 x 的第 $k+1$ 个余项. 进而，有

$$\frac{1}{2q_{k+1}^2} \leq \frac{1}{q_k(q_k + q_{k+1})} \leq \left| x - \frac{p_k}{q_k} \right| \leq \frac{1}{q_k q_{k+1}} \leq \frac{1}{q_k^2}.$$

证明

因为 $x = [a_0, a_1, ..., a_k, r_{k+1}]$，而且对形式连分数也成立渐近分数的差分公式，所以有

$$x - x_k = \frac{(-1)^k}{q_k(r_{k+1}q_k + q_{k-1})},$$

其中，$r_{k+1}q_k + q_{k-1}$ 就是按照递推公式得到的这个形式连分数的第 $k+1$ 个渐近分数的分母.

要完成随后的不等式估计，只需要注意到当 $x_k \neq x$ 时，总成立

$$1 \leq a_{k+1} \leq r_{k+1} \leq a_{k+1} + 1,$$

所以有

$$q_{k+1} = a_{k+1}q_k + q_{k-1} \leq r_{k+1}q_k + q_{k-1} \leq q_k + (a_{k+1}q_k + q_{k-1}) = q_k + q_{k+1}.$$

因此，有不等式

$$\frac{1}{q_k(q_k + q_{k+1})} \leq \left| x - \frac{p_k}{q_k} \right| = \frac{1}{q_k(r_{k+1}q_k + q_{k-1})} \leq \frac{1}{q_k q_{k+1}}.$$

要得到外侧的放缩，再注意到 $q_k \leq q_{k+1}$ 就可以了.

本节的差分公式还有一个简单推论：渐近分数 $\dfrac{p_k}{q_k}$ 都是既约的.

推论

对于任何实数 x，且渐近分数 $x_k = \dfrac{p_k}{q_k}$ 的分子和分母由递推公式给出，则 $\dfrac{p_k}{q_k}$ 是既约分数，即 $\gcd(p_k, q_k) = 1$.

证明

对差分公式应用 裴蜀定理→9.11.8@p1218 即可.

其实，二元一次不定方程的解可以通过连分数的方法求解.

二元一次不定方程的求解

给定 $A, B, C \in \mathbf{Z}$. 查找 $x, y \in \mathbf{Z}$，使得 $Ax + By = C$ 成立.

解答

虽然这个问题通常是用 扩展欧几里得算法→9.11.8@p1218 解决的，但是同样可以通过连分数求解. 设 $\frac{A}{B}=[a_0,a_1,...,a_k]$. 上面证明了 $p_kq_{k-1}-p_{k-1}q_k=(-1)^{k-1}$. 将 p_k 和 q_k 替换为 A 和 B，得到

$$Aq_{k-1}-Bp_{k-1}=(-1)^{k-1}g,$$

其中，$g=\gcd(A,B)$. 如果 g 整除 C，则一组解为 $x=(-1)^{k-1}\frac{C}{g}q_{k-1}$ 和 $y=(-1)^k\frac{C}{g}p_{k-1}$；否则无解.

- C++

```cpp
// Return (x,y) such that Ax+By=C.
// Assume that such (x,y) exists.
auto dio(int A, int B, int C) {
  std::vector<int> p, q;
  std::tie(p, q) = convergents(fraction(A, B));
  C /= A / p.back();
  int t = p.size() % 2 ? -1 : 1;
  return std::make_pair(t * C * q.end()[-2], -t * C * p.end()[-2]);
}

```

- Python

```python
# Return (x, y) such that Ax+By=C.
# Assume that such (x, y) exists.
def dio(A, B, C):
    p, q = convergents(fraction(A, B))
    C //= A // p[-1]  # divide by gcd(A, B)
    t = (-1) if len(p) % 2 else 1
    return t * C * q[-2], -t * C * p[-2]


```

丢番图逼近

连分数理论的一个重要应用就是丢番图逼近理论. 丢番图逼近（Diophantine approximation）是指用有理数逼近实数. 当然，由于有理数的稠密性，如果不加以限制，可以得到误差任意小的逼近. 因此，需要对可以使用的有理数做出限制，比如只能选择分母小于某个值的有理数. 本节就讨论了这种限制下的最佳逼近和连分数的关系.

用渐近分数逼近实数

首先，利用渐近分数的误差估计，立刻得到如下结果：

定理（Dirichlet）

对于无理数 x，存在无穷多个既约分数 $\frac{p}{q}$ 使得

$$\left|x-\frac{p}{q}\right|<\frac{1}{q^2}$$

成立.

证明

根据渐近分数的误差估计，对于无理数 x 的第 k 个渐近分数 $x_k = \dfrac{p_k}{q_k}$，有

$$\left| x - \frac{p_k}{q_k} \right| \leq \frac{1}{q_k^2}.$$

检查误差公式的证明即可知，对于任一无理数 x，取等条件并不成立．因此，它的所有渐近分数的分子和分母都满足要求．

这个定理也可以看作是 Dirichlet 逼近定理[4] 的推论．这几乎已经是最好的结果了．不等式右侧分母中的指数 2 已经不能再改进，但是常数上可以做得更好．Hurwitz 定理说明，不等式右侧可以缩小到 $\dfrac{1}{\sqrt{5}q^2}$，且这是最好的界．

Hurwitz 定理

对于无理数 x，存在无穷多个既约分数 $\dfrac{p}{q}$ 使得

$$\left| x - \frac{p}{q} \right| < \frac{1}{\sqrt{5}q^2}$$

成立，而且不等式右侧的 $\sqrt{5}$ 不能换成更大的实数．

证明（Borel）

Borel 实际上证明了，无理数 x 的连续三个渐近分数中，必然有至少一个满足上述条件．因为渐近分数无穷多，且都是既约分数，那么 Hurwitz 定理的第一部分就必然成立．

反证法．不妨设存在无理数 x 和它的渐近分数 x_{k-1}, x_k, x_{k+1}，使得

$$\left| x - \frac{p_{k-1}}{q_{k-1}} \right| \geq \frac{1}{\sqrt{5}q_{k-1}^2}, \left| x - \frac{p_k}{q_k} \right| \geq \frac{1}{\sqrt{5}q_k^2}, \left| x - \frac{p_{k+1}}{q_{k+1}} \right| \geq \frac{1}{\sqrt{5}q_{k+1}^2}$$

成立．因为相邻的渐近分数必然位于 x 两侧，所以由差分公式知

$$\frac{1}{q_{k-1}q_k} = \left| \frac{p_{k-1}}{q_{k-1}} - \frac{p_k}{q_k} \right| = \left| x - \frac{p_{k-1}}{q_{k-1}} \right| + \left| x - \frac{p_k}{q_k} \right| \geq \frac{1}{\sqrt{5}q_{k-1}^2} + \frac{1}{\sqrt{5}q_k^2}.$$

它可以写成关于商 $\dfrac{q_k}{q_{k-1}}$ 的不等式

$$\frac{q_k}{q_{k-1}} + \frac{q_{k-1}}{q_k} \leq \sqrt{5}.$$

因为左侧是有理数，右侧是无理数，等号必然无法取得．又因为 $q_k \geq q_{k-1}$，所以可以解得

$$1 \leq \frac{q_k}{q_{k-1}} < \frac{\sqrt{5}+1}{2}.$$

同理，可以证明

$$1 \leq \frac{q_{k+1}}{q_k} < \frac{\sqrt{5}+1}{2}.$$

但是根据递推公式，并结合两式可知，

$$a_{k+1} = \frac{q_{k+1}}{q_k} - \frac{q_{k-1}}{q_k} < \frac{\sqrt{5}+1}{2} - \frac{\sqrt{5}-1}{2} = 1$$

这与简单连分数的定义矛盾．所以，Borel 的结论成立．

要说明这样得到的界是最好的，只需要找到 x 使得对于任何 $C > \sqrt{5}$，都只存在有限多个既约分数 $\dfrac{p}{q}$ 使得不等式

$$\left|x-\frac{p}{q}\right|<\frac{1}{Cq^2}$$

成立．下面证明 $\phi=\dfrac{\sqrt{5}+1}{2}$ 就是这样的 x．[62]

设 $\phi'=\dfrac{-\sqrt{5}+1}{2}$ 是 ϕ 的共轭根．它们都是方程 $x^2-x-1=0$ 的根．因而，对任意实数 x，都有

$$x^2-x-1=(x-\phi)(x-\phi').$$

代入既约分数 $\dfrac{p}{q}$，就得到

$$\frac{1}{q^2}\le\frac{|\,p^2-pq-q^2|}{q^2}=\left|\frac{p}{q}-\phi\right|\left|\frac{p}{q}-\phi'\right|\le\left|\frac{p}{q}-\phi\right|\left(\left|\frac{p}{q}-\phi\right|+|\,\phi-\phi'|\right)<\frac{1}{Cq^2}\left(\frac{1}{Cq^2}+\sqrt{5}\right).$$

对于 $C>\sqrt{5}$，可以直接解出 $q<\dfrac{1}{\sqrt{C(C-\sqrt{5})}}$，因而不可能存在无穷多组解满足上述不等式．

这些定理的证明说明，渐近分数提供了相当好的丢番图逼近．但是，这未必是最佳逼近．要讨论最佳逼近，需要说明逼近程度的度量．这常常有两种选择．

⚠ 可能存在最佳逼近相关结论不成立的情形

接下来的两节，会叙述一些关于最佳逼近的结果．这些结果可能对个别无趣的情形并不成立．比如，最佳逼近的两类定义都要求严格不等号，但是对于半奇数 $x=n+\dfrac{1}{2}$ 且 $n\in\mathbf{Z}$，它的连分数的形式可以是 $[n,1,1]$．此时，它的前两个渐近分数 $x_0=n$ 和 $x_1=n+1$ 的分母都是 1，而且到 x 的距离是一样的．这说明，它们都不是最佳逼近．对于本节的结论的叙述，读者应当默认这样的情形已经排除在外．如果读者不关心最末尾的几个渐近分数，抑或是只关心无理数的逼近，那么不必理会这些额外的复杂情形．

第一类最佳逼近：中间分数

第一类最佳逼近使用

$$\left|x-\frac{p}{q}\right|$$

衡量逼近的程度．

自 第一类最佳逼近

对于实数 x 和有理数 $\dfrac{p}{q}$，如果对于任意的 $\dfrac{p'}{q'}\ne\dfrac{p}{q}$ 且 $0<q'\le q$ 都有

$$\left|x-\frac{p}{q}\right|<\left|x-\frac{p'}{q'}\right|,$$

就称有理数 $\dfrac{p}{q}$ 是实数 x 的 **第一类最佳逼近**（best approximation of the first kind）．

第一类最佳逼近未必是渐近分数，而是一类更宽泛的分数．

[62] 此时不能默认既约分数 $\dfrac{p}{q}$ 一定是渐近分数，虽然 Legendre 定理表明 $\dfrac{p}{q}$ 确实只能是某个渐近分数．对于渐近分数的情形，可以通过渐近分数逼近实数的误差入手加以证明．

[63] 不同文献可能对此处的 t 的取值范围是否包括端点有不同的处理．

自 中间分数

设实数 x 有渐近分数 $x_{k+1}=[a_0,a_1,...,a_k,a_{k+1}]$，且整数 t 满足 $0\le t\le a_{k+1}$[63]，则分数 $x_{k,t}=[a_0,a_1,...,a_k,t]$ 称为 x 的 **中间分数**（intermediate fraction）、**半收敛子**（semiconvergent）或 **次渐近分数**（secondary convergent）.[64]

类似于渐近分数的情形，大于（小于）x 的中间分数称为 **上（下）中间分数**（upper (lower) semiconvergent）.

根据递推公式，中间分数可以写成

$$x_{k,t}=\frac{tp_k+p_{k-1}}{tq_k+q_{k-1}}.$$

它必然是既约分数，而且位于渐近分数 x_{k-1} 和 x_{k+1} 之间. 随着 t 增大，它也逐渐向 x_{k+1} 靠拢：（以 k 为偶数的情形为例）

$$x_{k-1}=x_{k,0}<x_{k,1}<x_{k,2}<\cdots<x_{k,a_{k+1}}=x_{k+1}.$$

因为渐近分数的分子和分母都是递增的，中间分数 $x_{k,t}$ $(t\neq 0)$ 的分子和分母落在了 x_k 和 x_{k+1} 之间. 如果将这些分数按照分母大小排列，中间分数就是位于相邻的渐近分数中间的一些分数.

所有的第一类最佳逼近都是中间分数，但是并不是所有的中间分数都是第一类最佳逼近.

定理

所有的第一类最佳逼近都是中间分数.

证明

因为 $a_0\le x\le a_0+1$，所以第一类最佳逼近必然位于 $x_{1,0}=a_0$ 与 $x_{0,1}=a_0+1$ 之间. 所有中间分数从小到大可以排列成

$$x_{1,0}<x_{1,1}<\cdots<x_{1,a_2}=x_{3,0}<\cdots<x<\cdots<x_{2,0}=x_{0,a_1}<\cdots<x_{0,1}.$$

同阶的中间分数是连续出现的，而不同阶的中间分数之间又没有间隔. 这意味着，任何位于 $x_{1,0}=a_0$ 与 $x_{0,1}=a_0+1$ 之间的有理数 $\frac{p}{q}$ 必然落在两个同阶的中间分数 $x_{k,t}$ 和 $x_{k,t+1}$ 之间. 不妨设它不是中间分数且小于 x，因而有

$$x_{k,t}<\frac{p}{q}<x_{k,t+1}<x.$$

此时，一方面有

$$\left|x_{k,t}-\frac{p}{q}\right|<|x_{k,t}-x_{k,t+1}|=\frac{1}{((t+1)q_k+q_{k-1})(tq_k+q_{k-1})}.$$

另一方面有

$$\left|x_{k,t}-\frac{p}{q}\right|=\frac{|\,q(tp_k+p_{k-1})-p(tq_k+q_{k-1})|}{q(tq_k+q_{k-1})}\ge\frac{1}{q(tq_k+q_{k-1})}.$$

因此，必然有

$$q>(t+1)q_k+q_{k-1}.$$

也就是说，有理数 $\frac{p}{q}$ 的分母一定大于 $x_{k,t+1}$ 的分母，但是它并不是更好的逼近：

$$\left|x-\frac{p}{q}\right|>|x-x_{k,t+1}|$$

[64] $t=0$ 时，应理解为形式连分数，相当于截断到连分数的倒数第二项.

因此，它不可能是第一类最佳逼近．这就说明，不是中间分数，就不是第一类最佳逼近；亦即所有第一类最佳逼近都是中间分数．

反过来，并不能断言所有的中间分数都是第一类最佳逼近．但是，的确可以给出中间分数成为第一类最佳逼近的条件．

定理

所有渐近分数都是第一类最佳逼近．除此之外，设 $0 < t < a_{k+1}$，则中间分数 $x_{k,t}$ 是第一类最佳逼近，当且仅当 $t > \dfrac{a_{k+1}}{2}$ 或者 $t = \dfrac{a_{k+1}}{2}$ 且 $r_{k+2} > \dfrac{q_k}{q_{k-1}}$．

证明

下面会证明，渐近分数都是第二类最佳逼近，因而必然是第一类最佳逼近．关键在于那些不是渐近分数的中间分数．

前文已经说过，中间分数 $x_{k,t}$ 的分母位于 x_k 和 x_{k+1} 之间，且随着 t 的增加逐渐增大，但是 $x_{k,t}$ 却逐渐接近 x_{k+1}，因而更接近 x．以 $x_{k,t} < x$ 为例，它与相邻的中间分数的相对位置关系满足：

$$x_{k-1} < x_{k,t} < x_{k+1} < x < x_k.$$

因为 x_k 的分母小于 $x_{k,t}$，所以 $x_{k,t}$ 成为第一类最佳逼近的必要条件就是，它比 x_k 更接近 x．这也是充分条件，因为作为渐近分数，没有比 x_k 分母更小但是距离更近的了，而那些比 x_k 分母还要大的中间分数，必然与 $x_{k,t}$ 同阶，但是分母更小，就必然距离 x 更远．对于渐近分数和中间分数的误差，经计算可知

$$\begin{aligned} |x_k - x| &= \frac{1}{q_k(r_{k+1}q_k + q_{k-1})}, \\ |x_{k,t} - x| &= \left|\frac{tp_k + p_{k-1}}{tq_k + q_{k-1}} - \frac{r_{k+1}p_k + p_{k-1}}{r_{k+1}q_k + q_{k-1}}\right| \\ &= \frac{r_{k+1} - t}{(tq_k + q_{k-1})(r_{k+1}q_k + q_{k-1})}. \end{aligned}$$

其中用到了 $r_{k+1} \geq a_{k+1} > t$．因此，$x_{k,t}$ 比 x_k 更接近 x，成为第一类最佳逼近，当且仅当

$$\frac{r_{k+1} - t}{tq_k + q_{k-1}} < \frac{1}{q_k} \iff r_{k+1} < 2t + \frac{q_{k-1}}{q_k}.$$

此时，有三种可能的情况：

1. 如果 $t < \dfrac{a_{k+1}}{2}$，那么 $2t < a_{k+1}$．因为两侧都是整数，所以 $2t \leq a_{k+1} - 1$，故而 $2t + \dfrac{q_{k-1}}{q_k} \leq 2t + 1 \leq a_{k+1} \leq r_{k+1}$．此时，$x_{k,t}$ 不是第一类最佳逼近；
2. 如果 $t > \dfrac{a_{k+1}}{2}$，那么 $2t > a_{k+1}$．因为两侧都是整数，所以 $2t \geq a_{k+1} + 1 > r_{k+1}$．此时，$x_{k,t}$ 是第一类最佳逼近；
3. 如果 a_{k+1} 是偶数，还有第三种情况，即 $t = \dfrac{a_{k+1}}{2}$．上述条件等价于 $\dfrac{1}{r_{k+2}} = r_{k+1} - a_{k+1} < \dfrac{q_{k-1}}{q_k}$，亦即 $r_{k+2} > \dfrac{q_k}{q_{k-1}}$．

所以，如果将实数 x 的所有第一类最佳逼近按照分母自小到大的顺序排列，那么它会根据与 x 的大小关系分成若干段．每一段总是由若干个（可以是零个）连续的同阶的中间分数组成，且总以渐近分数结尾．段内总能保持在实数 x 的一侧，段与段之间则交错排列在 x 两侧．

例子：圆周率 π 的第一类最佳逼近

圆周率 $\pi = [3, 7, 15, 1, 292, \ldots]$，因而它分母最小的前 15 个第一类最佳逼近是：

$$x_0=\frac{3}{1},x_{0,4}=\frac{13}{4},x_{0,5}=\frac{16}{5},x_{0,6}=\frac{19}{6},x_1=\frac{22}{7},$$
$$x_{1,8}=\frac{179}{57},x_{1,9}=\frac{201}{64},x_{1,10}=\frac{223}{71},x_{1,11}=\frac{245}{78},x_{1,12}=\frac{267}{85},$$
$$x_{1,13}=\frac{289}{92},x_{1,14}=\frac{311}{99},x_2=\frac{333}{106},x_3=\frac{355}{113},x_{3,146}=\frac{52163}{16604}.$$

第二类最佳逼近

第二类最佳逼近使用 $|qx-p|$ 来衡量逼近的程度.

自 第二类最佳逼近

对于实数 x 和有理数 $\frac{p}{q}$，如果对于任意的 $\frac{p'}{q'}\neq\frac{p}{q}$ 且 $0<q'\leq q$ 都有

$$|qx-p|<|q'x-p'|,$$

就称有理数 $\frac{p}{q}$ 是实数 x 的 **第二类最佳逼近**（best approximation of the second kind）.

第二类最佳逼近的条件等价于

$$\left|x-\frac{p}{q}\right|<\frac{q'}{q}\left|x-\frac{p'}{q'}\right|.$$

因为 $q'\leq q$，所以第二类最佳逼近的条件比第一类最佳逼近更为严苛.

第二类最佳逼近能且仅能是渐近分数.

定理

所有的第二类最佳逼近一定是渐近分数，所有的渐近分数也一定是第二类最佳逼近.

证明

要证明第一部分, 因为第二类最佳逼近也一定是第一类最佳逼近, 所以只需要证明不是渐近分数的中间分数不能成为第二类最佳逼近就可以了. 为此，设 $x_{k,t}=\frac{p}{q}$ 是中间分数但是不是渐近分数，那么，设 $x_{k,t}<x$，有

$$x_{k-1}<x_{k,t}<x_{k+1}<x<x_k.$$

因为 $x_{k,t}$ 与 x 之间的误差

$$|\,x_{k,t}-x\,|\geq|\,x_{k,t}-x_{k+1}|=\left|\frac{p}{q}-\frac{p_{k+1}}{q_{k+1}}\right|=\frac{|\,pq_{k+1}-p_{k+1}q\,|}{qq_{k+1}}\geq\frac{1}{qq_{k+1}},$$

并利用渐近分数的误差估计，所以总是有

$$|\,qx_{k,t}-p\,|\geq\frac{1}{q_{k+1}}\geq|\,q_kx_k-p_k|,$$

即 $x_{k,t}$ 的逼近程度不优于分母更小的 x_k 逼近程度，所以不可能是第二类最佳逼近.

反过来，要证明第二部分，即每个渐近分数 $x_k=\frac{p_k}{q_k}$ 都是第二类最佳逼近. 这就是要说明，对于所有分数 $\frac{p}{q}$ 且 $q\leq q_k$，都有 $|\,q_kx-p_k|<|\,qx-p\,|$. 不考虑半奇数的情形，则可以假定 $k>0$. 首先，根据渐近分数逼近实数的误差估计，有

$$|\,q_{k-1}x-p_{k-1}|\geq\frac{1}{q_{k-1}+q_k}\geq\frac{1}{q_{k+1}}\geq|\,q_kx-p_k|\,.$$

不等式全部成立等号，当且仅当 $a_{k+1}=1$ 且是连分数的末项．不考虑这样的情形，那么 $x_{k-1}=\frac{p_{k-1}}{q_{k-1}}$ 严格劣于 $x_k=\frac{p_k}{q_k}$.

任取一分数 $\frac{p}{q}\neq x_k$ 且 $0<q\leq q_k$，因为有差分公式 $p_kq_{k-1}-p_{k-1}q_k=(-1)^{k-1}$，所以由 Cramer 法则可知，线性方程组

$$\begin{cases}\lambda p_k+\mu p_{k-1}=p,\\ \lambda q_k+\mu q_{k-1}=q\end{cases}$$

必然存在唯一的整数解 (λ,μ)．如果 $\lambda\mu>0$，那么 $q>|\lambda|q_k\geq q_k$，矛盾．否则，$\lambda\mu\leq 0$，即 λ 和 μ 异号，那么因为 $q_{k-1}x-p_{k-1}$ 和 q_kx-p_k 也异号，就有 $\lambda(q_{k-1}x-p_{k-1})$ 和 $\mu(q_kx-p_k)$ 同号，故而

$$|qx-p|=|\lambda||q_kx-p_k|+|\mu||q_{k-1}x-p_{k-1}|>|q_kx-p_k|.$$

最后的不等号是严格的，因为 x_{k-1} 严格劣于 x_k，且 $\frac{p}{q}\neq x_k$．这就说明，x_k 是第二类最佳逼近．

这个性质表明，渐近分数确实是相当好的丢番图逼近．

渐近分数的判定

第二类最佳逼近提供了判断某个分数是否是渐近分数的充分必要条件. 这说明, 可以通过检查某个分数逼近的相对程度来判断它是否是渐近分数. Legendre 判别法则提供了根据逼近的绝对程度来判断渐近分数的方法. Legendre 判别法的原始表述提供了充分必要条件, 但是它的形式并不实用. 本节提供了 Legendre 判别法的简化版本，并说明它并没有漏掉太多的渐近分数.

定理（Legendre）

对于实数 x 与分数 $\frac{p}{q}$，如果有

$$\left|x-\frac{p}{q}\right|<\frac{1}{2q^2}$$

那么 $\frac{p}{q}$ 一定是 x 的渐近分数.

证明

设 $\epsilon\in\{-1,1\}$ 和 $\theta\in(0,1/2)$ 是使得

$$x-\frac{p}{q}=\frac{\epsilon\theta}{q^2}$$

成立的常数．将有理数 $\frac{p}{q}$ 展开成连分数 $[a_0,a_1,\ldots,a_n]$．此处，有理数有两种连分数表示，其中的 n 恰好相差一，所以可以取连分数表示使得 $(-1)^n=\epsilon$，并记这个连分数表示的渐近分数为 $\frac{p_k}{q_k}$．设实数 ω 满足

$$x=\frac{\omega p_n+p_{n-1}}{\omega q_n+q_{n-1}}.$$

那么，必然有

$$\frac{\epsilon\theta}{q^2}=x-\frac{p}{q}=x-\frac{p_n}{q_n}=\frac{p_{n-1}q_n-p_nq_{n-1}}{(\omega q_n+q_{n-1})q_n}=\frac{(-1)^n}{(\omega q_n+q_{n-1})q_n}.$$

故而，有

$$\theta=\frac{q_n}{\omega q_n+q_{n-1}}.$$

这说明

$$\omega=\frac{1}{\theta}-\frac{q_{n-1}}{q_n}>1.$$

将 ω 也展成连分数 $[b_0,b_1,...]$，则

$$x=\frac{\omega p_n+p_{n-1}}{\omega q_n+q_{n-1}}=[a_0,a_1,...,a_n,\omega]=[a_0,a_1,...,a_n,b_0,b_1,...].$$

这是合法的简单连分数，所以 $\frac{p}{q}$ 就是 x 的渐近分数.

这个证明实际说明 $\frac{p}{q}$ 成为渐近分数的充分必要条件是上述证明中的 $\omega>1$，这正是 Legendre 判别法的原始形式.

这个判别法说明，只要逼近的程度足够好，就一定是渐近分数. 下一个定理说明，这样好的渐近分数足够多：至少有一半的渐近分数都符合这个条件.

定理（Vahlen）

实数 x 的相邻两个渐近分数中至少有一个满足

$$\left|x-\frac{p}{q}\right|<\frac{1}{2q^2}.$$

证明

假设不然. 存在实数 x 有两个相邻的渐近分数 x_k 和 x_{k+1} 满足

$$\left|x-\frac{p_k}{q_k}\right|\geq\frac{1}{2q_k^2},\left|x-\frac{p_{k+1}}{q_{k+1}}\right|\geq\frac{1}{2q_{k+1}^2}.$$

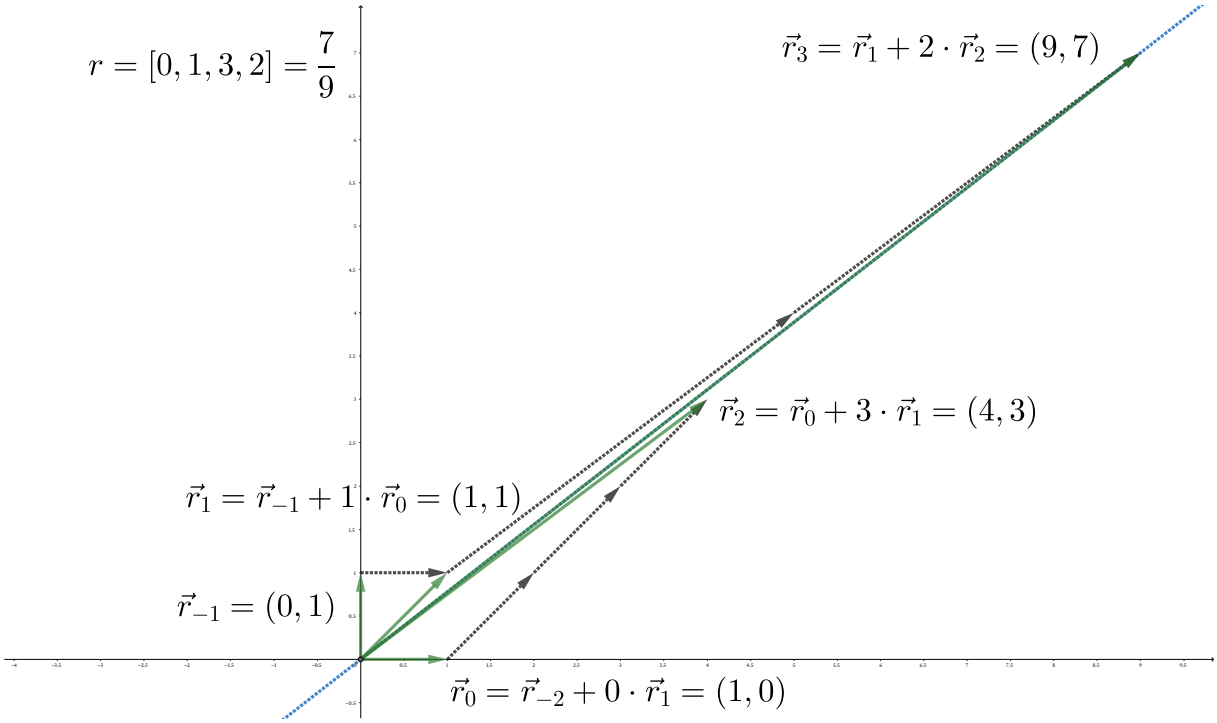
因为 x 位于 x_k 和 x_{k+1} 之间，所以

$$\frac{1}{2q_k^2}+\frac{1}{2q_{k+1}^2}\leq\left|x-\frac{p_k}{q_k}\right|+\left|x-\frac{p_{k+1}}{q_{k+1}}\right|=\left|\frac{p_k}{q_k}-\frac{p_{k+1}}{q_{k+1}}\right|=\frac{1}{q_kq_{k+1}}.$$

这说明 $q_k=q_{k+1}$. 故而必然有 $k=0$ 且 $a_1=1$. 此时前两个渐近分数是 $x_0=a_0$ 和 $x_1=a_0+1$. 故而命题的唯一反例是半奇数，按照前文的说明，本文不考虑这种情形.

几何解释

连分数理论有着优美的几何解释.



如图所示，对于实数 $\xi > 0$，直线 $y = \xi x$ 将第一象限（包括 x 和 y 坐标轴上的点但不包括原点，下同）上的整点（lattice point）分成上下两部分．对于有理数 ξ 的情形，直线 $y = \xi x$ 上的点既算作直线上方的点，又算作直线下方的点．考虑这两部分的点的凸包．那么，奇数项渐近分数是上半部分的凸包的顶点，偶数项渐近分数是下半部分的凸包的顶点．凸包上两个相邻顶点之间的连线上的整点就是中间分数．图中展示了 $\xi = \frac{9}{7}$ 的渐近分数和中间分数（灰点）．

前文关于连分数的大部分结论都有相应的几何解释：

几何解释

- 每个分数 $\nu = \frac{p}{q}$ 都对应着第一象限内的一个整点 $\vec{v} = (q,p)$，分数的大小对应着与原点连线的斜率．
- 直线 $y = \xi x$ 的方向向量是 $\vec{\xi} = (1,\xi)$．利用 叉积→9.14.3@p1711 $(x_1,y_1) \times (x_2,y_2) = x_1y_2 - x_2y_1$ 的概念，可以通过 $\vec{\xi} \times \vec{v} = p - q\xi$ 的正负判断点在直线上方还是下方．因而，在直线上方的点就对应着大于等于 ξ 的分数，在直线下方的点就对应着小于等于 ξ 的分数．叉积的绝对值 $|\vec{\xi} \times \vec{v}|$ 正比于点 $\vec{v}$ 与直线 $y = \xi x$ 的距离
  $$\frac{|\ p - q\xi\ |}{\sqrt{1+\xi^2}},$$
  对应着分数 ν 对实数 ξ 的逼近程度．
- 将渐近分数 $\xi_k = \frac{p_k}{q_k}$ 对应的点记作 $\vec{\xi}_k = (q_k,p_k)$，则递推公式就可以写作
  $$\vec{\xi}_k = a_k\vec{\xi}_{k-1} + \vec{\xi}_{k-2}.$$
  递归的起点是 $\vec{\xi}_{-2} = (1,0)$ 和 $\vec{\xi}_{-1} = (0,1)$．
- 对于整数 t，如果 $0 \le t \le a_k$，那么点
  $$\vec{\xi}_{k-1,t} = t\vec{\xi}_{k-1} + \vec{\xi}_{k-2}$$
  就落在连结点 $\vec{\xi}_{k-2}$ 和点 $\vec{\xi}_k$ 的线段上．它们对应着中间分数 $\xi_{k-1,t}$．
- 利用几何的方法可以构造出所有的渐近分数和中间分数．从点 $\vec{\xi}_{-2} = (1,0)$ 和点 $\vec{\xi}_{-1} = (0,1)$ 开始，两个点位于直线 $y = \xi x$ 两侧，这意味着 $\vec{\xi} \times \vec{\xi}_{-2}$ 和 $\vec{\xi} \times \vec{\xi}_{-1}$ 符号相反．将 $\vec{\xi}_{-1}$ 按照向量的加

法添加到 $\vec{\xi}_{-2}$ 上，直到无法继续添加而不穿过直线 $y=\xi x$ 为止，将结果记作 $\vec{\xi}_0$，此时仍与 $\vec{\xi}_{-1}$ 不同侧．再将 $\vec{\xi}_0$ 添加到 $\vec{\xi}_{-1}$ 上，直到无法继续添加而不穿过直线 $y=\xi x$ 为止，将结果记作 $\vec{\xi}_1$，此时仍与 $\vec{\xi}_0$ 不同侧．这个过程可以一直持续到无穷，除非在有限步内某个 $\vec{\xi}_n$ 就恰好落在直线 $y=\xi x$ 上．后者意味着向量 $\vec{\xi}$ 与 $\vec{\xi}_n$ 共线，即 $\xi=\dfrac{p_n}{q_n}$ 为有理点．这个过程就可以得到前面示意图中的图形．Boris Delaunay 将这个过程形象地称为鼻子拉伸算法（nose-stretching algorithm）[65]．

- 如果需要快速计算每一步将 $\vec{\xi}_{k-1}$ 添加到 $\vec{\xi}_{k-2}$ 需要的次数，可以借助叉积．因为 $\vec{\xi}\times\vec{\xi}_{k-1}$ 与 $\vec{\xi}\times\vec{\xi}_{k-2}$ 符号相反，所以如果记 $\vec{\xi}_{k-1,t}=t\vec{\xi}_{k-1}+\vec{\xi}_{k-2}$ 为向 $\vec{\xi}_{k-2}$ 添加 t 次 $\vec{\xi}_{k-1}$ 得到的结果，则 $\vec{\xi}\times\vec{\xi}_{k-1,t}=t(\vec{\xi}\times\vec{\xi}_{k-1})+(\vec{\xi}\times\vec{\xi}_{k-2})$ 不改变符号，就意味着没有穿过直线．在不变号之前，$\vec{\xi}\times\vec{\xi}_{k-1,t}$ 的绝对值会逐渐下降．记

  $$r_k=\left|\frac{\vec{\xi}\times\vec{\xi}_{k-2}}{\vec{\xi}\times\vec{\xi}_{k-1}}\right|=-\frac{\vec{\xi}\times\vec{\xi}_{k-2}}{\vec{\xi}\times\vec{\xi}_{k-1}}.$$

  那么，最大可以下降的次数就是

  $$a_k=\lfloor r_k\rfloor=\left\lfloor\left|\frac{q_{k-1}\xi-p_{k-1}}{q_{k-2}\xi-p_{k-2}}\right|\right\rfloor.$$

  这就是连分数展开的第 k 项．而且，r_k 就是连分数展开的余项，它满足关系式：

  $$r_k=-\frac{q_{k-1}\xi-p_{k-1}}{q_{k-2}\xi-p_{k-2}}\iff\xi=\frac{p_{k-1}r_k+p_{k-2}}{q_{k-1}r_k+q_{k-2}}.$$

  这就是连分数关系式 $\xi=[a_0,a_1,\ldots,a_{k-1},r_k]$．
- 因为每次添加向量造成的 $\vec{\xi}\times\vec{\xi}_{k-1,t}$ 的变化的步长都是 $|\vec{\xi}\times\vec{\xi}_{k-1}|$，所以最后剩余的距离 $|\vec{\xi}\times\vec{\xi}_k|$ 必然严格小于 $|\vec{\xi}\times\vec{\xi}_{k-1}|$．这说明，渐近分数的逼近程度（由 $|qx-p|$ 衡量）是随着 k 的增加严格更优的．
- 利用叉积的运算法则，有

  $$\vec{\xi}_k\times\vec{\xi}_{k+1}=\vec{\xi}_k\times(a_{k+1}\vec{\xi}_k+\vec{\xi}_{k-1})=\vec{\xi}_k\times\vec{\xi}_{k-1}=-\vec{\xi}_{k-1}\times\vec{\xi}_k.$$

  归纳可知

  $$\vec{\xi}_k\times\vec{\xi}_{k+1}=(-1)^{k+2}\vec{\xi}_{k-2}\times\vec{\xi}_{k-1}=(-1)^k.$$

  这就是渐近分数的差分公式 $p_{k+1}q_k-p_kq_{k+1}=(-1)^k$．
- 上下两个凸包之间的面积可以剖分成若干个（可能是无穷多个）三角形，其中每个三角形的顶点分别是 $\vec{\xi}_{k-2}$、$\vec{\xi}_k$ 和 $\vec{0}$．这样的三角形的面积是

  $$\frac{1}{2}|\vec{\xi}_{k-2}\times\vec{\xi}_k|=\frac{1}{2}|\vec{\xi}_{k-2}\times(a_k\vec{\xi}_{k-1}+\vec{\xi}_{k-2})|=\frac{a_k}{2}|\vec{\xi}_{k-2}\times\vec{\xi}_{k-1}|=\frac{a_k}{2}.$$

  根据 Pick 定理→12.5@p3098，这意味着如果设 I 和 B 分别为三角形内部和边界上的整点个数，则

  $$I+\frac{B}{2}-1=\frac{a_k}{2}.$$

  又已知三角形边界上已经有了 $\{\vec{0}\}\cup\{\vec{\xi}_{k-1,t}:0\le t\le a_k\}$ 这共计 a_k+2 个整点．这说明，就一定有 $I=0$ 和 $B=a_k+2$．因而，三角形的边上没有更多的整点，三角形内部也没有整点．也就是说，q_k 和 p_k 是既约的，中间分数是连结 $\vec{\xi}_{k-2}$ 和 $\vec{\xi}_k$ 的边上的全部整点，且第一象限的所有整点都在上下两个凸包内．

这样得到的上下两个凸包称为 Klein 多边形．在高维空间内也可以做类似定义，得到 Klein 多面体[6]（Klein polyhedron），它可以将连分数的概念推广到高维空间内．

连分数的树

主条目：Stern–Brocot 树与 Farey 序列→9.11.31@p1482

[65]此说法并非专业术语．可能转译自俄文文献 ЦЕПНЫЕ ДРОБИ[5]，在 Алгоритм «вытягивания носов» 一节．

Stern–Brocot 树是存储了所有位于 $[0, \infty]$ 之间的分数的 二叉搜索树→10.15.1@p2213．有限连分数实际上编码了 Stern–Brocot 树上从根到某个分数所在位置的路径．也就是说，有理数 x 的连分数表示 $[a_0, a_1, ..., a_{n-1}, 1]$ 意味着从树根 $\frac{1}{1}$ 开始，需要先向右子节点移动 a_0 次，再向左子节点移动 a_1 次，交替方向移动，直到向某个方向移动了 a_{n-1} 次为止．应当注意，此处只能使用末尾为 1 的连分数表示．

将连分数表示理解为 Stern–Brocot 树上的路径，可以得到比较连分数大小的算法．

连分数大小比较

给定连分数 $\alpha = [\alpha_0, \alpha_1, ..., \alpha_n]$ 和 $\beta = [\beta_0, \beta_1, ..., \beta_m]$，比较两者大小．

解答

首先将两个连分数表示都转化成末尾是 1 的形式．不妨设题目所给的已经是这样形式的连分数，即 $\alpha_n = \beta_m = 1$．因为偶数位置（下标从 0 开始）是向右移动的步数，奇数位置是向左移动的步数，所以，$\alpha < \beta$ 当且仅当按照 字典序→8.2@p890 比较时，有

$$(\alpha_0, -\alpha_1, \alpha_2, ..., (-1)^{n-1}\alpha_{n-1}, 0, ...) < (\beta_0, -\beta_1, \beta_2, ..., (-1)^{m-1}\beta_{m-1}, 0, ...).$$

相较于连分数表示，交替地添加正负号，删去末尾的 1，并且长度不足的位置用 0 补齐．

- C++

```cpp
// Expand [..., n] to [..., n-1, 1] if needed.
void expand(std::vector<int>& a) {
  if (a.size() == 1 || a.back() > 1) {
    --a.back();
    a.push_back(1);
  }
}

// Check if a is smaller than b.
bool less_than(std::vector<int> a, std::vector<int> b) {
  expand(a);
  expand(b);
  for (int i = 0; i < a.size() - 1 || i < b.size() - 1; ++i) {
    int d = (i < a.size() - 1 ? a[i] : 0) - (i < b.size() - 1 ? b[i] : 0);
    if (i & 1) d = -d;
    if (d < 0) {
      return true;
    } else if (d > 0) {
      return false;
    }
  }
  return false;
}

```

- Python

```python
# Expand [..., n] to [..., n-1, 1] if needed.
def expand(a):
    if a[-1] != 1 or len(a) == 1:
        a[-1] -= 1
        a.append(1)
```

```python
    return a


# Check if a is smaller than b.
def less_than(a, b):
    a = expand(a)
    b = expand(b)
    a = [(-1) ** i * a[i] for i in range(len(a))]
    b = [(-1) ** i * b[i] for i in range(len(b))]
    return a < b


```

最佳内点

对于 $\frac{0}{1} \le \frac{p_0}{q_0} < \frac{p_1}{q_1} \le \frac{1}{0}$，求使得 $\frac{p_0}{q_0} < \frac{p}{q} < \frac{p_1}{q_1}$ 成立且 (q,p) 最小的有理数 $\frac{p}{q}$.

解答

因为 Stern–Brocot 树既是 $[0,\infty]$ 中的分数的二叉搜索树，又是二元组 (q,p) 的 笛卡尔树 →10.15.6@p2311，所以题意几乎可以转化为求 Stern–Brocot 树上两个点的 LCA（最近公共祖先）．但是，LCA 只能处理闭区间内的情形，LCA 可能是端点本身．为了避免额外的讨论，可以首先构造出 $\frac{p_0}{q_0} + \varepsilon$ 和 $\frac{p_1}{q_1} - \varepsilon$，再计算 LCA．在已经通过连分数计算出根到节点的路径的情况下，LCA 只要取最长的公共路径即可．

要构造出 $x \pm \varepsilon$，只需要在节点 x 处首先向右（左）移动一次，再向左（右）移动 ∞ 次即可．转化成连分数的语言，对于分数 $x = [a_0, a_1, ..., a_{n-1}, 1]$，可以知道 $x \pm \varepsilon$ 必然是 $[a_0, a_1, ..., a_{n-1} + 1, \infty]$ 和 $[a_0, a_1, ..., a_{n-1}, 1, \infty]$，因而只需要比较这两个连分数，将较大（小）的定义为 $x \pm \varepsilon$.

- C++

```cpp
// Get X +- EPSILON.
auto pm_eps(std::vector<int> a) {
  constexpr int inf = 0x3f3f3f3f;
  // Deal with empty continued fraction for 1/0.
  if (a.empty()) {
    a.emplace_back(inf);
  }
  auto b = a;
  expand(b);
  a.emplace_back(inf);
  b.emplace_back(inf);
  return less_than(a, b) ? std::make_pair(a, b) : std::make_pair(b, a);
}

// Find the lexicographically smallest (q, p)
//   such that p0/q0 < p/q < p1/q1.
auto middle(int p0, int q0, int p1, int q1) {
  auto a0 = pm_eps(fraction(p0, q0)).second;
  auto a1 = pm_eps(fraction(p1, q1)).first;
  std::vector<int> a;
```

```cpp
  for (int i = 0; i < a0.size() || i < a1.size(); ++i) {
    if (a0[i] == a1[i]) {
      a.emplace_back(a0[i]);
    } else {
      a.emplace_back(std::min(a0[i], a1[i]) + 1);
      break;
    }
  }
  auto pq = convergents(a);
  return std::make_pair(pq.first.back(), pq.second.back());
}

```

- Python

```python
# Get X +- EPSILON.
def pm_eps(a):
    # Deal with empty continued fraction for 1/0.
    if not a:
        a.append(float("inf"))
    b = expand(a.copy())
    a.append(float("inf"))
    b.append(float("inf"))
    return (a, b) if less_than(a, b) else (b, a)


# Find the lexicographically smallest (q, p)
#   such that p0/q0 < p/q < p1/q1.
def middle(p0, q0, p1, q1):
    a0 = pm_eps(fraction(p0, q0))[1]
    a1 = pm_eps(fraction(p1, q1))[0]
    a = []
    for i in range(min(len(a0), len(a1))):
        if a0[i] == a1[i]:
            a.append(a0[i])
        else:
            a.append(int(min(a0[i], a1[i])) + 1)
            break
    p, q = convergents(a)
    return p[-1], q[-1]


```

GCJ 2019, Round 2 - New Elements: Part 2[7]

给定 N 个正整数对 (C_i, J_i)，求正整数对 (x, y) 使得 $\{C_i x + J_i y\}$ 严格递增．在所有符合要求的数对中，输出字典序最小的一对．

解答

不妨设 $A_i = C_i - C_{i-1}$ 和 $B_i = J_i - J_{i-1}$．问题转化为求 (x, y) 使得所有 $A_i x + B_i y$ 都是正数．这些数对可以分为四种情形：

1. $A_i, B_i > 0$ 的情形可以忽略，因为已经假设 $(x, y) > 0$；
2. $A_i, B_i \le 0$ 的情形直接输出「IMPOSSIBLE」；
3. $A_i > 0, B_i \le 0$ 的情形相当于约束 $\frac{y}{x} < \frac{A_i}{-B_i}$；
4. $A_i \le 0, B_i > 0$ 的情形相当于约束 $\frac{y}{x} > \frac{-A_i}{B_i}$.

因此，取 $\frac{p_0}{q_0}$ 是第四种情形中最大的 $\frac{-A_i}{B_i}$，再取 $\frac{p_1}{q_1}$ 是第三种情形中最小的 $\frac{A_i}{-B_i}$. 原问题就变成了找到字典序最小的 (q, p) 使得 $\frac{p_0}{q_0} < \frac{p}{q} < \frac{p_1}{q_1}$ 成立.

- C++

```cpp
void solve() {
  int n;
  std::cin >> n;
  std::vector<int> C(n), J(n);
  // p0/q0 < y/x < p1/q1
  int p0 = 0, q0 = 1, p1 = 1, q1 = 0;
  bool fail = false;
  for (int i = 0; i < n; ++i) {
    std::cin >> C[i] >> J[i];
    if (i) {
      int A = C[i] - C[i - 1];
      int B = J[i] - J[i - 1];
      if (A <= 0 && B <= 0) {
        fail = true;
        break;
      } else if (B > 0 && A < 0) {  // y/x > (-A)/B if B > 0
        if ((-A) * q0 > p0 * B) {
          p0 = -A;
          q0 = B;
        }
      } else if (B < 0 && A > 0) {  // y/x < A/(-B) if B < 0
        if (A * q1 < p1 * (-B)) {
          p1 = A;
          q1 = -B;
        }
      }
    }
  }
  if (fail || p0 * q1 >= p1 * q0) {
    printf("IMPOSSIBLE\n");
  } else {
    auto pq = middle(p0, q0, p1, q1);
    printf("%d %d\n", pq.first, pq.second);
  }
}

```

- Python

```python
def solve():
    n = int(input())
    C = [0] * n
    J = [0] * n
```

```
    # p0/q0 < y/x < p1/q1
    p0, q0 = 0, 1
    p1, q1 = 1, 0
    fail = False
    for i in range(n):
        C[i], J[i] = map(int, input().split())
        if i > 0:
            A = C[i] - C[i - 1]
            B = J[i] - J[i - 1]
            if A <= 0 and B <= 0:
                fail = True
                break
            elif B > 0 and A < 0:  # y/x > (-A)/B if B > 0
                if (-A) * q0 > p0 * B:
                    p0, q0 = -A, B
            elif B < 0 and A > 0:  # y/x < A/(-B) if B < 0
                if A * q1 < p1 * (-B):
                    p1, q1 = A, -B
    if fail or p0 * q1 >= p1 * q0:
        return "IMPOSSIBLE"
    p, q = middle(p0, q0, p1, q1)
    return str(p) + " " + str(q)


```

想要了解更多 Stern–Brocot 树的性质和应用，可以参考其主条目页面.

分式线性变换

和连分数相关的另一个重要概念是所谓的分式线性变换.

分式线性变换

分式线性变换（fractional linear transformation）是指函数 $L : \mathbf{R} \to \mathbf{R}$，使得

$$L(x) = \frac{ax + b}{cx + d},$$

其中，$a, b, c, d \in \mathbf{R}$ 且 $ad - bc \neq 0$.

关于条件 $ad - bc \neq 0$

容易验证，当 $ad - bc = 0$ 时，函数可能没有定义或者是常函数.

分式线性变换有如下性质：

分式线性变换的性质

设 L_1, L_2, L_3 是分式线性变换，并记 L_i 的系数形成的矩阵为

$$M_i = \begin{pmatrix} a_i & b_i \\ c_i & d_i \end{pmatrix}$$

则它们有如下性质：[66]

[66]这些性质表明，全体分式线性变换的群同构于 射影线性群[8] $PGL_2(\mathbf{R})$.

1. 分式线性变换的复合 $L_1 \circ L_2$ 和逆变换 L_1^{-1} 仍然是分式线性变换，即全体分式线性变换构成群→9.16.1@p1805；
2. 分式线性变换在系数同乘以非零常数后保持不变，即对于任意 $\lambda \neq 0$，如果 $M_2 = \lambda M_1$，则 $L_2 = L_1$；
3. 分式线性变换的复合的系数矩阵，对应着系数矩阵的乘积，即如果 $M_1 M_2 = M_3$，则 $L_1 \circ L_2 = L_3$；
4. 分式线性变换的逆变换的系数矩阵，对应着系数矩阵的逆矩阵，即如果 $M_1^{-1} = M_2$，则 $L_1^{-1} = L_2$.

证明

此处仅提供分式线性变换的复合和逆变换的形式．得到这个形式后，所有性质都是容易验证的.

分式线性变换 L_1 和 L_2 的复合：

$$L_1 \circ L_2 = \frac{a_1 \dfrac{a_2 x + b_2}{c_2 x + d_2} + b_1}{c_1 \dfrac{a_2 x + b_2}{c_2 x + d_2} + d_1} = \frac{(a_1 a_2 + b_1 c_2)x + (a_1 b_2 + b_1 d_2)}{(c_1 a_2 + d_1 c_2)x + (c_1 b_2 + d_1 d_2)}.$$

分式线性变换 $L_1(x)$ 的逆变换：

$$y = L_1(x) = \frac{a_1 x + b_1}{c_1 x + d_1} \iff x = L_1^{-1}(y) = \frac{d_1 y - b_1}{-c_1 y + a_1}.$$

有限连分数 $[a_0, a_1, ..., a_n]$ 可以看做是一系列分式线性变换复合的结果．设

$$L_i(x) = \frac{a_i x + 1}{x} = [a_i, x].$$

那么，有限连分数

$$[a_0, a_1, ..., a_n] = L_0 \circ L_1 \circ \cdots \circ L_n(\infty).$$

其中，分式线性变换 $L(x) = \dfrac{ax + b}{cx + d}$ 变换在 $x = \infty$ 处的取值是 $\dfrac{a}{c}$，这是函数在 $x \to \pm\infty$ 时的极限值.

对于一般的连分数，设实数 x 的余项为 r_{k+1}，即 $x = [a_0, ..., a_k, r_{k+1}]$，则有

$$x = L_0 \circ L_1 \circ \cdots \circ L_k(r_{k+1}) = \frac{p_k r_{k+1} + p_{k-1}}{q_k r_{k+1} + q_{k-1}}.$$

这同时也给出了分式线性变换 $L_0 \circ L_1 \circ \cdots \circ L_k$ 的形式.

当然也可以直接验证这个表达式．最开始的时候是

$$x = \frac{x + 0}{0x + 1} = \frac{p_{-1} x + p_{-2}}{q_{-1} x + q_{-2}}.$$

随后，如果 $L_0 \circ L_1 \circ \cdots \circ L_{k-1}$ 具有

$$\frac{p_{k-1} x + p_{k-2}}{q_{k-1} x + q_{k-2}}$$

的形式，那么根据分式线性变换的复合公式，有

$$L_0 \circ L_1 \circ \cdots \circ L_{k-1} \circ L_k = \frac{(p_{k-1} a_k + p_{k-2})x + p_{k-1}}{(q_{k-1} a_k + q_{k-2})x + q_{k-1}} = \frac{p_k x + p_{k-1}}{q_k x + q_{k-1}}.$$

这就可以归纳地得到了上述形式．分式线性变换也提供了递推公式和初值条件的另一个角度的理解.

DMOPC '19 Contest 7 P4 - Bob and Continued Fractions[9]

给定正整数数组 $a_1, ..., a_n$ 和 m 组查询，每次查询给定 $l \leq r$，并要求计算 $[a_l, ..., a_r]$ 的值.

解答

将连分数理解为一列分式线性变换的复合在 $x = \infty$ 处取值的结果，只需要能够多次查询一段分式线性变换的复合即可. 因为每个分式线性变换都可以取逆, 所以可以预处理前缀和再用差分的方法查询，复杂度为 $O(n + m)$ 的；如果需要修改，也可以用线段树等结构存储.

- C++

```cpp
#include <algorithm>
#include <iostream>
#include <tuple>
#include <vector>

constexpr int M = 1e9 + 7;

// FLTs. Essentially 2x2 matrix.
struct FracLinearTrans {
  int mat[4];

  FracLinearTrans() : mat{} {}

  FracLinearTrans(int x) : mat{x, 1, 1, 0} {}

  FracLinearTrans(int a, int b, int c, int d) : mat{a, b, c, d} {}

  FracLinearTrans operator*(const FracLinearTrans& rhs) const {
    return FracLinearTrans(
        ((long long)mat[0] * rhs.mat[0] + (long long)mat[1] * rhs.mat[2]) % M,
        ((long long)mat[0] * rhs.mat[1] + (long long)mat[1] * rhs.mat[3]) % M,
        ((long long)mat[2] * rhs.mat[0] + (long long)mat[3] * rhs.mat[2]) % M,
        ((long long)mat[2] * rhs.mat[1] + (long long)mat[3] * rhs.mat[3]) % M);
  }

  FracLinearTrans inv() const {
    return FracLinearTrans(mat[3], M - mat[1], M - mat[2], mat[0]);
  }
};

int main() {
  int n, q;
  std::cin >> n >> q;
  // Get prefix sum of FLTs.
  std::vector<FracLinearTrans> ps(1, {1, 0, 0, 1});
  ps.reserve(n + 1);
  for (int i = 1; i <= n; ++i) {
    int a;
    std::cin >> a;
    ps[i] = ps[i - 1] * FracLinearTrans(a);
  }
  // Query.
```

```
  for (; q; --q) {
    int l, r;
    std::cin >> l >> r;
    // Difference.
    auto res = ps[l - 1].inv() * ps[r];
    int u = res.mat[0], d = res.mat[2];
    // Correct signs.
    if (!(l & 1)) {
      if (u) u = M - u;
      if (d) d = M - d;
    }
    printf("%d %d\n", u, d);
  }
  return 0;
}

```

- Python

```
# PYTHON IS TOO SLOW TO PASS THIS PROBLEM.
# JUST FOR REFERENCE.
M = 10**9 + 7


def mul(a, b):
    return (
        (a[0] * b[0] + a[1] * b[2]) % M,
        (a[0] * b[1] + a[1] * b[3]) % M,
        (a[2] * b[0] + a[3] * b[2]) % M,
        (a[2] * b[1] + a[3] * b[3]) % M,
    )


def inv(a):
    return (a[3], M - a[1], M - a[2], a[0])


n, q = map(int, input().split())
ps = [(1, 0, 0, 1)]
# Get presum.
for a in map(int, input().split()):
    ps.append(mul(ps[-1], (a, 1, 1, 0)))
for _ in range(q):
    l, r = map(int, input().split())
    res = mul(inv(ps[l - 1]), ps[r])
    u, d = res[0], res[2]
    if l % 2 == 0:
        if u:
            u = M - u
        if d:
            d = M - d
    print(u, d)

```

连分数的四则运算

利用分式线性变换，可以完成连分数的四则运算．这个算法最早由 Gosper 提出．

算法的基石是计算连分数的分式线性变换．本节以有限连分数为例，但是因为算法每输出一位，只需要读入有限多个连分数的项，所以对于无限连分数也是适用的，而且可以算到任意精度．结合前文的连分数比较算法，可以精确地比较任意精度的实数差异．

连分数的分式线性变换

给定分式线性变换 $L(x)=\dfrac{ax+b}{cx+d}$ 和连分数 $\alpha=[\alpha_0,\alpha_1,...,\alpha_n]$，求 $\beta=L(\alpha)$ 的连分数表示 $[\beta_0,\beta_1,...,\beta_m]$.

解答

算法的基本思路就是逐个确定 β_i 的值．记

$$L_\gamma(x)=\gamma+\frac{1}{x}=\frac{\gamma x+1}{x}.$$

因为连分数

$$L(\alpha)=L\circ L_{\alpha_0}\circ L_{\alpha_1}\circ\cdots\circ L_{\alpha_n}(\infty),$$

所以，可以向 L 通过逐步复合 L_{α_k} 的方式计算 $L(\alpha)$ 的大小．但是，如果是希望得到 $L(\alpha)$ 的连分数表示，那么并不需要完全计算 $L(\alpha)$ 的值再求出连分数表示．可以在复合 L_{α_i} 的过程中就能判断 $\beta_0,\beta_1,...$ 的值．

比如，假设当前计算到

$$L\circ L_{\alpha_0}\circ L_{\alpha_1}\circ\cdots\circ L_{\alpha_k}(x)=\frac{a_kx+b_k}{c_kx+d_k}$$

且 c_k,d_k 同号．那么，$L\circ L_{\alpha_0}\circ L_{\alpha_1}\circ\cdots\circ L_{\alpha_k}(x)$ 在 $[0,\infty]$ 上单调，且必然取值在 $\dfrac{a_k}{c_k}$ 和 $\dfrac{b_k}{d_k}$ 之间．所以，如果

$$\left\lfloor\frac{a_k}{c_k}\right\rfloor=\left\lfloor\frac{b_k}{d_k}\right\rfloor,$$

就可以确定它就是 $L\circ L_{\alpha_0}\circ L_{\alpha_1}\circ\cdots\circ L_{\alpha_k}(x)$ 的整数部分 β_0．此时，在左侧复合 $L_{\beta_0}^{-1}$ 就可以得到

$$L_{\beta_0}^{-1}\circ L\circ L_{\alpha_0}\circ L_{\alpha_1}\circ\cdots\circ L_{\alpha_k}.$$

此时，继续添加 $L_{\alpha_{k+1}},L_{\alpha_{k+2}},...$ 就可以确定新的整数部分，即 β_1．这样计算下去，直到确定出所有的 β_j 的值．

算法要求 c 和 d 同号，是因为要保证函数的不连续点不在 $[0,\infty]$ 范围内．这总是可能的，因为简单连分数的定义要求（除了 α_0 外的）系数都是正整数．由此可以证明，必然在有限步内成立 c 和 d 同号，且将在之后一直保持同号．

具体实现时，只需要维护当前的分式线性变换的系数矩阵 $\begin{pmatrix}a&b\\c&d\end{pmatrix}$ 并检查 c 和 d 是否同号以及 $\dfrac{a}{c}$ 和 $\dfrac{b}{d}$ 是否有相同的整数部分．右复合 L_{α_i} 时，就可以得到 $\begin{pmatrix}a\alpha_i+b&a\\c\alpha_i+d&c\end{pmatrix}$．如果两者整数部分相同为 β_j，则在结果的连分数内添加 β_j，并且左复合 $L_{\beta_j}^{-1}$，这相当于计算 $\begin{pmatrix}c&d\\a\bmod c&b\bmod d\end{pmatrix}$.

连分数的分式线性变换已经可以用于计算分数和连分数的四则运算问题：

$$\frac{p}{q}\pm x=\frac{\pm qx+p}{0x+q},\frac{p}{q}x=\frac{px+0}{0x+q},\frac{p}{q}/x=\frac{0x+p}{qx+0}.$$

对于一般的连分数之间的四则运算，需要用到双分式线性变换：

$$x+y=\frac{0xy+x+y+0}{0xy+0x+0y+1},xy=\frac{1xy+0x+0y+0}{0xy+0x+0y+1},\frac{x}{y}=\frac{0xy+x+0y+0}{0xy+0x+y+0}.$$

连分数的双分式线性变换

给定双分式线性变换 $L(x,y)=\dfrac{axy+bx+cy+d}{exy+fx+gy+h}$ 和连分数 $\alpha=[\alpha_0,\alpha_1,...,\alpha_n]$ 和 $\beta=[\beta_0,\beta_1,...,\beta_m]$，求 $\gamma=L(\alpha,\beta)$ 的连分数表示 $[\gamma_0,\gamma_1,...,\gamma_\ell]$.

解答

与单变量的分式线性变换的情形类似，要确定整数部分只需要保证当前的分式线性变换在 $(x,y)\in[0,\infty]\times[0,\infty]$ 内的整数部分保持不变，即 e,f,g,h 皆同号，且

$$\left\lfloor\frac{a}{e}\right\rfloor=\left\lfloor\frac{b}{f}\right\rfloor=\left\lfloor\frac{c}{g}\right\rfloor=\left\lfloor\frac{d}{h}\right\rfloor.$$

右复合要替换成计算 $L(x,y)\mapsto L(L_{\alpha_i}(x),y)$ 和 $L(x,y)\mapsto L(x,L_{\beta_j}(y))$，这同样表示成系数的线性变换．左复合则和单变量的情形完全一致，只需要计算取模就可以了．

相较于单变量的情形，双变量的情形需要决定要先复合 L_{α_i} 还是 L_{β_j}．因为复合的顺序与最后的结果无关，所以可以自由选择复合顺序，比如交替地复合 L_{α_i} 和 L_{β_j}．或者采用经验法则，优先复合比值差距更大的维度：如果 $\left|\dfrac{b}{f}-\dfrac{d}{h}\right|>\left|\dfrac{c}{g}-\dfrac{d}{h}\right|$，那么就先复合 L_{α_i}；否则，就先复合 L_{β_j}.

循环连分数

类似于循环小数的概念，如果连分数的系数形成了循环，就称为循环连分数.

循环连分数

设连分数 $x=[a_0,a_1,a_2,...]$，且存在自然数 K 和正整数 L 使得对于任何 $k\geq K$，都有 $a_k=a_{k+L}$，就称连分数 x 为 **循环连分数**（periodic continued fraction）．满足这个条件的最小的 L 称为它的最小正周期，而在连分数中重复出现的 $a_k,...,a_{k+L-1}$ 序列就称为它的循环节．利用循环节，循环连分数可以写作 $x=[a_0,...,a_{k-1},\overline{a_k,...,a_{k+L-1}}]$．如果 K 可以取作 0，即 $x=[\overline{a_0,...,a_{L-1}}]$，就称它为 **纯循环连分数**（purely periodic continued fraction），否则称它为 **混循环连分数**（eventually periodic continued fraction）．

二次无理数

与循环连分数密切相关的概念是（实）二次无理数→9.11.32@p1495（quadratic irrational），即整系数二次方程的无理数解．所有的二次无理数都可以表示成

$$a+b\sqrt{D}$$

的形式，其中，a,b 是有理数且 D 是无平方因子的正整数．本文提到的二次无理数都默认是实数．而且，$a+b\sqrt{D}$ 的共轭是指 $a-b\sqrt{D}$.

Euler 的结果说明，所有循环连分数都是二次无理数.

定理（Euler）

循环连分数表示的都是二次无理数.

证明

对于一般的循环连分数 $x=[a_0,...,a_{k-1},\overline{a_k,...,a_{k+L-1}}]$，可以设 $y=[\overline{a_k,...,a_{k+L-1}}]$，则

$$x=[a_0,...,a_{k-1},y]=L_0(y),$$

$$y=[a_k,...,a_{k+L-1},y]=L_1(y),$$

其中，$L_0(\cdot)$ 和 $L_1(\cdot)$ 都是分式线性变换．于是，得到 x 满足的方程

$$x = L_0 \circ L_1 \circ L_0^{-1}(x).$$

不妨设分式线性变换 $L_0 \circ L_1 \circ L_0^{-1}(x) = \dfrac{ax+b}{cx+d}$，则得到 x 满足的方程

$$cx^2 + (d-a)x - b = 0.$$

因而，循环连分数都是整系数二次方程的解．又因为无限连分数都是无理数，所以循环连分数都表示了二次无理数．

Lagrange 的结果说明反过来也成立，因而二次无理数和循环连分数是等价的．

定理（Lagrange）

二次无理数可以表示成循环连分数．

证明

思路是证明余项会重复出现．设二次无理数 x 可以写作

$$x = \frac{P_0 + \sqrt{D}}{Q_0}$$

的形式，其中，P_0, Q_0, D 都是整数且 $Q_0 \mid D - P_0^2$．这总是可能的，比如二次无理数 x 总可以写成

$$a + b\sqrt{D'} = \frac{p_a}{q_a} + \frac{p_b}{q_b}\sqrt{D'} = \frac{p_a q_b + \sqrt{p_b^2 q_a^2 D'}}{q_a q_b} = \frac{p_a q_a q_b^2 + \sqrt{(q_a q_b)^2 p_b^2 q_a^2 D'}}{(q_a q_b)^2}$$

再令 $P = p_a q_a q_b^2$，$Q = (q_a q_b)^2$ 和 $D = Q p_b^2 q_a^2 D'$ 即可（这里设 $b > 0$；若 $b < 0$，只需将 P 和 Q 同时取相反数）．

将它写成这种形式的好处是，可以证明它的所有余项都具有类似的形式：

$$r_k = \frac{P_k + \sqrt{D}}{Q_k},$$

其中，P_k, Q_k 是整数且 $Q_k \mid D - P_k^2$．其中，条件 $Q_k \mid D - P_k^2$ 保证了所有余项的分子中，$\sqrt{D}$ 前面的系数都是 1．

为得到余项的形式，可以使用数学归纳法．当 $k = 0$ 时，显然．假设已经得到了 r_k 的形式，并设 $a_k = \lfloor r_k \rfloor$，就有

$$r_k = a_k + \frac{1}{r_{k+1}}.$$

设 r_{k+1} 也有类似形式，并和 r_k 一起代入上式，有

$$\frac{P_k + \sqrt{D}}{Q_k} = a_k + \frac{Q_{k+1}}{P_{k+1} + \sqrt{D}} = a_k + \frac{Q_{k+1}P_{k+1} - Q_{k+1}\sqrt{D}}{P_{k+1}^2 - D}.$$

因为二次无理数表示成 $a + b\sqrt{D}$ 的方式是唯一的，所以比较两侧系数可知

$$\frac{P_k}{Q_k} = a_k + \frac{Q_{k+1}P_{k+1}}{P_{k+1}^2 - D}, \frac{1}{Q_k} = -\frac{Q_{k+1}}{P_{k+1}^2 - D}.$$

将第二个等式代入第一个等式可以解出 P_{k+1}：

$$\frac{P_k}{Q_k} = a_k - \frac{P_{k+1}}{Q_k} \iff P_{k+1} = a_k Q_k - P_k.$$

再代入第二个等式，就可以解出 Q_{k+1}：

$$Q_{k+1} = \frac{D - P_{k+1}^2}{Q_k} = \frac{D - (a_k Q_k - P_k)^2}{Q_k} = -a_k^2 Q_k + 2a_k P_k + \frac{D - P_k^2}{Q_k}.$$

根据归纳假设，$Q_k \mid D - P_k^2$，所以确实 P_{k+1} 和 Q_{k+1} 都是整数，即 r_{k+1} 也具有所要求的形式．

最后，证明余项只能取得有限多个值，故而必然重复．前文已经求得余项

$$\frac{P_k+\sqrt{D}}{Q_k}=r_k=-\frac{q_{k-2}x-p_{k-2}}{q_{k-1}x-p_{k-1}}$$

而且对于无理数，总有 $r_k>1$．同时，它的共轭

$$\frac{P_k-\sqrt{D}}{Q_k}=r_k^*=-\frac{q_{k-2}x^*-p_{k-2}}{q_{k-1}x^*-p_{k-1}}=-\frac{q_{k-2}}{q_{k-1}}\frac{x^*-\dfrac{p_{k-2}}{q_{k-2}}}{x^*-\dfrac{p_{k-1}}{q_{k-1}}}$$

对于充分大的 k 必然小于 0，因为

$$\frac{q_{k-2}}{q_{k-1}}>0,\ \lim_{k\to\infty}\frac{x^*-\dfrac{p_{k-2}}{q_{k-2}}}{x^*-\dfrac{p_{k-1}}{q_{k-1}}}=\frac{x^*-x}{x^*-x}=1.$$

这就说明

$$\frac{2\sqrt{D}}{Q_k}=r_k-r_k^*>1\Longleftrightarrow 0<Q_k\leq 2\sqrt{D}.$$

因此，Q_k 只能取有限多个值．进而，

$$D-P_k^2=Q_kQ_{k-1}>0\Longleftrightarrow |P_k|<\sqrt{D},$$

所以，P_k 也只能取有限多个值．故而，余项 r_k 只有有限多个可能的取值，必然在无限项内重复．

定理的证明也提供了一个计算二次无理数的余项的递推公式：

二次无理数的余项递推公式

二次无理数总可以表示成

$$x=\frac{P_0+\sqrt{D}}{Q_0}$$

的形式，且 $Q_0\mid D-P_0^2$．它的余项

$$r_k=\frac{P_k+\sqrt{D}}{Q_k}$$

中，P_k,Q_k 都是整数，且满足递推关系

$$P_{k+1}=a_kQ_k-P_k,$$

$$Q_{k+1}=\frac{D-P_{k+1}^2}{Q_k}.$$

这个递推公式可以直接用于二次无理数的连分数的计算，而且根据定理的证明，$|P_k|<\sqrt{D}$ 且 $Q_k\leq 2\sqrt{D}$．该算法的复杂度取决于循环节的长度，而后者可以证明是 $O(\sqrt{D}\log D)$ 的[67]．

二次无理数

给定二次无理数 $\alpha=\dfrac{x+y\sqrt{n}}{z}$，求出其连分数的表示．其中，$x,y,z,n\in\mathbf{Z}$ 且 $n>0$ 不是完全平方．

[67] 证明见 维基百科页面[10] 的参考文献.

解答

首先将二次无理数表示成上述形式，再利用递推公式计算即可．连分数的项由 $a_k = \lfloor r_k \rfloor$ 给出．为了求出循环节，需要存储 (P_k, Q_k) 首次出现的下标．

- C++

```cpp
// Return the continued fraction and minimal positive period
//   of a quadratic irrational (x + y * sqrt(n)) / z.
auto quadratic_irrational(int x, int y, int z, int n) {
  int p = x * z;
  int d = n * y * y * z * z;
  int q = z * z;
  int dd = (int)sqrt(n) * y * z;
  int i = 0;
  std::vector<int> a;
  std::unordered_map<size_t, int> used;
  while (!used.count(((1LL << 32) * p) | q)) {
    a.emplace_back((p + dd) / q);
    used[((1LL << 32) * p) | q] = i++;
    p = a.back() * q - p;
    q = (d - p * p) / q;
  }
  return std::make_pair(a, i - used[((1LL << 32) * p) | q]);
}

```

- Python

```python
# Return the continued fraction and minimal positive period
#   of a quadratic irrational (x + y * sqrt(n)) / z.
def quadratic_irrational(x, y, z, n):
    p = x * z
    d = n * y * y * z * z
    q = z * z
    dd = floor(sqrt(n)) * y * z
    i = 0
    a = []
    used = dict()
    while (p, q) not in used:
        a.append((p + dd) // q)
        used[p, q] = i
        p = a[-1] * q - p
        q = (d - p * p) // q
        i += 1
    return a, i - used[p, q]


```

Tavrida NU Akai Contest - Continued Fraction[11]

给定 x 和 k，且 x 不是完全平方数，$0 \le k \le 10^9$．求出 $\sqrt{x}$ 的第 k 个渐近分数 x_k.

解答

首先利用上述算法解出 $\sqrt{x}$ 的周期，将循环节表示成分式线性变换，就可以用 快速幂→9.7@p1131 获得 x_k 的值．当然，对于没有进入循环节和不足一个循环节的部分，需要单独处理．

- C++

```cpp
#include <algorithm>
#include <cmath>
#include <iostream>
#include <tuple>
#include <unordered_map>
#include <vector>

// Return the continued fraction and minimal positive period
//   of a quadratic irrational (x + y * sqrt(n)) / z.
auto quadratic_irrational(int x, int y, int z, int n) {
  int p = x * z;
  int d = n * y * y * z * z;
  int q = z * z;
  int dd = (int)sqrt(n) * y * z;
  int i = 0;
  std::vector<int> a;
  std::unordered_map<size_t, int> used;
  while (!used.count(((1LL << 32) * p) | q)) {
    a.emplace_back((p + dd) / q);
    used[((1LL << 32) * p) | q] = i++;
    p = a.back() * q - p;
    q = (d - p * p) / q;
  }
  return std::make_pair(a, i - used[((1LL << 32) * p) | q]);
}

// Fractional Linear Transformation.
struct FracLinearTrans {
  static constexpr int M = 1e9 + 7;
  int mat[4];

  FracLinearTrans(int a, int b, int c, int d) : mat{a, b, c, d} {}

  FracLinearTrans operator*(const FracLinearTrans& rhs) const {
    return FracLinearTrans(
        ((long long)mat[0] * rhs.mat[0] + (long long)mat[1] *
rhs.mat[2]) % M,
        ((long long)mat[0] * rhs.mat[1] + (long long)mat[1] *
rhs.mat[3]) % M,
        ((long long)mat[2] * rhs.mat[0] + (long long)mat[3] *
rhs.mat[2]) % M,
        ((long long)mat[2] * rhs.mat[1] + (long long)mat[3] *
rhs.mat[3]) % M);
  }
};

int main() {
```

```
  int x, k, L;
  std::cin >> x >> k;
  std::vector<int> a;
  std::tie(a, L) =
      quadratic_irrational(0, 1, 1, x);  // L==a.size()-1 for sqrt(x)
  FracLinearTrans cyc(1, 0, 0, 1);
  for (int i = a.size() - 1; i; --i) {
    cyc = FracLinearTrans(a[i], 1, 1, 0) * cyc;
  }
  // 1/0=Inf.
  FracLinearTrans res(0, 1, 0, 0);
  // Tail terms.
  for (int i = k % L; i; --i) {
    res = FracLinearTrans(a[i], 1, 1, 0) * res;
  }
  // Binary exponentiation.
  for (int b = k / L; b; b >>= 1) {
    if (b & 1) res = cyc * res;
    cyc = cyc * cyc;
  }
  // First term.
  res = FracLinearTrans(a[0], 1, 1, 0) * res;
  printf("%d/%d", res.mat[1], res.mat[3]);
  return 0;
}

```

- Python

```
from math import sqrt, floor


# Return the continued fraction and minimal positive period
#   of a quadratic irrational (x + y * sqrt(n)) / z.
def quadratic_irrational(x, y, z, n):
    p = x * z
    d = n * y * y * z * z
    q = z * z
    dd = floor(sqrt(n)) * y * z
    i = 0
    a = []
    used = dict()
    while (p, q) not in used:
        a.append((p + dd) // q)
        used[p, q] = i
        p = a[-1] * q - p
        q = (d - p * p) // q
        i += 1
    return a, i - used[p, q]


# Compose (A[0]*x + A[1]) / (A[2]*x + A[3]) and (B[0]*x + B[1]) /
(B[2]*x + B[3])
```

```python
def combine(A, B):
    return [
        t % mod
        for t in [
            A[0] * B[0] + A[1] * B[2],
            A[0] * B[1] + A[1] * B[3],
            A[2] * B[0] + A[3] * B[2],
            A[2] * B[1] + A[3] * B[3],
        ]
    ]


# Binary exponentiation.
def bpow(A, n):
    return (
        [1, 0, 0, 1]
        if not n
        else combine(A, bpow(A, n - 1))
        if n % 2
        else bpow(combine(A, A), n // 2)
    )


mod = 10**9 + 7
x, k = map(int, input().split())
a, T = quadratic_irrational(0, 1, 1, x)

A = (1, 0, 0, 1)  # (x + 0) / (0*x + 1) = x

# apply ak + 1/x = (ak*x+1)/(1x+0) to (Ax + B) / (Cx + D)
for i in reversed(range(1, len(a))):
    A = combine([a[i], 1, 1, 0], A)

C = (0, 1, 0, 0)  # = 1 / 0
while k % T:
    i = k % T
    C = combine([a[i], 1, 1, 0], C)
    k -= 1

C = combine(bpow(A, k // T), C)
C = combine((a[0], 1, 1, 0), C)
print(str(C[1]) + "/" + str(C[3]))

```

纯循环连分数

二次无理数是有循环连分数表示的充分必要条件，本节的讨论则给出了实数具有纯循环连分数表示的充分必要条件.

首先，因为纯循环连分数具有类似有限连分数的形式，所以可以做「反序」操作. 类似于反序定理，这样得到的连分数表示和原来的连分数表示之间有确定的关系.

定理（Galois）

对于纯循环连分数

$$x=[\overline{a_0,a_1,...,a_\ell}],$$

记

$$x'=[\overline{a_\ell,...,a_1,a_0}].$$

则 x 和 x' 互为「倒数负共轭」，即 x 的共轭的倒数的相反数是 x'.

证明

因为不要求 $\ell+1$ 是最小正周期，所以不妨设 $\ell>0$. 利用反序定理可知，

$$\frac{p_\ell}{p_{\ell-1}}=[a_\ell,...,a_1,a_0]=\frac{p'_\ell}{q'_\ell},$$

$$\frac{q_\ell}{q_{\ell-1}}=[a_\ell,...,a_1]=\frac{p'_{\ell-1}}{q'_{\ell-1}}.$$

因为等式两侧都是既约分数，所以

$$p'_\ell=p_\ell,q'_\ell=p_{\ell-1},p'_{\ell-1}=q_\ell,q'_{\ell-1}=q_{\ell-1}.$$

对于纯循环连分数 x，它的第 $\ell+1$ 个余项就是 x，因此，有

$$x=\frac{xp_\ell+p_{\ell-1}}{xq_\ell+q_{\ell-1}}.$$

所以，它满足二次方程

$$q_\ell x^2+(q_{\ell-1}-p_\ell)x-p_{\ell-1}=0.$$

同理，x' 满足二次方程

$$q'_\ell(x')^2+(q'_{\ell-1}-p'_\ell)x'-p'_{\ell-1}=0.$$

利用系数的关系可知，这个方程可以写作

$$p_{\ell-1}(x')^2+(q_{\ell-1}-p_\ell)x'-q_\ell=0.$$

令 $y=-\dfrac{1}{x'}$，则 x 和 y 满足同一个方程. 但是，$x>0>y$，因此它们并非同一个根，而是互为共轭的关系，这就证明了原命题.

Galois 利用这个观察，进一步地给出了二次无理数有纯循环连分数表示的充分必要条件.

定理（Galois）

二次无理数 x 可以表示为纯循环连分数，当且仅当 $x>1$ 且它的共轭 $-1<x^*<0$.

证明

如果 x 是纯循环连分数，那么利用前文的记号，有 $a_0=a_{\ell+1}\geq 1$，故而 $x>1$. 又因为它的倒数负共轭也是循环连分数，所以它的共轭 x^* 满足 $-\dfrac{1}{x^*}>1$，亦即 $-1<x^*<0$. 这就证明了纯循环连分数都满足该条件.

反过来，设二次无理数 $x>1$ 且 $-1<x^*<0$. 对于 x 的余项 r_k，有递推关系

$$r_k=a_k+\frac{1}{r_{k+1}}.$$

等式两边都是二次无理数，取共轭可知

$$r_k^* = a_k + \frac{1}{r_{k+1}^*}.$$

利用这个递推关系，可以证明 $-1 < r_k^* < 0$ 对所有 $k \geq 0$ 都成立.

首先，对于 $k = 0$，有 $-1 < r_0^* = x^* < 0$，显然. 对于 $k \geq 0$，由简单连分数定义和 $x > 1$ 可知，$a_k \geq 1$. 故而，假设 $-1 < r_k^* < 0$，就有

$$-1 < -\frac{1}{a_k} < r_{k+1}^* = \frac{1}{r_k^* - a_k} < -\frac{1}{1 + a_k} < 0.$$

这就归纳地证明了 $-1 < r_k^* < 0$ 对所有 $k \geq 0$ 都成立. 因此，有

$$a_k = -\frac{1}{r_{k+1}^*} + r_k^* = \left\lfloor -\frac{1}{r_{k+1}^*} \right\rfloor.$$

因为二次无理数一定是循环连分数，所以存在正整数 L 和至少某个充分大的 k，有 $r_k = r_{k+L}$. 但是，此时必然也有

$$a_{k-1} = \left\lfloor -\frac{1}{r_k^*} \right\rfloor = \left\lfloor -\frac{1}{r_{k+L}^*} \right\rfloor = a_{k+L-1}.$$

故而，

$$r_{k-1} = a_{k-1} + \frac{1}{r_k} = a_{k+L-1} + \frac{1}{r_{k+L}} = r_{k+L-1}.$$

这意味着最小的能够使得 $r_k = r_{k+L}$ 成立的 k 必然是 0. 也就是说，x 可以表示成纯循环连分数.

Galois 定理揭示了纯二次不尽根（pure quadratic surd）——即形如 $\sqrt{r}$ 的二次无理数——的连分数表示的规律.

推论

对于有理数 $r > 1$，如果 $\sqrt{r}$ 是无理数，那么

$$\sqrt{r} = [\lfloor \sqrt{r} \rfloor, \overline{a_1, ..., a_\ell, 2\lfloor \sqrt{r} \rfloor}]$$

且对于任意 $1 \leq k \leq \ell$，都有 $a_k = a_{\ell+1-k}$.

证明

对于二次无理数 $\sqrt{r}$，因为 $\lfloor \sqrt{r} \rfloor + \sqrt{r} > 1$ 且 $-1 < \lfloor \sqrt{r} \rfloor - \sqrt{r} < 0$，所以 $\lfloor \sqrt{r} \rfloor + \sqrt{r}$ 是纯循环连分数：

$$\lfloor \sqrt{r} \rfloor + \sqrt{r} = [\overline{2\lfloor \sqrt{r} \rfloor, a_1, ..., a_\ell}].$$

根据上述定理，它的倒数负共轭具有形式

$$\frac{1}{\sqrt{r} - \lfloor \sqrt{r} \rfloor} = [\overline{a_\ell, ..., a_1, 2\lfloor \sqrt{r} \rfloor}].$$

利用连分数的基本性质可知

$$\sqrt{r} = \lfloor \sqrt{r} \rfloor + \frac{1}{\dfrac{1}{\sqrt{r} - \lfloor \sqrt{r} \rfloor}} = [\lfloor \sqrt{r} \rfloor, \overline{a_\ell, ..., a_1, 2\lfloor \sqrt{r} \rfloor}].$$

但是，又由 $\lfloor \sqrt{r} \rfloor + \sqrt{r}$ 的连分数表示可知，

$$\sqrt{r} = -\lfloor \sqrt{r} \rfloor + (\lfloor \sqrt{r} \rfloor + \sqrt{r}) = [\lfloor \sqrt{r} \rfloor, \overline{a_1, ..., a_\ell, 2\lfloor \sqrt{r} \rfloor}].$$

因为无理数的连分数表示是唯一的，所以比较中间的系数就知道，$a_k = a_{\ell+1-k}$ 对所有 $1 \leq k \leq \ell$ 都成立.

例子：$\sqrt{74}$ 的连分数展开

$\sqrt{74}$ 的连分数可以计算如下：（此处仅是为了说明，编程计算应使用前文提到的递归算法）

$$\begin{aligned}\sqrt{74} &= 8+(-8)+\sqrt{74} = \left[8, \frac{8+\sqrt{74}}{10}\right] \\ &= \left[8, 1+\frac{-2+\sqrt{74}}{10}\right] = \left[8, 1, \frac{2+\sqrt{74}}{7}\right] \\ &= \left[8, 1, 1+\frac{-5+\sqrt{74}}{7}\right] = \left[8, 1, 1, \frac{5+\sqrt{74}}{7}\right] \\ &= \left[8, 1, 1, 1+\frac{-2+\sqrt{74}}{7}\right] = \left[8, 1, 1, 1, \frac{2+\sqrt{74}}{10}\right] \\ &= \left[8, 1, 1, 1, 1+\frac{-8+\sqrt{74}}{10}\right] = \left[8, 1, 1, 1, 1, 8+\sqrt{74}\right] \\ &= \left[8, 1, 1, 1, 1, 16+(-8)+\sqrt{74}\right] = \left[8, \overline{1, 1, 1, 1, 16}\right]\end{aligned}$$

各个余项分别是：

$$\begin{aligned}r_1 &= \tfrac{8+\sqrt{74}}{10} &&= \left[\overline{1, 1, 1, 1, 16}\right] \\ r_2 &= \tfrac{2+\sqrt{74}}{7} &&= \left[\overline{1, 1, 1, 16, 1}\right] \\ r_3 &= \tfrac{5+\sqrt{74}}{7} &&= \left[\overline{1, 1, 16, 1, 1}\right] \\ r_4 &= \tfrac{2+\sqrt{74}}{10} &&= \left[\overline{1, 16, 1, 1, 1}\right] \\ r_5 &= 8+\sqrt{74} &&= \left[\overline{16, 1, 1, 1, 1}\right]\end{aligned}$$

根据 Galois 的结论，余项 r_k 和 r_{L+1-k} 循环部分恰好相反，因此互为倒数负共轭．如果 $\sqrt{D}$ 的循环节长度 L 为奇数，那么中间的一项就与自身互为倒数负共轭；如果循环节长度 L 为偶数，就不存在这样的项．Pell 方程一节的讨论会说明，循环节长度的奇偶性将决定方程 $x^2-Dy^2=-1$ 是否有解．

二次无理数 $\sqrt{D}$ 的连分数展开主要应用在 Pell 方程→9.11.33@p1503 的求解中．

例题

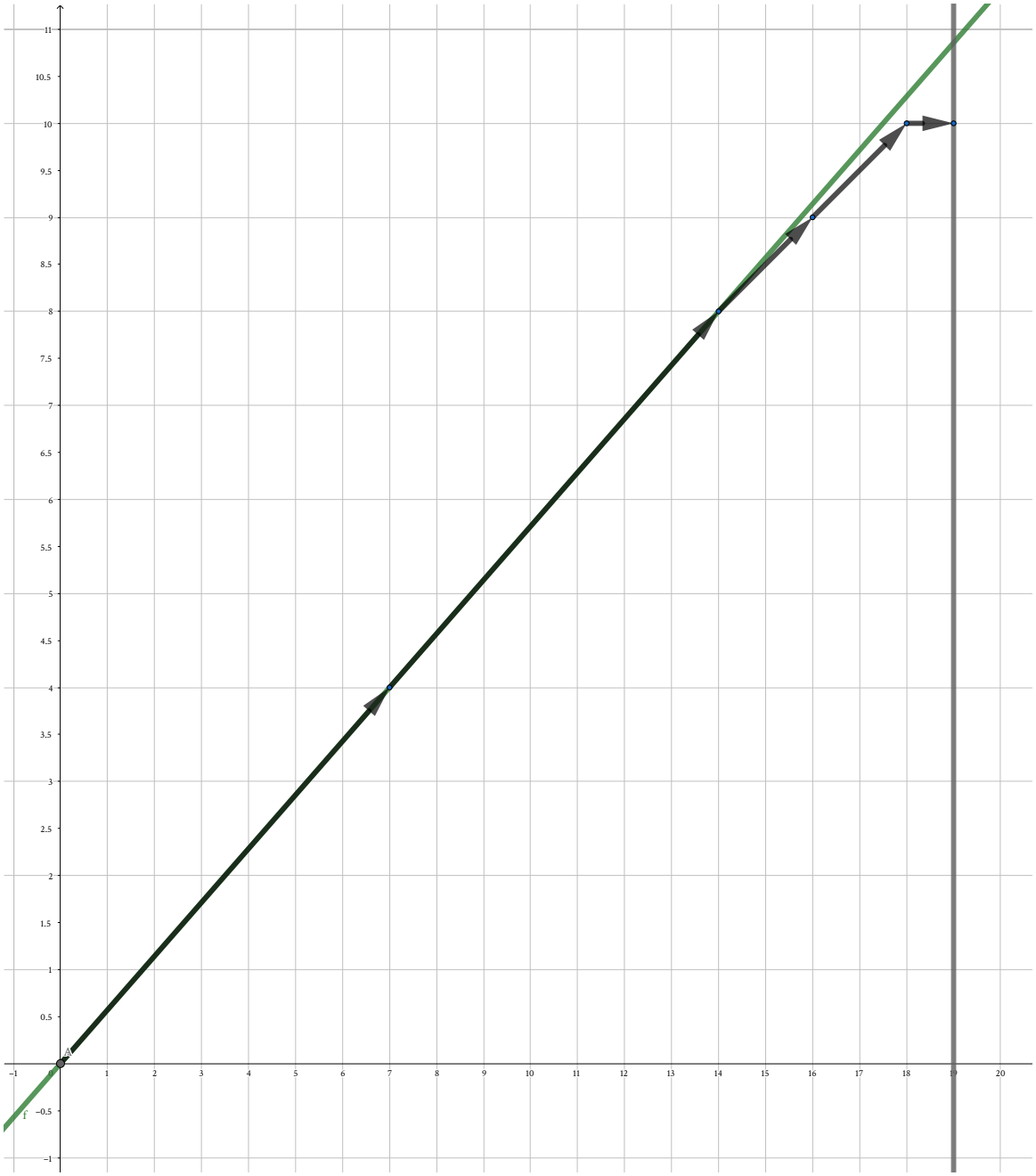
在掌握了基础概念后，需要研究一些具体的例题来理解如何在算法竞赛中应用连分数的方法．

线下凸包

给定 $r=[a_0, a_1, ..., a_n]$，求出满足 $0 \le x \le N$ 和 $0 \le y \le rx$ 的整点 (x, y) 的集合的凸包．

解答

对于无界集合 $x \ge 0$，上凸壳就是直线 $y=rx$ 本身．然而，如下图所示，如果还要求 $x \le N$，那么上凸壳最终会偏离直线．



从 $(0,0)$ 开始，可以自左向右地求出上凸壳的所有整点．假设当前已经求出的上凸壳的最后一个整点是 (x,y)．现在要求出下一个整点 (x',y')．顶点 (x',y') 在 (x,y) 右上方，记 $(\Delta x,\Delta y)=(x'-x,y'-y)$ 为两者的差值．那么，必然有

$$0<\Delta x\le N-x,0\le\Delta y\le r\Delta x.$$

第二个不等式成立，因为条件 $\Delta y>r\Delta x$ 与 (x,y) 已经在上凸壳上这件事矛盾．观察 $(\Delta x,\Delta y)$ 需要满足的条件，对于不同的点 (x,y)，只有 Δx 的上界在变化．所以，只要能解决这个子问题，就可以递归地求出原问题的所有整点．

进而，考虑子问题的解法．对比于原问题，子问题相当于将 x 的上界修改为 N'，并求出上凸壳中与原点相邻的第一个整点．记子问题的解为 (q,p)．那么，p 与 q 必然是互素的（否则不是第一个整点），且与原点连线的斜率 $\frac{p}{q}$ 是所有位于直线 $y=rx$ 下方且横坐标不超过 N' 的整点中最大的（否则不在凸包上）．结合前文的 几何解释→9.11@p1160 可知，这样的点 (x,y) 必然对应于 r 的一个下中间

分数．因为分母越大的下中间分数离 r 越近，所以子问题的解 (q,p) 对应着所有分母不超过 N' 的下中间分数中分母最大的那个．

当然，实际求解时，没必要对每个子问题都重新求出这样的下中间分数．应该首先求出所有的渐近分数，这相当于提供了遍历所有的下中间分数的方法．然后分母从大到小地遍历下中间分数，每次都尝试将它加到前一个整点 (x,y) 上，直到不能添加为止才继续尝试下一个下中间分数．

此处有一些显然的优化．首先，对于下中间分数 (q,p)，必然存在奇数 k 和 $0\le t<a_{k+1}$ 使得 $(q,p)=(q_{k-1},p_{k-1})+t(q_k,p_k)$．只要找到最大的 t 使得 $q_{k-1}+tq_k+x\le N$ 满足就好了，亦即 $t=\left\lfloor\dfrac{N-q_{k-1}-x}{q_k}\right\rfloor$．不用担心 t 越界，因为更大的下渐近分数 (q_{k+1},p_{k+1}) 已经添加完了．而每次确定添加的次数的时候，直接计算 $\left\lfloor\dfrac{N-x}{q}\right\rfloor$ 即可，不必逐个尝试．

优化后的算法的复杂度是 $O(n)$ 的．虽然下中间分数对应的整点可能有很多，但是真正成为增量的并不多．下面要说明，所有 $0\le t<a_{k+1}$ 的下中间分数 $(q,p)=(q_{k-1},p_{k-1})+t(q_k,p_k)$ 中，至多会出现两个增量．假设这些下中间分数中确实出现了增量，则此时必然有 $q_{k-1}\le N-x<q_{k+1}$．不妨设 $t=\left\lfloor\dfrac{N-q_{k-1}-x}{q_k}\right\rfloor$．如果 $t=0$，则增量就有 $\Delta x=q_{k-1}$，故而添加完增量后，就有 $N-x'<q_{k-1}$，不会再在这些下中间分数中出现新的增量；如果 $t>0$，那么添加完增量后，必然有 $N-x'=(N-q_{k-1}-x)\bmod q_k<q_k$，即使还会在同一段下中间分数中出现新的增量，下次也只能有 $t'=0$．因此，在这样的一段下中间分数中，至多只能出现两个增量．这就说明，总的时间复杂度是 $O(n)$ 的．

- C++

```cpp
// Find [ah, ph, qh] such that points r[i]=(ph[i], qh[i]) constitute
// upper convex hull of lattice points on 0 <= x <= N and 0 <= y <= r * x,
// where r = [a0, a1, a2, ...] and there are ah[i]-1 integer points on the
// segment between r[i] and r[i+1].
auto hull(std::vector<int> a, int N) {
  std::vector<int> p, q;
  std::tie(p, q) = convergents(a);
  int t = N / q.back();
  std::vector<int> ah = {t};
  std::vector<int> ph = {0, t * p.back()};
  std::vector<int> qh = {0, t * q.back()};

  for (int i = q.size() - 1; i; --i) {
    if (i % 2) {
      while (qh.back() + q[i - 1] <= N) {
        t = (N - qh.back() - q[i - 1]) / q[i];
        int dp = p[i - 1] + t * p[i];
        int dq = q[i - 1] + t * q[i];
        int k = (N - qh.back()) / dq;
        ah.push_back(k);
        ph.push_back(ph.back() + k * dp);
        qh.push_back(qh.back() + k * dq);
      }
    }
  }
  return make_tuple(ah, ph, qh);
}

```

- Python

```python
# Find [ah, ph, qh] such that points r[i]=(ph[i], qh[i]) constitute
# upper convex hull of lattice points on 0 <= x <= N and 0 <= y <= r * x,
# where r = [a0, a1, a2, ...] and there are ah[i]-1 integer points on the
# segment between r[i] and r[i+1].
def hull(a, N):
    p, q = convergents(a)
    t = N // q[-1]
    ah = [t]
    ph = [0, t * p[-1]]
    qh = [0, t * q[-1]]
    for i in reversed(range(len(q))):
        if i % 2 == 1:
            while qh[-1] + q[i - 1] <= N:
                t = (N - qh[-1] - q[i - 1]) // q[i]
                dp = p[i - 1] + t * p[i]
                dq = q[i - 1] + t * q[i]
                k = (N - qh[-1]) // dq
                ah.append(k)
                ph.append(ph[-1] + k * dp)
                qh.append(qh[-1] + k * dq)
    return ah, ph, qh


```

Timus - Crime and Punishment[12]

给定正整数 $A, B, N \leq 2 \times 10^9$，求 $x, y \geq 0$ 使得 $Ax + By \leq N$ 且 $Ax + By$ 尽可能大.

解答

这个问题有一个复杂度为 $O(\sqrt{N})$ 的解法：不妨设 $A \geq B$，因为 $A(B + x) + By = Ax + B(A + y)$，所以只需要在 $x \leq \min\{N/A, B\}$ 中搜索答案即可．这足够通过本题．但是，如果应用连分数方法，那么时间复杂度就可以降低到 $O(\log N)$.

为了讨论方便，首先通过代换 $x \mapsto \lfloor N/A \rfloor - x$ 来改变 x 的符号．令 $C = N \bmod A$ 和 $M = \lfloor N/A \rfloor$，则原问题转化为在 $0 \leq x \leq M$ 且 $By - Ax \leq C$ 的条件下，求最优的 (x, y) 使得 $By - Ax$ 最大．对于每个固定的 x，最优的 y 的取值为 $\left\lfloor \dfrac{Ax + C}{B} \right\rfloor$.

接下来要说明的是，这个问题和上一个例题具有类似的解法．但是，与上一个例题中使用下中间分数偏离直线不同，本题需要使用上中间分数来接近直线．具体来说，$C - (By - Ax)$ 的值正比于点 (x, y) 与直线 $By - Ax = C$ 的距离．要最大化 $By - Ax$，就等价于最小化这个距离．算法的目标是要找到直线 $By - Ax = C$ 下方距离它最近的可行的整点．算法的思路就是从最左侧的点开始，沿着这些整点的上凸壳搜索，逐步缩小与直线的距离，直到得到最优解.

在 (x, y) 的坐标系内，算法从 $(0, \lfloor C/B \rfloor)$ 出发，递归地寻找并添加最优的增量 $(\Delta x, \Delta y)$，且保证添加后的点比起之前更靠近直线 $By - Ax = C$，但是不能到达直线的另一侧，也不能让横坐标大于 M．设已经得到的点是 (x, y)，那么增量 $(\Delta x, \Delta y)$ 满足的条件就是

$$0 < B\Delta y - A\Delta x \le C - (By - Ax), 0 < \Delta x \le M - x.$$

按照沿下凸壳搜索的思路，只需要找到满足这些条件的点中 Δx 最小的即可．将第一个不等式改写成

$$\Delta y \le \frac{A}{B}\Delta x + \frac{C - (By - Ax)}{B}.$$

结合前文的 几何解释→9.11@p1160 可知，只要后面的常数项小于 1，那么满足这个不等式的整点 $(\Delta x, \Delta y)$ 中横坐标最小的，一定对应着某个上中间分数．这是因为它是所有分母不超过它的分母的分数中，从上方逼近某个实数效果最好的，这只能是上中间分数．而每次添加增量后，都会导致 Δy 的上界变得更紧，这意味着必须考察分母更大的上中间分数．

仿照上一个例题的思路．分母从小到大考察所有上中间分数，如果能够找到横坐标和纵坐标都不越界的上中间分数，就添加进去，并更新相应的上界．当所有可行的上中间分数都添加结束后，得到的就是最优解．相较于之前，这个题目需要同时保证横纵坐标都不越界，需要格外注意．基于和上一个例题类似的论述，不过这次是使用 $B\Delta y - A\Delta x$ 代替之前的 Δx，可以说明这个算法的复杂度是 $O(\log \min\{A, B\})$ 的．

- C++

```cpp
// Find (x, y) such that y = (A*x+B)/C,
// such that Cy - Ax is max and 0 <= x <= N.
auto closest(int A, int B, int C, int N) {
  // y <= (A*x + B)/C <=> diff(x, y) <= B
  auto diff = [&](int x, int y) -> int { return C * y - A * x; };
  std::vector<int> p, q;
  std::tie(p, q) = convergents(fraction(A, C));
  int qh = 0, ph = B / C;
  for (int i = 2; i < q.size() - 1; ++i) {
    if (i % 2 == 0) {
      while (diff(qh + q[i + 1], ph + p[i + 1]) <= B) {
        int t = 1 + (diff(qh + q[i - 1], ph + p[i - 1]) - B - 1) /
                        (-diff(q[i], p[i]));
        int dp = p[i - 1] + t * p[i];
        int dq = q[i - 1] + t * q[i];
        int k = (N - qh) / dq;
        if (k == 0) {
          return std::make_pair(qh, ph);
        }
        if (diff(dq, dp) != 0) {
          k = std::min(k, (B - diff(qh, ph)) / diff(dq, dp));
        }
        qh += k * dq;
        ph += k * dp;
      }
    }
  }
  return std::make_pair(qh, ph);
}

auto solve(int A, int B, int N) {
  int x, y;
  std::tie(x, y) = closest(A, N % A, B, N / A);
  return std::make_pair(N / A - x, y);
```

```
}

```

- Python

```python
# Find (x, y) such that y = (A*x+B) // C,
# such that Cy - Ax is max and 0 <= x <= N.
def closest(A, B, C, N):
    # y <= (A*x + B)/C <=> diff(x, y) <= B
    def diff(x, y):
        return C * y - A * x

    p, q = convergents(fraction(A, C))
    qh, ph = 0, B // C
    for i in range(2, len(q) - 1):
        if i % 2 == 0:
            while diff(qh + q[i + 1], ph + p[i + 1]) <= B:
                t = 1 + (diff(qh + q[i - 1], ph + p[i - 1]) - B - 1) // (
                    -diff(q[i], p[i])
                )
                dp = p[i - 1] + t * p[i]
                dq = q[i - 1] + t * q[i]
                k = (N - qh) // dq
                if k == 0:
                    return qh, ph
                if diff(dq, dp) != 0:
                    k = min(k, (B - diff(qh, ph)) // diff(dq, dp))
                qh, ph = qh + k * dq, ph + k * dp
    return qh, ph


def solve(A, B, N):
    x, y = closest(A, N % A, B, N // A)
    return N // A - x, y


```

June Challenge 2017 - Euler Sum[13]

求 $\sum_{x=1}^{N} \lfloor \mathrm{e}x \rfloor$ 的值，其中，e 是自然对数的底.

提示：$\mathrm{e} = [2, 1, 2, 1, 1, 4, 1, 1, 6, 1, ..., 1, 2n, 1, ...]$. [68]

解答

这个和等于集合 $\{(x, y) : 1 \le x \le N, 1 \le y \le \mathrm{e}x\}$ 中的整点个数. 在构建完直线 $y = \mathrm{e}x$ 下的整点的凸包后，可以使用 Pick 定理→12.5@p3098 计算整点个数. 时间复杂度为 $O(\log N)$.

原问题要求 $N \le 10^{4000}$. 此处 C++ 代码仅作示意，并没有实现高精度计算类.

- C++

[68] 关于自然对数的底 e 的连分数展开的证明可以参考 此处[14].

```cpp
// Find sum of floor(k * x) for k in [1, N] and x = [a0; a1, a2, ...]
int sum_floor(std::vector<int> a, int N) {
  N++;
  std::vector<int> ah, ph, qh;
  std::tie(ah, ph, qh) = hull(a, N);

  // The number of lattice points within a vertical right trapezoid
  // on points (0; 0) - (0; y1) - (dx; y2) - (dx; 0) that has
  // a+1 integer points on the segment (0; y1) - (dx; y2).
  auto picks = [](int y1, int y2, int dx, int a) -> int {
    int b = y1 + y2 + a + dx;
    int A = (y1 + y2) * dx;
    return (A + b) / 2 - y2;  // = (A - b + 2) // 2 + b - (y2 + 1)
  };

  int ans = 0;
  for (size_t i = 1; i < qh.size(); i++) {
    ans += picks(ph[i - 1], ph[i], qh[i] - qh[i - 1], ah[i - 1]);
  }
  return ans - N;
}

```

- Python

```python
# Find sum of floor(k * x) for k in [1, N] and x = [a0; a1, a2, ...].
def sum_floor(a, N):
    N += 1
    ah, ph, qh = hull(a, N)

    # The number of lattice points within a vertical right trapezoid
    # on points (0; 0) - (0; y1) - (dx; y2) - (dx; 0) that has
    # a+1 integer points on the segment (0; y1) - (dx; y2) but with
    # the number of points on the vertical right line excluded.
    def picks(y1, y2, dx, a):
        b = y1 + y2 + a + dx
        A = (y1 + y2) * dx
        return (A + b) // 2 - y2  # = (A - b + 2) // 2 + b - (y2 + 1)

    ans = 0
    for i in range(1, len(qh)):
        ans += picks(ph[i - 1], ph[i], qh[i] - qh[i - 1], ah[i - 1])
    return ans - N


```

NAIPC 2019 - It's a Mod, Mod, Mod, Mod World[15]

给定正整数 p, q, n，求 $\sum_{i=1}^{n} [pi \bmod q]$ 的值.

解答

因为和式可以变形为

$$\sum_{i=1}^{n}[pi \bmod q]=\sum_{i=1}^{n}\left(pi-q\left\lfloor\frac{pi}{q}\right\rfloor\right)=\frac{pn(n+1)}{2}-q\sum_{i=1}^{n}\left\lfloor\frac{p}{q}i\right\rfloor,$$

这个问题可以转化为上一个问题，只要用 $\frac{p}{q}$ 替代 e 即可．单次查询的时间复杂度为 $O(\log\min\{p,q\})$.

- C++

```cpp
int solve(int p, int q, int n) {
  return p * n * (n + 1) / 2 - q * sum_floor(fraction(p, q), n);
}

```

- Python

```python
def solve(p, q, N):
    return p * N * (N + 1) // 2 - q * sum_floor(fraction(p, q), N)


```

Library Checker - Sum of Floor of Linear[16]

给定正整数 N,M,A,B，求 $\sum_{i=0}^{N-1}\left\lfloor\frac{A\cdot i+B}{M}\right\rfloor$ 的值.

解答

这是到目前为止最为复杂的题目．它可以通过 类欧几里得算法→9.11.29@p1410 计算．此处给出基于连分数的算法，时间复杂度是 $O(\log\min\{A,M\})$.

可以通过构造直线 $y=\frac{Ax+B}{M}$ 以下且 $0\le x<N$ 的全部整点的凸包，并用 Pick 定理计算整点的个数．之前已经解决 $B=0$ 的情形．对于一般的情形，可以分为两步进行．首先通过添加上中间分数来逐步接近直线（即第二个例题），直到找到最接近直线的点，再通过添加下中间分数来逐步远离直线（即第一个例题）.

- C++

```cpp
// Find convex hull of lattice (x, y) such that C*y <= A*x+B.
auto hull(int A, int B, int C, int N) {
  auto diff = [&](int x, int y) -> int { return C * y - A * x; };
  auto a = fraction(A, C);
  std::vector<int> p, q;
  std::tie(p, q) = convergents(a);
  std::vector<int> ah(0), ph(1, B / C), qh(1, 0);
  auto insert = [&](int dq, int dp) -> void {
    int k = (N - qh.back()) / dq;
    if (diff(dq, dp) > 0) {
      k = std::min((int)k, (B - diff(qh.back(), ph.back())) / diff(dq,
dp));
    }
    ah.emplace_back(k);
    qh.emplace_back(qh.back() + k * dq);
```

```
    ph.emplace_back(ph.back() + k * dp);
  };
  for (int i = 1; i < q.size() - 1; ++i) {
    if (i % 2 == 0) {
      while (diff(qh.back() + q[i + 1], ph.back() + p[i + 1]) <= B) {
        int t = (B - diff(qh.back() + q[i + 1], ph.back() + p[i
+ 1])) /
                (-diff(q[i], p[i]));
        int dp = p[i + 1] - t * p[i];
        int dq = q[i + 1] - t * q[i];
        if (dq < 0 || qh.back() + dq > N) break;
        insert(dq, dp);
      }
    }
  }
  insert(q.back(), p.back());
  for (int i = q.size() - 1; i; --i) {
    if (i % 2 == 1) {
      while (qh.back() + q[i - 1] <= N) {
        int t = (N - qh.back() - q[i - 1]) / q[i];
        int dp = p[i - 1] + t * p[i];
        int dq = q[i - 1] + t * q[i];
        insert(dq, dp);
      }
    }
  }
  return std::make_tuple(ah, ph, qh);
}

// Sum of floor (Ax+B)/M from 0 to N-1.
auto solve(int N, int M, int A, int B) {
  std::vector<int> ah, ph, qh;
  std::tie(ah, ph, qh) = hull(A, B, M, N);
  // The number of lattice points within a vertical right trapezoid
  // on points (0; 0) - (0; y1) - (dx; y2) - (dx; 0) that has
  // a+1 integer points on the segment (0; y1) - (dx; y2) but with
  // the number of points on the vertical right line excluded.
  auto picks = [&](int y1, int y2, int dx, int a) -> int {
    int b = y1 + y2 + a + dx;
    int A = (y1 + y2) * dx;
    return (A + b) / 2 - y2;  // = (A - b + 2) // 2 + b - (y2 + 1)
  };
  int ans = 0;
  for (int i = 1; i < qh.size(); ++i) {
    ans += picks(ph[i - 1], ph[i], qh[i] - qh[i - 1], ah[i - 1]);
  }
  return ans - N;
}

```

- Python

```
# Find convex hull of lattice (x, y) such that C*y <= A*x+B.
```

```
def hull(A, B, C, N):
    def diff(x, y):
        return C * y - A * x

    a = fraction(A, C)
    p, q = convergents(a)
    ah = []
    ph = [B // C]
    qh = [0]

    def insert(dq, dp):
        k = (N - qh[-1]) // dq
        if diff(dq, dp) > 0:
            k = min(k, (B - diff(qh[-1], ph[-1])) // diff(dq, dp))
        ah.append(k)
        qh.append(qh[-1] + k * dq)
        ph.append(ph[-1] + k * dp)

    for i in range(1, len(q) - 1):
        if i % 2 == 0:
            while diff(qh[-1] + q[i + 1], ph[-1] + p[i + 1]) <= B:
                t = (B - diff(qh[-1] + q[i + 1], ph[-1] + p[i + 1])) // (
                    -diff(q[i], p[i])
                )
                dp = p[i + 1] - t * p[i]
                dq = q[i + 1] - t * q[i]
                if dq < 0 or qh[-1] + dq > N:
                    break
                insert(dq, dp)

    insert(q[-1], p[-1])

    for i in reversed(range(len(q))):
        if i % 2 == 1:
            while qh[-1] + q[i - 1] <= N:
                t = (N - qh[-1] - q[i - 1]) // q[i]
                dp = p[i - 1] + t * p[i]
                dq = q[i - 1] + t * q[i]
                insert(dq, dp)
    return ah, ph, qh


def solve(N, M, A, B):
    ah, ph, qh = hull(A, B, M, N)

    # The number of lattice points within a vertical right trapezoid
    # on points (0; 0) - (0; y1) - (dx; y2) - (dx; 0) that has
    # a+1 integer points on the segment (0; y1) - (dx; y2) but with
    # the number of points on the vertical right line excluded.
    def picks(y1, y2, dx, a):
        b = y1 + y2 + a + dx
```

```python
        A = (y1 + y2) * dx
        return (A + b) // 2 - y2  # = (A - b + 2) // 2 + b - (y2 + 1)

    ans = 0
    for i in range(1, len(qh)):
        ans += picks(ph[i - 1], ph[i], qh[i] - qh[i - 1], ah[i - 1])
    return ans - N


```

OKC 2 - From Modular to Rational[17]

有个未知的有理数 $\frac{p}{q}$ 且 $1 \le p, q \le 10^9$，可以询问对某个素数 $m \in [10^9, 10^{12}]$ 取模后的 pq^{-1} 的值．请在不超过十次询问内确定 p 和 q 的值．

这个问题等价于找到 $[1, n]$ 中使得 $rx \bmod m$ 最小的 x，其中 r 为 pq^{-1} 对 m 取模的结果，$n = 10^9$．

解答

根据 中国剩余定理→9.11.12@p1236，询问对多个素数取模后的结果，相当于询问对这些素数的乘积取模的结果．因此，本题可以看作是询问分数对足够大的模数 m 取模后的结果，要求确定分数的分子和分母．

对于某个模数 m，使得 $qr \equiv p \pmod m$ 成立的数对 (p, q) 可能并不唯一．假设 (p_1, q_1) 和 (p_2, q_2) 都可以使得这个等式成立，那么必然有 $(p_1q_2 - p_2q_1)r \equiv 0 \pmod m$．根据 r 的构造可知，r 与 m 互素，所以 $p_1q_2 - p_2q_1 \equiv 0 \pmod m$，亦即 $m \mid (p_1q_2 - p_2q_1)$．如果 $p_1q_2 - p_2q_1$ 不为零，那么它的绝对值至少是 m．问题中限制了 $p, q \in [1, 10^9]$，这意味着这个差值不应该超过 10^{18}，因此只要取 $m > 10^{18}$ 就可以保证求出的 (p, q) 是唯一的．

现在的问题归结为，给定模数 m 和余数 r，求不超过 n 的正整数对 (p, q) 使得 $qr \equiv p \pmod m$．在已知这样的解是唯一的情况下，其实只要找到 $q \in [1, n]$ 时使得 $qr \bmod m$ 最小的 q 即可，因为此时有且仅有一个 q 使得余数不超过 n．这正是前面提到的等价表述．

在 (q, k) 所在的平面坐标系内，这相当于要找到 $q \in [1, n]$ 时在直线 $qr - km = 0$ 下方最接近它的整点，因为余数 $qr \bmod m$ 就正比于整点与直线的距离．结合前文的 几何解释→9.11@p1160 可知，这样的整点必然对应着有理分数 $\frac{r}{m}$ 的某个下中间分数．算法复杂度是 $O(\log \min\{r, m\})$．

- C++

```cpp
// Find Q that minimizes Q*r mod m for 1 <= k <= n < m.
int mod_min(int r, int n, int m) {
  auto a = fraction(r, m);
  std::vector<int> p, q;
  std::tie(p, q) = convergents(a);
  for (int i = 2; i < q.size(); ++i) {
    if (i % 2 == 1 && (i + 1 == q.size() || q[i + 1] > n)) {
      int t = (n - q[i - 1]) / q[i];
      return q[i - 1] + t * q[i];
    }
  }
  return 0;
}
```

```
14
```

- Python

```python
# Find Q that minimizes Q*r mod m for 1 <= k <= n < m.
def mod_min(r, n, m):
    a = fraction(r, m)
    p, q = convergents(a)
    for i in range(2, len(q)):
        if i % 2 == 1 and (i + 1 == len(q) or q[i + 1] > n):
            t = (n - q[i - 1]) // q[i]
            return q[i - 1] + t * q[i]
    return 0


```

习题

- UVa OJ - Continued Fractions[18]
- ProjectEuler+ #64: Odd period square roots[19]
- 「LibreOJ NOI Round #2」单枪匹马[20]
- Codeforces Round #184 (Div. 2) - Continued Fractions[21]
- Codeforces Round #201 (Div. 1) - Doodle Jump[22]
- Codeforces Round #325 (Div. 1) - Alice, Bob, Oranges and Apples[23]
- POJ Founder Monthly Contest 2008.03.16 - A Modular Arithmetic Challenge[24]
- 2019 Multi-University Training Contest 5 - fraction[25]
- SnackDown 2019 Elimination Round - Election Bait[26]
- Luogu P5179. Fraction[27]
- Luogu P7739. [NOI2021] 密码箱[28]

参考文献与拓展阅读

- Hardy, G. H., Wright, E. M., Heath-Brown, R., & Silverman, J. (2008). An Introduction to the Theory of Numbers. Oxford Mathematics.
- 朱尧辰，王连祥《丢番图逼近引论》
- FatFish 的博客 - 连分数入门[29]
- Simple continued fraction - Wikipedia[30]
- Periodic continued fraction - Wikipedia[31]
- Gosper's original notes on continued fraction arithmetic algorithms[32]
- Understanding Bill Gosper's continued fraction arithmetic (implemented in Python)[33]

本页面主要内容译自博文 Continued fractions[34]，版权协议为 CC-BY-SA 4.0，内容有改动。

[1] 三对角矩阵 

[2] H. Heilbronn, On the Average Length of a Class of Finite Continued Fractions, 1969. 

[3] 洛谷 P17229 [Math×Girl²] 英雄变奏曲 

[4] Dirichlet 逼近定理 

[5] ЦЕПНЫЕ ДРОБИ 

[6] Klein 多面体 

[7] GCJ 2019, Round 2 - New Elements: Part 2 

[8] 射影线性群 

[9] DMOPC '19 Contest 7 P4 - Bob and Continued Fractions 

[10] 维基百科页面 

[11] Tavrida NU Akai Contest - Continued Fraction 

[12] Timus - Crime and Punishment 

[13] June Challenge 2017 - Euler Sum

[14] 此处

[15] NAIPC 2019 - It's a Mod, Mod, Mod, Mod World

[16] Library Checker - Sum of Floor of Linear

[17] OKC 2 - From Modular to Rational

[18] UVa OJ - Continued Fractions

[19] ProjectEuler+ #64: Odd period square roots

[20] 「LibreOJ NOI Round #2」单枪匹马

[21] Codeforces Round #184 (Div. 2) - Continued Fractions

[22] Codeforces Round #201 (Div. 1) - Doodle Jump

[23] Codeforces Round #325 (Div. 1) - Alice, Bob, Oranges and Apples

[24] POJ Founder Monthly Contest 2008.03.16 - A Modular Arithmetic Challenge

[25] 2019 Multi-University Training Contest 5 - fraction

[26] SnackDown 2019 Elimination Round - Election Bait

[27] Luogu P7179. Fraction

[28] Luogu P7739. [NOI2021] 密码箱

[29] FatFish 的博客 - 连分数入门

[30] Simple continued fraction - Wikipedia

[31] Periodic continued fraction - Wikipedia

[32] Gosper's original notes on continued fraction arithmetic algorithms

[33] Understanding Bill Gosper's continued fraction arithmetic (implemented in Python)

[34] Continued fractions

9.11.31 Stern–Brocot 树与 Farey 序列

引入

本文介绍存储最简分数的数据结构以及其它相关概念. 它们与 连分数→9.11.30@p1431 紧密相关, 在算法竞赛中可以用于解决一系列数论问题, 并可能作为某些题目的隐含背景出现.

Stern–Brocot 树

Stern–Brocot 树是一种维护分数的优雅的结构, 包含所有不同的正有理数. 这个结构分别由 Moritz Stern 在 1858 年和 Achille Brocot 在 1861 年独立发现.

构造

逐层构造

Stern–Brocot 树可以在迭代构造第 k 阶 Stern–Brocot 序列（Stern–Brocot sequence of order k）的过程中得到. 第 0 阶 Stern–Brocot 序列由两个简单的分数组成:

$$\frac{0}{1}, \frac{1}{0}.$$

此处的 $\frac{1}{0}$ 严格意义上并不算是有理分数, 可以理解为表示 ∞ 的最简分数.

在第 k 阶 Stern–Brocot 序列中相邻的两个分数 $\frac{a}{b}$ 和 $\frac{c}{d}$ 中间插入它们的中位分数（mediant）[69] $\frac{a+c}{b+d}$, 就得到第 $k+1$ 阶 Stern–Brocot 序列. 尽管中位分数的定义本身允许分数的约分, 但是在 Stern–Brocot 树的构造中, 只需要直接将分子和分母分别相加即可, 而不必担心约分的问题. 由此, 可以迭代地构造出所有阶的 Stern–Brocot 序列. 前几次迭代的结果如下:

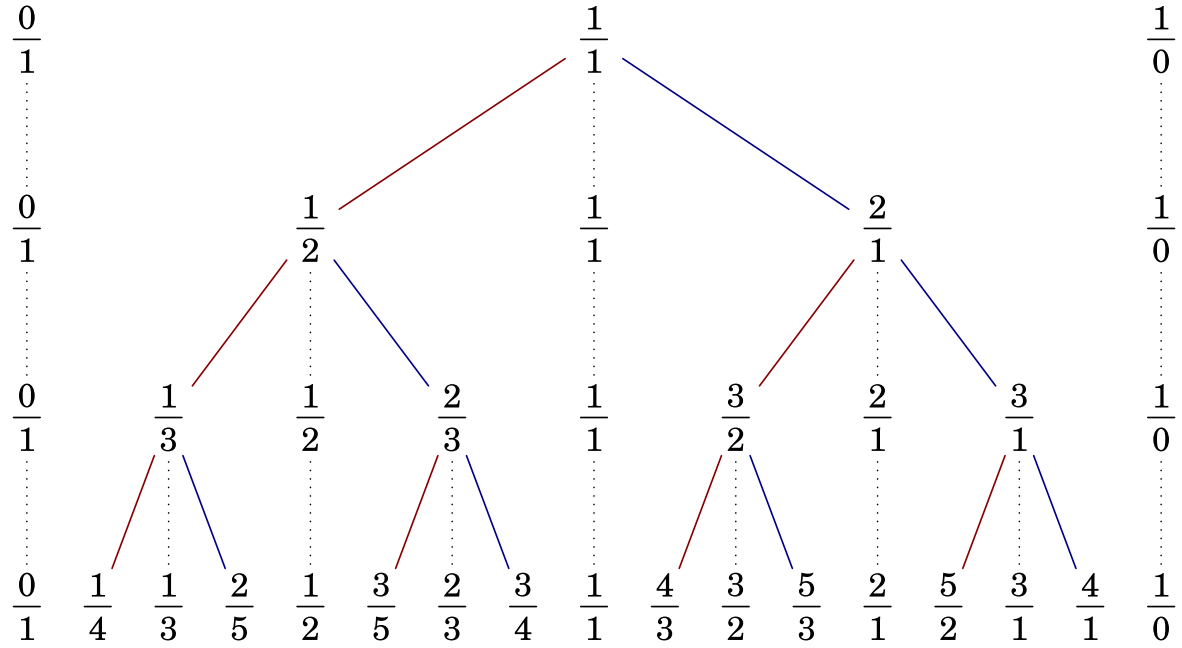
$$\frac{0}{1}, \frac{1}{1}, \frac{1}{0}$$

$$\frac{0}{1}, \frac{1}{2}, \frac{1}{1}, \frac{2}{1}, \frac{1}{0}$$

$$\frac{0}{1}, \frac{1}{3}, \frac{1}{2}, \frac{2}{3}, \frac{1}{1}, \frac{3}{2}, \frac{2}{1}, \frac{3}{1}, \frac{1}{0}$$

[69] 译名来自张明尧、张凡翻译的《具体数学》第 4.5 节.

将每次迭代中新添加的分数连结成树状结构，就得到了 Stern–Brocot 树．如下图所示：



第 k 阶 Stern–Brocot 序列，不计左右端点，就是深度为 $k-1$ 的 Stern–Brocot 树的中序遍历．

三元组构造

另一种等价的构造方式是以三元组

$$\left(\frac{0}{1},\frac{1}{1},\frac{1}{0}\right)$$

作为根节点，且在每个节点

$$\left(\frac{a}{b},\frac{p}{q},\frac{c}{d}\right)$$

后都分别添加

$$\left(\frac{a}{b},\frac{a+p}{b+q},\frac{p}{q}\right),\left(\frac{p}{q},\frac{p+c}{q+d},\frac{c}{d}\right)$$

为左右子节点，就可以构造出整个 Stern–Brocot 树．Stern–Brocot 树的每个节点记录的三元组中，实际存储的分数是位于三元组中间的分数 $\frac{p}{q}$，而左右两个分数 $\frac{a}{b}$ 和 $\frac{c}{d}$ 是更早就出现过的分数．而且，考虑前一种构造就会发现，分数 $\frac{p}{q}$ 正是通过插入 $\frac{a}{b}$ 和 $\frac{c}{d}$ 的中位分数得到的．

矩阵表示与 Stern–Brocot 数系

三元组构造意味着每个 Stern–Brocot 树上的节点都对应着一个矩阵

$$S=\begin{pmatrix}b & d\\ a & c\end{pmatrix}.$$

Stern–Brocot 树的根节点是单位矩阵 I，而向左子节点移动和向右子节点移动则分别对应将当前节点矩阵右乘以矩阵

$$L=\begin{pmatrix}1 & 1\\ 0 & 1\end{pmatrix},\ R=\begin{pmatrix}1 & 0\\ 1 & 1\end{pmatrix}.$$

每个节点实际对应的分数就是 $f(S)=\dfrac{a+c}{b+d}$．每个节点的矩阵 S 都可以写成一系列矩阵 L 和 R 的乘积，这也可以理解为是由 L 和 R 构成的字符串，表示了从根节点到达它的路径．将所有正的有理数唯一

地表示为这样的字符串，这可以看作得到了正有理数的一种表示，故而也称作 **Stern–Brocot 数系**（Stern–Brocot number system）.

建树实现

建树算法只需要模拟上述过程即可. 下面是对前 n 层的 Stern–Brocot 树做中序遍历的代码.

建树

- C++

```cpp
// In-Order Transversal of Stern-Brocot Tree till Layer N.
void build(int n, int a = 0, int b = 1, int c = 1, int d = 0, int level = 1) {
  if (level > n) return;  // Only first n layers.
  int x = a + c, y = b + d;
  build(n, a, b, x, y, level + 1);
  std::cout << x << '/' << y << ' ';  // Output the current fraction.
  build(n, x, y, c, d, level + 1);
}

```

- Python

```python
# In-Order Transversal of Stern-Brocot Tree till Layer N.
def build(n, a=0, b=1, c=1, d=0, level=1):
    if level > n:
        return
    x, y = a + c, b + d
    build(n, a, b, x, y, level + 1)
    print(f"{x}/{y}", end=" ")
    build(n, x, y, c, d, level + 1)


```

建树算法的复杂度是 $O(2^n)$ 的.

性质

接下来讨论 Stern–Brocot 树的性质. 简而言之，Stern–Brocot 树是包含所有正的最简有理分数的 二叉搜索树→10.15.1@p2213，它也是分子和分母的 堆→10.7.2@p2026，也是分母和分子构成的二元组的 笛卡尔树→10.15.6@p2311. 如果考虑前文的三元组构造中的左右端点形成的区间，则 Stern–Brocot 树也可以看作是 $[0,\infty]$ 上的 线段树→10.13.1@p2098. 这些陈述可以通过如下的三条基本性质导出：

单调性

在上面的构造中，每一层的分数都是单调递增的. 为此只需要归纳证明. 因为如果 $\frac{a}{b}<\frac{c}{d}$，那么必然有

$$\frac{a}{b}<\frac{a+c}{b+d}<\frac{c}{d}.$$

这一点可以通过消去不等式的分母得出. 归纳起点是 $\frac{0}{1}<\frac{1}{0}$，单调性也显然成立.

最简性

在上面的构造中，每个分数都是最简分数. 为此同样需要归纳证明上面的构造中，每一层相邻的分数 $\frac{a}{b}$ 和 $\frac{c}{d}$ 都满足等式

$$bc-ad=\det\begin{pmatrix}b & d\\ a & c\end{pmatrix}=1.$$

根节点处是单位矩阵，这显然成立．向下移动时，乘以的矩阵 L 和 R 的行列式都是 1，根据 行列式→9.14.6@p1729 的性质可知，在下一层同样成立

$$\det\begin{pmatrix} b & b+d \\ a & a+c \end{pmatrix} = \det\begin{pmatrix} b+d & d \\ a+c & c \end{pmatrix} = 1.$$

对于归纳起点 $\frac{0}{1}$ 和 $\frac{1}{0}$ 这也是显然的．由此，根据 裴蜀定理→9.11.8@p1218，必然可知每个分数的分子和分母都是互素的，即所有分数都是最简分数．

完全性

最后，需要说明 Stern–Brocot 树包括了所有正的最简分数．因为前两条性质已经说明 Stern–Brocot 树是二叉搜索树，而任何正的最简分数 $\frac{p}{q}$ 必然位于 $\frac{0}{1}$ 和 $\frac{1}{0}$ 之间，根据二叉搜索树上做搜索的方法，二叉搜索树上没有 $\frac{p}{q}$ 的唯一可能性就是搜索过程无限长．下面将证明，对于任意正有理数 $\frac{p}{q}$，无限长的搜索过程是不可能的．

设搜索过程中得到的 $\frac{p}{q}$ 的下界为 $\frac{a}{b}$，上界为 $\frac{c}{d}$．下界初始值为 $\frac{0}{1}$，上界初始值为 $\frac{1}{0}$，随后上下界通过 $\frac{a+c}{b+d}$ 和 $\frac{p}{q}$ 的比较结果逐步调整．若搜索过程无限长，则每一步都有

$$\frac{a}{b} < \frac{p}{q} < \frac{c}{d},$$

那么必然在无限步内都有

$$bp - aq \geq 1,\ cq - dp \geq 1.$$

将两个不等式分别乘以 $(c+d)$ 和 $(a+b)$，就能够得到

$$(c+d)(bp-aq) + (a+b)(cq-dp) \geq a+b+c+d.$$

利用前面已经说明的等式 $bc - ad = 1$ 可将上式化简为

$$p + q \geq a + b + c + d.$$

然而，每次搜索更深入一层的时候，不等式右侧都严格地增加，而左侧保持不变，因此上述条件无法在无限步内成立．因此，搜索过程会在有限步内停止．

查找分数

在实际应用 Stern–Brocot 树的过程中，经常需要查询给定分数在 Stern–Brocot 树上的位置．

朴素算法

因为 Stern–Brocot 树是二叉搜索树，只需要通过比较当前分数和要寻找的分数的大小关系，就可以确定从根节点到给定分数的路径．将路径上向左子节点移动和向右子节点移动分别记作 L 和 R，则每个路径就对应一个由 L 和 R 构成的字符串，这就是上文提到的有理数在 Stern–Brocot 数系中的表示．求得到达某个有理数的路径的过程就相当于求得该有理数在 Stern–Brocot 数系中的表示．

朴素的分数查找算法的实现如下：

朴素分数查找

- C++

```cpp
// Locate a given fraction in the Stern-Brocot tree.
std::string find(int x, int y, int a = 0, int b = 1, int c = 1, int d = 0) {
  int m = a + c, n = b + d;
  if (x == m && y == n) return "";
  if (x * n < y * m) {
    return 'L' + find(x, y, a, b, m, n);
  } else {
    return 'R' + find(x, y, m, n, c, d);
```

```
  }
}

```

- Python

```python
# Locate a given fraction in the Stern-Brocot tree.
def find(x, y, a=0, b=1, c=1, d=0):
    m = a + c
    n = b + d
    if x == m and y == n:
        return ""
    if x * n < y * m:
        return "L" + find(x, y, a, b, m, n)
    else:
        return "R" + find(x, y, m, n, c, d)


```

算法的复杂度是 $O(p+q)$ 的，因此在算法竞赛中并不实用.

在 Stern–Brocot 数系中，每个正的无理数都对应着唯一的无限长的字符串. 可以使用同样的算法构造出这个字符串. 这个无限长的字符串的每个前缀都对应着一个有理的最简分数. 将这些最简分数排成一列，数列中的分数的分母是严格递增的，而数列的极限就是该无理数. 因此，Stern–Brocot 树可以用于找到某个无理数的任意精度的有理逼近. 但是，应当注意的是，这个有理数数列和无理数之间的差距并非严格递减的. 对于有理逼近的严格理论，应当参考连分数页面的 丢番图逼近→9.11.30@p1431 一节. 在实际应用 Stern–Brocot 树寻找某个实数在分母不超过某范围的最佳逼近时，最后应当注意比较左右区间端点到该实数的距离.

快速算法

查找分数的朴素算法效率并不高, 但是经过简单的优化就可以得到 $O(\log(p+q))$ 的快速查找算法. 优化的关键是将连续的 L 和 R 合并处理.

如果要查找的分数 $\frac{p}{q}$ 落入 $\frac{a}{b}$ 和 $\frac{c}{d}$ 之间，那么连续 t 次向右移动时，右侧边界保持不动，而左侧边界移动到 $\frac{a+tc}{b+td}$ 处；反过来，连续 t 次向左移动时，左侧边界保持不动，而右侧边界移动到 $\frac{ta+c}{tb+d}$ 处. 因此，可以直接通过 $\frac{a+tc}{b+td} < \frac{p}{q}$ 或 $\frac{p}{q} < \frac{ta+c}{tb+d}$ 来确定向右和向左移动的次数. 此处取严格不等号，是因为算法移动的是左右端点，而要寻找的分数是作为最后得到的端点的中位分数出现的.

快速分数查找

- C++

```cpp
// Locate a given fraction in the Stern-Brocot tree.
auto find(int x, int y) {
  std::vector<std::pair<int, char>> res;
  int a = 0, b = 1, c = 1, d = 0;
  bool right = true;
  while (x != a + c || y != b + d) {
    if (right) {
      auto t = (b * x - a * y - 1) / (y * c - x * d);
      res.emplace_back(t, 'R');
      a += t * c;
```

```
      b += t * d;
    } else {
      auto t = (y * c - x * d - 1) / (b * x - a * y);
      res.emplace_back(t, 'L');
      c += t * a;
      d += t * b;
    }
    right ^= 1;
  }
  return res;
}

```

- Python

```
# Locate a given fraction in the Stern-Brocot tree.
def find(x, y):
    res = []
    a, b, c, d = 0, 1, 1, 0
    right = True
    while x != a + c or y != b + d:
        if right:
            t = (b * x - a * y - 1) // (y * c - x * d)
            res.append((t, "R"))
            a += t * c
            b += t * d
        else:
            t = (y * c - x * d - 1) // (b * x - a * y)
            res.append((t, "L"))
            c += t * a
            d += t * b
        right ^= 1
    return res


```

当前查找算法需要分数 $\frac{p}{q}$ 已知. 如果目标分数未知, 往往需要在每次向右或向左移动时, 对移动次数进行倍增查找或者二分查找. 此时, 分数查找算法的复杂度是 $O(\log^2(p+q))$.

基于连分数的算法

对于分数已知的情形, 可以利用连分数给出更为简单的算法. 不妨假设首组移动是向右的; 如果不然, 则设首组向右移动的次数为零. 向右、向左交替移动端点, 将每组移动后的端点位置排列如下:

$$\frac{p_0}{q_0}, \frac{p_1}{q_1}, \frac{p_2}{q_2}, \ldots, \frac{p_{n-2}}{q_{n-2}}, \frac{p_{n-1}}{q_{n-1}}, \frac{p_n}{q_n}.$$

其中, 偶数组移动是向右的, 故而记录的是左端点的位置; 奇数组移动是向左的, 故而记录的是右端点的位置. 在这一列端点前面再添加两个端点

$$\frac{p_{-2}}{q_{-2}} = \frac{0}{1}, \frac{p_{-1}}{q_{-1}} = \frac{1}{0}.$$

设第 k 组移动的次数为 t_k, 那么根据上面得到的移动次数与端点位置之间的关系可知

$$\frac{p_k}{q_k} = \frac{t_k p_{k-1} + p_{k-2}}{t_k q_{k-1} + q_{k-2}}.$$

根据连分数的 递推关系→9.11.30@p1431 就可以知道, 端点

$$\frac{p_k}{q_k} = [t_0, t_1, \cdots, t_k].$$

最后得到的连分数是

$$\frac{p}{q} = \frac{p_n + p_{n-1}}{q_n + q_{n-1}} = [t_0, t_1, \cdots, t_{n-1}, t_n, 1].$$

因此，在目标分数的末尾为一的连分数表示→9.11.30@p1431 中，不计最后的一，前面的项就编码了 Stern–Brocot 树上自根节点到当前节点的路径．其中，偶数项（下标自 0 开始）是向右子节点的边，奇数项是向左子节点的边．

有理数的连分数表示可以通过辗转相除法求得，因此基于连分数表示的分数查找算法的复杂度是 $O(\log \min\{p, q\})$ 的．

基于连分数的分数查找

- C++

```cpp
// Locate a given fraction in the Stern-Brocot tree.
auto find(int x, int y) {
  std::vector<std::pair<int, char>> res;
  bool right = true;
  while (y) {
    res.emplace_back(x / y, right ? 'R' : 'L');
    x %= y;
    std::swap(x, y);
    right ^= 1;
  }
  --res.back().first;
  return res;
}

```

- Python

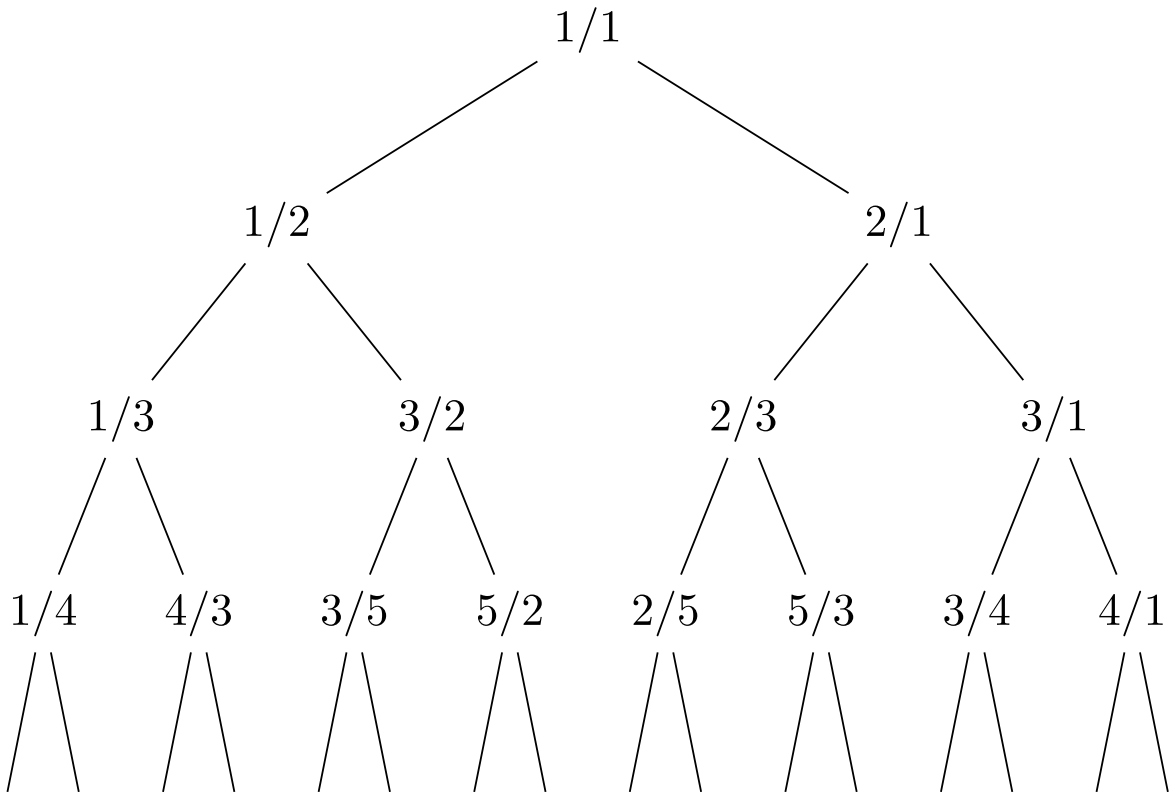
```python
# Locate a given fraction in the Stern-Brocot tree.
def find(x, y):
    res = []
    right = True
    while y:
        res.append([x // y, ("R" if right else "L")])
        x, y = y, x % y
        right ^= 1
    res[-1][0] -= 1
    return res


```

利用连分数表示，可以简单地表达出某个节点的父节点和子节点．对于节点 $[t_0, t_1, \cdots, t_n, 1]$，它的父节点就是沿最后的移动方向少移动一步的节点：在 $t_n > 1$ 时，父节点是 $[t_0, t_1, \cdots, t_n - 1, 1]$；否则，父节点是 $[t_0, t_1, \cdots, t_{n-1}, 1]$．它的两个子节点则分别是 $[t_0, t_1, \cdots, t_n + 1, 1]$ 和 $[t_0, t_1, \cdots, t_n, 1, 1]$；两个节点哪个是左子节点，哪个是右子节点，需要根据 n 的奇偶性判断．

Calkin–Wilf 树

另外一种更为简单的存储正有理分数的结构是 Calkin–Wilf 树．它通常如下所示：



树的根节点为 $\frac{1}{1}$. 然后, 对于分数 $\frac{p}{q}$ 所在的节点, 其左右子节点分别为 $\frac{p}{p+q}$ 和 $\frac{p+q}{q}$. 与 Stern–Brocot 树类似，它的每个分数都是最简分数，且包括全体正的最简分数各恰好一次.

与连分数的关系

与 Stern–Brocot 树不同，Calkin–Wilf 树不是二叉搜索树，因此不能用于二分查找有理数.

在 Calkin–Wilf 树中，当 $p > q$ 时，分数 $\frac{p}{q}$ 的父节点为 $\frac{p-q}{q}$；当 $p < q$ 时，为 $\frac{p}{q-p}$. 对于第一种情形，自 $\frac{p}{q}$ 出发，它是父节点的右子节点，可以一直通过父节点的右边向上移动，直到分子不再大于分母为止，此时节点存储的分数是 $\frac{p \bmod q}{q}$，本组移动的次数是 $\left\lfloor \frac{p}{q} \right\rfloor$；对于第二种情形，它是父节点的左子节点，可以一直通过父节点的左边向上移动，直到分母不再大于分子为止，此时节点存储的分数是 $\frac{p}{q \bmod p}$，本组移动的次数是 $\left\lfloor \frac{q}{p} \right\rfloor$.

利用连分数的语言，设当前的节点存储的是某个连分数的余项 s_k，则沿父节点的右边向上移动 $\lfloor s_k \rfloor$ 次，到达分数 $\frac{1}{s_{k+1}}$，随后沿父节点的左边移动 $\lfloor s_{k+1} \rfloor$ 次，到达分数 s_{k+2}. 因此，从节点 $s_0 = \frac{p}{q}$ 开始向上到达根节点 $\frac{1}{1}$ 的路径由连分数 $[t_0, t_1, \cdots, t_n, 1]$ 编码：除去最后的 1，偶数项（下标自 0 开始）表示沿父节点的右边移动的次数，奇数项表示沿父节点的左边移动的次数.

对于分数 $\frac{p}{q} = [t_0, t_1, \cdots, t_n, 1]$ 所在的节点，它的父节点有如下表示：

1. 当 $t_0 > 0$ 时，它的父节点为 $\frac{p-q}{q} = [t_0 - 1, t_1, \cdots, t_n, 1]$；
2. 当 $t_0 = 0$ 且 $t_1 > 1$ 时，它的父节点为 $\frac{p}{q-p} = [0, t_1 - 1, t_2, \cdots, t_n, 1]$；
3. 当 $t_0 = 0$ 且 $t_1 = 1$ 时，它的父节点为 $\frac{p}{q-p} = [t_2, t_3, \cdots, t_n, 1]$.

反过来, 它的子节点则分别是 $\frac{p+q}{q} = [t_0 + 1, t_1, \cdots, t_n, 1]$ 和 $\frac{p}{p+q} = [0, 1, t_0, t_1, \cdots, t_n, 1]$. 对于第二个子节点的连分数表示，在 $t_0 = 0$ 时，应当理解为 $[0, 1 + t_1, t_2, \cdots, t_n, 1]$.

与 Stern–Brocot 树的关系

同样和连分数建立起联系，Stern–Brocot 树中路径上节点呈现的是渐近分数的递归关系，而 Calkin–Wilf 树中路径上节点呈现的是余项的递归关系. 同一个分数的连分数表示是一定的, 因此它在 Calkin–Wilf 树上到达根节点的路径的编码和 Stern–Brocot 树的根节点到它的路径的编码是完全一样的. 但是，由于路径的方向相反, 所以虽然 Stern–Brocot 树和 Calkin–Wilf 树同一层存储的分数是一样的, 但是位置并不相同.

如果对两个树分别做 广度优先搜索→11.5@p2616 并依次对节点编号，根节点编号为 1，那么编号为 v 的节点的左右子节点分别是 $2v$ 和 $2v+1$. 从编号的二进制表示来看，除去起始的 1，从高位至低位每个 1 均表示向右子节点移动，每个 0 均表示向左子节点移动. Calkin–Wilf 树上，连分数 $[t_0, t_1, \cdots, t_n, 1]$ 表示的有理数所在节点的编号是

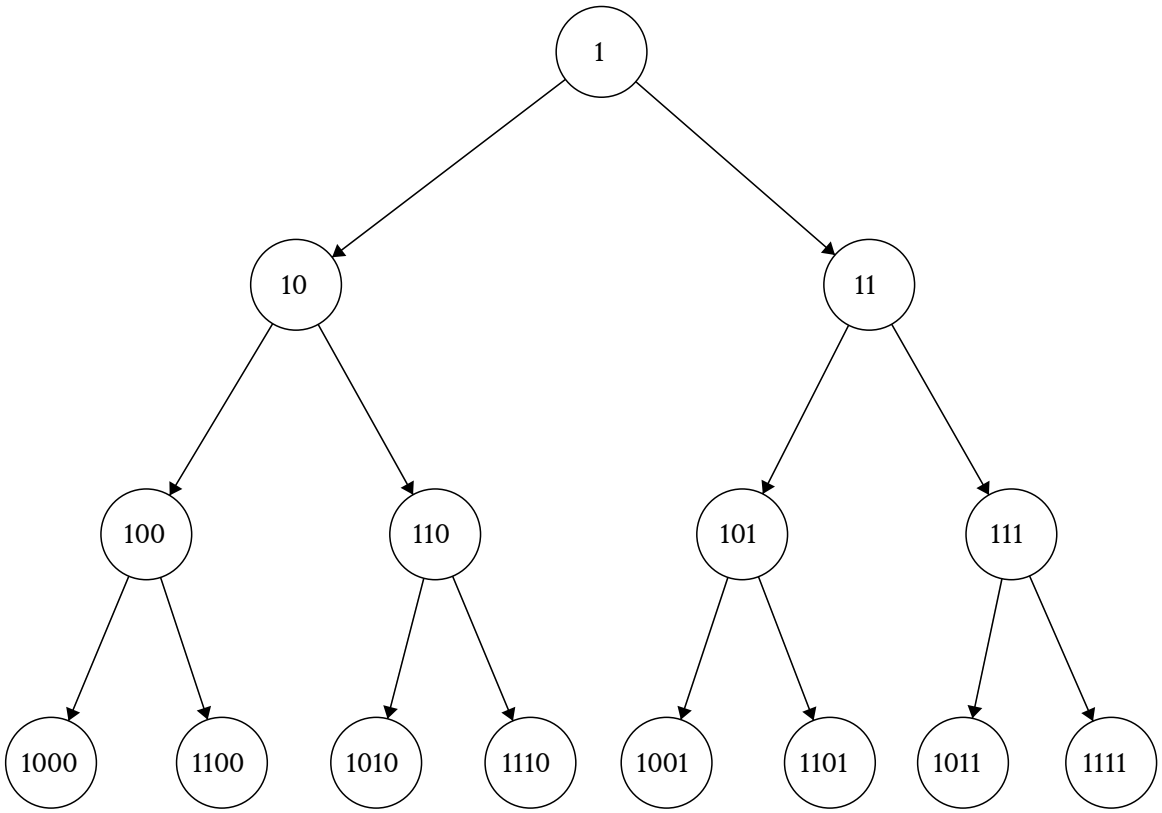
$$1\cdots\underbrace{0\cdots0}_{t_3}\underbrace{1\cdots1}_{t_2}\underbrace{0\cdots0}_{t_1}\underbrace{1\cdots1}_{t_0}.$$

相应地，Stern–Brocot 树上，连分数 $[t_0, t_1, \cdots, t_n, 1]$ 表示的有理数所在节点的编号是

$$1\underbrace{1\cdots1}_{t_0}\underbrace{0\cdots0}_{t_1}\underbrace{1\cdots1}_{t_2}\underbrace{0\cdots0}_{t_3}\cdots.$$

删去初始的 1, 余下的二进制位组成的编号, 恰为同一层的顶点自左向右的编号（自 0 开始）. 此处的推导说明, Stern–Brocot 树和 Calkin–Wilf 树中同一层的分数的排列互为位逆序置换（bit-reversal permutation），即将下标的二进制位（补齐起始的 0）反转得到的 $0 \sim (2^k-1)$ 上的置换.

正因如此，Stern–Brocot 树上的节点有时会按照对应的 Calkin–Wilf 树上的节点编号，由此得到的编号如下图所示：



这个编号可以递归地构造：根节点编号为 1，每次移动到左子节点时，就将编号首位的 1 替换成 10，而移动到右子节点时，就将编号首位的 1 替换成 11. 对该编号自右向左解读就可以得到自根节点到该节点的路径.

Stern 双原子序列

将 Calkin–Wilf 树中的所有分数按照广度优先搜索的编号排列，或将 Stern–Brocot 树中的所有分数按照上图所示的编号排列，就得到如下序列：

$$\frac{1}{1}, \frac{1}{2}, \frac{2}{1}, \frac{1}{3}, \frac{3}{2}, \frac{2}{3}, \frac{3}{1}, \frac{1}{4}, \frac{4}{3}, \frac{3}{5}, \frac{5}{2}, \cdots.$$

利用 Calkin–Wilf 树的构造过程，可以证明，该序列中相邻的两个分数，前一个的分母必然等于后一个的分子．将分子单独列出，就得到 Stern 双原子序列（Stern diatomic sequence，OEIS A002487[1]），也称为 Stern–Brocot 序列（Stern–Brocot sequence）．上述序列的编号从 1 开始，并补充规定第 0 个数是 0.

设 a_n 是 Stern 双原子序列中第 n 个数，那么，它满足如下递归关系：

$$a_{2n} = a_n,$$

$$a_{2n+1} = a_n + a_{n+1}.$$

递归起点是 $a_0 = 0$ 和 $a_1 = 1$．要求得 Stern 双原子序列中 a_n 的值，直接利用递归关系复杂度为 $O(\log^2 n)$，并不优秀．更好的做法是，将它视为 Calkin–Wilf 树上编号为 n 的分数的分子，利用上文描述的基于连分数的递归关系求解，复杂度为 $O(\log n)$.

Farey 序列

Farey 序列与 Stern–Brocot 树有着极其相似的特征．记 **第 n 阶 Farey 序列**（Farey sequence of order n）为 F_n．它表示把分母小于等于 n 的所有位于 $[0, 1]$ 之间的最简分数按大小顺序排列形成的序列：

$$F_1 = \left\{ \frac{0}{1}, \quad \frac{1}{1} \right\}$$

$$F_2 = \left\{ \frac{0}{1}, \quad \frac{1}{2}, \quad \frac{1}{1} \right\}$$

$$F_3 = \left\{ \frac{0}{1}, \quad \frac{1}{3}, \quad \frac{1}{2}, \quad \frac{2}{3}, \quad \frac{1}{1} \right\}$$

$$F_4 = \left\{ \frac{0}{1}, \quad \frac{1}{4}, \frac{1}{3}, \quad \frac{1}{2}, \quad \frac{2}{3}, \frac{3}{4}, \quad \frac{1}{1} \right\}$$

$$F_5 = \left\{ \frac{0}{1}, \frac{1}{5}, \frac{1}{4}, \frac{1}{3}, \frac{2}{5}, \frac{1}{2}, \frac{3}{5}, \frac{2}{3}, \frac{3}{4}, \frac{4}{5}, \frac{1}{1} \right\}$$

根据 Farey 序列的定义，它自然满足单调性、最简性和完全性．如上所示，F_k 相较于 F_{k-1} 新添加的分数总是 F_{k-1} 中相邻分数的中位分数.

上文中构建 Stern–Brocot 树的算法同样适用于构建 Farey 序列．因为 Stern–Brocot 树中包括所有最简分数，因此只要将边界条件修改为对分母的限制就可以得到构造 Farey 序列的代码．可以将第 n 阶 Farey 序列 F_n 看作是第 n 阶 Stern–Brocot 序列的子序列.

构建 Farey 序列

- C++

```cpp
// Farey Sequence of Order N.
void build(int n, int a = 0, int b = 1, int c = 1, int d = 1,
           bool init = true) {
  int x = a + c, y = b + d;
  if (y > n) return;              // Only first n layers.
  if (init) std::cout << "0/1 ";  // Output 0/1;
  build(n, a, b, x, y, false);
  std::cout << x << '/' << y << ' ';  // Output the current fraction.
  build(n, x, y, c, d, false);
  if (init) std::cout << "1/1 ";  // Output 1/1;
}
```

```
12
```

- Python

```python
# Farey Sequence of Order N.
def build(n, a=0, b=1, c=1, d=1, init=True):
    x, y = a + c, b + d
    if y > n:
        return
    if init:
        print("0/1", end=" ")
    build(n, a, b, x, y, False)
    print(f"{x}/{y}", end=" ")
    build(n, x, y, c, d, False)
    if init:
        print("1/1", end=" ")

```

直接构建 Farey 序列的复杂度是 $O(|F_n|) = O(n^2)$ 的.

序列长度与分数查找

Farey 序列的长度可以递归求出. 相较于 F_{n-1}，序列 F_n 多出来的分数的分母都是 n，而分子不超过 n 且与 n 互素，因此有:

$$|F_n| = |F_{n-1}| + \varphi(n) = 1 + \sum_{k=1}^{n} \varphi(k).$$

此处的 $\varphi(n)$ 是 欧拉函数→9.11.5@p1196. 该式利用 线性筛→9.11.6@p1199 可以 $O(n)$ 求出，而利用 杜教筛→9.11.24@p1353 可以将复杂度降到 $O(n^{2/3})$.

相较于直接求出序列的长度，更为常见的情景是需要求出序列 F_n 中某一分数 $r = \frac{p}{q}$ 的编号. 这相当于求值

$$1 + \sum_{k=1}^{n} \sum_{i=1}^{\lfloor rk \rfloor} [i \perp k] = 1 + \sum_{d=1}^{n} \mu(d) \sum_{j=1}^{\lfloor n/d \rfloor} \lfloor rj \rfloor.$$

要得到右式，应用了 莫比乌斯反演→9.11.23@p1331. 线性筛和枚举因子结合，可以做到 $O(n)$ 预处理和 $O(n \log n)$ 询问；杜教筛和 类欧几里得算法→9.11.29@p1410 结合，可以做到 $O(n^{2/3})$ 预处理和 $O(\sqrt{n} \log n)$ 询问.

反过来，已知编号求解分数，需要对 $[0, 1]$ 之间的实数进行二分，或者在 Stern–Brocot 树上进行 二分→9.11@p1160. 前者可能受到浮点数精度限制，对分数编号的询问次数为 $O(\log V)$，其中 V 是精度范围；后者不受浮点数精度限制，但是对分数编号的询问次数为 $O(\log^2 n)$.

Farey 邻项

如果分数 $\frac{a}{b}$ 和 $\frac{c}{d}$ 在某个 Farey 序列中相邻，就称它们为 **Farey 邻项** (Farey neighbors)，也称为 Farey 对 (Farey pair).

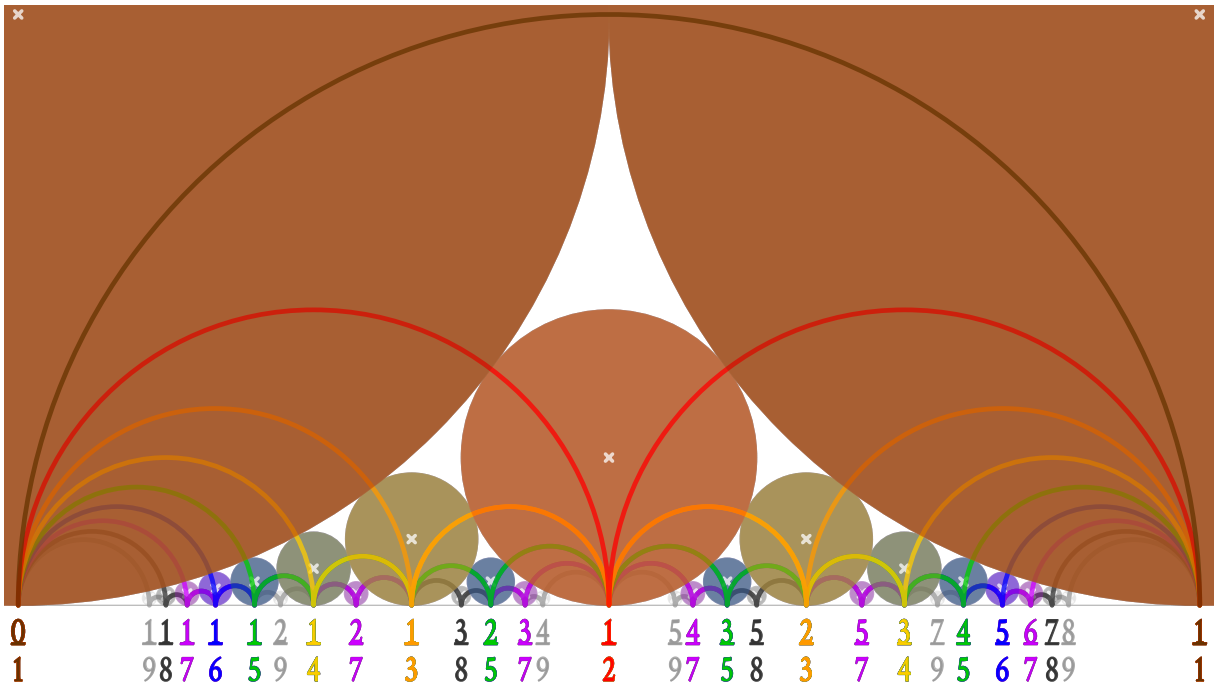
设 $\frac{a}{b} < \frac{c}{d}$. 从 Farey 序列的构造过程可知，两个相邻分数中后加入的分数必然是另一个分数与它先前的邻项的中位分数，因此 Farey 邻项在某一阶 Stern–Brocot 序列中也相邻，根据 最简性→9.11@p1160 中证明的结论可知，必然也有

$$bc - ad = 1.$$

反过来，这也是两个最简真分数成为 Farey 邻项的充分条件．现在说明这一点．不妨设 $\frac{a}{b}$ 是两个分数中分母较大的，那么两个分数都会出现在 F_b 中．设 $\frac{e}{f}$ 是序列 F_b 中排在 $\frac{a}{b}$ 右侧的一项，按照已经说明的必要性可知，$be-af=1$．但是，线性同余方程→9.11.11@p1233 $bx-ay=1$ 只有一组 $y\leq b$ 的正整数解，故而必然有 $(e,f)=(c,d)$．

其实，因为下一个会出现在两者之间的最简分数必然是 $\frac{a+c}{b+d}$，所以 $\frac{a}{b}$ 和 $\frac{c}{d}$ 在第 $\max\{b,d\}$ 到第 $(b+d-1)$ 阶 Farey 序列中都是相邻的．

分母不超过 9 的 Farey 邻项关系如下图所示：



图中的圆称为 Ford 圆[2]：对于每个 $[0,1]$ 内的最简分数 $\frac{p}{q}$，都以 $\left(\frac{p}{q},\frac{1}{2q^2}\right)$ 为圆心、以 $\frac{1}{2q^2}$ 为半径绘制一个圆．图示说明，两个分数对应的 Ford 圆只能相切或相离，且两圆相切，当且仅当两分数是 Farey 邻项．而且，对于图中相切的任何两个圆，总存在唯一的第三个圆和两圆都相切．这第三个圆对应的分数就是两个圆对应的分数的中位分数．

要验证相切的圆总是对应着 Farey 邻项，可以直接计算两圆心的距离：

$$\left(\frac{a}{b}-\frac{c}{d}\right)^2+\left(\frac{1}{2b^2}-\frac{1}{2d^2}\right)^2=\left(\frac{1}{2b^2}+\frac{1}{2d^2}\right)^2+\frac{(bc-ad)^2-1}{b^2d^2}.$$

因为两个最简分数并不相同，所以 $|\,bc-ad\,|\geq 1$，故而两圆只能相切或相离．而且，两圆相切，当且仅当 $|\,bc-ad\,|=1$，这就等价于两分数是 Farey 邻项．

最后，要计算 Farey 邻项的数目．除了 $\left(\frac{0}{1},\frac{1}{1}\right)$，其余 Farey 邻项的分母都不相同．不妨设 $\frac{p}{q}$ 是其中分母较大的那个，则另一个分数必然可以从二元一次不定方程

$$qx-py=\pm 1$$

中解得．两个方程各恰有一组满足 $y<q$ 的正整数解，分别对应位于 $\frac{p}{q}$ 左右两侧的邻项．因而，每个位于 $(0,1)$ 内的真分数都有两个分母小于它的分母的 Farey 邻项，再加上 $\frac{0}{1}$ 和 $\frac{1}{1}$，这说明序列 F_n 中共计 $(2\mid F_n\mid -3)$ 对 Farey 邻项．

当然，在这个过程中找到的分数 $\frac{a}{b}<\frac{c}{d}$ 就是在序列中插入 $\frac{p}{q}$ 时它的左右邻项．因此，它们本就是 Farey 邻项且 $\frac{p}{q}$ 是它们的中位分数．设 $\frac{p}{q}=[t_0,t_1,\cdots,t_n,1]$，则这两个分母更小的 Farey 邻项就分别是 $[t_0,t_1,\cdots,t_n]$ 和 $[t_0,t_1,\cdots,t_{n-1}]$．

要计算当前分数 $\frac{p}{q}$ 的其它 Farey 邻项，只需要利用 扩展欧几里得算法→9.11.8@p1218 求出所有符合条件的解即可.

递推关系

Farey 序列有一个简洁的递推关系，可以自左向右地顺序求出第 n 阶 Farey 序列的全部分数.

首先，前面的分析指出，在 F_n 中，新添加的项 $\frac{p}{q}$ 总是左右相邻两个分数的中位分数. 其实，这个关系对于 F_n 中（除了两端点）所有分数都成立. 设 $\frac{a}{b}<\frac{p}{q}<\frac{c}{d}$，根据 Farey 邻项的充要条件，总是有

$$bp-aq=1=cq-dp\iff\frac{p}{q}=\frac{a+c}{b+d}.$$

不过，对于一般的情形，分数 $\frac{a+c}{b+d}$ 可能需要约分.

利用这个观察就可以构建如下的递推关系. 设 $\frac{a}{b}$ 和 $\frac{p}{q}$ 是已知的，要求出第三个分数 $\frac{c}{d}$ 的值. 此时，存在 k 使得

$$a+c=kp,b+d=kq$$

成立. 因为差值

$$\frac{kp-a}{kq-b}-\frac{p}{q}=\frac{bp-aq}{q(kq-b)}=\frac{1}{q(kq-b)}$$

随着 k 增加而减小，且紧邻着 $\frac{p}{q}$ 的应该是所有满足 $kq-b\le n$ 的分数中该差值最小的那个，因而，必然有

$$k=\left\lfloor\frac{n+b}{q}\right\rfloor.$$

可以验证这样得到的分数必然在 F_n 中. 因此，F_n 中的分数的分子和分母满足递推关系：

$$p_k=\left\lfloor\frac{n+q_{k-2}}{q_{k-1}}\right\rfloor p_{k-1}-p_{k-2},$$

$$q_k=\left\lfloor\frac{n+q_{k-2}}{q_{k-1}}\right\rfloor q_{k-1}-q_{k-2}.$$

递推起点是 $(p_0,q_0)=(0,1)$ 和 $(p_1,q_1)=(1,n)$.

习题

以本文材料为背景的题目：

- LOJ 6685. 迷宫[3]
- UVa 10077. The Stern-Brocot Number System[4]
- Luogu P8058. [BalkanOI2003] Farey 序列[5]
- UVa 12995. Farey Sequence[6]
- UVa 10408. Farey Sequences[7]
- UVa 12438. Farey Polygon[8]
- AtCoder ARC123F. Insert Addition[9]

需要在 Stern–Brocot 树上作二分的题目：

- AtCoder ABC333G. Nearest Fraction[10]
- SPOJ DIVCNT1 - Counting Divisors[11]
- SPOJ AFS3 - Amazing Factor Sequence (hard)[12]

参考资料与注释

- Stern–Brocot tree - Wikipedia[13]
- Calkin–Wilf tree - Wikipedia[14]
- Farey sequence - Wikipedia[15]

本页面部分内容译自博文 Дерево Штерна-Броко. Ряд Фарея[16] 与其英文翻译版 The Stern-Brocot Tree and Farey Sequences[17]. 其中俄文版版权协议为 Public Domain + Leave a Link；英文版版权协议为 CC-BY-SA 4.0. 本页面另有部分内容译自博文 Continued fractions[18], 版权协议为 CC-BY-SA 4.0. 内容均有改动.

[1] OEIS A002487

[2] Ford 圆

[3] LOJ 6685. 迷宫

[4] UVa 10077. The Stern-Brocot Number System

[5] Luogu P8058. [BalkanOI2003] Farey 序列

[6] UVa 12995. Farey Sequence

[7] UVa 10408. Farey Sequences

[8] UVa 12438. Farey Polygon

[9] AtCoder ARC123F. Insert Addition

[10] AtCoder ABC333G. Nearest Fraction

[11] SPOJ DIVCNT1 - Counting Divisors

[12] SPOJ AFS3 - Amazing Factor Sequence (hard)

[13] Stern–Brocot tree - Wikipedia

[14] Calkin–Wilf tree - Wikipedia

[15] Farey sequence - Wikipedia

[16] Дерево Штерна-Броко. Ряд Фарея

[17] The Stern-Brocot Tree and Farey Sequences

[18] Continued fractions

9.11.32 二次域

本文简要介绍二次域的相关概念. 与之相关的两个重要例子是 Gauss 整数和 Eisenstein 整数, 它们可以用于解决一些数论问题.

基本概念

本节首先介绍一系列基本概念. 二次域和二次整数环都是更一般的代数扩域和代数整数环的概念的特例, 因此本节的几乎所有定义和结论都可以恰当地推广到一般的情形. 本节的讨论将限于二次域这一特殊情形, 而不引入过多的代数数论内容.

二次域

二次域的元素都是二次代数数.

二次代数数（quadratic algebraic number）是可以表示为整系数一元二次方程的解的复数. 由一元二次方程的求根公式可知, 所有二次代数数均可以表示成

$$a+b\sqrt{d}$$

的形式, 其中, a 与 b 为有理数, d 为整数且无平方因子. 任意这种形式的数也都是二次代数数. 二次代数数可以分为有理数和 **二次无理数**（quadratic irrational number）. 显然, 二次无理数表示成上述形式的方法是唯一的.

对于任意无平方因子的整数 $d\neq 0,1$, 都可以验证集合 $\mathbf{Q}(\sqrt{d})=\{a+b\sqrt{d}:a,b\in\mathbf{Q}\}$ 对于加、减、乘、除是封闭的. 由于对于四则运算封闭的集合称为 域→9.16.1@p1805, 集合 $\mathbf{Q}(\sqrt{d})$ 就称为 **二次域**（quadratic field）. 每一个二次域都包含全体有理数, 因此它们都是有理数域 $\mathbf{Q}$ 的 二次扩域→9.16.4@p1849. 当 $d>0$ 时, 二次域 $\mathbf{Q}(\sqrt{d})$ 中所有数都是实数, 此时的 $\mathbf{Q}(\sqrt{d})$ 称为实二次域; 当 $d<0$ 时, 二次域中除了有理数都不是实数, 此时的 $\mathbf{Q}(\sqrt{d})$ 称为虚二次域.

共轭与范数

二次无理数 $a+b\sqrt{d}$ 的 **共轭**（conjugate）定义为 $a-b\sqrt{d}$. 共轭的二次无理数是同一个整系数二次方程的两个相异的根. 在实二次域中, 二次无理数的共轭与它作为复数的共轭并不一致; 在虚二次域中, 二次无理数的共轭就是它作为复数的共轭. 最后, 有理数的共轭规定为它自身. 这样就对于全体二次代数数定义了共轭的概念.

任意二次域中，由加减乘除四则运算产生的等式，无法区分共轭关系．也就是说，在等式中将每一个数换成它的共轭，即将每一个二次根号的符号改变，等式仍然成立．

利用共轭，可以构造若干个从二次代数数到有理数的映射，它们可以将对二次代数数进行的讨论转化为对有理数的讨论．较为简单的是二次代数数 α 的 **迹**（trace），即一个数与它的共轭的和，记作 $\operatorname{tr}(\alpha)$．因为这其实就是它的有理数部分的 2 倍，所以并没有提供太多信息．

更为有用的是二次代数数的范数．二次代数数与它的共轭的积称为它的 **范数**（norm）：

$$N(a+b\sqrt{d})=a^2-db^2$$

显然，在虚二次域中，范数的概念，与复数的模的平方的概念一致；但是，在实二次域中这两个概念并不一致．

范数具有良好的性质．首先，因为 d 不含平方因子，不可能是平方数，所以只有 0 的范数是 0．其次，范数保持乘法和除法：

$$N(a_1+b_1\sqrt{d})N(a_2+b_2\sqrt{d})=N((a_1+b_1\sqrt{d})(a_2+b_2\sqrt{d})),$$

$$\frac{N(a_1+b_1\sqrt{d})}{N(a_2+b_2\sqrt{d})}=N\left(\frac{a_1+b_1\sqrt{d}}{a_2+b_2\sqrt{d}}\right).$$

另外，二次代数数的倒数就是它的共轭与它的范数之比：

$$\frac{1}{a+b\sqrt{d}}=\frac{a-b\sqrt{d}}{N(a+b\sqrt{d})}.$$

利用 Vieta 定理可知，二次代数数 α 实际上是方程

$$x^2-\operatorname{tr}(\alpha)x+N(\alpha)=0$$

的根．该方程的判别式也称为该二次代数数的 **判别式**（discriminant），记作 $\operatorname{disc}(\alpha)$，取值为 $4db^2$．

矩阵表示

类似于复数，二次代数数也可以用矩阵表示．令 $d\neq 0,1$ 为无平方因子的整数且 a,b 为有理数，则 $a+b\sqrt{d}$ 可以表示为矩阵

$$\begin{pmatrix}a & b\\ db & a\end{pmatrix}.$$

可以验证，该矩阵的加、减、乘、除四则运算的结果就对应于二次代数数的运算结果．该矩阵的迹和行列式分别对应了二次代数数的迹和范数，该矩阵的特征多项式的判别式就是二次代数数的判别式．矩阵的伴随矩阵对应着二次代数数的共轭．

二次整数环

二次代数数中较为特殊的是二次整数．**二次整数**（quadratic integer）指可以表示为二次项系数为一的整系数一元二次方程的解的复数．与二次代数数的唯一不同就是对于二次项系数的限制．根据求根公式，二次方程 $x^2+px+q=0$ 的两个根是

$$\frac{-p\pm\sqrt{p^2-4q}}{2}.$$

如果 p 是偶数，即 $p=2k$，那么两个根可以写作 $-k\pm\sqrt{k^2-q}$；如果 p 是奇数，即 $p=2k+1$，那么两个根可以写作 $-k-\dfrac{1\pm\sqrt{4(k^2+k-q)+1}}{2}$．从这两种情形中可以归纳出 $\mathbf{Q}(\sqrt{d})$ 中的二次整数必然可以写作

$$a+b\omega$$

的形式，其中，a 和 b 是整数，而

$$\omega=\begin{cases}\dfrac{1+\sqrt{d}}{2}, d\equiv 1 \pmod 4,\\ \sqrt{d}, \quad d\equiv 2,3 \pmod 4.\end{cases}$$

反过来，任何这种形式的复数都是二次整数．每个非有理数的二次整数写成该形式的方法都是唯一的．

二次域 $\mathbf{Q}(\sqrt{d})$ 中的全体二次整数构成的集合记作 $\mathbf{Z}[\omega]$．因为这个集合对于加、减、乘都是封闭的，所以它也称为 **二次整数环**（quadratic integer ring）．二次整数环中的有理数恰为全体整数．如果考察 $\mathbf{Z}[\omega]$ 中的二次整数的全体比值构成的集合，就得到相应的二次域 $\mathbf{Q}(\sqrt{d})$．

二次整数的迹、范数和判别式都是整数．二次整数环 $\mathbf{Z}[\omega]$ 中全体二次无理数的判别式中绝对值最小者也称为相应的二次域 $\mathbf{Q}(\sqrt{d})$ 的判别式．当 $d\equiv 1 \pmod 4$ 时，判别式是 d；当 $d\equiv 2,3 \pmod 4$ 时，判别式是 $4d$．

整除、相伴与单位数

类似于整数，对于二次整数同样可以建立整除理论．当然，讨论必须在同一个二次整数环内进行．

对于二次整数环 $\mathbf{Z}[\omega]$ 中的二次整数 α 和 β，如果存在同一个环中的二次整数 γ 使得 $\beta=\alpha\gamma$ 成立，就称 α 整除 β，记作 $\alpha\mid\beta$．整除关系是二次整数环上的 预序→9.20@p1951 关系．如果同时有 $\alpha\mid\beta$ 和 $\beta\mid\alpha$，那么 α 和 β 在研究整除理论时就可以视作同一个数，它们称为 **相伴**（associate）．相伴关系是二次整数环上的等价关系．

类比到整数的整除理论上，相伴关系就是互为相反数．整除理论通常只需要考虑自然数就好了，而不必考察负数的情形．对于二次整数而言，相伴关系可能更为复杂一些．如果二次整数 α 和 β 相伴，则存在二次整数 γ 和 δ 使得 $\beta=\alpha\gamma$ 和 $\alpha=\beta\delta$ 成立．因此，β 和 α 的比值 γ 必然是特殊的二次整数，即存在二次整数 δ 使得 $\gamma\delta=1$．这样的二次整数 γ 称为 **单位数**（unit），也称为可逆元．两个二次整数相伴，当且仅当它们的比值是单位数．因此，要理解二次整数环上的相伴关系，就要理解它的单位数的结构．

因为范数可以保持乘法运算，且二次整数的范数总是整数，所以利用范数可以将二次整数的整除关系转化为整数的整除关系．也就是说，如果 $\alpha\mid\beta$，那么必然有 $N(\alpha)\mid N(\beta)$．同样地，二次整数 α 是单位数，当且仅当它的范数 $N(\alpha)=\pm 1$．因此，要获得二次整数环 $\mathbf{Z}[\omega]$ 中的全体单位数，只需要求解不定方程：

$$N(a+b\omega)=\pm 1,$$

其中，范数的表达式为

$$N(a+b\omega)=\begin{cases}a^2+ab+\dfrac{1-d}{4}b^2, d\equiv 1 \pmod 4,\\ a^2-db^2, \quad d\equiv 2,3 \pmod 4.\end{cases}$$

对于虚二次整数环，即 $d<0$ 时，因为范数必然是非负整数，容易验证对于所有的无平方因子的负数 $d\neq -1,-3$，解都只有 $(a,b)=(\pm 1,0)$，即除了 $\mathbf{Z}[\sqrt{-1}]$ 和 $\mathbf{Z}[\frac{1+\sqrt{-3}}{2}]$，虚二次整数环的单位数都只有 ± 1．记 $\mathrm{i}=\sqrt{-1}$，则二次整数环 $\mathbf{Z}[\mathrm{i}]$ 又称作 Gauss 整数环，它的单位数的集合为 $\{\pm 1,\pm\mathrm{i}\}$．记 $\omega=\frac{1+\sqrt{-3}}{2}$，则二次整数环 $\mathbf{Z}[\omega]$ 又称为 Eisenstein 整数环，它的单位数的集合为 $\{\pm 1,\pm\omega,\pm\omega^2\}$．

对于实二次整数环，即 $d>0$ 时，情形较为复杂，可以转化为对相应的 Pell 方程→9.11.33@p1503 的讨论．由 Pell 方程的相关结论可知，此时的全体单位数的集合可以写作 $\{\pm u^k : k\in\mathbf{Z}\}$，其中的 u 称为 **基本单位数**（fundamental unit）．基本单位数可以通过相应的 Pell 方程进行求解．基本单位数并不是唯一的：如果 u 是基本单位数，那么它的共轭 $\bar{u}$ 以及 $-u$ 和 $-\bar{u}$ 都是基本单位数．

二次整数环的单位数的结构可以推广到一般的 代数整数环→9.16.4@p1849．代数数论中的 Dirichlet 单位定理[1] 指出，所有代数整数环的全体单位数都构成 有限生成 Abel 群→9.16.2@p1812，同时这一定理也给出了这个群的秩．

整数的整除理论中的最大公因数、带余除法、Bezout 定理、唯一分解定理等内容都可以推广到部分或全部的二次整数环上．在某个二次整数环上能否进行这样的推广，其实反映了该二次整数环性质是否足够

接近于整数环. 并非所有二次整数环上都成立唯一分解定理; 在成立唯一分解定理的二次整数环中, 也只有一部分二次整数环上可以进行带余除法. 对于这部分内容的讨论, 可以参考环论部分的 二次整数环→9.16.3@p1830 一节或相关书籍.

唯一分解

整数的唯一分解定理如果可以推广到二次整数环上, 应当具有这样的形式: 环 $\mathbf{Z}[\omega]$ 中的每个二次整数都可以写成若干个不可约元的乘积, 且这个分解在不计相伴和次序的意义下唯一. 其中的 不可约元→9.16.3@p1830 是指不能够继续分解为非单位数的乘积的二次整数, 类似于整数唯一分解定理中的素数. 前文已经提及, 并非所有的二次整数环都成立唯一分解定理.

例如, 在 $\mathbf{Z}[\sqrt{-5}]$ 中有分解 $9 = 3 \times 3 = (2+\sqrt{-5}) \times (2-\sqrt{-5})$, 但是, 3 和 $2 \pm \sqrt{-5}$ 都已经是不可约元, 因而分解并不唯一. 要说明这三个数都是不可约元, 可以通过考察它们的范数: 它们的范数都是 9, 如果它们可以分解成非单位数的乘积, 那么分解得到的因子的范数只能是 3, 但是 $\mathbf{Z}[\sqrt{-5}]$ 中并没有范数为 3 的二次整数.

一般的二次整数环中唯一分解定理不成立的主要原因是仅仅使用二次整数进行的分解是不够彻底的. 例如要分解乘积 $abcd$, 但是可选的基本元素只有 $\{ab, cd, ac, bd\}$ 时, 得到的分解自然不唯一; 要得到唯一分解定理, 必须进一步考虑 $\{a, b, c, d\}$. 在二次整数环中, 比二次整数更为细致的结构就是二次整数环的 理想→9.16.3@p1830, 将二次整数映射至它对应的主理想可以将它所在的相伴等价类嵌入到理想的集合中, 因而整数的分解只是理想的分解的特殊情形. 如果考虑二次整数环中理想的分解, 那么可以证明所有二次整数环的理想都可以唯一地分解为若干个素理想的乘积. 这说明, 二次整数环都是 Dedekind 整环[2]. 更一般地, 所有的代数整数环都是 Dedekind 整环.

如果一个二次整数环上成立唯一分解定理, 那么它的素理想和不可约元 (在相伴意义下的等价类) 一一对应, 因此, 对于这一类二次整数环, 将理想分解为素理想就等价于将二次整数分解为不可约元. 而且, 此时的不可约元也称为 素元→9.16.3@p1830, 它就相当于整数中的素数的概念. 下面的讨论将使用素理想及其相关概念, 不熟悉这一概念的读者, 可以将它们逐字逐句替换成素元, 那么这些论述在唯一分解定理成立的情形下也是对的.

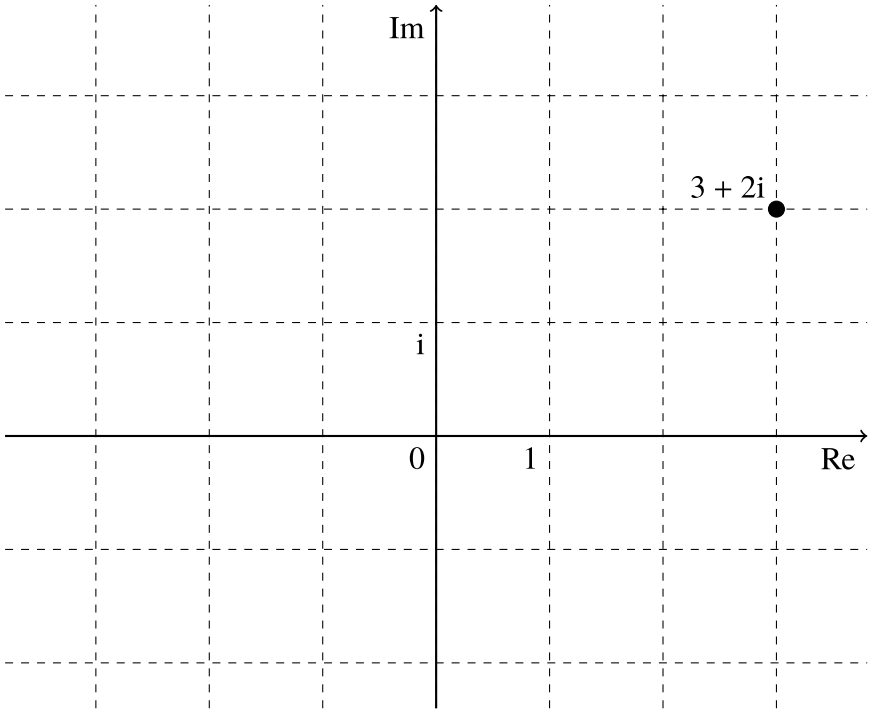
要完全地搞清楚一般的二次整数环 $\mathbf{Z}[\omega]$ 上的唯一分解, 需要知道 $\mathbf{Z}[\omega]$ 中的素理想都有哪些. 首先, $\mathbf{Z}[\omega]$ 中的每个素理想都整除它的范数 (的主理想). 将它的范数在整数内分解为若干素数的乘积, 则根据唯一分解定理, 该素理想必然整除这些素数因子中的某个 (的主理想). 因此, $\mathbf{Z}[\omega]$ 中的素理想一定是将整数中的素数进一步分解得到的. 要列举出 $\mathbf{Z}[\omega]$ 中的全部素理想, 只要给出 $\mathbf{Z}$ 中的素数 p (的主理想) 在 $\mathbf{Z}[\omega]$ 中的唯一分解就可以了. 因为素数 p (的主理想) 在 $\mathbf{Z}[\omega]$ 中的范数是 p^2, 而它分解成素理想意味着素理想的范数必然是 p^2 的因子, 故而只能是 p 或 p^2. 这说明只有如下三种可能性:

1. p 在 $\mathbf{Z}[\omega]$ 中是 **惯性** (inert) 的, 即 (p) 在 $\mathbf{Z}[\omega]$ 中仍然是素理想;
2. p 在 $\mathbf{Z}[\omega]$ 中 **分裂** (split), 即 (p) 在 $\mathbf{Z}[\omega]$ 中可以写成两个不同的共轭素理想的乘积;
3. p 在 $\mathbf{Z}[\omega]$ 中 **分歧** (ramify), 即 (p) 在 $\mathbf{Z}[\omega]$ 中可以写成某个素理想的平方.

进一步可以证明, 要判定某个素数 p 落入这三种情形中的哪一种, 只需要对 $\mathbf{Z}[\omega]$ 所在的二次域的判别式 D 和该素数 p 计算 Kronecker 符号[3] $\left(\dfrac{D}{p}\right)$ 就可以了. 这三种情形分别对应着 Kronecker 符号的三种取值: -1, $+1$ 和 0. 当 p 是奇素数时, Kronecker 符号就是 Legendre 符号→9.11.17@p1263, 这三种情形分别对应于 D 是 p 的 二次非剩余→9.11.17@p1263, D 是 p 的 二次剩余→9.11.17@p1263 以及 p 整除 D. 当 $p = 2$ 时, 这三种情形分别对应于 $D \equiv \pm 3 \pmod 8$, $D \equiv \pm 1 \pmod 8$ 和 2 整除 D.

Gauss 整数

本节中, 令 $\mathrm{i} = \sqrt{-1}$ 表示虚数单位. 二次域 $\mathbf{Q}(\mathrm{i})$ 也称为 Gauss 域, 它也是四次 分圆域→9.16.4@p1849. 相应的二次整数环 $\mathbf{Z}[\mathrm{i}]$ 也称为 Gauss 整数环, 其中的元素称为 **Gauss 整数** (Gaussian integer). Gauss 整数的单位数共四个, 即 ± 1 和 $\pm \mathrm{i}$, 因而每个非零的 Gauss 整数有四个相伴元 (含自身). 在复平面上, Gauss 整数表示的是全体整点, 而 Gauss 整数的范数 $N(a + b\mathrm{i}) = a^2 + b^2$ 就是复平面上的范数.



Gauss 整数上可以做带余除法：对于 Gauss 整数 a 和 $b \neq 0$，总存在 Gauss 整数 q 和 r 使得 $a = bq + r$ 成立，且 $N(r) < N(b)$. 要计算这样的带余除法，可以首先在 $\mathbf{Q}(\mathrm{i})$ 内计算 $\frac{a}{b}$，再在复平面内寻找最近的整点作为 q，然后计算 $r = a - bq$；这样得到的余数总是满足 $N(r) \leq \frac{1}{2}N(b)$. 利用带余除法，可以将辗转相除法（Euclid 算法）和 Bezout 定理等迁移到 Gauss 整数上，进而建立唯一分解定理.

Gauss 素数

利用上节的结论可以求出 Gauss 整数内的素元（也称为 Gauss 素数）. 因为 Gauss 整数环的判别式是 -4，而 Kronecker 符号 $\left(\frac{-4}{n}\right)$ 在偶数 n 上取值为 0，在奇数 n 上取值等于 $\left(\frac{-1}{n}\right) = (-1)^{(n-1)/2}$，故而有

$$\left(\frac{-4}{n}\right) = \begin{cases} +1, n \equiv 1 \pmod 4, \\ -1, n \equiv 3 \pmod 4, \\ 0, \quad 2 \mid n. \end{cases}$$

所以 Gauss 素数总共有如下三类：

1. 整数中 $4k+3$ 型素数；
2. 整数中 $4k+1$ 型素数的两个共轭的 Gauss 素数因子；
3. 素数 2 的因子 $1+\mathrm{i}$，它的共轭与它相伴.

比如在 $\mathbf{Z}[\mathrm{i}]$ 中，有分解 $60 = 2^2 \times 3 \times 5 = -(1+\mathrm{i})^4 \times 3 \times (2+\mathrm{i}) \times (2-\mathrm{i})$.

此处讨论素数 p 能够在 $\mathbf{Z}[\mathrm{i}]$ 中进一步分解，其实等价于讨论素数 p 能够写成整数平方和 $a^2 + b^2$ 的形式. 因此，由此处的结论可以推得，素数 p 可以写成平方和的形式，当且仅当 $p = 2$ 或 $p \equiv 1 \pmod 4$. 这就是 Fermat 平方和定理[4].

圆上整点问题

在复平面上，Gauss 整数表示了所有整点. Gauss 整数的范数就是格点到原点 Euclid 距离的平方. 因此，具有相同范数的二次整数对应着复平面以原点为圆心的圆上的整点. 也就是说，圆 $x^2 + y^2 = n$ 上的整点个数就是范数等于 n 的 Gauss 整数 $x + y\mathrm{i}$ 的个数.

要求解方程 $N(x + y\mathrm{i}) = n$，可以考虑 Gauss 整数 $x + y\mathrm{i}$ 分解成素因子，则这些素因子的范数的乘积就等于 n. 因此，只要首先得到 n 的因式分解，就能够根据 n 的素因子判断出 $x + y\mathrm{i}$ 可能具有的素因子. 设整数 n 的素因数分解为

$$n = 2^k p_1^{r_1} \cdots p_\ell^{r_\ell} q_1^{s_1} \cdots q_m^{s_m},$$

其中，$p_1, \cdots, p_\ell$ 为 $4k+1$ 型素因子，$q_1, \cdots, q_m$ 为 $4k+3$ 型素因子.

首先，方程 $N(x+y\mathrm{i}) = n$ 有解，当且仅当 n 的 $4k+3$ 型素因子的指数 $s_1, \cdots, s_m$ 都是偶数；这是因为 $q_1, \cdots, q_m$ 也是 Gauss 素数，它们的范数等于自身的平方，因此范数的素因数分解中它们必然成对出现.

现在假设方程有解. 那么，方程的解必然具有形式

$$u(1+\mathrm{i})^k (a_1+b_1\mathrm{i})^{r_1^+} (a_1-b_1\mathrm{i})^{r_1^-} \cdots (a_\ell+b_\ell\mathrm{i})^{r_\ell^+} (a_\ell-b_\ell\mathrm{i})^{r_\ell^-} q_1^{s_1/2} \cdots q_m^{s_m/2},$$

其中，u 是单位数，$a_j \pm b_j\mathrm{i}$ 是 p_j 在 Gauss 整数环中的共轭素因子，且 $r_j^+ + r_j^- = r_j$. 因此，方程的解的个数就等于

$$4(1+r_1)\cdots(1+r_\ell).$$

令 $f(n)$ 表示方程 $x^2+y^2=n$ 的整数解的个数. 当它有解时，$f(n)$ 由上述表达式给出；否则，$f(n)=0$. 容易验证，$\dfrac{1}{4}f(n)$ 是 积性函数→9.11.1@p1160. 积性函数的值由它在素数幂上的取值确定. 根据 $f(n)$ 的表达式可以确定，$\dfrac{1}{4}f(n)$ 在素数幂 p^k 上的取值如下：

1. 若 p 为 $4k+3$ 型素数，那么 $\dfrac{1}{4}f(1)=1, \dfrac{1}{4}f(p)=0, \dfrac{1}{4}f(p^2)=1, \dfrac{1}{4}f(p^3)=0, \cdots$；
2. 若 p 为 $4k+1$ 型素数，那么 $\dfrac{1}{4}f(p^k)=k+1$；
3. 若 $p=2$，那么 $\dfrac{1}{4}f(2^k)=1$.

容易验证，这三种情形都可以写成

$$\frac{1}{4}f(p^k) = \sum_{j=0}^{k} \left(\frac{-4}{p}\right)^j = \sum_{j=0}^{k} \left(\frac{-4}{p^j}\right) = \sum_{d \mid p^k} \left(\frac{-4}{d}\right).$$

由于 Kronecker 符号 $\left(\dfrac{-4}{n}\right)$ 是完全积性函数，可以得到

$$f(n) = 4\sum_{d \mid n} \left(\frac{-4}{d}\right) = 4\sum_{d \mid n} \chi_{4,3}(d).$$

最右侧的求和式的记号利用了 Kronecker 符号 $\left(\dfrac{-4}{n}\right)$ 是模 4 的实 Dirichlet 特征[5] 这一事实.

勾股方程

利用 Gauss 整数可以求出勾股方程的通解. 勾股方程指如下二次不定方程：

$$x^2 + y^2 = z^2.$$

对比上一节的内容可知，这相当于求解方程 $N(x+y\mathrm{i}) = z^2$. 设 z 在 $\mathbf{Z}$ 中有素因子分解

$$z = 2^k p_1^{r_1} \cdots p_\ell^{r_\ell} q_1^{s_1} \cdots q_m^{s_m},$$

因而它的解 $x+y\mathrm{i}$ 在 $\mathbf{Z}[\mathrm{i}]$ 中有素因子分解

$$x+y\mathrm{i} = u(1+\mathrm{i})^{2k} (a_1+b_1\mathrm{i})^{r_1^+} (a_1-b_1\mathrm{i})^{r_1^-} \cdots (a_\ell+b_\ell\mathrm{i})^{r_\ell^+} (a_\ell-b_\ell\mathrm{i})^{r_\ell^-} q_1^{s_1} \cdots q_m^{s_m}$$

且 $r_j^+ + r_j^- = 2r_j$. 可以计算 $x+y\mathrm{i}$ 和 z 的最大公因子，得到如下整数：

$$\kappa = 2^k p_1^{\min\{r_1^+, r_1^-\}} \cdots p_\ell^{\min\{r_\ell^+, r_\ell^-\}} q_1^{s_1} \cdots q_m^{s_m}.$$

消去这一公因子，那么 $\kappa^{-1}(x+y\mathrm{i})$ 中只含有 $a_j \pm b_j\mathrm{i}$ 形式的素因子，共轭的因子不会成对出现，且这些因子的指数 $|\, r_j^+ - r_j^- \,|$ 必然是偶数（因为它们的和 $2r_j$ 是偶数）. 因此，$\kappa^{-1}(x+y\mathrm{i})$ 是某个二次整数 $u+v\mathrm{i}$ 的平方. 由此，得到如下方程：

$$x+y\mathrm{i} = \kappa(u+v\mathrm{i})^2,\ z = \kappa N(u+v\mathrm{i}).$$

在整数 $\mathbf{Z}$ 内，这相当于如下方程组：

$$x = \kappa(u^2-v^2),\ y = 2\kappa uv,\ z = \kappa(u^2+v^2).$$

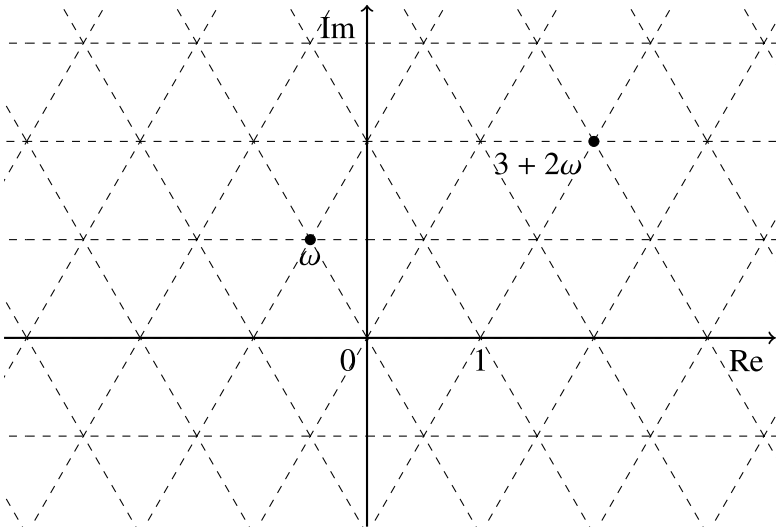
反过来，对于任何整数 u 和 v，上述表达式得到的 (x, y, z) 都满足勾股方程. 因此，这就是勾股方程的通解.

根据上面的过程可以知道，本原勾股数 (x,y,z)（即 x,y,z 公因子为一）中的 x,y 必然一奇一偶，z 是奇数且只含有 $4k+1$ 型素因子.

利用类似的方法还可以得到方程 $x^2+y^2=z^3$ 的通解，或者证明方程 $x^4+y^4=z^4$ 无解. 当然，利用勾股方程的通解和无穷递降法，可以证明更强的结论，即方程 $x^4+y^4=z^2$ 无解.

Eisenstein 整数

本节中，令 $\omega=\dfrac{-1+\sqrt{3}\mathrm{i}}{2}=e^{2\pi\mathrm{i}/3}$.[70] 二次域 $\mathbf{Q}(\sqrt{3}\mathrm{i})$ 是三次和六次 分圆域→9.16.4@p1849，其中的代数整数称为 Eisenstein 整数. 全体 Eisenstein 整数构成的环 $\mathbf{Z}[\omega]$ 称为 Eisenstein 整数环. Eisenstein 整数环的单位数共六个，分别是 ±1，$\pm\omega$ 和 $\pm\omega^2$. 在复平面上，所有 Eisenstein 整数构成三角形的格点. 与 Gauss 整数不同，此处的格点一般不是整点.



Eisenstein 整数的范数为

$$N(a+b\omega)=a^2-ab+b^2,$$

它也是复平面上上述格点到原点的距离的平方.

Eisenstein 整数与 Gauss 整数也十分相似. 在 Eisenstein 整数上同样可以利用范数 $N(\cdot)$ 定义带余除法，并建立辗转相除法、Bezout 定理、唯一分解定理等结论. 类似于上文，可以推导出素数在 Eisenstein 整数环中的因子. 为此，注意到 $\mathbf{Z}[\omega]$ 的判别式是 -3，而 Kronecker 符号

$$\left(\frac{-3}{n}\right)=\begin{cases}+1, & n\equiv 1 \pmod 3,\\ -1, & n\equiv 2 \pmod 3,\\ 0, & 3\mid n,\end{cases}$$

因此，Eisenstein 整数环内的素元，亦即 Eisenstein 素数，有如下三类：

1. 整数中 $3k+2$ 型素数，即 2 和 $6k+5$ 型素数；
2. 整数中 $3k+1$ 型素数（也就是 $6k+1$ 型素数）的两个共轭的 Eisenstein 素数因子；
3. 素数 3 的因子 $(3+\sqrt{3}\mathrm{i})/2$，它的共轭与它相伴.

利用上一节的方法，可以求出范数为 n 的 Eisenstein 整数的数目. 设这个数目为 $f(n)$，则它就等于

$$f(n)=6\sum_{d\mid n}\left(\frac{-3}{d}\right)=6\sum_{d\mid n}\chi_{3,2}(d).$$

其中，$\chi_{3,2}(n)=\left(\dfrac{-3}{n}\right)$ 是模 3 的实 Dirichlet 特征. 该表达式说明，这样的 Eisenstein 整数存在，当且仅当 n 的全部 $3k+2$ 型素因子的指数都是偶数.

[70] 注意，此处 ω 的选择与上文的选择并不相同. 按照惯例，在讨论一般的二次整数环时常规定 $\omega=(1+\sqrt{d})/2$（当 $d\equiv 1 \pmod 4$ 时），而对于 Eisenstein 整数环又往往规定 $\omega=(-1+\sqrt{-3})/2$. 这个区别没有任何实质影响，但是可能会引起部分表达式的形式改变. 注意，不应将 Eisenstein 整数环写成 $\mathbf{Z}[\sqrt{-3}]$.

利用前文的范数公式，$f(n)$ 也是不定方程 $x^2-xy+y^2=n$ 或不定方程 $x^2+xy+y^2=n$ 的整数解的数目．它对应的几何意义是斜椭圆 $x^2\pm xy+y^2=n$ 上的整点数目．

与这些方程密切相关的是另一个不定方程 $x^2+3y^2=n$，它的整数解对应着标准椭圆上的整点．做变量代换 $x=(u+v)/2$ 和 $y=(u-v)/2$ 可以将它转化为已经解决的方程 $u^2-uv+v^2=n$．但是，虽然 $x^2+3y^2=n$ 的整数解总对应着 $u^2-uv+v^2=n$ 的整数解，反过来却不一定正确．如果 n 是偶数，那么 u 和 v 必然都是偶数，这意味着 x 和 y 必然是整数，此时 $x^2+3y^2=n$ 的整数解的数目仍然是 $f(n)$．但是，如果 n 是奇数，那么 u 和 v 可能一奇一偶，也可能都是奇数，因而相对应的 x 和 y 可能都是半整数或者都是整数．这种情形需要更细致的讨论．

当已知方程 $u^2-uv+v^2=n$ 一个整数解 (u,v) 时，与 $u+v\omega$ 相伴的 Eisenstein 整数（含自身）对应的整数解分别是

$$(u,v),(u-v,u),(-v,u-v),(-u,-v),(v-u,-u),(v,v-u).$$

因为 $u,v,u-v$ 三个整数只要不全是偶数，必然是两个奇数一个偶数，所以这六组整数解中必然有两组全是奇数，四组一奇一偶．这说明，当 n 为奇数时，方程 $u^2-uv+v^2=n$ 的整数解中只有 $\frac{1}{3}$ 全是奇数，此时它才能够对应 $x^2+3y^2=n$ 的整数解．因而，当 n 为奇数时，$x^2+3y^2=n$ 的整数解的数目为 $\frac{1}{3}f(n)$．

最后，类似于解决勾股方程的方法，可以利用 Eisenstein 整数解决如下不定方程：

$$x^2-xy+y^2=z^2$$

$$x^2+xy+y^2=z^2$$

$$x^2+3y^2=z^2$$

$$x^2+3y^2=z^3$$

它们的通解在此不再赘述．类似的方法还可以用于证明 $x^3+y^3=z^3$ 无解．

参考文献与注释

- Quadratic field - Wikipedia[6]
- Quadratic integer - Wikipedia[7]
- Gaussian integer - Wikipedia[8]
- Eisenstein integer - Wikipedia[9]
- Kronecker symbol - Wolfram MathWorld[10]
- Dirichlet character - Wikipedia[11]
- Theodorus J. Dekker's Notes on Primes in Quadratic Fields[12]
- Franz Lemmermeyer's Notes on Ideals in Quadratic Number Fields[13]
- J.S. Milne - Algebraic Number Theory[14]

[1] Dirichlet 单位定理

[2] Dedekind 整环

[3] Kronecker 符号

[4] Fermat 平方和定理

[5] Dirichlet 特征

[6] Quadratic field - Wikipedia

[7] Quadratic integer - Wikipedia

[8] Gaussian integer - Wikipedia

[9] Eisenstein integer - Wikipedia

[10] Kronecker symbol - Wolfram MathWorld

[11] Dirichlet character - Wikipedia

[12] Theodorus J. Dekker's Notes on Primes in Quadratic Fields

[13] Franz Lemmermeyer's Notes on Ideals in Quadratic Number Fields

[14] J.S. Milne - Algebraic Number Theory

9.11.33 Pell 方程

前置知识：连分数→9.11.30@p1431、二次域→9.11.32@p1495

引入

本文讨论（广义）Pell 方程的求解．广义 Pell 方程是指关于 x 和 y 的不定方程

$$x^2 - Dy^2 = N,$$

其中，D 是正整数且不是完全平方数[71]，N 是非零整数．狭义的 Pell 方程特指 $N = 1$ 或 $N = \pm 1$ 的特殊情形，有时也包括 $N = \pm 4$ 的情形．广义 Pell 方程与在实二次整数环内寻找范数为 N 的二次整数紧密相关，而常常称作（狭义）Pell 方程的这些情形可以看作是寻找实二次整数环内的单位数．

当本文提及 Pell 方程时，特指 $N = 1$ 的情形．相应地，$N = -1$ 的情形称为负 Pell 方程[72]（negative Pell's equation）．

解的结构

广义 Pell 方程的整数解 (x, y) 和二次整数 $x + y\sqrt{D}$ 联系密切，因此文献中 Pell 方程的解常常写作 $x + y\sqrt{D}$ 的形式．因为二次整数的范数

$$N(x + y\sqrt{D}) = x^2 - Dy^2,$$

所以，广义 Pell 方程大致相当于在求解范数为 N 的二次整数．但是，两者确实有细微的区别．当 x 和 y 都是整数时，$x + y\sqrt{D}$ 一定是二次整数；反过来，二次整数未必要求 x 和 y 都是整数——在 $D \equiv 1 \pmod 4$ 的情形，x 和 y 还可以同时是半整数[73]．

这个区别在求解基本单位数时格外重要．因为二次整数环中的单位数是指范数为 ± 1 的二次整数．对于 $D \equiv 2, 3 \pmod 4$，要找到这样的单位数，只需要求解广义 Pell 方程在 $N = \pm 1$ 的情形即可；但是对于 $D \equiv 1 \pmod 4$，还需要考虑 $N = \pm 4$ 的情形．下文→9.11@p1160 会讨论单位数的求解方法．

要理解广义 Pell 方程解的结构，需要从 Brahmagupta 恒等式[1] 入手：

$$(x_1^2 - Dy_1^2)(x_2^2 - Dy_2^2) = (x_1x_2 + Dy_1y_2)^2 - D(x_1y_2 + x_2y_1)^2.$$

它相当于二次整数的范数保持乘法，即

$$\begin{aligned} N\left(x_1 + y_1\sqrt{D}\right)N\left(x_2 + y_2\sqrt{D}\right) &= N\left((x_1 + y_1\sqrt{D})(x_2 + y_2\sqrt{D})\right) \\ &= N\left((x_1x_2 + Dy_1y_2) + (x_1y_2 + x_2y_1)\sqrt{D}\right). \end{aligned}$$

利用这一恒等式，可以利用方程 $x^2 - Dy^2 = N_1$ 和方程 $x^2 - Dy^2 = N_2$ 的整数解复合出方程 $x^2 - Dy^2 = N_1N_2$ 的整数解．当然，从二次整数的角度看，解的复合就是二次整数的乘法，这就体现了将 Pell 方程的解记作二次整数形式的方便之处．特别地，取 $N_1 = N$ 和 $N_2 = 1$ 就可以发现，如果已知 $x^2 - Dy^2 = N$ 的一组解和相应的 Pell 方程 $x^2 - Dy^2 = 1$ 的全体解，就可以得到 $x^2 - Dy^2 = N$ 更多的解．当然，这种方法未必能够生成全部的解．但是，这至少说明理解 Pell 方程的解的结构对理解广义 Pell 方程的解的结构有重要作用．

Pell 方程

方程 $x^2 - Dy^2 = 1$ 的几何意义是实轴在 x 轴、虚轴在 y 轴的双曲线．双曲线上的每个点都唯一对应了 $x + y\sqrt{D}$ 的一个非零取值：双曲线的左支对应着 $x + y\sqrt{D}$ 的负值，右支则对应正值．而且，在每一支上，双曲线自下而上对应的 $x + y\sqrt{D}$ 的取值是严格递增的．二次整数的取值给 Pell 方程的解赋予了自然的顺序．

---

[71] 当 D 是完全平方数时，直接做因式分解可知，$(x + y\sqrt{D})(x - y\sqrt{D}) = N$，因此所有解可以通过遍历 N 的因数得知．特别地，当 $N = 1$ 时，方程只有解 $(\pm 1, 0)$；当 $N = -1$ 且 $D \neq 1$ 时，方程没有解．

[72] 有些中文文献也称它为第二型 Pell 方程．

[73] 即形如 $n + \frac{1}{2}$ 且 $n \in \mathbf{Z}$ 的有理数．

双曲线同时关于 x 轴和 y 轴对称，因此讨论 Pell 方程的解只需要考虑在第一象限内的那一段即可，其余解可以通过对称性获得．这相当于只考虑 $x+y\sqrt{D}>1$ 的解．如果方程除了 $(\pm1,0)$ 之外还存在不平凡的解，那么一定在第一象限内存在 $x+y\sqrt{D}$ 取值最小的解 (x_1,y_1)，这也是第一象限内（不含坐标轴）横纵坐标都最小的整点，它称为 Pell 方程的基本解（fundamental solution）[74]．根据前文的讨论，满足 $x_k+y_k\sqrt{D}=(x_1+y_1\sqrt{D})^k$ 的整数对 (x_k,y_k) 都是 Pell 方程的解，而且都在第一象限．反过来，这也的确是 Pell 方程在第一象限内的全部解．再利用对称性，就可以得到如下结论：

定理

设 Pell 方程 $x^2-Dy^2=1$ 的基本解是 (x_1,y_1)．那么，它的全部解就是

$$\{(x,y):x+y\sqrt{D}=\pm(x_1+y_1\sqrt{D})^k,k\in\mathbf{Z}\}.$$

证明

首先证明第一象限中不存在其他解．不妨设存在其他解 $x+y\sqrt{D}$ 且对于某个 $k\geq0$ 有

$$x_k+y_k\sqrt{D}<x+y\sqrt{D}<x_{k+1}+y_{k+1}\sqrt{D}.$$

几何意义上，这相当于说整点 (x,y) 落入了双曲线上 (x_k,y_k) 和 (x_{k+1},y_{k+1}) 之间（不含端点）．将不等式同时乘以 $x_k-y_k\sqrt{D}=(x_k+y_k\sqrt{D})^{-1}$ 就得到

$$1<(x+y\sqrt{D})(x_k-y_k\sqrt{D})=(xx_k-Dyy_k)+(x_ky-xy_k)\sqrt{D}<x_1+y_1\sqrt{D}.$$

根据前文提到的单调性，这个不等式就说明 (xx_k-Dyy_k,x_ky-xy_k) 是位于 $(1,0)$ 和 (x_1,y_1) 之间的整点．这与 (x_1,y_1) 的选取矛盾．

将第一象限的解扩展到整个平面时，指数 k 取相反数（即整体取倒数）就是关于 x 轴对称，整体取相反数则是关于原点对称．再加上 $k=0$ 时的平凡解，就得到 Pell 方程的全部解．

前文的讨论只是假设了基本解的存在．现在要说明的是，Pell 方程总是存在非平凡的解．

定理

Pell 方程 $x^2-Dy^2=1$ 总是存在除了 $(\pm1,0)$ 之外的整数解．

证明

首先，Dirichlet 定理→9.11.30@p1431 表明存在无数对正整数 (x,y) 使得

$$\left|\frac{x}{y}-\sqrt{D}\right|\leq\frac{1}{y^2}$$

成立，它们都满足不等式

$$|x^2-Dy^2|=y^2\left|\frac{x}{y}-\sqrt{D}\right|\left|\frac{x}{y}+\sqrt{D}\right|\leq\frac{1}{y^2}+2\sqrt{D}<1+2\sqrt{D}.$$

因而，必然存在整数 $m\in(-1-2\sqrt{D},1+2\sqrt{D})$ 使得有无数对 (x,y) 都满足 $x^2-Dy^2=m$．将这些 (x,y) 根据对 m 的余数分类，就知道对某一对整数 (x_0,y_0)，一定存在无数对 (x,y) 使得 $x\equiv x_0\pmod m$ 以及 $y\equiv y_0\pmod m$ 成立．任取满足这些条件的两对互异的 (x_1,y_1) 和 (x_2,y_2)，则

$$\frac{x_1+y_1\sqrt{D}}{x_2+y_2\sqrt{D}}=\frac{x_1x_2-Dy_1y_2}{m}+\frac{x_2y_1-x_1y_2}{m}\sqrt{D}.$$

因为根据同余关系有

[74]注意，Pell 方程中基本解的定义与实二次整数环中的基本单位数的定义并不一致．首先，部分实二次整数环中的基本单位数 $x+y\sqrt{D}$ 中的 x,y 是半整数，因而并非 Pell 方程的解．其次，同一个实二次整数环的基本单位数有四个，但是基本解只有一个，因为基本解要求 x,y 都是正数．

$$x_1x_2 - Dy_1y_2 \equiv x_0^2 - Dy_0^2 = m \equiv 0 \pmod{|\,m\,|},$$
$$x_2y_1 - x_1y_2 \equiv x_0y_0 - x_0y_0 = 0 \pmod{|\,m\,|},$$

这说明上式右侧得到的是整数解．而且，因为 $(x_1, y_1) \neq (x_2, y_2)$，它并不平凡．这就说明 Pell 方程的确存在非平凡解．

当然，本节提供的是非构造性的证明，在下文讨论 Pell 方程的解法时，会直接利用连分数的渐近分数构造出 Pell 方程的解，因而提供了 Pell 方程存在非平凡解的另一种证明．另外，尽管本节得到的 Pell 方程解的结构与实二次整数环的单位数的结构是一致的，但是对于 $D \equiv 1 \pmod 4$ 的情形，本节并没有完全解决相应的二次整数环的单位数的结构问题，下文将进一步讨论．

广义 Pell 方程

广义 Pell 方程 $x^2 - Dy^2 = N$ 的图像同样是平面上的双曲线，同样以 x 轴和 y 轴为对称轴．前文已经指出，方程 $x^2 - Dy^2 = N$ 的部分解可能只相差一个 Pell 方程解的因子，这意味着可以将方程 $x^2 - Dy^2 = N$ 的解划分为等价类．对于方程 $x^2 - Dy^2 = N$ 的两个解 (x_1, y_1) 和 (x_2, y_2)，如果存在 Pell 方程的解 (u, v) 使得 $x_2 + y_2\sqrt{D} = (x_1 + y_1\sqrt{D})(u + v\sqrt{D})$ 成立，那么称解 (x_1, y_1) 和 (x_2, y_2) 等价．两个解等价的充分必要条件是

$$N \mid (x_1x_2 - Dy_1y_2),\ N \mid (x_2y_1 - x_1y_2).$$

因为 Pell 方程的解相对容易求出，一个自然的想法是在上述的每个等价类中各求出一个解．只要知道这些解，就可以利用相应的 Pell 方程的解得到所要求的广义 Pell 方程的全部解．在广义 Pell 方程的解的等价类中，由于对称性，每个等价类都存在纵坐标 y 非负但是尽可能小的解：如果这样的解唯一，它就称为该等价类的基本解；否则，该等价类必然有两个 y 非负且最小的解，而且它们关于 y 轴对称，此时选择 $x > 0$ 的那个作为基本解．由此，求解广义 Pell 方程 $x^2 - Dy^2 = N$，就相当于求解它的基本解集 U．设它对应的 Pell 方程的基本解是 (r, s)，则广义 Pell 方程的全部解的集合是

$$\{(x, y) : x + y\sqrt{D} = \pm(r + s\sqrt{D})^k(u + v\sqrt{D}), k \in \mathbf{Z}, u + v\sqrt{D} \in U\}.$$

广义 Pell 方程的基本解必然是有限的．这是因为从上面的通解表达式可知，绝对值 $|\,u + v\sqrt{D}|$ 必然位于 $r - s\sqrt{D}$ 和 $r + s\sqrt{D}$ 之间．文末的参考文献中提供了关于基本解的坐标的范围的更严格的估计．当然，与 Pell 方程的情形不同，广义 Pell 方程可能没有解．

利用广义 Pell 方程的一个解 (u, v) 和 Pell 方程的基本解 (r, s) 得到同一个等价类中的全部解的方法，除了利用解的复合之外，还可以利用如下递推关系

$$x_k = 2rx_{k-1} - x_{k-2},\ y_k = 2ry_{k-1} - y_{k-2},$$

其中，$x_k + y_k\sqrt{D} = (r + s\sqrt{D})^k(u + v\sqrt{D})$．这是因为 x_k 和 y_k 都可以对某一对实数 (A, B) 写成 $A(r + s\sqrt{D})^k + B(r - s\sqrt{D})^k$ 的形式，而根据 Vieta 定理，$r \pm s\sqrt{D}$ 是方程 $x^2 - 2rx + 1 = 0$ 的两个实根，进而 x_k 和 y_k 都满足上述的二阶常系数递推关系．相较于解的复合，该递推公式有更少的乘法次数．

求解方法

Pell 方程和广义 Pell 方程的求解都可以基于连分数进行．

PQa 算法

本文讨论的算法都基于 PQa 算法，它可以用于求出特定的二次无理数的连分数展开．

设整数 P_0, Q_0, D 满足 $Q_0 \neq 0$，$D > 0$ 且不是完全平方数，以及 $P_0^2 \equiv D \pmod{Q_0}$．那么，二次无理数

$$\omega = \frac{P_0 + \sqrt{D}}{Q_0}$$

的连分数展开 $[a_0, a_1, \cdots]$ 可以通过如下 递推公式→9.11.30@p1431 求得：

$$a_k=\left\lfloor\frac{P_k+\sqrt{D}}{Q_k}\right\rfloor,\ P_{k+1}=a_kQ_k-P_k,\ Q_{k+1}=\frac{D-P_{k+1}^2}{Q_k}.$$

进而，ω 的第 k 个渐近分数的分子和分母 A_k 和 B_k 由如下 递推公式→9.11.30@p1431 给出：

$$A_k=a_kA_{k-1}+A_{k-2},\ B_k=a_kB_{k-1}+B_{k-2}$$

且 $A_{-1}=1$，$A_{-2}=0$，$B_{-1}=0$，$B_{-2}=1$.

这些公式的正确性已经在连分数一文中得到证明．那里也说明了，因为二次无理数是 循环连分数→9.11.30@p1431，所以，三元组 (P_k,Q_k,a_k) 最终将进入循环，算法总可以在有限步内终止．不妨设循环节的最小长度是 ℓ，且循环的最早的起始位置是 k_0，则二次无理数的连分数展开可以写作

$$\omega=[a_0,\cdots,a_{k_0-1},\overline{a_{k_0},\cdots,a_{k_0+\ell-1}}].$$

利用 PQa 算法解决 Pell 方程，需要建立如下结论：

定理

继续上述记号．设 $G_k=Q_0A_k-P_0B_k$，则整数对 (G_{k-1},B_{k-1}) 满足关系式

$$G_{k-1}^2-DB_{k-1}^2=(-1)^kQ_0Q_k,$$

且它们的最大公因数 $\gcd(G_{k-1},B_{k-1})$ 整除 Q_k.

证明

设 ω 的连分数展开中，第 k 个余项（完全商）为 ω_k，即

$$\omega=[a_0,a_1,\cdots,a_{k-1},\omega_k]=\frac{\omega_kA_{k-1}+A_{k-2}}{\omega_kB_{k-1}+B_{k-2}}.$$

将 $\omega=(P_0+\sqrt{D})/Q_0$ 和 $\omega_k=(P_k+\sqrt{D})/Q_k$ 代入上式，就得到

$$\frac{P_0+\sqrt{D}}{Q_0}=\frac{(P_k+\sqrt{D})A_{k-1}+Q_kA_{k-2}}{(P_k+\sqrt{D})B_{k-1}+Q_kB_{k-2}}.$$

消去左右两侧的分母并比较有理部分和无理部分的系数，再代入 G_k 的表达式，就得到如下等式：

$$G_{k-1}=P_kB_{k-1}+Q_kB_{k-2},$$

$$DB_{k-1}=P_kG_{k-1}+Q_kG_{k-2}.$$

因此，将一式乘以 G_{k-1} 减去二式乘以 B_{k-1}，就有

$$\begin{aligned}G_{k-1}^2-DB_{k-1}^2&=(B_{k-2}G_{k-1}-B_{k-1}G_{k-2})Q_k\\&=(A_{k-1}B_{k-2}-B_{k-1}A_{k-2})Q_0Q_k\\&=(-1)^kQ_0Q_k.\end{aligned}$$

最后一步利用了渐近分数的 差分公式→9.11.30@p1431．这就证明了第一个结论.

为了证明第二个结论，将 G_k 的表达式代入第一个结论，有

$$(Q_0A_{k-1}-P_0B_{k-1})^2-DB_{k-1}^2=(-1)^kQ_0Q_k.$$

所以，利用 $Q_0\mid(P_0^2-D)$ 有

$$Q_0A_{k-1}^2+\left(\frac{P_0^2-D}{Q_0}B_{k-1}-2P_0A_{k-1}\right)B_{k-1}=(-1)^kQ_k.$$

故而，$\gcd(G_{k-1},B_{k-1})=\gcd(Q_0A_{k-1},B_{k-1})$ 整除 Q_k.

这个结论提供了一种寻找方程 $x^2-Dy^2=N$ 的解的方法．如果合理地选择 $Q_0>0$ 并选择 P_0 为同余方程 $P_0^2\equiv D\pmod{Q_0}$ 的解，那么通过对 $(P_0+\sqrt{D})/Q_0$ 执行 PQa 算法，直到找到 $(-1)^kQ_0Q_k=N$，此时 (G_{k-1},B_{k-1}) 就成为原方程的一组解．而且，如果 $Q_k=\pm1$，那么这样得到的解一定是本原解，也就是说 G_{k-1} 和 B_{k-1} 一定是互素的.

这个思想是解决 Pell 方程和广义 Pell 方程的核心．理解了这一思想后，下面就着手处理算法的一些细节，并证明所有的解都可以通过该方式得到．

Pell 方程

要解决 Pell 方程 $x^2-Dy^2=1$，只需要对 $(P_0,Q_0,D)=(0,1,D)$ 运行 PQa 算法，直到出现 $(-1)^kQ_k=1$，此时 (A_{k-1},B_{k-1}) 就是 Pell 方程的一组解（因为 G_{k-1} 此时就是 A_{k-1}）．当然，对于 Pell 方程，对该过程可以进行更精确的描述．

首先，解一定出现在循环节的末尾处．上述过程相当于对 $\sqrt{D}$ 做连分数展开．对此，已经有 结论→9.11.30@p1431：

$$\sqrt{D}=[\lfloor\sqrt{D}\rfloor,\overline{a_1,\cdots,a_{\ell-1},2\lfloor\sqrt{D}\rfloor}].$$

此处，循环节长度为 ℓ，且起始位置是第 1 项（下标从 0 开始）．而且，它的第 ℓ 个余项等于 $\lfloor\sqrt{D}\rfloor+\sqrt{D}$，这说明 $Q_\ell=1$．因此，如果 ℓ 是偶数，那么 $(A_{\ell-1},B_{\ell-1})$ 就是 Pell 方程的一组非平凡解；如果 ℓ 是奇数，那么 $(A_{2\ell-1},B_{2\ell-1})$ 就是 Pell 方程的一组非平凡解．

接下来要说明，刚刚得到的这组解一定是基本解．这个结论基于两点理由：第一，Pell 方程的所有正整数解 (x,y) 对应的分数 x/y 都出现在 $\sqrt{D}$ 的渐近分数中，这就保证了 (x,y) 必定是 PQa 算法过程中的某个 (A_k,B_k)；第二，除了循环节末尾，不会再出现其他位置有 $Q_k=1$，因为 A_k 和 B_k 的递推关系保证了它们的大小随着下标增加而增加，所以最小的正整数解（即基本解）必然出现在刚刚指明的位置．这两点理由可以分别从如下的两条定理得出：

定理

设方程 $x^2-Dy^2=N$ 有正整数解 (x,y)，如果 $|N|<\sqrt{D}$，那么 $\dfrac{x}{y}$ 一定等于 $\sqrt{D}$ 的渐近分数．

证明

当 $N>0$ 时，因为 $x^2-Dy^2>0$，所以 $x>y\sqrt{D}$．故而，有

$$\left|\frac{x}{y}-\sqrt{D}\right|=\frac{N}{y(x+y\sqrt{D})}<\frac{N}{2y^2\sqrt{D}}<\frac{1}{2y^2}.$$

根据 Legendre 判别法→9.11.30@p1431 可知，$\dfrac{x}{y}$ 是 $\sqrt{D}$ 的渐近分数．

当 $N<0$ 时，$x>y\sqrt{D}$ 不再成立．所以，转而考虑方程 $y^2-\dfrac{1}{D}x^2=-\dfrac{N}{D}$ 的解．因为 $\dfrac{|N|}{D}<\sqrt{\dfrac{1}{D}}$，所以上面的论证依然成立．这说明 $\dfrac{y}{x}$ 是 $\dfrac{1}{\sqrt{D}}$ 的渐近分数．根据 倒数定理→9.11.30@p1431 可知，$\dfrac{x}{y}$ 也是 $\sqrt{D}$ 的渐近分数．

定理

在对 $(P_0,Q_0,D)=(0,1,D)$ 运行上述 PQa 算法的过程中，$Q_k=1$ 必然推出 $\ell\mid k$．

证明

在 $\sqrt{D}$ 的连分数展开中，除了第 0 个余项，所有其他余项都是 纯循环连分数→9.11.30@p1431．设 $Q_k=1$．根据 Galois 的结论，必然有余项 $\omega_k=P_k+\sqrt{D}>1$，且它的共轭 $-1<P_k-\sqrt{D}<0$，这说明 $P_k=\lfloor\sqrt{D}\rfloor$．因此，余项 ω_k 就等于 ω_ℓ．但是，余项的重复意味着连分数的循环，如果 k 不是 ℓ 的整数倍，就与 ℓ 是最小正周期相矛盾．所以，必然有 $\ell\mid k$．

综合本节的讨论可知，只要对 $\sqrt{D}$ 做连分数展开，亦即以 $(P_0,Q_0,D)=(0,1,D)$ 为起点做 PQa 算法，当首次得到 $Q_\ell=1$ 时，就到达了第一个循环节的末尾．此时，如果 ℓ 是偶数，那么 $(A_{\ell-1},B_{\ell-1})$ 就是 Pell

方程的基本解；否则，$(A_{2\ell-1}, B_{2\ell-1})$ 是 Pell 方程的基本解．对于循环节长度 ℓ 为奇数的情形，并不需要继续 PQa 算法到两倍的循环节处，马上就会说明 $A_{2\ell-1}+B_{2\ell-1}\sqrt{D}=(A_{\ell-1}+B_{\ell-1}\sqrt{D})^2$，因而可以直接从 $(A_{\ell-1}, B_{\ell-1})$ 直接计算出 Pell 方程的基本解．Pell 方程所有其他解都可以通过 Pell 方程的基本解计算．

示例

1. 求解方程 $x^2-14y^2=1$.

   对 $(P_0, Q_0, D)=(0,1,14)$ 运行 PQa 算法结果如下：（标红部分为第一个循环节）

k	P	Q	a	A	B	G	G^2-DB^2
0	0	1	3	3	1	3	-5
1	3	5	1	4	1	4	2
2	2	2	2	11	3	11	-5
3	2	5	1	15	4	15	1
4	3	1	6	101	27	101	-5
5	3	5	1	116	31	116	2

   循环节长度 $\ell=4$ 为偶数．方程的最小正整数解为 $(G_3, B_3)=(15,4)$.

2. 求解方程 $x^2-41y^2=1$.

   对 $(P_0, Q_0, D)=(0,1,41)$ 运行 PQa 算法结果如下：（标红部分为第一个循环节）

k	P	Q	a	A	B	G	G^2-DB^2
0	0	1	6	6	1	6	-5
1	6	5	2	13	2	13	5
2	4	5	2	32	5	32	-1
3	6	1	12	397	62	397	5
4	6	5	2	826	129	826	-5
5	4	5	2	2049	320	2049	1
6	6	1	12	25414	3969	25414	-5
7	6	5	2	52877	8258	52877	5

   循环节长度 $\ell=3$ 为奇数．方程的最小正整数解为 $(G_5, B_5)=(2049, 320)$．它也可以通过 $(G_2, B_2)=(32,5)$ 计算得出：

   $(32+5\sqrt{41})^2=2049+320\sqrt{41}$.

负 Pell 方程

根据上一节的讨论可知，负 Pell 方程的解也必然对应于 $\sqrt{D}$ 的渐近分数，而且只能出现在 $(-1)^kQ_k=-1$ 处．这只能出现在循环节的末尾．因此，负 Pell 方程有解，当且仅当循环节长度 ℓ 是奇数．当解存在时，$(A_{\ell-1}, B_{\ell-1})$ 就是负 Pell 方程的基本解．它的求解方法和上一节是一致的．

利用前文对于 Pell 方程解的结构的证明相仿的思路，可以证明如下结论：

定理

设方程 $x^2-Dy^2=-1$ 有解，且基本解是 (x_1, y_1)．那么，$x^2-Dy^2=\pm1$ 的所有整数解都属于集合

$$\{(x,y): x+y\sqrt{D}=\pm(x_1+y_1\sqrt{D})^k, k\in\mathbf{Z}\}.$$

特别地，满足 $x_2+y_2\sqrt{D}=(x_1+y_1\sqrt{D})^2$ 的整数解 (x_2,y_2) 正是 $x^2-Dy^2=1$ 的基本解.

证明

由于对称性，只需要考虑正整数解，即 $x+y\sqrt{D}>1$ 的情形．但是由于 $x^2-Dy^2=\pm1$ 是两段双曲线，所以 $x+y\sqrt{D}$ 无法和 (x,y) 建立一一对应．为了处理这种困难，首先证明上述的 (x_2,y_2) 是 $x^2-Dy^2=1$ 的基本解.

显然，(x_2,y_2) 是 $x^2-Dy^2=1$ 的解．如果设 (z,w) 是 $x^2-Dy^2=1$ 的基本解，那么必然有 $1<z+w\sqrt{D}\le x_2+y_2\sqrt{D}$．如果右侧的不等式不含有等号，那么将不等式同除以 $x_1+y_1\sqrt{D}$ 就得到 $-x_1+y_1\sqrt{D}<(z+w\sqrt{D})(-x_1+y_1\sqrt{D})<x_1+y_1\sqrt{D}$．将该不等式的中间项的表达式展开就能得到 $x'+y'\sqrt{D}$ 的形式，它的范数是 -1 且 (x',y') 也是整数解．将该不等式取倒数，就发现 $-x'+y'\sqrt{D}$ 同样落入 $-x_1+y_1\sqrt{D}$ 和 $x_1+y_1\sqrt{D}$ 之间．二次整数 $\pm x'+y'\sqrt{D}$ 互为倒数，必然有一个大于 1．但是 1 和 $x_1+y_1\sqrt{D}$ 之间不应该再出现别的范数为 -1 的二次整数，这与 $x_1+y_1\sqrt{D}$ 的最小性矛盾．所以，必然成立 $x_2+y_2\sqrt{D}=z+w\sqrt{D}$，即 (x_2,y_2) 是方程 $x^2-Dy^2=1$ 的基本解.

基于此，如果出现方程 $x^2-Dy^2=\pm1$ 的解 (x,y) 不对应某个 $(x_1+y_1\sqrt{D})^k$，那么必然存在 k 使得 $(x_1+y_1\sqrt{D})^{2k}<x+y\sqrt{D}<(x_1+y_1\sqrt{D})^{2k+2}$，消去因子 $(x_1+y_1\sqrt{D})^{2k+1}$，就说明存在位于 $-x_1+y_1\sqrt{D}$ 和 $x_1+y_1\sqrt{D}$ 之间的范数为 ±1 的二次整数 $x'+y'\sqrt{D}\ne1$．重复上一段利用倒数的论证可知，这与 $x_1+y_1\sqrt{D}$ 的最小性矛盾．因而原命题得证.

因为 $(A_{\ell-1},B_{\ell-1})$ 是负 Pell 方程的最小正整数解，而 $x^2-Dy^2=\pm1$ 的所有正整数解都出现在集合

$$\{(x,y):x+y\sqrt{D}=(A_{\ell-1}+B_{\ell-1}\sqrt{D})^k,k\in\mathbf{N}_+\}$$

中，又因为这些正整数解必然对应 $\sqrt{D}$ 的在循环节末尾（前一位）处的渐近分数，而且渐近分数的分子和分母是严格单调递增的，所以对所有 $k\in\mathbf{N}_+$ 总是有

$$(A_{\ell-1}+B_{\ell-1}\sqrt{D})^k=A_{k\ell-1}+B_{k\ell-1}\sqrt{D}.$$

在所有这些正整数解中，k 为奇数时就是负 Pell 方程的解，k 为偶数时就是 Pell 方程的解，两者交替出现.

判断负 Pell 方程是否有解，需要计算 $\sqrt{D}$ 连分数展开的循环节的长度，这并不容易计算，因此希望能够找到更简单的判断方法．但是，目前并没有条件简明、容易计算的判断方法[75]．此处仅仅提供一个简单的结论.

定理

方程 $x^2-Dy^2=-1$ 有解，则 4 不能整除 D 且 D 不含有 $4k+3$ 型的素因子．反过来，如果 $D=2$ 或 D 是 $4k+1$ 型的素数，那么方程必然有解.

证明

首先，负 Pell 方程有解，就意味着 -1 是 D 的二次剩余，因而 -1 也是 D 的任意一个因子 d 的二次剩余，故而 $d\ne4$ 且 d 不是 $4k+3$ 型素数．反过来，方程 $x^2-2y^2=-1$ 有非平凡解 $(1,1)$．剩下的就是 D 是 $4k+1$ 型素数的情形.

设 D 是 $4k+1$ 型素数，要证明方程 $x^2-Dy^2=-1$ 有解．思路是通过 Pell 方程 $x^2-Dy^2=1$ 的基本解 (u,v) 入手，构造出 $x^2-Dy^2=-1$ 的解 (α,β)．如果 u 是偶数，将 $u^2-Dv^2=1$ 两侧对 4 取模，就得到 $v^2\equiv-1\pmod4$，但是 -1 并不是模 4 的二次剩余．这个矛盾说明 u 是奇数．考察等式

[75] 一个较为实用的判断方法和工具在 这里[2] 及其参考文献．使得方程 $x^2-Dy^2=-1$ 有解的正整数 D 的列表是 OEIS A031396[3].

$Dv^2=u^2-1=(u+1)(u-1)$．因为 u 是奇数，所以 $\gcd(u+1,u-1)=\gcd(u+1,2)=2$．根据这一事实，将 Dv^2 的因子分给 $u+1$ 和 $u-1$，必然一个是 $2\alpha^2$，另一个是 $2D\beta^2$，其中，α 和 β 是互素的正整数而且 $v=2\alpha\beta$．将 $u=\alpha^2+D\beta^2$ 和 $v=2\alpha\beta$ 代入 $u^2-Dv^2=1$ 就得到 $\alpha^2-D\beta^2=\pm1$．因为 (u,v) 是 Pell 方程的基本解而且 (α,β) 是比 (u,v) 更小的正整数对，这个等式右侧不能是 $+1$，故而只能是 -1．这就证明 $x^2-Dy^2=-1$ 存在解 (α,β)．

如果 D 是合数，那么不含有 $4k+3$ 型素因子且没有平方因子也不能保证方程 $x^2-Dy^2=-1$ 有解，例如 $x^2-34y^2=-1$ 就没有解．

示例

利用上面的示例中的计算结果可知，方程 $x^2-14y^2=-1$ 无解，且方程 $x^2-41y^2=-1$ 的最小正整数解为 $(G_2,B_2)=(32,5)$．

范数为 ±4 的情形

接下来讨论方程 $x^2-Dy^2=\pm4$ 的解．此时，解的性态取决于 $D \bmod 4$ 的大小．

有些情形是容易的．如果 $D\equiv0 \pmod 4$，那么 x 是偶数，因而 $(x/2,y)$ 是方程 $u^2-(D/4)v^2=\pm1$ 的解．其余的情形，必然有 x,y 同时是奇数或者同时是偶数．如果 x,y 同时是奇数，方程两侧对 4 取模就得到 $D\equiv1 \pmod 4$．所以，如果 $D\equiv2,3 \pmod 4$，那么 x,y 只能同时是偶数，因而 $(x/2,y/2)$ 是方程 $u^2-Dv^2=\pm1$ 的解．因此，除了 $D\equiv1 \pmod 4$ 的情形，方程 $x^2-Dy^2=\pm4$ 的解都可以通过相应的（负）Pell 方程的解得到．

现在讨论 $D\equiv1 \pmod 4$ 的情形，它不能简单地转化为已经解决的情形．为了找到基本解，可以对 $(P_0,Q_0,D)=(1,2,D)$ 应用 PQa 算法，当首次得到 $Q_\ell=2$ 时，就到达了第一个循环节的末尾．如果循环节长度 ℓ 是偶数，那么 $(G_{\ell-1},B_{\ell-1})$ 就是方程 $x^2-Dy^2=4$ 的基本解；否则，$(G_{\ell-1},B_{\ell-1})$ 就是方程 $x^2-Dy^2=-4$ 的基本解．从 $(G_{\ell-1},B_{\ell-1})$ 出发，可以得到方程 $x^2-Dy^2=\pm4$ 的所有解：

$$\left\{(x,y):\frac{x+y\sqrt{D}}{2}=\pm\left(\frac{G_{\ell-1}+B_{\ell-1}\sqrt{D}}{2}\right)^k,k\in\mathbf{Z}\right\}.$$

如果循环节长度 ℓ 是偶数，所有这些都是方程 $x^2-Dy^2=4$ 的解；否则，当 k 是奇数时，(x,y) 是方程 $x^2-Dy^2=-4$ 的解，而当 k 是偶数时，(x,y) 是方程 $x^2-Dy^2=4$ 的解．

这个算法的正确性依赖于如下事实：

定理

设方程 $x^2-Dy^2=\pm4$ 有正整数解 (x,y)．如果 $D\equiv1 \pmod 4$，那么，$\dfrac{(x+y)/2}{y}$ 一定是 $\dfrac{1+\sqrt{D}}{2}$ 的渐近分数．

证明

首先注意到，此时 x,y 必然奇偶性相同，所以 $(x+y)/2$ 是整数．如果 (x,y) 是方程 $x^2-Dy^2=4$ 的解，那么 $x>y\sqrt{D}>2y$，所以，

$$\left|\frac{(x+y)/2}{y}-\frac{1+\sqrt{D}}{2}\right|=\frac{2}{y(x+y\sqrt{D})}<\frac{1}{2y^2}.$$

根据 Legendre 判别法→9.11.30@p1431 可知，$\dfrac{(x+y)/2}{y}$ 是 $\dfrac{1+\sqrt{D}}{2}$ 的渐近分数．

如果 (x,y) 是方程 $x^2-Dy^2=-4$ 的解，那么要建立上述不等式，只需要证明 $4y<x+y\sqrt{D}$. 这至少对于除了 $D=5,13$ 之外的情形都成立. 对于 $D=5,13$ 的情形，将 $x=\sqrt{Dy^2-4}$ 代入该不等式可知，它等价于 $2(\sqrt{D}-2)y^2>1$ 成立. 除了 $(D,y)=(5,1)$ 之外，该不等式对于所有 $D=5,13$ 和正整数 y 都成立. 剩下的就是验证 $(D,y)=(5,1)$ 的情形，此时，方程 $x^2-5y^2=-4$ 的解是 $(x,y)=(1,1)$，需要验证的是 $\frac{1}{1}$ 是 $\frac{1+\sqrt{5}}{2}=[\overline{1}]$ 的渐近分数，而这是显然成立的.

定理

设 D 是正整数但不是完全平方数. 二次无理数 $\omega=\dfrac{1+\sqrt{D}}{2}$ 的连分数展开具有形式

$$\omega=[\lfloor\omega\rfloor,\overline{a_1,\cdots,a_{\ell-1},2\lfloor\omega\rfloor-1}],$$

其中，ℓ 为循环节长度，且 $a_k=a_{\ell-k}$ 对任何 $1\le k<\ell$ 都成立.

证明

因为 $\lfloor\omega\rfloor-1+\omega>1$，而且它的共轭等于 $\lfloor\omega\rfloor-\omega$，位于 -1 和 0 之间，所以根据 Galois 的结论 →9.11.30@p1431 可知，$\lfloor\omega\rfloor-1+\omega$ 是纯循环连分数，可以写成

$$\lfloor\omega\rfloor-1+\omega=[\overline{2\lfloor\omega\rfloor-1,a_1,\cdots,a_{\ell-1}}].$$

而 Galois 关于倒数负共轭的结论说明

$$\frac{1}{\omega-\lfloor\omega\rfloor}=[\overline{a_{\ell-1},\cdots,a_1,2\lfloor\omega\rfloor-1}].$$

因此，根据连分数的定义，有

$$\lfloor\omega\rfloor-1+\omega=2\lfloor\omega\rfloor-1+\cfrac{1}{\overline{\omega-\lfloor\omega\rfloor}}=[2\lfloor\omega\rfloor-1,\overline{a_{\ell-1},\cdots,a_1,2\lfloor\omega\rfloor-1}].$$

连分数展开的唯一性就说明 $a_k=a_{\ell-k}$ 对所有 $1\le k<\ell$ 都成立，因而要证明的展开式也成立.

定理

设 $D\equiv1\pmod 4$. 在对 $(P_0,Q_0,D)=(1,2,D)$ 运行上述 PQa 算法的过程中，$Q_k=2$ 必然推出 $\ell\mid k$.

证明

在 $\dfrac{1+\sqrt{D}}{2}$ 的连分数展开中，除了第 0 个余项，所有其他余项都是 纯循环连分数→9.11.30@p1431. 设 $Q_k=2$. 根据 Galois 的结论，必然有余项 $\omega_k=\dfrac{P_k+\sqrt{D}}{2}$ 的共轭 $-1<\dfrac{P_k-\sqrt{D}}{2}<0$，亦即 $\sqrt{D}-2<P_k<\sqrt{D}$. 因为 PQa 算法中总有 $Q_k\mid P_k^2-D$（见 算法正确性证明→9.11.30@p1431），所以 P_k 一定是奇数，这就说明 P_k 取值唯一，即 $P_k=P_0+2(\lfloor\omega\rfloor-1)$，也就是说余项 $\omega_k=\omega_\ell$. 但是，余项的重复意味着连分数的循环，如果 k 不是 ℓ 的整数倍，就与 ℓ 是最小正周期相矛盾. 所以，必然有 $\ell\mid k$.

定理

设方程 $x^2-Dy^2=\pm4$ 的最小正整数解为 (x_1,y_1). 那么，它的全部解就是

$$\left\{(x,y):\frac{x+y\sqrt{D}}{2}=\pm\left(\frac{x_1+y_1\sqrt{D}}{2}\right)^k,k\in\mathbf{Z}\right\}.$$

证明

根据对称性，只需要考虑正整数解 (x,y) 即可．此处需要证明的只有集合中的实数对 (x,y) 确实是方程 $x^2-Dy^2=\pm4$ 的整数解．剩下的事情只需要复述对方程 $x^2-Dy^2=\pm1$ 的解的结构的证明即可．

实际上要证明的是，对于方程 $x^2-Dy^2=\pm4$ 的任何整数解 (x_1,y_1) 和 (x_2,y_2)，如下定义的正实数对 (x_3,y_3) 仍然是整数解：

$$\frac{x_3+y_3\sqrt{D}}{2}=\frac{x_1+y_1\sqrt{D}}{2}\frac{x_2+y_2\sqrt{D}}{2}.$$

展开右侧，比较有理项和无理项的系数可知

$$x_3=\frac{x_1x_2+Dy_1y_2}{2},\ y_3=\frac{x_1y_2+x_2y_1}{2}.$$

因为对 $i=1,2$ 有 $x_i\equiv x_i^2\equiv Dy_i^2\equiv Dy_i \pmod 2$，所以

$$2x_3=x_1x_2+Dy_1y_2\equiv D^2y_1y_2+Dy_1y_2=D(D+1)y_1y_2\equiv0 \pmod 2,$$

$$2y_3=x_1y_2+x_2y_1\equiv Dy_1y_2+Dy_2y_1=2Dy_1y_2\equiv0 \pmod 2.$$

这说明 x_3 和 y_3 都是整数．再利用范数保持乘法的性质可知，(x_3,y_3) 是 $x^2-Dy^2=\pm4$ 的解．

综合这些事实，重复前文几节的论述，就可以说明用于解决方程 $x^2-Dy^2=\pm4$ 的上述算法的正确性．这些结果说明，方程 $x^2-Dy^2=\pm4$ 具有和方程 $x^2-Dy^2=\pm1$ 类似的简单的解的结构：它的所有解都可以通过其最小正整数解表示出来，而无需求解其它方程．

其实，方程 $x^2-Dy^2=\pm1$ 的所有解都可以在方程 $x^2-Dy^2=\pm4$ 的解中找到，因而从这个角度看，方程 $x^2-Dy^2=\pm4$ 更为基础．显然，(x,y) 是方程 $x^2-Dy^2=\pm1$ 的解，当且仅当 $(2x,2y)$ 是方程 $x^2-Dy^2=\pm4$ 的解．前文的分析指出，当 $D\equiv2,3 \pmod 4$ 时，方程 $x^2-Dy^2=\pm4$ 的所有解都一定同为偶数，因而对应于 $x^2-Dy^2=\pm1$ 的解．

当 $D\equiv0 \pmod 4$ 时，方程 $x^2-Dy^2=\pm4$ 的解 (x,y) 中，x 一定是偶数，但是 y 可能是奇数．如果在方程 $x^2-Dy^2=\pm4$ 的最小正整数解 (x_1,y_1) 中，y_1 是偶数，那么在所有解中 y 也一定是偶数，此时这些整数解和方程 $x^2-Dy^2=\pm1$ 的整数解一一对应；但是，如果在最小整数解 (x_1,y_1) 中，y_1 是奇数，那么 y_k 的奇偶性将和 k 一致，交替变化，因而只有当 k 是偶数时，才对应于方程 $x^2-Dy^2=\pm1$ 的解．如果 $x^2-Dy^2=\pm4$ 的最小正整数解中 y_1 是奇数而且 $x_1+y_1\sqrt{D}$ 的范数是 -4，那么，对于这样的 D，$x^2-Dy^2=-4$ 有解，但是 $x^2-Dy^2=-1$ 无解．

当 $D\equiv1 \pmod 4$ 时，方程 $x^2-Dy^2=\pm4$ 的解 (x,y) 可能同时是奇数，也可能同时是偶数．如果最小正整数解 (x_1,y_1) 已经同时是偶数，那么它的所有整数解也一定同时是偶数，所以总是对应于方程 $x^2-Dy^2=\pm1$ 的整数解．如果最小正整数解 (x_1,y_1) 同时是奇数，那么有如下结论：

定理

设方程 $x^2-Dy^2=\pm4$ 的最小正整数解是 (x_1,y_1)．如果 x_1 和 y_1 同时是奇数，那么，$D\equiv5 \pmod 8$，且该方程的整数解 (x,y) 同时是偶数，当且仅当

$$\frac{x+y\sqrt{D}}{2}=\pm\left(\frac{x_1+y_1\sqrt{D}}{2}\right)^{3k},k\in\mathbf{Z}.$$

证明

等式 $x_1^2-Dy_1^2=\pm4$ 的两边同时对 8 取模，得到 $D\equiv5 \pmod 8$．要证明第二个结论，首先证明 (x_3,y_3) 都是偶数，因为

$$\frac{x_3+y_3\sqrt{D}}{2}=\left(\frac{x_1+y_1\sqrt{D}}{2}\right)^3=\frac{x_1^3+3Dx_1y_1^2}{8}+\frac{3x_1^2y_1+Dy_1^3}{8}\sqrt{D},$$

所以，只需要证明右侧是整数．因为奇数的平方模 8 余 1，所以，有

$$x_1^3+3Dx_1y_1^2=x_1(x_1^2+3Dy_1^2)\equiv x_1(1+3\times5\times1)=16x_1=0\pmod 8,$$

$$3x_1^2y_1+Dy_1^3=y_1(3x_1^2+Dy_1^2)\equiv y_1(3\times1+5\times1)=8y_1=0\pmod 8.$$

这就说明了 x_3,y_3 都是偶数．进而，对于所有 $k\in\mathbf{Z}$，都有

$$\frac{x+y\sqrt{D}}{2}=\pm\left(\frac{x_1+y_1\sqrt{D}}{2}\right)^{3k}=\pm\left(\frac{x_3+y_3\sqrt{D}}{2}\right)^{k}\in\mathbf{Z}[\sqrt{D}],$$

所以此时的 (x,y) 都是偶数．反过来，对于 $r=1,2$，总有

$$\pm\left(\frac{x_1+y_1\sqrt{D}}{2}\right)^{3k+r}=\pm\left(\frac{x_3+y_3\sqrt{D}}{2}\right)^{k}\left(\frac{x_r+y_r\sqrt{D}}{2}\right).$$

要证明相应的 (x,y) 不是整数，只需要证明该式不是整数，也就是右侧的乘积中第二项不是整数．这在 $r=1$ 时，就是已知条件；在 $r=2$ 时，因为

$$\frac{x_2+y_2\sqrt{D}}{2}=\left(\frac{x_1+y_1\sqrt{D}}{2}\right)^2=\frac{x_1^2+Dy_1^2}{4}+\frac{x_1y_1}{2}\sqrt{D},$$

而且 $x_1^2+Dy_1^2\equiv1+1\times1=2\pmod 4$，$x_1y_1\equiv1\pmod 2$，所以该式也不是整数．这就证明了，只有幂次是 3 的倍数的时候，相应的解才都是偶数．

也就是说，方程 $x^2-Dy^2=\pm4$ 的每三个解中就有一个同时是偶数，它对应着 $x^2-Dy^2=\pm1$ 的整数解．这也说明，对于 $D\equiv1\pmod 4$，方程 $x^2-Dy^2=-4$ 有解，当且仅当方程 $x^2-Dy^2=-1$ 有解．

到目前为止的讨论，已经足够计算实二次整数环的基本单位数．设 D 是正整数且不含平方因子．对于 $D\equiv2,3\pmod 4$ 的情形，只需要求出 $x^2-Dy^2=\pm1$ 的最小正整数解；而对于 $D\equiv1\pmod 4$ 的情形，只需要求出 $x^2-Dy^2=\pm4$ 的最小正整数解．当得到最小正整数解 (x,y) 时，对于 $D\equiv2,3\pmod 4$，基本单位数就是 $\pm x\pm y\sqrt{D}$；对于 $D\equiv1\pmod 4$，基本单位数就是 $\dfrac{\pm x\pm y\sqrt{D}}{2}$．

示例

1. 求解方程 $x^2-14y^2=\pm4$．

   利用前文的示例中的计算可知，方程 $x^2-14y^2=4$ 的最小正整数解为 $(30,8)$，方程 $x^2-14y^2=-4$ 无解．

2. 求解方程 $x^2-41y^2=\pm4$．

   对 $(P_0,Q_0,D)=(1,2,41)$ 运行 PQa 算法结果如下：（标红部分为第一个循环节）

k	P	Q	a	A	B	G	G^2-DB^2
0	1	2	3	3	1	5	-16
1	5	8	1	4	1	7	8
2	3	4	2	11	3	19	-8
3	5	4	2	26	7	45	16
4	3	8	1	37	10	64	-4
5	5	2	5	211	57	365	16
6	5	8	1	248	67	429	-8

7	3	4	2	707	191	1223	8
8	5	4	2	1662	449	2875	-16
9	3	8	1	2369	640	4098	4
10	5	2	5	13507	3649	23365	-16
11	5	8	1	15876	4289	27463	8

循环节长度 $\ell = 5$ 为奇数．方程 $x^2 - 41y^2 = -4$ 的最小正整数解为 $(G_4, B_4) = (64, 10)$，方程 $x^2 - 41y^2 = 4$ 的最小正整数解为 $(G_9, B_9) = (4098, 640)$．它们之间有如下关系：

$$\frac{4098 + 640\sqrt{41}}{2} = \left(\frac{64 + 10\sqrt{41}}{2}\right)^2.$$

当然，因为 $D \equiv 1 \pmod 8$，根据前文结论，此时方程 $x^2 - 41y^2 = \pm 4$ 的最小正整数解一定都是偶数，且总是方程 $x^2 - 41y^2 = \pm 1$ 的最小正整数解的 2 倍，所以也可以直接利用前文的示例得出．

3. 求解方程 $x^2 - 13y^2 = \pm 4$.

   对 $(P_0, Q_0, D) = (1, 2, 13)$ 运行 PQa 算法结果如下：（标红部分为第一个循环节）

k	P	Q	a	A	B	G	$G^2 - DB^2$
0	1	2	2	2	1	3	-4
1	3	2	3	7	3	11	4
2	3	2	3	23	10	36	-4
3	3	2	3	74	33	119	4

   循环节长度 $\ell = 1$ 为奇数．方程 $x^2 - 13y^2 = -4$ 的最小正整数解为 $(G_0, B_0) = (3, 1)$，方程 $x^2 - 13y^2 = 4$ 的最小正整数解为 $(G_1, B_1) = (11, 3)$.

   因为该方程的最小正整数解都是奇数，所以可以利用第三个循环节末尾处的数对 $(G_2, B_2) = (36, 10)$ 得到相应的（负）Pell 方程 $x^2 - 13y^2 = \pm 1$ 的最小正整数解 $(18, 5)$．它也可以通过直接计算得到：

   $$\frac{36 + 10\sqrt{13}}{2} = \left(\frac{3 + \sqrt{13}}{2}\right)^3.$$

   而且，这是负 Pell 方程的解．相应的 Pell 方程的最小正整数解是 $(649, 180)$.

4. 求解方程 $x^2 - 52y^2 = \pm 4$.

   因为方程 $x^2 - 13y^2 = \pm 1$ 的最小正整数解分别是 $(18, 5)$ 和 $(649, 180)$，所以方程 $x^2 - 52y^2 = \pm 4$ 的最小正整数解分别是 $(36, 10)$ 和 $(1298, 360)$.

一般情形

最后，讨论广义 Pell 方程的解法.

对于 $|\ N\ | < \sqrt{D}$ 的情形有一个简单的解法. 前文的结论说明方程 $x^2 - Dy^2 = N$ 的解 (x, y) 一定满足 $\dfrac{x}{y}$ 等于 $\sqrt{D}$ 的某个渐近分数．而且，根据前文讨论的解的结构可知，每个基本解 (x, y) 都满足 $x + y\sqrt{D}$ 小于等于相应的 Pell 方程 $x^2 - Dy^2 = 1$ 的基本解 $x_1 + y_1\sqrt{D}$．利用 PQa 算法中分母序列 B_k 的单调性可知，广义 Pell 方程的这些基本解一定出现在相应的 Pell 方程的基本解出现之前．由此，只需要对 $(P_0, Q_0, D) = (0, 1, D)$ 运行 PQa 算法，直到 $Q_{\ell'} = 1$ 且 ℓ' 为偶数时为止，过程中对出现的每个 (A_k, B_k) 检验是否存在整数 f 使得

$$A_k^2 - DB_k^2 = (-1)^{k+1} Q_{k+1} = N/f^2$$

成立，如果成立，则记录 (fA_k, fB_k) 是一个最小正整数解．这个过程记录的所有 (fA_k, fB_k) 就是方程 $x^2 - Dy^2 = N$ 的全部最小正整数解．利用 $(A_{\ell'-1}, B_{\ell'-1})$，即相应的 Pell 方程的基本解，就可以根据求出的这些最小正整数解，生成广义 Pell 方程的所有解．注意，取决于循环节长度 ℓ 是偶数还是奇数，上述的 ℓ' 可能是 ℓ 或是 2ℓ.

对于更为一般的 N 的情形，上述方法不再适用．首先，枚举 N 的所有平方因子 f^2，设 $m = N/f^2$，并枚举同余方程 $z^2 \equiv D \pmod{|m|}$ 的所有满足 $-|m|/2 < z \le |m|/2$ 的解 z．然后，对 $(P_0, Q_0, D) = (z, |m|, D)$ 运行 PQa 算法，直到 $Q_k = \pm 1$ 或已经结束了一个循环节．在第二种情形下，与该组 (f, z) 相关的方程的解并不存在．在第一种情形，需要进一步判断 $(-1)^k Q_k = N/|N|$ 与否．如果符号一致，那么 (fG_{k-1}, fB_{k-1}) 就是方程 $x^2 - Dy^2 = N$ 的解．否则，它是方程 $x^2 - Dy^2 = -N$ 的解，而且当且仅当相应的负 Pell 方程的解存在时，才可以通过复合它与相应的负 Pell 方程的基本解来得到方程 $x^2 - Dy^2 = N$ 的解．当完成对所有组 (f, z) 的遍历之后，就可以得到方程 $x^2 - Dy^2 = N$ 在每个解的等价类中各恰好一个解，且该解为该等价类中的基本解或最小正整数解．利用它们和相应的 Pell 方程的基本解，可以生成该方程的所有整数解．这一算法称为 **Lagrange–Matthews–Mollin 算法**.

该算法的正确性由如下定理保证：

定理

设方程 $x^2 - Dy^2 = N$ 有整数解 (x, y) 且 $x \ge 0, y > 0, \gcd(x, y) = 1$．令 $Q_0 = |N|$，则 $\gcd(Q_0, y) = 1$．设 P_0 是同余方程 $x \equiv -P_0 y \pmod{Q_0}$ 的解且 $-Q_0/2 < P_0 \le Q_0/2$，并设整数 X 使得 $x = Q_0 X - P_0 y$ 成立．那么，$P_0^2 \equiv D \pmod{Q_0}$，$\dfrac{X}{y}$ 是 $\omega = \dfrac{P_0 + \sqrt{D}}{Q_0}$ 的一个渐近分数 $\dfrac{A_{k-1}}{B_{k-1}}$，且 $Q_k = (-1)^k \dfrac{N}{|N|}$.

证明

利用 $x \equiv -P_0 y \pmod{Q_0}$ 和 $x^2 - Dy^2 = N \equiv 0 \pmod{Q_0}$，显然有 $P_0^2 \equiv D \pmod{Q_0}$．因而，

$$P_0 x + Dy \equiv -P_0^2 y + Dy = (D - P_0^2) y \equiv 0 \pmod{Q_0}.$$

由此，可以考察整系数矩阵

$$\begin{pmatrix} P & R \\ Q & S \end{pmatrix} = \begin{pmatrix} X & \dfrac{P_0 x + Dy}{Q_0} \\ y & x \end{pmatrix}.$$

它的行列式

$$PS - QR = \frac{x(x + P_0 y) - y(P_0 x + Dy)}{Q_0} = \frac{x^2 - Dy^2}{Q_0} = \pm 1.$$

而且，设 $\zeta = \sqrt{D} > 1$，就有

$$\frac{P\zeta + R}{Q\zeta + S} = \frac{(x + P_0 y)\sqrt{D} + (P_0 x + Dy)}{(x + y\sqrt{D})Q_0} = \frac{P_0 + \sqrt{D}}{Q_0} = \omega.$$

下面要证明 $\dfrac{P}{Q}$ 是 ω 的一个渐近分数．不妨设 $\dfrac{P}{Q}$ 有 连分数展开→9.11.30@p1431

$$\frac{P}{Q} = [a_0, a_1, \cdots, a_k]$$

且 $PS - QR = (-1)^{k-1}$．如果设 $\dfrac{p_k}{q_k}$ 是它的第 k 个渐近分数，那么 $(p_k, q_k) = (P, Q)$，且根据 渐近分数的差分公式→9.11.30@p1431 可知，$p_k q_{k-1} - q_k p_{k-1} = (-1)^{k-1}$．这说明

$$p_k(S - q_{k-1}) = q_k(R - p_{k-1}).$$

分情况讨论：

- 如果 $S=0$，那么容易验证 $Q=R=1$，因而 $\omega=P+\zeta^{-1}=[P,\zeta]$，故而 $\dfrac{P}{Q}=P$ 是 ω 的第 0 个渐近分数；
- 如果 $Q=S>0$，那么 $Q=S=1$ 且 $P-R=\pm1$．此时，
  - 如果 $P=R+1$，那么 $\omega=R+\dfrac{1}{1+\zeta^{-1}}=[R,1,\zeta]$，故而 $\dfrac{P}{Q}=\dfrac{R+1}{1}=[R,1]$ 是 ω 的第 1 个渐近分数；
  - 如果 $P=R-1$，那么 $\omega=R-1+\dfrac{1}{1+\zeta}=[R-1,\zeta-1]$，故而 $\dfrac{P}{Q}=R-1$ 是 ω 的第 0 个渐近分数；
- 如果 $Q\neq S>0$，那么由于 $Q=q_k\mid(S-q_{k-1})$，总存在整数 κ 使得 $S=\kappa q_k+q_{k-1}$ 和 $R=\kappa p_k+p_{k-1}$ 成立．因为 $q_k\geq q_{k-1}$ 且 $S>0$，所以 $\kappa\geq0$．因而，$\omega=\dfrac{(\kappa+\zeta)p_k+p_{k-1}}{(\kappa+\zeta)q_k+q_{k-1}}=[a_0,a_1,\cdots,a_k,\kappa+\zeta]$，故而 $\dfrac{P}{Q}$ 是它的第 k 个渐近分数．

综上所述，总有 $\dfrac{X}{y}$ 是 $\omega=\dfrac{P_0+\sqrt{D}}{Q_0}$ 的渐近分数，并按照 PQa 算法中的记号记作 $\dfrac{A_{k-1}}{B_{k-1}}$．因为 $A_{k-1}^2-DB_{k-1}^2=(-1)^kQ_0Q_k$，所以 $Q_k=(-1)^k\dfrac{N}{\mid N\mid}$．

该定理保证了方程的所有正解都存在于相应的二次无理数的渐近分数中．因为利用 PQa 算法计算渐近分数时，只要进入循环节，就一定能保证渐近分数总是正的．所以，只要枚举所有定理条件所允许的二次无理数，计算它的渐近分数直到一个循环节内，就能够找到一个解．因为在同一个二次无理数的渐近分数中出现的两个解，一定是等价的，所以只要得到第一个满足 $(-1)^kQ_k=N/\mid N\mid$ 的解，就可以停止继续后面的计算．与前面的所有算法都不同的是，此处满足条件的 k 可能出现在尚未进入循环节时．

示例

1. 求解方程 $x^2-157y^2=12$.

   因为 $12^2<157$，所以对 $(P_0,Q_0,D)=(0,1,157)$ 运行 PQa 算法结果如下：（标红部分为第一个循环节）

k	P	Q	a	A	B	G	G^2-DB^2
0	0	1	12	12	1	12	-13
1	12	13	1	13	1	13	12
2	1	12	1	25	2	25	-3
3	11	3	7	188	15	188	19
4	10	19	1	213	17	213	-4
5	9	4	5	1253	100	1253	9
6	11	9	2	2719	217	2719	-12
7	7	12	1	3972	317	3972	11
8	5	11	1	6691	534	6691	-11
9	6	11	1	10663	851	10663	12
10	5	12	1	17354	1385	17354	-9
11	7	9	2	45371	3621	45371	4
12	11	4	5	244209	19490	244209	-19
13	9	19	1	289580	23111	289580	3
14	10	3	7	2271269	181267	2271269	-12
15	11	12	1	2560849	204378	2560849	13

16	1	13	1	4832118	385645	4832118	−1
17	12	1	24	118531681	9459858	118531681	13
18	12	13	1	123363799	9845503	123363799	−12
19	1	12	1	241895480	19305361	241895480	3
20	11	3	7	1816632159	144983030	1816632159	−19
21	10	19	1	2058527639	164288391	2058527639	4
22	9	4	5	12109270354	966424985	12109270354	−9
23	11	9	2	26277068347	2097138361	26277068347	12
24	7	12	1	38386338701	3063563346	38386338701	−11
25	5	11	1	64663407048	5160701707	64663407048	11
26	6	11	1	1030497457498	82242650531	103049745749	−12
27	5	12	1	1677131527971	133849667601	167713152797	9
28	7	9	2	4384760513433	349941985734	438476051343	−4
29	11	4	5	2360093409512	188355959622	2360093409512	19
30	9	19	1	2798569460855	223350158192	2798569460855	−3
31	10	3	7	21950079635497	1751807067011	21950079635497	12
32	11	12	1	247486490963520	19751572252047	247486490963520	−13
33	1	13	1	46698728731849	3726964292220	46698728731849	1
34	12	1	24	1145518138660728	91422300238148	1145518138660728	−13
35	12	13	1	11922168673925770	951492645301092	11922168673925770	12

循环节长度 $\ell = 17$ 为奇数，所以需要考察两个循环节内 $G_k^2 - 157B_k^2$ 与 12 相差一个平方因子的情形，即 $k = 1, 9, 13, 19, 23, 31$ 的情形．它们对应的解就是下表中的 (fG, fB)：

k	f	fG_k	fB_k	x	y
1	1	13	1	13	1
9	1	10663	851	10663	851
13	2	579160	46222	579160	46222
19	2	483790960	38610722	−579160	46222
23	1	26277068347	2097138361	−10663	851
31	1	21950079635497	1751807067011	−13	1

全体 (fG, fB) 就是方程 $x^2 - 157y^2 = 12$ 的解集的所有等价类中的最小正整数解．要通过这些解得到全部解，可以利用相应的 Pell 方程的基本解 $(46698728731849, 3726964292220)$．比如说，可以将它们转化为该等价类中的基本解 (x, y)，相应的解也一并列于上表中．

2. 求解方程 $x^2 - 157y^2 = 12$.

这次利用 Lagrange–Matthews–Mollin 算法求解．首先，枚举 $N = 12$ 的平方因子：

- $f^2 = 1^2$ 时，有 $m = 12$，同余方程 $z^2 \equiv 157 \pmod{12}$ 有解 $z = \pm 1, \pm 5$；
- $f^2 = 2^2$ 时，有 $m = 3$，同余方程 $z^2 \equiv 157 \pmod{3}$ 有解 $z = \pm 1$.

对于所有可能的 (f, z) 组合运行初始参数为 $(P_0, Q_0, D) = (z, |\ m\ |, D)$ 的 PQa 算法，并找到首个 $(-1)^k Q_k = 1$ 的位置，对应的 (fG_{k-1}, fB_{k-1}) 就是一组解．结果如下表所示：

f	z	m	k	fG_{k-1}	fB_{k-1}
1	1	12	32	21950079635497	1751807067011
1	−1	12	2	13	1

1	5	12	24	26277068347	2097138361
1	−5	12	10	10663	851
2	1	3	20	483790960	38610722
2	−1	3	14	579160	46222

这就是前文列举的所有等价类中的最小正整数解，可以利用 Pell 方程的基本解将它们转化为基本解.

3. 求解方程 $x^2 - 79y^2 = \pm 101$.

   仍然使用 Lagrange–Matthews–Mollin 算法求解. 因为 $N = 101$ 是素数，所以一定有 $f = 1$. 此时，$m = 101$，相应的同余方程 $z^2 \equiv 79 \pmod{101}$ 有解 $z = \pm 33$.

   对 $(P_0, Q_0, D) = (33, 101, 79)$ 运行 PQa 算法结果如下：（标红部分为第一个循环节）

k	P	Q	a	A	B	G	$G^2 - DB^2$
0	33	101	0	0	1	−33	1010
1	−33	−10	2	1	2	35	909
2	13	9	2	2	5	37	−606
3	5	6	2	5	12	109	505
4	7	5	3	17	41	364	−303
5	8	3	5	90	217	1929	1010
6	7	10	1	107	258	2293	−707
7	3	7	1	197	475	4222	909
8	4	9	1	304	733	6515	−606
9	5	6	2	805	1941	17252	505

   循环节长度 $\ell = 6$ 为偶数. 直到一个循环节结束，都不存在 $Q_k = \pm 1$，因而，该情形无解. 相应地，对于 $(P_0, Q_0, D) = (-33, 101, 79)$ 运行 PQa 算法也可以观察到类似的情况. 因此，该方程无解.

习题

- LOJ 6687.「Project Euler 66」解方程[4]
- SPOJ EQU2 - Yet Another Equation[5]
- SPOJ PELL2 - Pell (Mid pelling)[6]
- UVa 12909. Numeric Center[7]
- UVa 10241. Semi-triangular and also Square[8]

参考文献与注释

- Pell's equation - Wikipedia[9]
- John P. Robertson - Solving the generalized Pell equation $x^2 - Dy^2 = N$[10]
- Keith Matthews - The Diophantine Equation $x^2 - Dy^2 = N$,$D > 0$[11]
- Existence of Solution to Pell's Equation - Suryateja Gavva's Blog[12]
- Calculating the simple continued fraction of a quadratic irrational - Number Theory Web[13]（PQa 算法）
- Solving the diophantine equation x2–Dy2 = N, D > 0 and not a perfect square, N ≠ 0 - Number Theory Web[14]（Lagrange–Matthews–Mollin 算法）

[1] Brahmagupta 恒等式 [2] 这里 [3] OEIS A031396

[4] LOJ 6687.「Project Euler 66」解方程 
[5] SPOJ EQU2 - Yet Another Equation 
[6] SPOJ PELL2 - Pell (Mid pelling) 
[7] UVa 12909. Numeric Center 
[8] UVa 10241. Semi-triangular and also Square 
[9] Pell's equation - Wikipedia 
[10] John P. Robertson - Solving the generalized Pell equation $x^2 - Dy^2 = N$ 
[11] Keith Matthews - The Diophantine Equation $x^2 - Dy^2 = N, D > 0$ 
[12] Existence of Solution to Pell's Equation - Suryateja Gavva's Blog 
[13] Calculating the simple continued fraction of a quadratic irrational - Number Theory Web 
[14] Solving the diophantine equation x2–Dy2 = N, D > 0 and not a perfect square, N ≠ 0 - Number Theory Web 

9.12 多项式与生成函数

9.12.1 多项式与生成函数简介

多项式与生成函数

操纵 有限项/无限项 的多项式是 OI 数学中，尤其是生成函数中的重要内容.

以 快速傅里叶变换→9.12.3@p1527 为基石的多项式算法赋予了算法竞赛选手直接操纵生成函数的能力.

基本概念

对于求和式 $\sum a_n x^n$，如果是有限项相加，称为多项式，记作 $f(x) = \sum_{n=0}^{m} a_n x^n$.

可列项相加的求和式称为级数. 在和式 $\sum_{n=0}^{\infty} a_n x^n$ 中，每项均为非负整数次幂函数乘常数系数，这种形式的级数称为幂级数.

研究多项式算术时先考虑较简单的多项式，幂级数概念仅用于方便理解. 到了数学分析中会进一步研究幂级数的敛散性.

环、域及其衍生结构的一般定义详见 抽象代数基本概念→9.16.1@p1805.

对于一般环 R，定义 R 上的 **多项式环**（polynomial ring）$R[x]$.

每个元素 f 称为 R 上的 **多项式**（polynomial），可表示为

$$f =< f_0, f_1, f_2, \cdots, f_n > \quad (f_0, f_1, f_2, \cdots, f_n \in R)$$

换言之，我们将多项式直接定义为系数序列. 也可以表示为

$$f(x) = f_0 + f_1 x + f_2 x^2 + \cdots + f_n x^n$$

此处我们认为 x 只是一个 **形式符号**，一个对系数位置的标识符.

如果我们还允许无穷项的存在，即

$$f(x) = f_0 + f_1 x + f_2 x^2 + \cdots$$

则可得到 **形式幂级数环**（formal power series ring）$R[[x]]$，其中的每个元素 f 称为 **形式幂级数**（formal power series），以下简称幂级数.

多项式的次数

对于一个多项式 $f(x)$，称其最高次项的次数为该多项式的 **次数（degree）**，记作 $\deg f$.

多项式的乘法

最核心的操作是两个多项式的乘法，即给定多项式 $f(x)$ 和 $g(x)$:

$$f(x) = a_0 + a_1 x + ... + a_n x^n \qquad (1)$$

$$g(x) = b_0 + b_1 x + ... + b_m x^m \qquad (2)$$

要计算多项式 $Q(x) = f(x) \cdot g(x)$:

$$\boxed{Q(x)=\sum_{i=0}^{n}\sum_{j=0}^{m}a_ib_jx^{i+j}}=c_0+c_1x+\ldots+c_{n+m}x^{n+m}$$

多项式或幂级数的乘法，满足结合律，关于加法满足分配律．若 R 为交换环或幺环，乘法相应的有交换律和单位元．

若 R 上存在 2^n 次单位根，快速傅里叶变换→9.12.3@p1527 允许我们在 $O(n2^n)$ 而不是 $O(2^{2n})$ 的时间内计算两个 2^n 次多项式的乘积．

复合

定义 $R[[x]]$ 中元素 f 的乘方为

$$f^1=f,f^k=f^{k-1}\times f$$

在此基础上，定义 $R[[x]]$ 中元素 f,g 的复合为

$$(f\circ g)(x)=f(g(x))=f_0+\sum_{k=1}^{+\infty}f_kg^k(x)$$

我们规定 $f\circ g$ 存在当且仅当 f 为有限项或 $g_0=0$，这样就不涉及 R 上的极限了．

$\circ$ 满足结合律（$(f\circ g)\circ h$ 和 $f\circ(g\circ h)$ 均存在时），不满足交换律．R 为幺环时 $\circ$ 存在单位元 $1\times x$．

多项式复合（与复合逆）有 $\Theta(n\log^2 n)$ 的做法，其由 Yasunori Kinoshita 与李白天于 2024 年提出，详见 形式幂级数复合 | 复合逆→9.12.14@p1589．

导数

尽管一般环甚至未必存在极限，

我们依然可以定义形式幂级数的 **形式导数**（formal derivative）为

$$\left(\sum_{k=0}^{+\infty}f_kx^k\right)'=\sum_{k=1}^{+\infty}kf_kx^{k-1}$$

其中

$$kf_k=\underbrace{f_k+f_k+\cdots+f_k}_{k\text{个}f_k}$$

基本求导法则——加法法则、乘法法则、链式法则（复合允许的情况下）依然是正确的．

如果 R 上允许作除法，同样可以类似定义形式幂级数的 **形式不定积分**（formal indefinite integral）．

乘法逆元

根据例子

$$\frac{1}{1-x}=1+x+x^2+\cdots$$

可以知道，多项式的倒数是能展开为无穷级数的．存在倒数，当且仅当常数项不为 0，并且倒数也满足常数项不为 0．

因此定义：对于形式幂级数 f，若 $f_0\neq 0$，其 **乘法逆元**（multiplicative inversion）f^{-1} 为另一形式幂级数，满足

$$f\times f^{-1}=f^{-1}\times f=1$$

用形式幂级数乘法定义展开该式，可得 f^{-1} 系数的递推式

$$f_0^{-1}=\frac{1}{f_0},f_n^{-1}=\frac{-1}{f_0}\sum_{k=0}^{n-1}f_k^{-1}f_{n-k}$$

直接用递推式计算前 n 项是 $O(n^2)$ 的，运用 FFT→9.12.9@p1559 可得到 $O(n\log n)$ 的算法．

> **注**
>
> 容易发现，$f(x)$ 的倒数就是 $\frac{1}{f(x)}$ 的无穷项麦克劳林展开（在 $x=0$ 处的无穷项泰勒展开）.

常见的幂级数展开式

在数学分析中，在某点处或某区间上若干阶可导的一元函数可以在相应范围进行多项式展开，一般统称为泰勒展开，如果在 0 处展开，也称为麦克劳林展开.

如果无穷阶可导，则可以进行幂级数展开．最常见的还是在 0 处展开.

在复变函数中，一些函数在奇点上虽然不能进行泰勒展开，但是可以进行洛朗展开.

下列等式只在幂级数收敛时成立，在不收敛时不成立．这里只写出展开式，不讨论收敛域.

基本的展开式有以下两个，指数函数和幂函数：

$$\mathrm{e}^x=1+x+\frac{1}{2!}x^2+...+\frac{1}{n!}x^n+...$$

$$(1+x)^a=1+ax+\frac{a(a-1)}{2!}x^2+...+\frac{a(a-1)...(a-n+1)}{n!}x^n+...$$

更多的展开式经常由上述两个变形得到．比如正余弦函数由指数函数代入复数得到：

$$\cos x=1-\frac{1}{2!}x^2+\frac{1}{4!}x^4+...+\frac{(-1)^n}{(2n)!}x^{2n}+...$$

$$\sin x=x-\frac{1}{3!}x^3+\frac{1}{5!}x^5+...+\frac{(-1)^n}{(2n+1)!}x^{2n+1}+...$$

对数和反正切、反正弦函数由积分得到：

$$\frac{1}{1+x}=1-x+x^2+...+(-1)^nx^n+...$$

$$\ln(1+x)=x-\frac{1}{2}x^2+\frac{1}{3}x^3+...+\frac{(-1)^{n-1}}{n}x^n+...$$

$$\frac{1}{1+x^2}=1-x^2+x^4+...+(-1)^nx^{2n}+...$$

$$\arctan x=x-\frac{1}{3}x^3+\frac{1}{5}x^5+...+\frac{(-1)^n}{2n+1}x^{2n+1}+...$$

$$\frac{1}{\sqrt{1+x}}=1-\frac{1}{2}x+\frac{3}{8}x^2+...+(-1)^n\frac{(2n)!}{(n!)^2 4^n}x^n+...$$

$$\frac{1}{\sqrt{1-x^2}}=1+\frac{1}{2}x^2+\frac{3}{8}x^4+...+\frac{(2n)!}{(n!)^2 4^n}x^{2n}+...$$

$$\arcsin x=x+\frac{1}{6}x^3+\frac{3}{40}x^5+...+\frac{(2n)!}{(n!)^2(2n+1)4^n}x^{2n+1}+...$$

复合逆

复合逆（compound inversion）即反函数概念在形式幂级数环上的推广.

对于满足 $f_0=0$ 且 $f_1\neq 0$ 的形式幂级数 f，其复合逆为满足 $g(f(x))=f(g(x))=x$ 的形式幂级数 g. 由拉格朗日反演可得对于任意整数 n,k 有

$$n[x^n]f^k=k[x^{n-k}]\left(\frac{x}{g}\right)^n$$

其中 $[x^k]f(x)$ 表示 $f(x)$ 在 x^k 处的系数.

多项式整除

对于多项式 $f(x)$ 和多项式 $g(x)$，如果存在一个多项式 $h(x)$，使得：

$$f(x) = g(x)h(x)$$

则多项式 $g(x)$ 整除多项式 $f(x)$.

显而易见，多项式 $g(x)$ 整除多项式 $f(x)$，当且仅当 $g(x)$ 的根全部为 $f(x)$ 的根，并且在 $g(x)$ 中的重数不超过在 $f(x)$ 中的相应重数.

多项式的余数和商

对于多项式 $f(x), g(x)$，存在 **唯一** 的 $Q(x), R(x)$ 满足：

$$f(x) = Q(x)g(x) + R(x)$$

$$\deg R < \deg g$$

当 $\deg f \geq \deg g$ 时有 $\deg Q = \deg f - \deg g$，否则有 $Q(x) = 0$. 我们称 $Q(x)$ 为 $g(x)$ 除 $f(x)$ 的 **商（quotient）**，$R(x)$ 为 $g(x)$ 除 $f(x)$ 的 **余数（remainder）**.

模多项式

模多项式是多项式环的子环，由多项式环除以同余的等价关系得到.

在上文提到的带余除法中，多项式 $f(x)$ 与它的余式 $R(x)$ 在模多项式 $g(x)$ 的意义下同余.

$$f(x) \equiv R(x) \pmod{g(x)}$$

这个同余式也意味着，对于多项式 $g(x)$ 的任意一个根 x_0，代入 $f(x)$ 和 $R(x)$ 中，得到的点值相同. 即：

$$f(x_0) = R(x_0)$$

并且，如果根 x_0 在多项式 $g(x)$ 中的重数是 k，即 $(x - x_0)^k$ 整除 $g(x)$，则对任意大于等于 0 小于 k 的整数 t，有：

$$f^t(x_0) = R^t(x_0)$$

这里的记号表示 t 阶导数.

模多项式同余可以应用于幂级数. 一个无限项的幂级数，可以在模具体的多项式情形下，和一个有限项的多项式同余. 例如：

$$1 + x + x^2 + x^3 + ... \equiv 1 + x + ... + x^{n-1} \pmod{x^n}$$

显然剩余的所有项都被 x^n 整除，因此模 x^n 的操作等价于「截断」，将无穷项的幂级数截断到前 n 项，直接将更高位的信息丢失.

在一些特定的情况下，也可以模其他的多项式，下文将解释相应情况.

多项式的多点求值和插值

多项式的多点求值（multi-point evaluation） 即给出一个多项式 $f(x)$ 和 n 个点 $x_1, x_2, ..., x_n$，求

$$f(x_1), f(x_2), ..., f(x_n)$$

多项式的插值（interpolation） 即给出 $n + 1$ 个点

$$(x_0, y_0), (x_1, y_1), ..., (x_n, y_n)$$

求一个 n 次多项式 $f(x)$ 使得这 $n + 1$ 个点都在 $f(x)$ 上.

这两种操作的实质就是将多项式在 **系数表示** 和 **点值表示** 间转化. 多点求值将多项式的系数表示转为点值表示，插值将多项式的点值表示转为系数表示.

> **注**
>
> 按照幂级数的观点看，多点求值相当于将无穷项的信息「压缩」到有限个点值表示，因此丢失了一些信息，而插值相当于还原到相应次数的系数表示.

编程常见的求值与插值，例如离散傅里叶变换（及其逆变换）等等，选择的 $n+1$ 个点重数均为 1，即两两不同，免去求导的麻烦.

这种「压缩」只保证了在 $n+1$ 个点上的一致. 根据上文对模多项式同余的解释, 如果幂级数 $f(x)$ 经过在 x_0 到 x_n 点处求值再插值，得到多项式 $R(x)$，则作多项式：

$$g(x)=(x-x_0)...(x-x_n)$$

就有：

$$f(x)\equiv R(x)\pmod{g(x)}$$

由于 $R(x)$ 的次数严格小于 $g(x)$，所以利用求值与插值求出的 $R(x)$ 就是余式. 因此在这种情况下，如果幂级数在根处可以求值，就可以模多项式. 一个反例例如：

$$\frac{1}{1-x}=1+x+x^2+x^3+...$$

在 $x=1$ 处不可求值，因此级数 $1+x+x^2+x^3+...$ 不能模多项式 $x-1$.

由于幂级数的任意阶导数，在 0 处的值总是存在，因此模 x^n 始终可以计算，与上文「截断」的意义一致. 离散傅里叶变换（及其逆变换）相当于模多项式 x^n-1.

因式分解和欧几里得

初等数论中的许多结论可以推广到多项式上.

复数域上，由代数基本定理可得，对于 n 次多项式 f，方程

$$f(x)=0$$

有且仅有 n 个解（重根按重数计）.

于是 $f(x)$ 在复数域内可唯一因式分解为如下形式

$$a(x-x_1)^{c_1}(x-x_2)^{c_2}\cdots(x-x_m)^{c_m}$$

$$c_1+c_2+\cdots+c_m=n,x_1,x_2,\cdots,x_m \text{ 互不相同}$$

此时类比正整数的最大公因数，可得多项式的 **最大公因式**→9.11.4@p1188 (greatest common divisor, gcd). 其可用欧几里得算法求解

$$\gcd(f,0)=f,\gcd(f,g)=\gcd(g,f \bmod g)$$

该性质可以推广到较为一般的情况：

> 对于任意域 P 上的多项式环 $P[x]$,
>
> 多项式均可唯一因式分解, 且可用欧几里得算法计算最大公因式. 需要注意的是, 对于一般环上的多项式，该结论未必成立.

欧几里得算法成立时，可用扩展欧几里得给出不定方程

$$f(x)P(x)+g(x)Q(x)=\gcd(f(x),g(x))$$

的一组特解 $(P(x),Q(x))$，并用 裴蜀定理→9.11.8@p1218 判断不定方程

$$f(x)P(x)+g(x)Q(x)=h(x)$$

的可解性.

HALF-GCD[1] 允许我们在 $O(n\log^2 n)$ 时间内计算多项式欧几里得.

模多项式的乘法逆元

在模多项式 $h(x)$ 意义下，幂级数 $f(x)$ 有时存在逆元. 逆元就是幂级数 $f(x)$ 的倒数模多项式 $h(x)$ 得到的余式.

这个定义也等价于，对于多项式 $f(x)$，若存在 $g(x)$ 满足：

$$f(x)g(x) \equiv 1 \pmod{h(x)}$$

则称 $g(x)$ 为 $f(x)$ 在模 $h(x)$ 意义下的 **逆元（inverse element）**．当多项式欧几里得允许时，逆元存在当且仅当 $\gcd(f,h)=1$.

模多项式 $h(x)$ 意义下逆元总是唯一的．如果多项式 $f(x)$ 的次数也小于 $h(x)$，则得到的 $g(x)$ 与 $f(x)$ 互为逆元.

考虑「截断」的概念，一般把模 x^n 意义下的逆元记作 $f^{-1}(x)$，也是后文默认使用的逆元概念．如果不加说明，逆元的模就是指 x^n.

注

一个问题是，可不可以用各种插值变换，直接求解「逆元」，比如计算：

$$IDFT\left(\frac{DFT(1)}{DFT(f(x))}\right)$$

答案是否定的，不可以这样做．根据上文的解释，这里利用离散傅里叶变换（及其逆变换）直接求出的逆元是模多项式 x^n-1 的逆元，不是通常的模多项式 x^n 的逆元．并且，由于原多项式在各点值处可能为 0，这个求解未必可以进行.

生成函数

生成函数（generating function），又称母函数，是一种形式幂级数，其每一项的系数可以提供关于这个序列的信息.

生成函数有许多不同的种类，但大多可以表示为单一的形式：

$$F(x)=\sum_n a_n k_n(x)$$

其中 $k_n(x)$ 被称为核函数．不同的核函数会导出不同的生成函数，拥有不同的性质．举个例子：

1. 普通生成函数：$k_n(x)=x^n$.
2. 指数生成函数：$k_n(x)=\dfrac{x^n}{n!}$.
3. 狄利克雷生成函数→9.11.22@p1322：$k_n(x)=\dfrac{1}{n^x}$.

参考资料与拓展阅读

- Picks's Blog[2]
- Miskcoo's Space[3]
- Polynomial ring - Wikipedia[4]
- Formal power series - Wikipedia[5]
- 《信息学竞赛中的生成函数计算理论框架》

外部参考资料

[1] HALF-GCD 
[2] Picks's Blog 
[3] Miskcoo's Space 
[4] Polynomial ring - Wikipedia 
[5] Formal power series - Wikipedia 

9.12.2 代数基本定理

定义

任何复系数一元 n 次多项式（n 至少为 1）方程在复数域上至少有一根.

由此推出，n 次复系数多项式方程在复数域内有且只有 n 个根，重根按重数计算.

有时这个定理也表述为：

任何一个非零的一元 n 次复系数多项式，都正好有 n 个复数根.

代数基本定理的证明，一般会用到复变函数或者近世代数，因此往往作为一个熟知结论直接应用.

根据代数基本定理，一个复系数多项式 $f(x)=a_nx^n+a_{n-1}x^{n-1}+...+a_0$ 一定可以唯一地分解为：

$$f(x)=a_n(x-x_1)^{k_1}(x-x_2)^{k_2}...(x-x_t)^{k_t}$$

其中各个根均为复数，$k_1+k_2+...+k_t=n$.

虚根成对定理

代数基本定理的研究对象是复系数多项式. 当对实系数多项式进行研究时，虽然也能分解出复数根，却需要将研究范围扩大，不太方便.

虚根：非实数根.

定理：实系数多项式的根的共轭复数也是该多项式的根.

证明：直接在代数基本定理的等式两端取共轭即证毕.

如果根本身是实数，则取共轭仍为它本身，不受影响.

如果根是虚根，则虚根的共轭复数也是原多项式的根. 那么，两个虚根就可以配对.

定理：实数系数方程的共轭虚根一定成对出现，并且共轭虚根的重数相等.

证明：假设一个根为 $a+b\mathrm{i}$，则另一个根为 $a-b\mathrm{i}$. 这意味着在分解式中存在两项：

$$(x-a-b\mathrm{i})(x-a+b\mathrm{i})=x^2-2ax+a^2+b^2$$

可以看到两项乘在一起, 各项系数会全部变为实数. 这个等式右端的二次实系数多项式整除原始的多项式.

于是, 在代数基本定理的等式中, 两遍同时除以这个二次三项式, 得到的仍旧是实系数多项式的等式. 对新等式重复操作，随着次数的下降，若干次后即不存在虚根.

因此，每对共轭虚根的重数相等. 证毕.

以下是虚根成对定理的推论：

- 实系数奇次多项式至少有一个实根，并且总共有奇数个实根.
- 实系数偶次多项式可能没有实根，总共有偶数个实根.

称上述二次三项式 $x^2-2ax+a^2+b^2=x^2+px+q$ 为二次实系数不可约因式. 不可约是指它在实数范围内不可约.

定理：实系数多项式一定是一次或者二次实系数不可约因式的积.

证明：

只要实系数多项式有一个实根 c，就有一个实系数因式 $x-c$ 和它对应；有一对虚根 $a\pm b\mathrm{i}$，就有一个实系数因式 $x^2-2ax+a^2+b^2$ 和它对应.

因此，只要在原始的代数基本定理分解式中，利用虚根成对定理进行配对，即证毕.

根据虚根成对定理，一个实系数多项式 $f(x)=a_nx^n+a_{n-1}x^{n-1}+...+a_0$ 一定可以唯一地分解为：

$f(x)=a_n(x-x_1)^{k_1}(x-x_2)^{k_2}...(x-x_t)^{k_t}(x^2+p_1x+q_1)^{l_1}(x^2+p_2x+q_2)^{l_2}...(x^2+p_sx+q_s)^{l_s}$

其中各项系数均为实数，$k_1+k_2+...+k_t+2(l_1+l_2+...+l_s)=n$.

林士谔算法

简介

怎样对实系数多项式进行代数基本定理的分解？如果将数域扩充至复数会很复杂.

如果只在实数范围内进行分解，只能保证，当次数大于 2 的时候，一定存在实系数二次三项式因式.

这是因为，如果该多项式有虚根，直接凑出一对共轭虚根即可. 如果该多项式只有实根，任取两个实根对应的一次因式乘在一起，也能得到实系数二次三项式因式.

找到二次三项式因式之后, 再从二次式中解实根或复根就极为容易. 于是便有逐次**找出一个二次因子**来求得方程的复根的计算方法，这种方法避免了复数运算.

在 1940 年 8 月、1943 年 8 月和 1947 年 7 月，林士谔先后在 MIT 出版的《数学物理》杂志上接连正式发表了 3 篇关于解算高阶方程式复根方法的论文[1]，每次均有改进.

这个方法今天还在现代计算机中进行快速运算，计算机程序包（如 MATLAB）中的多项式求根程序依据的原理也是这个算法.

过程

要想找到一个二次三项式因子，就要将多项式分解为:

$$f(x)=(x^2+p_1x+q_1)g(x)$$

由于无法一下子找到二次三项式因子，按照迭代求解的思路，对于初始值有:

$$f(x)=(x^2+px+q)g(x)+rx+s$$

会产生一个一次式作为余项. 只要余项足够小，即可近似地找到待求因子.

我们希望最终解是初始值加一个偏移修正:

$$p_1=p+dp$$

$$q_1=q+dq$$

余式中的两个数 (r,s) 由除式的给定系数 (p,q) 决定. 有偏导数关系:

$$dr=\frac{\partial r}{\partial p}dp+\frac{\partial r}{\partial q}dq$$

$$ds=\frac{\partial s}{\partial p}dp+\frac{\partial s}{\partial q}dq$$

在初始的等式中，被除式 $f(x)$ 是给定的，商式 $g(x)$ 和余式 $rx+s$ 随着除式 x^2+px+q 的变化而变化. 因此有偏导数关系

$$0=xg(x)+\frac{\partial g(x)}{\partial p}(x^2+px+q)+\frac{\partial r}{\partial p}x+\frac{\partial s}{\partial p}$$

$$0=g(x)+\frac{\partial g(x)}{\partial q}(x^2+px+q)+\frac{\partial r}{\partial q}x+\frac{\partial s}{\partial q}$$

注意到，偏导数只是一个数值，与变元 x 无关. 因此有整除关系

$$xg(x)=-\frac{\partial g(x)}{\partial p}(x^2+px+q)-\frac{\partial r}{\partial p}x-\frac{\partial s}{\partial p}$$

$$g(x)=-\frac{\partial g(x)}{\partial q}(x^2+px+q)-\frac{\partial r}{\partial q}x-\frac{\partial s}{\partial q}$$

这里的结论是，待求的偏导数，恰好是对商式继续做除法的余式. 多项式对给定二次三项式的除法，直接计算即可. 这里就求得了四个偏导数.

我们希望 s 和 r 加上偏移 ds 与 dr 得到 0，即 ds 与 dr 是 s 和 r 的相反数. 因此要解方程:

$$-\frac{\partial r}{\partial p}dp-\frac{\partial r}{\partial q}dq=r$$

$$-\frac{\partial s}{\partial p}dp-\frac{\partial s}{\partial q}dq=s$$

[1]林士谔. 论劈因法解高阶特征方程根值的应用问题. 数学进展，1963(03):207-217.[1]

从上述方程组中解得 p 和 q 相应的偏移 dp 和 dq，直接用二阶行列式求解即可.

实现

```
// a 是原始的多项式，n 是多项式次数，p 是待求的一次项，q 是待求的常数项
void Shie(double a[], int n, double *p, double *q) {
  // 数组 b 是多项式 a 除以当前迭代二次三项式的商
  memset(b, 0, sizeof(b));
  // 数组 c 是多项式 b 乘以 x 平方再除以当前迭代二次三项式的商
  memset(c, 0, sizeof(c));
  *p = 0;
  *q = 0;
  double dp = 1;
  double dq = 1;
  while (dp > eps || dp < -eps || dq > eps || dq < -eps)  // eps 自行设定
  {
    double p0 = p;
    double q0 = q;
    b[n - 2] = a[n];
    c[n - 2] = b[n - 2];
    b[n - 3] = a[n - 1] - p0 * b[n - 2];
    c[n - 3] = b[n - 3] - p0 * b[n - 2];
    int j;
    for (j = n - 4; j >= 0; j--) {
      b[j] = a[j + 2] - p0 * b[j + 1] - q0 * b[j + 2];
      c[j] = b[j] - p0 * c[j + 1] - q0 * c[j + 2];
    }
    double r = a[1] - p0 * b[0] - q0 * b[1];
    double s = a[0] - q0 * b[0];
    double rp = c[1];
    double sp = b[0] - q0 * c[2];
    double rq = c[0];
    double sq = -q0 * c[1];
    dp = (rp * s - r * sp) / (rp * sq - rq * sp);
    dq = (r * sq - rq * s) / (rp * sq - rq * sp);
    *p += dp;
    *q += dq;
  }
}
```

参考资料与注释

[1] 林士谔. 论劈因法解高阶特征方程根值的应用问题. 数学进展，1963(03):207-217.

9.12.3 快速傅里叶变换

Authors: H-J-Granger, ranwen, abc1763613206, Ahacad, Allenyou1126, AndrewWayne, AngelKitty, AtomAlpaca, Backl1ght, billchenchina, c-forrest, CCXXXI, Cheuring, Chrogeek, ChungZH, countercurrent-time, DepletedPrism, Early0v0, EarthMessenger, Enter-tainer, F1shAndCat, GavinZhengOI, Gesrua, Great-designer, greyqz, Haohu Shen, henryrabbit, heroming, hly1204, Ir1d, isdanni, jiang1997, kenlig, Lewy Zeng, lucifer1004, Menci, muoshuosha, NachtgeistW, needtocalmdown, opsiff, ouuan, ouuan, partychicken, schtonn, Sshwy, sshwy, StudyingFather, SukkaW, Taoran-01, Tiphereth-A, TrisolarisHD, untitledunrevised, Xeonacid, YouXam, Yukimaikoriya

前置知识：复数→9.10@p1153.

本文将介绍一种算法，它支持在 $O(n\log n)$ 的时间内计算两个 n 次多项式的乘法，比朴素的 $O(n^2)$ 算法更高效. 由于两个整数的乘法也可以被当作多项式乘法，因此这个算法也可以用来加速大整数的乘法计算.

引入

我们现在引入两个多项式 A 和 B：

$$A = 5x^2 + 3x + 7$$

$$B = 7x^2 + 2x + 1$$

两个多项式相乘的积 $C = A \times B$，我们可以在 $O(n^2)$ 的时间复杂度中解得（这里 n 为 A 或者 B 多项式的次数）：

$$\begin{aligned} C &= A \times B \\ &= 35x^4 + 31x^3 + 60x^2 + 17x + 7 \end{aligned}$$

很明显，多项式 C 的系数 c_i 满足 $c_i = \sum_{j=0}^{i} a_j b_{i-j}$. 而对于这种朴素算法而言，计算每一项的时间复杂度都为 $O(n)$，一共有 $O(n)$ 项，那么时间复杂度为 $O(n^2)$.

能否加速使得它的时间复杂度降低呢? 如果使用快速傅里叶变换的话，那么我们可以使得其复杂度降低到 $O(n\log n)$.

傅里叶变换

傅里叶变换（Fourier Transform）是一种分析信号的方法，它可分析信号的成分，也可用这些成分合成信号. 许多波形可作为信号的成分，傅里叶变换用正弦波作为信号的成分.

设 $f(t)$ 是关于时间 t 的函数，则傅里叶变换可以检测频率 ω 的周期在 $f(t)$ 出现的程度：

$$F(\omega) = \mathbb{F}[f(t)] = \int_{-\infty}^{\infty} f(t)\mathrm{e}^{-\mathrm{i}\omega t}dt$$

它的逆变换是

$$f(t) = \mathbb{F}^{-1}[F(\omega)] = \frac{1}{2\pi}\int_{-\infty}^{\infty} F(\omega)\mathrm{e}^{\mathrm{i}\omega t}d\omega$$

逆变换的形式与正变换非常类似，分母 2π 恰好是指数函数的周期.

傅里叶变换相当于将时域的函数与周期为 2π 的复指数函数进行连续的内积. 逆变换仍旧为一个内积.

傅里叶变换有相应的卷积定理，可以将时域的卷积转化为频域的乘积，也可以将频域的卷积转化为时域的乘积.

离散傅里叶变换

离散傅里叶变换（Discrete Fourier transform，DFT）是傅里叶变换在时域和频域上都呈离散的形式，将信号的时域采样变换为其 DTFT（discrete-time Fourier transform）的频域采样.

傅里叶变换是积分形式的连续的函数内积，离散傅里叶变换是求和形式的内积.

设 $\{x_n\}_{n=0}^{N-1}$ 是某一满足有限性条件的序列，它的离散傅里叶变换（DFT）为：

$$X_k = \sum_{n=0}^{N-1} x_n \mathrm{e}^{-\mathrm{i}\frac{2\pi}{N}kn}$$

其中 e 是自然对数的底数，i 是虚数单位. 通常以符号 $\mathcal{F}$ 表示这一变换，即

$$\hat{x} = \mathcal{F}x$$

类似于积分形式，它的 **逆离散傅里叶变换**（IDFT）为：

$$x_n = \frac{1}{N}\sum_{k=0}^{N-1} X_k \mathrm{e}^{\mathrm{i}\frac{2\pi}{N}kn}$$

可以记为：

$$x = \mathcal{F}^{-1}\hat{x}$$

实际上，DFT 和 IDFT 变换式中和式前面的归一化系数并不重要．在上面的定义中，DFT 和 IDFT 前的系数分别为 1 和 $\frac{1}{N}$．有时我们会将这两个系数都改 $\frac{1}{\sqrt{N}}$．

离散傅里叶变换仍旧是时域到频域的变换．由于求和形式的特殊性，可以有其他的解释方法．

如果把序列 x_n 看作多项式 $f(x)$ 的 x^n 项系数，则计算得到的 X_k 恰好是多项式 $f(x)$ 代入单位根 $\mathrm{e}^{\frac{-2\pi\mathrm{i}k}{N}}$ 的点值 $f(\mathrm{e}^{\frac{-2\pi\mathrm{i}k}{N}})$．

这便构成了卷积定理的另一种解释办法，即对多项式进行特殊的求值操作．离散傅里叶变换恰好是多项式在单位根处进行求值．

例如计算：

$$\binom{n}{3}+\binom{n}{7}+\binom{n}{11}+\binom{n}{15}+\ldots$$

定义函数 $f(x)$ 为：

$$f(x)=(1+x)^n=\binom{n}{0}x^0+\binom{n}{1}x^1+\binom{n}{2}x^2+\binom{n}{3}x^3+\ldots$$

然后可以发现，代入四次单位根 $f(\mathrm{i})$ 得到这样的序列：

$$f(\mathrm{i})=(1+\mathrm{i})^n=\binom{n}{0}+\binom{n}{1}\mathrm{i}-\binom{n}{2}-\binom{n}{3}\mathrm{i}+\ldots$$

于是下面的求和恰好可以把其余各项消掉：

$$f(1)+\mathrm{i}f(\mathrm{i})-f(-1)-\mathrm{i}f(-\mathrm{i})=4\binom{n}{3}+4\binom{n}{7}+4\binom{n}{11}+4\binom{n}{15}+\ldots$$

因此这道数学题的答案为：

$$\binom{n}{3}+\binom{n}{7}+\binom{n}{11}+\binom{n}{15}+\ldots=\frac{2^n+\mathrm{i}(1+\mathrm{i})^n-\mathrm{i}(1-\mathrm{i})^n}{4}$$

这道数学题在单位根处求值，恰好构成离散傅里叶变换．

矩阵公式

由于离散傅立叶变换是一个 **线性** 算子，所以它可以用矩阵乘法来描述．在矩阵表示法中，离散傅立叶变换表示如下：

$$\begin{bmatrix} X_0 \\ X_1 \\ X_2 \\ \vdots \\ X_{N-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \alpha & \alpha^2 & \cdots & \alpha^{N-1} \\ 1 & \alpha^2 & \alpha^4 & \cdots & \alpha^{2(N-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \alpha^{N-1} & \alpha^{2(N-1)} & \cdots & \alpha^{(N-1)(N-1)} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_{N-1} \end{bmatrix}$$

其中 $\alpha = \mathrm{e}^{-\mathrm{i}\frac{2\pi}{N}}$．

快速傅里叶变换

FFT 是一种高效实现 DFT 的算法，称为快速傅立叶变换（Fast Fourier Transform，FFT）．它对傅里叶变换的理论并没有新的发现，但是对于在计算机系统或者说数字系统中应用离散傅立叶变换，可以说是进了一大步．快速数论变换（NTT）是快速傅里叶变换（FFT）在数论基础上的实现．

在 1965 年，Cooley 和 Tukey 发表了快速傅里叶变换算法．事实上 FFT 早在这之前就被发现过了，但是在当时现代计算机并未问世，人们没有意识到 FFT 的重要性．一些调查者认为 FFT 是由 Runge 和 König 在 1924 年发现的．但事实上高斯早在 1805 年就发明了这个算法，但一直没有发表．

分治法实现

FFT 算法的基本思想是分治．就 DFT 来说，它分治地来求当 $x=\omega_n^k$ 的时候 $f(x)$ 的值．基 - 2 FFT 的分治思想体现在将多项式分为奇次项和偶次项处理．

举个例子，对于一共 8 项的多项式：

$$f(x)=a_0+a_1x+a_2x^2+a_3x^3+a_4x^4+a_5x^5+a_6x^6+a_7x^7$$

按照次数的奇偶来分成两组，然后右边提出来一个 x：

$$\begin{aligned}f(x)&=(a_0+a_2x^2+a_4x^4+a_6x^6)+(a_1x+a_3x^3+a_5x^5+a_7x^7)\\&=(a_0+a_2x^2+a_4x^4+a_6x^6)+x(a_1+a_3x^2+a_5x^4+a_7x^6)\end{aligned}$$

分别用奇偶次次项数建立新的函数：

$$G(x)=a_0+a_2x+a_4x^2+a_6x^3$$

$$H(x)=a_1+a_3x+a_5x^2+a_7x^3$$

那么原来的 $f(x)$ 用新函数表示为：

$$f(x)=G(x^2)+x\times H(x^2)$$

利用偶数次单位根的性质 $\omega_n^i=-\omega_n^{i+n/2}$，和 $G(x^2)$ 和 $H(x^2)$ 是偶函数，我们知道在复平面上 ω_n^i 和 $\omega_n^{i+n/2}$ 的 $G(x^2)$ 的 $H(x^2)$ 对应的值相同．得到：

$$\begin{aligned}f(\omega_n^k)&=G((\omega_n^k)^2)+\omega_n^k\times H((\omega_n^k)^2)\\&=G(\omega_n^{2k})+\omega_n^k\times H(\omega_n^{2k})\\&=G(\omega_{n/2}^k)+\omega_n^k\times H(\omega_{n/2}^k)\end{aligned}$$

和：

$$\begin{aligned}f(\omega_n^{k+n/2})&=G(\omega_n^{2k+n})+\omega_n^{k+n/2}\times H(\omega_n^{2k+n})\\&=G(\omega_n^{2k})-\omega_n^k\times H(\omega_n^{2k})\\&=G(\omega_{n/2}^k)-\omega_n^k\times H(\omega_{n/2}^k)\end{aligned}$$

因此我们求出了 $G(\omega_{n/2}^k)$ 和 $H(\omega_{n/2}^k)$ 后，就可以同时求出 $f(\omega_n^k)$ 和 $f(\omega_n^{k+n/2})$．于是对 G 和 H 分别递归 DFT 即可．

考虑到分治 DFT 能处理的多项式长度只能是 $2^m(m\in\mathbf{N}^*)$，否则在分治的时候左右不一样长，右边就取不到系数了．所以要在第一次 DFT 之前就把序列向上补成长度为 $2^m(m\in\mathbf{N}^*)$（高次系数补 0）、最高项次数为 2^m-1 的多项式．

在代入值的时候，因为要代入 n 个不同值，所以我们代入 $\omega_n^0,\omega_n^1,\omega_n^2,\cdots,\omega_n^{n-1}(n=2^m(m\in\mathbf{N}^*))$ 一共 2^m 个不同值．

代码实现方面，STL 提供了复数的模板，当然也可以手动实现．两者区别在于，使用 STL 的 `complex` 可以调用 `exp` 函数求出 ω_n．但事实上使用欧拉公式得到的虚数来求 ω_n 也是等价的．

以上就是 FFT 算法中 DFT 的介绍，它将一个多项式从系数表示法变成了点值表示法．

值的注意的是，因为是单位复根，所以说我们需要令 n 项式的高位补为零，使得 $n=2^k,k\in\mathbf{N}^*$．

递归版 FFT

```cpp
#include <cmath>
#include <complex>

using Comp = std::complex<double>;  // STL complex

constexpr Comp I(0, 1);  // i
```

```cpp
constexpr int MAX_N = 1 << 20;

Comp tmp[MAX_N];

// rev=1, DFT; rev=-1, IDFT
// 应用完本函数后需要注意归一化系数的处理
void DFT(Comp* f, int n, int rev) {
  if (n == 1) return;
  for (int i = 0; i < n; ++i) tmp[i] = f[i];
  // 偶数放左边，奇数放右边
  for (int i = 0; i < n; ++i) {
    if (i & 1)
      f[n / 2 + i / 2] = tmp[i];
    else
      f[i / 2] = tmp[i];
  }
  Comp *g = f, *h = f + n / 2;
  // 递归 DFT
  DFT(g, n / 2, rev), DFT(h, n / 2, rev);
  // cur 是当前单位复根，对于 k = 0 而言，它对应的单位复根 omega^0_n = 1.
  // step 是两个单位复根的差，即满足 omega^k_n = step*omega^{k-1}*n,
  // 定义等价于 exp(I*(-2*M_PI/n*rev))
  Comp cur(1, 0), step(cos(2 * M_PI / n), sin(-2 * M_PI * rev / n));
  for (int k = 0; k < n / 2;
       ++k) {  // F(omega^k_n) = G(omega^k*{n/2}) +
omega^k*n\*H(omega^k*{n/2})
    tmp[k] = g[k] + cur * h[k];
    // F(omega^{k+n/2}*n) = G(omega^k*{n/2}) - omega^k_n*H(omega^k\_{n/2})
    tmp[k + n / 2] = g[k] - cur * h[k];
    cur *= step;
  }
  for (int i = 0; i < n; ++i) f[i] = tmp[i];
}
```

时间复杂度 $O(n \log n)$.

倍增法实现

这个算法还可以从「分治」的角度继续优化. 对于基 - 2 FFT，我们每一次都会把整个多项式的奇数次项和偶数次项系数分开，一直分到只剩下一个系数. 但是，这个递归的过程需要更多的内存. 因此，我们可以先「模仿递归」把这些系数在原数组中「拆分」，然后再「倍增」地去合并这些算出来的值.

对于「拆分」，可以使用位逆序置换实现.

对于「合并」，使用蝶形运算优化可以做到只用 $O(1)$ 的额外空间来完成.

位逆序置换

以 8 项多项式为例，模拟拆分的过程:

- 初始序列为 $\{x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7\}$
- 一次二分之后 $\{x_0, x_2, x_4, x_6\}, \{x_1, x_3, x_5, x_7\}$
- 两次二分之后 $\{x_0, x_4\}\{x_2, x_6\}, \{x_1, x_5\}, \{x_3, x_7\}$
- 三次二分之后 $\{x_0\}\{x_4\}\{x_2\}\{x_6\}\{x_1\}\{x_5\}\{x_3\}\{x_7\}$

规律：其实就是原来的那个序列，每个数用二进制表示，然后把二进制翻转对称一下，就是最终那个位置的下标．比如 x_1 是 001，翻转是 100，也就是 4，而且最后那个位置确实是 4．我们称这个变换为位逆序置换（bit-reversal permutation），证明留给读者自证．

根据它的定义，我们可以在 $O(n)$ 的时间内求出每个数变换后的结果：

位逆序置换实现（$O(n)$）

```
/*
 * 进行 FFT 和 IFFT 前的反置变换
 * 位置 i 和 i 的二进制反转后的位置互换
 * len 必须为 2 的幂
 */
void change(Complex y[], int len) {
  // 一开始 i 是 0...01，而 j 是 10...0，在二进制下相反对称．
  // 之后 i 逐渐加一，而 j 依然维持着和 i 相反对称，一直到 i = 1...11．
  for (int i = 1, j = len / 2, k; i < len - 1; i++) {
    // 交换互为小标反转的元素，i < j 保证交换一次
    if (i < j) swap(y[i], y[j]);
    // i 做正常的 + 1，j 做反转类型的 + 1，始终保持 i 和 j 是反转的．
    // 这里 k 代表了 0 出现的最高位．j 先减去高位的全为 1 的数字，
    // 直到遇到了 0，之后再加上即可．
    // 考虑 j 中比特位的翻转次数，最高位将会翻转 n 次，
    // 第二高位将会翻转 n/2 次，以此类推，所以时间复杂度为：
    // T(n) = n + n/2 + n/4 + ... = O(n)
    k = len / 2;
    while (j >= k) {
      j = j - k;
      k = k / 2;
    }
    j += k;
  }
}
```

位逆序置换也可以 $O(n)$ 从小到大递推实现，设 $len = 2^k$，其中 k 表示二进制数的长度，设 $R(x)$ 表示长度为 k 的二进制数 x 翻转后的数（高位补 0）．我们要求的是 $R(0), R(1), \cdots, R(n-1)$．

首先 $R(0) = 0$．

我们从小到大求 $R(x)$．因此在求 $R(x)$ 时，$R\left(\left\lfloor \frac{x}{2} \right\rfloor\right)$ 的值是已知的．因此我们把 x 右移一位（除以 2），然后翻转，再右移一位，就得到了 x **除了（二进制）个位** 之外其它位的翻转结果．

考虑个位的翻转结果：如果个位是 0，翻转之后最高位就是 0．如果个位是 1，则翻转后最高位是 1，因此还要加上 $\frac{len}{2} = 2^{k-1}$．综上

$$R(x) = \left\lfloor \frac{R(\lfloor \frac{x}{2} \rfloor)}{2} \right\rfloor + (x \bmod 2) \times \frac{len}{2}$$

举个例子：设 $k = 5$，$len = (100000)_2$．为了翻转 $(11001)_2$：

1. 考虑 $(1100)_2$，我们知道 $R((1100)_2) = R((01100)_2) = (00110)_2$，再右移一位就得到了 $(00011)_2$．
2. 考虑个位，如果是 1，它就要翻转到数的最高位，即翻转数加上 $(10000)_2 = 2^{k-1}$，如果是 0 则不用更改．

位逆序置换实现（$O(n)$）

```cpp
// 同样需要保证 len 是 2 的幂
// 记 rev[i] 为 i 翻转后的值
void change(Complex y[], int len) {
  for (int i = 0; i < len; ++i) {
    rev[i] = rev[i >> 1] >> 1;
    if (i & 1) {  // 如果最后一位是 1，则翻转成 len/2
      rev[i] |= len >> 1;
    }
  }
  for (int i = 0; i < len; ++i) {
    if (i < rev[i]) {  // 保证每对数只翻转一次
      swap(y[i], y[rev[i]]);
    }
  }
  return;
}
```

蝶形运算优化

已知 $G(\omega_{n/2}^k)$ 和 $H(\omega_{n/2}^k)$ 后，需要使用下面两个式子求出 $f(\omega_n^k)$ 和 $f(\omega_n^{k+n/2})$：

$$f(\omega_n^k) = G(\omega_{n/2}^k) + \omega_n^k \times H(\omega_{n/2}^k)$$

$$f(\omega_n^{k+n/2}) = G(\omega_{n/2}^k) - \omega_n^k \times H(\omega_{n/2}^k)$$

使用位逆序置换后，对于给定的 n, k：

- $G(\omega_{n/2}^k)$ 的值存储在数组下标为 k 的位置，$H(\omega_{n/2}^k)$ 的值存储在数组下标为 $k + \dfrac{n}{2}$ 的位置.
- $f(\omega_n^k)$ 的值将存储在数组下标为 k 的位置，$f(\omega_n^{k+n/2})$ 的值将存储在数组下标为 $k + \dfrac{n}{2}$ 的位置.

因此可以直接在数组下标为 k 和 $k + \frac{n}{2}$ 的位置进行覆写，而不用开额外的数组保存值. 此方法即称为**蝶形运算**，或更准确的，基 - 2 蝶形运算.

再详细说明一下如何借助蝶形运算完成所有段长度为 $\frac{n}{2}$ 的合并操作：

1. 令段长度为 $s = \frac{n}{2}$；
2. 同时枚举序列 $\{G(\omega_{n/2}^k)\}$ 的左端点 $l_g = 0, 2s, 4s, \cdots, N-2s$ 和序列 $\{H(\omega_{n/2}^k)\}$ 的左端点 $l_h = s, 3s, 5s, \cdots, N-s$；
3. 合并两个段时，枚举 $k = 0, 1, 2, \cdots, s-1$，此时 $G(\omega_{n/2}^k)$ 存储在数组下标为 $l_g + k$ 的位置，$H(\omega_{n/2}^k)$ 存储在数组下标为 $l_h + k$ 的位置；
4. 使用蝶形运算求出 $f(\omega_n^k)$ 和 $f(\omega_n^{k+n/2})$，然后直接在原位置覆写.

快速傅里叶逆变换

傅里叶逆变换可以用傅里叶变换表示. 对此我们有两种理解方式.

线性代数角度

IDFT（傅里叶反变换）的作用，是把目标多项式的点值形式转换成系数形式. 而 DFT 本身是个线性变换，可以理解为将目标多项式当作向量，左乘一个矩阵得到变换后的向量，以模拟把单位复根代入多项式的过程：

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n^1 & \omega_n^2 & \omega_n^3 & \cdots & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^4 & \omega_n^6 & \cdots & \omega_n^{2(n-1)} \\ 1 & \omega_n^3 & \omega_n^6 & \omega_n^9 & \cdots & \omega_n^{3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \omega_n^{3(n-1)} & \cdots & \omega_n^{(n-1)^2} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1} \end{bmatrix}$$

现在我们已经得到最左边的结果了，中间的 x 值在目标多项式的点值表示中也是一一对应的，所以，根据矩阵的基础知识，我们只要在式子两边左乘中间那个大矩阵的逆矩阵就行了.

由于这个矩阵的元素非常特殊，它的逆矩阵也有特殊的性质，就是每一项 **取倒数**，再 **除以变换的长度** n，就能得到它的逆矩阵.

注意：傅里叶变换的长度，并不是多项式的长度，变换的长度应比乘积多项式的长度长. 待相乘的多项式不够长，需要在高次项处补 0.

为了使计算的结果为原来的倒数，根据欧拉公式，可以得到

$$\frac{1}{\omega_k} = \omega_k^{-1} = \mathrm{e}^{-\frac{2\pi \mathrm{i}}{k}} = \cos\left(\frac{2\pi}{k}\right) + \mathrm{i}\sin\left(-\frac{2\pi}{k}\right)$$

因此我们可以尝试着把单位根 ω_k 取成 $\mathrm{e}^{-\frac{2\pi \mathrm{i}}{k}}$，这样我们的计算结果就会变成原来的倒数，之后唯一多的操作就只有再 **除以它的长度** n，而其它的操作过程与 DFT 是完全相同的. 我们可以定义一个函数，在里面加一个参数 1 或者是 -1，然后把它乘到 π 上. 传入 1 就是 DFT，传入 -1 就是 IDFT.

单位复根周期性

利用单位复根的周期性同样可以理解 IDFT 与 DFT 之间的关系.

考虑原本的多项式是 $f(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1}x^{n-1} = \sum_{i=0}^{n-1} a_i x^i$. 而 IDFT 就是把你的点值表示还原为系数表示.

考虑 **构造法**. 我们已知 $y_i = f(\omega_n^i), i \in \{0, 1, \cdots, n-1\}$，求 $\{a_0, a_1, \cdots, a_{n-1}\}$. 构造多项式如下

$$A(x) = \sum_{i=0}^{n-1} y_i x^i$$

相当于把 $\{y_0, y_1, y_2, \cdots, y_{n-1}\}$ 当做多项式 A 的系数表示法.

这时我们有两种推导方式，这对应了两种实现方法.

方法一

设 $b_i = \omega_n^{-i}$，则多项式 A 在 $x = b_0, b_1, \cdots, b_{n-1}$ 处的点值表示法为 $\{A(b_0), A(b_1), \cdots, A(b_{n-1})\}$.

对 $A(x)$ 的定义式做一下变换，可以将 $A(b_k)$ 表示为

$$\begin{aligned} A(b_k) &= \sum_{i=0}^{n-1} f(\omega_n^i)\omega_n^{-ik} = \sum_{i=0}^{n-1} \omega_n^{-ik} \sum_{j=0}^{n-1} a_j (\omega_n^i)^j \\ &= \sum_{i=0}^{n-1}\sum_{j=0}^{n-1} a_j \omega_n^{i(j-k)} = \sum_{j=0}^{n-1} a_j \sum_{i=0}^{n-1} \left(\omega_n^{j-k}\right)^i \end{aligned}$$

记 $S(\omega_n^a) = \sum_{i=0}^{n-1} \left(\omega_n^a\right)^i$.

当 $a = 0 \pmod n$ 时，$S(\omega_n^a) = n$.

当 $a \neq 0 \pmod n$ 时，我们错位相减

$$S(\omega_n^a) = \sum_{i=0}^{n-1} (\omega_n^a)^i$$

$$\omega_n^a S(\omega_n^a) = \sum_{i=1}^{n} (\omega_n^a)^i$$

$$S(\omega_n^a) = \frac{(\omega_n^a)^n - (\omega_n^a)^0}{\omega_n^a - 1} = 0$$

也就是说

$$S(\omega_n^a) = \begin{cases} n, a = 0 \\ 0, a \neq 0 \end{cases}$$

那么代回原式

$$A(b_k) = \sum_{j=0}^{n-1} a_j S(\omega_n^{j-k}) = a_k \cdot n$$

也就是说给定点 $b_i = \omega_n^{-i}$，则 A 的点值表示法为

$$\begin{aligned} &\{(b_0, A(b_0)), (b_1, A(b_1)), \cdots, (b_{n-1}, A(b_{n-1}))\} \\ =\ &\{(b_0, a_0 \cdot n), (b_1, a_1 \cdot n), \cdots, (b_{n-1}, a_{n-1} \cdot n)\} \end{aligned}$$

综上所述，我们取单位根为其倒数，对 $\{y_0, y_1, y_2, \cdots, y_{n-1}\}$ 跑一遍 FFT，然后除以 n 即可得到 $f(x)$ 的系数表示.

方法二

我们直接将 ω_n^i 代入 $A(x)$.

推导的过程与方法一大同小异，最终我们得到 $A(\omega_n^k) = \sum_{j=0}^{n-1} a_j S(\omega_n^{j+k})$.

当且仅当 $j + k = 0 \pmod n$ 时有 $S(\omega_n^{j+k}) = n$，否则为 0. 因此 $A(\omega_n^k) = a_{n-k} \cdot n$.

这意味着我们将 $\{y_0, y_1, y_2, \cdots, y_{n-1}\}$ 做 DFT 变换后除以 n，再反转后 $n-1$ 个元素，同样可以还原 $f(x)$ 的系数表示.

代码实现

所以我们 FFT 函数可以集 DFT 和 IDFT 于一身. 代码实现如下：

非递归版 FFT（对应方法一）

```
/*
 * 做 FFT
 * len 必须是 2^k 形式
 * on == 1 时是 DFT，on == -1 时是 IDFT
 */
void fft(Complex y[], int len, int on) {
  // 位逆序置换
  change(y, len);
  // 模拟合并过程，一开始，从长度为一合并到长度为二，一直合并到长度为 len.
  for (int h = 2; h <= len; h <<= 1) {
    // wn：当前单位复根的间隔：w^1_h
    Complex wn(cos(2 * PI / h), sin(on * 2 * PI / h));
    // 合并，共 len / h 次.
    for (int j = 0; j < len; j += h) {
      // 计算当前单位复根，一开始是 1 = w^0_n，之后是以 wn 为间隔递增： w^1_n
      // ...
      Complex w(1, 0);
```

```
      for (int k = j; k < j + h / 2; k++) {
        // 左侧部分和右侧是子问题的解
        Complex u = y[k];
        Complex t = w * y[k + h / 2];
        // 这就是把两部分分治的结果加起来
        y[k] = u + t;
        y[k + h / 2] = u - t;
        // 后半个 「step」 中的ω一定和 「前半个」 中的成相反数
        // 「红圈」上的点转一整圈「转回来」，转半圈正好转成相反数
        // 一个数相反数的平方与这个数自身的平方相等
        w = w * wn;
      }
    }
  }
  // 如果是 IDFT，它的逆矩阵的每一个元素不只是原元素取倒数，还要除以长度 len.
  if (on == -1) {
    for (int i = 0; i < len; i++) {
      y[i].x /= len;
      y[i].y /= len;
    }
  }
}
```

非递归版 FFT（对应方法二）

```
/*
 * 做 FFT
 * len 必须是 2^k 形式
 * on == 1 时是 DFT，on == -1 时是 IDFT
 */
void fft(Complex y[], int len, int on) {
  change(y, len);
  for (int h = 2; h <= len; h <<= 1) {              // 模拟合并过程
    Complex wn(cos(2 * PI / h), sin(2 * PI / h));  // 计算当前单位复根
    for (int j = 0; j < len; j += h) {
      Complex w(1, 0);  // 计算当前单位复根
      for (int k = j; k < j + h / 2; k++) {
        Complex u = y[k];
        Complex t = w * y[k + h / 2];
        y[k] = u + t;  // 这就是把两部分分治的结果加起来
        y[k + h / 2] = u - t;
        // 后半个 「step」 中的ω一定和 「前半个」 中的成相反数
        // 「红圈」上的点转一整圈「转回来」，转半圈正好转成相反数
        // 一个数相反数的平方与这个数自身的平方相等
        w = w * wn;
      }
    }
  }
  if (on == -1) {
    reverse(y + 1, y + len);
    for (int i = 0; i < len; i++) {
```

```
      y[i].x /= len;
      y[i].y /= len;
    }
  }
}
```

FFT 模板（HDU 1402 - A * B Problem Plus[1]）

```cpp
#include <cmath>
#include <cstring>
#include <iostream>

const double PI = acos(-1.0);

struct Complex {
  double x, y;

  Complex(double _x = 0.0, double _y = 0.0) {
    x = _x;
    y = _y;
  }

  Complex operator-(const Complex &b) const {
    return Complex(x - b.x, y - b.y);
  }

  Complex operator+(const Complex &b) const {
    return Complex(x + b.x, y + b.y);
  }

  Complex operator*(const Complex &b) const {
    return Complex(x * b.x - y * b.y, x * b.y + y * b.x);
  }
};

/*
 * 进行 FFT 和 IFFT 前的反置变换
 * 位置 i 和 i 的二进制反转后的位置互换
 *len 必须为 2 的幂
 */
void change(Complex y[], int len) {
  int i, j, k;

  for (int i = 1, j = len / 2; i < len - 1; i++) {
    if (i < j) std::swap(y[i], y[j]);

    // 交换互为小标反转的元素，i<j 保证交换一次
    // i 做正常的 + 1，j 做反转类型的 + 1，始终保持 i 和 j 是反转的
    k = len / 2;

    while (j >= k) {
```

```
      j = j - k;
      k = k / 2;
    }

    if (j < k) j += k;
  }
}

/*
 * 做 FFT
 *len 必须是 2^k 形式
 *on == 1 时是 DFT, on == -1 时是 IDFT
 */
void fft(Complex y[], int len, int on) {
  change(y, len);

  for (int h = 2; h <= len; h <<= 1) {
    Complex wn(cos(2 * PI / h), sin(on * 2 * PI / h));

    for (int j = 0; j < len; j += h) {
      Complex w(1, 0);

      for (int k = j; k < j + h / 2; k++) {
        Complex u = y[k];
        Complex t = w * y[k + h / 2];
        y[k] = u + t;
        y[k + h / 2] = u - t;
        w = w * wn;
      }
    }
  }

  if (on == -1) {
    for (int i = 0; i < len; i++) {
      y[i].x /= len;
    }
  }
}

constexpr int MAXN = 200020;
Complex x1[MAXN], x2[MAXN];
char str1[MAXN / 2], str2[MAXN / 2];
int sum[MAXN];
using std::cin;
using std::cout;

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  while (cin >> str1 >> str2) {
    int len1 = strlen(str1);
    int len2 = strlen(str2);
    int len = 1;
```

```
96
97      while (len < len1 * 2 || len < len2 * 2) len <<= 1;
98
99      for (int i = 0; i < len1; i++) x1[i] = Complex(str1[len1 - 1 - i] -
    '0', 0);
100
101     for (int i = len1; i < len; i++) x1[i] = Complex(0, 0);
102
103     for (int i = 0; i < len2; i++) x2[i] = Complex(str2[len2 - 1 - i] -
    '0', 0);
104
105     for (int i = len2; i < len; i++) x2[i] = Complex(0, 0);
106
107     fft(x1, len, 1);
108     fft(x2, len, 1);
109
110     for (int i = 0; i < len; i++) x1[i] = x1[i] * x2[i];
111
112     fft(x1, len, -1);
113
114     for (int i = 0; i < len; i++) sum[i] = int(x1[i].x + 0.5);
115
116     for (int i = 0; i < len; i++) {
117       sum[i + 1] += sum[i] / 10;
118       sum[i] %= 10;
119     }
120
121     len = len1 + len2 - 1;
122
123     while (sum[len] == 0 && len > 0) len--;
124
125     for (int i = len; i >= 0; i--) cout << char(sum[i] + '0');
126
127     cout << '\n';
128   }
129
130   return 0;
131 }
```

参考文献

1. 桃酱的算法笔记[2].

外部参考资料

[1] HDU 1402 - A * B Problem Plus

[2] 桃酱的算法笔记

9.12.4 快速数论变换

Authors: ChungZH, Yukimaikoriya, tigerruanyifan, isdanni, Saisyc, 383494, Tiphereth-A, XuYueming520

简介

数论变换（number-theoretic transform, NTT）是离散傅里叶变换（DFT）在数论基础上的实现；**快速数论变换**（fast number-theoretic transform, FNTT）是 快速傅里叶变换→9.12.3@p1527（FFT）在数论基础上的实现.

数论变换 是一种计算卷积（convolution）的快速算法. 最常用算法就包括了前文提到的快速傅里叶变换. 然而快速傅立叶变换具有一些实现上的缺点, 举例来说, 资料向量必须乘上复数系数的矩阵加以处理, 而且每个复数系数的实部和虚部是一个正弦及余弦函数, 因此大部分的系数都是浮点数, 也就是说, 必须做复数而且是浮点数的运算, 因此计算量会比较大, 而且浮点数运算产生的误差会比较大.

NTT 解决的是多项式乘法带模数的情况, 可以说有些受模数的限制, 数也比较大. 目前最常见的模数是 998244353.

前置知识

学习数论变换需要前置知识：离散傅里叶变换、生成子群、原根→9.11.18@p1275、离散对数. 相关知识可以在对应页面中学习，此处不再赘述.

定义

数论变换

在数学中，NTT 是关于任意 环→9.16.1@p1805 上的离散傅立叶变换（DFT）. 在有限域的情况下，通常称为数论变换（NTT）.

数论变换（NTT）是通过将离散傅立叶变换化为 $F=\mathbb{Z}/p$，整数模质数 p. 这是一个 **有限域**，只要 n 可除 $p-1$，就存在本原 n 次方根，所以我们有 $p=\xi n+1$ 对于 正整数 ξ. 具体来说，对于质数 $p=qn+1,(n=2^m)$，原根 g 满足 $g^{qn}\equiv 1 \pmod p$, 将 $g_n=g^q \pmod p$ 看做 ω_n 的等价，则其满足相似的性质，比如 $g_n^n\equiv 1 \pmod p, g_n^{n/2}\equiv -1 \pmod p$.

因为这里涉及到数论变化，所以 N（为了区分 FFT 中的 n，我们把这里的 n 称为 N）可以比 FFT 中的 n 大，但是只要把 $\frac{qN}{n}$ 看做这里的 q 就行了，能够避免大小问题.

常见的有：

$$p=167772161=5\times 2^{25}+1, g=3$$

$$p=469762049=7\times 2^{26}+1, g=3$$

$$p=754974721=3^2\times 5\times 2^{24}+1, g=11$$

$$p=998244353=7\times 17\times 2^{23}+1, g=3$$

$$p=1004535809=479\times 2^{21}+1, g=3$$

就是 g^{qn} 的等价 $\mathrm{e}^{2\pi \mathrm{i}n}$.

迭代到长度 l 时 $g_l=g^{\frac{p-1}{l}}$，或者 $\omega_n=g_l=g_N^{\frac{N}{l}}=g_N^{\frac{p-1}{l}}$.

快速数论变换

快速数论变换（FNTT）是数论变换（NTT）增加分治操作之后的快速算法.

快速数论变换使用的分治办法，与快速傅里叶变换使用的分治办法完全一致. 这意味着，只需在快速傅里叶变换的代码基础上进行简单修改，即可得到快速数论变换的代码.

在算法竞赛中常提到的 NTT 一词，往往实际指的是快速数论变换，一般默认「数论变换」是指「快速数论变换」.

这样简写的逻辑与快速傅里叶变换相似. 事实上,「快速傅里叶变换」(FFT)一词指的是「快速离散傅里叶变换」(FDFT),但由于「快速」只能作用于离散,甚至是本原单位根阶数为 2 的幂的特殊情形,不能作用于连续,因此「离散」一词被省略掉,FDFT 变为 FFT,即 FFT 永远指的是特殊的离散情形.

数论变换或快速数论变换是在取模意义下进行的操作,不存在连续的情形,永远是离散的,自然也无需提到离散一词.

在算法领域,不进行提速的操作是无意义的. 在快速傅里叶变换中介绍 DFT 一词,是因为 DFT 在信号处理、图像处理领域也有其他的具体应用,同时 DFT 也是 FFT 的原理或前置知识.

在不引起混淆的情形下,常用 NTT 来代指 FNTT. 为了不引起下文进一步介绍的混淆,下文的 NTT 与 FNTT 两个词进行了分离.

DFT、FFT、NTT、FNTT 的具体关系是:

- 在 DFT 与 NTT 的基础上,增加分治操作,得到 FFT 与 FNTT. 分治操作的办法与原理,可以参见快速傅里叶变换一文.
- 在 DFT 与 FFT 的基础上,将复数加法与复数乘法替换为模 p 意义下的加法和乘法,一般大小限制在 0 到 $p-1$ 之间;将本原单位根改为模 p 意义下的相同阶数的本原单位根,阶数为 2 的幂,即可得到 NTT 与 FNTT.

由于替换的运算只涉及加法和乘法,因此 DFT、FFT、NTT、FNTT 拥有相同的原理,均在满足加法与乘法的环上进行,无需域上满足除法运算的更加严格的条件.

事实上,只要拥有原根,即群论中的生成元,该模数下的 NTT 或 FNTT 即可进行. 考虑到模数为 1、2 和 4 的情形太小,不具有实际意义,对于奇素数 p 和正整数 α,只要给出模数为 p^α 和 $2p^\alpha$ 的原根 g,采用同样的办法,则 NTT 或 FNTT 仍然可以进行.

模板

Library Checker - Convolution[1]

```cpp
#include <algorithm>
#include <cstdio>
using namespace std;

constexpr int N = 1 << 20, P = 998244353;

int qpow(int x, int y) {
  int res = 1;
  while (y) {
    if (y & 1) res = 1ll * res * x % P;
    x = 1ll * x * x % P;
    y >>= 1;
  }
  return res;
}

int r[N];

void ntt(int *x, int lim, int opt) {
  for (int i = 0; i < lim; ++i)
    if (r[i] < i) swap(x[i], x[r[i]]);
  for (int m = 2; m <= lim; m <<= 1) {
    int k = m >> 1;
```

```
    int gn = qpow(3, (P - 1) / m);
    for (int i = 0; i < lim; i += m) {
      int g = 1;
      for (int j = 0; j < k; ++j, g = 1ll * g * gn % P) {
        int tmp = 1ll * x[i + j + k] * g % P;
        x[i + j + k] = (x[i + j] - tmp + P) % P;
        x[i + j] = (x[i + j] + tmp) % P;
      }
    }
  }
  if (opt == -1) {
    reverse(x + 1, x + lim);
    int inv = qpow(lim, P - 2);
    for (int i = 0; i < lim; ++i) x[i] = 1ll * x[i] * inv % P;
  }
}

int A[N], B[N], C[N];

int main() {
  int n, m;
  scanf("%d %d", &n, &m);
  for (int i = 0; i < n; i++) scanf("%d", &A[i]);
  for (int i = 0; i < m; i++) scanf("%d", &B[i]);
  int N = max(n, m), lim = 1;
  while (lim < (N << 1)) lim <<= 1;
  for (int i = 0; i < lim; ++i) r[i] = (i & 1) * (lim >> 1) + (r[i >> 1]
>> 1);
  ntt(A, lim, 1);
  ntt(B, lim, 1);
  for (int i = 0; i < lim; ++i) C[i] = 1ll * A[i] * B[i] % P;
  ntt(C, lim, -1);
  for (int i = 0; i < n + m - 1; i++)
    printf("%d%c", C[i], " \n"[i == n + m - 2]);
  return 0;
}

```

参考资料与拓展阅读

1. FWT（快速沃尔什变换）零基础详解 qaq（ACM/OI）[2]
2. FFT（快速傅里叶变换）0 基础详解！附 NTT（ACM/OI）[3]
3. Number-theoretic transform(NTT) - Wikipedia[4]
4. Tutorial on FFT/NTT—The tough made simple. (Part 1)[5]
5. NTT 模板 - BlackJack_ CSDN 博客[6]

外部参考资料

[1] Library Checker - Convolution 

[2] FWT（快速沃尔什变换）零基础详解 qaq（ACM/OI） 

[3] FFT（快速傅里叶变换）0 基础详解！附 NTT（ACM/OI） 

[4] Number-theoretic transform(NTT) - Wikipedia 

[5] Tutorial on FFT/NTT—The tough made simple. (Part 1) 

[6] NTT 模板 - BlackJack_ CSDN 博客 

9.12.5 快速沃尔什变换

Authors: Xeonacid, nocriz, ZnPdCo

简介

沃尔什转换（Walsh Transform）[2]是在频谱分析上作为离散傅立叶变换的替代方案的一种方法，在信号处理中应用很广泛．FFT 是 double 类型的，但是 Walsh 把信号在不同震荡频率方波下拆解，因此所有的系数都是绝对值大小相同的整数，这使得不需要作浮点数的乘法运算，提高了运算速度．

所以，FWT 和 FFT 的核心思想应该是相同的，都是对数组的变换．我们记对数组 A 进行快速沃尔什变换后得到的结果为 $FWT[A]$．

那么 FWT 核心思想就是：

我们需要一个新序列 C，由序列 A 和序列 B 经过某运算规则得到，即 $C = A \cdot B$；

我们先正向得到序列 $FWT[A], FWT[B]$，再根据 $FWT[C] = FWT[A] \cdot FWT[B]$ 在 $O(n)$ 的时间复杂度内求出 $FWT[C]$，其中 $\cdot$ 是序列对应位置相乘；

然后逆向运算得到原序列 C．时间复杂度为 $O(n \log n)$．

在算法竞赛中，FWT 是用于解决对下标进行位运算卷积问题的方法．

公式：$C_i = \sum_{i=j \oplus k} A_j B_k$

（其中 $\oplus$ 是二元位运算中的某一种）

下面我们举 $\cup$（按位或）、$\cap$（按位与）和 $\oplus$（按位异或）为例．

FWT 的运算

或运算

如果有 $k = i \cup j$，那么 i 的二进制位为 1 的位置和 j 的二进制位为 1 的位置肯定是 k 的二进制位为 1 的位置的子集．

现在要得到 $FWT[C] = FWT[A] \cdot FWT[B]$，我们就要构造这个 FWT 的规则．

我们按照定义，显然可以构造 $FWT[A]_i = A'_i = \sum_{i=i \cup j} A_j$，来表示 j 满足二进制中 1 为 i 的子集．

那么有：

$$\begin{aligned} FWT[A]_i \cdot FWT[B]_i &= \left(\sum_{i \cup j = i} A_j \right) \left(\sum_{i \cup k = i} B_k \right) \\ &= \sum_{i \cup j = i} \sum_{i \cup k = i} A_j B_k \\ &= \sum_{i \cup (j \cup k) = i} A_j B_k \\ &= FWT[C]_i \end{aligned}$$

那么我们接下来看 $FWT[A]$ 怎么求．

首先肯定不能枚举了，复杂度为 $O(n^2)$．既然不能整体枚举，我们就考虑分治．

我们把整个区间二分，其实二分区间之后，下标写成二进制形式是有规律可循的．

我们令 A_0 表示 A 的前一半，A_1 表示区间的后一半，那么 A_0 就是 A 下标最大值的最高位为 0，他的子集就是他本身的子集（因为最高位为 0 了），但是 A_1 的最高位是 1，他满足条件的子集不仅仅是他本身，还包最高位为 0 的子集，即

[2]维基百科[1]

$$FWT[A] = merge(FWT[A_0], FWT[A_0] + FWT[A_1])$$

其中 merge 表示像字符串拼接一样把两个数组拼起来，$+$ 就是普通加法，表示对应二进制位相加.

这样我们就通过二分能在 $O(\log n)$ 的时间复杂度内完成拼接，每次拼接的时候要完成一次运算，也就是说在 $O(n \log n)$ 的时间复杂度得到了 $FWT[A]$.

接下来就是反演了，其实反演是很简单的，既然知道了 A_0 的本身的子集是他自己（$A_0 = FWT[A_0]$），A_1 的子集是 $FWT[A_0] + FWT[A_1]$，那就很简单的得出反演的递推式了：

$$UFWT[A'] = merge(UFWT[A_{(0)'}], UFWT[A_{(1)'}] - UFWT[A_{(0)'}])$$

下面我们给出代码实现. 容易发现顺变换和逆变换可以合并为一个函数，顺变换时 type $= 1$，逆变换时 type $= -1$.

实现

```
void Or(ll *a, ll type) {  // 迭代实现，常数更小
  for (ll x = 2; x <= n; x <<= 1) {
    ll k = x >> 1;
    for (ll i = 0; i < n; i += x) {
      for (ll j = 0; j < k; j++) {
        (a[i + j + k] += a[i + j] * type) %= P;
      }
    }
  }
}
```

与运算

与运算类比或运算可以得到类似结论

$$FWT[A] = merge(FWT[A_0] + FWT[A_1], FWT[A_1])$$

$$UFWT[A'] = merge(UFWT[A_{(0)'}] - UFWT[A_{(1)'}], UFWT[A_{(1)'}])$$

下面我们给出代码实现. 顺变换时 type $= 1$，逆变换时 type $= -1$.

实现

```
void And(ll *a, ll type) {
  for (ll x = 2; x <= n; x <<= 1) {
    ll k = x >> 1;
    for (ll i = 0; i < n; i += x) {
      for (ll j = 0; j < k; j++) {
        (a[i + j] += a[i + j + k] * type) %= P;
      }
    }
  }
}
```

异或运算

异或的卷积是基于如下原理：

若我们令 $x \circ y$ 表示 $x \cap y$ 中 1 数量的奇偶性，即 $x \circ y = \text{popcnt}(x \cap y) \bmod 2$，那么容易有 $(x \circ y) \oplus (x \circ z) = x \circ (y \oplus z)$.

对于 $FWT[A]$ 的运算其实也很好得到.

设 $FWT[A]_i=\sum_{i\circ j=0}A_j-\sum_{i\circ j=1}A_j$. 我们来证一下 $FWT[C]=FWT[A]\cdot FWT[B]$ 的正确性:

$$\begin{aligned}FWT[A]_iFWT[B]_i&=\left(\sum_{i\circ j=0}A_j-\sum_{i\circ j=1}A_j\right)\left(\sum_{i\circ k=0}B_k-\sum_{i\circ k=1}B_k\right)\\&=\left(\sum_{i\circ j=0}A_j\sum_{i\circ k=0}B_k+\sum_{i\circ j=1}A_j\sum_{i\circ k=1}B_k\right)-\left(\sum_{i\circ j=0}A_j\sum_{i\circ k=1}B_k+\sum_{i\circ j=1}A_j\sum_{i\circ k=0}B_k\right)\\&=\sum_{(j\oplus k)\circ i=0}A_jB_k-\sum_{(j\oplus k)\circ i=1}A_jB_k\\&=FWT[C]_i\end{aligned}$$

来看看怎么快速计算 A,B 的值，依旧是分治:

对于 i 在当前位为 0 的子数列 $FWT[A_0]$，进行 $\circ$ 运算时发现它和 0 计算或和 1 计算结果都不会变（因为 $0\cap 0=0,0\cap 1=0$），所以 $FWT[A]=\sum_{i\circ j=0}A_j-\sum_{i\circ j=1}A_j$ 中的 $\sum_{i\circ j=1}A_j=0$.

对于 i 在当前位为 1 的子数列 A_1，进行 $\circ$ 运算时发现它和 0 计算结果是 0，和 1 计算结果是 1（因为 $1\cap 0=0,1\cap 1=1$）.

综上，有:

$$FWT[A]=merge((FWT[A_0]+FWT[A_1])-0,FWT[A_0]-FWT[A_1])$$

也就是:

$$FWT[A]=merge(FWT[A_0]+FWT[A_1],FWT[A_0]-FWT[A_1])$$

逆变换易得:

$$UFWT[A']=merge(\frac{UFWT[A_{(0)'}]+UFWT[A_{(1)'}]}{2},\frac{UFWT[A_{(0)'}]-UFWT[A_{(1)'}]}{2})$$

给出代码，顺变换时 type $=1$，逆变换时 type $=\frac{1}{2}$.

实现

```
void Xor(ll *a, ll type) {
  for (ll x = 2; x <= n; x <<= 1) {
    ll k = x >> 1;
    for (ll i = 0; i < n; i += x) {
      for (ll j = 0; j < k; j++) {
        (a[i + j] += a[i + j + k]) %= P;
        (a[i + j + k] = a[i + j] - a[i + j + k] * 2) %= P;
        (a[i + j] *= type) %= P;
        (a[i + j + k] *= type) %= P;
      }
    }
  }
}
```

同或运算

类比异或运算给出公式:

$FWT[A]_i=\sum_{C_1}A_j-\sum_{C_2}A_j$（$C_1$ 表示 $\text{popcnt}(x\cup y)\bmod 2$ 为 0，C_2 表示 $\text{popcnt}(x\cup y)\bmod 2$ 为 1）

$$FWT[A]=merge(FWT[A_1]-FWT[A_0],FWT[A_1]+FWT[A_0])$$

$$UFWT[A']=merge(\frac{UFWT[A_{(1)'}]-UFWT[A_{(0)'}]}{2},\frac{UFWT[A_{(1)'}]+UFWT[A_{(0)'}]}{2})$$

另一个角度的 FWT

我们设 $c(i,j)$ 是 A_j 对 $FWT[A]_i$ 的贡献系数．我们可以重新描述 FWT 变换的过程：

$$FWT[A]_i=\sum_{j=0}^{n-1}c(i,j)A_j$$

因为有：

$$FWT[A]_i\cdot FWT[B]_i=FWT[C]_i$$

所以我们可以通过简单的证明得到：$c(i,j)c(i,k)=c(i,j\odot k)$．其中 $\odot$ 是任意一种位运算．

同时，c 函数还有一个重要的性质，它可以按位处理．

举个例子，我们变换的时候：

$$FWT[A]_i=\sum_{j=0}^{n-1}c(i,j)A_j$$

这么做是比较劣的，我们将其拆分：

$$FWT[A]_i=\sum_{j=0}^{n/2-1}c(i,j)A_j+\sum_{j=n/2}^{n-1}c(i,j)A_j$$

考虑前面的式子和后面的式子 i,j 的区别，发现只有最高位不同．

所以我们将 i,j 去除最高位的值为 i',j'，并记 i_0 为 i 的最高位．有：

$$FWT[A]_i=c(i_0,0)\sum_{j=0}^{n/2-1}c(i',j')A_j+c(i_0,1)\sum_{j=n/2}^{n-1}c(i',j')A_j$$

如果 $i_0=0$，则有：

$$FWT[A]_i=c(0,0)\sum_{j=0}^{n/2-1}c(i',j')A_j+c(0,1)\sum_{j=n/2}^{n-1}c(i',j')A_j$$

$i_0=1$ 则有：

$$FWT[A]_i=c(1,0)\sum_{j=0}^{n/2-1}c(i',j')A_j+c(1,1)\sum_{j=n/2}^{n-1}c(i',j')A_j$$

也就是说，我们只需要：

$$\begin{bmatrix}c(0,0) & c(0,1)\\ c(1,0) & c(1,1)\end{bmatrix}$$

四个数就可以完成变换了．我们称这个矩阵为位矩阵．

如果我们要进行逆变换，则需要上面的位矩阵的逆矩阵．

若逆矩阵为 c^{-1}，可以通过类似操作得到原数：

$$A_i=\sum_{j=0}^{n}c^{-1}(i,j)FWT[A]_j$$

逆矩阵不一定存在，比如如果有一排 0 或者一列 0 那么这个矩阵就没有逆，我们在构造时需要格外小心．

按位或

我们可以构造：

$$\begin{bmatrix}1 & 0\\ 1 & 1\end{bmatrix}$$

这样满足 $c(i,j)c(i,k)=c(i,j\cup k)$. 我们发现，这和我们前面推出的 $FWT[A]=\text{merge}(FWT[A_0],FWT[A_0]+FWT[A_1])$ 一模一样！同理，下面也是一个满足这个条件的矩阵，但我们一般使用上面这个：

$$\begin{bmatrix}1&1\\1&0\end{bmatrix}$$

虽然下面这个矩阵也满足 $c(i,j)c(i,k)=c(i,j\cup k)$，但这个矩阵存在一排 0，不存在逆，所以不合法：

$$\begin{bmatrix}0&0\\1&1\end{bmatrix}$$

如果我们要进行逆变换，则需要对矩阵求逆，以 **最上面** 这个矩阵为例，得：

$$\begin{bmatrix}1&0\\-1&1\end{bmatrix}$$

然后按照顺变换的方法，把逆变换矩阵代入即可.

按位与

我们可以构造：

$$\begin{bmatrix}1&1\\0&1\end{bmatrix}$$

这样满足 $c(i,j)c(i,k)=c(i,j\cap k)$.

逆矩阵：

$$\begin{bmatrix}1&-1\\0&1\end{bmatrix}$$

按位异或

我们可以构造：

$$\begin{bmatrix}1&1\\1&-1\end{bmatrix}$$

这样满足 $c(i,j)c(i,k)=c(i,j\oplus k)$.

逆矩阵：

$$\begin{bmatrix}0.5&0.5\\0.5&-0.5\end{bmatrix}$$

FWT 是线性变换

FWT 是线性变换. 也就是说，它满足：

$$FWT[A+B]=FWT[A]+FWT[B]$$

以及：

$$FWT[c\cdot A]=c\cdot FWT[A]$$

K 维 FWT

其实位运算的本质是对一个 n 维 $\{0,1\}$ 向量的运算. 或运算就是每一维取 max. 且运算就是每一维取 min. 异或运算则是每一维对应相加再 $\bmod 2$.

位运算有个特点：向量的每一位都是独立的.

我们把 $\{0,1\}$ 扩展到 $[0,K)\cap\mathbf{Z}$ 也就是扩展到 K 进制，看看会得到什么?

max 运算

我们将 $\cup$ 运算拓展到 K 进制，定义 $i\cup j$ 表示按位取 max，有：

$$c(i,j)c(i,k)=c(i,j\cup k)$$

若 $j=k$，那么上式又是：

$$c(i,j)c(i,j)=c(i,j)$$

也就是说，每一行的 1 必定只能在 0 的前面，如果在后面则不合法了．手玩一下可以发现一组合法构造：

$$\begin{bmatrix}1&0&0&0\\1&1&0&0\\1&1&1&0\\1&1&1&1\end{bmatrix}$$

求逆可得：

$$\begin{bmatrix}1&0&0&0\\-1&1&0&0\\0&-1&1&0\\0&0&-1&1\end{bmatrix}$$

min 运算

我们将 $\cap$ 运算拓展到 K 进制，定义 $i\cap j$ 表示按位取 min，有：

$$c(i,j)c(i,k)=c(i,j\cap k)$$

若 $j=k$，那么上式又是：

$$c(i,j)c(i,j)=c(i,j)$$

也就是说，每一行的 1 必定只能在 0 的后面，如果在前面则不合法了．手玩一下可以发现一组合法构造：

$$\begin{bmatrix}1&1&1&1\\0&1&1&1\\0&0&1&1\\0&0&0&1\end{bmatrix}$$

求逆可得：

$$\begin{bmatrix}1&-1&0&0\\0&1&-1&0\\0&0&1&-1\\0&0&0&1\end{bmatrix}$$

前两者用得比较少，用得比较多的是：

不进位加法

我们将 $\oplus$ 运算拓展到 K 进制，定义 $i\oplus j$ 表示按位相加再 $\bmod K$，有：

$$c(i,j)c(i,k)=c(i,j\oplus k)$$

我们构造 $c(i,j)=\omega_K^j$，就可以满足要求了：

$$\omega_K^j\omega_K^k=\omega_K^{j\oplus k}$$

但是每一行都一样矩阵也没有逆，所以我们可以构造 $c(i,j)=\omega_K^{(i-1)j}$ 即可．

有下面这个矩阵：

$$\begin{bmatrix}1&1&1&\cdots&1\\1&\omega_K^1&\omega_K^2&\cdots&\omega_K^{k-1}\\1&\omega_K^2&\omega_K^4&\cdots&\omega_K^{2(k-1)}\\1&\omega_K^3&\omega_K^6&\cdots&\omega_K^{3(k-1)}\\\vdots&\vdots&\vdots&\ddots&\vdots\\1&\omega_K^{k-1}&\omega_K^{2(k-1)}&\cdots&\omega_K^{(k-1)(k-1)}\end{bmatrix}$$

此即为 范德蒙德矩阵[2]，求逆可得：

$$\frac{1}{K}\begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_K^{-1} & \omega_K^{-2} & \cdots & \omega_K^{-(k-1)} \\ 1 & \omega_K^{-2} & \omega_K^{-4} & \cdots & \omega_K^{-2(k-1)} \\ 1 & \omega_K^{-3} & \omega_K^{-6} & \cdots & \omega_K^{-3(k-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_K^{-(k-1)} & \omega_K^{-2(k-1)} & \cdots & \omega_K^{-(k-1)(k-1)} \end{bmatrix}$$

如果我们题目给出的模数是存在单位根的，我们就可以简单实现.

但是 **单位根在模意义下可能不存在**，所以我们考虑扩域，就是人为地定义一个 x，满足 $x^K=1$，然后直接把 x 代入计算，这样每个数都是一个关于 x 的 $k-1$ 次多项式. 我们只需要在 $\bmod x^K-1$ 下计算即可. 那么矩阵可以这么表示：

$$\begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & x^1 & x^2 & \cdots & x^{k-1} \\ 1 & x^2 & x^4 & \cdots & x^{2(k-1)} \\ 1 & x^3 & x^6 & \cdots & x^{3(k-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x^{k-1} & x^{2(k-1)} & \cdots & x^{(k-1)(k-1)} \end{bmatrix}$$

但是这么做可能会存在零因子，也就是 **一个数有多种表示方法**，我们无法确定一个数的真实值.

我们考虑不 $\bmod x^K-1$ 了，我们 $\bmod$ 分圆多项式 $\Phi_K(x)$，他满足 x 的阶为 k，且在 $\mathbb{Q}$ 上不可约. 所以我们定义上面的计算是在 $\bmod \Phi_K(x)$ 下进行即可.

还有一个问题是，$\bmod \Phi_K(x)$ 常数大（因为 Φ 本身就是一个多项式）. 但是因为 $\Phi_K(x) \mid x^k-1$，我们只需要在计算时 $\bmod x^k-1$，最后再 $\bmod \Phi_K(x)$ 即可.

例题

「CF 1103E」Radix sum[3]

给定一个长度为 n 的序列 $a_1,a_2,...,a_n$，对于每一个 $p\in[0,n-1]$，求满足下列条件的整数序列 $i_1,i_2,...,i_n$ 的方案数，对 2^{58} 取模：

- $\forall j\in[1,n], i_j\in[1,n]$；
- $\sum\limits_{j=1}^{n} a_{i_j}=p$，这里的加法定义为十进制不进位加法.

$n\le 10^5, a_i\le 10^5$

题解

我们可以想到 dp：设计状态 $f_{i,s}$ 表示考虑到第 i 个数，当前加法状态为 s. 因为 FWT 变换是线性的，可以先变换为 FWT 点值表示法，然后变成自己的 n 次幂，最后再变换回来.

上面是平凡的，但是题目给出了模数 2^{58}. 发现没有单位根，所以考虑扩域.

这里的分圆多项式 $\Phi_{10}(x)=x^4-x^3+x^2-x+1$.

然而我们发现 UFWT 时，需要除去进制 10，然而我们发现 10 在 2^{58} 下没有逆元. 实际上我们发现 5 在 2^{58} 下是有逆元的：576460752303423 49，我们只需要再除去一个 2 就可以了. 设已经除以了 5 的答案为 x，真正的答案为 y，也就是 $2^5y\equiv x \pmod{2^{64}}$，显然，我们有 $y\equiv \frac{x}{2^5} \pmod{2^{64-5}}$，也就是 $y\equiv \frac{x}{2^5} \pmod{2^{59}}$，所以直接将最后的答案除以 2^5 即可. 虽然出题人不知道为什么要模 2^{58}，但再取下模即可.

【CF103329F】【XXII Opencup, Grand Prix of XiAn】The Struggle[4]

给出一个椭圆 E，其中所有整点的坐标均在 $[1,4\cdot 10^6]$ 之间．求 $\sum_{(x,y)\in E}(x\oplus y)^{33}x^{-2}y^{-1} \bmod 10^9+7$ 的值．

题解

这是一道比较不裸的题，出题人提供了详细的英文题解，具体请见 此链接[5]．

参考资料

- 桃酱的算法笔记[6]
- ZnPdCo 的博客[7]

[1] 维基百科

[2] 范德蒙德矩阵

[3] 「CF 1103E」Radix sum

[4] 【CF103329F】【XXII Opencup, Grand Prix of XiAn】The Struggle

[5] 此链接

[6] 桃酱的算法笔记

[7] ZnPdCo 的博客

9.12.6 Chirp Z 变换

与离散傅里叶变换类似，Chirp Z 变换是给出多项式 $f(x)=\sum_{i=0}^{m-1}f_ix^i\in\mathbb{C}[x]$ 和 $q\in\mathbb{C}\setminus\{0\}$ 求出 $f(1),f(q),\ldots,f(q^{n-1})$ 的一种算法，不要求 q 为单位根．也可用于数论变换．后文将介绍 Chirp Z 变换与其逆变换．

Chirp Z 变换

根据定义，Chirp Z 变换可以写作

$$\mathsf{CZT}_n:(f(x),q)\mapsto[f(1)\ \ f(q)\ \ \cdots\ \ f(q^{n-1})]$$

其中 $f(x):=\sum_{i=0}^{m-1}f_ix^i\in\mathbb{C}[x]$ 且 $q\in\mathbb{C}\setminus\{0\}$．

Bluestein 算法

考虑

$$ij=\binom{i}{2}+\binom{-j}{2}-\binom{i-j}{2}$$

其中 $i,j\in\mathbb{Z}$，我们可以构造

$$G(x):=\sum_{i=-(m-1)}^{n-1}q^{-\binom{i}{2}}x^i,$$

$$F(x):=\sum_{i=0}^{m-1}f_iq^{\binom{-i}{2}}x^i.$$

其中 $G(x)\in\mathbb{C}[x,x^{-1}]$，且对于 $i=0,\ldots,n-1$ 我们有

$$\begin{aligned}[x^i](G(x)F(x))&=\sum_{j=0}^{m-1}(([x^{i-j}]G(x))([x^j]F(x)))\\&=\sum_{j=0}^{m-1}f_jq^{\binom{-j}{2}-\binom{i-j}{2}}\\&=q^{-\binom{i}{2}}f(q^i)\end{aligned}$$

且 $q^{\binom{i+1}{2}} = q^{\binom{i}{2}} \cdot q^i$，$\binom{-i}{2} = \binom{i+1}{2}$. 可以由一次多项式乘法完成求算，该算法被称为 Bluestein 算法.

模板（P6800【模板】Chirp Z-Transform[1]）

```
std::vector<uint> CZT(const std::vector<uint>& f, uint q, int n) {
  if (f.empty() || n == 0) return std::vector<uint>(n);
  const int m = f.size();
  if (q == 0) {
    std::vector<uint> res(n, f[0]);
    for (int i = 1; i < m; ++i)
      if ((res[0] += f[i]) >= MOD) res[0] -= MOD;
    return res;
  }
  // H[i] = q^(binom(i + 1, 2))
  std::vector<uint> H(std::max(m, n - 1));
  H[0] = 1;     // H[0] = q^0
  uint qi = 1;  // qi = q^i
  for (int i = 1; i < (int)H.size(); ++i) {
    qi = (ull)qi * q % MOD;
    // H[i] = q^(binom(i, 2)) * q^i
    H[i] = (ull)H[i - 1] * qi % MOD;
  }
  // F[i] = f[i] * q^(binom(i + 1, 2))
  std::vector<uint> F(m);
  for (int i = 0; i < m; ++i) F[i] = (ull)f[i] * H[i] % MOD;
  std::vector<uint> G_p(m + n - 1);
  // G[i] = q^(-binom(i, 2)), -(m - 1) ≤ i < n
  uint* const G = G_p.data() + (m - 1);
  const uint iq = InvMod(q);
  // G[-(m - 1)] = q^(-binom(-(m - 1), 2)),
  // binom(-(m - 1), 2) = (-(m - 1)) * (-(m - 1) - 1) / 2
  //                    = (m - 1) * m / 2
  G[-(m - 1)] = PowMod(iq, (ull)(m - 1) * m / 2);
  uint qmi = PowMod(q, m - 1);  // q^(-i), -(m - 1) ≤ i < n
  for (int i = -(m - 1) + 1; i < n; ++i) {
    // q^(-binom(i, 2)) = q^(-binom(i - 1, 2)) * q^(-(i - 1))
    G[i] = (ull)G[i - 1] * qmi % MOD;
    // q^(-i) = q^(-(i - 1)) * q^(-1)
    qmi = (ull)qmi * iq % MOD;
  }
  // res[i] = q^(-binom(i, 2)) * f(q^i), 0 ≤ i < n
  std::vector<uint> res = MiddleProduct(G_p, F);
  for (int i = 1; i < n; ++i) res[i] = (ull)res[i] * H[i - 1] % MOD;
  return res;
}

```

逆 Chirp Z 变换

逆 Chirp Z 变换可以写作

$$\mathrm{ICZT}_n : \left([f(1)\ \ f(q)\ \ \cdots\ \ f(q^{n-1})], q\right) \mapsto f(x)$$

其中 $f(x) \in \mathbb{C}[x]_{<n}$ 且 $q \in \mathbb{C} \setminus \{0\}$，并且 $q^i \neq q^j$ 对于所有 $i \neq j$ 成立，这是多项式插值的条件.

Bostan–Schost 算法

回顾 Lagrange 插值公式→9.19.1@p1930 为

$$f(x)=\sum_{i=0}^{n-1}\left(f(x_i)\prod_{0\le j<n\neq i}\frac{x-x_j}{x_i-x_j}\right)$$

且 $x_i\neq x_j$ 对于所有 $i\neq j$ 成立. 与 多项式的快速插值→9.12.8@p1558 中相同, 我们令 $M(x):=\prod_{i=0}^{n-1}(x-x_i)$，根据洛必达法则，有

$$M'(x_i)=\lim_{x\to x_i}\frac{M(x)}{x-x_i}=\prod_{0\le j<n\neq i}(x_i-x_j)$$

修正 Lagrange 插值公式 就是

$$f(x)=M(x)\left(\sum_{i=0}^{n-1}\frac{f(x_i)/M'(x_i)}{x-x_i}\right)$$

那么现在我们有

$$f(x)=M(x)\left(\sum_{i=0}^{n-1}\frac{f(q^i)/M'(q^i)}{x-q^i}\right)$$

其中 $M(x)=\prod_{j=0}^{n-1}(x-q^j)$. 若我们设 n 为偶数，令 $n=2k$ 和 $H(x):=\prod_{j=0}^{k-1}(x-q^j)$，那么

$$M(x)=H(x)\cdot q^{k^2}\cdot H\left(\frac{x}{q^k}\right)$$

这使得我们可以快速计算 $M(x)$. 然后用 Bluestein 算法来计算 $M'(1),\dots,M'(q^{n-1})$. 令 $c_i:=f(q^i)/M'(q^i)$，我们有

$$f(x)=M(x)\left(\sum_{i=0}^{n-1}\frac{c_i}{x-q^i}\right)$$

因为 $\deg f(x)<n$，我们只需计算 $\sum_{i=0}^{n-1}\frac{c_i}{x-q^i}\bmod x^n$，其中 $\frac{c_i}{x-q^i}\in\mathbb{C}[[x]]$，也就是

$$\begin{aligned}\sum_{i=0}^{n-1}\frac{c_i}{x-q^i}\bmod x^n&=-\sum_{i=0}^{n-1}\left(\sum_{j=0}^{n-1}c_iq^{-i(j+1)}x^j\right)\\&=-\sum_{j=0}^{n-1}C(q^{-j-1})x^j\end{aligned}$$

其中 $C(x)=\sum_{i=0}^{n-1}c_ix^i$. 我们可以用 Bluestein 算法来计算 $C(q^{-1}),\dots,C(q^{-n})$.

简单来说，我们分别进行下面的计算:

1. 用减治法（decrease and conquer）计算 $M(x)$;
2. 用 Bluestein 算法计算 $M'(1),\dots,M'(q^{n-1})$;
3. 用 Bluestein 算法计算 $C(q^{-1}),\dots,C(q^{-n})$;
4. 用一次多项式乘法计算 $f(x)$.

其中每一步的时间复杂度都等于两个次数小于等于 n 的多项式相乘的时间复杂度.

模板实现

```cpp
std::vector<uint> InvCZT(const std::vector<uint>& f, uint q) {
  if (f.empty()) return {};
  const int n = f.size();
  if (q == 0) {
    // f[0] = f(1), f[1] = f(0)
    assert(n <= 2);
    if (n == 1) return {f[0]};                  // deg(f) < 1
```

```
    return {f[1], (f[0] + MOD - f[1]) % MOD};  // deg(f) < 2
  }
  // prod[0 ≤ i < n] (x - q^i)
  const auto DaC = [q](auto&& DaC, int n) -> std::vector<uint> {
    if (n == 1) return {MOD - 1, 1u};
    // H = prod[0 ≤ i < ⌊n/2⌋] (x - q^i)
    const std::vector<uint> H = DaC(DaC, n / 2);
    // HH = H(x / q^(⌊n/2⌋)) * q^(⌊n/2⌋^2)
    std::vector<uint> HH = H;
    const uint iqn = InvMod(PowMod(q, n / 2));
    uint qq = PowMod(q, (ull)(n / 2) * (n / 2));
    for (int i = 0; i <= n / 2; ++i)
      HH[i] = (ull)HH[i] * qq % MOD, qq = (ull)qq * iqn % MOD;
    std::vector<uint> res = Product(H, HH);
    if (n & 1) {
      res.resize(n + 1);
      const uint qnm1 = MOD - PowMod(q, n - 1);
      for (int i = n; i > 0; --i) {
        if ((res[i] += res[i - 1]) >= MOD) res[i] -= MOD;
        res[i - 1] = (ull)res[i - 1] * qnm1 % MOD;
      }
    }
    return res;
  };
  const std::vector<uint> M = DaC(DaC, n);
  // C[i] = (M'(q^i))^(-1)
  std::vector<uint> C = InvModBatch(CZT(Deriv(M), q, n));
  // C[i] = f(q^i) / M'(q^i)
  for (int i = 0; i < n; ++i) C[i] = (ull)f[i] * C[i] % MOD;
  // C(x) ← C(q^(-1)x)
  const uint iq = InvMod(q);
  uint qmi = 1;
  for (int i = 0; i < n; ++i)
    C[i] = (ull)C[i] * qmi % MOD, qmi = (ull)qmi * iq % MOD;
  C = CZT(C, iq, n);
  for (int i = 0; i < n; ++i)
    if (C[i] != 0) C[i] = MOD - C[i];
  std::vector<uint> res = Product(M, C);
  res.resize(n);
  return res;
}

```

参考文献

1. Bostan, A. (2010). Fast algorithms for polynomials and matrices. JNCF 2010. Algorithms Project, INRIA.[2]

外部参考资料

[1] P6800【模板】Chirp Z-Transform 

[2] Bostan, A. (2010). Fast algorithms for polynomials and matrices. JNCF 2010. Algorithms Project, INRIA. 

9.12.7 多项式牛顿迭代

描述

给定多项式 $G(x,y)$，已知多项式 $f(x)$ 满足：

$$G(x,f(x))\equiv 0 \pmod{x^n}$$

且存在数值 f_1 使 $G(x,y)$ 满足以下条件：

- $G(0,f_1)=0$；
- $\dfrac{\partial G}{\partial y}(0,f_1)\neq 0$.

求出模 x^n 意义下的 $f(x)$.

Newton's Method

考虑倍增.

首先当 $n=1$ 时，$[x^0]G(x,f(x))=0$ 的解需要单独求出，假设中的 f_1 即为一个解.

假设现在已经得到了模 $x^{\lceil\frac{n}{2}\rceil}$ 意义下的解 $f_{\lceil\frac{n}{2}\rceil}(x)$，要求模 x^n 意义下的解 $f(x)=f_n(x)$.

将 $G(x,f(x))$ 对 $f(x)$ 在 $f(x)=f_{\lceil\frac{n}{2}\rceil}(x)$ 处进行泰勒展开，有：

$$\sum_{i=0}^{+\infty}\frac{\frac{\partial^i G}{\partial y^i}\left(x,f_{\lceil\frac{n}{2}\rceil}(x)\right)}{i!}\left(f(x)-f_{\lceil\frac{n}{2}\rceil}(x)\right)^i\equiv 0 \pmod{x^n}$$

因为 $f(x)-f_{\lceil\frac{n}{2}\rceil}(x)$ 的最低非零项次数最低为 $\lceil\frac{n}{2}\rceil$，故有：

$$\forall 2\leqslant i:\left(f(x)-f_{\lceil\frac{n}{2}\rceil}(x)\right)^i\equiv 0 \pmod{x^n}$$

则：

$$\begin{aligned}\sum_{i=0}^{+\infty}\frac{\frac{\partial^i G}{\partial y^i}\left(x,f_{\lceil\frac{n}{2}\rceil}(x)\right)}{i!}\left(f(x)-f_{\lceil\frac{n}{2}\rceil}(x)\right)^i&\equiv G\left(x,f_{\lceil\frac{n}{2}\rceil}(x)\right)+\frac{\partial G}{\partial y}\left(x,f_{\lceil\frac{n}{2}\rceil}(x)\right)\left[f(x)-f_{\lceil\frac{n}{2}\rceil}(x)\right]\\&\equiv 0 \pmod{x^n}\end{aligned}$$

$$f_n(x)\equiv f_{\lceil\frac{n}{2}\rceil}(x)-\frac{G\left(x,f_{\lceil\frac{n}{2}\rceil}(x)\right)}{\frac{\partial G}{\partial y}\left(x,f_{\lceil\frac{n}{2}\rceil}(x)\right)} \pmod{x^n}$$

或者

$$f_{2n}(x)\equiv f_n(x)-\frac{G(x,f_n(x))}{\frac{\partial G}{\partial y}(x,f_n(x))} \pmod{x^{2n}}$$

例题

多项式求逆→9.12.9@p1559

设给定函数为 $h(x)$，有：

$$G(x,y)=\frac{1}{y}-h(x)$$

应用 Newton's Method 可得：

$$\begin{aligned}f_{2n}(x)&\equiv f_n(x)-\frac{1/f_n(x)-h(x)}{-1/f_n^2(x)} \pmod{x^{2n}}\\&\equiv 2f_n(x)-f_n^2(x)h(x) \pmod{x^{2n}}\end{aligned}$$

时间复杂度

$$T(n)=T\left(\frac{n}{2}\right)+O(n\log n)=O(n\log n)$$

多项式开方→9.12.9@p1559

设给定函数为 $h(x)$，有：

$$G(x,y)=y^2-h(x)\equiv 0$$

应用 Newton's Method 可得：

$$\begin{aligned} f_{2n}(x) &\equiv f_n(x)-\frac{f_n^2(x)-h(x)}{2f_n(x)} \pmod{x^{2n}} \\ &\equiv \frac{f_n^2(x)+h(x)}{2f_n(x)} \pmod{x^{2n}} \end{aligned}$$

时间复杂度

$$T(n)=T\left(\frac{n}{2}\right)+O(n\log n)=O(n\log n)$$

多项式指数函数→9.12.9@p1559

设给定函数为 $h(x)$，有：

$$G(x,y)=\ln y-h(x)$$

应用 Newton's Method 可得：

$$\begin{aligned} f_{2n}(x) &\equiv f_n(x)-\frac{\ln f_n(x)-h(x)}{1/f_n(x)} \pmod{x^{2n}} \\ &\equiv f_n(x)(1-\ln f_n(x)+h(x)) \pmod{x^{2n}} \end{aligned}$$

时间复杂度

$$T(n)=T\left(\frac{n}{2}\right)+O(n\log n)=O(n\log n)$$

手算演示

为了方便理解，这里举几个例子演示一下算法流程.

复数多项式模多项式的平方根

假设 h 是一个不被 x 整除（有常数项）的复数多项式，求它对模 x^n 的平方根.

我们有方程：

$$G(f(x))=f^2(x)-h(x)\equiv 0 \pmod{x^n}$$

Taylor 展开 G 得到下式. 注意这里是对 f 的展开，所以导数都是对 f 的偏导数，x 在这里是当成常数算的.

$$G(f(x))=\sum_{i=0}^{+\infty}\frac{G^{(i)}(f_0(x))}{i!}(f(x)-f_0(x))^i=G(f_0(x))+2f_0(x)(f(x)-f_0(x))+(f(x)-f_0(x))^2$$

用倍增计算. 假设倍增中的中间结果是 $f_0(x),f_1(x),\ldots,f_j(x)$，或者数学严谨地说 $f_j(x)$ 是满足 $G(f_j(x))\equiv 0 \pmod{x^{2^j}}$ 的一个复数多项式，且为了唯一性它同时满足以下两个条件：

- $f_j(x)$ 次数不超过 x^{2^j}；
- $f_{j+k}(x)-f_j(x)\equiv 0 \pmod{x^{2^j}}$，对所有 k.

把 $f_{j+1}(x)$ 和 $f_j(x)$ 代入上面的式子就有：

$$G(f_{j+1}(x))=G(f_j(x))+2f_j(f_{j+1}(x)-f_j(x))+(f_{j+1}(x)-f_j(x))^2\equiv 0 \pmod{x^{2^{j+1}}}$$

显然 $f_{j+1}(x)-f_j(x)$ 必然是 x^{2^j} 的倍数. 于是得到

$$f_{j+1}(x)\equiv f_j(x)-\frac{f_j^2(x)-h(x)}{2f_j(x)}\equiv\frac{f_j(x)^2+h(x)}{2f_j(x)} \pmod{x^{2^{j+1}}}$$

如果 $f_j(x)$ 存在，那么 $2f_j(x)$ 不被 x 整除（有常数项），所以必然有模 $x^{2^{j+1}}$ 的逆元. 因此数列 $f_0, f_1 ..., f_j$ 存在当且仅当 f_0 存在．不被 x 整除的复数多项式 $h(x)$ 模 x 的平方根是一定存在的，因为 $h(x)$ 模掉 x 就是个普通非零复数，一定有两个平方根．所以可以对所有有常数项的 $h(x)$ 用这个算法.

选 $h(x)=x+1$ 举例计算如下：

- $f_0(x)=1, f_1(x)=\dfrac{1^2+x+1}{2\times 1} \bmod x^2=\dfrac{1}{2}x+1, f_2(x)=\dfrac{\left(\frac{1}{2}x+1\right)^2+x+1}{2\times\left(\frac{1}{2}x+1\right)} \bmod x^4=\dfrac{1}{16}x^3-\dfrac{1}{8}x^2+\dfrac{1}{2}x+1, ...$
- $f_0(x)=-1, f_1(x)=\dfrac{(-1)^2+x+1}{2\times(-1)} \bmod x^2=-\dfrac{1}{2}x-1, ...$（等于前一个取负）

可以验证两个都是正确的模平方根多项式列.

整数模素数幂的平方根

牛顿迭代算法还可以迁移到整数模素数的幂的情况．假设 h 是一个不被 3 整除的「方便的」整数.（「方便」指「必然有解」，具体条件后文再言）假设要算 h 在模 3^n 意义下的平方根 f．有方程：

$$G(f)=f^2-h\equiv 0 \pmod{3^n}$$

Taylor 展开 G 得到：

$$G(f)=\sum_{i=0}^{+\infty}\frac{G^{(i)}(f_0)}{i!}(f-f_0)^i=G(f_0)+2f_0(f-f_0)+(f-f_0)^2$$

用倍增计算．假设倍增中得到的中间结果是 $f_0, f_1, ..., f_j$，或者严谨地说 f_j 是满足 $G(f_j)\equiv 0 \pmod{3^{2^j}}$ 的一个整数，且为了唯一性它同时满足以下两个条件：

- $0<f_j<3^{2^j}$；
- $f_{j+k}-f_j\equiv 0 \pmod{3^{2^j}}$，对所有 k.

把 f_{j+1} 和 f_j 代入上面的式子就有：

$$G(f_{j+1})=G(f_j)+2f_j(f_{j+1}-f_j)+(f_{j+1}-f_j)^2\equiv 0 \pmod{3^{2^{j+1}}}$$

显然 $f_{j+1}-f_j$ 必然是 3^{2^j} 的倍数．于是得到

$$f_{j+1}\equiv f_j-\frac{f_j^2-h}{2f_j}\equiv\frac{f_j^2+h}{2f_j} \pmod{3^{2^{j+1}}}$$

如果 f_j 存在，那么 $2f_j$ 不被 3 整除，所以必然有模 $3^{2^{j+1}}$ 的逆元．因此数列 $f_0, f_1 ..., f_j$ 存在当且仅当 f_0 存在．不被 3 整除的整数 h 模 3 的平方根要么不存在，要么有两个．所以 h 有模 3 平方根就是整个算法能跑的唯一条件.

这里选 $h=46$ 实际计算示例.

- $f_0=1, f_1=\dfrac{1^2+46}{2\times 1} \bmod 9=1, f_2=\dfrac{1^2+46}{2\times 1} \bmod 81=64, f_3=\dfrac{64^2+46}{2\times 64} \bmod 6561=955, ...$
- $f_0=2, f_1=\dfrac{2^2+46}{2\times 2} \bmod 9=8, f_2=\dfrac{8^2+46}{2\times 8} \bmod 81=17, f_3=\dfrac{17^2+46}{2\times 17} \bmod 6561=5606, ...$（等于前一个取负）

可以验证一下两个都是正确的模平方根数列.

代数证明

这一节对前文进行引申，用抽象代数的语言证明只要 f 满足初始解条件，牛顿法对所有的 n 都能给出解，并且可以得到全部的解.

有解的证明

引理 1

设 整环→9.16.3@p1830 R 有多项式或 形式幂级数→9.16.3@p1830 $f(X)=\sum_{i\geq 0}a_iX^i$ 和 $r,p\in R$ 使得 $f(r)\in Rp$（亦即 r 是 $f(X)$ 在模 p 意义下的根）且 $f'(r)\in R$ 在模 p 意义下是可逆的. 这里 $f'(X):=\sum_{i\geq 0}(i+1)a_{i+1}X^i$ 是 $f(X)$ 的 **形式导数**. 那么 $f\left(r-\dfrac{f(r)}{f'(r)}\right)\equiv 0\pmod{p^2}$.

证明

对所有 $s\in R$,

$$\begin{aligned}f(r+sp)&=\sum_{i\geq 0}a_i(r+sp)^i\\&=\sum_{i\geq 0}a_ir^i+sp\sum_{i\geq 1}ia_ir^{i-1}+s^2p^2(\dots)\\&=f(r)+spf'(r)+s^2p^2\left(\frac{f''(r)}{2!}+\cdots\right),\end{aligned}$$

所以

$$f(r+sp)\in Rp^2\iff f(r)+f'(r)sp\in Rp^2$$

因为 $f(r)\in Rp$，且 $f'(r)$ 可逆，所以取 $sp=-\dfrac{f(r)}{f'(r)}$ 即可，这里 $\dfrac{1}{f'(r)}$ 是模 p^2 意义下的逆元. 因为 $f'(r)$ 在模 p 意义下可逆，所以它在模 p^2 意义下也必定存在逆元：设有 $a,b,c\in R$ 使 $af'(r)=bp+1$ 和 $f(r)=cp$，那么 $(a^2f'(r)-2)f'(r)=b^2p^2+1$，故可以取 $s=c(2-a^2f'(r))$.

对于域 k 上的多项式环 $k[X]$，设有 $G(X,Y)\in k[X,Y]$ 和 $f_n\in k[X]$ 使 $G(X,f_n(X))\in k[X]X^n$，那么应用引理 1 就可得到

$$G\left(X,f_n(X)-\frac{G(X,f_n(X))}{\frac{\partial G}{\partial Y}(X,f_n(X))}\right)\equiv 0\pmod{X^{2n}}$$

而倍增的初始条件只要有 $f_1\in k$ 使得 $G(X,f_1)\equiv 0\pmod X$ 和 $\dfrac{\partial G}{\partial Y}(X,f_1)\not\equiv 0\pmod X$. 后一个条件保证了 $\dfrac{\partial G}{\partial Y}$ 有非零常数项，同时因为 $X\left|\dfrac{G(X,f_n(X))}{\frac{\partial G}{\partial Y}(X,f_n(X))}\right.$，故而对所有的 n，$\dfrac{\partial G}{\partial Y}(X,f_n)$ 总是模 X^n 意义下可逆的，也就满足了下一次迭代的条件.

得到全部解的证明

引理 2

若 R 为 UFD→9.16.3@p1830，f,r,p 定义同引理 1. 则引理 1 给出的 $r-\dfrac{f(r)}{f'(r)}$ 是模 p^2 意义下唯一满足以下两条件的 x 的值：

- $f(x)\in Rp^2$
- $x-r\in Rp$

亦即

$$\forall x\in R,\qquad p^2\mid f(x)\wedge p\mid(x-r)\Longrightarrow x\equiv r-\frac{f(r)}{f'(r)}\pmod{p^2}$$

证明

令 $s=-\dfrac{f(r)}{f'(r)p}$ 和 $u=r+sp$, 引理 1 保证 u 满足两个条件, 且 $f(r)+f'(r)sp\in Rp^2$. 设 v 是满足上述条件的值, 则有 $v=r+tp$ 和 $f(r)+f'(r)tp\in Rp^2$. 于是有 $f'(r)(t-s)p\in Rp^2$ 和 $v-u\in Rp^2$.

牛顿法可以保证得到模 X^{2^n} 的全部解. 假设 $G(X,h)\equiv 0 \pmod{X^{2^n}}$, 那么令 $h_{2^i}:=h \pmod{X^{2^i}}$, 然后取 $f_1=h_1$ 并用牛顿法, 根据引理 2 可得 $f_{2^i}\equiv h_{2^i} \pmod{X^{2^i}}$, 所以一定有 $f_{2^n}=h$.

上面的论证也说明了, 在 $\dfrac{\partial G}{\partial y}(0,y)$ 永远可逆时, $G(X,f)\equiv 0 \pmod{X^n}$ 的解的个数等于 $G(0,f)\equiv 0 \pmod{X}$ 的解的个数. 这个结论并非平凡. 请看下面的例子.

牛顿法无效时解的个数随次数而变多的例子

模 X 意义下 X^2 的平方根只有 0, 但是模 X^4 意义下 X^2 的平方根有 $X,-X,X^3+X,....$

9.12.8 多项式多点求值|快速插值

多项式的多点求值

描述

给出一个多项式 $f(x)$ 和 n 个点 $x_1,x_2,...,x_n$, 求

$$f(x_1),f(x_2),...,f(x_n)$$

解法

考虑使用分治来将问题规模减半.

将给定的点分为两部分:

$$X_0=\left\{x_1,x_2,...,x_{\lfloor\frac{n}{2}\rfloor}\right\}$$

$$X_1=\left\{x_{\lfloor\frac{n}{2}\rfloor+1},x_{\lfloor\frac{n}{2}\rfloor+2},...,x_n\right\}$$

构造多项式

$$g_0(x)=\prod_{x_i\in X_0}(x-x_i)$$

则有 $\forall x\in X_0:g_0(x)=0$.

考虑将 $f(x)$ 表示为 $g_0(x)Q(x)+f_0(x)$ 的形式, 即:

$$f_0(x)\equiv f(x) \pmod{g_0(x)}$$

则有 $\forall x\in X_0:f(x)=g_0(x)Q(x)+f_0(x)=f_0(x)$, X_1 同理.

至此, 问题的规模被减半, 可以使用分治 + 多项式取模解决.

时间复杂度

$$T(n)=2T\left(\frac{n}{2}\right)+O(n\log n)=O(n\log^2 n)$$

多项式的快速插值

描述

给出一个 $n+1$ 个点的集合

$$X=\{(x_0,y_0),(x_1,y_1),...,(x_n,y_n)\}$$

求一个 n 次多项式 $f(x)$ 使得其满足 $\forall(x,y)\in X:f(x)=y$.

解法

考虑拉格朗日插值公式

$$f(x)=\sum_{i=1}^{n}\prod_{j\neq i}\frac{x-x_j}{x_i-x_j}y_i$$

记多项式 $M(x)=\prod_{i=1}^{n}(x-x_i)$，由洛必达法则可知

$$\prod_{j\neq i}(x_i-x_j)=\lim_{x\to x_i}\frac{\prod_{j=1}^{n}(x-x_j)}{x-x_i}=M'(x_i)$$

因此多项式被表示为

$$f(x)=\sum_{i=1}^{n}\frac{y_i}{M'(x_i)}\prod_{j\neq i}(x-x_j)$$

我们首先通过分治计算出 $M(x)$ 的系数表示，接着可以通过多点求值在 $O(n\log^2 n)$ 时间内计算出所有的 $M'(x_i)$.

我们令 $v_i=\frac{y_i}{M'(x_i)}$，接下来考虑计算出 $f(x)$. 对于 $n=1$ 的情况，有 $f(x)=v_1, M(x)=x-x_1$. 否则令

$$f_0(x)=\sum_{i=1}^{\lfloor\frac{n}{2}\rfloor}v_i\prod_{j\neq i\wedge j\leq\lfloor\frac{n}{2}\rfloor}(x-x_j)$$

$$M_0(x)=\prod_{i=1}^{\lfloor\frac{n}{2}\rfloor}(x-x_i)$$

$$f_1(x)=\sum_{i=\lfloor\frac{n}{2}\rfloor+1}^{n}v_i\prod_{j\neq i\wedge\lfloor\frac{n}{2}\rfloor<j\leq n}(x-x_j)$$

$$M_1(x)=\prod_{i=\lfloor\frac{n}{2}\rfloor+1}^{n}(x-x_i)$$

可得 $f(x)=f_0(x)M_1(x)+f_1(x)M_0(x), M(x)=M_0(x)M_1(x)$，因此可以分治计算，这一部分的复杂度同样是 $O(n\log^2 n)$.

9.12.9 多项式初等函数

> **Authors:** 97littleleaf11, abc1763613206, CCXXXI, EndlessCheng, Enter-tainer, fps5283, Great-designer, H-J-Granger, hly1204, hsfzLZH1, huayucaiji, Ir1d, kenlig, Marcythm, ouuan, SamZhangQingChuan, shuzhouliu, sshwy, StudyingFather, test12345-pupil, Tiphereth-A, TrisolarisHD, untitledunrevised

本页面包含多项式常见的初等函数操作. 具体而言，本页面包含如下内容:

1. 多项式求逆
2. 多项式开方
3. 多项式除法
4. 多项式取模
5. 多项式指数函数
6. 多项式对数函数
7. 多项式三角函数
8. 多项式反三角函数

[3] Elementary function——Wikipedia[1]

初等函数与非初等函数

初等函数的定义如下[3]:

若域 F 中存在映射 $u \to \partial u$ 满足:

1. $\partial(u+v) = \partial u + \partial v$
2. $\partial(uv) = u\partial v + v\partial u$

则称这个域为 **微分域**.

若微分域 F 上的函数 u 满足以下的任意一条条件，则称该函数 u 为初等函数:

1. u 是 F 上的代数函数.
2. u 是 F 上的指数性函数，即存在 $a \in F$ 使得 $\partial u = u\partial a$.
3. u 是 F 上的对数性函数，即存在 $a \in F$ 使得 $\partial u = \frac{\partial a}{a}$.

以下是常见的初等函数:

1. 代数函数: 存在有限次多项式 P 使得 $P(f(x)) = 0$ 的函数 $f(x)$，如 $2x+1, \sqrt{x}, (1+x^2)^{-1}, \mid x \mid$.
2. 指数函数
3. 对数函数
4. 三角函数
5. 反三角函数
6. 双曲函数
7. 反双曲函数
8. 以上函数的复合，如:

$$\frac{\mathrm{e}^{\tan x}}{1+x^2}\sin\left(\sqrt{1+\ln^2 x}\right)$$

$$-\mathrm{i}\ln\left(x+\mathrm{i}\sqrt{1-x^2}\right)$$

以下是常见的非初等函数:

1. 误差函数:

$$\operatorname{erf}(x) := \frac{2}{\sqrt{\pi}}\int_0^x \exp(-t^2)\mathrm{d}t$$

多项式求逆

给定多项式 $f(x)$，求 $f^{-1}(x)$.

解法

倍增法

首先，易知

$$[x^0]f^{-1}(x) = \left([x^0]f(x)\right)^{-1}$$

假设现在已经求出了 $f(x)$ 在模 $x^{\lceil \frac{n}{2} \rceil}$ 意义下的逆元 $f_0^{-1}(x)$. 有:

$$f(x)f_0^{-1}(x) \equiv 1 \pmod{x^{\lceil \frac{n}{2} \rceil}}$$

$$f(x)f^{-1}(x) \equiv 1 \pmod{x^{\lceil \frac{n}{2} \rceil}}$$

$$f^{-1}(x) - f_0^{-1}(x) \equiv 0 \pmod{x^{\lceil \frac{n}{2} \rceil}}$$

两边平方可得:

$$f^{-2}(x) - 2f^{-1}(x)f_0^{-1}(x) + f_0^{-2}(x) \equiv 0 \pmod{x^n}$$

两边同乘 $f(x)$ 并移项可得:

$$f^{-1}(x) \equiv f_0^{-1}(x)(2 - f(x)f_0^{-1}(x)) \pmod{x^n}$$

递归计算即可.

时间复杂度

$$T(n) = T\left(\frac{n}{2}\right) + O(n \log n) = O(n \log n)$$

Newton's Method

参见 Newton's Method→9.12.7@p1554.

Graeffe 法

欲求 $f^{-1}(x) \bmod x^{2n}$ 考虑

$$\begin{aligned} f^{-1}(x) \bmod x^{2n} &= f(-x)(f(x)f(-x))^{-1} \bmod x^{2n} \\ &= f(-x)g^{-1}(x^2) \bmod x^{2n} \end{aligned}$$

只需求出 $g^{-1}(x) \bmod x^n$ 即可还原出 $g^{-1}(x^2) \bmod x^{2n}$ 因为 $f(x)f(-x)$ 是偶函数，时间复杂度同上.

代码

多项式求逆

```
constexpr int MAXN = 262144;
constexpr int mod = 998244353;

using i64 = long long;
using poly_t = int[MAXN];
using poly = int *const;

void polyinv(const poly &h, const int n, poly &f) {
  /* f = 1 / h = f_0 (2 - f_0 h) */
  static poly_t inv_t;
  std::fill(f, f + n + n, 0);
  f[0] = fpow(h[0], mod - 2);
  for (int t = 2; t <= n; t <<= 1) {
    const int t2 = t << 1;
    std::copy(h, h + t, inv_t);
    std::fill(inv_t + t, inv_t + t2, 0);

    DFT(f, t2);
    DFT(inv_t, t2);
    for (int i = 0; i != t2; ++i)
      f[i] = (i64)f[i] * sub(2, (i64)f[i] * inv_t[i] % mod) % mod;
    IDFT(f, t2);

    std::fill(f + t, f + t2, 0);
  }
}
```

例题

1. 有标号简单无向连通图计数：「POJ 1737」Connected Graph[2]

多项式开方

给定多项式 $g(x)$，求 $f(x)$，满足：

$$f^2(x) \equiv g(x) \pmod{x^n}$$

解法

倍增法

首先讨论 $[x^0]g(x)$ 不为 0 的情况.

易知:

$$[x^0]f(x) = \sqrt{[x^0]g(x)}$$

若 $[x^0]g(x)$ 没有平方根，则多项式 $g(x)$ 没有平方根.

> $[x^0]g(x)$ 可能有多个平方根，选取不同的根会求出不同的 $f(x)$.

假设现在已经求出了 $g(x)$ 在模 $x^{\lceil \frac{n}{2} \rceil}$ 意义下的平方根 $f_0(x)$，则有:

$$\begin{aligned}
f_0^2(x) &\equiv g(x) && \left(\bmod\ x^{\lceil \frac{n}{2} \rceil}\right) \\
f_0^2(x) - g(x) &\equiv 0 && \left(\bmod\ x^{\lceil \frac{n}{2} \rceil}\right) \\
\left(f_0^2(x) - g(x)\right)^2 &\equiv 0 && \left(\bmod\ x^{n}\right) \\
\left(f_0^2(x) + g(x)\right)^2 &\equiv 4f_0^2(x)g(x) && \left(\bmod\ x^{n}\right) \\
\left(\frac{f_0^2(x) + g(x)}{2f_0(x)}\right)^2 &\equiv g(x) && \left(\bmod\ x^{n}\right) \\
\frac{f_0^2(x) + g(x)}{2f_0(x)} &\equiv f(x) && \left(\bmod\ x^{n}\right) \\
2^{-1}f_0(x) + 2^{-1}f_0^{-1}(x)g(x) &\equiv f(x) && \left(\bmod\ x^{n}\right)
\end{aligned}$$

倍增计算即可.

时间复杂度

$$T(n) = T\left(\frac{n}{2}\right) + O(n \log n) = O(n \log n)$$

还有一种常数较小的写法就是在倍增维护 $f(x)$ 的时候同时维护 $f^{-1}(x)$ 而不是每次都求逆.

> 当 $[x^0]g(x) \neq 1$ 时，可能需要使用二次剩余来计算 $[x^0]f(x)$.

上述方法需要知道 $f_0(x)$ 的逆，所以常数项不能为 0.

若 $[x^0]g(x) = 0$，则将 $g(x)$ 分解成 $x^k h(x)$，其中 $[x^0]h(x) \neq 0$.

- 若 k 是奇数，则 $g(x)$ 没有平方根.
- 若 k 是偶数，则求出 $h(x)$ 的平方根 $\sqrt{h(x)}$，然后得到 $f(x) \equiv x^{k/2}\sqrt{h(x)} \pmod{x^n}$.

洛谷模板题 P5205【模板】多项式开根[3] 参考代码

```cpp
#include <algorithm>
#include <iostream>
using namespace std;

constexpr int MAXN = 1 << 20, mod = 998244353;

int a[MAXN], b[MAXN], g[MAXN], gg[MAXN];

int qpow(int x, int y) {  // 快速幂
```

```
  int ans = 1;

  while (y) {
    if (y & 1) {
      ans = (long long)1 * ans * x % mod;
    }
    x = (long long)1 * x * x % mod;
    y >>= 1;
  }
  return ans;
}

int inv2 = qpow(2, mod - 2);  // 逆元

void change(int *f, int len) {
  for (int i = 1, j = len / 2; i < len - 1; i++) {
    if (i < j) {
      swap(f[i], f[j]);
    }

    int k = len / 2;
    while (j >= k) {
      j -= k;
      k /= 2;
    }
    if (j < k) {
      j += k;
    }
  }
}

void NTT(int *f, int len, int type) {  // NTT
  change(f, len);

  for (int q = 2; q <= len; q <<= 1) {
    int nxt = qpow(3, (mod - 1) / q);
    for (int i = 0; i < len; i += q) {
      int w = 1;

      for (int k = i; k < i + (q >> 1); k++) {
        int x = f[k];
        int y = (long long)1 * w * f[k + (q / 2)] % mod;

        f[k] = (x + y) % mod;
        f[k + (q / 2)] = (x - y + mod) % mod;
        w = (long long)1 * w * nxt % mod;
      }
    }
  }

  if (type == -1) {
    reverse(f + 1, f + len);
```

```
    int iv = qpow(len, mod - 2);

    for (int i = 0; i < len; i++) {
      f[i] = (long long)1 * f[i] * iv % mod;
    }
  }
}

void inv(int deg, int *f, int *h) {  // 求逆元
  if (deg == 1) {
    h[0] = qpow(f[0], mod - 2);
    return;
  }

  inv((deg + 1) >> 1, f, h);

  int len = 1;
  while (len < deg * 2) {  // 倍增
    len *= 2;
  }

  copy(f, f + deg, gg);
  fill(gg + deg, gg + len, 0);

  NTT(gg, len, 1);
  NTT(h, len, 1);
  for (int i = 0; i < len; i++) {
    h[i] = (long long)1 * (2 - (long long)1 * gg[i] * h[i] % mod + mod) %
mod *
           h[i] % mod;
  }

  NTT(h, len, -1);
  fill(h + deg, h + len, 0);
}

int n, t[MAXN];

// deg:次数
// f:被开根数组
// h:答案数组
void sqrt(int deg, int *f, int *h) {
  if (deg == 1) {
    h[0] = 1;
    return;
  }

  sqrt((deg + 1) >> 1, f, h);

  int len = 1;
  while (len < deg * 2) {  // 倍增
    len *= 2;
```

```
  }
  fill(g, g + len, 0);
  inv(deg, h, g);
  copy(f, f + deg, t);
  fill(t + deg, t + len, 0);
  NTT(t, len, 1);
  NTT(g, len, 1);
  NTT(h, len, 1);

  for (int i = 0; i < len; i++) {
    h[i] = (long long)1 * inv2 *
           ((long long)1 * h[i] % mod + (long long)1 * g[i] * t[i] % mod)
% mod;
  }
  NTT(h, len, -1);
  fill(h + deg, h + len, 0);
}

int main() {
  cin >> n;

  for (int i = 0; i < n; i++) {
    cin >> a[i];
  }
  sqrt(n, a, b);

  for (int i = 0; i < n; i++) {
    cout << b[i] << ' ';
  }

  return 0;
}

```

Newton's Method

参见 Newton's Method→9.12.7@p1554.

例题

1. 「Codeforces Round #250」 E. The Child and Binary Tree[4]

多项式除法 & 取模

给定多项式 $f(x), g(x)$，求 $g(x)$ 除 $f(x)$ 的商 $Q(x)$ 和余数 $R(x)$.

解法

发现若能消除 $R(x)$ 的影响则可直接 多项式求逆→9.12@p1519 解决.

考虑构造变换

$$f^R(x) = x^{\deg f} f\left(\frac{1}{x}\right)$$

观察可知其实质为反转 $f(x)$ 的系数.

设 $n = \deg f, m = \deg g$.

将 $f(x)=Q(x)g(x)+R(x)$ 中的 x 替换成 $\frac{1}{x}$ 并将其两边都乘上 x^n，得到：

$$x^n f\left(\frac{1}{x}\right)=x^{n-m}Q\left(\frac{1}{x}\right)x^m g\left(\frac{1}{x}\right)+x^{n-m+1}x^{m-1}R\left(\frac{1}{x}\right)$$

$$f^R(x)=Q^R(x)g^R(x)+x^{n-m+1}R^R(x)$$

注意到上式中 $R^R(x)$ 的系数为 x^{n-m+1}，则将其放到模 x^{n-m+1} 意义下即可消除 $R^R(x)$ 带来的影响.

又因 $Q^R(x)$ 的次数为 $(n-m)<(n-m+1)$，故 $Q^R(x)$ 不会受到影响.

则：

$$f^R(x)\equiv Q^R(x)g^R(x)\pmod{x^{n-m+1}}$$

使用多项式求逆即可求出 $Q(x)$，将其反代即可得到 $R(x)$.

时间复杂度 $O(n\log n)$.

多项式对数函数 & 指数函数

给定多项式 $f(x)$，求模 x^n 意义下的 $\ln f(x)$ 与 $\exp f(x)$.

解法

普通方法

- 多项式对数函数　　首先，对于多项式 $f(x)$，若 $\ln f(x)$ 存在，则由其 定义→9.12.1@p1519，其必须满足：

  $[x^0]f(x)=1$

  对 $\ln f(x)$ 求导再积分，可得：

$$\frac{\mathrm{d}\ln f(x)}{\mathrm{d}x}\equiv\frac{f'(x)}{f(x)}\pmod{x^n}$$

$$\ln f(x)\equiv\int\mathrm{d}\ln f(x)\equiv\int\frac{f'(x)}{f(x)}\mathrm{d}x\pmod{x^n}$$

  多项式的求导，积分时间复杂度为 $O(n)$，求逆时间复杂度为 $O(n\log n)$，故多项式求 $\ln$ 时间复杂度 $O(n\log n)$.
- 多项式指数函数　　首先，对于多项式 $f(x)$，若 $\exp f(x)$ 存在，则其必须满足：

  $[x^0]f(x)=0$

  否则 $\exp f(x)$ 的常数项不收敛.

  对 $\exp f(x)$ 求导，可得：

$$\frac{\mathrm{d}\exp f(x)}{\mathrm{d}x}\equiv\exp f(x)f'(x)\pmod{x^n}$$

  比较两边系数可得：

$$[x^{n-1}]\frac{\mathrm{d}\exp f(x)}{\mathrm{d}x}=\sum_{i=0}^{n-1}([x^i]\exp f(x))([x^{n-i-1}]f'(x))$$

$$n[x^n]\exp f(x)=\sum_{i=0}^{n-1}([x^i]\exp f(x))((n-i)[x^{n-i}]f(x))$$

  使用分治 FFT 即可解决.

  时间复杂度 $O(n\log^2 n)$.

Newton's Method

使用 Newton's Method→9.12.7@p1554 即可在 $O(n\log n)$ 的时间复杂度内解决多项式 exp.

代码

多项式 ln/exp

```
constexpr int MAXN = 262144;
constexpr int mod = 998244353;

using i64 = long long;
using poly_t = int[MAXN];
using poly = int *const;

void derivative(const poly &h, const int n, poly &f) {
  for (int i = 1; i != n; ++i) f[i - 1] = (i64)h[i] * i % mod;
  f[n - 1] = 0;
}

void integrate(const poly &h, const int n, poly &f) {
  for (int i = n - 1; i; --i) f[i] = (i64)h[i - 1] * inv[i] % mod;
  f[0] = 0; /* C */
}

void polyln(const poly &h, const int n, poly &f) {
  /* f = ln h = ∫ h' / h dx */
  assert(h[0] == 1);
  static poly_t ln_t;
  const int t = n << 1;

  derivative(h, n, ln_t);
  std::fill(ln_t + n, ln_t + t, 0);
  polyinv(h, n, f);

  DFT(ln_t, t);
  DFT(f, t);
  for (int i = 0; i != t; ++i) ln_t[i] = (i64)ln_t[i] * f[i] % mod;
  IDFT(ln_t, t);

  integrate(ln_t, n, f);
}

void polyexp(const poly &h, const int n, poly &f) {
  /* f = exp(h) = f_0 (1 - ln f_0 + h) */
  assert(h[0] == 0);
  static poly_t exp_t;
  std::fill(f, f + n + n, 0);
  f[0] = 1;
  for (int t = 2; t <= n; t <<= 1) {
    const int t2 = t << 1;

    polyln(f, t, exp_t);
    exp_t[0] = sub(pls(h[0], 1), exp_t[0]);
    for (int i = 1; i != t; ++i) exp_t[i] = sub(h[i], exp_t[i]);
    std::fill(exp_t + t, exp_t + t2, 0);

    DFT(f, t2);
```

```
    DFT(exp_t, t2);
    for (int i = 0; i != t2; ++i) f[i] = (i64)f[i] * exp_t[i] % mod;
    IDFT(f, t2);

    std::fill(f + t, f + t2, 0);
  }
}
```

例题

1. 计算 $f^k(x)$

   普通做法为多项式快速幂，时间复杂度 $O(n \log n \log k)$.

   当 $[x^0]f(x) = 1$ 时，有：

   $f^k(x) = \exp(k \ln f(x))$

   当 $[x^0]f(x) \neq 1$ 时，设 $f(x)$ 的最低次项为 $f_i x^i$，则：

   $$f^k(x) = f_i^k x^{ik} \exp\left(k \ln \frac{f(x)}{f_i x^i}\right)$$

   时间复杂度 $O(n \log n)$.

多项式三角函数

给定多项式 $f(x)$，求模 x^n 意义下的 $\sin f(x), \cos f(x)$ 与 $\tan f(x)$.

解法

首先由 Euler's formula→9.10@p1153 $\left(\mathrm{e}^{\mathrm{i}x} = \cos x + \mathrm{i} \sin x\right)$ 可以得到 三角函数的另一个表达式[5]：

$$\sin x = \frac{\mathrm{e}^{\mathrm{i}x} - \mathrm{e}^{-\mathrm{i}x}}{2\mathrm{i}}$$

$$\cos x = \frac{\mathrm{e}^{\mathrm{i}x} + \mathrm{e}^{-\mathrm{i}x}}{2}$$

那么代入 $f(x)$ 就有：

$$\sin f(x) = \frac{\exp(\mathrm{i}f(x)) - \exp(-\mathrm{i}f(x))}{2\mathrm{i}}$$

$$\cos f(x) = \frac{\exp(\mathrm{i}f(x)) + \exp(-\mathrm{i}f(x))}{2}$$

直接按上述表达式编写程序即可得到模 x^n 意义下的 $\sin f(x)$ 与 $\cos f(x)$. 再由 $\tan f(x) = \frac{\sin f(x)}{\cos f(x)}$ 可求得 $\tan f(x)$.

代码

多项式三角函数

注意到我们是在 $\mathbb{Z}_{998244353}$ 上做 NTT，那么相应地，虚数单位 i 应该被换成 86583718 或 911660635：

$$\begin{aligned} & \mathrm{i} = \sqrt{-1} \equiv \sqrt{998244352} \pmod{998244353} \\ \Longrightarrow \quad & \mathrm{i} \equiv 86583718 \pmod{998244353} \\ \text{or} \quad & \mathrm{i} \equiv 911660635 \pmod{998244353} \end{aligned}$$

```
constexpr int MAXN = 262144;
```

```cpp
constexpr int mod = 998244353;
constexpr int imgunit = 86583718; /* sqrt(-1) = sqrt(998233452) */

using i64 = long long;
using poly_t = int[MAXN];
using poly = int *const;

void polytri(const poly &h, const int n, poly &sin_t, poly &cos_t) {
  /* sin(f) = (exp(i * f) - exp(- i * f)) / 2i */
  /* cos(f) = (exp(i * f) + exp(- i * f)) / 2 */
  /* tan(f) = sin(f) / cos(f) */
  assert(h[0] == 0);
  static poly_t tri1_t, tri2_t;

  for (int i = 0; i != n; ++i) tri2_t[i] = (i64)h[i] * imgunit % mod;
  polyexp(tri2_t, n, tri1_t);
  polyinv(tri1_t, n, tri2_t);

  if (sin_t != nullptr) {
    const int invi = fpow(pls(imgunit, imgunit), mod - 2);
    for (int i = 0; i != n; ++i)
      sin_t[i] = (i64)(tri1_t[i] - tri2_t[i] + mod) * invi % mod;
  }
  if (cos_t != nullptr) {
    for (int i = 0; i != n; ++i) cos_t[i] = div2(pls(tri1_t[i], tri2_t[i]));
  }
}
```

多项式反三角函数

给定多项式 $f(x)$，求模 x^n 意义下的 $\arcsin f(x), \arccos f(x)$ 与 $\arctan f(x)$.

解法

仿照求多项式 $\ln$ 的方法，对反三角函数求导再积分可得：

$$\frac{\mathrm{d}}{\mathrm{d}x}\arcsin x = \frac{1}{\sqrt{1-x^2}}$$

$$\arcsin x = \int \frac{1}{\sqrt{1-x^2}}\mathrm{d}x$$

$$\frac{\mathrm{d}}{\mathrm{d}x}\arccos x = -\frac{1}{\sqrt{1-x^2}}$$

$$\arccos x = -\int \frac{1}{\sqrt{1-x^2}}\mathrm{d}x$$

$$\frac{\mathrm{d}}{\mathrm{d}x}\arctan x = \frac{1}{1+x^2}$$

$$\arctan x = \int \frac{1}{1+x^2}\mathrm{d}x$$

那么代入 $f(x)$ 就有：

$$\frac{\mathrm{d}}{\mathrm{d}x}\arcsin f(x)=\frac{f'(x)}{\sqrt{1-f^2(x)}}$$

$$\arcsin f(x)=\int\frac{f'(x)}{\sqrt{1-f^2(x)}}\mathrm{d}x$$

$$\frac{\mathrm{d}}{\mathrm{d}x}\arccos f(x)=-\frac{f'(x)}{\sqrt{1-f^2(x)}}$$

$$\arccos f(x)=-\int\frac{f'(x)}{\sqrt{1-f^2(x)}}\mathrm{d}x$$

$$\frac{\mathrm{d}}{\mathrm{d}x}\arctan f(x)=\frac{f'(x)}{1+f^2(x)}$$

$$\arctan f(x)=\int\frac{f'(x)}{1+f^2(x)}\mathrm{d}x$$

直接按式子求就可以了.

代码

多项式反三角函数

```cpp
constexpr int MAXN = 262144;
constexpr int mod = 998244353;

using i64 = long long;
using poly_t = int[MAXN];
using poly = int *const;

void derivative(const poly &h, const int n, poly &f) {
  for (int i = 1; i != n; ++i) f[i - 1] = (i64)h[i] * i % mod;
  f[n - 1] = 0;
}

void integrate(const poly &h, const int n, poly &f) {
  for (int i = n - 1; i; --i) f[i] = (i64)h[i - 1] * inv[i] % mod;
  f[0] = 0; /* C */
}

void polyarcsin(const poly &h, const int n, poly &f) {
  /* arcsin(f) = ∫ f' / sqrt(1 - f^2) dx  */
  static poly_t arcsin_t;
  const int t = n << 1;
  std::copy(h, h + n, arcsin_t);
  std::fill(arcsin_t + n, arcsin_t + t, 0);

  DFT(arcsin_t, t);
  for (int i = 0; i != t; ++i) arcsin_t[i] = sqr(arcsin_t[i]);
  IDFT(arcsin_t, t);

  arcsin_t[0] = sub(1, arcsin_t[0]);
  for (int i = 1; i != n; ++i)
    arcsin_t[i] = arcsin_t[i] ? mod - arcsin_t[i] : 0;

```

```
  polysqrt(arcsin_t, n, f);
  polyinv(f, n, arcsin_t);
  derivative(h, n, f);

  DFT(f, t);
  DFT(arcsin_t, t);
  for (int i = 0; i != t; ++i) arcsin_t[i] = (i64)f[i] * arcsin_t[i] % mod;
  IDFT(arcsin_t, t);

  integrate(arcsin_t, n, f);
}

void polyarccos(const poly &h, const int n, poly &f) {
  /* arccos(f) = - ∫ f' / sqrt(1 - f^2) dx  */
  polyarcsin(h, n, f);
  for (int i = 0; i != n; ++i) f[i] = f[i] ? mod - f[i] : 0;
}

void polyarctan(const poly &h, const int n, poly &f) {
  /* arctan(f) = ∫ f' / (1 + f^2) dx  */
  static poly_t arctan_t;
  const int t = n << 1;
  std::copy(h, h + n, arctan_t);
  std::fill(arctan_t + n, arctan_t + t, 0);

  DFT(arctan_t, t);
  for (int i = 0; i != t; ++i) arctan_t[i] = sqr(arctan_t[i]);
  IDFT(arctan_t, t);

  inc(arctan_t[0], 1);
  std::fill(arctan_t + n, arctan_t + t, 0);

  polyinv(arctan_t, n, f);
  derivative(h, n, arctan_t);

  DFT(f, t);
  DFT(arctan_t, t);
  for (int i = 0; i != t; ++i) arctan_t[i] = (i64)f[i] * arctan_t[i] % mod;
  IDFT(arctan_t, t);

  integrate(arctan_t, n, f);
}
```

参考资料与链接

[1] Elementary function—— Wikipedia

[2] 「POJ 1737」 Connected Graph

[3] P5205【模板】多项式开根

[4]「Codeforces Round #250」E. The Child and Binary Tree

[5] 三角函数的另一个表达式

9.12.10 常系数齐次线性递推

简介

常系数齐次线性递推数列（又称为 C-finite 或 C-recursive 数列）是常见的一类基础的递推数列.

对于数列 $\left(a_j\right)_{j\geq 0}$ 和其递推式

$$a_n=\sum_{j=1}^{d}c_ja_{n-j},\qquad (n\geq d)$$

其中 c_j 不全为零，我们的目标是在给出初值 $a_0,...,a_{d-1}$ 和递推式中的 $c_1,...,c_d$ 后求出 a_k. 如果 $k\gg d$，我们想要更快速的算法.

这里 $\left(a_j\right)_{j\geq 0}$ 被称为 d 阶的常系数齐次线性递推数列.

Fiduccia 算法

Fiduccia 算法使用多项式取模和快速幂来计算 a_k，时间为 $O(M(d)\log k)$，其中 $O(M(d))$ 表示两个次数为 $O(d)$ 的多项式相乘的时间.

算法：构造多项式 $\Gamma(x):=x^d-\sum_{j=0}^{d-1}c_{d-j}x^j$ 和 $A(x):=\sum_{j=0}^{d-1}a_jx^j$，那么

$$a_k=\left\langle x^k\bmod\Gamma(x),A(x)\right\rangle$$

其中定义 $\left\langle\left(\sum_{j=0}^{n-1}f_jx^j\right),\left(\sum_{j=0}^{n-1}g_jx^j\right)\right\rangle:=\sum_{j=0}^{n-1}f_jg_j$ 为内积.

证明：我们定义 $\Gamma(x)$ 的友矩阵为

$$C_\Gamma:=\begin{bmatrix}&&&c_d\\1&&&c_{d-1}\\&\ddots&&\vdots\\&&1&c_1\end{bmatrix}$$

我们定义多项式 $b(x):=\sum_{j=0}^{d-1}b_jx^j$ 和

$$B_b:=\begin{bmatrix}b_0&b_1&\cdots&b_{d-1}\end{bmatrix}^\top$$

观察到

$$\underbrace{\begin{bmatrix}&&&c_d\\1&&&c_{d-1}\\&\ddots&&\vdots\\&&1&c_1\end{bmatrix}}_{C_\Gamma}\underbrace{\begin{bmatrix}b_0\\b_1\\\vdots\\b_{d-1}\end{bmatrix}}_{B_b}=\underbrace{\begin{bmatrix}c_db_{d-1}\\b_0+c_{d-1}b_{d-1}\\\vdots\\b_{d-2}+c_1b_{d-1}\end{bmatrix}}_{B_{xb\bmod\Gamma}}$$

且

$$\begin{aligned}C_\Gamma&=\begin{bmatrix}B_{x\bmod\Gamma}&B_{x^2\bmod\Gamma}&\cdots&B_{x^d\bmod\Gamma}\end{bmatrix},\\(C_\Gamma)^2&=\begin{bmatrix}B_{x^2\bmod\Gamma}&B_{x^3\bmod\Gamma}&\cdots&B_{x^{d+1}\bmod\Gamma}\end{bmatrix},\\&\cdots\\(C_\Gamma)^k&=\begin{bmatrix}B_{x^k\bmod\Gamma}&B_{x^{k+1}\bmod\Gamma}&\cdots&B_{x^{k+d}\bmod\Gamma}\end{bmatrix}\end{aligned}$$

我们将这个递推用矩阵表示有

$$\begin{bmatrix}a_k\\a_{k+1}\\\vdots\\a_{k+d-1}\end{bmatrix}=\underbrace{\begin{bmatrix}&1&&\\&&\ddots&\\&&&1\\c_d&c_{d-1}&\cdots&c_1\end{bmatrix}^k}_{\left((C_\Gamma)^\top\right)^k=\left((C_\Gamma)^k\right)^\top}\begin{bmatrix}a_0\\a_1\\\vdots\\a_{d-1}\end{bmatrix}$$

可知 $\left((C_\Gamma)^k\right)^\top$ 的第一行为 $B_{x^k\bmod\Gamma}$，根据矩阵乘法的定义得证.

表示为有理函数

对于上述数列 $\left(a_j\right)_{j\geq 0}$ 一定存在有理函数

$$\frac{P(x)}{Q(x)}=\sum_{j\geq 0}a_jx^j$$

且 $Q(x)=x^d\Gamma(x^{-1})$，$\deg P<d$．我们称其为「**有理函数**」是因为 $P(x),Q(x)$ 是「**多项式**」.

证明：对于 $P(x)=\sum_{j=0}^{d-1}p_jx^j$ 和 $Q(x):=\sum_{j=0}^{d}q_jx^j$ 考虑 $\dfrac{P(x)}{Q(x)}=\sum_{j\geq 0}\tilde{q}_jx^j$ 的系数定义，这几乎就是形式幂级数「**除法**」的定义，

$$\tilde{q}_N=\begin{cases}p_0q_0^{-1}, & \text{if } N=0,\\ \left(p_N-\sum_{j=1}^{N}q_j\tilde{q}_{N-j}\right)\cdot q_0^{-1}, & \text{else if } N<d,\\ -q_0^{-1}\sum_{j=1}^{d}q_j\tilde{q}_{N-j}, & \text{otherwise.}\end{cases}$$

我们只需要令

$$P(x)=\left(\left(\sum_{j\geq 0}a_jx^j\right)\cdot x^d\Gamma(x^{-1})\right)\bmod x^d$$

那么根据 $\tilde{q}_N$ 的定义，必然有 $\dfrac{P(x)}{Q(x)}=\sum_{j\geq 0}a_jx^j$.

Bostan–Mori 算法

计算单项

我们的目标仍然是给出上述多项式 $P(x),Q(x)$，求算 $[x^k]\dfrac{P(x)}{Q(x)}$.

Bostan–Mori 算法基于 Graeffe 迭代，对于上述多项式 $P(x),Q(x)$ 有

$$\frac{P(x)}{Q(x)}=\frac{P(x)Q(-x)}{Q(x)Q(-x)}=\frac{U_0(x^2)+xU_1(x^2)}{V(x^2)}$$

因为分母 $V(x^2)$ 是偶函数，所以子问题只需考虑其中的一侧

$$[x^k]\frac{P(x)}{Q(x)}=\left[x^{\lfloor k/2\rfloor}\right]\frac{U_{k\bmod 2}(x)}{V(x)}$$

我们付出两次多项式乘法的代价使得问题至少减少为原先的一半，而当 $k=0$ 时显然有 $[x^0]\dfrac{P(x)}{Q(x)}=\dfrac{P(0)}{Q(0)}$，时间复杂度同上.

计算连续若干项

目标是给出上述多项式 $P(x),Q(x)$，求算 $\left[x^{[L,R)}\right]\dfrac{P(x)}{Q(x)}$．下面的计算中我们只需考虑对答案「**有影响**」的系数，这是 Bostan–Mori 算法的关键.

我们不妨假设 $\deg P<\deg Q$，否则我们也可以通过一次带余除法使问题回到这种情况.

我们先考虑更简单的问题：

$$\left[x^{[L,R)}\right]\frac{1}{Q(x)}=\left[x^{[L,R)}\right]\frac{1}{Q(x)Q(-x)}\cdot Q(-x)$$

我们需要求出 $\left[x^{[L-\deg Q,R)}\right]\dfrac{1}{Q(x)Q(-x)}$ 然后作一次乘法并取出 $x^L,\ldots,x^{R-1}$ 的系数．令 $V(x^2)=Q(x)Q(-x)$ 那么我们只需求出

$$\left[x^{\left[\left\lceil\frac{L-\deg Q}{2}\right\rceil,\left\lceil\frac{R}{2}\right\rceil\right)}\right]\frac{1}{V(x)}$$

就可以还原出 $\left[x^{[L-\deg Q,R)}\right]\dfrac{1}{Q(x)Q(-x)}$．进而我们只需求出 $\left[x^{[L-\deg P,R)}\right]\dfrac{1}{Q(x)}$ 再和 $P(x)$ 作一次乘法即可求出 $\left[x^{[L,R)}\right]\dfrac{P(x)}{Q(x)}$.

上面的算法虽然已经可以工作, 但是每一次的递归的时间复杂度与 $R-L$ 相关, 我们希望能至少在递归求算时摆脱 $R-L$，更具体的，我们先考虑求算 $\left[x^{[L,L+\deg Q+1)}\right]\dfrac{1}{Q(x)}$，考虑

$$\left[x^{[L,L+\deg Q+1)}\right]\frac{1}{Q(x)}=\left[x^{[L,L+\deg Q+1)}\right]\frac{1}{Q(x)Q(-x)}\cdot Q(-x)$$

我们需要求出

$$\left[x^{[L-\deg Q,L+\deg Q+1)}\right]\frac{1}{Q(x)Q(-x)}$$

那么对于 $V(x^2)=Q(x)Q(-x)$ 而言，我们只需求出

$$\left[x^{[\lceil (L-\deg Q)/2\rceil,\lceil (L+\deg Q+1)/2\rceil)}\right]\frac{1}{V(x)}$$

这是因为

$$[x^k]\frac{1}{Q(x)Q(-x)}=\begin{cases}[x^{k/2}]\dfrac{1}{V(x)},\text{if } k\equiv 0 \pmod 2,\\ 0, \qquad\qquad \text{otherwise.}\end{cases}$$

我们知道 $L+\deg Q$ 和 $L-\deg Q$ 的奇偶性是一样的，所以

$$\left\lceil\frac{L+\deg Q+1}{2}\right\rceil-\left\lceil\frac{L-\deg Q}{2}\right\rceil=\begin{cases}\deg Q+1,\text{if } L+\deg Q\equiv 0 \pmod 2,\\ \deg Q, \qquad \text{otherwise.}\end{cases}$$

这样我们可以写出伪代码

Algorithm $\mathrm{Slice}-\mathrm{Coefficients}(Q,L)$:
Input:$Q(x)\in\mathbb{C}[x],L\in\mathbb{Z}$.
Output:$\left[x^{[L,L+\deg Q+1)}\right]Q(x)^{-1}$.
if $L\le 1$ **then return** $\left[x^{[L,L+\deg Q+1)}\right]Q(x)^{-1}$
Use other algorithm to compute $Q(x)^{-1}$
$V(x^2)\leftarrow Q(x)Q(-x)$
$k\leftarrow\left\lceil\frac{L-\deg Q}{2}\right\rceil$
$(t_k,...,t_{k+\deg Q})\leftarrow\mathrm{Slice}-\mathrm{Coefficients}(V,k)$
$T(x)\leftarrow x^{(L-\deg Q)\bmod 2}\sum_{j=0}^{\deg Q}t_{j+k}x^{2j}$
return $\left[x^{[\deg Q,2\deg Q+1)}\right]T(x)Q(-x)$

但是只有这个算法还不够，我们需要重新找到一个有理函数并求算更多系数.

找到新的有理函数表示

我们知道 $Q(x)$ 本身和 $Q(x)^{-1}$ 的一部分连续的系数比如 $\left[x^{[L,L+\deg Q)}\right]Q(x)^{-1}$ 和 $L\ge 0$，我们希望求出 $\left[x^{[L+\deg Q,L+2\deg Q)}\right]Q(x)^{-1}$，这等价于我们要求某个 $P(x)$ 且 $\deg P<\deg Q$ 使得 $\dfrac{P(x)}{Q(x)}$ 的前 $\deg Q$ 项与 $\left[x^{[L,L+\deg Q)}\right]Q(x)^{-1}$ 相同. 简单来说：递推关系（有理函数的分母）是不变的，我们所做的只是更换初值（有理函数的分子）.

具体的，考虑

$$\frac{P(x)}{Q(x)}=\sum_{j\ge 0}a_jx^j$$

我们现在希望将递推前进 n 项，那么就是

$$\sum_{j\ge n}a_jx^{j-n}=\frac{P(x)}{Q(x)x^n}-\frac{Q(x)\sum_{j=0}^{n-1}a_jx^j}{Q(x)x^n}$$

我们先用一次 $\mathrm{Slice}-\mathrm{Coefficients}(Q,L-\deg P)$ 计算出 $\left[x^{[L-\deg P,L-\deg P+\deg Q+1)}\right]Q(x)^{-1}$，然后我们扩展合并出 $\left[x^{[L-\deg P,L+\deg Q)}\right]Q(x)^{-1}$，再重新计算一个分子使得

$$\frac{\tilde{P}(x)}{Q(x)}=\sum_{j\ge 0}\left([x^{L+j}]\frac{P(x)}{Q(x)}\right)x^j$$

最后我们使用形式幂级数的除法计算出 $\left[x^{[0,R-L)}\right]\dfrac{\tilde{P}(x)}{Q(x)}$，时间为 $O(M(d)\log L+M(R-L))$.

参考文献

1. Alin Bostan, Ryuhei Mori.A Simple and Fast Algorithm for Computing the N-th Term of a Linearly Recurrent Sequence[1].

外部参考资料

[1] A Simple and Fast Algorithm for Computing the N-th Term of a Linearly Recurrent Sequence



9.12.11 多项式平移|连续点值平移

多项式平移

多项式平移是简单情况的多项式复合变换，给出 $f(x)=\sum_{i=0}^{n}f_ix^i$ 的系数和一个常数 c，求 $f(x+c)$ 的系数，即 $f(x)\mapsto f(x+c)$.

分治法

令

$$f(x)=f_0(x)+x^{\lfloor n/2\rfloor}f_1(x)$$

那么

$$f(x+c)=f_0(x+c)+(x+c)^{\lfloor n/2\rfloor}f_1(x+c)$$

$(x+c)^{\lfloor n/2\rfloor}$ 的系数为二项式系数，那么

$$T(n)=2T(n/2)+O(n\log n)=O(n\log^2 n)$$

其中 $O(n\log n)$ 为多项式乘法的时间.

Taylor 公式法

对 $f(x)$ 在 c 处应用 Taylor 公式，有

$$f(x)=f(c)+\frac{f'(c)}{1!}(x-c)+\frac{f''(c)}{2!}(x-c)^2+\cdots+\frac{f^{(n)}(c)}{n!}(x-c)^n$$

那么

$$f(x+c)=f(c)+\frac{f'(c)}{1!}x+\frac{f''(c)}{2!}x^2+\cdots+\frac{f^{(n)}(c)}{n!}x^n$$

观察到对于 $t\geq 0$ 有

$$\begin{aligned}t![x^t]f(x+c)&=f^{(t)}(c)\\&=\sum_{i=t}^{n}f_i i!\frac{c^{i-t}}{(i-t)!}\\&=\sum_{i=0}^{n-t}f_{i+t}(i+t)!\frac{c^i}{i!}\end{aligned}$$

令

$$A_0(x)=\sum_{i=0}^{n}f_{n-i}(n-i)!x^i$$

$$B_0(x)=\sum_{i=0}^{n}\frac{c^i}{i!}x^i$$

那么

$$\begin{aligned}[x^{n-t}](A_0(x)B_0(x)) &= \sum_{i=0}^{n-t}([x^{n-t-i}]A_0(x))([x^i]B_0(x)) \\ &= \sum_{i=0}^{n-t} f_{i+t}(i+t)!\frac{c^i}{i!} \\ &= t![x^t]f(x+c)\end{aligned}$$

二项式定理法

考虑二项式定理 $(a+b)^n = \sum_{i=0}^{n}\binom{n}{i}a^i b^{n-i}$ 那么

$$\begin{aligned}f(x+c) &= \sum_{i=0}^{n} f_i(x+c)^i \\ &= \sum_{i=0}^{n} f_i\left(\sum_{j=0}^{i}\binom{i}{j}x^j c^{i-j}\right) \\ &= \sum_{i=0}^{n} f_i i!\left(\sum_{j=0}^{i}\frac{x^j}{j!}\frac{c^{i-j}}{(i-j)!}\right) \\ &= \sum_{i=0}^{n}\frac{x^i}{i!}\left(\sum_{j=i}^{n} f_j j!\frac{c^{j-i}}{(j-i)!}\right)\end{aligned}$$

得到的结果与上述方法相同.

连续点值平移

> **例题 LOJ 166 拉格朗日插值 2**[1]
>
> 给出度数小于等于 n 的多项式 f 的连续点值 $f(0), f(1), ..., f(n)$，在模 998244353 意义下计算 $f(c), f(c+1), ..., f(c+n)$，其中 $1 \le n \le 10^5, n < m \le 10^8$.

Lagrange 插值公式法

考虑 Lagrange 插值公式→9.19.1@p1930

$$\begin{aligned}f(x) &= \sum_{0\le i\le n} f(i)\prod_{0\le j\le n\wedge j\neq i}\frac{x-j}{i-j} \\ &= \sum_{0\le i\le n} f(i)\frac{x!}{(x-n-1)!(x-i)}\frac{(-1)^{n-i}}{i!(n-i)!} \\ &= \frac{x!}{(x-n-1)!}\sum_{0\le i\le n}\frac{f(i)}{(x-i)}\frac{(-1)^{n-i}}{i!(n-i)!}\end{aligned}$$

上式虽然是卷积形式但不能保证分母上 $x-i\neq 0$，所以下面仅考虑 $c>n$ 的情况，其他情况（如系数在模素数意义下时须避免 $B_0(x)$ 系数的分母出现零）可以分类讨论解决，令

$$A_0(x) = \sum_{0\le i\le n}\frac{f(i)(-1)^{n-i}}{i!(n-i)!}x^i$$

$$B_0(x) = \sum_{i\ge 0}\frac{1}{c-n+i}x^i$$

那么对于 $t\ge 0$ 有

$$\begin{aligned}[x^{n+t}](A_0(x)B_0(x)) &= \sum_{i=0}^{n+t}([x^i]A_0(x))([x^{n+t-i}]B_0(x)) \\ &= \sum_{i=0}^{n}\frac{f(i)(-1)^{n-i}}{i!(n-i)!}\frac{1}{c+t-i} \\ &= \frac{(c+t-n-1)!}{(c+t)!}f(c+t)\end{aligned}$$

实现中取 $B_0(x)$ 需要的部分截断可求出更多点值，且可利用循环卷积.

对问题稍加修改，假设对于某个 d 给出的点值为 $f(d), f(d+k), ..., f(d+nk)$，我们可以计算 $f(c+d), f(c+d+k), ..., f(c+d+nk)$，视作平移 $g(x)=f(d+kx)$ 的点值 $g(0), g(1), ..., g(n)$ 为 $g(c/k), g(c/k+1), ..., g(c/k+n)$.

Lagrange 插值公式也给出了通过维护一些前后缀积的线性计算单个点值的方法.

应用

同一行第一类无符号 Stirling 数

例题 P5408 第一类斯特林数·行[2]

在模素数 167772161 意义下求 $\begin{bmatrix}n\\0\end{bmatrix}, \begin{bmatrix}n\\1\end{bmatrix}, ..., \begin{bmatrix}n\\n\end{bmatrix}$，其中 $1 \le n < 262144$.

考虑

$$x^{\overline{n}} = \sum_{i=0}^{n}\begin{bmatrix}n\\i\end{bmatrix}x^i, \quad n \ge 0$$

其中 $x^{\overline{n}} = x\cdot(x+1)\cdots(x+n-1)$ 为上升阶乘幂，令 $f_n(x)=x^{\overline{n}}$ 那么

$$f_{2n}(x) = x^{\overline{n}}\cdot(x+n)^{\overline{n}} = f_n(x)f_n(x+n)$$

通过多项式平移可在 $O(n\log n)$ 求出 $f_n(x+n)$，问题被缩小为原先的一半即求出 $f_n(x)$ 的系数，那么

$$T(n) = T(n/2) + O(n\log n) = O(n\log n)$$

模素数意义下阶乘

例题 P5282【模板】快速阶乘算法[3]

求 $n! \bmod p$，其中 p 为素数且 $1 \le n < p \le 2^{31}-1$.

令 $v=\lfloor\sqrt{n}\rfloor$ 和 $g(x)=\prod_{i=1}^{v}(x+i)$ 那么

$$n! \equiv \left(\prod_{i=0}^{v-1}g(iv)\right)\cdot\prod_{i=v^2+1}^{n} i \pmod p$$

其中 $\prod_{i=v^2+1}^{n} i$ 可在 $O(\sqrt{n})$ 时间计算，我们希望可以快速计算上式的前半部分.

多项式多点求值

$g(x)$ 系数的计算可用上述多项式平移算法在 $O(n\log n)$ 时间得到，但多点求值计算 $g(0), g(v), g(2v), ..., g(v^2-v)$ 需要 $O(\sqrt{n}\log^2 n)$ 时间.

连续点值平移

令 $g_d(x)=\prod_{i=1}^{d}(x+i)$，我们可以用 $d+1$ 个点值 $g_d(0), g_d(v), ..., g_d(dv)$ 唯一确定这个次数为 d 的多项式，又

$$g_{2d}(x) = g_d(x)g_d(x+d)$$

所以只需 $2d+1$ 个点值可以唯一确定 $g_{2d}(x)$，那么使用连续点值平移计算 $g_d((d+1)v), g_d((d+2)v), ..., g_d(2dv)$（即平移 $h(x)=g_d(vx)$ 的点值 $h(0), h(1), ..., h(d)$ 为 $h(d+1), h(d+2), ..., h(2d)$）和 $g_d(d), g_d(v+d), ..., g_d(2dv+d)$（即平移 $h(x)=g_d(vx)$ 的点值 $h(0), h(1), ..., h(d)$ 为 $h(d/v), h(d/v+1), h(d/v+2), ..., h(d/v+2d)$）后将这两者的对应点值相乘即得 $g_{2d}(0), g_{2d}(v), ..., g_{2d}(2dv)$.

由 $g_d(0), g_d(v), ..., g_d(dv)$ 计算 $g_{d+1}(0), g_{d+1}(v), ..., g_{d+1}(dv), g_{d+1}((d+1)v)$ 考虑

$$g_{d+1}(x)=(x+d+1)\cdot g_d(x)$$

额外增加的一个点值使用线性时间的算法即可. 那么在开始时维护 $g_1(0)=1, g_1(v)=v+1$ 后使用连续点值平移来倍增地维护这些点值，有

$$T(n)=T(n/2)+O(n\log n)=O(n\log n)$$

而我们只需要约 $\sqrt{n}$ 个点值，所以时间复杂度为 $O(\sqrt{n}\log n)$.

模素数意义下二项式系数前缀和

例题 LOJ 6386 组合数前缀和[4]

求 $\sum_{i=0}^{m}\binom{n}{i} \bmod 998244353$，其中 $0\le m\le n\le 9\times 10^8$.

考虑使用矩阵描述 $n!=n\cdot(n-1)!$ 这一步递推，我们有

$$[n!]=\left(\prod_{i=0}^{n-1}[i+1]\right)[1]$$

类似的可以将二项式系数前缀和的递推描述为

$$\begin{bmatrix}\binom{n}{m+1}\\ \sum_{i=0}^{m}\binom{n}{i}\end{bmatrix}=\begin{bmatrix}(n-m)/(m+1) & 0\\ 1 & 1\end{bmatrix}\begin{bmatrix}\binom{n}{m}\\ \sum_{i=0}^{m-1}\binom{n}{i}\end{bmatrix}$$

注意矩阵乘法的顺序，那么

$$\begin{aligned}\begin{bmatrix}\binom{n}{m+1}\\ \sum_{i=0}^{m}\binom{n}{i}\end{bmatrix}&=\left(\prod_{i=0}^{m}\begin{bmatrix}(n-i)/(i+1) & 0\\ 1 & 1\end{bmatrix}\right)\begin{bmatrix}1\\0\end{bmatrix}\\&=\frac{1}{(m+1)!}\left(\prod_{i=0}^{m}\begin{bmatrix}n-i & 0\\ i+1 & i+1\end{bmatrix}\right)\begin{bmatrix}1\\0\end{bmatrix}\end{aligned}$$

令 $v=\lfloor\sqrt{m}\rfloor$，考虑维护矩阵

$$\begin{aligned}M_d(x)&=\prod_{i=1}^{d}\begin{bmatrix}-x+n+1-i & 0\\ x+i & x+i\end{bmatrix}\\&=\begin{bmatrix}f_d(x) & 0\\ g_d(x) & h_d(x)\end{bmatrix}\end{aligned}$$

的点值 $M_d(0), M_d(v), ..., M_d(dv)$ 即 $f_d(0), f_d(v), ..., f_d(dv)$、$h_d(0), ..., h_d(dv)$ 和 $g_d(0), ..., g_d(dv)$，又

$$\begin{aligned}M_{2d}(x)&=\prod_{i=1}^{2d}\begin{bmatrix}-x+n+1-i & 0\\ x+i & x+i\end{bmatrix}\\&=\left(\prod_{i=1}^{d}\begin{bmatrix}-x-d+n+1-i & 0\\ x+d+i & x+d+i\end{bmatrix}\right)\left(\prod_{i=1}^{d}\begin{bmatrix}-x+n+1-i & 0\\ x+i & x+i\end{bmatrix}\right)\\&=\begin{bmatrix}f_d(x+d) & 0\\ g_d(x+d) & h_d(x+d)\end{bmatrix}\begin{bmatrix}f_d(x) & 0\\ g_d(x) & h_d(x)\end{bmatrix}\\&=\begin{bmatrix}f_d(x+d)f_d(x) & 0\\ g_d(x+d)f_d(x)+h_d(x+d)g_d(x) & h_d(x+d)h_d(x)\end{bmatrix}\end{aligned}$$

且矩阵右下角元素恰为我们在阶乘算法中所维护的，那么

$$\prod_{i=0}^{m}\begin{bmatrix}n-i & 0\\ i+1 & i+1\end{bmatrix}=\left(\prod_{i=(k+1)v}^{m}\begin{bmatrix}n-i & 0\\ i+1 & i+1\end{bmatrix}\right)\begin{bmatrix}f_v(kv) & 0\\ g_v(kv) & h_v(kv)\end{bmatrix}\cdots\begin{bmatrix}f_v(0) & 0\\ g_v(0) & h_v(0)\end{bmatrix}$$

可在 $O(\sqrt{m}\log m)$ 时间完成计算.

模素数意义下调和数

例题 P5702 调和级数求和[5]

求 $\sum_{i=1}^{n} i^{-1} \bmod p$，其中 p 为素数且 $1 \le n < p < 2^{30}$.

记 $H_n=\sum_{k=1}^{n}k^{-1}$，一步递推为

$$\begin{bmatrix}(n+1)!\\ (n+1)!H_{n+1}\end{bmatrix}=\begin{bmatrix}n+1 & 0\\ 1 & n+1\end{bmatrix}\begin{bmatrix}n!\\ n!H_n\end{bmatrix}$$

那么

$$\begin{bmatrix}\begin{bmatrix}n+1\\ 1\end{bmatrix}\\ \begin{bmatrix}n+1\\ 2\end{bmatrix}\end{bmatrix}=\begin{bmatrix}n!\\ n!H_n\end{bmatrix}=\left(\prod_{i=0}^{n-1}\begin{bmatrix}i+1 & 0\\ 1 & i+1\end{bmatrix}\right)\begin{bmatrix}1\\ 0\end{bmatrix}$$

在这里 $\begin{bmatrix}n+1\\ 1\end{bmatrix}$ 和 $\begin{bmatrix}n+1\\ 2\end{bmatrix}$ 为第一类无符号 Stirling 数. 维护点值矩阵的方法同上.

整式递推

对于更一般的情况，类似于上述快速阶乘算法的案例，我们期望得到一个怎么样的算法?

例题 P6115【模板】整式递推[6]

现有数列 a 满足 $\forall n \ge m, \sum_{k=0}^{m} a_{n-k}P_k(n)=0$，其中 P_k 为不超过 d 次的多项式.

给定所有 P_k 的系数，和 $a_0, a_1, \ldots, a_{m-1}$，求 a_n. 对 998244353 取模. $n \le 6\times 10^8$，$1 \le m, d \le 7$，时限 $7s$.

为了更系统地描述上述几道例题中构造矩阵的过程，我们引入 λ 矩阵→9.14.12@p1769 的概念.

为了实现整式递推，我们应当注意到快速阶乘算法过程中，我们维护的点值其实并不是 $n!$，而是 $\prod_{i=0}^{T-1}(aT+i)$，即 **一对点值之间的倍数关系**.

由于整式递推阶数 m 不止是 1 了，我们就 **不能直接维护一对数之间的倍数关系了**；而是维护出 **一对 m 维向量之间的线性变换**，即 $m\times m$ 的一个矩阵，**矩阵的每一项对应于某个多项式的一个点值**.

容易发现，对于一般的整式递推远处系数求值问题，我们可以构造

$$-\frac{1}{P_0(n)}\begin{bmatrix}P_1(n) & P_2(n) & P_3(n) & \cdots & P_{m-1}(n) & P_m(n)\\ -P_0(n) & & & & & \\ & -P_0(n) & & & & \\ & & -P_0(n) & & & \\ & & & \ddots & & \\ & & & & -P_0(n) & \end{bmatrix}\begin{bmatrix}a_{n-1}\\ a_{n-2}\\ a_{n-3}\\ \vdots\\ a_{n-m+1}\\ a_{n-m}\end{bmatrix}=\begin{bmatrix}a_n\\ a_{n-1}\\ a_{n-2}\\ \vdots\\ a_{n-m+2}\\ a_{n-m+1}\end{bmatrix}$$

设

$$B(\lambda)=\begin{bmatrix}P_1(\lambda) & P_2(\lambda) & P_3(\lambda) & \cdots & P_{m-1}(\lambda) & P_m(\lambda)\\ -P_0(\lambda) & & & & & \\ & -P_0(\lambda) & & & & \\ & & -P_0(\lambda) & & & \\ & & & \ddots & & \\ & & & & -P_0(\lambda) & \end{bmatrix}$$

我们先撇开前面的 $-\frac{1}{P_0(n)}$ 因子不论，我们现在要维护 $\prod_{i=0}^{T-1} B(aT+m+i)$ 这种形式的量，其中乘法自右往左.

容易发现 $B_T(\lambda)=\prod_{i=0}^{T-1} B(\lambda+i)$ 是一个各项次数不高于 dT 的 λ 矩阵，只用 $dT+1$ 个值即可维护.

于是我们维护出 $B_T(m)$，$B_T(m+T)$，$B_T(m+2T)$，...，$B_T(m+(dT-1)T)$，$B_T(m+dT^2)$ 这几个 λ 矩阵的 **点值**，然后用类似于快速阶乘算法的方式暴力进行多项式点值平移和倍增就好了.

具体地，为了让 $t=\log_2 T$ 抬高 1，我们这么干:

1. 在 $O(m^2 dT\log(dT))$ 时间内获取 $B_T(p+dT^2)$，$B_T(p+(dT+1)T)$，$B_T(p+(dT+2)T)$，$\cdots$，$B_T(p+(2dT-1)T)$，$B_T(p+(2dT)dT)$.
2. 在 $O(m^2 dT\log(dT))$ 时间内获取 $B_T(p+2dT^2)$，$B_T(p+(2dT+1)T)$，$B_T(p+(2dT+2)T)$，$\cdots$，$B_T(p+(3dT-1)T)$，$B_T(p+(3dT)dT)$.
3. 在 $O(m^2 dT\log(dT))$ 时间内获取 $B_T(p+3dT^2)$，$B_T(p+(3dT+1)T)$，$B_T(p+(3dT+2)T)$，$\cdots$，$B_T(p+(4dT-1)T)$，$B_T(p+(4dT)dT)$.
4. 计算 $B_{2T}(v)=B_T(v+T)B_T(v)$.

我们每轮花费 $O(m^2 dT\log(dT))$ 的复杂度进行平移；同时，我们每轮只用做 $\Theta(dT)$ 次矩阵乘法，复杂度可以认为是 $O(m^3 dT)$.

最后，我们只用做到 $T\geq\sqrt{n/d}$ 即可.

之前的 $-\frac{1}{P_0(n)}$ 因子可以用类似的方法解决.

这样，我们预处理的复杂度即为 $\Theta(\sqrt{nd}(m^3+m^2\log(nd)))$.

考虑查询，我们只用 $\Theta(n/T)$ 次向量与矩阵的乘法，以及 $O(T)$ 次暴力转移.

容易发现这部分计算并不是复杂度瓶颈.

因此，该算法的总复杂度为 $\Theta(\sqrt{nd}(m^3+m^2\log(nd)))$.

编码时，我们可以使用循环卷积的技巧来减小 NTT 的常数.

在实际应用时，我们往往是对一个已知的微分有限的 GF 提取其远处系数，从而 m,d 均为常数，也即做到了 $\Theta(\sqrt{n}\log n)$ 的远处系数求值.

参考文献

- Alin Bostan, Pierrick Gaudry, and Eric Schost. Linear recurrences with polynomial coefficients and application to integer factorization and Cartier–Manin operator.
- Min_25 的博客
- ZZQ 的博客 - 阶乘模大质数[7]

外部参考资料

[1] LOJ 166 拉格朗日插值 2 [2] P5408 第一类斯特林数·行 [3] P5282【模板】快速阶乘算法

[4] LOJ 6386 组合数前缀和 [5] P5702 调和级数求和 [6] P6115【模板】整式递推

[7] ZZQ 的博客 - 阶乘模大质数

9.12.12 符号化方法

符号化方法（symbolic method）是将组合对象快速转换成生成函数的一种方法，我们将考虑对于集合上定义的特定运算，然后导出其对应的生成函数的运算.

我们称一个组合类（或简称为类）为 $(\mathcal{A}, |\cdot|)$，其中 $\mathcal{A}$ 为组合对象的集合，函数 $|\cdot|$ 将每一个组合对象映射为一个非负整数，一般称为大小函数. 需要注意的是这个非负整数不能是无限大的. 例如对于字符集为 $\{0,1\}$ 的字符串，可以将字符串的长度设置为其大小函数；对于树或图可将节点的数量设置为其大小函数，注意这并非绝对，也可能将某些特定节点的大小函数设置为 0 等.

本文是基于 Analytic Combinatorics 一书第一章的简化.

无标号体系

在无标号体系中将使用普通生成函数（OGF）. 对于集合 $\mathcal{A}$ 其对应 OGF 记为

$$A(z) = \sum_{\alpha \in \mathcal{A}} z^{|\alpha|} = \sum_{n \geq 0} a_n z^n$$

我们约定使用同一组的字母表示同一个类对应的生成函数等，例如用 a_n 表示 $[z^n]A(z)$ 即 $A(z)$ 中 z^n 的系数，用 $\mathcal{A}_n$ 表示 $\mathcal{A}$ 中大小函数为 n 的对象的集合（所以 $a_n = \operatorname{card}(\mathcal{A}_n)$ 其中 card 为基数（cardinality））.

本文将不讨论可容许性（admissibility），读者可参考文献中的内容.

下面将引入两种特殊的组合类和组合对象：

- 记 ϵ 为中性对象（neutral object）和 $\mathcal{E} = \{\epsilon\}$ 为中性类（neutral class），中性对象的大小为 0，中性类的 OGF 为 $E(z) = 1$.
- 记 $\circ$ 或 $\bullet$ 为原子对象（atom object）和 $\mathcal{Z}_\circ = \{\circ\}$ 或 $\mathcal{Z}_\bullet = \{\bullet\}$ 或简写为 $\mathcal{Z}$ 为原子类（atom class），原子对象的大小为 1，原子类的 OGF 为 $Z(z) = z$.

对于两个组合类 $\mathcal{A}$ 和 $\mathcal{B}$ 在组合意义上同构记为 $\mathcal{A} = \mathcal{B}$ 或 $\mathcal{A} \cong \mathcal{B}$，但仅当该同构不平凡时才使用后者的记号.

我们有

$$\mathcal{A} \cong \mathcal{E} \times \mathcal{A} \cong \mathcal{A} \times \mathcal{E}$$

其中 $\times$ 为二元运算，表示集合的笛卡尔积.

集合的（不相交）并构造

对于类 $\mathcal{A}$ 和 $\mathcal{B}$ 的并记为

$$\mathcal{A} + \mathcal{B} = (\mathcal{E}_1 \times \mathcal{A}) + (\mathcal{E}_2 \times \mathcal{B})$$

如此定义可以不违背集合论中集合不相交的要求，我们可以想象成将 $\mathcal{A}$ 中的对象染色成红色，将 $\mathcal{B}$ 中的对象染色成蓝色.

对应 OGF 为

$$A(z) + B(z)$$

考虑

$$A(z) + B(z) = \sum_{\alpha \in \mathcal{A}} z^{|\alpha|} + \sum_{\beta \in \mathcal{B}} z^{|\beta|} = \sum_{n \geq 0} (a_n + b_n) z^n$$

对应形式幂级数的加法.

集合的笛卡尔积构造

对于类 $\mathcal{A}$ 和 $\mathcal{B}$ 的笛卡尔积记为

$$\mathcal{A} \times \mathcal{B} = \{(\alpha, \beta) \mid \alpha \in \mathcal{A}, \beta \in \mathcal{B}\}$$

对应 OGF 为

$$A(z) \cdot B(z)$$

我们定义 (α, β) 的大小为其组成部分的大小之和，那么显然也有

$$\gamma = (\alpha_1, \alpha_2, ..., \alpha_n) \Longrightarrow |\gamma| = |\alpha_1| + |\alpha_2| + \cdots + |\alpha_n|$$

所以

$$A(z)\cdot B(z)=\left(\sum_{\alpha\in\mathcal{A}}z^{|\alpha|}\right)\left(\sum_{\beta\in\mathcal{B}}z^{|\beta|}\right)=\sum_{(\alpha,\beta)\in(\mathcal{A}\times\mathcal{B})}z^{|\alpha|+|\beta|}=\sum_{n\geq 0}\sum_{i+j=n}a_ib_jz^n$$

对应形式幂级数的乘法.

集合的 Sequence 构造

Sequence 构造生成了所有可能的组合.

例

$$\begin{aligned}\mathrm{SEQ}(\{a\})&=\{\epsilon\}+\{a\}+\{(a,a)\}+\{(a,a,a)\}+\cdots\\ \mathrm{SEQ}(\{a,b\})&=\{\epsilon\}+\{a,b\}+\{(a,b)\}+\{(b,a)\}+\{(a,a)\}+\{(b,b)\}\\ &\quad+\{(a,b,a)\}+\{(a,b,b)\}+\{(a,a,b)\}\\ &\quad+\{(b,b,a)\}+\{(b,a,b)\}+\{(b,b,b)\}+\{(a,a,a)\}+\{(b,a,a)\}\\ &\quad+\cdots\end{aligned}$$

可以看到 $\{(a,b)\},\{(b,a)\}$ 这样组成部分的顺序不同的元素被生成了，可以认为 Sequence 构造生成了有序的组合.

我们定义

$$\mathrm{SEQ}(\mathcal{A})=\mathcal{E}+\mathcal{A}+(\mathcal{A}\times\mathcal{A})+(\mathcal{A}\times\mathcal{A}\times\mathcal{A})+\cdots$$

且要求 $\mathcal{A}_0=\emptyset$，也就是 $\mathcal{A}$ 中没有大小为 0 的对象.

对应 OGF 为

$$Q(A(z))=1+A(z)+A(z)^2+A(z)^3+\cdots=\frac{1}{1-A(z)}$$

其中 Q 为 Pólya 准逆（quasi-inversion）.

例：有序有根树（ordered rooted tree）

我们可以使用 Sequence 构造来定义有序有根树，即孩子之间的顺序有意义的有根树，设该组合类为 $\mathcal{T}$ 那么一棵树为一个根节点和树的 Sequence，即

$$\mathcal{T}=\{\bullet\}\times\mathrm{SEQ}(\mathcal{T})$$

对应 OGF 为

$$T(z)=\frac{z}{1-T(z)}$$

前几项系数为 `0 1 1 2 5 14 42 132 429 1430 4862 16796`，忽略常数项即 OEIS A000108[1].

集合的 Multiset 构造

Multiset 构造生成了所有可能的组合，但不区分组成部分的元素之间的顺序.

例

$$\begin{aligned}\mathrm{MSET}(\{a\})&=\{\epsilon\}+\{a\}+\{(a,a)\}+\{(a,a,a)\}+\cdots\\ \mathrm{MSET}(\{a,b\})&=\{\epsilon\}+\{a\}+\{(a,a)\}+\{(a,a,a)\}+\cdots\\ &\quad+\{b\}+\{(a,b)\}+\{(a,a,b)\}+\cdots\\ &\quad+\{(b,b)\}+\{(a,b,b)\}+\{(a,a,b,b)\}+\cdots\\ &\quad+\cdots\end{aligned}$$

注意到 $\{(b,a)\},\{(a,b,a)\}$ 在 $\mathrm{SEQ}(\{a,b\})$ 中出现，但在 $\mathrm{MSET}(\{a,b\})$ 没有出现，可以认为 Multiset 生成了无序的组合.

我们定义其递推式为

$$\mathrm{MSET}(\{\alpha_0,\alpha_1,...,\alpha_n\})=\mathrm{MSET}(\{\alpha_0,\alpha_1,...,\alpha_{n-1}\})\times\mathrm{SEQ}(\{\alpha_n\})$$

即

$$\mathrm{MSET}(\mathcal{A})=\prod_{\alpha\in\mathcal{A}}\mathrm{SEQ}(\{\alpha\})$$

且要求 $\mathcal{A}_0=\emptyset$. 或者也可以给出等价的

$$\mathrm{MSET}(\mathcal{A})=\mathrm{SEQ}(\mathcal{A})/\mathbf{R}$$

其中 $\mathbf{R}$ 为等价关系，我们说 $(\alpha_1,...,\alpha_n)\mathbf{R}(\beta_1,...,\beta_n)$ 当且仅当存在任一置换 σ 对于所有 j 满足 $\beta_j=\alpha_{\sigma(j)}$.

对应 OGF 为

$$\mathrm{Exp}(A(z))=\prod_{\alpha\in\mathcal{A}}\left(1-z^{|\alpha|}\right)^{-1}=\prod_{n\geq1}(1-z^n)^{-a_n}$$

注意到

$$\ln(1+z)=\frac{z}{1}-\frac{z^2}{2}+\frac{z^3}{3}-\cdots=\sum_{n\geq1}\frac{(-1)^{n-1}z^n}{n}$$

且 $A(z)=\exp(\ln(A(z)))$ 所以

$$\begin{aligned}\mathrm{Exp}(A(z))&=\exp\left(\sum_{n\geq1}-a_n\cdot\ln(1-z^n)\right)\\&=\exp\left(\sum_{n\geq1}-a_n\cdot\sum_{m\geq1}\frac{-z^{nm}}{m}\right)\\&=\exp\left(\frac{A(z)}{1}+\frac{A(z^2)}{2}+\frac{A(z^3)}{3}+\cdots\right)\end{aligned}$$

其中 Exp 为 Pólya 指数，也被称为 Euler 变换.

例题 LOJ 6268. 分拆数[2]

题意：令 $f(n)$ 表示将 n 进行分拆的方案数，求 $f(1),f(2),...,f(10^5)$ 对 998244353 取模的值.

解：设全体正整数类为 $\mathcal{I}$，那么 $\mathcal{I}=\mathrm{SEQ}_{\geq1}(\mathcal{Z})=\mathcal{Z}\times\mathrm{SEQ}(\mathcal{Z})$（下标 ≥1 为有限制的构造，见后文）. 所求即

$$\mathrm{MSET}(\mathcal{I})$$

对应 OGF 前几项系数为 `1 2 3 5 7 11 15 22 30 42`（忽略常数项）即 OEIS A000041[3].

例题 洛谷 P4389 付公主的背包[4]

题意：给出 n 种体积分别为 $v_1,...,v_n$ 的商品和正整数 m，求体积为 $1,2,...,m$ 的背包装满的方案数（商品数量不限，有同体积的不同种商品）对 998244353 取模的值. 约定 $1\leq n,m\leq10^5$ 且 $1\leq v_i\leq m$.

解：设商品的组合类为 $\mathcal{A}$，所求即 $\mathrm{MSET}(\mathcal{A})$ 对应 OGF 的系数.

例题 洛谷 P5900 无标号无根树计数[5]

题意：求出 n 个节点的无标号无根树的个数对 998244353 取模的值. 约定 $1\leq n\leq2\times10^5$.

解：设无标号有根树的组合类为 $\mathcal{T}$，那么

$$\mathcal{T}=\{\bullet\}\times\mathrm{MSET}(\mathcal{T})$$

根据 Richard Otter 的论文 The Number of Trees[6] 中的描述，对应无根树的 OGF 为

$$T(z)-\frac{1}{2}T^2(z)+\frac{1}{2}T(z^2)$$

前几项系数为 `1 1 1 2 3 6 11 23 47 106`（忽略常数项）即 OEIS A000055[7].

集合的 Powerset 构造

Powerset 构造生成了所有子集.

例

$$\begin{aligned}\mathrm{PSET}(\{a\})&=\{\epsilon\}+\{a\}\\ \mathrm{PSET}(\{a,b\})&=\{\epsilon\}+\{a\}+\{b\}+\{(a,b)\}\\ \mathrm{PSET}(\{a,b,c\})&=\{\epsilon\}+\{a\}+\{b\}+\{(a,b)\}+\{c\}+\{(a,c)\}+\{(b,c)\}+\{(a,b,c)\}\end{aligned}$$

我们定义其递推式为

$$\mathrm{PSET}(\{\alpha_0,\alpha_1,...,\alpha_n\})=\mathrm{PSET}(\{\alpha_0,\alpha_1,...,\alpha_{n-1}\})\times(\{\epsilon\}+\{\alpha_n\})$$

即

$$\mathrm{PSET}(\mathcal{A})\cong\prod_{\alpha\in\mathcal{A}}(\{\epsilon\}+\{\alpha\})$$

且要求 $\mathcal{A}_0=\emptyset$.

对应 OGF 为

$$\begin{aligned}\overline{\mathrm{Exp}}(A(z))&=\prod_{\alpha\in\mathcal{A}}\left(1+z^{|\alpha|}\right)=\prod_{n\geq1}(1+z^n)^{a_n}\\ &=\exp\left(\sum_{n\geq1}a_n\cdot\ln(1+z^n)\right)\\ &=\exp\left(\sum_{n\geq1}a_n\cdot\sum_{m\geq1}\frac{(-1)^{m-1}z^{nm}}{m}\right)\\ &=\exp\left(\frac{A(z)}{1}-\frac{A(z^2)}{2}+\frac{A(z^3)}{3}-\cdots\right)\end{aligned}$$

其中 $\overline{\mathrm{Exp}}$ 为 Pólya 指数·改.

容易发现 $\mathrm{PSET}(\mathcal{A})\subset\mathrm{MSET}(\mathcal{A})$.

集合的 Cycle 构造

Cycle 构造生成了所有可能的组合，但不区分仅轮换不同的组合.

我们定义为

$$\mathrm{CYC}(\mathcal{A})=(\mathrm{SEQ}(\mathcal{A})\setminus\{\epsilon\})/\mathbf{S}$$

其中 $\mathbf{S}$ 为等价关系，我们说 $(\alpha_1,...,\alpha_n)\mathbf{S}(\beta_1,...,\beta_n)$ 当且仅当存在任一循环移位 τ 对于所有 j 都满足 $\beta_j=\alpha_{\tau(j)}$.

例

为了简便我们令 `a`, `b` 均为大小为 1 的字符，这里仅列举大小为 3 和 4 的字符串:

$$\mathrm{CYC}(\{\mathtt{a},\mathtt{b}\})_3=\{\mathtt{aaa}\}+\{\mathtt{aab}\}+\{\mathtt{abb}\}+\{\mathtt{bbb}\}$$

其中 `aab`$\mathbf{S}$`baa`$\mathbf{S}$`aba` 只保留其一，同样的 `abb`$\mathbf{S}$`bab`$\mathbf{S}$`bba` 只保留其一.

$$\mathrm{CYC}(\{\mathtt{a},\mathtt{b}\})_4=\{\mathtt{aaaa}\}+\{\mathtt{aaab}\}+\{\mathtt{aabb}\}+\{\mathtt{abbb}\}+\{\mathtt{bbbb}\}+\{\mathtt{abab}\}$$

其中 $\mathtt{aaab}\mathbf{S}\mathtt{baaa}\mathbf{S}\mathtt{abaa}\mathbf{S}\mathtt{aaba}$，$\mathtt{aabb}\mathbf{S}\mathtt{baab}\mathbf{S}\mathtt{bbaa}\mathbf{S}\mathtt{abba}$，$\mathtt{abbb}\mathbf{S}\mathtt{babb}\mathbf{S}\mathtt{bbab}\mathbf{S}\mathtt{bbba}$ 和 $\mathtt{abab}\mathbf{S}\mathtt{baba}$.

对应 OGF 为

$$\mathrm{Log}(A(z))=\sum_{n\geq 1}\frac{\varphi(n)}{n}\ln\frac{1}{1-A(z^n)}$$

其中 φ 为 Euler 函数，Log 为 Pólya 对数.

由于证明较复杂，读者可参考 Flajolet 的论文 The Cycle Construction[8] 或 Analytic Combinatorics 的附录.

有限制的构造

对于上述所有构造，我们都没有限制其「组成部分」的个数，若在 SEQ 的下标给一个作用于整数的谓词用于约束其组成部分，如

$$\mathrm{SEQ}_{=k}(\mathcal{B}),\quad \mathrm{SEQ}_{\geq k}(\mathcal{B}),\quad \mathrm{SEQ}_{1..k}(\mathcal{B})$$

其中 $\mathrm{SEQ}_{=k}(\mathcal{B})$ 也常简写为 $\mathrm{SEQ}_k(\mathcal{B})$，$\mathrm{SEQ}_{1..k}(\mathcal{B})$ 表示在区间 $[1..k]$ 上.

令 $\mathfrak{K}$ 为任意上述 SEQ, PSET, MSET, CYC 之一，以及

$$\mathcal{A}=\mathfrak{K}_k(\mathcal{B})$$

即我们需要对于 $\alpha\in\mathcal{A}$ 有

$$\alpha=\{(\beta_1,\beta_2,...,\beta_k)\mid\beta\in\mathcal{B}\}$$

设 χ 函数作用于组合对象上为其组成部分的个数，也就是要令 $\chi(\alpha)=k$，不妨增加一元来「跟踪」组成部分的个数.

令

$$A_{n,k}=\mathrm{card}\{\alpha\in\mathcal{A}\mid|\alpha|=n,\chi(\alpha)=k\}$$

那么

$$A(z,u)=\sum_{n,k}A_{n,k}u^kz^n=\sum_{\alpha\in\mathcal{A}}z^{|\alpha|}u^{\chi(\alpha)}$$

然后我们只要提取出 u^k 的系数即可获得对应表达式，例如 $\mathcal{A}=\mathrm{SEQ}_k(\mathcal{B})$ 可直接导出

$$A(z,u)=\sum_{k\geq 0}u^kB(z)^k=\frac{1}{1-uB(z)}$$

$$\Longrightarrow A(z)=B(z)^k$$

显然也有

$$\mathcal{A}=\mathrm{SEQ}_{\geq k}(\mathcal{B})\Longrightarrow A(z)=\frac{B(z)^k}{1-B(z)}$$

而对于 $\mathrm{MSET}_k(\mathcal{B})$ 和 $\mathrm{PSET}_k(\mathcal{B})$ 已经有

$$A(z,u)=\prod_n(1-uz^n)^{-b_n}$$

$$\Longrightarrow A(z)=[u^k]\exp\left(\frac{u}{1}B(z)+\frac{u^2}{2}B(z^2)+\frac{u^3}{3}B(z^3)+\cdots\right)$$

和

$$A(z,u)=\prod_n(1+uz^n)^{b_n}$$

$$\Longrightarrow A(z)=[u^k]\exp\left(\frac{u}{1}B(z)-\frac{u^2}{2}B(z^2)+\frac{u^3}{3}B(z^3)-\cdots\right)$$

对于 $\mathrm{CYC}_k(\mathcal{B})$ 同理.

使用上式计算 $\mathrm{MSET}_3(\mathcal{B})$ 和 $\mathrm{MSET}_4(\mathcal{B})$ 对应 OGF

尝试计算 $\mathcal{A}=\mathrm{MSET}_3(\mathcal{B})$ 为

$$\begin{aligned}[u^3]A(z,u)=&\frac{1}{0!}([u^3]1)+\frac{1}{1!}\left([u^3]\left(\frac{u}{1}B(z)+\frac{u^2}{2}B(z^2)+\frac{u^3}{3}B(z^3)+\cdots\right)\right)\\&+\frac{1}{2!}\left([u^3]\left(\frac{u}{1}B(z)+\frac{u^2}{2}B(z^2)+\cdots\right)^2\right)\\&+\frac{1}{3!}\left([u^3]\left(\frac{u}{1}B(z)+\cdots\right)^3\right)\\&=\frac{B(z)^3}{6}+\frac{B(z)B(z^2)}{2}+\frac{B(z)^3}{3}\end{aligned}$$

尝试计算 $\mathcal{A}=\mathrm{MSET}_4(\mathcal{B})$ 为

$$\begin{aligned}[u^4]A(z,u)=&\frac{1}{0!}([u^4]1)+\frac{1}{1!}\left([u^4]\left(\frac{u}{1}B(z)+\frac{u^2}{2}B(z^2)+\frac{u^3}{3}B(z^3)+\frac{u^4}{4}B(z^4)+\cdots\right)\right)\\&+\frac{1}{2!}\left([u^4]\left(\frac{u}{1}B(z)+\frac{u^2}{2}B(z^2)+\frac{u^3}{3}B(z^3)+\cdots\right)^2\right)\\&+\frac{1}{3!}\left([u^4]\left(\frac{u}{1}B(z)+\frac{u^2}{2}B(z^2)+\cdots\right)^3\right)\\&+\frac{1}{4!}\left([u^4]\left(\frac{u}{1}B(z)+\cdots\right)^4\right)\\&=\frac{B(z^4)}{4}+\frac{1}{2!}\left(\frac{B(z^2)^2}{4}+\frac{2B(z)B(z^3)}{3}\right)+\frac{1}{3!}\left(\frac{3B(z)^2B(z^2)}{2}\right)+\frac{B(z)^4}{4!}\\&=\frac{B(z)^4}{24}+\frac{B(z)^2B(z^2)}{4}+\frac{B(z)B(z^3)}{3}+\frac{B(z^2)^2}{8}+\frac{B(z^4)}{4}\end{aligned}$$

我们发现 $\mathcal{A}=\mathfrak{K}_k(\mathcal{B})$ 中 $A(z)$ 是关于 $B(z),B(z^2),...,B(z^k)$ 的一个表达式.

需要注意的是对于有限制的构造 $\mathfrak{K}_k(\mathcal{B})$ 并没有要求 $\mathcal{B}_0=\emptyset$.

常用有限制的构造

$$\begin{aligned}\mathrm{PSET}_2(\mathcal{A}):&\quad\frac{A(z)^2}{2}-\frac{A(z^2)}{2}\\\mathrm{MSET}_2(\mathcal{A}):&\quad\frac{A(z)^2}{2}+\frac{A(z^2)}{2}\\\mathrm{CYC}_2(\mathcal{A}):&\quad\frac{A(z)^2}{2}+\frac{A(z^2)}{2}\\\mathrm{PSET}_3(\mathcal{A}):&\quad\frac{A(z)^3}{6}-\frac{A(z)A(z^2)}{2}+\frac{A(z^3)}{3}\\\mathrm{MSET}_3(\mathcal{A}):&\quad\frac{A(z)^3}{6}+\frac{A(z)A(z^2)}{2}+\frac{A(z^3)}{3}\\\mathrm{CYC}_3(\mathcal{A}):&\quad\frac{A(z)^3}{3}+\frac{2A(z^3)}{3}\end{aligned}$$

$$\begin{aligned}
\mathrm{PSET}_4(\mathcal{A}):&\quad \frac{A(z)^4}{24}-\frac{A(z)^2A(z^2)}{4}+\frac{A(z)A(z^3)}{3}+\frac{A(z^2)^2}{8}-\frac{A(z^4)}{4}\\
\mathrm{MSET}_4(\mathcal{A}):&\quad \frac{A(z)^4}{24}+\frac{A(z)^2A(z^2)}{4}+\frac{A(z)A(z^3)}{3}+\frac{A(z^2)^2}{8}+\frac{A(z^4)}{4}\\
\mathrm{CYC}_4(\mathcal{A}):&\quad \frac{A(z)^4}{4}+\frac{A(z^2)^2}{4}+\frac{A(z^4)}{2}
\end{aligned}$$

上面的计算方法虽然有效但比较麻烦，读者可阅读 WolframMathWorld 网站的 Pólya Enumeration Theorem[9] 和 Cycle Index[10] 等相关资料，后者 Cycle Index 在 OEIS 的生成函数表达式中也经常出现.

例题 LOJ 6538. 烷基计数 加强版 加强版[11]

题意：求出 n 个节点的有根且根节点度数不超过 3，其余节点度数不超过 4 的无序树的个数对 998244353 取模的值. 约定 $1\le n\le 10^5$.

解：设组合类为 $\mathcal{T}$ 那么

$$\mathcal{T}=\{\bullet\}\times\mathrm{MSET}_{0,1,2,3}(\mathcal{T})$$

或令组合类 $\hat{\mathcal{T}}=\mathcal{T}+\{\epsilon\}$ 那么

$$\hat{\mathcal{T}}=\{\epsilon\}+\{\bullet\}\times\mathrm{MSET}_3(\hat{\mathcal{T}})$$

可得到相同的结果.

参考文献

- Philippe Flajolet and Robert Sedgewick.Analytic Combinatorics[12].

外部参考资料

[1] A000108

[2] LOJ 6268. 分拆数

[3] A000041

[4] 洛谷 P4389 付公主的背包

[5] 洛谷 P5900 无标号无根树计数

[6] The Number of Trees

[7] A000055

[8] The Cycle Construction

[9] Pólya Enumeration Theorem

[10] Cycle Index

[11] LOJ 6538. 烷基计数 加强版 加强版

[12] Analytic Combinatorics

9.12.13 Lagrange 反演

形式 Laurent 级数

我们已经知道形式幂级数环 $\mathbb{C}[[x]]$ 了，定义形式 Laurent 级数环：

$$\mathbb{C}((x)):=\left\{\sum_{k\ge N}a_kx^k:N\in\mathbb{Z},a_k\in\mathbb{C}\right\}$$

我们可以仿照形式幂级数的乘法逆元定义来定义 $\mathbb{C}((x))$ 上元素的乘法逆元：

若对于 $f:=\sum_{k\ge N}f_kx^k$ 且 $f_N\neq 0$ 存在 $g=\sum_{k\ge -N}g_kx^k$ 满足 $fg=1$ 那么

$$g_k:=\begin{cases}f_N^{-1}, & \text{if } k=-N,\\ -f_N^{-1}\sum_{i>N}f_ig_{k-i}, & \text{otherwise}\end{cases}$$

与形式幂级数类似的，我们也对非零的 $f(x)=\sum_{k\ge N}f_kx^k$ 定义：

$$\operatorname{ord} f := \min\{k : f_k \neq 0\}$$

显然对于 $g \neq 0$ 有

$$\operatorname{ord}(fg) = \operatorname{ord}(f) + \operatorname{ord}(g)$$

形式留数

形式留数是形式 Laurent 级数中 x^{-1} 项的系数. 记 $\operatorname{res} f := [x^{-1}]f$.

引理: 对于任何形式 Laurent 级数 f 有 $\operatorname{res} f' = 0$.

证明: 考虑形式导数的定义 $\left(x^k\right)' = kx^{k-1}$.

引理: 对于任何形式 Laurent 级数 f, g 有 $\operatorname{res}(f'g) = -\operatorname{res}(fg')$.

证明: 考虑乘法法则 $(fg)' = f'g + fg'$ 所以 $0 = \operatorname{res}((fg)') = \operatorname{res}(f'g) + \operatorname{res}(fg')$.

引理: 对于形式 Laurent 级数 $f(x) \neq 0$ 有 $\operatorname{res}(f'/f) = \operatorname{ord} f$.

证明: 设 $\operatorname{ord} f = k$ 那么

$$\begin{aligned}
\operatorname{res}\left(\frac{f'}{f}\right) &= \operatorname{res}\left(\frac{kf_k x^{k-1} + \cdots}{f_k x^k + f_{k+1}x^{k+1} + \cdots}\right) \\
&= \operatorname{res}\left(\frac{kf_k x^{-1} + \cdots}{f_k + f_{k+1}x + \cdots}\right) \\
&= k
\end{aligned}$$

引理: 对于形式 Laurent 级数 f 和形式幂级数 $g \neq 0$ 有 $\operatorname{res}(f)\operatorname{ord}(g) = \operatorname{res}(f(g)g')$.

证明: 考虑线性性, 我们只需证明 $f = x^k$ 其中 $k \in \mathbb{Z}$ 的情况即可, 若 $k \neq -1$ 那么

$$\begin{aligned}
\operatorname{res} x^k &= 0 \\
\operatorname{res}(g^k g') &= \operatorname{res}\left(\frac{1}{k+1}\left(g^{k+1}\right)'\right) \\
&= \frac{1}{k+1}\operatorname{res}\left(\left(g^{k+1}\right)'\right) \\
&= 0
\end{aligned}$$

若 $k = -1$ 那么

$$\begin{aligned}
\operatorname{res} f &= \operatorname{res}\left(x^{-1}\right) = 1 \\
\operatorname{res}(f(g)g') &= \operatorname{res}(g'/g) \\
&= \operatorname{ord}(g) \\
&= \operatorname{res}(f)\operatorname{ord}(g)
\end{aligned}$$

复合逆

记 $A(x) \circ B(x) := A(B(x))$.

命题: $f(x) := \sum_{k\geq 1} f_k x^k$ 存在复合逆 $f^{\langle -1\rangle}(x)$ 当且仅当 $f(0) = 0 \neq f'(0)$, 此时 $f^{\langle -1\rangle}(x)$ 是唯一的. 进一步说: 若 $g(x) = \sum_{k\geq 1} g_k x^k$ 满足 $f(g(x)) = x$ 或 $g(f(x)) = x$ 那么 $g(x) = f^{\langle -1\rangle}(x)$.

证明: 考虑

$$\begin{aligned}
g(f(x)) = {} & g_1(f_1 x + f_2 x^2 + f_3 x^3 + \cdots) \\
& + g_2(f_1 x + f_2 x^2 + \cdots)^2 \\
& + g_3(f_1 x + \cdots)^3 \\
& + \cdots \\
= {} & g_1 f_1 x + (g_1 f_2 + g_2 f_1^2)x^2 + (g_1 f_3 + 2g_2 f_1 f_2 + g_3 f_1^3)x^3 + \cdots
\end{aligned}$$

因为 $g(f(x))=x$ 所以有下面的方程组

$$\begin{cases} g_1 f_1 & =1 \\ g_1 f_2+g_2 f_1^2 & =0 \\ g_1 f_3+2g_2 f_1 f_2+g_3 f_1^3 & =0 \\ \vdots \end{cases}$$

我们只能在 $f_1\neq 0$ 时才能解出第一个等式，然后依次可以解出 $g_2,\dots$.

特别的，考虑 $f(h(x))=x$ 那么 $g(f(h(x)))=g(x)$，进而 $g(x)=g\circ f\circ h(x)=x\circ h(x)=h(x)$.

Lagrange 反演公式

令 $f(x),g(x)\in\mathbb{C}[[x]]$ 满足 $f(g(x))=g(f(x))=x$. 取 $\Phi(x)\in\mathbb{C}[[x]]$（或 $\Phi(x)\in\mathbb{C}((x))$），那么

$$\begin{aligned}[x^n]\Phi(f(x)) &= [x^{n-1}]\Phi(x)\frac{g'(x)}{g(x)}\left(\frac{x}{g(x)}\right)^n \\ &= [x^{-1}]\frac{\Phi(x)g'(x)}{g(x)^{n+1}}\end{aligned}$$

证明:

$$\begin{aligned}[x^n]\Phi(f(x)) &= \operatorname{res}\left(\frac{\Phi(f(x))}{x^{n+1}}\right) \\ &= \operatorname{res}\left(\frac{\Phi(f(g(x)))g'(x)}{g(x)^{n+1}}\right)\cdot(\operatorname{ord}(g(x)))^{-1} \\ &= \operatorname{res}\left(\frac{\Phi(x)g'(x)}{g(x)^{n+1}}\right)\end{aligned}$$

一些读者可能会更加熟悉下面的版本：对于 $k\in\mathbb{Z}_{\geq 0},n\in\mathbb{Z}_{>0}$ 有

$$[x^n]f(x)^k=\frac{k}{n}[x^{n-k}]\left(\frac{x}{g(x)}\right)^n$$

或者

$$\begin{aligned}[x^n]\Phi(f(x)) &= \frac{1}{n}[x^{n-1}]\Phi'(x)\left(\frac{x}{g(x)}\right)^n \\ &= \frac{1}{n}[x^{-1}]\frac{\Phi'(x)}{g(x)^n}\end{aligned}$$

发现

$$\begin{aligned}\operatorname{res}\left(\frac{\Phi'(x)}{g(x)^n}-n\frac{\Phi(x)g'(x)}{g(x)^{n+1}}\right) &= \operatorname{res}\left(\left(\frac{\Phi(x)}{g(x)^n}\right)'\right) \\ &= 0\end{aligned}$$

可以通过我们已经证明的部分导出.

参考文献

1. Richard P. Stanley and Sergey P. Fomin. Enumerative Combinatorics Volume 2 (Edition 1).
2. Ira M. Gessel. Lagrange Inversion.

9.12.14 形式幂级数复合|复合逆

形式幂级数的复合和复合逆也是常见的形式幂级数操作，对于没有特殊性质的 f 之前我们一直使用的多是 $O(n^2)$ 的算法（该算法仍需使用 FFT）来计算 $f(g)\bmod x^n$ 其中 $f\in\mathbb{C}[[x]],g\in x\mathbb{C}[[x]]$，但是因为效率较低应用较少. 我们介绍 Kinoshita–Li 的 $O(M(n)\log n)$ 的算法，其中 $O(M(n))$ 为两个次数为 $O(n)$ 的多项式相乘的时间.

形式幂级数/多项式的复合

若要计算 $f(g(x)) \bmod x^n$ 那么需要 $f(g(x))$ 的每一项系数都是有限项之和，所以之前要求 $f(x) \in \mathbb{C}[[x]], g(x) \in x\mathbb{C}[[x]]$，而如果 $f(x), g(x) \in \mathbb{C}[x]$ 也可以满足这个条件．因为我们需要将 $f(g(x))$ 的系数截断，不妨直接考虑 $f(x), g(x)$ 都是多项式的情况．对于 $f(x) = \sum_{j=0}^{n-1} f_j x^j$，有

$$f(g(x)) = \sum_{j=0}^{n-1} f_j g(x)^j$$

我们考虑环 $\mathbb{C}[x]((y))$ 上的有理函数

$$\frac{f(y^{-1})}{1 - y \cdot g(x)} = \sum_{j \geq 0} (\cdots + f_j y^{-j} + \cdots) g(x)^j y^j$$

$$f(g(x)) = [y^0] \frac{f(y^{-1})}{1 - y \cdot g(x)}$$

根据 常系数齐次线性递推→9.12.10@p1572 中提到的 Bostan–Mori 算法，Kinoshita 和 Li 指出可以将其修改为二元形式：

$$\begin{aligned}
\frac{P(y)}{Q(x,y)} \bmod x^n &= \left(\frac{P(y)}{Q(x,y)Q(-x,y)} \bmod x^n \right) Q(-x,y) \bmod x^n \\
&= \left(\frac{P(y)}{V(x^2,y)} \bmod x^n \right) Q(-x,y) \bmod x^n \\
&= \left(\frac{P(y)}{V(z,y)} \bmod z^{\lceil n/2 \rceil} \right) \Bigg|_{z=x^2} Q(-x,y) \bmod x^n
\end{aligned}$$

这样递归的计算在 $n = 1$ 时我们只需计算

$$\frac{P(y)}{Q(x,y)} \bmod x = \frac{P(y)}{Q(0,y)} \in \mathbb{C}((y))$$

在计算 $\dfrac{P(y)}{V(z,y)} \bmod z^{\lceil n/2 \rceil} \in \mathbb{C}[z]((y))$ 时我们不需要保留所有 y 的系数，因为最后我们只需要提取 y^0 的系数，所以 $y^{>0}$ 的系数是不需要的，而因为求出前者之后需要将其乘以若干个形如 $Q(-x,y) \in \mathbb{C}[x,y]$ 的「**多项式**」，所以只需要保留对于 y^0 有贡献的系数即可．我们准备好给出伪代码：

Algorithm $\mathsf{Comp}(P(y), Q(x,y), n, m)$:
Input: $P = \sum_{0 \leq j < n} p_j y^{-j} \in \mathbb{C}((y)), Q \in \mathbb{C}[x,y], n, m \in \mathbb{N}_{>0}$.
Output: $\left[y^{(-m,0]}\right] \dfrac{P(y)}{Q(x,y)} \bmod x^n$.
Require: $[x^0 y^0] Q = 1$.

if $n = 1$ **then return** $\left([y^{-m+1}] \frac{P(y)}{Q(0,y)}, \ldots, [y^0] \frac{P(y)}{Q(0,y)} \right)$
$V(x^2, y) \leftarrow Q(x,y)Q(-x,y) \bmod x^n$
$d \leftarrow \deg_y Q(-x,y)$
$\left(t_{-(m+d)+1}, \ldots, t_0\right) \leftarrow \mathsf{Comp}(P(y), V(x,y), \lceil n/2 \rceil, m+d)$
$T(x,y) \leftarrow \sum_{j=-(m+d)+1}^{0} t_j y^j$
$U(x,y) = \sum_{j=-(m+d)+1}^{d} u_j y^j \leftarrow T(x^2,y)Q(-x,y) \bmod x^n$
return $\left(u_{-m+1}, \ldots, u_0\right)$

那么我们有

$$f(g(x)) \bmod x^n = \mathsf{Comp}(f(y^{-1}), 1 - y \cdot g(x), \max\{1 + \deg f, n\}, 1) \bmod x^n$$

注意第三个参数是因为 $g(0)$ 可能不为零，如果 $\deg f \geq n$ 此时不能截断 $f(x)$ 来计算 $f(g(x))$，我们也可以选择计算 $f(g) = f \circ (x + g(0)) \circ (g - g(0))$，此时可以取 $F := f(x + g(0)) \bmod x^n$ 和 $G := g - g(0)$ 转而计算 $\mathsf{Comp}(F(y^{-1}), 1 - y \cdot G(x), n, 1)$.

另外因为调用的限制最后递归终止时的 $Q(0,y)^{-1}$ 是可以直接导出的，不需要使用形式幂级数的乘法逆元算法来计算，我们只需计算一次乘法然后提取需要的系数.

常见的特殊形式复合

我们常用的 多项式初等函数→9.12.9@p1559 都可以通过复合计算：

$$g(0)=1,\ g^{-1}=1+(1-g)+(1-g)^2+\cdots$$

$$g(0)=1,\ \log g=-\frac{1-g}{1}-\frac{(1-g)^2}{2}-\frac{(1-g)^3}{3}-\cdots$$

$$g(0)=0,\ \exp g=1+\frac{g}{1!}+\frac{g^2}{2!}+\frac{g^3}{3!}+\cdots$$

$$g(0)=1,\ g^e=1+\frac{e}{1!}(g-1)+\frac{e(e-1)}{2!}(g-1)^2+\cdots$$

在复合逆的计算中我们也会用到求幂函数.

Kronecker 代换

在分析时间复杂度之前我们先考虑如何作二元多项式乘法，一种想法是将系数「打包」，这一方法由 Kronecker 在 1882 年通过 $y\mapsto x^N$ 将 $R[x,y]$ 上的乘法缩减为 $R[x]$ 上的乘法，但是要求 N 足够大.

不妨设 $\deg_x(AB)<N$，那么我们计算 $A(x,x^N)B(x,x^N)$ 之后仍然可以还原出 $A(x,y)B(x,y)$ 且「打包」和「拆包」的时间为线性.

我们使用 Kronecker 代换再计算一元多项式乘法即可，不难发现在 n 为二的幂时上述算法可以在 $O(M(n)\log n)$ 时间完成，因为每一次递归中 y 的次数翻倍，但是 x 的次数减半.

模板（P5373【模板】多项式复合函数[1]）

代码相对于原算法作了一些简化及修改，使得代码更短.

```cpp
#include <cassert>
#include <cstdio>
#include <tuple>
#include <utility>
#include <vector>

using uint = unsigned;
using ull = unsigned long long;
constexpr uint MOD = 998244353;

constexpr uint PowMod(uint a, ull e) {
  for (uint res = 1;; a = (ull)a * a % MOD) {
    if (e & 1) res = (ull)res * a % MOD;
    if ((e /= 2) == 0) return res;
  }
}

constexpr uint InvMod(uint a) { return PowMod(a, MOD - 2); }

constexpr uint QUAD_NONRESIDUE = 3;
constexpr int LOG2_ORD = 23;  // __builtin_ctz(MOD - 1)
constexpr uint ZETA = PowMod(QUAD_NONRESIDUE, (MOD - 1) >> LOG2_ORD);
constexpr uint INV_ZETA = InvMod(ZETA);

// 返回做 n 长 FFT 所需的单位根数组，长度为一半
std::pair<std::vector<uint>, std::vector<uint>> GetFFTRoot(int n) {
  assert((n & (n - 1)) == 0);
  if (n / 2 == 0) return {};
```

```
  std::vector<uint> root(n / 2), inv_root(n / 2);
  root[0] = inv_root[0] = 1;
  for (int i = 0; (1 << i) < n / 2; ++i)
    root[1 << i] = PowMod(ZETA, 1LL << (LOG2_ORD - i - 2)),
                inv_root[1 << i] = PowMod(INV_ZETA, 1LL << (LOG2_ORD - i
- 2));
  for (int i = 1; i < n / 2; ++i)
    root[i] = (ull)root[i - (i & (i - 1))] * root[i & (i - 1)] % MOD,
    inv_root[i] =
        (ull)inv_root[i - (i & (i - 1))] * inv_root[i & (i - 1)] % MOD;
  return {root, inv_root};
}

void Butterfly(int n, uint a[], const uint root[]) {
  assert((n & (n - 1)) == 0);
  for (int i = n; i >= 2; i /= 2)
    for (int j = 0; j < n; j += i)
      for (int k = j; k < j + i / 2; ++k) {
        const uint u = a[k];
        a[k + i / 2] = (ull)a[k + i / 2] * root[j / i] % MOD;
        if ((a[k] += a[k + i / 2]) >= MOD) a[k] -= MOD;
        if ((a[k + i / 2] = u + MOD - a[k + i / 2]) >= MOD) a[k + i / 2] -
= MOD;
      }
}

void InvButterfly(int n, uint a[], const uint root[]) {
  assert((n & (n - 1)) == 0);
  for (int i = 2; i <= n; i *= 2)
    for (int j = 0; j < n; j += i)
      for (int k = j; k < j + i / 2; ++k) {
        const uint u = a[k];
        if ((a[k] += a[k + i / 2]) >= MOD) a[k] -= MOD;
        a[k + i / 2] = (ull)(u + MOD - a[k + i / 2]) * root[j / i] % MOD;
      }
}

void FFT(int n, uint a[], const uint root[]) { Butterfly(n, a, root); }

void InvFFT(int n, uint a[], const uint root[]) {
  InvButterfly(n, a, root);
  const uint inv_n = InvMod(n);
  for (int i = 0; i < n; ++i) a[i] = (ull)a[i] * inv_n % MOD;
}

// 形式幂级数复合，求出 f(g) mod x^n 要求 g(0) = 0
std::vector<uint> FPSComposition(std::vector<uint> f, std::vector<uint> g,
                                 int n) {
  assert(g.empty() || g[0] == 0);
  int len = 1;
  while (len < n) len *= 2;
  std::vector<uint> root, inv_root;
```

```
  std::tie(root, inv_root) = GetFFTRoot(len * 4);
  // [y^(-1)] (f(y) / (-g(x) + y)) mod x^n in R[x]((y^(-1)))
  const auto KinoshitaLi = [&](auto &&KinoshitaLi, const std::vector<uint>
&P,
                               const std::vector<uint> &Q, int d, int n) {
    assert((int)P.size() == d * n);
    assert((int)Q.size() == d * n);
    if (n == 1) return P;
    std::vector<uint> dftQ(d * n * 4);
    for (int i = 0; i < d; ++i)
      for (int j = 0; j < n; ++j) dftQ[i * n * 2 + j] = Q[i * n + j];
    dftQ[d * n * 2] = 1;
    FFT(d * n * 4, dftQ.data(), root.data());
    std::vector<uint> V(d * n * 2);
    for (int i = 0; i < d * n * 4; i += 2)
      V[i / 2] = (ull)dftQ[i] * dftQ[i + 1] % MOD;
    InvFFT(d * n * 2, V.data(), inv_root.data());
    assert(V[0] == 1);
    V[0] = 0;
    for (int i = 1; i < d * 2; ++i)
      for (int j = 0; j < n / 2; ++j) V[i * (n / 2) + j] = V[i * n + j];
    V.resize(d * n);
    const std::vector<uint> T = KinoshitaLi(KinoshitaLi, P, V, d * 2, n /
2);
    std::vector<uint> dftT(d * n * 2);
    for (int i = 0; i < d * 2; ++i)
      for (int j = 0; j < n / 2; ++j) dftT[i * n + j] = T[i * (n / 2) + j];
    FFT(d * n * 2, dftT.data(), root.data());
    for (int i = 0; i < d * n * 4; i += 2) {
      const uint u = dftQ[i];
      dftQ[i] = (ull)dftT[i / 2] * dftQ[i + 1] % MOD;
      dftQ[i + 1] = (ull)dftT[i / 2] * u % MOD;
    }
    InvFFT(d * n * 4, dftQ.data(), inv_root.data());
    for (int i = 0; i < d; ++i)
      for (int j = 0; j < n; ++j) dftQ[i * n + j] = dftQ[(i + d) * (n * 2)
+ j];
    dftQ.resize(d * n);
    return dftQ;
  };
  f.resize(len);
  g.resize(len);
  for (int i = 0; i < len; ++i) g[i] = (g[i] != 0 ? MOD - g[i] : 0);
  std::vector<uint> res = KinoshitaLi(KinoshitaLi, f, g, 1, len);
  res.resize(n);
  return res;
}

std::pair<std::vector<uint>, std::vector<uint>> GetFactorial(int n) {
  if (n == 0) return {};
  std::vector<uint> factorial(n), inv_factorial(n);
  factorial[0] = inv_factorial[0] = 1;
```

```cpp
  if (n == 1) return {factorial, inv_factorial};
  std::vector<uint> inv(n);
  inv[1] = 1;
  for (int i = 2; i < n; ++i)
    inv[i] = (ull)(MOD - MOD / i) * inv[MOD % i] % MOD;
  for (int i = 1; i < n; ++i)
    factorial[i] = (ull)factorial[i - 1] * i % MOD,
    inv_factorial[i] = (ull)inv_factorial[i - 1] * inv[i] % MOD;
  return {factorial, inv_factorial};
}

// f(x) |-> f(x + c)
std::vector<uint> TaylorShift(std::vector<uint> f, uint c) {
  if (f.empty() || c == 0) return f;
  const int n = f.size();
  std::vector<uint> factorial, inv_factorial;
  std::tie(factorial, inv_factorial) = GetFactorial(n);
  for (int i = 0; i < n; ++i) f[i] = (ull)f[i] * factorial[i] % MOD;
  std::vector<uint> g(n);
  uint cp = 1;
  for (int i = 0; i < n; ++i)
    g[i] = (ull)cp * inv_factorial[i] % MOD, cp = (ull)cp * c % MOD;
  int len = 1;
  while (len < n * 2 - 1) len *= 2;
  std::vector<uint> root, inv_root;
  std::tie(root, inv_root) = GetFFTRoot(len);
  f.resize(len);
  g.resize(len);
  FFT(len, f.data(), inv_root.data());
  FFT(len, g.data(), root.data());
  for (int i = 0; i < len; ++i) f[i] = (ull)f[i] * g[i] % MOD;
  InvFFT(len, f.data(), root.data());
  f.resize(n);
  for (int i = 0; i < n; ++i) f[i] = (ull)f[i] * inv_factorial[i] % MOD;
  return f;
}

// 多项式复合，求出 f(g) mod x^n
std::vector<uint> PolyComposition(const std::vector<uint> &f,
                                  const std::vector<uint> &g, int n) {
  if (g.empty() || g[0] == 0) return FPSComposition(f, g, n);
  std::vector<uint> gg = g;
  gg[0] = 0;
  return FPSComposition(TaylorShift(f, g[0]), gg, n);
}

int main() {
  int n, m;
  std::scanf("%d%d", &n, &m);
  std::vector<uint> f(n + 1), g(m + 1);
  for (int i = 0; i <= n; ++i) std::scanf("%u", &f[i]);
  for (int i = 0; i <= m; ++i) std::scanf("%u", &g[i]);
```

```
180   const std::vector<uint> res = PolyComposition(f, g, n + 1);
181   for (int i = 0; i <= n; ++i) std::printf("%u%c", res[i], " \n"[i == n]);
182   return 0;
183 }
184
```

形式幂级数的复合逆

现给出 $f \in x\mathbb{C}[[x]]$ 且 $f'(0) \neq 0$，求出 $g(x) \bmod x^n$ 满足 $f(g) \equiv g(f) \equiv x \pmod{x^n}$.

根据 Lagrange 反演→9.12.13@p1587，对于 $n > 1, k \geq 0$ 我们有

$$[x^{n-1}]f(x)^k = \frac{k}{n-1}[x^{n-1-k}]\left(\frac{g(x)}{x}\right)^{-(n-1)}$$

也就是我们如果能对 $k = 0, 1, \ldots, n-1$ 求出 $[x^{n-1}]f(x)^k$，那么就可以求出其复合逆.

Kinoshita 和 Li 指出我们可以考虑二元有理函数

$$\frac{1}{1 - y \cdot f(x)} = \sum_{j \geq 0} f(x)^j y^j$$

且这个问题有一个更一般的形式即 Power Projection 问题：我们考虑计算

$$u := [x^{n-1}]\frac{P(x,y)}{Q(x,y)} \bmod y^m$$

当 $n - 1 = 0$ 时显然有 $u = \dfrac{P(0,y)}{Q(0,y)} \bmod y^m$，否则我们有

$$\frac{P(x,y)}{Q(x,y)} = \frac{P(x,y)Q(-x,y)}{Q(x,y)Q(-x,y)} = \frac{U_e(x^2,y) + xU_o(x^2,y)}{V(x^2,y)}$$

那么

$$u = \begin{cases} [x^{n-1}]\dfrac{U_e(x^2,y)}{V(x^2,y)} & \text{if } n-1 \text{ is even,} \\ [x^{n-1}]\dfrac{xU_o(x^2,y)}{V(x^2,y)} & \text{if } n-1 \text{ is odd.} \end{cases}$$

$$= \begin{cases} [x^{\lceil n/2 \rceil - 1}]\dfrac{U_e(x,y)}{V(x,y)} & \text{if } n-1 \text{ is even,} \\ [x^{\lceil n/2 \rceil - 1}]\dfrac{U_o(x,y)}{V(x,y)} & \text{if } n-1 \text{ is odd.} \end{cases}$$

我们给出其伪代码：

Algorithm PowProj$(P(x,y),Q(x,y),n,m)$:
Input:$P,Q\in\mathbb{C}[x,y],n,m\in\mathbb{N}_{>0}$.
Output:$[x^{n-1}]\dfrac{P(x,y)}{Q(x,y)}\bmod y^m$.
Require:$[x^0y^0]Q=1$.

while $n>1$ **do**
  $U(x,y)\leftarrow P(x,y)Q(-x,y)\bmod x^n\bmod y^m$
  if $n-1$ is even **then**
    $P(x,y)\leftarrow\sum_{j=0}^{\lceil n/2\rceil+1}([x^{2j}]U(x,y))x^j$
  else
    $P(x,y)\leftarrow\sum_{j=0}^{\lceil n/2\rceil+1}([x^{2j+1}]U(x,y))x^j$
  end if
  $V(x,y)\leftarrow Q(x,y)Q(-x,y)\bmod x^n\bmod y^m$
  $Q(x,y)\leftarrow\sum_{j=0}^{\lceil n/2\rceil+1}([x^{2j}]V(x,y))x^j$
  $n\leftarrow\lceil n/2\rceil$
end while
return $\left(\frac{P(0,y)}{Q(0,y)}\bmod y^m\right)$

同样的我们也可以直接导出 $Q(0,y)^{-1}$ 而不需要计算形式幂级数的乘法逆元，那么复合逆的算法就是

Algorithm Rev$(f(x),n)$:
Input:$f\in x\mathbb{C}[[x]],f'(0)\neq0,n\in\mathbb{N}_{\geq2}$.
Output:$g(x)\bmod x^n$ such that $f(g)\equiv g(f)\equiv x\pmod{x^n}$.

$t\leftarrow f'(0)$
$F(x)\leftarrow f(t^{-1}x)$
$\sum_{k=0}^{n-1}a_ky^k\leftarrow$ PowProj$(1,1-y\cdot F(x),n,n)$
$G(x)\leftarrow\sum_{k=1}^{n-1}\frac{n-1}{k}a_kx^{n-1-k}$
$H(x)\leftarrow\left(G(x)^{1/(n-1)}\right)^{-1}\bmod x^{n-1}$
return $((t^{-1}x)\circ(x\cdot H))$

模板（P5809【模板】多项式复合逆[2]）

代码相对于原算法作了一些简化及修改，使得代码更短.

```cpp
#include <algorithm>
#include <cassert>
#include <cstdio>
#include <tuple>
#include <utility>
#include <vector>

using uint = unsigned;
using ull = unsigned long long;
constexpr uint MOD = 998244353;

constexpr uint PowMod(uint a, ull e) {
  for (uint res = 1;; a = (ull)a * a % MOD) {
    if (e & 1) res = (ull)res * a % MOD;
    if ((e /= 2) == 0) return res;
  }
}

constexpr uint InvMod(uint a) { return PowMod(a, MOD - 2); }

constexpr uint QUAD_NONRESIDUE = 3;
constexpr int LOG2_ORD = 23;  // __builtin_ctz(MOD - 1)
constexpr uint ZETA = PowMod(QUAD_NONRESIDUE, (MOD - 1) >> LOG2_ORD);
constexpr uint INV_ZETA = InvMod(ZETA);
```

```cpp

// 返回做 n 长 FFT 所需的单位根数组，长度为一半
std::pair<std::vector<uint>, std::vector<uint>> GetFFTRoot(int n) {
  assert((n & (n - 1)) == 0);
  if (n / 2 == 0) return {};
  std::vector<uint> root(n / 2), inv_root(n / 2);
  root[0] = inv_root[0] = 1;
  for (int i = 0; (1 << i) < n / 2; ++i)
    root[1 << i] = PowMod(ZETA, 1LL << (LOG2_ORD - i - 2)),
    inv_root[1 << i] = PowMod(INV_ZETA, 1LL << (LOG2_ORD - i - 2));
  for (int i = 1; i < n / 2; ++i)
    root[i] = (ull)root[i - (i & (i - 1))] * root[i & (i - 1)] % MOD,
    inv_root[i] =
        (ull)inv_root[i - (i & (i - 1))] * inv_root[i & (i - 1)] % MOD;
  return {root, inv_root};
}

void Butterfly(int n, uint a[], const uint root[]) {
  assert((n & (n - 1)) == 0);
  for (int i = n; i >= 2; i /= 2)
    for (int j = 0; j < n; j += i)
      for (int k = j; k < j + i / 2; ++k) {
        const uint u = a[k];
        a[k + i / 2] = (ull)a[k + i / 2] * root[j / i] % MOD;
        if ((a[k] += a[k + i / 2]) >= MOD) a[k] -= MOD;
        if ((a[k + i / 2] = u + MOD - a[k + i / 2]) >= MOD) a[k + i / 2] -= MOD;
      }
}

void InvButterfly(int n, uint a[], const uint root[]) {
  assert((n & (n - 1)) == 0);
  for (int i = 2; i <= n; i *= 2)
    for (int j = 0; j < n; j += i)
      for (int k = j; k < j + i / 2; ++k) {
        const uint u = a[k];
        if ((a[k] += a[k + i / 2]) >= MOD) a[k] -= MOD;
        a[k + i / 2] = (ull)(u + MOD - a[k + i / 2]) * root[j / i] % MOD;
      }
}

void FFT(int n, uint a[], const uint root[]) { Butterfly(n, a, root); }

void InvFFT(int n, uint a[], const uint root[]) {
  InvButterfly(n, a, root);
  const uint inv_n = InvMod(n);
  for (int i = 0; i < n; ++i) a[i] = (ull)a[i] * inv_n % MOD;
}

// 形式幂级数复合，求出 f(g) mod x^n 要求 g(0) = 0
std::vector<uint> FPSComposition(std::vector<uint> f, std::vector<uint> g,
```

```
                                  int n) {
  assert(g.empty() || g[0] == 0);
  int len = 1;
  while (len < n) len *= 2;
  std::vector<uint> root, inv_root;
  std::tie(root, inv_root) = GetFFTRoot(len * 4);
  // [y^(-1)] (f(y) / (-g(x) + y)) mod x^n in R[x]((y^(-1)))
  const auto KinoshitaLi = [&](auto &&KinoshitaLi, const std::vector<uint>
&P,
                               const std::vector<uint> &Q, int d, int n) {
    assert((int)P.size() == d * n);
    assert((int)Q.size() == d * n);
    if (n == 1) return P;
    std::vector<uint> dftQ(d * n * 4);
    for (int i = 0; i < d; ++i)
      for (int j = 0; j < n; ++j) dftQ[i * n * 2 + j] = Q[i * n + j];
    dftQ[d * n * 2] = 1;
    FFT(d * n * 4, dftQ.data(), root.data());
    std::vector<uint> V(d * n * 2);
    for (int i = 0; i < d * n * 4; i += 2)
      V[i / 2] = (ull)dftQ[i] * dftQ[i + 1] % MOD;
    InvFFT(d * n * 2, V.data(), inv_root.data());
    assert(V[0] == 1);
    V[0] = 0;
    for (int i = 1; i < d * 2; ++i)
      for (int j = 0; j < n / 2; ++j) V[i * (n / 2) + j] = V[i * n + j];
    V.resize(d * n);
    const std::vector<uint> T = KinoshitaLi(KinoshitaLi, P, V, d * 2, n /
2);
    std::vector<uint> dftT(d * n * 2);
    for (int i = 0; i < d * 2; ++i)
      for (int j = 0; j < n / 2; ++j) dftT[i * n + j] = T[i * (n / 2) + j];
    FFT(d * n * 2, dftT.data(), root.data());
    for (int i = 0; i < d * n * 4; i += 2) {
      const uint u = dftQ[i];
      dftQ[i] = (ull)dftT[i / 2] * dftQ[i + 1] % MOD;
      dftQ[i + 1] = (ull)dftT[i / 2] * u % MOD;
    }
    InvFFT(d * n * 4, dftQ.data(), inv_root.data());
    for (int i = 0; i < d; ++i)
      for (int j = 0; j < n; ++j) dftQ[i * n + j] = dftQ[(i + d) * (n * 2)
+ j];
    dftQ.resize(d * n);
    return dftQ;
  };
  f.resize(len);
  g.resize(len);
  for (int i = 0; i < len; ++i) g[i] = (g[i] != 0 ? MOD - g[i] : 0);
  std::vector<uint> res = KinoshitaLi(KinoshitaLi, f, g, 1, len);
  res.resize(n);
  return res;
}
```

```cpp

// Power Projection: [x^(n-1)] (fg^i) for i=0,..,n-1 要求 g(0) = 0
std::vector<uint> PowerProjection(std::vector<uint> f, std::vector<uint> g,
                                  int n) {
  assert(g.empty() || g[0] == 0);
  int len = 1;
  while (len < n) len *= 2;
  std::vector<uint> root, inv_root;
  std::tie(root, inv_root) = GetFFTRoot(len * 4);
  // [x^(n-1)] (f(x) / (-g(x) + y)) in R[x]((y^(-1)))
  const auto KinoshitaLi = [&](auto &&KinoshitaLi, std::vector<uint> P,
                               std::vector<uint> Q, int d,
                               int n) -> std::vector<uint> {
    assert((int)P.size() == d * n);
    assert((int)Q.size() == d * n);
    if (n == 1) return P;
    std::vector<uint> dftQ(d * n * 4), dftP(d * n * 4);
    for (int i = 0; i < d; ++i)
      for (int j = 0; j < n; ++j)
        dftP[(2 * (i + d) + 1) * n + j] = P[i * n + j],
                                      dftQ[i * n * 2 + j] = Q[i * n + j];
    dftQ[d * n * 2] = 1;
    FFT(d * n * 4, dftP.data(), inv_root.data());
    FFT(d * n * 4, dftQ.data(), root.data());
    P.resize(d * n * 2);
    Q.resize(d * n * 2);
    for (int i = 0; i < d * n * 4; i += 2) {
      P[i / 2] =
          ((ull)dftP[i] * dftQ[i + 1] + (ull)dftP[i + 1] * dftQ[i]) % MOD;
      if (P[i / 2] & 1) P[i / 2] += MOD;
      P[i / 2] /= 2;
      Q[i / 2] = (ull)dftQ[i] * dftQ[i + 1] % MOD;
    }
    InvFFT(d * n * 2, P.data(), root.data());
    InvFFT(d * n * 2, Q.data(), inv_root.data());
    for (int i = 0; i < d * 2; ++i)
      for (int j = 0; j < n / 2; ++j) P[i * (n / 2) + j] = P[i * n + n / 2
+ j];
    assert(Q[0] == 1);
    Q[0] = 0;
    for (int i = 1; i < d * 2; ++i)
      for (int j = 0; j < n / 2; ++j) Q[i * (n / 2) + j] = Q[i * n + j];
    P.resize(d * n);
    Q.resize(d * n);
    return KinoshitaLi(KinoshitaLi, P, Q, d * 2, n / 2);
  };
  f.insert(f.begin(), len - n, 0);
  f.resize(len);
  g.resize(len);
  std::reverse(f.begin(), f.end());
  for (int i = 0; i < len; ++i) g[i] = (g[i] != 0 ? MOD - g[i] : 0);
  auto res = KinoshitaLi(KinoshitaLi, f, g, 1, len);

```

```cpp
  res.resize(n);
  return res;
}

// 形式幂级数幂函数，计算 g^e mod x^n 要求 g(0) = 1
std::vector<uint> FPSPow1(std::vector<uint> g, uint e, int n) {
  assert(!g.empty() && g[0] == 1);
  if (n == 1) return std::vector<uint>{1u};
  std::vector<uint> inv(n), f(n);
  inv[1] = f[0] = 1;
  for (int i = 2; i < n; ++i)
    inv[i] = (ull)(MOD - MOD / i) * inv[MOD % i] % MOD;
  for (int i = 1; i < n; ++i)
    f[i] = (ull)f[i - 1] * (e + MOD + 1 - i) % MOD * inv[i] % MOD;
  g[0] = 0;
  return FPSComposition(f, g, n);
}

// 形式幂级数复合逆
// 计算 g mod x^n 满足 g(f) = f(g) = x 要求 g(0) = 0 且 g'(0) ≠ 0
std::vector<uint> FPSReversion(std::vector<uint> f, int n) {
  assert(f.size() >= 2 && f[0] == 0 && f[1] != 0);
  if (n == 1) return std::vector<uint>{0u};
  f.resize(n);
  const uint invf1 = InvMod(f[1]);
  uint invf1p = 1;
  for (int i = 0; i < n; ++i)
    f[i] = (ull)f[i] * invf1p % MOD, invf1p = (ull)invf1p * invf1 % MOD;
  std::vector<uint> inv(n);
  inv[1] = 1;
  for (int i = 2; i < n; ++i)
    inv[i] = (ull)(MOD - MOD / i) * inv[MOD % i] % MOD;
  std::vector<uint> proj = PowerProjection(std::vector<uint>{1u}, f, n);
  for (int i = 1; i < n; ++i)
    proj[i] = (ull)proj[i] * (n - 1) % MOD * inv[i] % MOD;
  std::reverse(proj.begin(), proj.end());
  std::vector<uint> res = FPSPow1(proj, InvMod(MOD + 1 - n), n - 1);
  for (int i = 0; i < n - 1; ++i) res[i] = (ull)res[i] * invf1 % MOD;
  res.insert(res.begin(), 0);
  return res;
}

int main() {
  int n;
  std::scanf("%d", &n);
  std::vector<uint> f(n);
  for (int i = 0; i < n; ++i) std::scanf("%u", &f[i]);
  const std::vector<uint> res = FPSReversion(f, n);
  for (int i = 0; i < n; ++i) std::printf("%u%c", res[i], " \n"[i + 1
== n]);
  return 0;
}
```

```
226
```

由转置原理导出

Power Projection 问题是 Modular Composition 的转置，Kinoshita 和 Li 指出我们前文的复合算法可以由 Power Projection 算法直接转置得到．同样的，如果优化可以应用于 Power Projection 算法，其也可以应用于 Modular Composition 算法．我们省略细节．

参考文献

1. Yasunori Kinoshita, Baitian Li.Power Series Composition in Near-Linear Time[3]. FOCS 2024.
2. Alin Bostan, Ryuhei Mori.A Simple and Fast Algorithm for Computing the N-th Term of a Linearly Recurrent Sequence[4]. SOSA 2021: 118-132
3. R. P. Brent and H. T. Kung. 1978.Fast Algorithms for Manipulating Formal Power Series[5]. J. ACM 25, 4 (Oct. 1978), 581–595.
4. Daniel J. Bernstein. "Fast multiplication and its applications[6]." Pages 325–384 in Algorithmic number theory: lattices, number fields, curves and cryptography, edited by Joe Buhler, Peter Stevenhagen, Cambridge University Press, 2008, ISBN 978-0521808545.

外部参考资料

[1] P5373【模板】多项式复合函数 

[2] P5809【模板】多项式复合逆 

[3] Power Series Composition in Near-Linear Time 

[4] A Simple and Fast Algorithm for Computing the N-th Term of a Linearly Recurrent Sequence 

[5] Fast Algorithms for Manipulating Formal Power Series 

[6] Fast multiplication and its applications 

9.12.15 普通生成函数

Authors: sshwy

序列 a 的普通生成函数（ordinary generating function，OGF）定义为形式幂级数：

$$F(x)=\sum_n a_n x^n$$

a 既可以是有穷序列，也可以是无穷序列．常见的例子（假设 a 以 0 为起点）：

1. 序列 $a=\langle 1,2,3\rangle$ 的普通生成函数是 $1+2x+3x^2$.
2. 序列 $a=\langle 1,1,1,\cdots\rangle$ 的普通生成函数是 $\sum_{n\geq 0} x^n$.
3. 序列 $a=\langle 1,2,4,8,16,\cdots\rangle$ 的生成函数是 $\sum_{n\geq 0} 2^n x^n$.
4. 序列 $a=\langle 1,3,5,7,9,\cdots\rangle$ 的生成函数是 $\sum_{n\geq 0}(2n+1)x^n$.

换句话说，如果序列 a 有通项公式，那么它的普通生成函数的系数就是通项公式.

基本运算

考虑两个序列 a,b 的普通生成函数，分别为 $F(x),G(x)$．那么有

$$F(x)\pm G(x)=\sum_n (a_n\pm b_n)x^n$$

因此 $F(x)\pm G(x)$ 是序列 $\langle a_n\pm b_n\rangle$ 的普通生成函数.

考虑乘法运算，也就是卷积：

$$F(x)G(x)=\sum_n x^n\sum_{i=0}^n a_i b_{n-i}$$

因此 $F(x)G(x)$ 是序列 $\langle\sum_{i=0}^n a_i b_{n-i}\rangle$ 的普通生成函数.

封闭形式

在运用生成函数的过程中, 我们不会一直使用形式幂级数的形式, 而会适时地转化为封闭形式以更好地化简.

例如 $\langle 1,1,1,\cdots\rangle$ 的普通生成函数 $F(x)=\sum_{n\geq 0}x^n$，我们可以发现

$$F(x)x+1=F(x)$$

那么解这个方程得到

$$F(x)=\frac{1}{1-x}$$

这就是 $\sum_{n\geq 0}x^n$ 的封闭形式.

考虑等比数列 $\langle 1,p,p^2,p^3,p^4,\cdots\rangle$ 的生成函数 $F(x)=\sum_{n\geq 0}p^nx^n$，有

$$F(x)px+1=F(x)$$

$$F(x)=\frac{1}{1-px}$$

等比数列的封闭形式与展开形式是常用的变换手段.

小练习

请求出下列数列的普通生成函数（形式幂级数形式和封闭形式）. 难度是循序渐进的.

1. $a=\langle 0,1,1,1,1,\cdots\rangle$.
2. $a=\langle 1,0,1,0,1,\cdots\rangle$.
3. $a=\langle 1,2,3,4,\cdots\rangle$.
4. $a_n=\binom{m}{n}$（m 是常数，$n\geq 0$）.
5. $a_n=\binom{m+n}{n}$（m 是常数，$n\geq 0$）.

答案

第一个:

$$F(x)=\sum_{n\geq 1}x^n=\frac{x}{1-x}$$

第二个:

$$\begin{aligned}F(x)&=\sum_{n\geq 0}x^{2n}\\&=\sum_{n\geq 0}(x^2)^n\\&=\frac{1}{1-x^2}\end{aligned}$$

第三个（求导）:

$$\begin{aligned}F(x)&=\sum_{n\geq 0}(n+1)x^n\\&=\sum_{n\geq 1}nx^{n-1}\\&=\sum_{n\geq 0}(x^n)'\\&=\left(\frac{1}{1-x}\right)'\\&=\frac{1}{(1-x)^2}\end{aligned}$$

第四个（二项式定理）：

$$F(x)=\sum_{n\geq 0}\binom{m}{n}x^n=(1+x)^m$$

第五个：

$$F(x)=\sum_{n\geq 0}\binom{m+n}{n}x^n=\frac{1}{(1-x)^{m+1}}$$

可以使用归纳法证明.

首先当 $m=0$ 时，有 $F(x)=\dfrac{1}{1-x}$.

而当 $m>0$ 时，有

$$\begin{aligned}\frac{1}{(1-x)^{m+1}}&=\frac{1}{(1-x)^m}\frac{1}{1-x}\\&=\left(\sum_{n\geq 0}\binom{m+n-1}{n}x^n\right)\left(\sum_{n\geq 0}x^n\right)\\&=\sum_{n\geq 0}x^n\sum_{i=0}^{n}\binom{m+i-1}{i}\\&=\sum_{n\geq 0}\binom{m+n}{n}x^n\end{aligned}$$

斐波那契数列的生成函数

接下来我们来推导斐波那契数列的生成函数.

斐波那契数列定义为 $a_0=0,a_1=1,a_n=a_{n-1}+a_{n-2}\ (n>1)$．设它的普通生成函数是 $F(x)$，那么根据它的递推式，我们可以类似地列出关于 $F(x)$ 的方程：

$$F(x)=xF(x)+x^2F(x)-a_0x+a_1x+a_0$$

那么解得

$$F(x)=\frac{x}{1-x-x^2}$$

那么接下来的问题是，如何求出它的展开形式?

展开方式一

不妨将 $x+x^2$ 当作一个整体，那么可以得到

$$\begin{aligned}F(x)&=\frac{x}{1-(x+x^2)}\\&=x\sum_{k=0}^{\infty}(x+x^2)^k\\&=x\sum_{k=0}^{\infty}\sum_{i=0}^{k}\binom{k}{i}x^{k-i}(x^2)^i\\&=\sum_{k=0}^{\infty}\sum_{i=0}^{k}\binom{k}{i}x^{k+i+1}\\&=\sum_{n=1}^{\infty}\sum_{i=0}^{\lfloor (n-1)/2\rfloor}\binom{n-i-1}{i}x^n.\end{aligned}$$

最后一步中，令 $n=k+i+1$ 并更换求和顺序．由此，可以得到通项公式：

$$a_n = \sum_{i=0}^{\lfloor (n-1)/2 \rfloor} \binom{n-i-1}{i}.$$

这并不是我们熟知的有关黄金分割比的形式.

展开方式二

考虑求解一个待定系数的方程:

$$\frac{A}{1-ax} + \frac{B}{1-bx} = \frac{x}{1-x-x^2}$$

通分得到

$$\frac{A - Abx + B - aBx}{(1-ax)(1-bx)} = \frac{x}{1-x-x^2}$$

待定项系数相等，我们得到

$$\begin{cases} A + B = 0 \\ -Ab - aB = 1 \\ a + b = 1 \\ ab = -1 \end{cases}$$

解得

$$\begin{cases} A = \frac{1}{\sqrt{5}} \\ B = -\frac{1}{\sqrt{5}} \\ a = \frac{1+\sqrt{5}}{2} \\ b = \frac{1-\sqrt{5}}{2} \end{cases}$$

那么我们根据等比数列的展开式，就可以得到斐波那契数列的通项公式:

$$\frac{x}{1-x-x^2} = \sum_{n \geq 0} x^n \frac{1}{\sqrt{5}} \left(\left(\frac{1+\sqrt{5}}{2} \right)^n - \left(\frac{1-\sqrt{5}}{2} \right)^n \right)$$

这也被称为斐波那契数列的另一个封闭形式（$\frac{x}{1-x-x^2}$ 是一个封闭形式）.

对于任意多项式 $P(x), Q(x)$，生成函数 $\dfrac{P(x)}{Q(x)}$ 的展开式都可以使用上述方法求出. 在实际运用的过程中，我们往往先求出 $Q(x)$ 的根，把分母表示为 $\prod (1-p_i x)^{d_i}$ 的形式，然后再求分子.

当对分母进行因式分解但有重根时，每有一个重根就要多一个分式，如考虑生成函数

$$G(x) = \frac{1}{(1-x)(1-2x)^2}$$

的系数的通项公式，那么有

$$G(x) = \frac{c_0}{1-x} + \frac{c_1}{1-2x} + \frac{c_2}{(1-2x)^2}$$

解得

$$\begin{cases} c_0 = 1 \\ c_1 = -2 \\ c_2 = 2 \end{cases}$$

那么

$$[x^n]G(x) = 1 - 2^{n+1} + (n+1) \cdot 2^{n+1}$$

牛顿二项式定理

我们重新定义组合数的运算:

$$\binom{r}{k} = \frac{r^{\underline{k}}}{k!} \quad (r \in \mathbf{C}, k \in \mathbf{N})$$

注意 r 的范围是复数域．在这种情况下．对于 $\alpha \in \mathbf{C}$，有

$$(1+x)^{\alpha} = \sum_{n \geq 0} \binom{\alpha}{n} x^n$$

二项式定理其实是牛顿二项式定理的一个特殊情况．

卡特兰数的生成函数

参考 Catalan 数形式的代数推演→9.13.6@p1637.

应用

接下来给出一些例题，来介绍生成函数在 OI 中的具体应用．

食物

食物[1]

在许多不同种类的食物中选出 n 个，每种食物的限制如下：

1. 承德汉堡：偶数个
2. 可乐：0 个或 1 个
3. 鸡腿：0 个，1 个或 2 个
4. 蜜桃多：奇数个
5. 鸡块：4 的倍数个
6. 包子：0 个，1 个，2 个或 3 个
7. 土豆片炒肉：不超过一个．
8. 面包：3 的倍数个

每种食物都是以「个」为单位，只要总数加起来是 n 就算一种方案．对于给出的 n 你需要计算出方案数，对 10007 取模．

这是一道经典的生成函数题．对于一种食物，我们可以设 a_n 表示这种食物选 n 个的方案数，并求出它的生成函数．而两种食物一共选 n 个的方案数的生成函数，就是它们生成函数的卷积．多种食物选 n 个的方案数的生成函数也是它们生成函数的卷积．

在理解了方案数可以用卷积表示以后，我们就可以构造生成函数（标号对应题目中食物的标号）：

1. $\sum_{n \geq 0} x^{2n} = \dfrac{1}{1-x^2}$.
2. $1+x$.
3. $1+x+x^2 = \dfrac{1-x^3}{1-x}$.
4. $\dfrac{x}{1-x^2}$.
5. $\sum_{n \geq 0} x^{4n} = \dfrac{1}{1-x^4}$.
6. $1+x+x^2+x^3 = \dfrac{1-x^4}{1-x}$.
7. $1+x$.
8. $\dfrac{1}{1-x^3}$.

那么全部乘起来，得到答案的生成函数：

$$F(x) = \frac{(1+x)(1-x^3)x(1-x^4)(1+x)}{(1-x^2)(1-x)(1-x^2)(1-x^4)(1-x)(1-x^3)} = \frac{x}{(1-x)^4}$$

然后将它转化为展开形式（使用封闭形式练习中第五个练习）：

$$F(x) = \sum_{n \geq 1} \binom{n+2}{n-1} x^n$$

因此答案就是 $\binom{n+2}{n-1} = \binom{n+2}{3}$.

Sweet

「CEOI2004」Sweet[2]

有 n 堆糖果．不同的堆里糖果的种类不同（即同一个堆里的糖果种类是相同的，不同的堆里的糖果的种类是不同的）．第 i 个堆里有 m_i 个糖果．现在要吃掉至少 a 个糖果，但不超过 b 个．求有多少种方案．

两种方案不同当且仅当吃的个数不同，或者吃的糖果中，某一种糖果的个数在两个方案中不同．

$n \le 10, 0 \le a \le b \le 10^7, m_i \le 10^6$．

在第 i 堆吃 j 个糖果的方案数（显然为 1）的生成函数为

$$F_i(x) = \sum_{j=0}^{m_i} x^j = \frac{1-x^{m_i+1}}{1-x}$$

因此总共吃 i 个糖果的方案数的生成函数就是

$$G(x) = \prod_{i=1}^{n} F_i(x) = (1-x)^{-n} \prod_{i=1}^{n} (1-x^{m_i+1})$$

现在我们要求的是 $\sum_{i=a}^{b} [x^i] G(x)$．

由于 $n \le 10$，因此我们可以暴力展开 $\prod_{i=1}^{n}(1-x^{m_i+1})$（最多只有 2^n 项）．

然后对 $(1-x)^{-n}$ 使用牛顿二项式定理：

$$\begin{aligned}(1-x)^{-n} &= \sum_{i \ge 0} \binom{-n}{i} (-x)^i \\ &= \sum_{i \ge 0} \binom{n-1+i}{i} x^i\end{aligned}$$

我们枚举 $\prod_{i=1}^{n}(1-x^{m_i+1})$ 中 x^k 项的系数，假设为 c_k．那么它和 $(1-x)^{-n}$ 相乘后，对答案的贡献就是

$$c_k \sum_{i=a-k}^{b-k} \binom{n-1+i}{i} = c_k \left(\binom{n+b-k}{b-k} - \binom{n+a-k-1}{a-k-1} \right)$$

这样就可以 $O(b)$ 地求出答案了．

时间复杂度 $O(2^n + b)$．

外部参考资料

[1] 食物  [2] 「CEOI2004」Sweet 

9.12.16 指数生成函数

Authors: sshwy, ComeIntoCalm

序列 a 的指数生成函数（exponential generating function，EGF）定义为形式幂级数：

$$\hat{F}(x) = \sum_n a_n \frac{x^n}{n!}$$

基本运算

指数生成函数的加减法与普通生成函数是相同的，也就是对应项系数相加．

考虑指数生成函数的乘法运算．对于两个序列 a, b，设它们的指数生成函数分别为 $\hat{F}(x), \hat{G}(x)$，那么

$$\begin{aligned}\hat{F}(x)\hat{G}(x)&=\sum_{i\geq 0}a_i\frac{x^i}{i!}\sum_{j\geq 0}b_j\frac{x^j}{j!}\\&=\sum_{n\geq 0}x^n\sum_{i=0}^{n}a_ib_{n-i}\frac{1}{i!(n-i)!}\\&=\sum_{n\geq 0}\frac{x^n}{n!}\sum_{i=0}^{n}\binom{n}{i}a_ib_{n-i}\end{aligned}$$

因此 $\hat{F}(x)\hat{G}(x)$ 是序列

$$\left\langle\sum_{i=0}^{n}\binom{n}{i}a_ib_{n-i}\right\rangle$$

的指数生成函数.

封闭形式

我们同样考虑指数生成函数的封闭形式.

序列 $\langle 1,1,1,\cdots\rangle$ 的指数生成函数是:

$$\hat{F}(x)=\sum_{n\geq 0}\frac{x^n}{n!}=\mathrm{e}^x$$

因为你将 e^x 在 $x=0$ 处泰勒展开就得到了它的无穷级数形式.

类似地，等比数列 $\langle 1,p,p^2,\cdots\rangle$ 的指数生成函数是:

$$\hat{F}(x)=\sum_{n\geq 0}\frac{p^nx^n}{n!}=\mathrm{e}^{px}$$

指数生成函数与普通生成函数

如何理解指数生成函数？我们定义序列 a 的指数生成函数是:

$$F(x)=\sum_{n\geq 0}a_n\frac{x^n}{n!}$$

但 $F(x)$ 实际上也是序列 $\left\langle\frac{a_n}{n!}\right\rangle$ 的普通生成函数.

这两种理解没有任何问题. 也就是说，不同的生成函数只是对问题理解方式的转变.

EGF 中多项式 exp 的组合意义

如果您还没有学习多项式 exp 请先跳过这里，这是由 exp 理解引出的意义，从某种意义上来说可以加深对 EGF 的理解.

EGF 中 $f^n(x)$ 的 f 默认是一个 EGF，那么我们首先考虑任意两个 EGF 的乘积

$$\hat{H}(x)=\hat{F}(x)\hat{G}(x)=\sum_{n\geq 0}\left[\sum_{i=0}^{n}\binom{n}{i}f_ig_{n-i}\right]\frac{x^n}{n!}$$

对于两个 EGF 相乘得到的 $[x^k]\hat{H}(x)$, 实际上是一个卷积. 而如果考虑多个 EGF 相乘得到的 $[x^k]\hat{H}(x)$, 实际上就是对每个 EGF 选择一项 x^{a_i} 使得 $\sum_i a_i=k$ 时每种情况系数的和. 从集合的角度来理解就是把 n 个有标号元素划分为 $k>0$ 个有标号集合的方案数.

> 如果 $k=0$ 则系数显然为原 EGF 各项常数的积，但是多项式 exp 中某些要求导致 exp 的 $f(x)$ 常数项必须为 0，具体的原因在下文中会做出一些说明.

多项式系数定义（具体请参考排列组合一栏中的多项式系数组合意义）里是默认集合有序的，但是 $\exp(f(x))$ 中 $f^k(x)$，k 个 $f(x)$ 相乘得到的 EGF 相同，而集合划分显然是无序的，因此其系数应该乘上 $\frac{1}{k!}$.

设 $F_k(n)$ 为 n 个有标号元素划分成 k 个非空无序集合（因为是 exp 所以有非空要求）的情况，f_i 为 i 个元素组成一个集合时，i 个元素的有限集上特定组合结构的数量（是原有的 EGF，对这一个集合元素计数的方案，仅仅与该集合大小有关），那么 $F_k(n)$

$$F_k(n)=\frac{n!}{k!}\sum_{\sum_i^k a_i=n}\prod_{j=1}^{k}\frac{f_{a_j}}{a_{(j)!}}$$

设 f_n 的 EGF 为 $\hat{F}(x)$，即：

$$\hat{F}(x)=\sum_{n\geq 0}f_n\frac{x^n}{n!}$$

设 $F_k(n)$ 的 EGF 为 $G_k(x)$，则：

$$\begin{aligned}G_k(x)&=\sum_{n\geq 0}F_k(n)\frac{x^n}{n!}\\&=\sum_{n\geq 0}x^n\frac{1}{k!}\sum_{\sum_i^k a_i=n}\prod_{j=1}^{k}\frac{f_{a_j}}{a_{(j)!}}\\&=\frac{1}{k!}\sum_{n\geq 0}\sum_{\sum_i^k a_i=n}\prod_{j=1}^{k}\frac{f_{a_j}x^{a_j}}{a_{(j)!}}\\&=\frac{1}{k!}\hat{F}^k(x)\end{aligned}$$

对于所有的 $k\geq 0$：

$$\sum_{k\geq 0}G_k(x)=\sum_{k\geq 0}\frac{\hat{F}^k(x)}{k!}=\exp\hat{F}(x)$$

上面是从组合角度直接列式理解，我们也可以从递推方面来证明 $\exp(f(x))$ 和 $f(x)$ 两者间的关系.

同样设 $F_k(n)$ 为 n 个有标号元素划分成 k 个非空集合（无标号）的情况，g_i 为 i 个元素组成一个集合内部的方案数（意义同上文中的 f_i），并令 $G(x)$ 为 $\{g_i\}$ 的 EGF，$H_k(x)$ 为 $\{F_k(n)\}$ 的 EGF.

n 个元素中取出 i 个元素作为一个单独划分出去的集合共有 g_i 种方案，剩下的 $n-i$ 个元素构成 $k-1$ 个集合共 $F_{k-1}(n-i)$ 种方案，但最后的划分方案中，一个方案里的每个集合都会被枚举为单独划分出去的集合，所以重复计算了 k 次，故还需要除以 k.

$$\begin{aligned}H_k(x)&=\sum_{n\geq 0}\frac{x^n}{n!}F_k(n)\\&=\sum_{n\geq 0}\frac{x^n}{n!}\sum_{i=1}^{n-k+1}\binom{n}{i}F_{k-1}(n-i)\times g_i\times\frac{1}{k}\\&=\frac{1}{k}\sum_{n\geq 0}\frac{x^n}{n!}\sum_{i=0}^{n}\binom{n}{i}F_{k-1}(n-i)\times g_i\\&=\frac{1}{k}\cdot H_{k-1}(x)G(x)\end{aligned}$$

上界是由非空集合划分推出的 $n-(k-1)\geq i$（前 $k-1$ 个集合每个集合最少有一个元素），但是如果超过枚举上界涉及的 $F_{k-1}(n-i)$ 设为 0，那么就没有影响.

得到递推式之后可递归展开，边界为 $k=1$ 时 $H_1(x)=G(x)$.

$$\begin{aligned}H_k(x)&=\frac{1}{k}\cdot H_{k-1}(x)G(x)\\&=\frac{1}{k}\cdot\frac{1}{k-1}\cdot H_{k-2}(x)G^2(x)\\&=\cdots\\&=\frac{1}{k}\cdot\frac{1}{k-1}\cdots\frac{1}{2}\cdot H_1(x)G^{k-1}(x)\\&=\frac{1}{k!}G^k(x)\end{aligned}$$

同样的有：

$$\sum_{k\geq 0}H_k(x)=\sum_{k\geq 0}\frac{G^k(x)}{k!}=\exp G(x)$$

显然**定义成划分为非空集合**（$g_0=0$）是符合本身的意义的，如果**包含空集**（$g_0=1$），那么对应 $[x^n]G^k$ 中就会有 $[x^n]G^y,y>k$ 的贡献（在至少一个 G 中选择常数项），有计重，得不到所求量.

从递推式的角度讲，多个 EGF 的乘积也可以看作一个类似于背包的组合（合并两组计数对象的过程）.

总结多项式 exp 的意义就是：**有标号元素构成的集合的生成集族有多少种情况**，或划分为任意个非空子集的总方案数.

排列与圆排列

长度为 n 的排列数的指数生成函数是

$$\hat{P}(x)=\sum_{n\geq 0}\frac{n!x^n}{n!}=\sum_{n\geq 0}x^n=\frac{1}{1-x}$$

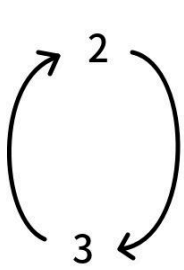
圆排列的定义是把 $1,2,\cdots,n$ 排成一个环的方案数. 也就是说旋转后的方案的等价的（但翻转是不等价的）.

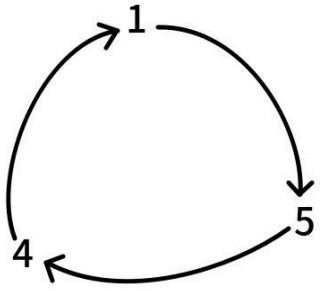
n 个数的圆排列数显然是 $(n-1)!$. 因此 n 个数的圆排列数的指数生成函数是

$$\hat{Q}(x)=\sum_{n\geq 1}\frac{(n-1)!x^n}{n!}=\sum_{n\geq 1}\frac{x^n}{n}=-\ln(1-x)=\ln\left(\frac{1}{1-x}\right)$$

也就是说 $\exp\hat{Q}(x)=\hat{P}(x)$. 但这只是数学层面的推导. 我们该怎样直观理解：圆排列数的 EGF 的 exp 是排列数的 EGF？

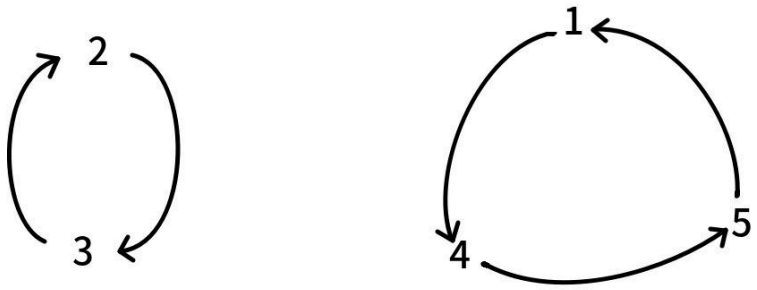
一个排列，是由若干个置换环构成的. 例如 $p=[4,3,2,5,1]$ 有两个置换环：





（也就是说我们从 p_i 向 i 连有向边）

而不同的置换环，会导出不同的排列. 例如将第二个置换环改成



那么它对应的排列就是 $[5,3,2,1,4]$.

也就是说，长度为 n 的排列的方案数是

1. 把 $1,2,\cdots,n$ 分成若干个集合
2. 每个集合形成一个置换环

的方案数. 而一个集合的数形成置换环的方案数显然就是这个集合大小的圆排列方案数. 因此长度为 n 的排列的方案数就是：把 $1,2,\cdots,n$ 分成若干个集合，每个集合的圆排列方案数之积.

这就是多项式 exp 的直观理解.

推广之

- 如果 n 个点 **带标号** 生成树的 EGF 是 $\hat{F}(x)$，那么 n 个点 **带标号** 生成森林的 EGF 就是 $\exp \hat{F}(x)$——直观理解为，将 n 个点分成若干个集合，每个集合构成一个生成树的方案数之积.
- 如果 n 个点带标号无向连通图的 EGF 是 $\hat{F}(x)$，那么 n 个点带标号无向图的 EGF 就是 $\exp \hat{F}(x)$，后者可以很容易计算得到

  $$\exp \hat{F}(x)=\sum_{n\geq 0}2^{\binom{n}{2}}\frac{x^n}{n!}$$

  因此要计算前者，只需要一次多项式 ln 即可.

接下来我们来看一些指数生成函数的应用.

应用

错排数

错排数

定义长度为 n 的一个错排是满足 $p_i\neq i$ 的排列.

求错排数的指数生成函数.

从置换环的角度考虑，错排就是指置换环中不存在自环的排列. 也就是说不存在长度为 1 的置换环. 后者的指数生成函数是

$$\sum_{n\geq 2}\frac{x^n}{n}=-\ln(1-x)-x$$

因此错排数的指数生成函数就是 $\exp(-\ln(1-x)-x)$.

不动点

不动点[1]

题意：求有多少个映射 $f:\{1,2,\cdots,n\}\to\{1,2,\cdots,n\}$，使得

$$\underbrace{f\circ f\circ\cdots\circ f}_{k}=\underbrace{f\circ f\circ\cdots\circ f}_{k-1}$$

$nk \le 2 \times 10^6, 1 \le k \le 3$.

考虑 i 向 $f(i)$ 连边．相当于我们从任意一个 i 走 k 步和走 $k-1$ 步到达的是同一个点．也就是说基环树的环是自环且深度不超过 k（根结点深度为 1）．把这个基环树当成有根树是一样的．因此我们的问题转化为：n 个点带标号，深度不超过 k 的有根树森林的计数．

考虑 n 个点带标号深度不超过 k 的有根树，假设它的生成函数是：

$$\widehat{F_k}(x) = \sum_{n \ge 0} f_{n,k} \frac{x^n}{n!}$$

考虑递推求 $\widehat{F_k}(x)$．深度不超过 k 的有根树，实际上就是深度不超过 $k-1$ 的若干棵有根树，把它们的根结点全部连到一个结点上去．因此

$$\widehat{F_k}(x) = x \exp \hat{F}_{k-1}(x)$$

那么答案的指数生成函数就是 $\exp \hat{F}_k(x)$．求它的第 n 项即可．

Lust

Lust[2]

给你一个 n 个数的序列 $a_1, a_2, \cdots, a_n$，和一个初值为 0 的变量 s，要求你重复以下操作 k 次：

- 在 $1, 2, \cdots, n$ 中等概率随机选择一个 x．
- 令 s 加上 $\prod_{i \ne x} a_i$．
- 令 a_x 减一．

求 k 次操作后 s 的期望．

$1 \le n \le 5000, 1 \le k \le 10^9, 0 \le a_i \le 10^9$.

假设 k 次操作后 a_i 减少了 b_i，那么实际上

$$s = \prod_{i=1}^{n} a_i - \prod_{i=1}^{n} (a_i - b_i)$$

因此实际上我们的问题转化为，求 k 次操作后 $\prod_{i=1}^{n} (a_i - b_i)$ 的期望．

不妨考虑计算每种方案的 $\prod_{i=1}^{n} (a_i - b_i)$ 的和，最后除以 n^k．

而 k 次操作序列中，要使得 i 出现了 b_i 次的方案数是

$$\frac{k!}{b_{(1)!} b_{(2)!} \cdots b_{(n)!}}$$

这与指数生成函数乘法的系数类似．

设 a_j 的指数生成函数是

$$F_j(x) = \sum_{i \ge 0} (a_j - i) \frac{x^i}{i!}$$

那么答案就是

$$[x^k] \prod_{j=1}^{n} F_j(x)$$

为了快速计算答案，我们需要将 $F_j(x)$ 转化为封闭形式：

$$\begin{aligned} F_j(x) &= \sum_{i \ge 0} a_j \frac{x^i}{i!} - \sum_{i \ge 1} \frac{x^i}{(i-1)!} \\ &= a_j \mathrm{e}^x - x \mathrm{e}^x \\ &= (a_j - x) \mathrm{e}^x \end{aligned}$$

因此我们得到

$$\prod_{j=1}^{n} F_j(x) = \mathrm{e}^{nx} \prod_{j=1}^{n} (a_j - x)$$

其中 $\prod_{j=1}^{n}(a_j - x)$ 是一个 n 次多项式，可以暴力计算出来．假设它的展开式是 $\sum_{i=0}^{n} c_i x^i$，那么

$$\begin{aligned}\prod_{j=1}^{n} F_j(x) &= \left(\sum_{i\geq 0} \frac{n^i x^i}{i!}\right)\left(\sum_{i=0}^{n} c_i x^i\right) \\ &= \sum_{i\geq 0}\sum_{j=0}^{i} c_j x^j \frac{n^{i-j}x^{i-j}}{(i-j)!} \\ &= \sum_{i\geq 0} \frac{x^i}{i!} \sum_{j=0}^{i} n^{i-j} i^{\underline{j}} c_j\end{aligned}$$

计算这个多项式的 x^k 项系数即可．

外部参考资料

[1] 不动点  [2] Lust 

9.13 组合数学

9.13.1 排列组合

引入

排列组合是组合数学中的基础. 排列就是指从给定个数的元素中取出指定个数的元素进行排序; 组合则是指从给定个数的元素中仅仅取出指定个数的元素, 不考虑排序. 排列组合的中心问题是研究给定要求的排列和组合可能出现的情况总数．排列组合与古典概率论关系密切.

在高中初等数学中，排列组合多是利用列表、枚举等方法解题.

加法 & 乘法原理

加法原理

完成一个工程可以有 n 类办法，$a_i(1 \leq i \leq n)$ 代表第 i 类方法的数目．那么完成这件事共有 $S = a_1 + a_2 + \cdots + a_n$ 种不同的方法.

乘法原理

完成一个工程需要分 n 个步骤，$a_i(1 \leq i \leq n)$ 代表第 i 个步骤的不同方法数目．那么完成这件事共有 $S = a_1 \times a_2 \times \cdots \times a_n$ 种不同的方法.

排列与组合基础

排列数

从 n 个不同元素中，任取 m（$m \leq n$，m 与 n 均为自然数，下同）个元素按照一定的顺序排成一列，叫做从 n 个不同元素中取出 m 个元素的一个排列；从 n 个不同元素中取出 $m(m \leq n)$ 个元素的所有排列的个数，叫做从 n 个不同元素中取出 m 个元素的排列数，用符号 A_n^m（或者是 P_n^m）表示.

排列的计算公式如下:

$$\mathrm{A}_n^m = n(n-1)(n-2)\cdots(n-m+1) = \frac{n!}{(n-m)!}$$

$n!$ 代表 n 的阶乘，即 $6! = 1 \times 2 \times 3 \times 4 \times 5 \times 6$.

公式可以这样理解：n 个人选 m 个来排队 ($m \le n$)．第一个位置可以选 n 个，第二位置可以选 $n-1$ 个，以此类推，第 m 个（最后一个）可以选 $n-m+1$ 个，得：

$$\mathrm{A}_n^m = n(n-1)(n-2)\cdots(n-m+1) = \frac{n!}{(n-m)!}$$

全排列：n 个人全部来排队，队长为 n．第一个位置可以选 n 个，第二位置可以选 $n-1$ 个，以此类推得：

$$\mathrm{A}_n^n = n(n-1)(n-2)\cdots 3\times 2\times 1 = n!$$

全排列是排列数的一个特殊情况.

组合数

从 n 个不同元素中，任取 $m \le n$ 个元素组成一个集合，叫做从 n 个不同元素中取出 m 个元素的一个组合；从 n 个不同元素中取出 $m \le n$ 个元素的所有组合的个数，叫做从 n 个不同元素中取出 m 个元素的组合数，用符号 $\binom{n}{m}$ 来表示，读作「n 选 m」.

组合数计算公式

$$\binom{n}{m} = \frac{\mathrm{A}_n^m}{m!} = \frac{n!}{m!(n-m)!}$$

如何理解上述公式？我们考虑 n 个人选 m 个出来（$m \le n$），不排队，不在乎顺序．如果在乎顺序那么就是 A_n^m，如果不在乎那么就要除掉重复，那么重复了多少？同样选出来的 m 个人，他们还要「全排」得 $m!$，所以得：

$$\binom{n}{m} \times m! = \mathrm{A}_n^m$$

$$\binom{n}{m} = \frac{\mathrm{A}_n^m}{m!} = \frac{n!}{m!(n-m)!}$$

组合数也常用 C_n^m 表示，即 $\mathrm{C}_n^m = \binom{n}{m}$．现在数学界普遍采用 $\binom{n}{m}$ 的记号而非 C_n^m.

组合数也被称为「二项式系数」，下文二项式定理将会阐述其中的联系.

特别地，规定当 $m > n$ 时，$\mathrm{A}_n^m = \binom{n}{m} = 0$.

插板法

插板法（Stars and bars）是用于求一类给相同元素分组的方案数的一种技巧，也可以用于求一类线性不定方程的解的组数.

正整数和的数目

问题一：现有 n 个 **完全相同** 的元素，要求将其分为 k 组，保证每组至少有一个元素，一共有多少种分法?

考虑拿 $k-1$ 块板子插入到 n 个元素两两形成的 $n-1$ 个空里面.

因为元素是完全相同的，所以答案就是 $\binom{n-1}{k-1}$.

本质是求 $x_1 + x_2 + \cdots + x_k = n$ 的正整数解的组数.

非负整数和的数目

问题二：如果问题变化一下，每组允许为空呢?

显然此时没法直接插板了，因为有可能出现很多块板子插到一个空里面的情况，非常不好计算.

我们考虑创造条件转化成有限制的问题一，先借 k 个元素过来，在这 $n+k$ 个元素形成的 $n+k-1$ 个空里面插板，答案为

$$\binom{n+k-1}{k-1} = \binom{n+k-1}{n}$$

虽然不是直接求的原问题，但这个式子就是原问题的答案，可以这么理解：

开头我们借来了 k 个元素，用于保证每组至少有一个元素，插完板之后再把这 k 个借来的元素从 k 组里面拿走．因为元素是相同的，所以转化过的情况和转化前的情况可以一一对应，答案也就是相等的．

由此可以推导出插板法的公式：$\binom{n+k-1}{n}$．

本质是求 $x_1+x_2+\cdots+x_k=n$ 的非负整数解的组数（即要求 $x_i\geq 0$）．

不同下界整数和的数目

问题三：如果再扩展一步，要求对于第 i 组，至少要分到 $a_i,\sum a_i\leq n$ 个元素呢?

本质是求 $x_1+x_2+\cdots+x_k=n$ 的解的数目，其中 $x_i\geq a_i$．

类比无限制的情况，我们借 $\sum a_i$ 个元素过来，保证第 i 组至少能分到 a_i 个．也就是令

$$x_i'=x_i-a_i$$

得到新方程：

$$\begin{aligned}(x_1'+a_1)+(x_2'+a_2)+\cdots+(x_k'+a_k)&=n\\ x_1'+x_2'+\cdots+x_k'&=n-a_1-a_2-\cdots-a_k\\ x_1'+x_2'+\cdots+x_k'&=n-\sum a_i\end{aligned}$$

其中

$$x_i'\geq 0$$

然后问题三就转化成了问题二，直接用插板法公式得到答案为

$$\binom{n-\sum a_i+k-1}{n-\sum a_i}$$

不相邻的排列

$1\sim n$ 这 n 个自然数中选 k 个，这 k 个数中任何两个数都不相邻的组合有 $\binom{n-k+1}{k}$ 种．

二项式定理

在进入排列组合进阶篇之前，我们先介绍一个与组合数密切相关的定理——二项式定理．

二项式定理阐明了一个展开式的系数：

$$(a+b)^n=\sum_{i=0}^{n}\binom{n}{i}a^{n-i}b^i$$

证明可以采用数学归纳法，利用 $\binom{n}{k}+\binom{n}{k-1}=\binom{n+1}{k}$ 做归纳．

二项式定理也可以很容易扩展为多项式的形式：

设 n 为正整数，x_i 为实数，

$$(x_1+x_2+\cdots+x_t)^n=\sum_{\text{满足}n_1+\cdots+n_t=n\text{的非负整数解}}\binom{n}{n_1,n_2,\cdots,n_t}x_1^{n_1}x_2^{n_2}\cdots x_t^{n_t}$$

其中的 $\binom{n}{n_1,n_2,\cdots,n_t}$ 是多项式系数，它的性质也很相似：

$$\sum\binom{n}{n_1,n_2,\cdots,n_t}=t^n$$

排列与组合进阶篇

接下来我们介绍一些排列组合的变种．

多重集的排列数|多重组合数

请大家一定要区分 **多重组合数** 与 **多重集的组合数**！两者是完全不同的概念！

多重集是指包含重复元素的广义集合. 设 $S=\{n_1\cdot a_1,n_2\cdot a_2,\cdots,n_k\cdot a_k\}$ 表示由 n_1 个 a_1, n_2 个 a_2, ..., n_k 个 a_k 组成的多重集，S 的全排列个数为

$$\frac{n!}{\prod_{i=1}^{k}n_{(i)!}}=\frac{n!}{n_{(1)!}n_{(2)!}\cdots n_{(k)!}}$$

相当于把相同元素的排列数除掉了. 具体地，你可以认为你有 k 种不一样的球，每种球的个数分别是 $n_1,n_2,\cdots,n_k$，且 $n=n_1+n_2+...+n_k$. 这 n 个球的全排列数就是 **多重集的排列数**. 多重集的排列数常被称作 **多重组合数**. 我们可以用多重组合数的符号表示上式:

$$\binom{n}{n_1,n_2,\cdots,n_k}=\frac{n!}{\prod_{i=1}^{k}n_{(i)!}}$$

可以看出，$\binom{n}{m}$ 等价于 $\binom{n}{m,n-m}$，只不过后者较为繁琐，因而不采用.

多重集的组合数 1

设 $S=\{n_1\cdot a_1,n_2\cdot a_2,\cdots,n_k\cdot a_k\}$ 表示由 n_1 个 a_1，n_2 个 a_2，...，n_k 个 a_k 组成的多重集. 那么对于整数 $r(r<n_i,\forall i\in[1,k])$，从 S 中选择 r 个元素组成一个多重集的方案数就是 **多重集的组合数**. 这个问题等价于 $x_1+x_2+\cdots+x_k=r$ 的非负整数解的数目，可以用插板法解决，答案为

$$\binom{r+k-1}{k-1}$$

多重集的组合数 2

考虑这个问题: 设 $S=\{n_1\cdot a_1,n_2\cdot a_2,\cdots,n_k\cdot a_k,\}$ 表示由 n_1 个 a_1，n_2 个 a_2，...，n_k 个 a_k 组成的多重集. 那么对于正整数 r，从 S 中选择 r 个元素组成一个多重集的方案数.

这样就限制了每种元素的取的个数. 同样的，我们可以把这个问题转化为带限制的线性方程求解:

$$\forall i\in[1,k],x_i\le n_i,\sum_{i=1}^{k}x_i=r$$

于是很自然地想到了容斥原理. 容斥的模型如下:

1. 全集: $\sum_{i=1}^{k}x_i=r$ 的非负整数解.
2. 属性: $x_i\le n_i$.

于是设满足属性 i 的集合是 S_i，$\overline{S_i}$ 表示不满足属性 i 的集合，即满足 $x_i\ge n_i+1$ 的集合（转化为上面插板法的问题三）. 那么答案即为

$$\left|\bigcap_{i=1}^{k}S_i\right|=|U|-\left|\bigcup_{i=1}^{k}\overline{S_i}\right|$$

根据容斥原理，有:

$$\begin{aligned}\left|\bigcup_{i=1}^{k}\overline{S_i}\right|&=\sum_i\left|\overline{S_i}\right|-\sum_{i,j}\left|\overline{S_i}\cap\overline{S_j}\right|+\sum_{i,j,k}\left|\overline{S_i}\cap\overline{S_j}\cap\overline{S_k}\right|-\cdots\\&\quad+(-1)^{k-1}\left|\bigcap_{i=1}^{k}\overline{S_i}\right|\\&=\sum_i\binom{k+r-n_i-2}{k-1}-\sum_{i,j}\binom{k+r-n_i-n_j-3}{k-1}+\sum_{i,j,k}\binom{k+r-n_i-n_j-n_k-4}{k-1}-\cdots\\&\quad+(-1)^{k-1}\binom{k+r-\sum_{i=1}^{k}n_i-k-1}{k-1}\end{aligned}$$

拿全集 $|U|=\binom{k+r-1}{k-1}$ 减去上式，得到多重集的组合数

$$Ans=\sum_{p=0}^{k}(-1)^p\sum_{A}\binom{k+r-1-\sum_A n_{A_i}-p}{k-1}$$

其中 A 是充当枚举子集的作用，满足 $|A|=p, A_i<A_{i+1}$.

圆排列

n 个人全部来围成一圈，所有的排列数记为 Q_n^n. 考虑其中已经排好的一圈，从不同位置断开，又变成不同的队列. 所以有

$$\mathrm{Q}_n^n\times n=\mathrm{A}_n^n\Longrightarrow \mathrm{Q}_n=\frac{\mathrm{A}_n^n}{n}=(n-1)!$$

由此可知部分圆排列的公式:

$$\mathrm{Q}_n^r=\frac{\mathrm{A}_n^r}{r}=\frac{n!}{r\times(n-r)!}$$

组合数性质 | 二项式推论

由于组合数在 OI 中十分重要，因此在此介绍一些组合数的性质.

$$\binom{n}{m}=\binom{n}{n-m} \tag{1}$$

相当于将选出的集合对全集取补集，故数值不变.（对称性）

$$\binom{n}{k}=\frac{n}{k}\binom{n-1}{k-1} \tag{2}$$

由定义导出的递推式.

$$\binom{n}{m}=\binom{n-1}{m}+\binom{n-1}{m-1} \tag{3}$$

组合数的递推式（杨辉三角的公式表达）. 我们可以利用这个式子，在 $O(n^2)$ 的复杂度下推导组合数.

$$\binom{n}{0}+\binom{n}{1}+\cdots+\binom{n}{n}=\sum_{i=0}^{n}\binom{n}{i}=2^n \tag{4}$$

这是二项式定理的特殊情况. 取 $a=b=1$ 就得到上式.

$$\sum_{i=0}^{n}(-1)^i\binom{n}{i}=[n=0] \tag{5}$$

二项式定理的另一种特殊情况，可取 $a=1,b=-1$. 式子的特殊情况是取 $n=0$ 时答案为 1.

$$\sum_{i=0}^{k}\binom{n}{i}\binom{m}{k-i}=\binom{m+n}{k} \tag{6}$$

拆组合数的式子，在处理某些数据结构题时会用到. 被称为 范德蒙恒等式[1].

$$\sum_{i=0}^{n}\binom{n}{i}^2=\binom{2n}{n} \tag{7}$$

这是 (6) 的特殊情况，取 $n=k=m$ 即可.

$$\sum_{i=0}^{n}i\binom{n}{i}=n2^{n-1} \tag{8}$$

带权和的一个式子，通过对 (4) 对应的多项式函数求导可以得证.

$$\sum_{i=0}^{n}i^2\binom{n}{i}=n(n+1)2^{n-2} \tag{9}$$

与上式类似，可以通过对多项式函数求导证明.

$$\sum_{l=0}^{n}\binom{l}{k}=\binom{n+1}{k+1} \tag{10}$$

通过组合分析一一考虑 $S=\{a_1,a_2,\cdots,a_{n+1}\}$ 的 $k+1$ 子集数可以得证, 在恒等式证明中比较常用. 被称为 朱世杰恒等式[2].

$$\binom{n}{r}\binom{r}{k}=\binom{n}{k}\binom{n-k}{r-k} \tag{11}$$

通过定义可以证明.

$$\sum_{i=0}^{n}\binom{n-i}{i}=F_{n+1} \tag{12}$$

其中 F 是斐波那契数列.

$$\binom{n+k}{k}^2=\sum_{j=0}^{k}\binom{k}{j}^2\binom{n+2k-j}{2k} \tag{13}$$

通过 (6) 可以证明. 被称为 李善兰恒等式[3].

二项式反演

记 f_n 表示恰好使用 n 个不同元素形成特定结构的方案数，g_n 表示从 n 个不同元素中选出 $i\geq 0$ 个元素形成特定结构的总方案数.

若已知 f_n 求 g_n，那么显然有:

$$g_n=\sum_{i=0}^{n}\binom{n}{i}f_i$$

若已知 g_n 求 f_n，那么:

$$f_n=\sum_{i=0}^{n}\binom{n}{i}(-1)^{n-i}g_i$$

上述已知 g_n 求 f_n 的过程，就称为 **二项式反演**.

证明

将反演公式的 g_i 展开得到:

$$\begin{aligned}f_n&=\sum_{i=0}^{n}\binom{n}{i}(-1)^{n-i}\left[\sum_{j=0}^{i}\binom{i}{j}f_j\right]\\&=\sum_{i=0}^{n}\sum_{j=0}^{i}\binom{n}{i}\binom{i}{j}(-1)^{n-i}f_j\end{aligned}$$

先枚举 j，再枚举 i，得到:

$$\begin{aligned}f_n&=\sum_{j=0}^{n}\sum_{i=j}^{n}\binom{n}{i}\binom{i}{j}(-1)^{n-i}f_j\\&=\sum_{j=0}^{n}f_j\sum_{i=j}^{n}\binom{n}{i}\binom{i}{j}(-1)^{n-i}\end{aligned}$$

使用 「组合数性质 | 二项式推论」→9.13@p1612 的公式 (11) 得到:

$$\begin{aligned}f_n&=\sum_{j=0}^{n}f_j\sum_{i=j}^{n}\binom{n}{j}\binom{n-j}{i-j}(-1)^{n-i}\\&=\sum_{j=0}^{n}\binom{n}{j}f_j\sum_{i=j}^{n}\binom{n-j}{i-j}(-1)^{n-i}\end{aligned}$$

令 $k=i-j$. 则 $i=k+j$，上式转换为:

$$f_n=\sum_{j=0}^{n}\binom{n}{j}f_j\sum_{k=0}^{n-j}\binom{n-j}{k}(-1)^{n-j-k}1^k$$

使用 「组合数性质 | 二项式推论」→9.13@p1612 的公式 (5) 得到:

$$f_n = \sum_{j=0}^{n} \binom{n}{j} f_j [n = j] = f_n$$

证毕.

外部参考资料

[1] 范德蒙恒等式  [2] 朱世杰恒等式  [3] 李善兰恒等式 

9.13.2 抽屉原理

定义

抽屉原理，亦称鸽巢原理（the pigeonhole principle）.

它常被用于证明存在性证明和求最坏情况下的解.

简单情况

将 $n+1$ 个物体，划分为 n 组，那么有至少一组有两个（或以上）的物体.

这个定理看起来比较显然，证明方法考虑反证法：假如每个分组有至多 1 个物体，那么最多有 $1 \times n$ 个物体，而实际上有 $n+1$ 个物体，矛盾.

推广

将 n 个物体，划分为 k 组，那么至少存在一个分组，含有大于或等于 $\left\lceil \frac{n}{k} \right\rceil$ 个物品.

推广的形式也可以使用反证法证明：若每个分组含有小于 $\left\lceil \frac{n}{k} \right\rceil$ 个物体，则其总和 $S \leq (\left\lceil \frac{n}{k} \right\rceil - 1) \times k = k\left\lceil \frac{n}{k} \right\rceil - k < k(\frac{n}{k} + 1) - k = n$ 矛盾.

此外，划分还可以弱化为覆盖结论不变.

给定集合 S, 一个 S 的非空子集构成的簇 $\{A_1, A_2 ... A_k\}$

- 若满足 $\bigcup_{i=1}^{k} A_i$ 则称为 S 的一个覆盖（cover）
- 若一个覆盖还满足 $i \neq j \rightarrow A_i \cap A_j = \emptyset$ 则称为 S 的一个划分.

鸽巢原理可以有如下叙述：对于 S 的一个覆盖 $\{A_1, A_2 ... A_k\}$ 有至少一个集合 A_i 满足 $|A_i| \geq \left\lceil \frac{|S|}{k} \right\rceil$.

参考文献

- Wikipedia: Pigeonhole principle[1]
- *Discrete Mathematics and Its Applications*: Chapter 6, Section 1

外部参考资料

[1] Wikipedia: Pigeonhole principle 

9.13.3 容斥原理

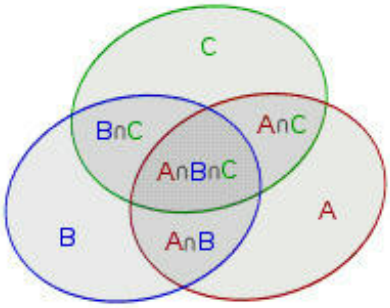
引入

> **入门例题**
>
> 假设班里有 10 个学生喜欢数学，15 个学生喜欢语文，21 个学生喜欢编程，班里至少喜欢一门学科的有多少个学生呢?

是 $10+15+21=46$ 个吗? 不是的，因为有些学生可能同时喜欢数学和语文，或者语文和编程，甚至还有可能三者都喜欢.

为了叙述方便，我们把喜欢语文、数学、编程的学生集合分别用 A,B,C 表示，则学生总数等于 $|A\cup B\cup C|$. 刚才已经讲过，如果把这三个集合的元素个数 $|A|,|B|,|C|$ 直接加起来，会有一些元素重复统计了，因此需要扣掉 $|A\cap B|,|B\cap C|,|C\cap A|$，但这样一来，又有一小部分多扣了，需要加回来，即 $|A\cap B\cap C|$. 即

$$|A\cup B\cup C|=|A|+|B|+|C|-|A\cap B|-|B\cap C|-|C\cap A|+|A\cap B\cap C|$$



把上述问题推广到一般情况，就是我们熟知的容斥原理.

定义

设 U 中元素有 n 种不同的属性，而第 i 种属性称为 P_i，拥有属性 P_i 的元素构成集合 S_i，那么

$$\left|\bigcup_{i=1}^{n}S_i\right|=\sum_i|S_i|-\sum_{i<j}|S_i\cap S_j|+\sum_{i<j<k}|S_i\cap S_j\cap S_k|-\cdots$$

$$+(-1)^{m-1}\sum_{a_i<a_{i+1}}\left|\bigcap_{i=1}^{m}S_{a_i}\right|+\cdots+(-1)^{n-1}|S_1\cap\cdots\cap S_n|$$

即

$$\left|\bigcup_{i=1}^{n}S_i\right|=\sum_{m=1}^{n}(-1)^{m-1}\sum_{a_i<a_{i+1}}\left|\bigcap_{i=1}^{m}S_{a_i}\right|$$

证明

对于每个元素使用二项式定理计算其出现的次数. 对于元素 x，假设它出现在 $T_1,T_2,\cdots,T_m$ 的集合中，那么它的出现次数为

$$Cnt=|\{T_i\}|-|\{T_i\cap T_j|\ i<j\}|+\cdots+(-1)^{k-1}\left|\left\{\bigcap_{i=1}^{k}T_{a_i}|\ a_i<a_{i+1}\right\}\right|$$

$$+\cdots+(-1)^{m-1}|\{T_1\cap\cdots\cap T_m\}|$$

$$=\binom{m}{1}-\binom{m}{2}+\cdots+(-1)^{m-1}\binom{m}{m}$$

$$=\binom{m}{0}-\sum_{i=0}^{m}(-1)^i\binom{m}{i}$$

$$=1-(1-1)^m=1$$

于是每个元素出现的次数为 1，那么合并起来就是并集. 证毕.

补集

对于全集 U 下的 **集合的并** 可以使用容斥原理计算，而集合的交则用全集减去 **补集的并集** 求得:

$$\left|\bigcap_{i=1}^{n} S_i\right| = |U| - \left|\bigcup_{i=1}^{n} \overline{S_i}\right|$$

右边使用容斥即可.

可能接触过容斥的读者都清楚上述内容，而更关心的是容斥的应用.

那么接下来我们给出 3 个层次不同的例题来为大家展示容斥原理的应用.

不定方程非负整数解计数

不定方程非负整数解计数

给出不定方程 $\sum_{i=1}^{n} x_i = m$ 和 n 个限制条件 $x_i \le b_i$，其中 $m, b_i \in \mathbb{N}$. 求方程的非负整数解的个数.

没有限制时

如果没有 $x_i \le b_i$ 的限制，那么不定方程 $\sum_{i=1}^{n} x_i = m$ 的非负整数解的数目为 $\binom{m+n-1}{n-1}$.

略证：插板法.

相当于你有 m 个球要分给 n 个盒子，允许某个盒子是空的. 这个问题不能直接用组合数解决.

于是我们再加入 $n-1$ 个球，于是问题就变成了在一个长度为 $m+n-1$ 的球序列中选择 $n-1$ 个球，然后这个 $n-1$ 个球把这个序列隔成了 n 份，恰好可以一一对应放到 n 个盒子中. 那么在 $m+n-1$ 个球中选择 $n-1$ 个球的方案数就是 $\binom{m+n-1}{n-1}$.

容斥模型

接着我们尝试抽象出容斥原理的模型：

1. 全集 U：不定方程 $\sum_{i=1}^{n} x_i = m$ 的非负整数解
2. 元素：变量 x_i.
3. 属性：x_i 的属性即 x_i 满足的条件，即 $x_i \le b_i$ 的条件

目标：所有变量满足对应属性时集合的大小，即 $|\bigcap_{i=1}^{n} S_i|$.

这个东西可以用 $\left|\bigcap_{i=1}^{n} S_i\right| = |U| - \left|\bigcup_{i=1}^{n} \overline{S_i}\right|$ 求解. $|U|$ 可以用组合数计算，后半部分自然使用容斥原理展开.

那么问题变成，对于一些 $\overline{S_{a_i}}$ 的交集求大小. 考虑 $\overline{S_{a_i}}$ 的含义，表示 $x_{a_i} \ge b_{a_i} + 1$ 的解的数目. 而交集表示同时满足这些条件. 因此这个交集对应的不定方程中，有些变量有 **下界限制**，而有些则没有限制.

能否消除这些下界限制呢? 既然要求的是非负整数解，而有些变量的下界又大于 0，那么我们直接 **把这个下界减掉**，就可以使得这些变量的下界变成 0，即没有下界啦. 因此对于

$$\left|\bigcap_{\substack{1\le i\le k\\ a_i<a_{i+1}}} S_{a_i}\right|$$

的不定方程形式为

$$\sum_{i=1}^{n} x_i = m - \sum_{i=1}^{k} (b_{a_i} + 1)$$

于是这个也可以组合数计算啦. 这个长度为 k 的 a 数组相当于在枚举子集.

HAOI2008 硬币购物

HAOI2008 硬币购物

4 种面值的硬币，第 i 种的面值是 C_i. n 次询问，每次询问给出每种硬币的数量 D_i 和一个价格 S，问付款方式.

$n \leq 10^3, S \leq 10^5$.

如果用背包做的话复杂度是 $O(4nS)$，无法承受. 这道题最明显的特点就是硬币一共只有四种. 抽象模型，其实就是让我们求方程 $\sum_{i=1}^{4} C_i x_i = S, x_i \leq D_i$ 的非负整数解的个数.

采用同样的容斥方式，x_i 的属性为 $x_i \leq D_i$. 套用容斥原理的公式，最后我们要求解

$$\sum_{i=1}^{4} C_i x_i = S - \sum_{i=1}^{k} C_{a_i}(D_{a_i} + 1)$$

也就是无限背包问题. 这个问题可以预处理，算上询问，总复杂度 $O(4S + 2^4 n)$.

代码实现

```cpp
#include <iostream>
using namespace std;
constexpr long long S = 1e5 + 5;
long long c[5], d[5], n, s;
long long f[S];

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  cin >> c[1] >> c[2] >> c[3] >> c[4] >> n;
  f[0] = 1;
  for (long long j = 1; j <= 4; j++)
    for (long long i = 1; i < S; i++)
      if (i >= c[j]) f[i] += f[i - c[j]];  // f[i]: 价格为i时的硬币组成方法数
  for (long long k = 1; k <= n; k++) {
    cin >> d[1] >> d[2] >> d[3] >> d[4] >> s;
    long long ans = 0;
    for (long long i = 1; i < 16;
         i++) {  // 容斥，因为物品一共有4种，所以从1到2^4-1=15循环
      long long m = s, bit = 0;
      for (long long j = 1; j <= 4; j++) {
        if ((i >> (j - 1)) % 2 == 1) {
          m -= (d[j] + 1) * c[j];
          bit++;
        }
      }
      if (m >= 0) ans += (bit % 2 * 2 - 1) * f[m];
    }
    cout << f[s] - ans << '\n';
  }
  return 0;
}

```

完全图子图染色问题

前面的三道题都是容斥原理的正向运用，这道题则需要用到容斥原理逆向分析.

完全图子图染色问题

A 和 B 喜欢对图（不一定连通）进行染色，而他们的规则是，相邻的结点必须染同一种颜色．今天 A 和 B 玩游戏，对于 n 阶 **完全图** $G=(V,E)$．他们定义一个估价函数 $F(S)$，其中 S 是边集，$S\subseteq E$.$F(S)$ 的值是对图 $G'=(V,S)$ 用 m 种颜色染色的总方案数．他们的另一个规则是，如果 $|S|$ 是奇数，那么 A 的得分增加 $F(S)$，否则 B 的得分增加 $F(S)$. 问 A 和 B 的得分差值.

数学形式

一看这道题的算法趋向并不明显，因此对于棘手的题目首先抽象出数学形式．得分差即为奇偶对称差，可以用 -1 的幂次来作为系数．我们求的是

$$Ans=\sum_{S\subseteq E}(-1)^{|S|-1}F(S)$$

容斥模型

相邻结点染同一种颜色，我们把它当作属性．在这里我们先不遵守染色的规则，假定我们用 m 种颜色直接对图染色．对于图 $G'=(V,S)$，我们把它当作 **元素**．**属性** $x_i=x_j$ 的含义是结点 i,j 染同色（注意，并未要求 i,j 之间有连边）.

而属性 $x_i=x_j$ 对应的 **集合** 定义为 $Q_{i,j}$，其含义是所有满足该属性的图 G' 的染色方案，集合的大小就是满足该属性的染色方案数，集合内的元素相当于所有满足该属性的图 G' 的染色图.

回到题目，「相邻的结点必须染同一种颜色」，可以理解为若干个 Q 集合的交集．因此可以写出

$$F(S)=\left|\bigcap_{(i,j)\in S}Q_{i,j}\right|$$

上述式子右边的含义就是说对于 S 内的每一条边 (i,j) 都满足 $x_i=x_j$ 的染色方案数，也就是 $F(S)$.

是不是很有容斥的味道了? 由于容斥原理本身没有二元组的形式，因此我们把 **所有** 的边 (i,j) 映射到 $T=\frac{n(n+1)}{2}$ 个整数上，假设将 (i,j) 映射为 $k,1\le k\le T$，同时 $Q_{i,j}$ 映射为 Q_k. 那么属性 $x_i=x_j$ 则定义为 P_k.

同时 S 可以表示为若干个 k 组成的集合，即 $S\iff K=\{k_1,k_2,\cdots,k_m\}$.（也就是说我们在边集与数集间建立了等价关系）.

而 E 对应集合 $M=\left\{1,2,\cdots,\frac{n(n+1)}{2}\right\}$. 于是乎

$$F(S)\iff F(\{k_i\})=\left|\bigcap_{k_i}Q_{k_i}\right|$$

逆向分析

那么要求的式子展开

$$\begin{aligned}Ans&=\sum_{K\subseteq M}(-1)^{|K|-1}\left|\bigcap_{k_i\in K}Q_{k_i}\right|\\&=\sum_i|Q_i|-\sum_{i<j}|Q_i\cap Q_j|+\sum_{i<j<k}|Q_i\cap Q_j\cap Q_k|-\cdots+(-1)^{T-1}\left|\bigcap_{i=1}^{T}Q_i\right|\end{aligned}$$

于是就出现了容斥原理的展开形式，因此对这个式子逆向推导

$$Ans=\left|\bigcup_{i=1}^{T}Q_i\right|$$

再考虑等式右边的含义，只要满足 $1\sim T$ 任一条件即可，也就是存在两个点同色（不一定相邻）的染色方案数！而我们知道染色方案的全集是 U，显然 $|U|=m^n$. 而转化为补集，就是求两两异色的染色方案数，即 $A_m^n=\frac{m!}{(m-n)!}$. 因此

$$Ans=m^n-A_m^n$$

解决这道题，我们首先抽象出题目数学形式，然后从题目中信息量最大的条件，$F(S)$ 函数的定义入手，将其转化为集合的交并补．然后将式子转化为容斥原理的形式，并 **逆向推导** 出最终的结果．这道题体现的正是容斥原理的逆用.

数论中的容斥

使用容斥原理能够巧妙地求解一些数论问题.

容斥原理求最大公约数为 k 的数对个数

考虑下面的问题:

> **求最大公约数为 k 的数对个数**
>
> 设 $1\le x,y\le N$，$f(k)$ 表示最大公约数为 k 的有序数对 (x,y) 的个数，求 $f(1)$ 到 $f(N)$ 的值.

这道题固然可以用欧拉函数或莫比乌斯反演的方法来做，但是都不如用容斥原理来的简单.

由容斥原理可以得知，先找到所有以 k 为 **公约数** 的数对，再从中剔除所有以 k 的倍数为 **公约数** 的数对，余下的数对就是以 k 为 **最大公约数** 的数对．即 $f(k)=$ 以 k 为 **公约数** 的数对个数 $-$ 以 k 的倍数为 **公约数** 的数对个数.

进一步可发现，以 k 的倍数为 **公约数** 的数对个数等于所有以 k 的倍数为 **最大公约数** 的数对个数之和．于是，可以写出如下表达式:

$$f(k)=\lfloor(N/k)\rfloor^2-\sum_{i=2}^{i*k\le N}f(i*k)$$

由于当 $k>N/2$ 时，我们可以直接算出 $f(k)=\lfloor(N/k)\rfloor^2$，因此我们可以倒过来，从 $f(N)$ 算到 $f(1)$ 就可以了．于是，我们使用容斥原理完成了本题.

```
for (long long k = N; k >= 1; k--) {
  f[k] = (N / k) * (N / k);
  for (long long i = k + k; i <= N; i += k) f[k] -= f[i];
}
```

上述方法的时间复杂度为 $O(\sum_{i=1}^{N}N/i)=O(N\sum_{i=1}^{N}1/i)=O(N\log N)$.

附赠三倍经验供大家练手.

- Luogu P2398 GCD SUM[1]
- Luogu P2158[SDOI2008] 仪仗队[2]
- Luogu P1447[NOI2010] 能量采集[3]

容斥原理推导欧拉函数

考虑下面的问题:

> **欧拉函数公式**
>
> 求欧拉函数 $\varphi(n)$．其中 $\varphi(n)=|\{1\le x\le n\mid\gcd(x,n)=1\}|$.

直接计算是 $O(n\log n)$ 的，用线性筛是 $O(n)$ 的，杜教筛是 $O(n^{\frac{2}{3}})$ 的（话说一道数论入门题用容斥做为什么还要扯到杜教筛上），接下来考虑用容斥推出欧拉函数的公式

判断两个数是否互质，首先分解质因数

$$n=\prod_{i=1}^{k}p_i^{c_i}$$

那么就要求对于任意 p_i，x 都不是 p_i 的倍数，即 $p_i\nmid x$. 把它当作属性，对应的集合为 S_i，因此有

$$\varphi(n)=\left|\bigcap_{i=1}^{k}S_i\right|=|\ U\ |-\left|\bigcup_{i=1}^{k}\overline{S_i}\right|$$

全集大小 $|\ U\ |=n$，而 $\overline{S_i}$ 表示的是 $p_i\mid x$ 构成的集合，显然 $|\ \overline{S_i}|=\frac{n}{p_i}$，并由此推出

$$\left|\bigcap_{a_i<a_{i+1}}S_{a_i}\right|=\frac{n}{\prod p_{a_i}}$$

因此可得

$$\begin{aligned}\varphi(n)&=n-\sum_i\frac{n}{p_i}+\sum_{i<j}\frac{n}{p_ip_j}-\cdots+(-1)^k\frac{n}{p_1p_2\cdots p_k}\\&=n\left(1-\frac{1}{p_1}\right)\left(1-\frac{1}{p_2}\right)\cdots\left(1-\frac{1}{p_k}\right)\\&=n\prod_{i=1}^{k}\left(1-\frac{1}{p_i}\right)\end{aligned}$$

这就是欧拉函数的数学表示啦

容斥原理一般化

容斥原理常用于集合的计数问题，而对于两个集合的函数 $f(S),g(S)$，若

$$f(S)=\sum_{T\subseteq S}g(T)$$

那么就有

$$g(S)=\sum_{T\subseteq S}(-1)^{|\ S\ |-|\ T\ |}f(T)$$

证明

接下来我们简单证明一下．我们从等式的右边开始推：

$$\begin{aligned}&\sum_{T\subseteq S}(-1)^{|\ S\ |-|\ T\ |}f(T)\\&=\sum_{T\subseteq S}(-1)^{|\ S\ |-|\ T\ |}\sum_{Q\subseteq T}g(Q)\\&=\sum_{Q}g(Q)\sum_{Q\subseteq T\subseteq S}(-1)^{|\ S\ |-|\ T\ |}\end{aligned}$$

我们发现后半部分的求和与 Q 无关，因此把后半部分的 Q 剔除：

$$=\sum_{Q}g(Q)\sum_{T\subseteq(S\setminus Q)}(-1)^{|\ S\setminus Q\ |-|\ T\ |}$$

记关于集合 P 的函数 $F(P)=\sum_{T\subseteq P}(-1)^{|\ P\ |-|\ T\ |}$，并化简这个函数：

$$\begin{aligned}F(P)&=\sum_{T\subseteq P}(-1)^{|\ P\ |-|\ T\ |}\\&=\sum_{i=0}^{|\ P\ |}\binom{|\ P\ |}{i}(-1)^{|\ P\ |-i}=\sum_{i=0}^{|\ P\ |}\binom{|\ P\ |}{i}1^i(-1)^{|\ P\ |-i}\\&=(1-1)^{|\ P\ |}=0^{|\ P\ |}\end{aligned}$$

因此原来的式子的值是

$$\sum_{Q}g(Q)\sum_{T\subseteq(S\setminus Q)}(-1)^{|\ S\setminus Q\ |-|\ T\ |}=\sum_{Q}g(Q)F(S\setminus Q)=\sum_{Q}g(Q)\cdot 0^{|\ S\setminus Q\ |}$$

分析发现，仅当 $|S \setminus Q| = 0$ 时有 $0^0 = 1$，这时 $Q = S$，对答案的贡献就是 $g(S)$，其他时候 $0^{|S \setminus Q|} = 0$，则对答案无贡献．于是得到

$$\sum_Q g(Q) \cdot 0^{|S \setminus Q|} = g(S)$$

综上所述，得证．

推论

该形式还有这样一个推论．在全集 U 下，对于函数 $f(S), g(S)$，如果

$$f(S) = \sum_{S \subseteq T} g(T)$$

那么

$$g(S) = \sum_{S \subseteq T} (-1)^{|T| - |S|} f(T)$$

这个推论其实就是补集形式，证法类似．

DAG 计数

DAG 计数

对 n 个点带标号的有向无环图进行计数，对 $10^9 + 7$ 取模．$n \leq 5 \times 10^3$．

直接 DP

考虑 DP，定义 $f[i, j]$ 表示 i 个点的 DAG，有 j 点个入度为 0 的图的个数．假设去掉这 j 个点后，有 k 个点入度为 0，那么在去掉前这 k 个点至少与这 j 个点中的某几个有连边，即 $2^j - 1$ 种情况；而这 j 个点除了与 k 个点连边，还可以与剩下的点任意连边，有 2^{i-j-k} 种情况．因此方程如下：

$$f[i, j] = \binom{i}{j} \sum_{k=1}^{i-j} (2^j - 1)^k 2^{(i-j-k)j} f[i - j, k]$$

计算上式的复杂度是 $O(n^3)$ 的．

放宽限制

上述 DP 的定义是恰好 j 个点入度为 0，太过于严格，可以放宽为至少 j 个点入度为 0．直接定义 $f[i]$ 表示 i 个点的 DAG 个数．可以直接容斥．考虑选出的 j 个点，这 j 个点可以和剩下的 $i - j$ 个点有任意的连边，即 $\left(2^{i-j}\right)^j = 2^{(i-j)j}$ 种情况：

$$f[i] = \sum_{j=1}^{i} (-1)^{j-1} \binom{i}{j} 2^{(i-j)j} f[i - j]$$

计算上式的复杂度是 $O(n^2)$ 的．

Min-max 容斥

对于满足 全序→9.20@p1951 关系并且其中元素满足可加减性的序列 $\{x_i\}$，设其长度为 n，并设 $S = \{1, 2, 3, \cdots, n\}$，则有：

$$\max_{i \in S} x_i = \sum_{T \subseteq S} (-1)^{|T| - 1} \min_{j \in T} x_j$$

$$\min_{i \in S} x_i = \sum_{T \subseteq S} (-1)^{|T| - 1} \max_{j \in T} x_j$$

证明： 考虑做一个到一般容斥原理的映射．对于 $x \in S$，假设 x 是第 k 小的元素．那么我们定义一个映射 $f : x \mapsto \{1, 2, \cdots, k\}$．显然这是一个双射．

那么容易发现，对于 $x,y\in S$，$f(\min(x,y))=f(x)\cap f(y)$，$f(\max(x,y))=f(x)\cup f(y)$. 因此我们得到：

$$\begin{aligned}\left|f\left(\max_{i\in S}x_i\right)\right|&=\left|\bigcup_{i\in S}f(x_i)\right|\\&=\sum_{T\subseteq S}(-1)^{|T|-1}\left|\bigcap_{j\in T}f(x_j)\right|\\&=\sum_{T\subseteq S}(-1)^{|T|-1}\left|f\left(\min_{j\in T}x_j\right)\right|\end{aligned}$$

然后再把 $|f(\max_{i\in S}x_i)|$ 映射回 $\max_{i\in S}x_i$，而 min 是类似的.

证毕.

但是你可能觉得这个式子非常蠢，最大值明明可以直接求. 之所以 min-max 容斥这么重要，是因为它在期望上也是成立的，即：

$$E\left(\max_{i\in S}x_i\right)=\sum_{T\subseteq S}(-1)^{|T|-1}E\left(\min_{j\in T}x_j\right)$$

$$E\left(\min_{i\in S}x_i\right)=\sum_{T\subseteq S}(-1)^{|T|-1}E\left(\max_{j\in T}x_j\right)$$

证明： 我们考虑计算期望的一种方法：

$$E\left(\max_{i\in S}x_i\right)=\sum_y P(y=x)\max_{j\in S}y_j$$

其中 y 是一个长度为 n 的序列.

我们对后面的 max 使用之前的式子：

$$\begin{aligned}E\left(\max_{i\in S}x_i\right)&=\sum_y P(y=x)\max_{j\in S}y_j\\&=\sum_y P(y=x)\sum_{T\subseteq S}(-1)^{|T|-1}\min_{j\in T}y_j\end{aligned}$$

调换求和顺序：

$$\begin{aligned}E\left(\max_{i\in S}x_i\right)&=\sum_y P(y=x)\sum_{T\subseteq S}(-1)^{|T|-1}\min_{j\in T}y_j\\&=\sum_{T\subseteq S}(-1)^{|T|-1}\sum_y P(y=x)\min_{j\in T}y_j\\&=\sum_{T\subseteq S}(-1)^{|T|-1}E\left(\min_{j\in T}y_j\right)\end{aligned}$$

min 是类似的.

证毕.

还有更强的：

$$\operatorname*{kthmax}_{i\in S}x_i=\sum_{T\subseteq S}(-1)^{|T|-k}\binom{|T|-1}{k-1}\min_{j\in T}x_j$$

$$\operatorname*{kthmin}_{i\in S}x_i=\sum_{T\subseteq S}(-1)^{|T|-k}\binom{|T|-1}{k-1}\max_{j\in T}x_j$$

$$E\left(\operatorname*{kthmax}_{i\in S}x_i\right)=\sum_{T\subseteq S}(-1)^{|T|-k}\binom{|T|-1}{k-1}E\left(\min_{j\in T}x_j\right)$$

$$E\left(\operatorname*{kthmin}_{i\in S}x_i\right)=\sum_{T\subseteq S}(-1)^{|T|-k}\binom{|T|-1}{k-1}E\left(\max_{j\in T}x_j\right)$$

规定若 $n<m$，则 $\binom{n}{m}=0$.

证明： 不妨设 $\forall 1\le i<n,x_i\le x_{i+1}$. 则有：

$$\begin{aligned}\sum_{T\subseteq S}(-1)^{|T|-k}\binom{|T|-1}{k-1}\min_{j\in T}x_j&=\sum_{i\in S}x_i\sum_{T\subseteq S}(-1)^{|T|-k}\binom{|T|-1}{k-1}\left[x_i=\min_{j\in T}x_j\right]\\&=\sum_{i\in S}x_i\sum_{j=k}^{n}\binom{n-i}{j-1}\binom{j-1}{k-1}(-1)^{j-k}\end{aligned}$$

又因为有组合恒等式：$\binom{a}{b}\binom{b}{c}=\binom{a}{c}\binom{a-c}{b-c}$，所以有：

$$\begin{aligned}\sum_{T\subseteq S}(-1)^{|T|-k}\binom{|T|-1}{k-1}\min_{j\in T}x_j&=\sum_{i\in S}x_i\sum_{j=k}^{n}\binom{n-i}{j-1}\binom{j-1}{k-1}(-1)^{j-k}\\&=\sum_{i\in S}x_i\sum_{j=k}^{n}\binom{n-i}{k-1}\binom{n-i-k+1}{j-k}(-1)^{j-k}\\&=\sum_{i\in S}\binom{n-i}{k-1}x_i\sum_{j=k}^{n}\binom{n-i-k+1}{j-k}(-1)^{j-k}\\&=\sum_{i\in S}\binom{n-i}{k-1}x_i\sum_{j=0}^{n-i-k+1}\binom{n-i-k+1}{j}(-1)^{j}\end{aligned}$$

当 $i=n-k+1$ 时：

$$\binom{n-i}{k-1}\sum_{j=0}^{n-i-k+1}\binom{n-i-k+1}{j}(-1)^j=1$$

否则：

$$\binom{n-i}{k-1}\sum_{j=0}^{n-i-k+1}\binom{n-i-k+1}{j}(-1)^j=0$$

所以：

$$\sum_{i\in S}\binom{n-i}{k-1}x_i\sum_{j=0}^{n-i-k+1}\binom{n-i-k+1}{j}(-1)^j=\operatorname*{kthmax}_{i\in S}x_i$$

剩下三个是类似的.

证毕.

根据 min-max 容斥，我们还可以得到下面的式子：

$$\operatorname*{lcm}_{i\in S}x_i=\prod_{T\subseteq S}\left(\gcd_{j\in T}x_j\right)^{(-1)^{|T|-1}}$$

因为 $\operatorname{lcm},\gcd,a^1,a^{-1}$ 分别相当于 $\max,\min,+,-$，就是说相当于对于指数做了一个 min-max 容斥，自然就是对的了

PKUWC2018 随机游走

PKUWC2018 随机游走[4]

给定一棵 n 个点的树，你从 x 出发，每次等概率随机选择一条与所在点相邻的边走过去.

有 Q 次询问. 每次询问给出一个集合 S，求如果从 x 出发一直随机游走，直到点集 S 中的点都至少经过一次的话，期望游走几步.

> 特别地，点 x（即起点）视为一开始就被经过了一次.
>
> 对 998244353 取模.
>
> $1 \le n \le 18, 1 \le Q \le 5000, 1 \le |S| \le n$.

期望游走的步数也就是游走的时间．那么设随机变量 x_i 表示第一次走到结点 i 的时间．那么我们要求的就是

$$E\left(\max_{i\in S} x_i\right)$$

使用 min-max 容斥可以得到

$$E\left(\max_{i\in S} x_i\right) = E\left(\sum_{T\subseteq S}(-1)^{|T|-1}\min_{i\in T} x_i\right) = \sum_{T\subseteq S}(-1)^{|T|-1}E\left(\min_{i\in T} x_i\right)$$

对于一个集合 $T \in [n]$，考虑求出 $F(T) = E(\min_{i\in T} x_i)$.

考虑 $E(\min_{i\in T} x_i)$ 的含义，是第一次走到 T 中某一个点的期望时间．不妨设 $f(i)$ 表示从结点 i 出发，第一次走到 T 中某个结点的期望时间.

- 对于 $i \in T$，有 $f(i) = 0$.
- 对于 $i \notin T$，有 $f(i) = 1 + \frac{1}{\deg(i)}\sum_{(i,j)\in E} f(j)$.

如果直接高斯消元，复杂度 $O(n^3)$．那么我们对每个 T 都计算 $F(T)$ 的总复杂度就是 $O(2^n n^3)$，不能接受．我们使用树上消元的技巧.

不妨设根结点是 1，结点 u 的父亲是 p_u．对于叶子结点 i，$f(i)$ 只会和 i 的父亲有关（也可能 $f(i) = 0$，那样更好）．因此我们可以把 $f(i)$ 表示成 $f(i) = A_i + B_i f(p_i)$ 的形式，其中 A_i, B_i 可以快速计算.

对于非叶结点 i，考虑它的儿子序列 $j_1, \cdots, j_k$．由于 $f(j_e) = A_{j_e} + B_{j_e} f(i)$．因此可以得到

$$f(i) = 1 + \frac{1}{\deg(i)}\sum_{e=1}^{k}\left(A_{j_e} + B_{j_e} f(i)\right) + \frac{f(p_i)}{\deg(i)}$$

那么变换一下可以得到

$$f(i) = \frac{\deg(i) + \sum_{e=1}^{k} A_{j_e}}{\deg(i) - \sum_{e=1}^{k} B_{j_e}} + \frac{f(p_i)}{\deg(i) - \sum_{e=1}^{k} B_{j_e}}$$

于是我们把 $f(i)$ 也写成了 $A_i + B_i f(p_i)$ 的形式．这样可以一直倒推到根结点．而根结点没有父亲．也就是说

$$f(1) = \frac{\deg(1) + \sum_{e=1}^{k} A_{j_e}}{\deg(1) - \sum_{e=1}^{k} B_{j_e}}$$

解一下这个方程我们就得到了 $f(1)$，再从上往下推一次就得到了每个点的 $f(i)$．那么 $F(T) = f(x)$．时间复杂度 $O(n)$.

这样，我们可以对于每一个 T 计算出 $F(T)$，时间复杂度 $O(2^n n)$.

回到容斥的部分，我们知道 $E(\max_{i\in S} x_i) = \sum_{T\subseteq S}(-1)^{|T|-1}F(T)$.

不妨设 $F'(T) = (-1)^{|T|-1}F(T)$，那么进一步得到 $E(\max_{i\in S} x_i) = \sum_{T\subseteq S} F'(T)$．因此可以使用 FMT（也叫子集前缀和，或者 FWT 或变换）在 $O(2^n n)$ 的时间内对每个 S 计算出 $E(\max_{i\in S} x_i)$，这样就可以 $O(1)$ 回答询问了.

习题

- ABC331- G - Collect Them All[5]
- 洛谷 P4707 重返现世[6]

参考文献

浅探容斥原理 - 王迪[7]，2013 年信息学奥林匹克中国国家队候选队员论文集

有标号的 DAG 计数系列问题 - Cyhlnj[8]

全序关系 - Wikipedia[9]

外部参考资料

[1] Luogu P2398 GCD SUM

[2] Luogu P2158[SDOI2008] 仪仗队

[3] Luogu P1447[NOI2010] 能量采集

[4] PKUWC2018 随机游走

[5] ABC331- G - Collect Them All

[6] 洛谷 P4707 重返现世

[7] 浅探容斥原理 - 王迪

[8] 有标号的 DAG 计数系列问题 - Cyhlnj

[9] 全序关系 - Wikipedia

9.13.4 斐波那契数列

斐波那契数列（The Fibonacci sequence，OEIS A000045[1]）的定义如下：

$$F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2}$$

该数列的前几项如下：

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \ldots$$

卢卡斯数列

卢卡斯数列（The Lucas sequence，OEIS A000032[2]）的定义如下：

$$L_0 = 2, L_1 = 1, L_n = L_{n-1} + L_{n-2}$$

该数列的前几项如下：

$$2, 1, 3, 4, 7, 11, 18, 29, 47, 76, 123, 199, \ldots$$

研究斐波那契数列，很多时候需要借助卢卡斯数列为工具.

斐波那契数列通项公式

第 n 个斐波那契数可以在 $\Theta(n)$ 的时间内使用递推公式计算. 但我们仍有更快速的方法计算.

解析解

解析解即公式解. 我们有斐波那契数列的通项公式（Binet's Formula）：

$$F_n = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n}{\sqrt{5}}$$

这个公式可以很容易地用归纳法证明，当然也可以通过生成函数的概念推导，或者解一个方程得到.

当然你可能发现，这个公式分子的第二项总是小于 1，并且它以指数级的速度减小. 因此我们可以把这个公式写成

$$F_n = \left[\frac{\left(\frac{1+\sqrt{5}}{2}\right)^n}{\sqrt{5}}\right]$$

这里的中括号表示取离它最近的整数.

这两个公式在计算的时候要求极高的精确度，因此在实践中很少用到. 但是请不要忽视！结合模意义下二次剩余和逆元的概念，在 OI 中使用这个公式仍是有用的.

卢卡斯数列通项公式

我们有卢卡斯数列的通项公式：

$$L_n = \left(\frac{1+\sqrt{5}}{2}\right)^n + \left(\frac{1-\sqrt{5}}{2}\right)^n$$

与斐波那契数列非常相似．事实上有：

$$\frac{L_n + F_n\sqrt{5}}{2} = \left(\frac{1+\sqrt{5}}{2}\right)^n$$

也就是说，L_n 和 F_n 恰好构成 $\left(\frac{1+\sqrt{5}}{2}\right)^n$ 二项式展开再合并同类项后的分子系数．也就是说，Pell 方程

$$x^2 - 5y^2 = -4$$

的全体解，恰好是

$$\frac{x_n + y_n\sqrt{5}}{2} = \frac{L_n + F_n\sqrt{5}}{2}$$

恰好是卢卡斯数列和斐波那契数列．因此有

$$L_n^2 - 5F_n^2 = -4$$

矩阵形式

斐波那契数列的递推可以用矩阵乘法的形式表达：

$$[F_{n-1}\ \ F_n] = [F_{n-2}\ \ F_{n-1}][0\ \ 11\ \ 1]$$

设 $P = [0\ \ 11\ \ 1]$，我们得到

$$[F_n\ \ F_{n+1}] = [F_0\ \ F_1]P^n$$

于是我们可以用矩阵乘法在 $\Theta(\log n)$ 的时间内计算斐波那契数列．此外，前一节讲述的公式也可通过矩阵对角化的技巧来得到．

快速倍增法

使用上面的方法我们可以得到以下等式：

$$F_{2k} = F_k(2F_{k+1} - F_k)$$

$$F_{2k+1} = F_{k+1}^2 + F_k^2$$

于是可以通过这样的方法快速计算两个相邻的斐波那契数（常数比矩乘小）．代码如下，返回值是一个二元组 (F_n, F_{n+1})．

```
pair<int, int> fib(int n) {
  if (n == 0) return {0, 1};
  auto p = fib(n >> 1);
  int c = p.first * (2 * p.second - p.first);
  int d = p.first * p.first + p.second * p.second;
  if (n & 1)
    return {d, c + d};
  else
    return {c, d};
}
```

性质

斐波那契数列拥有许多有趣的性质，这里列举出一部分简单的性质：

1. 卡西尼性质（Cassini's identity）：$F_{n-1}F_{n+1} - F_n^2 = (-1)^n$．
2. 附加性质：$F_{n+k} = F_kF_{n+1} + F_{k-1}F_n$．
3. 取上一条性质中 $k = n$，我们得到 $F_{2n} = F_n(F_{n+1} + F_{n-1})$．

4. 由上一条性质可以归纳证明，$\forall k \in \mathbb{N}, F_n \mid F_{nk}$.
5. 上述性质可逆，即 $\forall F_a \mid F_b, a \mid b$.
6. GCD 性质：$(F_m, F_n) = F_{(m,n)}$.
7. 以斐波那契数列相邻两项作为输入会使欧几里德算法达到最坏复杂度（具体参见 维基 - 拉梅[3]）.

斐波那契数列与卢卡斯数列的关系

不难发现，关于卢卡斯数列与斐波那契数列的等式，与三角函数公式具有很高的相似性．比如：

$$\frac{L_n + F_n\sqrt{5}}{2} = \left(\frac{1+\sqrt{5}}{2}\right)^n$$

与

$$\cos nx + i \sin nx = (\cos x + i \sin x)^n$$

很像．以及

$$L_n^2 - 5F_n^2 = -4$$

与

$$\cos^2 x + \sin^2 x = 1$$

很像．因此，卢卡斯数列与余弦函数很像，而斐波那契数列与正弦函数很像．比如，根据

$$\left(\frac{1+\sqrt{5}}{2}\right)^m \left(\frac{1+\sqrt{5}}{2}\right)^n = \left(\frac{1+\sqrt{5}}{2}\right)^{m+n}$$

可以得到两下标之和的等式：

$$2L_{m+n} = 5F_mF_n + L_mL_n$$
$$2F_{m+n} = F_mL_n + L_mF_n$$

于是推论就有二倍下标的等式：

$$L_{2n} = L_n^2 - 2(-1)^n$$
$$F_{2n} = F_nL_n$$

这也是一种快速倍增下标的办法．同样地，也可以仿照三角函数的公式，比如奇偶性、和差化积、积化和差、半角、万能代换等等，推理出更多有关卢卡斯数列与斐波那契数列的相应等式.

斐波那契编码

我们可以利用斐波那契数列为正整数编码．根据 齐肯多夫定理[4]，任何自然数 n 可以被唯一地表示成一些斐波那契数的和：

$$N = F_{k_1} + F_{k_2} + ... + F_{k_r}$$

并且 $k_1 \geq k_2 + 2, k_2 \geq k_3 + 2, ..., k_r \geq 2$（即不能使用两个相邻的斐波那契数）

于是我们可以用 $d_0d_1d_2...d_s1$ 的编码表示一个正整数，其中 $d_i = 1$ 则表示 F_{i+2} 被使用．编码末位我们强制给它加一个 1（这样会出现两个相邻的 1），表示这一串编码结束．举几个例子：

$$\begin{aligned}
1 &= 1 &&= F_2 &&= (11)_F \\
2 &= 2 &&= F_3 &&= (011)_F \\
6 &= 5 + 1 &&= F_5 + F_2 &&= (10011)_F \\
8 &= 8 &&= F_6 &&= (000011)_F \\
9 &= 8 + 1 &&= F_6 + F_2 &&= (100011)_F \\
19 &= 13 + 5 + 1 &&= F_7 + F_5 + F_2 &&= (1001011)_F
\end{aligned}$$

给 n 编码的过程可以使用贪心算法解决：

1. 从大到小枚举斐波那契数 F_i，直到 $F_i \leq n$.
2. 把 n 减掉 F_i，在编码的 $i-2$ 的位置上放一个 1（编码从左到右以 0 为起点）.

3. 如果 n 为正，回到步骤 1.
4. 最后在编码末位添加一个 1，表示编码的结束位置.

解码过程同理，先删掉末位的 1，对于编码为 1 的位置 i（编码从左到右以 0 为起点），累加一个 F_{i+2} 到答案. 最后的答案就是原数字.

模意义下周期性

对于模 m 意义下的斐波那契数列，可以容易地使用抽屉原理证明，该数列是有周期性的. 由于斐波那契数每一项的计算都依赖于前两项的取值，所以需要用相邻斐波那契数组成的数对描述数列当且所处的状态. 考虑模意义下前 m^2+1 个斐波那契数对：

$$(F_0, F_1), (F_1, F_2), ..., (F_{m^2}, F_{m^2+1})$$

模 m 的剩余系大小为 m，这意味着至多只可能有 m^2 种互不相同的数对. 因此，在前 m^2+1 个数对中必有两个相同的数对，于是从这两个数对可以往后生成相同的斐波那契数列. 那么，斐波那契数列就是周期性的，且（最小正）周期不会超过 m^2.

Pisano 周期

模 m 意义下斐波那契数列的最小正周期被称为 **Pisano 周期**（Pisano period，皮萨诺周期，OEIS A001175 [5]）. 本文中用 $\pi(m)$ 表示模 m 的 Pisano 周期.

这一观察可以用于计算第 n 项斐波那契数模 m 的值. 如果 n 非常大，就需要计算斐波那契数模 m 的周期. 当然，只需要计算周期，不一定是最小正周期.

为此，有如下结论：

1. 对于互素的模数 m_1, m_2，有 $\pi(m_1 m_2) = \operatorname{lcm}(\pi(m_1), \pi(m_2))$.
2. 对于素数 p 和正整数 e，有 $\pi(p^e) \mid p^{e-1}\pi(p)$.
3. 对于 $m = 2^e$ $(e \in \mathbf{N}_+)$，有 $\pi(m) = 3 \cdot 2^{e-1}$.
4. 对于 $m = 5^e$ $(e \in \mathbf{N}_+)$，有 $\pi(m) = 4 \cdot 5^e$.
5. 最后，对于素数 $p \equiv \pm 1 \pmod{10}$，有 $\pi(p) \mid (p-1)$；对于素数 $p \equiv \pm 3 \pmod{10}$，有 $\pi(p) \mid 2(p+1)$.

综合这些情形，可以说明：模 m 的 Pisano 周期不会超过 $6m$，等号当且仅当 $m = 2 \times 5^e$ $(e \in \mathbf{N}_+)$ 时取得.

利用上述结论，可以基于素因数分解算法，得到如下快速计算 Pisano 周期的方法：

参考代码

```cpp
// Get a period of Fibonacci sequence mod m.
// Not necessarily be the exact Pisano period.
uint32_t calc_cycle_from_mod(uint32_t m) {
  uint32_t res = 1;
  for (auto pe : factorize(m)) {
    auto p = pe.first;
    auto e = pe.second;
    uint64_t cur = pow(p, e - 1);
    if (p == 2) {
      cur *= 3;
    } else if (p == 5) {
      cur *= 20;
    } else if (p % 5 == 1 || p % 5 == 4) {
      cur *= p - 1;
    } else {
      cur *= 2 * (p + 1);
```

```
    }
    res = lcm(res, cur);
  }
  return res;
}

```

这样得到的周期可能只是 Pisano 周期的一个倍数. 要得到精确的 Pisano 周期, 可以进一步考察该周期的因数; 或者, 可以直接通过 BSGS 算法→9.11.19@p1286 以 $O(\sqrt{m})$ 的时间复杂度计算.

证明

最后, 本文简要证明上述关于 Pisano 周期的结论. 值得说明的是, 利用下文说明的方法, 类似的结论可以推广到一般的二阶常系数线性齐次递推数列. 尽管具体的常数有所差异, 这些数列模 m 的 Pisano 周期都是 $O(m)$ 的.

第一个观察是: 利用 中国剩余定理→9.11.12@p1236, 可以将讨论限制在素数幂模的情形. 设 m_1, m_2 是两个互素的模数. 斐波那契数列在模 m_1 下的周期是 $\pi(m_1)$ 及其倍数, 在模 m_2 下的周期是 $\pi(m_2)$ 及其倍数, 所以它在模 $m_1 m_2$ 下的最小正周期则恰为 $\pi(m_1)$ 和 $\pi(m_2)$ 的最小公倍数. 这就是前文的结论 1.

另一个观察是: 模 m 下的 Pisano 周期, 其实是最小的正整数 k, 使得

$$A^k = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^k \equiv I \pmod{m}.$$

也就是说, 它其实是矩阵 A 在模 m 下[1]的 阶→9.16.2@p1812.

对于素数幂模 $m = p^e$ 的情形, 可以通过经典的升幂论证联系到相应的素数模的情形. 设 $k = \pi(p^e)$, 就存在二阶方阵 Λ, 使得

$$A^k = p^e \Lambda + I$$

成立. 故而, 由 二项式定理→9.13.1@p1612 可知

$$A^{kp} = (p^e \Lambda + I)^p = I + \sum_{i=1}^{p} \binom{p}{i} (p^e \Lambda)^i \equiv I \pmod{p^{e+1}}.$$

因此, 由 阶的性质→9.11.18@p1275, 有 $\pi(p^{e+1}) \mid kp = p\pi(p^e)$. 对 e 归纳可知, $\pi(p^e) \mid p^{e-1}\pi(p)$ 总是成立.

对于素数模 p 的情形, 本文讨论两种证明方式.

- 利用通项公式　　一种是利用斐波那契数列的通项公式:

$$F_n = \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^n - \frac{1}{\sqrt{5}} \left(\frac{1-\sqrt{5}}{2} \right)^n.$$

将它用二项式定理展开, 并消去根式项:

$$F_n = \frac{1}{2^{n-1}} \sum_{i=0}^{\lfloor (n-1)/2 \rfloor} \binom{n}{2i+1} 5^i.$$

对于 $p = 2$, 这一表达式无法直接取模, 但可以验证对应的 Pisano 周期为 $\pi(2) = 3$. 对于 $p = 5$, 有 $F_n \equiv n \cdot 3^{n-1} \pmod{p}$, 可以直接验证对应的 Pisano 周期为 $\pi(5) = 20$. 对于剩余的奇素模数, 可以分为两种情形:

  - 如果 $p \equiv 1, 4 \pmod{5}$, 就有

[1]严格来说, 它是矩阵 A 在一般线性群 $GL_2(\mathbf{Z}_m)$ 中的阶.

$$F_p \equiv \frac{1}{2^{p-1}}\binom{p}{p}5^{(p-1)/2} \equiv 1 \pmod p,$$

$$F_{p+1} \equiv \frac{1}{2^p}\left(\binom{p+1}{1} + \binom{p+1}{p}5^{(p-1)/2}\right) \equiv 1 \pmod p.$$

化简过程中，利用了如下结论：由 Lucas 定理→9.11.15@p1251，对于 $0<k<p$ 都有 $\binom{p}{k} \equiv 0 \pmod p$，而对于 $1<k<p$ 都有 $\binom{p+1}{k} \equiv 0 \pmod p$；由 Fermat 小定理→9.11.9@p1223，有 $2^{p-1} \equiv 5^{p-1} \equiv 1 \pmod p$；对于 $p \equiv 1,4 \pmod 5$，都有 p 是模 5 的二次剩余，利用 二次互反律→9.11.17@p1263，也有 5 是模 p 的二次剩余，故而 $5^{(p-1)/2} \equiv 1 \pmod p$. 由此，有 $(F_p, F_{p+1}) \equiv (F_1, F_2) \pmod p$，所以 $(p-1)$ 是模 p 的一个周期. 所以，$\pi(p) \mid (p-1)$.

- 如果 $p \equiv 2,3 \pmod 5$，就有

$$F_{2p} \equiv \frac{1}{2^{2p-1}}\binom{2p}{p}5^{(p-1)/2} \equiv -1 \pmod p,$$

$$F_{2p+1} \equiv \frac{1}{2^{2p}}\left(\binom{2p+1}{1} + \binom{2p+1}{p}5^{(p-1)/2} + \binom{2p+1}{2p+1}5^p\right) \equiv -1 \pmod p.$$

化简过程中，利用了如下结论：由 Lucas 定理，对于 $0<k<p$ 和 $p<k<2p$ 都有 $\binom{p}{k} \equiv 0 \pmod p$，以及 $\binom{2p}{p} \equiv 2 \pmod p$，而对于 $1<k<p$ 和 $p+1<k<2p$ 都有 $\binom{p}{k} \equiv 0 \pmod p$，以及 $\binom{2p+1}{p} \equiv 2 \pmod p$；由 Fermat 小定理，有 $2^{p-1} \equiv 5^{p-1} \equiv 1 \pmod p$；对于 $p \equiv 2,3 \pmod 5$，都有 p 是模 5 的二次非剩余，利用二次互反律，也有 5 是模 p 的二次非剩余，故而 $5^{(p-1)/2} \equiv -1 \pmod p$. 由此，有 $(F_{2p}, F_{2p+1}) \equiv (F_{-2}, F_{-1}) \pmod p$，所以 $2(p+1)$ 是模 p 的一个周期. 所以，$\pi(p) \mid 2(p+1)$.

这就完成了证明. 这一方法的局限性在于它高度依赖于斐波那契数列的通项公式，所以较难直接推广到一般的情形.

- 利用扩域　　另一种证明方式则是试图直接计算矩阵 $A = \begin{pmatrix}1 & 1\\ 1 & 0\end{pmatrix}$ 的阶. 它的 特征多项式→9.14.10@p1759 是 $f(x) = x^2 - x - 1$，对应的判别式为 $\Delta = 5$. 对于模 $p = 5$，有 $\Delta \equiv 0 \pmod 5$，矩阵 A 有两个相同特征值 $\lambda = 3$，且不能对角化，需要单独计算. 对于模 $p \equiv 1,4 \pmod 5$，由二次互反律可知，判别式 $\Delta = 5$ 是模 p 的二次剩余，矩阵 A 在域 $\mathbf{F}_p$ 内有两个相异特征值 $\lambda_1 \neq \lambda_2$，矩阵 A 的阶就是 $\operatorname{lcm}(\operatorname{ord}(\lambda_1), \operatorname{ord}(\lambda_2))$，必然整除 $|\mathbf{F}_p^\times| = p-1$. 对于模 $p \equiv 2,3 \pmod 5$，由二次互反律可知，判别式 $\Delta = 5$ 是模 p 的二次非剩余，矩阵 A 在域 $\mathbf{F}_p$ 内没有特征值，而只有在 扩域→9.16.4@p1849 $\mathbf{F}_p[\sqrt{5}]$ 内才有两个相异特征值 $\lambda_1 \neq \lambda_2$，由于 Frobenius 自同态 $x \mapsto x^p$ 将两根交换，有 $\lambda_2 = \lambda_1^p$，故而 $\lambda_1^{p+1} = \lambda_2^{p+1} = \lambda_1\lambda_2 = -1$，亦即 $\lambda_1^{2(p+1)} = \lambda_2^{2(p+1)} = 1$，由此，矩阵 A 的阶就是 $\operatorname{lcm}(\operatorname{ord}(\lambda_1), \operatorname{ord}(\lambda_2))$，必然整除 $2(p+1)$. 这就得到了与前种方法一致的结论.

综上，对于不同的情形，相应地有：

- $\pi(2^e) = \frac{3}{2}\cdot 2^e$，$\frac{1}{4}\pi(5^e) = 5^e$.
- 当 $p \equiv \pm 1 \pmod{10}$ 时，$\pi(p^e) \mid (p-1)p^{e-1}$，所以 $\pi(p^e) \le p^e$.
- 当 $p \equiv \pm 3 \pmod{10}$ 时，$\frac{1}{4}\pi(p^e) \mid \frac{p+1}{2}p^{e-1}$，所以 $\frac{1}{4}\pi(p^e) \le p^e$.

所以，利用结论 1，对于一般的模数 $m = \prod_i p_i^{e_i}$，有

$$\begin{aligned}\pi(m) &= \operatorname{lcm}\{\pi(p_i^{e_i}) : p_i \in \mathbf{P}\} \\ &\le \operatorname{lcm}\{\pi(p_i^{e_i}) : p_i = 2 \text{ or } p_i \equiv \pm 1 \pmod{10}\} \\ &\quad \cdot 4 \cdot \operatorname{lcm}\{\pi(p_i^{e_i})/4 : p_i = 5 \text{ or } p_i \equiv \pm 3 \pmod{10}\} \\ &\le \prod\{\pi(p_i^{e_i}) : p_i = 2 \text{ or } p_i \equiv \pm 1 \pmod{10}\} \\ &\quad \cdot 4 \cdot \prod\{\pi(p_i^{e_i})/4 : p_i = 5 \text{ or } p_i \equiv \pm 3 \pmod{10}\} \\ &\le \frac{3}{2} \cdot \prod\{p_i^{e_i} : p_i = 2 \text{ or } p_i \equiv \pm 1 \pmod{10}\} \\ &\quad \cdot 4 \cdot \prod\{p_i^{e_i} : p_i = 5 \text{ or } p_i \equiv \pm 3 \pmod{10}\} \\ &= 6m.\end{aligned}$$

这就说明了斐波那契数列模 m 的 Pisano 周期总是不超过 $6m$，而且等号当且仅当在 $m = 2 \cdot 5^e$ 处取得.

习题

- SPOJ - Euclid Algorithm Revisited[6]
- SPOJ - Fibonacci Sum[7]
- HackerRank - Is Fibo[8]
- Project Euler - Even Fibonacci numbers[9]
- 洛谷 P4000 斐波那契数列[10]

参考文献与注释

- Fibonacci sequence - Wikipedia[11]
- Zeckendorf's theorem - Wikipedia[12]
- Pisano period - Wikipedia[13]

本页面主要译自博文 Числа Фибоначчи[14] 与其英文翻译版 Fibonacci Numbers[15]．其中俄文版版权协议为 Public Domain + Leave a Link；英文版版权协议为 CC-BY-SA 4.0．内容有改动．

[1] OEIS A000045

[2] OEIS A000032

[3] 维基 - 拉梅

[4] 齐肯多夫定理

[5] OEIS A001175

[6] SPOJ - Euclid Algorithm Revisited

[7] SPOJ - Fibonacci Sum

[8] HackerRank - Is Fibo

[9] Project Euler - Even Fibonacci numbers

[10] 洛谷 P4000 斐波那契数列

[11] Fibonacci sequence - Wikipedia

[12] Zeckendorf's theorem - Wikipedia

[13] Pisano period - Wikipedia

[14] Числа Фибоначчи

[15] Fibonacci Numbers

9.13.5 错位排列

错位排列

定义

错位排列（derangement）是没有任何元素出现在其有序位置的排列．即，对于 $1 \sim n$ 的排列 P，如果满足 $P_i \neq i$，则称 P 是 n 的错位排列.

例如，三元错位排列有 $\{2,3,1\}$ 和 $\{3,1,2\}$．四元错位排列有 $\{2,1,4,3\}$、$\{2,3,4,1\}$、$\{2,4,1,3\}$、$\{3,1,4,2\}$、$\{3,4,1,2\}$、$\{3,4,2,1\}$、$\{4,1,2,3\}$、$\{4,3,1,2\}$ 和 $\{4,3,2,1\}$．错位排列是没有不动点的排列，即没有长度为 1 的循环.

容斥原理的计算

全集 U 即为 $1\sim n$ 的排列，$|U|=n!$；令 S_i 是其中满足 $P_i\neq i$ 的排列．运用补集和 容斥原理 →9.13.3@p1618 的知识，问题变成求：

$$\left|\bigcap_{i=1}^{n}S_i\right|=|U|-\left|\bigcup_{i=1}^{n}\overline{S_i}\right|$$

$$=n!-\sum_{k=1}^{n}(-1)^{k-1}\sum_{a_i<a_{i+1}}\left|\bigcap_{i=1}^{k}\overline{S_{a_i}}\right|$$

其中求和的含义是 $1,2,\cdots,n$ 中取 $a_1,a_2,\cdots,a_k$ 且满足 $a_i<a_{i+1}$．于是

$$\left|\bigcap_{i=1}^{k}\overline{S_{a_i}}\right|$$

表示有 k 个数 $a_1,a_2,\cdots,a_k$ 满足 $P_{a_i}=a_i$，而剩下 $n-k$ 个数的位置任意的排列数，因此：

$$\left|\bigcap_{i=1}^{k}\overline{S_{a_i}}\right|=(n-k)!$$

这 k 个数的选择情况共 $\binom{n}{k}$ 种，对其求和有：

$$\sum_{k=1}^{n}(-1)^{k-1}\sum_{a_i<a_{i+1}}\left|\bigcap_{i=1}^{k}\overline{S_{a_i}}\right|$$

$$=\sum_{k=1}^{n}(-1)^{k-1}\binom{n}{k}(n-k)!$$

$$=\sum_{k=1}^{n}(-1)^{k-1}\frac{n!}{k!}$$

$$=n!\sum_{k=1}^{n}\frac{(-1)^{k-1}}{k!}$$

因此 n 个元素的错位排列数为：

$$D_n=n!-n!\sum_{k=1}^{n}\frac{(-1)^{k-1}}{k!}=n!\sum_{k=0}^{n}\frac{(-1)^k}{k!}$$

错位排列数列的前几项为 $0,1,2,9,44,265$（OEIS A000166[1]）.

递推的计算

把错位排列问题具体化，考虑这样一个问题：

n 封不同的信，编号分别是 $1,2,3,4,5$，现在要把这五封信放在编号 $1,2,3,4,5$ 的信封中，要求信封的编号与信的编号不一样．问有多少种不同的放置方法?

假设考虑到第 n 个信封，初始时暂时把第 n 封信放在第 n 个信封中，然后考虑两种情况的递推：

- 前面 $n-1$ 个信封全部装错；
- 前面 $n-1$ 个信封有一个没有装错其余全部装错.

对于第一种情况，前面 $n-1$ 个信封全部装错：因为前面 $n-1$ 个已经全部装错了，所以第 n 封只需要与前面任一一个位置交换即可，总共有 $D_{n-1}\times(n-1)$ 种情况.

对于第二种情况，前面 $n-1$ 个信封有一个没有装错其余全部装错：考虑这种情况的目的在于，若 $n-1$ 个信封中如果有一个没装错，那么把那个没装错的与 n 交换，即可得到一个全错位排列情况.

其他情况，不可能通过一次操作来把它变成一个长度为 n 的错排.

于是可得，错位排列数满足递推关系：

$$D_n=(n-1)(D_{n-1}+D_{n-2})$$

这里也给出另一个递推关系：

$$D_n = nD_{n-1} + (-1)^n$$

其他关系

错位排列数有一个简单的取整表达式，增长速度与阶乘仅相差常数：

$$D_n = \left\lfloor \frac{n!}{\mathrm{e}} + \frac{1}{2} \right\rfloor$$

随着元素数量的增加，形成错位排列的概率 P 接近：

$$P = \lim_{n\to\infty} \frac{D_n}{n!} = \frac{1}{\mathrm{e}}$$

外部参考资料

[1] OEIS A000166



9.13.6 卡特兰数

引入

Catalan 数经常出现在各类计数问题中．比利时数学家 Eugène Charles Catalan 在 1958 年研究括号序列计数问题时发现了这一数列，它也因此得名．清朝数学家明安图早在 18 世纪 30 年代就已经发现这一数列．

Catalan 数满足如下递推关系：

$$C_n = \begin{cases} 1, & n = 0, \\ \sum_{i=0}^{n-1} C_i C_{n-1-i}, & n > 0. \end{cases} \tag{1}$$

数列的前几项为：（OEIS: A000108[1]，下标从 0 开始）

$$1, 1, 2, 5, 14, 42, 132, 429, 1430, \ldots$$

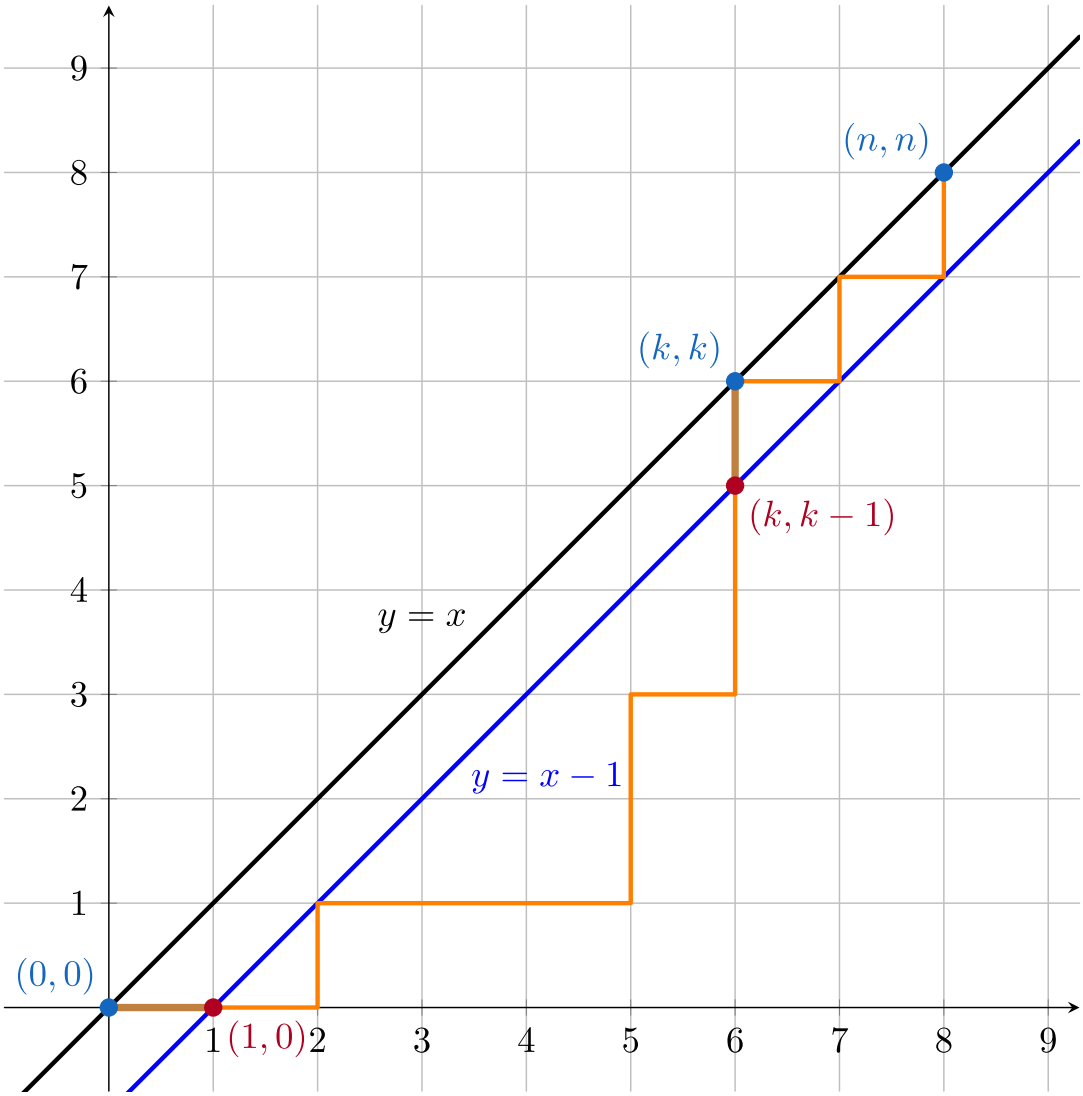
应用

Catalan 数 C_n 的递推关系有着天然的递归结构：规模为 n 的计数问题 C_n，可以通过枚举分界点，分拆为两个规模分别为 i 和 $(n-1-i)$ 的子问题．这一递推关系使得 Catalan 数广泛出现于各类具有类似递归结构的问题中．

- **路径计数问题**：有一个大小为 $n \times n$ 的方格图，左下角为 $(0,0)$，右上角为 (n,n)．从左下角开始，每次都只能向右或者向上走一单位，不走到对角线 $y = x$ 上方（但可以触碰）的情况下，到达右上角的路径总数为 C_n．

  证明

  设方案数为 T_n．考虑 $n \geq 2$ 的情况．设路径 **第一次** 走到对角线 $y = x$ 的点是 (k,k) $(k \in [1,n])$．考察从 $(0,0)$ 到 (k,k) 的除起点和终点外，中间的点 **不经过对角线（不能碰到）** 的路径．



如图所示，这些路径的第一步一定向右，从 $(0,0)$ 到 $(1,0)$；最后一步一定向上，从 $(k,k-1)$ 到 (k,k)．因此，这些路径就是从 $(1,0)$ 到 $(k,k-1)$ 的不越过直线 $y=x-1$ 的路径，这样路径的数目就是 T_{k-1}．同时，从 (k,k) 到 (n,n) 的合法路径数就是 T_{n-k}．根据乘法原理，第一次在 (k,k) 处触碰对角线的路径数目为 $T_{k-1}T_{n-k}$．枚举 k 的所有可能性，所有合法路径的数目为

$$T_n=\sum_{k=1}^{n}T_{k-1}T_{n-k}.$$

做代换 $k=i+1$ 就可以发现，这就是 Catalan 数的递推关系．由 $T_0=1$ 可知 $T_n=C_n$．

- **圆内不相交弦计数问题**：圆上有 $2n$ 个点，将这些点成对连接起来且使得所得到的 n 条线段两两不交的方案数是 C_n．

证明

记 $2n$ 个点的方案数为 T_n．将 $2n$ 个点按顺时针标号，分别为 $1,2,\ldots,2n$．由于弦两两不交，1 号点只能连接偶数号点；否则，两点之间的奇数个点无法在不穿过两点连线的情况下两两配对．如果连接了 1 和 $2k$ $(k\in[1,n])$，那么左边有 $2k-2$ 个点，右边有 $2n-2k$ 个点，由乘法原理，这样的方案数为 $T_{k-1}T_{n-k}$．因此，枚举 k，有 $T_n=\sum_{k=1}^{n}T_{k-1}T_{n-k}$．令 $k=i+1$，就得到 Catalan 数的递推关系．由 $T_0=1$ 可知 $T_n=C_n$．

- **三角剖分计数问题**：对角线不相交的情况下，将一个凸 $(n+2)$ 边形区域分成三角形区域的方法数为 C_n.

  证明

  设 $(n+2)$ 边形三角剖分的方案数为 T_n. 先选定一条边 $(1,n+2)$ 作为基边，它一定属于一个三角形，记该三角形的第三个点为 k $(k\in[2,n+1])$. 这样，原凸多边形变成了三个部分：
  - 三角形 $(1,k,n+2)$.
  - k 边形，顶点 $1\sim k$.
  - $(n+3-k)$ 边形，顶点 $k\sim(n+2)$.

  后面两个部分都是子问题，所以，有递推关系

  $$T_n=\sum_{k=2}^{n+1}T_{k-2}T_{n+1-k}.$$

  令 $k=i+2$，就得到 Catalan 数递归关系. 由 $T_0=T_1=1$ 可知 $T_n=C_n$.

- **二叉树计数问题**：含有 n 个结点的形态不同的二叉树数目为 C_n. 等价地，含有 n 个非叶结点的形态不同的满二叉树数目为 C_n.

  证明

  记 n 个结点的二叉树数目为 T_n. 任取一个根结点，枚举左右子树大小. 设左子树大小为 $i\in[0,n-1]$，则右子树大小为 $(n-1-i)$. 左右子树均为子问题，所以，有递推关系

  $$T_n=\sum_{i=0}^{n-1}T_iT_{n-1-i}.$$

  这就是 Catalan 数递推关系. 由 $T_0=T_1=1$ 可知 $T_n=C_n$.

- **括号序列计数问题**：由 n 对括号构成的合法括号序列数为 C_n.

  证明

  联系路径计数问题. 将左括号视为向上走，右括号视为向右走. 合法括号序列即为，在任意位置，左括号的数量不少于右括号的数量. 相当于路径计数问题中，在任意时刻，向上走的次数不少于向右走的次数. 因此，合法括号序列与合法路径之间存在双射. 合法括号序列的数目同样为 C_n.

- **出栈序列计数问题**：一个栈（无穷大）的进栈序列为 $1,2,3,\ldots,n$，合法出栈序列的数目为 C_n.

  证明

  联系括号序列计数问题. 将入栈视为左括号，出栈视为右括号. 任意时刻，入栈的次数不少于出栈的次数. 因此，合法出栈序列与合法括号序列之间存在双射. 合法出栈序列的数目同样为 C_n.

- **数列计数问题**：由 n 个 $+1$ 和 n 个 -1 组成的数列 $a_1,a_2,\ldots,a_{2n}$ 中，部分和满足 $a_1+a_2+\ldots+a_k\geq 0$ $(k=1,2,3,\ldots,2n)$ 的数列数目为 C_n.

  证明

  联系括号序列计数问题. 将 $+1$ 视为左括号，-1 视为右括号. 任意时刻，$+1$ 的数量不少于 -1 的数量. 因此，合法数列与合法括号序列之间存在双射. 合法数列的数目同样为 C_n.

尽管这一递推关系应用广泛，但是直接计算复杂度较高，需要寻找更为简单的公式.

常见形式

Catalan 数有如下常见的表达式：

$$C_n = \frac{1}{n+1}\binom{2n}{n} = \frac{(2n)!}{n!(n+1)!},\ n \geq 0. \tag{2}$$

$$C_n = \binom{2n}{n} - \binom{2n}{n+1},\ n \geq 0. \tag{3}$$

$$C_n = \frac{(4n-2)}{n+1}C_{n-1},\ n > 0,\ C_0 = 1. \tag{4}$$

Catalan 数的这些形式都可以高效计算：前两个形式将它转换为阶乘和组合数的计算问题，第三个形式则提供了顺次计算的递推公式.

对于这三种常见形式，本文提供两种证明方式.

代数推演

通过代数方法得出 Catalan 数的上述表达式共两步. 首先，验证三个形式相互等价.

证明表达式 (2) ~ (4) 等价

只需要证明表达式 (3) 可以转化为表达式 (2) 中阶乘形式：

$$\begin{aligned}
C_n &= \binom{2n}{n} - \binom{2n}{n+1} \\
&= \frac{(2n)!}{n!n!} - \frac{(2n)!}{(n-1)!(n+1)!} \\
&= \frac{(2n)!}{n!n!}\left(1 - \frac{n!}{(n-1)!(n+1)}\right) \\
&= \frac{(2n)!}{n!n!}\left(1 - \frac{n}{n+1}\right) \\
&= \frac{(2n)!}{n!(n+1)!}.
\end{aligned}$$

以及，表达式 (4) 也可以转化为表达式 (2) 中阶乘形式：

$$C_n = \prod_{i=1}^{n}\frac{(4i-2)}{i+1} = \prod_{i=1}^{n}\frac{2i(2i-1)}{i(i+1)} = \frac{(2n)!}{n!(n+1)!}.$$

因此，三个表达式互相等价.

紧接着，验证这些形式确实是 Catalan 数递推公式的解. 为此，考虑使用生成函数方法直接求出递推公式 (1) 的解.

利用生成函数方法求解递推公式 (1)

考虑 Catalan 数的普通生成函数 $C(x) = \sum_{n=0}^{\infty} C_n x^n$. 由于 Catalan 数的递推关系和卷积形式很相似，所以考虑用卷积构造 $C(x)$ 的方程：

$$\begin{aligned}C(x)&=\sum_{n=0}^{\infty}C_nx^n\\&=1+\sum_{n=1}^{\infty}\left(\sum_{i=0}^{n-1}C_iC_{n-i-1}\right)x^n\\&=1+x\sum_{n=1}^{\infty}\sum_{i=0}^{n-1}C_ix^iC_{n-i-1}x^{n-i-1}\\&=1+x\sum_{i=0}^{\infty}C_ix^i\sum_{j=0}^{\infty}C_jx^j\\&=1+xC^2(x).\end{aligned}$$

其中，倒数第二个等号交换了求和次序，并令 $j=n-1-i$．由此，解得：

$$C(x)=\frac{1\pm\sqrt{1-4x}}{2x}=\frac{2}{1\mp\sqrt{1-4x}}.$$

由初值条件 $C_0=1$ 可知，$C(0)=1$．代入检验可以发现唯一可行的解就是

$$C(x)=\frac{1-\sqrt{1-4x}}{2x}.$$

接下来，需要将它展开为幂级数的形式．利用 $(1+x)^a$ 的 幂级数展开式→9.12.1@p1519 可知：

$$\sqrt{1-4x}=\sum_{n=0}^{\infty}\frac{\left(\frac{1}{2}\right)_{-n}}{n!}(-4x)^n,$$

其中，$\left(\frac{1}{2}\right)_{-n}$ 是下降阶乘幂：

$$\begin{aligned}\left(\frac{1}{2}\right)_{-n}&=\prod_{k=0}^{n-1}\left(\frac{1}{2}-k\right)=\frac{1}{2^n}\prod_{k=1}^{n-1}(1-2k)=\frac{(-1)^{n-1}}{2^n}\prod_{k=1}^{n-1}(2k-1)\\&=\frac{(-1)^{n-1}}{2^{2n-1}}\prod_{k=1}^{n-1}\frac{(2k-1)2k}{k}=\frac{(-1)^{n-1}}{2^{2n-1}}\frac{(2n-2)!}{(n-1)!}.\end{aligned}$$

代入 $C(x)$ 的表达式，就有

$$\begin{aligned}C(x)&=\frac{1}{2x}\left(1-\sum_{n=0}^{\infty}\frac{\left(\frac{1}{2}\right)_{-n}}{n!}(-4x)^n\right)\\&=-\frac{1}{2x}\sum_{n=1}^{\infty}\frac{(-4x)^n}{n!}\left(\frac{1}{2}\right)_{-n}\\&=-\frac{1}{2x}\sum_{n=1}^{\infty}\frac{(-4x)^n}{n!}\frac{(-1)^{n-1}}{2^{2n-1}}\frac{(2n-2)!}{(n-1)!}\\&=\sum_{n=1}^{\infty}\frac{(2n-2)!}{(n-1)!n!}x^{n-1}\\&=\sum_{n=0}^{\infty}\frac{(2n)!}{n!(n+1)!}x^n.\end{aligned}$$

由此，就得到 C_n 的表达式 (2)．

组合意义

由于 Catalan 数具有明显的组合意义，所以只使用组合计数方法同样可以证明这些形式．本节为三个表达式分别提供一个组合意义的证明．

表达式 (2) 的证明

考虑数列计数问题. 对于任意由 ± 1 组成的序列 $\{a_i\}_{i=1}^{2n}$，定义它的部分和为 $S_i=\sum_{j=1}^{i}a_i$，并定义它的 **超额量**（exceedance）为 $S_i<0$ 且 $a_i=-1$ 的下标数量. 超额量为 0，就等价于数列合法；超额量的取值范围是 $[0,n]$，共 $(n+1)$ 种可能的取值. 需要证明的是，不同超额量的数列数量其实是一样的.

为此，可以构造一个从超额量为 $e>0$ 的数列到超额量为 $(e-1)$ 的数列的映射 f. 对于超额量为 $e>0$ 的序列 $\{a_i\}$，取下标 k 为使得 $S_i=0$ 且 $a_i=+1$ 成立的下标最小值. 将 a_k 左右两侧的序列交换，就得到如下序列 $\{a'_i\}$：

$$a_{k+1},a_{k+2},\cdots,a_{2n},a_k,a_1,a_2,\cdots,a_{k-1}.$$

由于原序列中 a_k 右侧部分在交换前后对应的部分和序列不变，所以它们贡献的超额量也不变. 对于原序列中 a_k 左侧部分，它们对应的部分和在交换后全部增加 1，因此，它们贡献的超额量会减少，而且减少的数量恰好等于原序列 a_k 左侧部分中满足 $S_i=-1$ 且 $a_i=-1$ 的下标数量. 因为 a_k 的选取保证了这样的下标有且仅有一个，所以，序列 $\{a'_i\}$ 的超额量就等于 $(e-1)$. 也就是说，映射 f 可以将序列的超额量恰好减少 1.

映射 f 是可逆的. 注意到序列 $\{a'_i\}$ 中，a_k 对应的位置恰好为满足 $S'_k=+1$ 且 $a'_i=+1$ 的下标最大值. 这是因为交换后，这些部分和都比交换前对应的部分和恰好大 1，因此，现在的部分和为 $+1$ 对应交换前部分和等于 0. 但是，根据 k 的选取，交换前这一部分（即原序列 a_k 左侧部分）是没有满足 $S_i=0$ 且 $a_i=+1$ 成立的下标的.

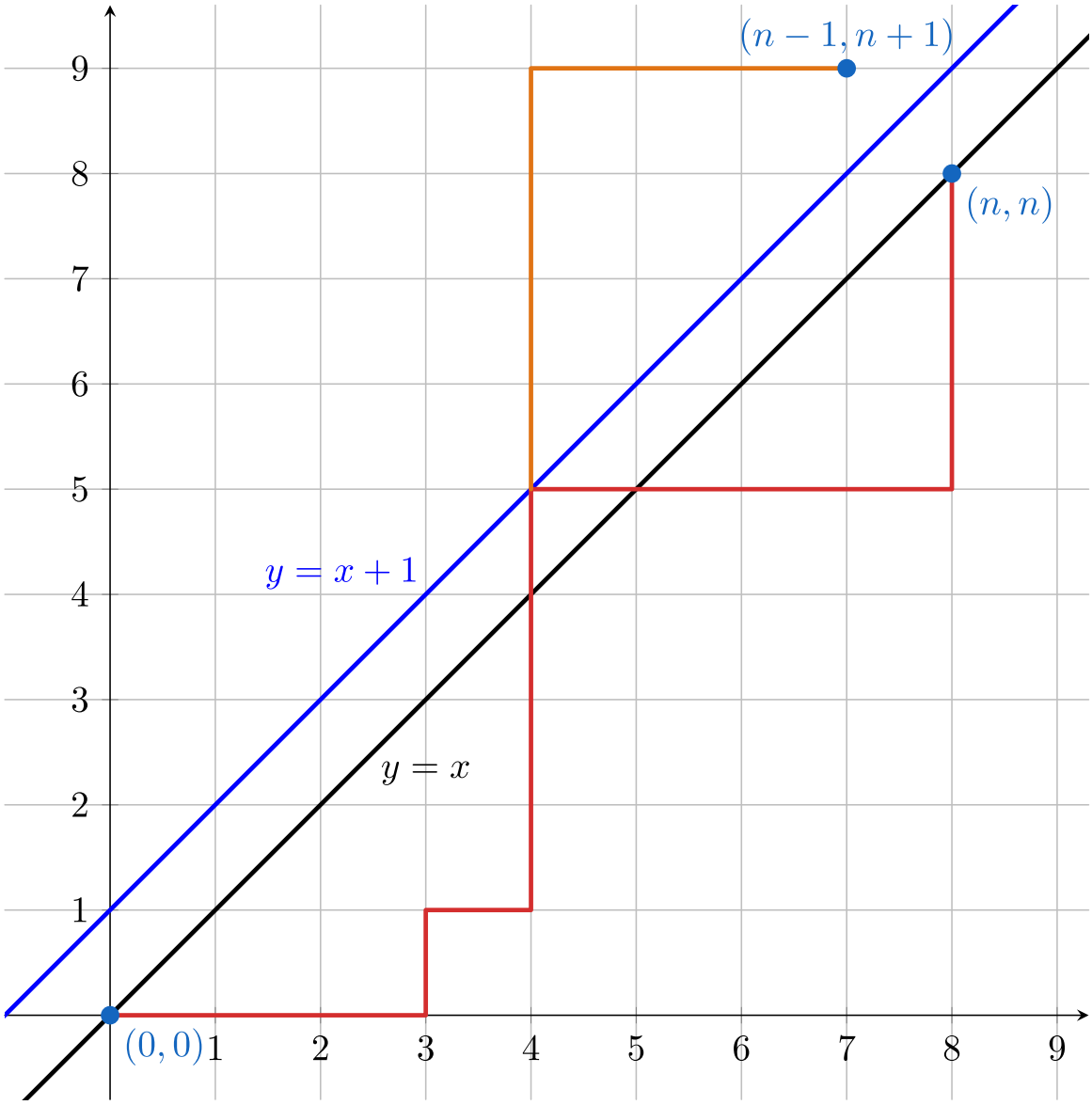
由此，映射 f 构成了超额量为 $e>0$ 的序列和超额量为 $(e-1)$ 的序列之间的双射. 这就说明，不同超额量的数列数量其实是一样的. 由于数列总数是 $\binom{2n}{n}$，合法数列（即超额量为 0 的数列）数量就等于

$$C_n=\frac{1}{n+1}\binom{2n}{n}.$$

这就证明了 Catalan 数的表达式 (2).

表达式 (3) 的证明

考虑路径计数问题. 这是典型的格路计数问题，可以通过反射原理求解. 具体到本问题，考虑用总路径数目减去不合法的路径数目. 总路径数一共要走 $2n$ 步，其中 n 步向右，所以方案数为 $\binom{2n}{n}$. 一条路径不合法，当且仅当它碰到了直线 $y=x+1$. 对于任意一条非法路径，可以找到第一次碰到直线 $y=x+1$ 的位置，并将该位置之后的路径关于直线 $y=x+1$ 做对称. 此时，可以发现，一条从 $(0,0)$ 到 (n,n) 的非法路径，变成了一条从 $(0,0)$ 到 $(n-1,n+1)$ 的路径.



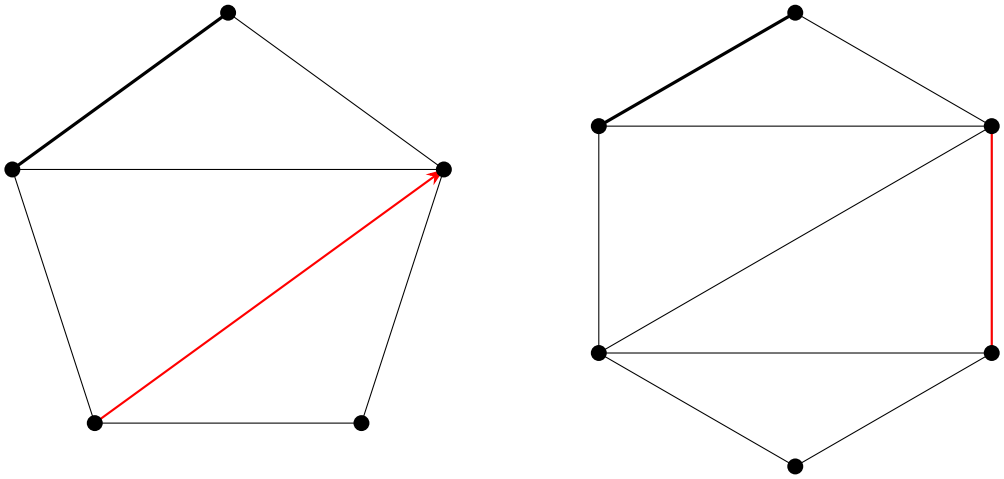
由于从 $(0,0)$ 到 $(n-1,n+1)$ 的路径必定要穿过直线 $y=x+1$，所以每条这样的路径都对应一条从 $(0,0)$ 到 (n,n) 的非法路径．类似总路径数的计算，非法路径数目的总数就是 $\binom{2n}{n+1}$．因此，合法路径的总数为

$$C_n=\binom{2n}{n}-\binom{2n}{n+1}.$$

这就是 Catalan 数的表达式 (3)．

表达式 (4) 的证明

考虑三角剖分计数问题．设 P 是凸 $(n+2)$ 边形，固定它的一个边为基边．对于多边形 P 的每一个三角剖分，都可以选择它的一个非基边（包括三角剖分时新加的边）标记，并定向．这共有 $(4n+2)C_n$ 种剖分加标记的方案．又设 Q 是凸 $(n+3)$ 边形，仍固定它的一个边为基边．对于多边形 Q，可以选择它的一条非基边标记，然后再做三角剖分．这共有 $(n+2)C_{n+1}$ 种标记加剖分的方案．



P Q

如图所示，这两组操作得到的结果之间存在明显的双射. 对于 P 剖分并标记的一个结果，可以将它的标记边扩展为三角形，定向所指向的终点扩展为一条新边，并将这条新边打上标记，这就得到对 Q 标记并剖分的一个结果；对于 Q 标记并剖分的一个结果，可以将它的标记边压缩为一个点，并将压缩得到的对角线打上标记，且指向压缩得到的顶点，这就得到对 P 剖分并标记的一个结果. 因此，

$$(4n+2)C_n = (n+2)C_{n+1}.$$

稍作整理，并结合 $C_0 = 1$，就得到 Catalan 数的表达式 (4).

例题

洛谷 P1044 栈[2]

入栈顺序为 $1, 2, \ldots, n$，求所有可能的出栈顺序的总数.

参考代码

- C++

```cpp
#include <iostream>
using namespace std;

int n;
long long f[25];

int main() {
  f[0] = 1;
  cin >> n;
  for (int i = 1; i <= n; i++) f[i] = f[i - 1] * (4 * i - 2) / (i + 1);
  // 这里用的是常见形式 4
  cout << f[n] << endl;
  return 0;
}

```

- Python

```python
f = [0] * 25
f[0] = 1
n = int(input())
for i in range(1, n + 1):
    f[i] = f[i - 1] * (4 * i - 2) // (i + 1)
    # 这里用的是常见形式 4
print(f[n])

```

习题

- Luogu P2532 [AHOI2012] 树屋阶梯[3]
- Luogu P1641 [SCOI2010] 生成字符串[4]
- Luogu P3200 [HNOI2009] 有趣的数列[5]
- AtCoder Beginner Contest 205 E - White and Black Balls[6]
- AtCoder Regular Contest 145 C - Split and Maximize[7]
- Luogu P5014 水の三角（修改版）[8]
- Luogu P3978 [TJOI2015] 概率论[9]

参考资料与注释

- Catalan number - Wikipedia[10]

外部参考资料

[1] OEIS: A000108

[2] 洛谷 P1044 栈

[3] Luogu P2532 [AHOI2012] 树屋阶梯

[4] Luogu P1641 [SCOI2010] 生成字符串

[5] Luogu P3200 [HNOI2009] 有趣的数列

[6] AtCoder Beginner Contest 205 E - White and Black Balls

[7] AtCoder Regular Contest 145 C - Split and Maximize

[8] Luogu P5014 水の三角（修改版）

[9] Luogu P3978 [TJOI2015] 概率论

[10] Catalan number - Wikipedia

9.13.7 斯特林数

第二类斯特林数（Stirling Number）

为什么先介绍第二类斯特林数

虽然被称作「第二类」，第二类斯特林数却在斯特林的相关著作和具体数学中被首先描述，同时也比第一类斯特林数常用得多.

第二类斯特林数（斯特林子集数）$\left\{ {n \atop k} \right\}$，也可记做 $S(n,k)$，表示将 n 个两两不同的元素，划分为 k 个互不区分的非空子集的方案数.

递推式

$$\left\{ {n \atop k} \right\} = \left\{ {n-1 \atop k-1} \right\} + k \left\{ {n-1 \atop k} \right\}$$

边界是 $\left\{ {n \atop 0} \right\} = [n=0]$.

考虑用组合意义来证明.

我们插入一个新元素时，有两种方案：

- 将新元素单独放入一个子集，有 $\left\{ {n-1 \atop k-1} \right\}$ 种方案；
- 将新元素放入一个现有的非空子集，有 $k\left\{ {n-1 \atop k} \right\}$ 种方案.

根据加法原理，将两式相加即可得到递推式.

通项公式

$$\left\{ {n \atop m} \right\} = \sum_{i=0}^{m} \frac{(-1)^{m-i} i^n}{i!(m-i)!}$$

使用容斥原理证明该公式. 设将 n 个两两不同的元素，划分到 i 个两两不同的集合（允许空集）的方案数为 G_i，将 n 个两两不同的元素，划分到 i 个两两不同的非空集合（不允许空集）的方案数为 F_i.

显然

$$G_i = i^n$$

$$G_i = \sum_{j=0}^{i} \binom{i}{j} F_j$$

根据二项式反演

$$\begin{aligned} F_i &= \sum_{j=0}^{i} (-1)^{i-j} \binom{i}{j} G_j \\ &= \sum_{j=0}^{i} (-1)^{i-j} \binom{i}{j} j^n \\ &= \sum_{j=0}^{i} \frac{i!(-1)^{i-j} j^n}{j!(i-j)!} \end{aligned}$$

考虑 F_i 与 $\left\{ {n \atop i} \right\}$ 的关系. 第二类斯特林数要求集合之间互不区分，因此 F_i 正好就是 $\left\{ {n \atop i} \right\}$ 的 $i!$ 倍. 于是

$$\left\{ {n \atop m} \right\} = \frac{F_m}{m!} = \sum_{i=0}^{m} \frac{(-1)^{m-i} i^n}{i!(m-i)!}$$

同一行第二类斯特林数的计算

「同一行」的第二类斯特林数指的是，有着不同的 i，相同的 n 的一系列 $\left\{ {n \atop i} \right\}$. 求出同一行的所有第二类斯特林数，就是对 $i = 0..n$ 求出了将 n 个不同元素划分为 i 个非空集的方案数.

根据上面给出的通项公式，卷积计算即可. 该做法的时间复杂度为 $O(n \log n)$.

下面的代码使用了名为 `poly` 的多项式类，仅供参考.

实现

```cpp
#ifndef _FEISTDLIB_POLY_
#define _FEISTDLIB_POLY_

/*
 * This file is part of the fstdlib project.
 * Version: Build v0.0.2
 * You can check for details at https://github.com/FNatsuka/fstdlib
 */

#include <algorithm>
#include <cmath>
```

```
#include <cstdio>
#include <vector>

namespace fstdlib {

using ll = long long;
int mod = 998244353, grt = 3;

class poly {
 private:
  std::vector<int> data;

  void out(void) {
    for (int i = 0; i < (int)data.size(); ++i) printf("%d ", data[i]);
    puts("");
  }

 public:
  poly(std::size_t len = std::size_t(0)) { data = std::vector<int>(len); }

  poly(const std::vector<int> &b) { data = b; }

  poly(const poly &b) { data = b.data; }

  void resize(std::size_t len, int val = 0) { data.resize(len, val); }

  std::size_t size(void) const { return data.size(); }

  void clear(void) { data.clear(); }
#if __cplusplus >= 201103L
  void shrink_to_fit(void) { data.shrink_to_fit(); }
#endif
  int &operator[](std::size_t b) { return data[b]; }

  const int &operator[](std::size_t b) const { return data[b]; }

  poly operator*(const poly &h) const;
  poly operator*=(const poly &h);
  poly operator*(const int &h) const;
  poly operator*=(const int &h);
  poly operator+(const poly &h) const;
  poly operator+=(const poly &h);
  poly operator-(const poly &h) const;
  poly operator-=(const poly &h);
  poly operator<<(const std::size_t &b) const;
  poly operator<<=(const std::size_t &b);
  poly operator>>(const std::size_t &b) const;
  poly operator>>=(const std::size_t &b);
  poly operator/(const int &h) const;
  poly operator/=(const int &h);
  poly operator==(const poly &h) const;
  poly operator!=(const poly &h) const;
```

```
  poly operator+(const int &h) const;
  poly operator+=(const int &h);
  poly inv(void) const;
  poly inv(const int &h) const;
  friend poly sqrt(const poly &h);
  friend poly log(const poly &h);
  friend poly exp(const poly &h);
};

int qpow(int a, int b, int p = mod) {
  int res = 1;
  while (b) {
    if (b & 1) res = (ll)res * a % p;
    a = (ll)a * a % p, b >>= 1;
  }
  return res;
}

std::vector<int> rev;

void dft_for_module(std::vector<int> &f, int n, int b) {
  static std::vector<int> w;
  w.resize(n);
  for (int i = 0; i < n; ++i)
    if (i < rev[i]) std::swap(f[i], f[rev[i]]);
  for (int i = 2; i <= n; i <<= 1) {
    w[0] = 1, w[1] = qpow(grt, (mod - 1) / i);
    if (b == -1) w[1] = qpow(w[1], mod - 2);
    for (int j = 2; j < i / 2; ++j) w[j] = (ll)w[j - 1] * w[1] % mod;
    for (int j = 0; j < n; j += i)
      for (int k = 0; k < i / 2; ++k) {
        int p = f[j + k], q = (ll)f[j + k + i / 2] * w[k] % mod;
        f[j + k] = (p + q) % mod, f[j + k + i / 2] = (p - q + mod) % mod;
      }
  }
}

poly poly::operator*(const poly &h) const {
  int N = 1;
  while (N < (int)(size() + h.size() - 1)) N <<= 1;
  std::vector<int> f(this->data), g(h.data);
  f.resize(N), g.resize(N);
  rev.resize(N);
  for (int i = 0; i < N; ++i)
    rev[i] = (rev[i >> 1] >> 1) | (i & 1 ? N >> 1 : 0);
  dft_for_module(f, N, 1), dft_for_module(g, N, 1);
  for (int i = 0; i < N; ++i) f[i] = (ll)f[i] * g[i] % mod;
  dft_for_module(f, N, -1), f.resize(size() + h.size() - 1);
  for (int i = 0, inv = qpow(N, mod - 2); i < (int)f.size(); ++i)
    f[i] = (ll)f[i] * inv % mod;
  return f;
}
```

```
116
117 poly poly::operator*=(const poly &h) { return *this = *this * h; }
118
119 poly poly::operator*(const int &h) const {
120   std::vector<int> f(this->data);
121   for (int i = 0; i < (int)f.size(); ++i) f[i] = (ll)f[i] * h % mod;
122   return f;
123 }
124
125 poly poly::operator*=(const int &h) {
126   for (int i = 0; i < (int)size(); ++i) data[i] = (ll)data[i] * h % mod;
127   return *this;
128 }
129
130 poly poly::operator+(const poly &h) const {
131   std::vector<int> f(this->data);
132   if (f.size() < h.size()) f.resize(h.size());
133   for (int i = 0; i < (int)h.size(); ++i) f[i] = (f[i] + h[i]) % mod;
134   return f;
135 }
136
137 poly poly::operator+=(const poly &h) {
138   std::vector<int> &f = this->data;
139   if (f.size() < h.size()) f.resize(h.size());
140   for (int i = 0; i < (int)h.size(); ++i) f[i] = (f[i] + h[i]) % mod;
141   return f;
142 }
143
144 poly poly::operator-(const poly &h) const {
145   std::vector<int> f(this->data);
146   if (f.size() < h.size()) f.resize(h.size());
147   for (int i = 0; i < (int)h.size(); ++i) f[i] = (f[i] - h[i] + mod) % mod;
148   return f;
149 }
150
151 poly poly::operator-=(const poly &h) {
152   std::vector<int> &f = this->data;
153   if (f.size() < h.size()) f.resize(h.size());
154   for (int i = 0; i < (int)h.size(); ++i) f[i] = (f[i] - h[i] + mod) % mod;
155   return f;
156 }
157
158 poly poly::operator<<(const std::size_t &b) const {
159   std::vector<int> f(size() + b);
160   for (int i = 0; i < (int)size(); ++i) f[i + b] = data[i];
161   return f;
162 }
163
164 poly poly::operator<<=(const std::size_t &b) { return *this = (*this) <<
    b; }
165
166 poly poly::operator>>(const std::size_t &b) const {
```

```
167   std::vector<int> f(size() - b);
168   for (int i = 0; i < (int)f.size(); ++i) f[i] = data[i + b];
169   return f;
170 }
171
172 poly poly::operator>>=(const std::size_t &b) { return *this = (*this) >>
    b; }
173
174 poly poly::operator/(const int &h) const {
175   std::vector<int> f(this->data);
176   int inv = qpow(h, mod - 2);
177   for (int i = 0; i < (int)f.size(); ++i) f[i] = (ll)f[i] * inv % mod;
178   return f;
179 }
180
181 poly poly::operator/=(const int &h) {
182   int inv = qpow(h, mod - 2);
183   for (int i = 0; i < (int)data.size(); ++i) data[i] = (ll)data[i] * inv
    % mod;
184   return *this;
185 }
186
187 poly poly::inv(void) const {
188   int N = 1;
189   while (N < (int)(size() + size() - 1)) N <<= 1;
190   std::vector<int> f(N), g(N), d(this->data);
191   d.resize(N), f[0] = qpow(d[0], mod - 2);
192   for (int w = 2; w < N; w <<= 1) {
193     for (int i = 0; i < w; ++i) g[i] = d[i];
194     rev.resize(w << 1);
195     for (int i = 0; i < w * 2; ++i)
196       rev[i] = (rev[i >> 1] >> 1) | (i & 1 ? w : 0);
197     dft_for_module(f, w << 1, 1), dft_for_module(g, w << 1, 1);
198     for (int i = 0; i < w * 2; ++i)
199       f[i] = (ll)f[i] * (2 + mod - (ll)f[i] * g[i] % mod) % mod;
200     dft_for_module(f, w << 1, -1);
201     for (int i = 0, inv = qpow(w << 1, mod - 2); i < w; ++i)
202       f[i] = (ll)f[i] * inv % mod;
203     for (int i = w; i < w * 2; ++i) f[i] = 0;
204   }
205   f.resize(size());
206   return f;
207 }
208
209 poly poly::operator==(const poly &h) const {
210   if (size() != h.size()) return 0;
211   for (int i = 0; i < (int)size(); ++i)
212     if (data[i] != h[i]) return 0;
213   return 1;
214 }
215
216 poly poly::operator!=(const poly &h) const {
```

```
  if (size() != h.size()) return 1;
  for (int i = 0; i < (int)size(); ++i)
    if (data[i] != h[i]) return 1;
  return 0;
}

poly poly::operator+(const int &h) const {
  poly f(this->data);
  f[0] = (f[0] + h) % mod;
  return f;
}

poly poly::operator+=(const int &h) { return *this = (*this) + h; }

poly poly::inv(const int &h) const {
  poly f(*this);
  f.resize(h);
  return f.inv();
}

int modsqrt(int h, int p = mod) { return 1; }

poly sqrt(const poly &h) {
  int N = 1;
  while (N < (int)(h.size() + h.size() - 1)) N <<= 1;
  poly f(N), g(N), d(h);
  d.resize(N), f[0] = modsqrt(d[0]);
  for (int w = 2; w < N; w <<= 1) {
    g.resize(w);
    for (int i = 0; i < w; ++i) g[i] = d[i];
    f = (f + f.inv(w) * g) / 2;
    f.resize(w);
  }
  f.resize(h.size());
  return f;
}

poly log(const poly &h) {
  poly f(h);
  for (int i = 1; i < (int)f.size(); ++i) f[i - 1] = (ll)f[i] * i % mod;
  f[f.size() - 1] = 0, f = f * h.inv(), f.resize(h.size());
  for (int i = (int)f.size() - 1; i > 0; --i)
    f[i] = (ll)f[i - 1] * qpow(i, mod - 2) % mod;
  f[0] = 0;
  return f;
}

poly exp(const poly &h) {
  int N = 1;
  while (N < (int)(h.size() + h.size() - 1)) N <<= 1;
  poly f(N), g(N), d(h);
  f[0] = 1, d.resize(N);
```

```
  for (int w = 2; w < N; w <<= 1) {
    f.resize(w), g.resize(w);
    for (int i = 0; i < w; ++i) g[i] = d[i];
    f = f * (g + 1 - log(f));
    f.resize(w);
  }
  f.resize(h.size());
  return f;
}

struct comp {
  long double x, y;

  comp(long double _x = 0, long double _y = 0) : x(_x), y(_y) {}

  comp operator*(const comp &b) const {
    return comp(x * b.x - y * b.y, x * b.y + y * b.x);
  }

  comp operator+(const comp &b) const { return comp(x + b.x, y + b.y); }

  comp operator-(const comp &b) const { return comp(x - b.x, y - b.y); }

  comp conj(void) { return comp(x, -y); }
};

const int EPS = 1e-9;

template <typename FLOAT_T>
FLOAT_T fabs(const FLOAT_T &x) {
  return x > 0 ? x : -x;
}

template <typename FLOAT_T>
FLOAT_T sin(const FLOAT_T &x, const long double &EPS = fstdlib::EPS) {
  FLOAT_T res = 0, delt = x;
  int d = 0;
  while (fabs(delt) > EPS) {
    res += delt, ++d;
    delt *= -x * x / ((2 * d) * (2 * d + 1));
  }
  return res;
}

template <typename FLOAT_T>
FLOAT_T cos(const FLOAT_T &x, const long double &EPS = fstdlib::EPS) {
  FLOAT_T res = 0, delt = 1;
  int d = 0;
  while (fabs(delt) > EPS) {
    res += delt, ++d;
    delt *= -x * x / ((2 * d) * (2 * d - 1));
  }
```

```
  return res;
}

const long double PI = std::acos((long double)(-1));

void dft_for_complex(std::vector<comp> &f, int n, int b) {
  static std::vector<comp> w;
  w.resize(n);
  for (int i = 0; i < n; ++i)
    if (i < rev[i]) std::swap(f[i], f[rev[i]]);
  for (int i = 2; i <= n; i <<= 1) {
    w[0] = comp(1, 0), w[1] = comp(cos(2 * PI / i), b * sin(2 * PI / i));
    for (int j = 2; j < i / 2; ++j) w[j] = w[j - 1] * w[1];
    for (int j = 0; j < n; j += i)
      for (int k = 0; k < i / 2; ++k) {
        comp p = f[j + k], q = f[j + k + i / 2] * w[k];
        f[j + k] = p + q, f[j + k + i / 2] = p - q;
      }
  }
}

class arbitrary_module_poly {
 private:
  std::vector<int> data;

  int construct_element(int D, ll x, ll y, ll z) const {
    x %= mod, y %= mod, z %= mod;
    return ((ll)D * D * x % mod + (ll)D * y % mod + z) % mod;
  }

 public:
  int mod;

  arbitrary_module_poly(std::size_t len = std::size_t(0),
                        int module_value = 1e9 + 7) {
    mod = module_value;
    data = std::vector<int>(len);
  }

  arbitrary_module_poly(const std::vector<int> &b, int module_value = 1e9 +
7) {
    mod = module_value;
    data = b;
  }

  arbitrary_module_poly(const arbitrary_module_poly &b) {
    mod = b.mod;
    data = b.data;
  }

  void resize(std::size_t len, const int &val = 0) { data.resize(len,
val); }
```

```
371
372   std::size_t size(void) const { return data.size(); }
373
374   void clear(void) { data.clear(); }
375 #if __cplusplus >= 201103L
376   void shrink_to_fit(void) { data.shrink_to_fit(); }
377 #endif
378   int &operator[](std::size_t b) { return data[b]; }
379
380   const int &operator[](std::size_t b) const { return data[b]; }
381
382   arbitrary_module_poly operator*(const arbitrary_module_poly &h) const;
383   arbitrary_module_poly operator*=(const arbitrary_module_poly &h);
384   arbitrary_module_poly operator*(const int &h) const;
385   arbitrary_module_poly operator*=(const int &h);
386   arbitrary_module_poly operator+(const arbitrary_module_poly &h) const;
387   arbitrary_module_poly operator+=(const arbitrary_module_poly &h);
388   arbitrary_module_poly operator-(const arbitrary_module_poly &h) const;
389   arbitrary_module_poly operator-=(const arbitrary_module_poly &h);
390   arbitrary_module_poly operator<<(const std::size_t &b) const;
391   arbitrary_module_poly operator<<=(const std::size_t &b);
392   arbitrary_module_poly operator>>(const std::size_t &b) const;
393   arbitrary_module_poly operator>>=(const std::size_t &b);
394   arbitrary_module_poly operator/(const int &h) const;
395   arbitrary_module_poly operator/=(const int &h);
396   arbitrary_module_poly operator==(const arbitrary_module_poly &h) const;
397   arbitrary_module_poly operator!=(const arbitrary_module_poly &h) const;
398   arbitrary_module_poly inv(void) const;
399   arbitrary_module_poly inv(const int &h) const;
400   friend arbitrary_module_poly sqrt(const arbitrary_module_poly &h);
401   friend arbitrary_module_poly log(const arbitrary_module_poly &h);
402 };
403
404 arbitrary_module_poly arbitrary_module_poly::operator*(
405     const arbitrary_module_poly &h) const {
406   int N = 1;
407   while (N < (int)(size() + h.size() - 1)) N <<= 1;
408   std::vector<comp> f(N), g(N), p(N), q(N);
409   const int D = std::sqrt(mod);
410   for (int i = 0; i < (int)size(); ++i)
411     f[i].x = data[i] / D, f[i].y = data[i] % D;
412   for (int i = 0; i < (int)h.size(); ++i) g[i].x = h[i] / D, g[i].y = h[i]
 % D;
413   rev.resize(N);
414   for (int i = 0; i < N; ++i)
415     rev[i] = (rev[i >> 1] >> 1) | (i & 1 ? N >> 1 : 0);
416   dft_for_complex(f, N, 1), dft_for_complex(g, N, 1);
417   for (int i = 0; i < N; ++i) {
418     p[i] = (f[i] + f[(N - i) % N].conj()) * comp(0.50, 0) * g[i];
419     q[i] = (f[i] - f[(N - i) % N].conj()) * comp(0, -0.5) * g[i];
420   }
421   dft_for_complex(p, N, -1), dft_for_complex(q, N, -1);
```

```
422   std::vector<int> r(size() + h.size() - 1);
423   for (int i = 0; i < (int)r.size(); ++i)
424     r[i] = construct_element(D, p[i].x / N + 0.5, (p[i].y + q[i].x) / N
    + 0.5,
425                              q[i].y / N + 0.5);
426   return arbitrary_module_poly(r, mod);
427 }
428
429 arbitrary_module_poly arbitrary_module_poly::operator*=(
430     const arbitrary_module_poly &h) {
431   return *this = *this * h;
432 }
433
434 arbitrary_module_poly arbitrary_module_poly::operator*(const int &h) const {
435   std::vector<int> f(this->data);
436   for (int i = 0; i < (int)f.size(); ++i) f[i] = (ll)f[i] * h % mod;
437   return arbitrary_module_poly(f, mod);
438 }
439
440 arbitrary_module_poly arbitrary_module_poly::operator*=(const int &h) {
441   for (int i = 0; i < (int)size(); ++i) data[i] = (ll)data[i] * h % mod;
442   return *this;
443 }
444
445 arbitrary_module_poly arbitrary_module_poly::operator+(
446     const arbitrary_module_poly &h) const {
447   std::vector<int> f(this->data);
448   if (f.size() < h.size()) f.resize(h.size());
449   for (int i = 0; i < (int)h.size(); ++i) f[i] = (f[i] + h[i]) % mod;
450   return arbitrary_module_poly(f, mod);
451 }
452
453 arbitrary_module_poly arbitrary_module_poly::operator+=(
454     const arbitrary_module_poly &h) {
455   if (size() < h.size()) resize(h.size());
456   for (int i = 0; i < (int)h.size(); ++i) data[i] = (data[i] + h[i]) % mod;
457   return *this;
458 }
459
460 arbitrary_module_poly arbitrary_module_poly::operator-(
461     const arbitrary_module_poly &h) const {
462   std::vector<int> f(this->data);
463   if (f.size() < h.size()) f.resize(h.size());
464   for (int i = 0; i < (int)h.size(); ++i) f[i] = (f[i] + mod - h[i]) % mod;
465   return arbitrary_module_poly(f, mod);
466 }
467
468 arbitrary_module_poly arbitrary_module_poly::operator-=(
469     const arbitrary_module_poly &h) {
470   if (size() < h.size()) resize(h.size());
471   for (int i = 0; i < (int)h.size(); ++i)
472     data[i] = (data[i] + mod - h[i]) % mod;
```

```
473   return *this;
474 }
475
476 arbitrary_module_poly arbitrary_module_poly::operator<<(
477     const std::size_t &b) const {
478   std::vector<int> f(size() + b);
479   for (int i = 0; i < (int)size(); ++i) f[i + b] = data[i];
480   return arbitrary_module_poly(f, mod);
481 }
482
483 arbitrary_module_poly arbitrary_module_poly::operator<<=(const std::size_t
&b) {
484   return *this = (*this) << b;
485 }
486
487 arbitrary_module_poly arbitrary_module_poly::operator>>(
488     const std::size_t &b) const {
489   std::vector<int> f(size() - b);
490   for (int i = 0; i < (int)f.size(); ++i) f[i] = data[i + b];
491   return arbitrary_module_poly(f, mod);
492 }
493
494 arbitrary_module_poly arbitrary_module_poly::operator>>=(const std::size_t
&b) {
495   return *this = (*this) >> b;
496 }
497
498 arbitrary_module_poly arbitrary_module_poly::inv(void) const {
499   int N = 1;
500   while (N < (int)(size() + size() - 1)) N <<= 1;
501   arbitrary_module_poly f(1, mod), g(N, mod), h(*this), f2(1, mod);
502   f[0] = qpow(data[0], mod - 2, mod), h.resize(N), f2[0] = 2;
503   for (int w = 2; w < N; w <<= 1) {
504     g.resize(w);
505     for (int i = 0; i < w; ++i) g[i] = h[i];
506     f = f * (f * g - f2) * (mod - 1);
507     f.resize(w);
508   }
509   f.resize(size());
510   return f;
511 }
512
513 arbitrary_module_poly arbitrary_module_poly::inv(const int &h) const {
514   arbitrary_module_poly f(*this);
515   f.resize(h);
516   return f.inv();
517 }
518
519 arbitrary_module_poly arbitrary_module_poly::operator/(const int &h) const {
520   int inv = qpow(h, mod - 2, mod);
521   std::vector<int> f(this->data);
522   for (int i = 0; i < (int)f.size(); ++i) f[i] = (ll)f[i] * inv % mod;
```

```
523   return arbitrary_module_poly(f, mod);
524 }
525
526 arbitrary_module_poly arbitrary_module_poly::operator/=(const int &h) {
527   int inv = qpow(h, mod - 2, mod);
528   for (int i = 0; i < (int)size(); ++i) data[i] = (ll)data[i] * inv % mod;
529   return *this;
530 }
531
532 arbitrary_module_poly arbitrary_module_poly::operator==(
533     const arbitrary_module_poly &h) const {
534   if (size() != h.size() || mod != h.mod) return 0;
535   for (int i = 0; i < (int)size(); ++i)
536     if (data[i] != h[i]) return 0;
537   return 1;
538 }
539
540 arbitrary_module_poly arbitrary_module_poly::operator!=(
541     const arbitrary_module_poly &h) const {
542   if (size() != h.size() || mod != h.mod) return 1;
543   for (int i = 0; i < (int)size(); ++i)
544     if (data[i] != h[i]) return 1;
545   return 0;
546 }
547
548 arbitrary_module_poly sqrt(const arbitrary_module_poly &h) {
549   int N = 1;
550   while (N < (int)(h.size() + h.size() - 1)) N <<= 1;
551   arbitrary_module_poly f(1, mod), g(N, mod), d(h);
552   f[0] = modsqrt(h[0], mod), d.resize(N);
553   for (int w = 2; w < N; w <<= 1) {
554     g.resize(w);
555     for (int i = 0; i < w; ++i) g[i] = d[i];
556     f = (f + f.inv(w) * g) / 2;
557     f.resize(w);
558   }
559   f.resize(h.size());
560   return f;
561 }
562
563 arbitrary_module_poly log(const arbitrary_module_poly &h) {
564   arbitrary_module_poly f(h);
565   for (int i = 1; i < (int)f.size(); ++i) f[i - 1] = (ll)f[i] * i % f.mod;
566   f[f.size() - 1] = 0, f = f * h.inv(), f.resize(h.size());
567   for (int i = (int)f.size() - 1; i > 0; --i)
568     f[i] = (ll)f[i - 1] * qpow(i, f.mod - 2, f.mod) % f.mod;
569   f[0] = 0;
570   return f;
571 }
572
573 using m_poly = arbitrary_module_poly;
574 }  // namespace fstdlib
```

```
575
576 #endif
```

实现

```
1  int main() {
2    scanf("%d", &n);
3    fact[0] = 1;
4    for (int i = 1; i <= n; ++i) fact[i] = (ll)fact[i - 1] * i % mod;
5    exgcd(fact[n], mod, ifact[n], ifact[0]),
6        ifact[n] = (ifact[n] % mod + mod) % mod;
7    for (int i = n - 1; i >= 0; --i) ifact[i] = (ll)ifact[i + 1] * (i + 1)
   % mod;
8    poly f(n + 1), g(n + 1);
9    for (int i = 0; i <= n; ++i)
10     g[i] = (i & 1 ? mod - 1ll : 1ll) * ifact[i] % mod,
11     f[i] = (ll)qpow(i, n) * ifact[i] % mod;
12   f *= g, f.resize(n + 1);
13   for (int i = 0; i <= n; ++i) printf("%d ", f[i]);
14   return 0;
15 }
```

同一列第二类斯特林数的计算

「同一列」的第二类斯特林数指的是，有着不同的 i，相同的 k 的一系列 $\left\{ {i \atop k} \right\}$. 求出同一列的所有第二类斯特林数，就是对 $i = 0..n$ 求出了将 i 个不同元素划分为 k 个非空集的方案数.

利用指数型生成函数计算.

一个盒子装 i 个物品且盒子非空的方案数是 $[i > 0]$. 我们可以写出它的指数型生成函数为 $F(x) = \sum\limits_{i=1}^{+\infty} \dfrac{x^i}{i!} = \mathrm{e}^x - 1$. 经过之前的学习，我们明白 $F^k(x)$ 就是 i 个有标号物品放到 k 个有标号盒子里的指数型生成函数，那么除掉 $k!$ 就是 i 个有标号物品放到 k 个无标号盒子里的指数型生成函数.

$\left\{ {i \atop k} \right\} = \dfrac{\left[\dfrac{x^i}{i!} \right] F^k(x)}{k!}$，$O(n \log n)$ 计算多项式幂即可.

另外，$\exp F(x) = \sum\limits_{i=0}^{+\infty} \dfrac{F^i(x)}{i!}$ 就是 i 个有标号物品放到任意多个无标号盒子里的指数型生成函数（EXP 通过每项除以一个 $i!$ 去掉了盒子的标号）. 这其实就是贝尔数的生成函数.

这里涉及到很多「有标号」「无标号」的内容，注意辨析.

实现

```
1  int main() {
2    scanf("%d%d", &n, &k);
3    poly f(n + 1);
4    fact[0] = 1;
5    for (int i = 1; i <= n; ++i) fact[i] = (ll)fact[i - 1] * i % mod;
6    for (int i = 1; i <= n; ++i) f[i] = qpow(fact[i], mod - 2);
7    f = exp(log(f >> 1) * k) << k, f.resize(n + 1);
```

```
  int inv = qpow(fact[k], mod - 2);
  for (int i = 0; i <= n; ++i)
    printf("%lld ", (ll)f[i] * fact[i] % mod * inv % mod);
  return 0;
}
```

第一类斯特林数（Stirling Number）

第一类斯特林数（斯特林轮换数）$\begin{bmatrix} n \\ k \end{bmatrix}$，也可记做 $s(n,k)$，表示将 n 个两两不同的元素，划分为 k 个互不区分的非空轮换的方案数.

一个轮换就是一个首尾相接的环形排列. 我们可以写出一个轮换 $[A,B,C,D]$，并且我们认为 $[A,B,C,D]=[B,C,D,A]=[C,D,A,B]=[D,A,B,C]$，即，两个可以通过旋转而互相得到的轮换是等价的. 注意，我们不认为两个可以通过翻转而相互得到的轮换等价，即 $[A,B,C,D]\neq[D,C,B,A]$.

递推式

$$\begin{bmatrix} n \\ k \end{bmatrix}=\begin{bmatrix} n-1 \\ k-1 \end{bmatrix}+(n-1)\begin{bmatrix} n-1 \\ k \end{bmatrix}$$

边界是 $\begin{bmatrix} n \\ 0 \end{bmatrix}=[n=0]$.

该递推式的证明可以考虑其组合意义.

我们插入一个新元素时，有两种方案：

- 将该新元素置于一个单独的轮换中，共有 $\begin{bmatrix} n-1 \\ k-1 \end{bmatrix}$ 种方案；
- 将该元素插入到任何一个现有的轮换中，共有 $(n-1)\begin{bmatrix} n-1 \\ k \end{bmatrix}$ 种方案.

根据加法原理，将两式相加即可得到递推式.

通项公式

第一类斯特林数没有实用的通项公式.

同一行第一类斯特林数的计算

类似第二类斯特林数，我们构造同行第一类斯特林数的生成函数，即

$F_n(x)=\sum\limits_{i=0}^{n}\begin{bmatrix} n \\ i \end{bmatrix}x^i$

根据递推公式，不难写出

$F_n(x)=(n-1)F_{n-1}(x)+xF_{n-1}(x)$

于是

$F_n(x)=\prod\limits_{i=0}^{n-1}(x+i)=\dfrac{(x+n-1)!}{(x-1)!}$

这其实是 x 的 n 次上升阶乘幂，记做 $x^{\overline{n}}$. 这个东西自然是可以暴力分治乘 $O(n\log^2 n)$ 求出的，但用上升幂相关做法可以 $O(n\log n)$ 求出，详情见 多项式平移 | 连续点值平移→9.12.11@p1575.

同一列第一类斯特林数的计算

仿照第二类斯特林数的计算，我们可以用指数型生成函数解决该问题. 注意，由于递推公式和行有关，我们不能利用递推公式计算同列的第一类斯特林数.

显然，单个轮换的指数型生成函数为

$F(x)=\sum\limits_{i=1}^{n}\dfrac{(i-1)!x^i}{i!}=\sum\limits_{i=1}^{n}\dfrac{x^i}{i}$

它的 k 次幂就是 $\begin{bmatrix} i \\ k \end{bmatrix}$ 的指数型生成函数，$O(n\log n)$ 计算即可.

实现

```
int main() {
  scanf("%d%d", &n, &k);
  fact[0] = 1;
  for (int i = 1; i <= n; ++i) fact[i] = (ll)fact[i - 1] * i % mod;
  ifact[n] = qpow(fact[n], mod - 2);
  for (int i = n - 1; i >= 0; --i) ifact[i] = (ll)ifact[i + 1] * (i + 1)
% mod;
  poly f(n + 1);
  for (int i = 1; i <= n; ++i) f[i] = (ll)fact[i - 1] * ifact[i] % mod;
  f = exp(log(f >> 1) * k) << k, f.resize(n + 1);
  for (int i = 0; i <= n; ++i)
    printf("%lld ", (ll)f[i] * fact[i] % mod * ifact[k] % mod);
  return 0;
}
```

应用

上升幂与普通幂的相互转化

我们记上升阶乘幂 $x^{\overline{n}} = \prod_{k=0}^{n-1}(x+k)$.

则可以利用下面的恒等式将上升幂转化为普通幂:

$$x^{\overline{n}} = \sum_k \begin{bmatrix} n \\ k \end{bmatrix} x^k$$

如果将普通幂转化为上升幂，则有下面的恒等式:

$$x^n = \sum_k \begin{Bmatrix} n \\ k \end{Bmatrix} (-1)^{n-k} x^{\overline{k}}$$

下降幂与普通幂的相互转化

我们记下降阶乘幂 $x^{\underline{n}} = \dfrac{x!}{(x-n)!} = \prod_{k=0}^{n-1}(x-k)$.

则可以利用下面的恒等式将普通幂转化为下降幂:

$$x^n = \sum_k \begin{Bmatrix} n \\ k \end{Bmatrix} x^{\underline{k}}$$

如果将下降幂转化为普通幂，则有下面的恒等式:

$$x^{\underline{n}} = \sum_k \begin{bmatrix} n \\ k \end{bmatrix} (-1)^{n-k} x^k$$

多项式下降阶乘幂表示与多项式点值表示的关系

在这里，多项式的下降阶乘幂表示就是用

$$f(x) = \sum_{i=0}^{n} b_i x^{\underline{i}}$$

的形式表示一个多项式，而点值表示就是用 $n+1$ 个点

$$(i, a_i), i = 0..n$$

来表示一个多项式.

显然，下降阶乘幂 b 和点值 a 间满足这样的关系:

$$a_k = \sum_{i=0}^{n} b_i k^i$$

即

$$a_k = \sum_{i=0}^{n} \frac{b_i k!}{(k-i)!}$$

$$\frac{a_k}{k!} = \sum_{i=0}^{k} b_i \frac{1}{(k-i)!}$$

这是一个卷积形式的式子，我们可以在 $O(n \log n)$ 的时间复杂度内完成点值和下降阶乘幂的互相转化.

习题

- HDU3625 Examining the Rooms[1]
- UOJ540 联合省选 2020 组合数问题[2]
- UOJ269 清华集训 2016 如何优雅地求和[3]

参考资料与注释

1. Stirling Number of the First Kind - Wolfram MathWorld[4]
2. Stirling Number of the Second Kind - Wolfram MathWorld[5]

外部参考资料

[1] HDU3625 Examining the Rooms

[2] UOJ540 联合省选 2020 组合数问题

[3] UOJ269 清华集训 2016 如何优雅地求和

[4] Stirling Number of the First Kind - Wolfram MathWorld

[5] Stirling Number of the Second Kind - Wolfram MathWorld

9.13.8 贝尔数

贝尔数 B_n 以埃里克·坦普尔·贝尔命名，是组合数学中的一组整数数列，开首是（OEIS A000110[1]）：

$$B_0 = 1, B_1 = 1, B_2 = 2, B_3 = 5, B_4 = 15, B_5 = 52, B_6 = 203, ...$$

B_n 是基数为 n 的集合的划分方法的数目. 集合 S 的一个划分是定义为 S 的两两不相交的非空子集的族，它们的并是 S. 例如 $B_3 = 5$ 因为 3 个元素的集合 a, b, c 有 5 种不同的划分方法：

$$\{\{a\}, \{b\}, \{c\}\}$$

$$\{\{a\}, \{b, c\}\}$$

$$\{\{b\}, \{a, c\}\}$$

$$\{\{c\}, \{a, b\}\}$$

$$\{\{a, b, c\}\}$$

B_0 是 1 因为空集正好有 1 种划分方法.

递推公式

贝尔数适合递推公式：

$$B_{n+1} = \sum_{k=0}^{n} \binom{n}{k} B_k$$

证明：

B_{n+1} 是含有 $n+1$ 个元素集合的划分个数，设 B_n 的集合为 $\{b_1, b_2, b_3, ..., b_n\}$，$B_{n+1}$ 的集合为 $\{b_1, b_2, b_3, ..., b_n, b_{n+1}\}$，那么可以认为 B_{n+1} 是有 B_n 增添了一个 b_{n+1} 而产生的，考虑元素 b_{n+1}.

- 假如它被单独分到一类，那么还剩下 n 个元素，这种情况下划分数为 $\binom{n}{n}B_n$
- 假如它和某 1 个元素分到一类，那么还剩下 $n-1$ 个元素，这种情况下划分数为 $\binom{n}{n-1}B_{n-1}$；
- 假如它和某 2 个元素分到一类，那么还剩下 $n-2$ 个元素，这种情况下划分数为 $\binom{n}{n-2}B_{n-2}$；
- ……

以此类推就得到了上面的公式.

每个贝尔数都是相应的 第二类斯特林数→9.13.7@p1645 的和. 因为第二类斯特林数是把基数为 n 的集合划分为正好 k 个非空集的方法数目.

$$B_n = \sum_{k=0}^{n} \left\{ {n \atop k} \right\}$$

贝尔三角形

用以下方法构造一个三角矩阵（形式类似杨辉三角形）：

- $a_{0,0} = 1$；
- 对于 $n \geq 1$，第 n 行首项等于上一行的末项，即 $a_{n,0} = a_{n-1,n-1}$；
- 对于 $m, n \geq 1$，第 n 行第 m 项等于它左边和左上角两个数之和，即 $a_{n,m} = a_{n,m-1} + a_{n-1,m-1}$.

部分结果如下：

1						
1	2					
2	3	5				
5	7	10	15			
15	20	27	37	52		
52	67	87	114	151	203	
203	255	322	409	523	674	877

每行的首项是贝尔数. 可以利用这个三角形来递推求出贝尔数.

参考实现

- C++

```cpp
constexpr int MAXN = 2000 + 5;
int bell[MAXN][MAXN];

void f(int n) {
  bell[0][0] = 1;
  for (int i = 1; i <= n; i++) {
    bell[i][0] = bell[i - 1][i - 1];
    for (int j = 1; j <= i; j++)
      bell[i][j] = bell[i - 1][j - 1] + bell[i][j - 1];
  }
}
```

- Python

```python
MAXN = 2000 + 5
bell = [[0 for i in range(MAXN + 1)] for j in range(MAXN + 1)]


def f(n):
    bell[0][0] = 1
    for i in range(1, n + 1):
```

```
        bell[i][0] = bell[i - 1][i - 1]
        for j in range(1, i + 1):
            bell[i][j] = bell[i - 1][j - 1] + bell[i][j - 1]
```

指数生成函数

考虑贝尔数的指数生成函数及其导函数：

$$\begin{aligned}\hat{B}(x) &= \sum_{n=0}^{+\infty} \frac{B_n}{n!} x^n \\ &= 1 + \sum_{n=0}^{+\infty} \frac{B_{n+1}}{(n+1)!} x^{n+1} \\ \hat{B}'(x) &= \sum_{n=0}^{+\infty} \frac{B_{n+1}}{n!} x^n\end{aligned}$$

根据贝尔数的递推公式可以得到：

$$\frac{B_{n+1}}{n!} = \sum_{k=0}^{n} \frac{1}{(n-k)!} \frac{B_k}{k!}$$

这是一个卷积的式子，因此有：

$$\hat{B}'(x) = \mathrm{e}^x \hat{B}(x)$$

这是一个微分方程，解得：

$$\hat{B}(x) = \exp(\mathrm{e}^x + C)$$

最后当 $x = 0$ 时 $\hat{B}(x) = 1$，带入后解得 $C = -1$，得到贝尔数指数生成函数的封闭形式：

$$\hat{B}(x) = \exp(\mathrm{e}^x - 1)$$

预处理出 $\mathrm{e}^x - 1$ 的前 n 项后做一次 多项式 exp→9.12.9@p1559 即可得出贝尔数前 n 项，时间复杂度瓶颈在多项式 exp，可做到 $O(n \log n)$ 的时间复杂度.

参考文献

https://en.wikipedia.org/wiki/Bell_number[2]

外部参考资料

[1] OEIS A000110

[2] https://en.wikipedia.org/wiki/Bell_number

9.13.9 伯努利数

伯努利数 B_n 是一个与数论有密切关联的有理数序列．前几项被发现的伯努利数分别为：

$B_0 = 1, B_1 = -\frac{1}{2}, B_2 = \frac{1}{6}, B_3 = 0, B_4 = -\frac{1}{30}, \ldots$

等幂求和

伯努利数是由雅各布·伯努利的名字命名的，他在研究 m 次幂和的公式时发现了奇妙的关系．我们记

$$S_m(n) = \sum_{k=0}^{n-1} k^m = 0^m + 1^m + \ldots + (n-1)^m$$

伯努利观察了如下一列公式，勾画出一种模式：

$$S_0(n) = n$$

$$S_1(n) = \frac{1}{2}n^2 - \frac{1}{2}n$$

$$S_2(n) = \frac{1}{3}n^3 - \frac{1}{2}n^2 + \frac{1}{6}n$$

$$S_3(n) = \frac{1}{4}n^4 - \frac{1}{2}n^3 + \frac{1}{4}n^2$$

$$S_4(n) = \frac{1}{5}n^5 - \frac{1}{2}n^4 + \frac{1}{3}n^3 - \frac{1}{30}n$$

可以发现，在 $S_m(n)$ 中 n^{m+1} 的系数总是 $\frac{1}{m+1}$，n^m 的系数总是 $-\frac{1}{2}$，n^{m-1} 的系数总是 $\frac{m}{12}$，n^{m-3} 的系数是 $-\frac{m(m-1)(m-2)}{720}$，n^{m-4} 的系数总是零等.

而 n^{m-k} 的系数总是某个常数乘以 $m^{\underline{k}}$，$m^{\underline{k}}$ 表示下降阶乘幂，即 $\frac{m!}{(m-k)!}$.

递推公式

$$\begin{aligned} S_m(n) &= \frac{1}{m+1}(B_0 n^{m+1} + \binom{m+1}{1} B_1 n^m + ... + \binom{m+1}{m} B_m n) \\ &= \frac{1}{m+1}\sum_{k=0}^{m}\binom{m+1}{k} B_k n^{m+1-k} \end{aligned}$$

伯努利数由隐含的递推关系定义：

$$\sum_{j=0}^{m}\binom{m+1}{j} B_j = 0, (m > 0)$$

$$B_0 = 1$$

例如，$\binom{2}{0}B_0 + \binom{2}{1}B_1 = 0$，前几个值显然是

n	0	1	2	3	4	5	6	7	8	...
B_n	1	$-\frac{1}{2}$	$\frac{1}{6}$	0	$-\frac{1}{30}$	0	$\frac{1}{42}$	0	$-\frac{1}{30}$	...

证明

利用归纳法证明

这个证明方法来自 Concrete Mathematics 6.5 BERNOULLI NUMBER.

运用二项式系数的恒等变换和归纳法进行证明：

$$\begin{aligned} S_{m+1}(n) + n^{m+1} &= \sum_{k=0}^{n-1}(k+1)^{m+1} \\ &= \sum_{k=0}^{n-1}\sum_{j=0}^{m+1}\binom{m+1}{j} k^j \\ &= \sum_{j=0}^{m+1}\binom{m+1}{j} S_j(n) \end{aligned}$$

令 $\hat{S}_m(n) = \frac{1}{m+1}\sum_{k=0}^{m}\binom{m+1}{k} B_k n^{m+1-k}$, 我们希望证明 $S_m(n) = \hat{S}_m(n)$, 假设对 $j \in [0, m)$, 有 $S_j(n) = \hat{S}_j(n)$.

将原式中两边都减去 $S_{m+1}(n)$ 后可以得到：

$$S_{m+1}(n)+n^{m+1}=\sum_{j=0}^{m+1}\binom{m+1}{j}S_j(n)$$

$$n^{m+1}=\sum_{j=0}^{m}\binom{m+1}{j}S_j(n)$$

$$=\sum_{j=0}^{m-1}\binom{m+1}{j}\hat{S}_j(n)+\binom{m+1}{m}S_m(n)$$

尝试在式子的右边加上 $\binom{m+1}{m}\hat{S}_m(n)-\binom{m+1}{m}\hat{S}_m(n)$ 再进行化简，可以得到：

$$n^{m+1}=\sum_{j=0}^{m}\binom{m+1}{j}\hat{S}_j(n)+(m+1)(S_m(n)-\hat{S}_m(n))$$

不妨设 $\Delta=S_m(n)-\hat{S}_m(n)$，并且将 $\hat{S}_j(n)$ 展开，那么有

$$n^{m+1}=\sum_{j=0}^{m}\binom{m+1}{j}\hat{S}_j(n)+(m+1)\Delta$$

$$=\sum_{j=0}^{m}\binom{m+1}{j}\frac{1}{j+1}\sum_{k=0}^{j}\binom{j+1}{k}B_k n^{j+1-k}+(m+1)\Delta$$

将第二个 $\sum$ 中的求和顺序改为逆向，再将组合数的写法恒等变换可以得到：

$$n^{m+1}=\sum_{j=0}^{m}\binom{m+1}{j}\frac{1}{j+1}\sum_{k=0}^{j}\binom{j+1}{j-k}B_{j-k}n^{k+1}+(m+1)\Delta$$

$$=\sum_{j=0}^{m}\binom{m+1}{j}\frac{1}{j+1}\sum_{k=0}^{j}\binom{j+1}{k+1}B_{j-k}n^{k+1}+(m+1)\Delta$$

$$=\sum_{j=0}^{m}\binom{m+1}{j}\frac{1}{j+1}\sum_{k=0}^{j}\frac{j+1}{k+1}\binom{j}{k}B_{j-k}n^{k+1}+(m+1)\Delta$$

$$=\sum_{j=0}^{m}\binom{m+1}{j}\sum_{k=0}^{j}\binom{j}{k}\frac{B_{j-k}}{k+1}n^{k+1}+(m+1)\Delta$$

对两个求和符号进行交换，可以得到：

$$n^{m+1}=\sum_{k=0}^{m}\frac{n^{k+1}}{k+1}\sum_{j=k}^{m}\binom{m+1}{j}\binom{j}{k}B_{j-k}+(m+1)\Delta$$

对 $\binom{m+1}{j}\binom{j}{k}$ 进行恒等变换：

$$\binom{m+1}{j}\binom{j}{k}=\binom{m+1}{k}\binom{m-k+1}{j-k}$$

那么式子就变成了：

$$n^{m+1}=\sum_{k=0}^{m}\frac{n^{k+1}}{k+1}\sum_{j=k}^{m}\binom{m+1}{k}\binom{m-k+1}{j-k}B_{j-k}+(m+1)\Delta$$

$$=\sum_{k=0}^{m}\frac{n^{k+1}}{k+1}\binom{m+1}{k}\sum_{j=k}^{m}\binom{m-k+1}{j-k}B_{j-k}+(m+1)\Delta$$

将所有的 $j-k$ 用 j 代替，那么就可以得到：

$$n^{m+1}=\sum_{k=0}^{m}\frac{n^{k+1}}{k+1}\binom{m+1}{k}\sum_{j=0}^{m-k}\binom{m-k+1}{j}B_j+(m+1)\Delta$$

考虑我们前面提到过的递归关系

$$\sum_{j=0}^{m}\binom{m+1}{j}B_j=0,(m>0)$$

$$B_0=1$$

$$\sum_{j=0}^{m}\binom{m+1}{j}B_j=[m=0]$$

代入后可以得到:

$$\begin{aligned}n^{m+1}&=\sum_{k=0}^{m}\frac{n^{k+1}}{k+1}\binom{m+1}{k}[m-k=0]+(m+1)\Delta\\&=\frac{n^{m+1}}{m+1}\binom{m+1}{m}+(m+1)\Delta\\&=n^{m+1}+(m+1)\Delta\end{aligned}$$

于是 $\Delta=0$，且有 $S_m(n)=\hat{S}_m(n)$.

利用指数生成函数证明

对递推式 $\sum_{j=0}^{m}\binom{m+1}{j}B_j=[m=0]$

两边都加上 B_{m+1}，即得到:

$$\sum_{j=0}^{m+1}\binom{m+1}{j}B_j=[m=0]+B_{m+1}$$

$$\sum_{j=0}^{m}\binom{m}{j}B_j=[m=1]+B_m$$

$$\sum_{j=0}^{m}\frac{B_j}{j!}\cdot\frac{1}{(m-j)!}=[m=1]+\frac{B_m}{m!}$$

设 $B(z)=\sum\limits_{i\geq 0}\dfrac{B_i}{i!}z^i$，注意到左边为卷积形式，故:

$$B(z)\mathrm{e}^z=z+B(z)$$

$$B(z)=\frac{z}{\mathrm{e}^z-1}$$

设 $F_n(z)=\sum_{m\geq 0}\dfrac{S_m(n)}{m!}z^m$，则:

$$\begin{aligned}F_n(z)&=\sum_{m\geq 0}\frac{S_m(n)}{m!}z^m\\&=\sum_{m\geq 0}\sum_{i=0}^{n-1}\frac{i^m z^m}{m!}\end{aligned}$$

调换求和顺序:

$$\begin{aligned}F_n(z)&=\sum_{i=0}^{n-1}\sum_{m\geq 0}\frac{i^m z^m}{m!}\\&=\sum_{i=0}^{n-1}\mathrm{e}^{iz}\\&=\frac{\mathrm{e}^{nz}-1}{\mathrm{e}^z-1}\\&=\frac{z}{\mathrm{e}^z-1}\cdot\frac{\mathrm{e}^{nz}-1}{z}\end{aligned}$$

代入 $B(z)=\dfrac{z}{\mathrm{e}^z-1}$:

$$
\begin{aligned}
F_n(z) &= B(z) \cdot \frac{\mathrm{e}^{nz}-1}{z} \\
&= \left(\sum_{i\geq 0} \frac{B_i}{i!}\right)\left(\sum_{i\geq 1} \frac{n^i z^{i-1}}{i!}\right) \\
&= \left(\sum_{i\geq 0} \frac{B_i}{i!}\right)\left(\sum_{i\geq 0} \frac{n^{i+1} z^{i}}{(i+1)!}\right)
\end{aligned}
$$

由于 $F_n(z) = \sum_{m\geq 0} \frac{S_m(n)}{m!} z^m$，即 $S_m(n) = m![z^m]F_n(z)$：

$$
\begin{aligned}
S \times m(n) &= m![z^m]F_n(z) \\
&= m! \sum_{i=0}^{m} \frac{B \times i}{i!} \cdot \frac{n^{m-i+1}}{(m-i+1)!} \\
&= \frac{1}{m+1} \sum_{i=0}^{m} \binom{m+1}{i} B_i n^{m-i+1}
\end{aligned}
$$

故得证.

参考实现

```cpp
using ll = long long;
constexpr int MAXN = 10000;
constexpr int mod = 1e9 + 7;
ll B[MAXN];         // 伯努利数
ll C[MAXN][MAXN];   // 组合数
ll inv[MAXN];       // 逆元（计算伯努利数）

void init() {
  // 预处理组合数
  for (int i = 0; i < MAXN; i++) {
    C[i][0] = C[i][i] = 1;
    for (int k = 1; k < i; k++) {
      C[i][k] = (C[i - 1][k] % mod + C[i - 1][k - 1] % mod) % mod;
    }
  }
  // 预处理逆元
  inv[1] = 1;
  for (int i = 2; i < MAXN; i++) {
    inv[i] = (mod - mod / i) * inv[mod % i] % mod;
  }
  // 预处理伯努利数
  B[0] = 1;
  for (int i = 1; i < MAXN; i++) {
    ll ans = 0;
    if (i == MAXN - 1) break;
    for (int k = 0; k < i; k++) {
      ans += C[i + 1][k] * B[k];
      ans %= mod;
    }
    ans = (ans * (-inv[i + 1]) % mod + mod) % mod;
    B[i] = ans;
  }
}
```

9.13.10 Entringer Number

恩特林格数

恩特林格数（Entringer number，OEIS A008281[1]）$E(n,k)$ 是满足下述条件的 0 到 n 共 $n+1$ 个数的置换数目：

- 首元素是 k；
- 首元素的下一个元素比首元素小，再下一个元素比前一个元素大，再下一个元素比前一个元素小……后面相邻元素的大小关系均满足这样的规则.

恩特林格数的初值有：

$$E(0,0)=1$$
$$E(n,0)=0$$

有递推关系：

$$E(n,k)=E(n,k-1)+E(n-1,n-k)$$

Seidel–Entringer–Arnold 三角

恩特林格数的一个适当排列的数字三角，称为 Seidel–Entringer–Arnold 三角（Seidel–Entringer–Arnold triangle，OEIS A008280[2]）. 该三角是按照「牛耕」顺序（ox-plowing order）排列的恩特林格数 $E_(n,k)$：

$$E(0,0)$$
$$E(1,0)\to E(1,1)$$
$$E(2,2)\leftarrow E(2,1)\leftarrow E(2,0)$$
$$E(3,0)\to E(3,1)\to E(3,2)\to E(3,3)$$
$$E(4,4)\leftarrow E(4,3)\leftarrow E(4,2)\leftarrow E(4,1)\leftarrow E(4,0)$$

即：

$$1$$
$$0\to 1$$
$$1\leftarrow 1\leftarrow 0$$
$$0\to 1\to 2\to 2$$
$$5\leftarrow 5\leftarrow 4\leftarrow 2\leftarrow 0$$

按照这种方式排列的恩特林格数的优势是，与它的递推关系 $E(n,k)=E(n,k-1)+E(n-1,n-k)$ 一致，可以方便记忆和理解.

恩特林格数有一个指数型生成函数：

$$\sum_{m=0}^{\infty}\sum_{n=0}^{\infty}E\left(m+n,\frac{1}{2}(m+n+(-1)^{m+n}(n-m))\right)\frac{x^m}{m!}\frac{x^n}{n!}=\frac{\cos x+\sin x}{\cos(x+y)}$$

这个生成函数的系数分布事实上是上面的 Seidel–Entringer–Arnold 三角的简单拉伸变形：

$$\begin{array}{ccccc}E(0,0) & E(1,1) & E(2,0) & E(3,3) & E(4,0)\\ E(1,0) & E(2,1) & E(3,2) & E(4,1) & \\ E(2,2) & E(3,1) & E(4,2) & & \\ E(3,0) & E(4,3) & & & \\ E(4,4) & & & & \end{array}$$

即：

$$\begin{array}{ccccc}1 & 1 & 0 & 2 & 0\\ 0 & 1 & 2 & 2 & \\ 1 & 1 & 4 & & \\ 0 & 5 & & & \\ 5 & & & & \end{array}$$

zigzag 置换

一个 zigzag 置换（zigzag permutation）是一个 1 到 n 的排列 c_1 到 c_i，使得任意一个元素 c_i 的大小都不介于 c_{i-1} 和 c_{i+1} 之间.

对于 zigzag 置换的个数 Z_n（OEIS A001250[3]），从 $n=0$ 开始有：

$$1,1,2,4,10,32,122,544,\cdots$$

例如，前几个 n 的交替置换有：

$$n=1:\{1\}$$

$$n=2:\{1,2\},\{2,1\}$$

$$n=3:\{1,3,2\},\{2,1,3\},\{2,3,1\},\{3,1,2\}$$

$$n=4:\{1,3,2,4\},\{1,4,2,3\},\{2,1,4,3\},\{2,3,1,4\},\{2,4,1,3\},$$
$$\{3,1,4,2\},\{3,2,4,1\},\{3,4,1,2\},\{4,1,3,2\},\{4,2,3,1\}$$

交替置换与 zigzag 数

（注意和「错位排列」进行概念上的区分.）

对于大于 1 的 n，每个 zigzag 置换翻转过来仍旧为 zigzag 置换，可以两两配对，所以必然为偶数.

这里再给出一种配对的方法：将 zigzag 置换分为交替置换（alternating permutation）和反交替置换（reverse alternating permutation）.

交替置换的首元素大于第二个元素，大小关系为：

$$c_1>c_2<c_3>\cdots$$

反交替置换的首元素小于第二个元素，大小关系为：

$$c_1<c_2>c_3<\cdots$$

如果将 1 和 n 位置互换，2 和 $n-1$ 位置互换，以此类推，即可将交替置换与反交替置换两个集合互换. 因此，交替置换与反交替置换的个数相等，恰好为 zigzag 置换的一半.

对于大于 1 的 n，记：

$$A_n=\frac{Z_n}{2}$$

定义初值：

$$A_0=A_1=1$$

这里的 A_n 称为 zigzag 数（Euler zigzag number，OEIS A000111[4]），从 $n=0$ 开始有：

$$1,1,1,2,5,16,61,272,\cdots$$

接下来试着求解 A_n.

从 1 到 n 之中，选取 k 个数构成子集，有 $\binom{n}{k}$ 种选法.

在这个 k 元子集中，选反交替置换 u，有 A_k 种选法；用全集减掉这个 k 元子集，剩余的 $n-k$ 元子集中，选反交替置换 v，有 A_{n-k} 种选法.

考虑 $n+1$ 元排列 w，将 u 倒置作为开头，接上 $n+1$，再接上 v. 那么，w 一定是 zigzag 置换，并且任意一个 $n+1$ 元 zigzag 置换，都可以在 $n+1$ 处截断得到对应的反交替置换 u 和 v，并且不同的 $n+1$ 元 zigzag 置换对应的 u 和 v 不同.

因此有递推关系：

$$2A_{n+1}=\sum_{k=0}^{n}\binom{n}{k}A_kA_{n-k}$$

$$2(n+1)\frac{A_{n+1}}{(n+1)!}=\sum_{k=0}^{n}\frac{A_k}{k!}\frac{A_{n-k}}{(n-k)!}$$

当 n 为 0 时并不满足这个递推式，初值 A_0 和 A_1 都是 1.

可见，这是一个指数型生成函数的卷积．假设 A_n 的指数型生成函数为 y，就有微分方程：

$$2\frac{\mathrm{d}y}{\mathrm{d}x} = y^2 + 1$$

等式右面加 1 是为了处理 n 为 0 时的特殊情况．该方程的通解为：

$$y = \tan\left(\frac{1}{2}x + C\right)$$

代入第 0 项为 1 之后，可以得到特解：

$$y = \tan x + \sec x$$

正切函数是奇函数，正割函数是偶函数，两者之和构成 zigzag 数的生成函数.

恩特林格数与 zigzag 数的关系

根据恩特林格数的定义，恩特林格数 $E(n,k)$ 是首元素为 k 的 0 到 n 的交替置换个数．因此恩特林格数与 zigzag 数事实上有关系：

$$A_n = E(n,n)$$

将 A_n 称为「zigzag 数」也有原因：记 E_n 是欧拉数（Euler number），B_n 是伯努利数.

当 n 为偶数时，偶数项下标的 zigzag 数也称「正割数」S_n 或者「zig 数」．有关系：

$$A_n = (-1)^{n/2}E_n$$

前几项为（OEIS A000364[5]）：

$$1, 1, 5, 61, 1385, \cdots$$

当 n 为奇数时，奇数项下标的 zigzag 数也称「正切数」T_n 或者「zag 数」．有关系：

$$A_n = \frac{(-1)^{(n-1)/2}2^{n+1}(2^{n+1}-1)B_{n+1}}{n+1}$$

前几项为（OEIS A000182[6]）：

$$1, 2, 16, 272, 7936, \cdots$$

于是对于在 $x = 0$ 处的泰勒展开，可以给出正割数和正切数：

$$\sec x = A_0 + A_2\frac{x^2}{2!} + A_4\frac{x^4}{4!} + \cdots$$

$$\tan x = A_1 x + A_3\frac{x^3}{3!} + A_5\frac{x^5}{5!} + \cdots$$

或者写到一起：

$$\sec x + \tan x = A_0 + A_1 x + A_2\frac{x^2}{2!} + A_3\frac{x^3}{3!} + A_4\frac{x^4}{4!} + A_5\frac{x^5}{5!} + \cdots$$

构成 zigzag 数的生成函数.

参考资料与链接

1. Alternating permutation - Wikipedia[7]

外部参考资料

[1] OEIS A008281

[2] OEIS A008280

[3] OEIS A001250

[4] OEIS A000111

[5] OEIS A000364

[6] OEIS A000182

[7] Alternating permutation - Wikipedia

9.13.11 Eulerian Number

> ⚠ **注意**
>
> 下文中的欧拉数特指 Eulerian number．注意与 Euler number，以及 Euler's number（指与欧拉相关的数学常数例如 γ 或 e）作区分．

在计算组合中，**欧拉数**（Eulerian Number）是从 1 到 n 中正好满足 m 个元素大于前一个元素（具有 m 个「上升」的排列）条件的排列 **个数**．定义为：

$$A(n,m) = \left\langle {n \atop m-1} \right\rangle$$

例如，从数字 1 到 3 一共有 4 种排列使得恰好有一个元素比前一个元素大：

排列	满足条件的相邻元素	个数
1 2 3	1, 2 & 2, 3	2
1 3 2	1, 3	1
2 1 3	1, 3	1
2 3 1	2, 3	1
3 1 2	1, 2	1
3 2 1		0

所以按照 $A(n,m)$ 定义：如果 n 等于 3，m 等于 1，欧拉数值为 4，表示共有 4 个有 1 个元素大于前一个元素的排列．

对于 n 和 m 值比较小的欧拉数来说，我们可以直接得到结果：

$A(n,m)$	满足要求的排列	个数
$A(1,0)$	(1)	1
$A(2,0)$	$(2,1)$	1
$A(2,1)$	$(1,2)$	1
$A(3,0)$	$(3,2,1)$	1
$A(3,1)$	$(1,3,2),(2,1,3),(2,3,1),(3,1,2)$	4
$A(3,2)$	$(1,2,3)$	1

公式

可以通过递推或者递归的方法计算欧拉数．

首先，当 $m \geq n$ 或 $n = 0$ 时，没有满足条件的排列，即此时欧拉数为 0．

其次，当 $m = 0$ 时，只有降序的排列满足条件，即此时欧拉数为 1．

最后，考虑在 $n-1$ 的排列的基础上插入 n 从而得到 n 的排列，由于插入 n 至多使欧拉数增加 1，所以 $A(n,m)$ 可以仅从 $A(n-1,m-1)$ 处和 $A(n-1,m)$ 处转移得到．

考虑 n 插入的位置：当 $p_{i-1} < p_i$ 时，若将 n 插到 p_i 之前，即将 n 插入到「上升」中，排列的欧拉数不变；此外，将 n 插在排列之前，排列的欧拉数也不变；否则，若将 n 插到其余位置，排列的欧拉数增加 1．

考虑从 $A(n-1,m-1)$ 转移到 $A(n,m)$，此时需要使欧拉数增加 1，此时不能将 n 插在「上升」中或者排列开头，共有 $n-(m-1)-1 = n-m$ 种方案．

考虑从 $A(n-1,m)$ 转移到 $A(n,m)$，此时需要欧拉数保持不变，只能将 n 插在「上升」中或者排列开头，共 $m+1$ 种方案．

综上所述，有

$$A(n,m)=\begin{cases}0, & m>n \text{ or } n=0,\\ 1, & m=0,\\ (n-m)\cdot A(n-1,m-1)+(m+1)\cdot A(n-1,m), & \text{otherwise.}\end{cases}$$

实现

- C++

```cpp
int eulerianNumber(int n, int m) {
  if (m >= n || n == 0) return 0;
  if (m == 0) return 1;
  return (((n - m) * eulerianNumber(n - 1, m - 1)) +
          ((m + 1) * eulerianNumber(n - 1, m)));
}
```

- Python

```python
def eulerianNumber(n, m):
    if m >= n or n == 0:
        return 0
    if m == 0:
        return 1
    return ((n - m) * eulerianNumber(n - 1, m - 1)) + (
        (m + 1) * eulerianNumber(n - 1, m)
    )
```

习题

- CF1349F1 Slime and Sequences (Easy Version)[1]
- CF1349F2 Slime and Sequences (Hard Version)[2]
- UOJ 593. 新年的军队[3]
- P7511 三到六[4]

外部参考资料

[1] CF1349F1 Slime and Sequences (Easy Version)

[2] CF1349F2 Slime and Sequences (Hard Version)

[3] UOJ 593. 新年的军队

[4] P7511 三到六

9.13.12 分拆数

分拆：将自然数 n 写成递降正整数和的表示.

$$n=r_1+r_2+\ldots+r_k \quad r_1\geq r_2\geq\ldots\geq r_k\geq 1$$

和式中每个正整数称为一个部分.

分拆数：p_n. 自然数 n 的分拆方法数.

自 0 开始的分拆数：

n	0	1	2	3	4	5	6	7	8
p_n	1	1	2	3	5	7	11	15	22

k 部分拆数

将 n 分成恰有 k 个部分的分拆，称为 k 部分拆数，记作 $p(n,k)$.

显然，k 部分拆数 $p(n,k)$ 同时也是下面方程的解数：

$$n-k=y_1+y_2+...+y_k \quad y_1 \geq y_2 \geq ... \geq y_k \geq 0$$

如果这个方程里面恰有 j 个部分非 0，则恰有 $p(n-k,j)$ 个解．因此有和式：

$$p(n,k)=\sum_{j=0}^{k} p(n-k,j)$$

相邻两个和式作差，得：

$$p(n,k)=p(n-1,k-1)+p(n-k,k)$$

如果列出表格，每个格里的数，等于左上方的数，加上该格向上方数，所在列数个格子中的数．

k	0	1	2	3	4	5	6	7	8
$p(0,k)$	1	0	0	0	0	0	0	0	0
$p(1,k)$	0	1	0	0	0	0	0	0	0
$p(2,k)$	0	1	1	0	0	0	0	0	0
$p(3,k)$	0	1	1	1	0	0	0	0	0
$p(4,k)$	0	1	2	1	1	0	0	0	0
$p(5,k)$	0	1	2	2	1	1	0	0	0
$p(6,k)$	0	1	3	3	2	1	1	0	0
$p(7,k)$	0	1	3	4	3	2	1	1	0
$p(8,k)$	0	1	4	5	5	3	2	1	1

例题

计算 k 部分拆数

计算 k 部分拆数 $p(n,k)$．多组输入，其中 n 上界为 10000，k 上界为 1000，对 1000007 取模．

观察表格与递推式，按列更新对于存储更有利．不难写出程序：

```
#include <cstdio>
#include <cstring>

int p[10005][1005]; /*将自然数n分拆为k个部分的方法数*/

int main() {
  int n, k;
  while (~scanf("%d%d", &n, &k)) {
    memset(p, 0, sizeof(p));
    p[0][0] = 1;
    int i;
    for (i = 1; i <= n; ++i) {
      int j;
      for (j = 1; j <= k; ++j) {
        if (i - j >= 0) /*p[i-j][j]所有部分大于1*/
        {
          p[i][j] = (p[i - j][j] + p[i - 1][j - 1]) %
                    1000007; /*p[i-1][j-1]至少有一个部分为1. */
        }
      }
    }
    printf("%d\n", p[n][k]);
  }
```

```
24 }
```

生成函数

由等比数列求和公式，有：

$$\frac{1}{1-x^k}=1+x^k+x^{2k}+x^{3k}+...$$

$$1+p_1x+p_2x^2+p_3x^3+...=\frac{1}{1-x}\frac{1}{1-x^2}\frac{1}{1-x^3}...$$

对于 k 部分拆数，生成函数稍微复杂．具体写出如下：

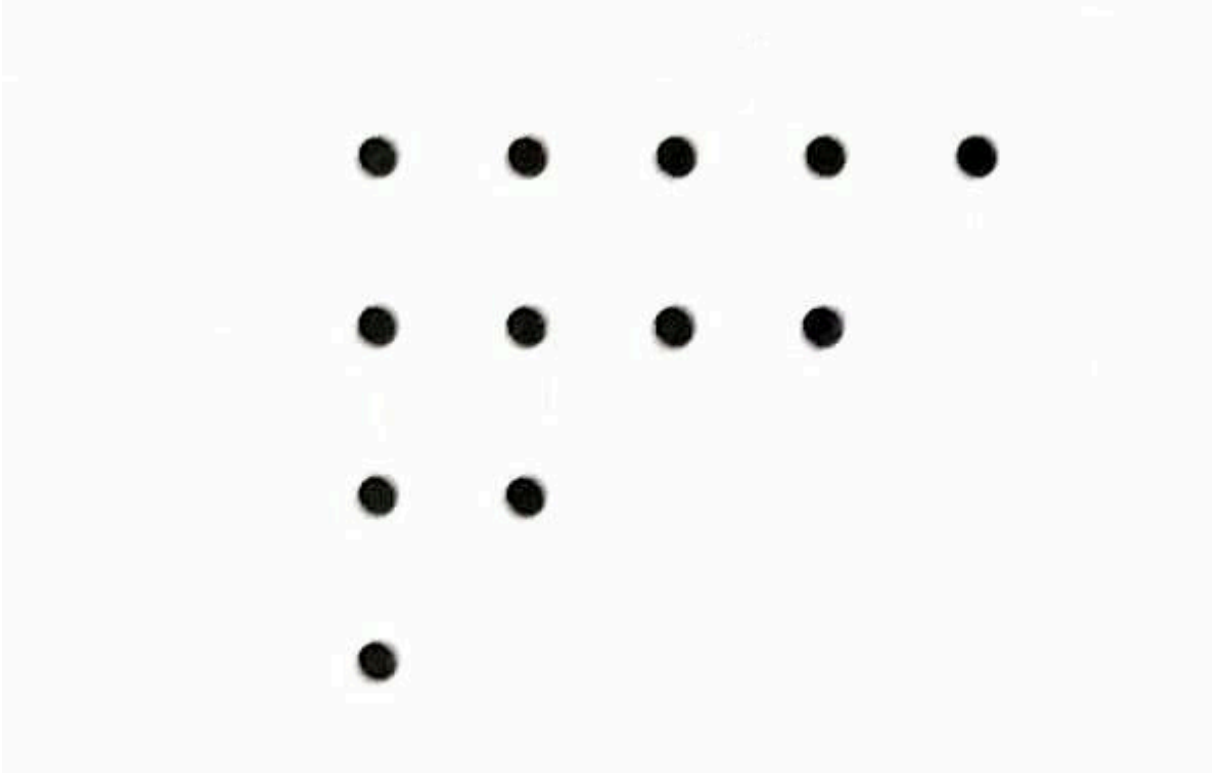
$$\sum_{n,k=0}^{\infty}p(n,k)x^ny^k=\frac{1}{1-xy}\frac{1}{1-x^2y}\frac{1}{1-x^3y}...$$

Ferrers 图

Ferrers 图：将分拆的每个部分用点组成的行表示．每行点的个数为这个部分的大小．

根据分拆的定义，Ferrers 图中不同的行按照递减的次序排放．最长行在最上面．

例如：分拆 $12=5+4+2+1$ 的 Ferrers 图．



将一个 Ferrers 图沿着对角线翻转，得到的新 Ferrers 图称为原图的共轭，新分拆称为原分拆的共轭．显然，共轭是对称的关系．

例如上述分拆 $12=5+4+2+1$ 的共轭是分拆 $12=4+3+2+2+1$.

最大 k 分拆数：自然数 n 的最大部分为 k 的分拆个数．

根据共轭的定义，有显然结论：

最大 k 分拆数与 k 部分拆数相同，均为 $p(n,k)$.

互异分拆数

互异分拆数：pd_n. 自然数 n 的各部分互不相同的分拆方法数.（Different）

n	0	1	2	3	4	5	6	7	8
pd_n	1	1	1	2	2	3	4	5	6

同样地，定义互异 k 部分拆数 $pd(n,k)$，表示最大拆出 k 个部分的互异分拆，是这个方程的解数：

$$n = r_1 + r_2 + ... + r_k \quad r_1 > r_2 > ... > r_k \geq 1$$

完全同上，也是这个方程的解数：

$$n - k = y_1 + y_2 + ... + y_k \quad y_1 > y_2 > ... > y_k \geq 0$$

这里与上面不同的是，由于互异，新方程中至多只有一个部分为零. 有不变的结论：恰有 j 个部分非 0，则恰有 $pd(n-k,j)$ 个解，这里 j 只取 k 或 $k-1$. 因此直接得到递推：

$$pd(n,k) = pd(n-k,k-1) + pd(n-k,k)$$

同样像组合数一样列出表格，每个格里的数，等于该格前一列上数，所在列数个格子中的数，加上该格向上方数，所在列数个格子中的数.

k	0	1	2	3	4	5	6	7	8
$pd(0,k)$	1	0	0	0	0	0	0	0	0
$pd(1,k)$	0	1	0	0	0	0	0	0	0
$pd(2,k)$	0	1	0	0	0	0	0	0	0
$pd(3,k)$	0	1	1	0	0	0	0	0	0
$pd(4,k)$	0	1	1	0	0	0	0	0	0
$pd(5,k)$	0	1	2	0	0	0	0	0	0
$pd(6,k)$	0	1	2	1	0	0	0	0	0
$pd(7,k)$	0	1	3	1	0	0	0	0	0
$pd(8,k)$	0	1	3	2	0	0	0	0	0

例题

计算互异分拆数

计算互异分拆数 pd_n. 多组输入，其中 n 上界为 50000，对 1000007 取模.

观察表格与递推式，按列更新对于存储更有利. 代码中将后一位缩减了空间，仅保留相邻两项.

```cpp
#include <cstdio>
#include <cstring>

int pd[50005][2]; /*将自然数n分拆为k个部分的互异方法数*/

int main() {
  int n;
  while (~scanf("%d", &n)) {
    memset(pd, 0, sizeof(pd));
    pd[0][0] = 1;
    int ans = 0;
    int j;
    for (j = 1; j < 350; ++j) {
      int i;
      for (i = 0; i < 350; ++i) {
        pd[i][j & 1] = 0; /*pd[i][j]只与pd[][j]和pd[][j-1]有关*/
```

```
      }
      for (i = 0; i <= n; ++i) {
        if (i - j >= 0) /*pd[i-j][j]所有部分大于1*/
        {
          pd[i][j & 1] = (pd[i - j][j & 1] + pd[i - j][(j - 1) & 1]) %
                         1000007; /*pd[i-j][j-1]至少有一个部分为1. */
        }
      }
      ans = (ans + pd[n][j & 1]) % 1000007;
    }
    printf("%d\n", ans);
  }
}
```

奇分拆数

奇分拆数：po_n. 自然数 n 的各部分都是奇数的分拆方法数. (Odd)

有一个显然的等式:

$$\prod_{i=1}^{\infty}(1+x^i)=\frac{\prod_{i=1}^{\infty}(1-x^{2i})}{\prod_{i=1}^{\infty}(1-x^i)}=\prod_{i=1}^{\infty}\frac{1}{1-x^{2i-1}}$$

最左边是互异分拆数的生成函数，最右边是奇分拆数的生成函数. 两者对应系数相同，因此，奇分拆数和互异分拆数相同:

$$po_n=pd_n$$

但显然 k 部奇分拆数和互异 k 部分拆数不是一个概念，这里就不列出了.

再引入两个概念:

互异偶分拆数：pde_n. 自然数 n 的部分数为偶数的互异分拆方法数. (Even)

互异奇分拆数：pdo_n. 自然数 n 的部分数为奇数的互异分拆方法数. (Odd)

因此有:

$$pd_n=pde_n+pdo_n$$

同样也有相应的 k 部概念. 由于过于复杂，不再列出.

五边形数定理

单独观察分拆数的生成函数的分母部分:

$$\prod_{i=1}^{\infty}(1-x^i)$$

将这部分展开，可以想到互异分拆，与互异分拆拆出的部分数奇偶性有关.

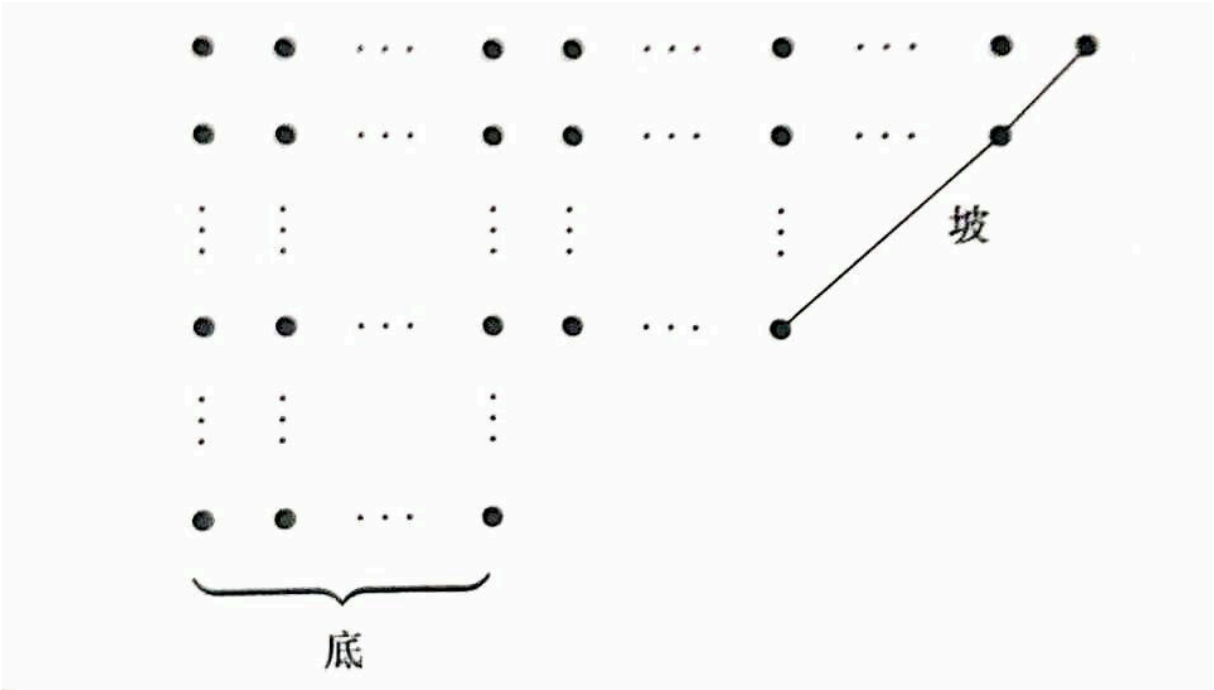
具体地，互异偶部分拆在展开式中被正向计数，互异奇部分拆在展开式中被负向计数. 因此展开式中各项系数为两方法数之差. 即:

$$\sum_{i=0}^{\infty}(pde_n-pdo_n)x^n=\prod_{i=1}^{\infty}(1-x^i)$$

接下来说明，多数情况下，上述两方法数相等，在展开式中系数为 0；仅在少数位置，两方法数相差 1 或 -1.

这里可以借助构造对应的办法.

画出每个互异分拆的 Ferrers 图．最后一行称为这个图的底，底上点的个数记为 b（Bottom）；连接最上面一行的最后一个点与图中某点的最长 45 度角线段，称为这个图的坡，坡上点的个数记为 s（Slide）．



要想在互异偶部分拆与互异奇部分拆之间构造对应，就要定义变换，在保证互异条件不变的前提下，使得行数改变 1：

变换 A：当 b 小于等于 s 的时候，就将底移到右边，成为一个新坡．

变换 B：当 b 大于 s 的时候，就将坡移到下边，成为一个新底．

这两个变换对于大多数 n 的任意互异分拆，恰有一个变换可以进行，就在互异偶部分拆与互异奇部分拆之间构造了一个一一对应．已经构造了一一对应的两部分分拆个数相等，因此这时展开式中第 n 项系数为 0．

但是对于某些 n，其存在恰一个互异分拆无法进行上述变换．

- 情况一：$b=s$ 且底与坡有一个公共点时，变换 A 不能进行．此时

$$n=s+(s+1)+...+(s+s-1)=\frac{s(3s-1)}{2}$$

展开式的第 n 项与分拆部分数的奇偶性有关，为 $(-1)^s x^n$．

- 情况二：$b=s+1$ 且底与坡有一个公共点时，变换 B 不能进行．此时

$$n=(s+1)+(s+2)+...+(s+s)=\frac{s(3s+1)}{2}$$

展开式的第 n 项为 $(-1)^s x^n$．

用 $-s$ 替换上式的 s，得到 $n=\frac{s(3s-1)}{2}$，其中 s 为负整数，展开式的第 n 项仍为 $(-1)^s x^n$..

由于两种情况不会在同一个 n 同时出现，我们可以把两个条件合起来，得到 n 需要满足的条件是

$$\exists k\in\mathbb{Z}, n=\frac{k(3k-1)}{2}$$

至此，我们就证明了：

$$(1-x)(1-x^2)(1-x^3)...=\sum_{k=-\infty}^{+\infty}(-1)^k x^{\frac{k(3k-1)}{2}}=...+x^{26}-x^{15}+x^7-x^2+1-x+x^5-x^{12}+x^{22}-...$$

回忆一下：这个式子是分拆数的生成函数的倒数，因此其与分拆数的生成函数相乘的结果是 1．整理并对比两边各项系数，就得到分拆数数列的递推式．

$$(1+p_1x+p_2x^2+p_3x^3+...)(1-x-x^2+x^5+x^7-x^{12}-x^{15}+x^{22}+x^{26}-...)=1$$

$$p_n=p_{n-1}+p_{n-2}-p_{n-5}-p_{n-7}+...$$

这个递推式有无限项，但是如果规定负数的分拆数是 0（0 的分拆数已经定义为 1），那么就简化为了有限项．

例题

计算分拆数

计算分拆数 p_n．多组输入，其中 n 上界为 50000，对 1000007 取模．

采用五边形数定理的方法．有代码：

```cpp
#include <cstdio>

long long a[100010];
long long p[50005];

int main() {
  p[0] = 1;
  p[1] = 1;
  p[2] = 2;
  int i;
  for (i = 1; i < 50005;
       i++) /*递推式系数1,2,5,7,12,15,22,26...i*(3*i-1)/2,i*(3*i+1)/2*/
  {
    a[2 * i] = i * (i * 3 - 1) / 2; /*五边形数为1,5,12,22...i*(3*i-1)/2*/
    a[2 * i + 1] = i * (i * 3 + 1) / 2;
  }
  for (
      i = 3; i < 50005;
      i++) /*p[n]=p[n-1]+p[n-2]-p[n-5]-p[n-7]+p[12]+p[15]-...+p[n-i*[3i-1]/2]+p[n-i*[3i+1]/2]*/
  {
    p[i] = 0;
    int j;
    for (j = 2; a[j] <= i; j++) /*有可能为负数,式中加1000007*/
    {
      if (j & 2) {
        p[i] = (p[i] + p[i - a[j]] + 1000007) % 1000007;
      } else {
        p[i] = (p[i] - p[i - a[j]] + 1000007) % 1000007;
      }
    }
  }
  int n;
  while (~scanf("%d", &n)) {
    printf("%lld\n", p[n]);
  }
}
```

9.13.13 范德蒙德卷积

引入

范德蒙德卷积是一种合并组合数的式子，主要应用于组合数学的公式推导.

范德蒙德卷积公式

$$\sum_{i=0}^{k}\binom{n}{i}\binom{m}{k-i}=\binom{n+m}{k}$$

证明

考虑用二项式定理证明:

$$\begin{aligned}\sum_{k=0}^{n+m}\binom{n+m}{k}x^k&=(x+1)^{n+m}\\&=(x+1)^n(x+1)^m\\&=\sum_{r=0}^{n}\binom{n}{r}x^r\sum_{s=0}^{m}\binom{m}{s}x^s\\&=\sum_{k=0}^{n+m}\sum_{r=0}^{k}\binom{n}{r}\binom{m}{k-r}x^k\end{aligned}$$

即有:

$$\binom{n+m}{k}=\sum_{r=0}^{k}\binom{n}{r}\binom{m}{k-r}$$

若考虑其组合意义证明:

在一个大小为 $n+m$ 的集合中取出 k 个数，可以等于把大小为 $n+m$ 的集合拆成两个集合，大小分别为 n 与 m，然后从 n 中取出 i 个数，从 m 中取出 $k-i$ 个数的方案数. 由于我们有了对于 i 的枚举，于是只需要考虑一种拆法，因为不同的拆法之间是等价的.

推论

推论 1 及证明

$$\sum_{i=-r}^{s}\binom{n}{r+i}\binom{m}{s-i}=\binom{n+m}{r+s}$$

证明与原公式证明相似.

推论 2 及证明

$$\sum_{i=1}^{n}\binom{n}{i}\binom{n}{i-1}=\binom{2n}{n-1}$$

根据基础的组合数学知识推导，有:

$$\sum_{i=1}^{n}\binom{n}{i}\binom{n}{i-1}=\sum_{i=0}^{n-1}\binom{n}{i+1}\binom{n}{i}=\sum_{i=0}^{n-1}\binom{n}{n-1-i}\binom{n}{i}=\binom{2n}{n-1}$$

推论 3 及证明

$$\sum_{i=0}^{n}\binom{n}{i}^2=\binom{2n}{n}$$

根据基础的组合数学知识推导，有:

$$\sum_{i=0}^{n}\binom{n}{i}^2=\sum_{i=0}^{n}\binom{n}{i}\binom{n}{n-i}=\binom{2n}{n}$$

推论 4 及证明

$$\sum_{i=0}^{m}\binom{n}{i}\binom{m}{i}=\binom{n+m}{m}$$

根据基础的组合数学知识推导，有：

$$\sum_{i=0}^{m}\binom{n}{i}\binom{m}{i}=\sum_{i=0}^{m}\binom{n}{i}\binom{m}{m-i}=\binom{n+m}{m}$$

其中 $\binom{n+m}{m}$ 是我们较为熟悉的网格图路径计数的方案数．所以我们可以考虑其组合意义的证明．

在一张网格图中，从 $(0,0)$ 走到 (n,m) 共走 $n+m$ 步．规定 $(0,0)$ 位于网格图左上角，其中向下走了 n 步，向右走了 m 步，方案数为 $\binom{n+m}{m}$．

换个视角，我们将 $n+m$ 步拆成两部分走，先走 n 步，再走 m 步，那么 n 步中若有 i 步向右，则 m 步中就有 $m-i$ 步向右，故得证．

习题

- CF785D Anton and School - 2[1]
- 洛谷 P2791 幼儿园篮球题[2]

参考资料与注释

1. Vandermonde's Convolution Formula[3]

外部参考资料

[1] CF785D Anton and School - 2 [2] 洛谷 P2791 幼儿园篮球题 [3] Vandermonde's Convolution Formula

9.13.14 Pólya 计数

Authors: Wajov, Early0v0, Enter-tainer, Great-designer, iamtwz, Ir1d, MegaOwIer, mgt, StudyingFather, Tiphereth-A, warzone-oier, Xeonacid, c-forrest

前置知识：置换和排列→9.8@p1138

引入

Pólya 计数原理通常用来解决一些涉及「本质不同」的计数问题．

本文可能涉及群论的相关内容

本文可能涉及到群论的相关内容．本文会对涉及到的群论概念做简单的解释，以便于不熟悉相关内容的读者理解和应用 Pólya 计数原理．关于群论的内容，严格的表述和讨论请参考 抽象代数基本概念→9.16.1@p1805、群论→9.16.2@p1812 等章节．

「空间对称群」、「对称群」与「置换群」

本文中将不可避免地同时使用这三类群的名字．尽管可能很容易造成混淆，但它们确实代指不同的概念．给定几何结构，它上面的对称操作指的是能够使它与它自身重合的几何变换，而空间对称群（symmetry group）就是这些对称操作的集合．对称群（symmetric group）是给定集合上的全体置换的

集合．置换群（permutation group）则是对称群的子群，即一些（未必是全体）置换构成的群．后文会解释如何将给定几何结构的空间对称群表示成置换群的形式，并用于计数问题．

Burnside 引理

相关阅读：Burnside 引理→9.16.2@p1812

Pólya 计数原理是 Burnside 引理的应用和推广．在介绍 Pólya 计数原理之前，需要先简单地回顾 Burnside 引理的内容．

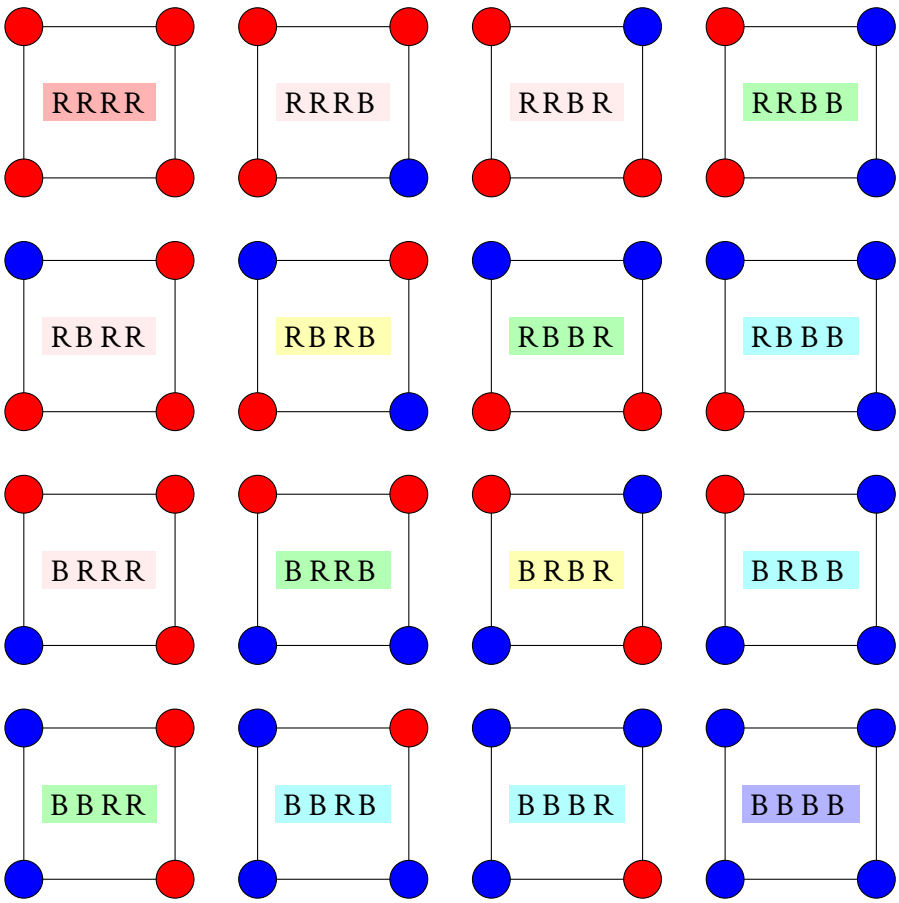
为了总结出一般的规律，首先考虑简单的例子．

项链染色

现在有一串共四个珠子的项链，每个珠子可以是红色或者蓝色，计算共有几种本质不同的项链．（如果两种染色的结果可以通过旋转项链重合，就认为是相同的．）

解答和分析

这个问题足够简单，可以通过枚举的方式加以解答．珠子共计 4 个，每个珠子可以染 2 种颜色，所以，项链所有可能的染色方案共计 $2^4 = 16$ 种．将可以通过旋转相互得到的分到一组，共计 6 组，如下图所示．（其中，单个染色方案的编码表示了自左下角的珠子开始顺时针染的颜色，B 表示蓝色，R 表示红色；分到同一组的染色方案的编码有着相同的背景颜色．）



从这个例子中可以看到，要计算本质不同的染色的种类数，关键其实是知道每种本质相同的染色对应几种不同的染色方案．也就是说，要搞清楚上图中每个分组的大小．

能够分到同一个组中的染色方案, 就是指它们之间能够通过旋转操作互相转化的染色方案. 总共有 4 种旋转的方式，即

$$G = \{r_0, r_1, r_2, r_3\},$$

分别表示旋转 $0, 1, 2, 3$ 次．旋转 0 次就是原地不动.

首先看染色方案 $RRBB$ 所在的分组．对它施加这四种操作，将分别得到

$$RRBB, RBBR, BBRR, BRRB.$$

四种染色方案互不相同，因此这个组就有 4 个元素.

再看染色方案 $BRBR$ 所在的分组．对它同样施加这四种操作，将分别得到

$$BRBR, RBRB, BRBR, RBRB.$$

此时，旋转两次的结果和不旋转的结果是一致的，旋转三次和旋转一次的结果是一致的．所以，这个组就有 2 个元素.

最后看染色方案 $BBBB$ 和 $RRRR$ 所在的分组. 对它们施加四种操作得到的结果都是它们自身. 因而，每个组就只有 1 个元素.

如果用 x 表示染色方案，Gx 表示对染色方案 x 操作后能够得到的颜色编码的集合，那么从上面的例子可以总结出一个规律，那就是 G 的操作对于 x 的影响存在某种「周期性」.

设 $|\ G\ |$ 表示操作的总个数，这种影响的「周期性」意味着，如果有 m 个不同的 G 中的操作将染色方案 x 变换到它自身，那么 x 在这些操作下的结果就会重复出现 m 次．因而，染色方案 x 在这些操作下共计有 $|\ G\ |\ /m$ 种不同的结果，这也就是 x 所在分组的大小.

这个例子中，因为只有旋转零次 r_0 才能够将 $RRBB$ 变换到它自身，所以，它所在分组的大小等于 $4/1 = 4$；而旋转零次 r_0 和两次 r_2 都能将 $BRBR$ 变换到它自身，所以，它所在分组的大小等于 $4/2 = 2$；无论旋转几次都能将 $BBBB$ 变换到它自身，所以，它所在分组的大小就是 $4/4 = 1$.

在下面的叙述中，用 G_x 表示能够将 x 变换到它自身的操作的集合，所以，$|\ G_x|$ 就是上面的 m. 此时，x 所在分组大小是 $|\ G\ |\ /|\ G_x|$. 要计算染色方案的分组的数目，只需要穷举所有可能的染色方案 $x \in X$，对所在分组大小为 $|\ Gx\ |$ 的染色方案 x 赋以权重 $1/|\ Gx\ |$，就能够将分组的数目表达为

$$|\ X/G\ | = \sum_{x \in X} \frac{1}{|\ Gx\ |} = \sum_{x \in X} \frac{|\ G_x|}{|\ G\ |}.$$

现在这个式子的形式并不便于应用．记 gx 是对染色方案 $x \in X$ 应用操作 $g \in G$ 的结果，那么上面描述的集合 G_x 就是 $\{g \in G : gx = x\}$，所以交换求和顺序就有

$$\begin{aligned}
\sum_{x \in X} |\ G_x| &= \sum_{x \in X} |\ \{g \in G : gx = x\}\ | \\
&= \sum_{x \in X} \sum_{g \in G} [gx = x] \\
&= \sum_{g \in G} \sum_{x \in X} [gx = x] \\
&= \sum_{g \in G} |\ \{x \in X : gx = x\}\ | \\
&= \sum_{g \in G} |\ X^g|\ .
\end{aligned}$$

其中 $[\cdot]$ 为 Iverson 括号．交换求和记号的结果中，$X^g = \{x \in X : gx = x\}$ 是指在操作 g 下保持不变的染色方案 x 的集合．简言之，它是操作 g 的不动点.

在这些讨论之后，现在可以将分组的个数写作

$$|X/G| = \frac{1}{|G|}\sum_{g\in G}|X^g|.$$

也就是说，分组的个数是各种旋转操作的不动点的平均个数.

作为这个结果的应用，再次计算项链染色的个数. 这些旋转操作的不动点可以列举如下.

操作	不动点
r_0	X
r_1	$\{BBBB, RRRR\}$
r_2	$\{BBBB, BRBR, RBRB, RRRR\}$
r_3	$\{BBBB, RRRR\}$

因而，分组的个数就等于

$$\frac{16+2+4+2}{4} = 6.$$

这样就得到了前面的结果.

从这个例子中，可以归纳出一般的结果，用于求解这类计数问题. 为了方便讨论，本文考虑的情景是染色问题，当然也可以应用到别的情景上去，后文会提供相应的例子.

染色问题是说，给定某个结构，在它的每个可以染色的对象（例如元素、顶点、边等）上染色，会得到不同的染色方案. 这个结构拥有某种对称性，使得看似不同的染色方案在经过一系列对称操作后能够互相转化. 这些能互相转化的染色方案就称为本质相同的. 问题是要求解本质不同的染色的数目.

根据例子中的分析，要求解这样的问题，首先要讨论给定的结构都有哪些对称操作. 这些对称操作的集合 G 称为给定结构的空间对称群. 实际应用中，大多时候无需了解群的定义，只需要能够不重不漏地讨论所有的空间对称操作就可以了. 本文后面分析了几个常见的空间对称群的结构，那里解释了群的定义.

所有染色方案的集合记作 X，其中的单个染色方案记作 x. 操作 $g \in G$ 作用在染色方案 $x \in X$ 的结果是 gx. 那么，能够通过某个操作作用在染色方案 x 上的所有结果就是 $Gx = \{gx : g \in G\}$，它称为群 G 作用下 x 的轨道. 同一轨道中的不同染色方案就是这类问题中所谓「本质相同」的. 故而，所有本质不同的染色的数目，就等价于不同轨道的数目.

例子中的分析可以推广到一般的情形.

Burnside 引理

给定群 G 在集合 X 上的作用，则所有不同的轨道的数目

$$|X/G| = \frac{1}{|G|}\sum_{g\in G}|X^g|.$$

这里，$X^g = \{x \in X : gx = x\}$ 是 $g \in G$ 的作用下的不动点集合.

它的证明几乎就是照搬上面例子中的分析. 但是，例子中用到了观察，即群 G 在单个元素 x 上的作用结果具有某种「周期性」，所以，这种周期重复的数目就等于能将 x 变换到它自身的操作的数目. 这个观察在一般的情形是正确的，但是因为群 G 的结构可能很复杂，它的「周期性」未必是例子中呈现的那么直接. 严格地表述这个观察，需要用到群论中的 轨道稳定子定理（orbit-stabilizer theorem）→9.16.2@p1812.

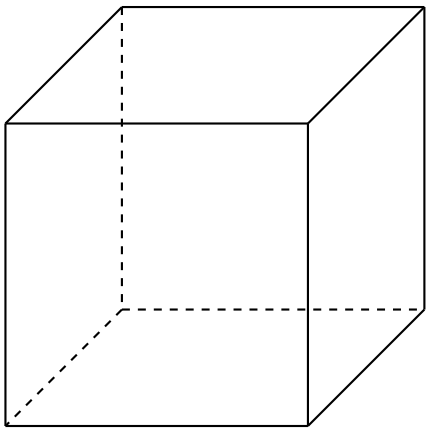
在应用的时候，只要能够列举出所有的对称操作，并且给出每个对称操作对应的不动点数目就可以解决对应的计数问题. 下面是一个稍微复杂的应用.

立方体染色

用三种颜色给一个立方体染色，求本质不同的方案数（经过空间旋转后相同的两种方案视为同一种）.

解答

因为立方体有 6 个面，每个面有 3 种染色方法，所以，总共有 3^6 种染色方案，即 $|X|=3^6$. 记立方体的空间对称群为 G.



接下来我们需要对 G 中的所有操作进行分析，它们可以分为以下几类（方便起见，将立方体的六个面分别称为前、后、上、下、左、右）：

- 不动：即恒等变换，因为所有染色方案经过恒等变换都不变，因此它对应的 $|X^g|=3^6$；
- 以两个相对面的中心连线为轴的 90° 旋转：相对面有 3 种选择，旋转的方向有 2 种选择，因此这类共有 6 个置换. 假设选择了前、后两个面中心的连线为轴，则必须要满足上、下、左、右四个面的颜色一样，才能使旋转后不变. 此时，有 3 个可以独立染色的区域，因此它对应的 $|X^g|=3^3$；
- 以两个相对面的中心连线为轴的 180° 旋转：相对面有 3 种选择，旋转方向的选择没有影响，因此这类共有 3 个置换. 假设选择了前、后两个面中心的连线为轴，则必须要满足上、下两个面的颜色一样，左、右两个面的颜色一样，才能使旋转后不变. 此时，有 4 个可以独立染色的区域，因此它对应的 $|X^g|=3^4$；
- 以两条相对棱的中点连线为轴的 180° 旋转：相对棱有 6 种选择，旋转方向对置换依然没有影响，因此这类共有 6 个置换. 假设选择了前、上两个面的边界和下、后两个面的边界作为相对棱，则必须要满足前、上两个面的颜色一样，下、后两个面的颜色一样，左、右两个面的颜色一样，才能使旋转后不变. 此时，有 3 个可以独立染色的区域，因此它对应的 $|X^g|=3^3$；
- 以两个相对顶点的连线为轴的 120° 旋转：相对顶点有 4 种选择，旋转的方向有 2 种选择，因此这类共有 8 个置换. 假设选择了前面的右上角和后面的左下角作为相对顶点，则必须满足前、上、右三个面的颜色一样，后、下、左三个面的颜色一样，才能使旋转后不变. 此时，有 2 个可以独立染色的区域，因此它对应的 $|X^g|=3^2$.

因此，所有本质不同的染色方案数为

$$\frac{1\times 3^6+6\times 3^3+3\times 3^4+6\times 3^3+8\times 3^2}{1+6+3+6+8}=57.$$

Pólya 计数原理

在 Burnside 引理的叙述中，并没有用到集合 X 是某结构上的全部染色方案这一性质．其实，Burnside 引理的应用范围并不局限于染色计数问题．对于染色计数问题，Pólya 计数原理则提供了更为具体的计算方法．它可以看作是一般性的 Burnside 引理在染色计数问题上的应用．

相较于 Burnside 引理，Pólya 计数原理的改进就是提供了不动点集合大小 $|X^g|$ 在染色计数问题中的具体计算方法．

这一点从上面的立方体染色的例子可以直观地看出来．对于立方体的各种对称操作，它的不动点集合的大小都是 $m^{c(g)}$ 的形式，这里 m 是颜色的数目，$c(g)$ 是在操作 g 下可以独立染色的区域数目．这个观察在一般的情形下也是成立的，不过需要进一步明晰如何对给定的 g 计算 $c(g)$ 的取值．

给某个结构选择一种染色方案，用数学语言表示，就是选择一个从这个结构的可以染色的对象（比如项链中的珠子、立方体的面等）的集合 X 到颜色集合 C 的映射 $f: X \to C$．因此，染色方案的集合就是 C^X．该结构的空间对称群 G 作用在结构上，自然也连带着作用在集合 X 上．这种对称操作，总对应着集合 X 上的双射，即 **置换**（permutation）．[2]

现在分析不动点集合 $(C^X)^g$ 的结构．给定 g，看作 X 上的置换，比照例子中的分析可以知道，如果 X 中的位置 x 在有限次重复操作 g 之后可以移动到位置 y，那么，作为不动点 $f \in (C^X)^g$，必然需要满足 $f(x) = f(y)$．用上一节轨道的语言来说，因为位置 x 和位置 y 处于操作 g 作用[3]的同一个轨道上，所以它们需要染相同的颜色．用置换的语言来说，在置换 g 的 轮换分解→9.8@p1138 中，位置 x 和位置 y 处于同一轮换，故而需要染相同的颜色．轮换分解中不同的轮换的染色不必相同，可以独立染色，所以此时可以独立染色的区域数目就是 $c(g)$，即 g 的轮换分解中的轮换数目．

由此，操作 g 的不动点的数目就是 $|C|^{c(g)}$．将这个结论代入 Burnside 引理，就能得到无权重版本的 **Pólya 计数原理**（Pólya enumeration theorem）．

Pólya 计数原理（无权重版本）

给定群 G 在集合 X 上的作用和颜色集合 C，则不同的染色方案的数目

$$|C^X/G| = \frac{1}{|G|}\sum_{g\in G} m^{c(g)},$$

这里，m 是颜色数目，$c(g)$ 是元素 $g \in G$ 的置换表示的轮换分解中的轮换数目．

关于群 G 的含义

这里略微有些滥用记号．如果群 G 作用在 X 上，那么染色方案集合 C^X 上的群作用是需要重新定义的，这里没有加以区分．

作为 Pólya 计数原理的简单应用，下面重新用 Pólya 计数原理计算前文的例子．

项链染色问题另解

将四个珠子标号 $1 \sim 4$，则例子中的群 G 中的元素分别有置换表示如下：（均写作轮换分解的形式）

- 旋转零次 $r_0 = (1)$，共计 4 个轮换（注意省略的 1-轮换）；
- 旋转一次 $r_1 = (1234)$，共计 1 个轮换；
- 旋转两次 $r_2 = (13)(24)$，共计 2 个轮换；
- 旋转三次 $r_3 = (1432)$，共计 1 个轮换．

[2]因此，空间对称群 G 可以表示是集合 X 上的置换群，即对称群 S_X 的子群．
[3]严格来说，是子群 $\langle g\rangle \leq G$ 的作用．

因此，本质不同染色的数目是

$$\frac{2^4+2^1+2^2+2^1}{4}=6.$$

立方体染色问题另解

由于前文的分析实质上已经给出了各类置换的轮换表示，只是没有用数字符号显式地书写出来，这里不再重复前文的分析．仅仅考虑以相对棱的中点连线为轴的 180° 旋转的情形，加以示例．将前、后、上、下、左、右六个面依次编号为 $1\sim6$，此时对应的置换是 $(13)(24)(56)$，因此 $c(g)=3$．其它类型的置换也可以类似分析，最后的计数的表达式也和上文完全一致．

带权重形式的推广

无权重版本的 Pólya 计数原理只能够给出所有的本质不同的染色问题的计数，但是在处理更为精细的问题时就无能为力了．比如说，如果在上述染色问题中，给定每种可以使用的颜色的数目，就不能套用上面的 Pólya 计数公式．在实际求解这类问题时，需要再次使用 Burnside 引理加以推导；而将这些结果总结为生成函数的形式，就是带权重版本的 Pólya 计数原理．

项链染色（带限制）

现在有一串共四个珠子的项链，每个珠子可以是红色或者蓝色，恰有两个红色珠子、两个蓝色珠子可以使用，计算共有几种本质不同的项链．（如果两种染色的结果可以通过旋转项链重合，就认为是相同的．）

解答和分析

考虑使用 Burnside 引理．红色、蓝色珠子各两个，共计有 $\binom{4}{2}=6$ 种染色方案．空间对称群 $G=\{r_0,r_1,r_2,r_3\}$ 分别对应旋转 $0\sim3$ 次，则它们对应的不动点集合分析如下：

- 旋转零次 $r_0=(1)$，全部 6 个染色方案都是不动点；
- 旋转一次 $r_1=(1234)$，不动点要求所有珠子染同样的颜色，没有不动点；
- 旋转两次 $r_2=(13)(24)$，有两个可独立染色的区域，大小都是 2，它们要分别染成红色和蓝色，则不动点集合的大小为 2；
- 旋转三次 $r_3=(1432)$，与旋转一次的情形相同，没有不动点．

所以，根据 Burnside 引理，本质不同的染色数目为

$$\frac{6+0+2+0}{4}=2.$$

从这个例子中可以总结出如下计算方法．对于限制不同颜色个数的问题，同样是要把空间对称群中各个置换的轮换分别染色，但是需要让每种颜色用到的次数恰好等于给定的数目．这样的组合问题通常没有显式解，除了可以通过 排列组合方法→9.13.1@p1612 计算的特殊情形外，往往需要看作 背包问题→7.4.1@p724 进行求解．

通过生成函数可以给出这类计数问题的答案．给定置换 g，如果它的 型→9.8@p1138 是 $1^{\alpha_1}2^{\alpha_2}\cdots n^{\alpha_n}$，即它有 α_k 个长度为 k 的轮换，且对于每个轮换可以染成 m 种颜色中的一种，那么生成函数

$$\prod_{k=1}^{n}\left(\sum_{i=1}^{m}x_i^k\right)^{\alpha_k}$$

中单项式 $x_1^{\beta_1}x_2^{\beta_2}\cdots x_m^{\beta_m}$ 的系数就是颜色 i 用了 β_i 次（$1\le i\le m$）的染色方案的计数．这里圆括号中的表达式 $\sum_{i=1}^m x_i^k$ 的组合意义是，对于长度为 k 的轮换，用到 k 次颜色 i 的染色方法的计数是 1，对于其它情形，计数是 0；这正描述了同一轮换中各位置染色一致的要求．

给定置换 g 下染色计数的生成函数，对各个单项式应用 Burnside 引理，就得到各种颜色组合下的本质不同的计数．因为对每个单项式，其系数所计数的染色方案的集合在 G 的作用下封闭，而取平均的运算对系数是线性的，所以本质不同染色方案的计数的生成函数是

$$\frac{1}{\mid G\mid}\sum_{g\in G}\prod_{k=1}^{n}\left(\sum_{i=1}^{m}x_i^k\right)^{\alpha_k}.$$

展开这个式子，每个单项式的系数就给出了给定颜色组合下的本质不同染色的计数．

在上述过程中，对每个轮换进行染色的生成函数 $\sum_{i=1}^m x_i^k$ 并无特殊之处，可以替换成其它的生成函数．因而，有如下的一般版本的 Pólya 计数原理．

置换群的轮换指标

给定置换群 G，则群 G 的 **轮换指标**（cycle index），常记作 Z_G 或 $Z(G)$，定义为

$$Z_G(t_1,t_2,\cdots,t_n)=\frac{1}{\mid G\mid}\sum_{g\in G}t_1^{c_1(g)}t_2^{c_2(g)}\cdots t_n^{c_n(g)},$$

其中，$c_k(g)$ 是置换 g 的轮换分解中长度为 k 的轮换的个数，即 $1^{c_1(g)}2^{c_2(g)}\cdots n^{c_n(g)}$ 是置换 g 的型．

Pólya 计数原理（带权重版本）

给定群 G 在集合 X 上的作用，每个点的染色方案的计数由生成函数 $f(x_1,x_2,\cdots,x_m)$ 给出，那么集合 X 的本质不同染色方案的计数的生成函数是

$$Z_G(f(x_1^1,x_2^1,\cdots,x_m^1),f(x_1^2,x_2^2,\cdots,x_m^2),\cdots,f(x_1^n,x_2^n,\cdots,x_m^n)),$$

这里，$Z_G(t_1,t_2,\cdots,t_n)$ 是群 G 的轮换指标．

这里，如果单个位置的染色的生成函数是 $f(x_1,x_2,\cdots,x_m)$，那么长度为 k 的轮换的染色的生成函数就是 $f(x_1^k,x_2^k,\cdots,x_m^k)$．这反映了如果某一染色方案是给定置换的不动点，那么同一轮换中的所有位置必须染相同的颜色．如果将生成函数在 $x_i=1$ 处取值，就得到上文的无权重版本的 Pólya 计数原理．

定理的叙述用到了置换群的轮换指标的概念．它和具体的染色问题无关．它描述了置换群的结构．

带限制的项链染色问题另解

旋转对称群的轮换指标是 $\frac{1}{4}(t_1^4+t_2^2+2t_4)$，单点染色的生成函数是 $r+b$，故而全体染色方案的生成函数是

$$\begin{aligned}F(r,b)&=\frac{1}{4}((r+b)^4+(r^2+b^2)^2+2(r^4+b^4))\\&=r^4+r^3b+2r^2b^2+rb^3+b^4.\end{aligned}$$

所求计数就是 r^2b^2 的系数，即共 2 种本质不同染色．顺便，这个式子也给出了其他限制下的计数．

应用

带权重版本的 Pólya 计数原理在组合计数问题中起到重要的作用．这里简单讨论它的应用，而更一般的讨论可以参考 组合问题的形式化方法→9.12.12@p1580．

钻石项链

现在有一串共四个相同珠子的项链，每个珠子上可以镶若干颗钻石. 如果有四枚钻石，总共有多少本质不同的镶钻方式.（如果两种镶钻的结果可以通过旋转项链重合，就认为是相同的.）

解答和分析

项链的空间对称群仍与前文所述相同. 不考虑钻石总数的限制，则单个位置的镶钻方案的生成函数是

$$f(x)=1+x+x^2+\cdots=\sum_{i=0}^{\infty}x^i=\frac{1}{1-x}.$$

应用带权重版本的 Pólya 计数原理可知，所有镶钻方案的生成函数为

$$\begin{aligned}F(x)&=\frac{1}{4}(f(x)^4+f(x^2)^2+2f(x^4))\\&=1+x+3x^2+5x^3+10x^4+\cdots.\end{aligned}$$

故而，所求镶钻方案的数目就是 x^4 的系数，即共计 10 种方案. 作为验证，通过枚举可知，它们分别是

$$4000,3100,3010,3001,2200,2020,2110,2101,2011,1111.$$

这里，每组四个数字分别表示每个珠子上的镶钻数目.

这个例子说明，带权重版本的 Pólya 计数原理能够解决的问题远比染色计数问题要广泛. 它提供了一种将单点的计数扩展到整个结构上本质不同的计数的方法. 染色问题只是这类问题的特例.

常见空间对称群

Pólya 计数相关问题的难点之一在于分析置换群的结构. 这里，简单讨论常见的空间对称群的结构，并用它们的轮换指标加以描述. 应当注意，对于同一个结构的空间对称群，如果考虑的作用对象的集合不同，相应的 群作用→9.16.2@p1812 也就不同，因而它们的置换表示也就不同. 比如说，正方体的空间对称群对于它的顶点、棱、面的作用就分别对应着正方体的顶点置换群、棱置换群和面置换群，顶点、棱、面的个数互不相同，故而这些置换群以及对应的轮换指标当然也各不相同. 所以，在具体问题的求解中，不能忽视群作用的对象的指定.

空间对称群和置换群的关系

虽然两者概念上十分相似，但是它们绝不是同一个对象. 用群论的语言说，给定空间对称群 G 和它在集合 X 上的群作用，群作用的置换表示实则提供了一个从群 G 到对称群 S_X 的同态 φ，而且这个置换表示在组合计数的语境下往往是忠实的，即 $\ker\varphi=\{e\}$，故而同态 φ 实则是群 G 到群 S_X 内的一个嵌入. 文中的置换群则是这个嵌入的像，即 $\varphi(G)$，它与本身的空间对称群 G 同构. 因此，对于同样的结构上的空间对称群 G，如果群作用的选取不一致，就会同构于不同的置换群 $\varphi(G)$，进而具有不同的轮换指标（同构的置换群的轮换指标未必相同）.

给定一个结构，它的空间对称群是所有能够将它变换到它自身的操作的集合. 它必然满足如下条件：

- 对给定结构连续应用两个对称操作，可以视作应用另一个对称操作，即对称操作的集合对于复合是满足封闭性的；
- 对称操作的复合满足结合律；
- 存在恒等的对称操作，即给定结构保持不变本身也视作一个操作；
- 任何操作都存在它的逆操作，可以抵消给定操作的效果.

群→9.16.1@p1805 是对所有满足这些条件的概念的抽象．对于群的结构的讨论，就是 群论→9.16.2@p1812 的主要研究内容．这里的分析主要集中在空间对称群，对它的结构的讨论也主要应用几何观点．这里给出了常见的例子，读者应当从中获得分析这类问题的常见思路．

循环群

给定正 n 边形，它的全体旋转操作构成的空间对称群称为循环群（cyclic group），记作 C_n．将逆时针旋转 $(360/n)^\circ$ 的操作记作 r，则群 C_n 的元素可以写作

$$C_n = \{e, r, r^2, \cdots, r^{n-1}\}.$$

这里，r^k 指对操作 r 重复 k 次的结果，即逆时针旋转 $(360k/n)^\circ$，而 $e = r^0$ 指恒等变换．

无论是考虑循环群对正 n 边形的全体顶点还是全体边的集合的作用，它的置换表示都是一样的．以全体顶点的集合为例分析群作用的置换表示．它的轮换指标是

$$Z(C_n) = \frac{1}{n}\sum_{d\mid n}\varphi(d)t_d^{n/d}.$$

这里，$\varphi(\cdot)$ 是数论中的 欧拉函数→9.11.5@p1196．

只计旋转操作，长度为 n 的项链的空间对称群就是 C_n．

分析

设顶点的集合按照逆时针顺序记为 $\{0, 1, \cdots, n-1\}$，则 $r^k(i) = (i+k) \bmod n$．顶点 i 所在的轮换中的顶点集合就是

$$\{(i + \ell k) \bmod n : \ell \in \mathbf{Z}\}.$$

显然，$i \equiv i + \ell k \pmod n$ 当且仅当

$$\frac{n}{\gcd(k,n)} \mid \ell.$$

这意味着，任何顶点 i 所在的轮换长度都是 $\dfrac{n}{\gcd(k,n)}$．因此，置换 r^k 有 $\gcd(k,n)$ 个等长的轮换．考虑在轮换指标的表达式中合并同类项，给定 $d \mid n$，则满足 $\gcd(k,n) = n/d$ 的 k 共计 $\varphi(d)$ 个，它们对应的单项式都是 $t_d^{n/d}$ 的形式，所以可以得到上面的轮换指标表达式．

二面体群

给定正 n 边形，它的全体旋转和关于对称轴翻转的操作也构成空间对称群，它称为二面体群（dihedral group），记作 D_{2n}．将逆时针旋转 $(360/n)^\circ$ 的操作记作 r，并将沿某个给定对称轴（比如中心与某个顶点的连线）翻转的操作记作 s，则群 D_{2n} 的操作可以写作

$$D_{2n} = \{e, r, \cdots, r^{n-1}, s, sr, \cdots, sr^{n-1}\}.$$

这里，r^k 依然是旋转操作，而 sr^k 虽然是先进行 k 次旋转再沿给定对称轴翻转，但是可以等价地看作沿着另一个对称轴翻转．因此，群 D_{2n} 中共计 1 个恒等变换、$(n-1)$ 个旋转操作和 n 个翻转操作．它对顶点集合和边集合的群作用也有着相同的轮换指标[4]，即

$$Z(D_{2n}) = \frac{1}{2}Z(C_n) + \begin{cases} \dfrac{1}{2}t_1 t_2^k, & n = 2k+1, \\ \dfrac{1}{4}\left(t_1^2 t_2^{k-1} + t_2^k\right), & n = 2k. \end{cases}$$

[4] 与循环群的情形不同，当 n 为偶数时，这两个群作用的置换表示并不相同：过相对顶点的对称轴上的翻转操作在顶点集合上的型是 $1^2 2^{k-1}$，在边集合上却是 2^k，过相对边中点的对称轴上的翻转操作则恰好相反．但是，这两类对称轴的数目相同，所以求和之后的轮换指标不变．

分析

群 D_{2n} 中的旋转操作 r^k 的集合（包括恒等变换）的分析和循环群 C_n 如出一辙，关键在于剩下的翻转操作的分析．此时需要对顶点个数 n 的奇偶性分类讨论．

当 $n=2k+1$ 时，所有的翻转操作的对称轴都是连结顶点和它对面的边的中点的，共计 n 条这样的对称轴．每个翻转操作后，对称轴上的顶点保持不动，而其它顶点成对地交换，因此有 1 个不动点（1-轮换）和 k 个 2-轮换．

当 $n=2k$ 时，有两种对称轴．其中，一半的对称轴是连接相对的顶点的；沿着这样的对称轴翻转，将保持对称轴上的两个顶点不动，而将其余的顶点成对地交换，因此有 2 个不动点（1-轮换）和 $(k-1)$ 个 2-轮换．另一半的对称轴是连接相对的边的中点的；沿着这样的对称轴翻转，将所有顶点都成对地交换，因此有 k 个 2-轮换．

根据这一分析，可以写出上面的轮换指标表达式．

对称群

给定 n 个元素，它上面的全体置换构成群，称为 n 次对称群（symmetric group），记作 S_n．它描述了这 n 个元素能拥有的全部对称性．它也是这些对称操作对元素集合的作用的置换表示．

根据 置换与排列→9.8@p1138 一文的分析，它的轮换指标是

$$Z(S_n)=\sum_{\alpha_1+2\alpha_2+\cdots+n\alpha_n=n}\frac{t_1^{\alpha_1}t_2^{\alpha_2}\cdots t_n^{\alpha_n}}{1^{\alpha_1}2^{\alpha_2}\cdots n^{\alpha_n}\alpha_{(1)!}\alpha_{(2)!}\cdots\alpha_{(n)!}}.$$

这里用到了型为 $1^{\alpha_1}2^{\alpha_2}\cdots n^{\alpha_n}$ 的置换的计数是

$$\frac{n!}{1^{\alpha_1}2^{\alpha_2}\cdots n^{\alpha_n}\alpha_{(1)!}\alpha_{(2)!}\cdots\alpha_{(n)!}}.$$

它满足递推关系

$$Z(S_n)=\frac{1}{n}\sum_{k=1}^{n}t_kZ(S_{n-k}),$$

而递推起点是 $Z(S_0)=1$．这一递推关系的组合意义是，要构造 n 元置换，可以首先选取点 n 所在轮换的长度 k，再对剩下的 $(n-k)$ 个元素的集合构造．

给定 n 个顶点的完全图，则它的空间对称群正是 S_n．它对全体顶点的集合的作用的轮换指标就由上文的 $Z(S_n)$ 给出．但是，它对全体边的集合的作用的置换表示并不相同．比如说，集合的大小就不相同，全体边的数目是 $n(n-1)/2$．对于边的情形，需要额外的分析．这里给出简单的例子，一般的情形可参考习题．

无向简单图计数

计算同构意义下有 4 个顶点的无向简单图的数目．

解答

这相当于在有 4 个顶点的完全图的边上用两种颜色染色，要求本质不同的染色数目．空间对称群是 S_4，现在分析它的边置换群 $S_4^{(2)}$ 的轮换指标．将四个顶点记作 a,b,c,d，并将六条边依次编号为 $1=ab$、$2=bc$、$3=cd$、$4=da$、$5=ac$、$6=bd$．

- 恒等变换（1 种）：边也保持不动，故对应单项式为 $1\cdot t_1^6$；
- 交换两顶点（6 种）：假设交换 a 和 b，则边 1 和边 3 保持不动，同时，边 2 和边 5 对换，边 4 和边 6 对换，故对应单项式为 $6t_1^2t_2^2$；

- 轮换三顶点（8 种）：假设轮换是 (abc)，则它们之间的连边 $1,2,5$ 也相应轮换，它们和第四点 d 的连边 $4,6,3$ 也相应轮换，故对应单项式为 $8t_3^2$；
- 交换两对顶点（3 种）：假设点 a 和点 b 对换，点 c 和点 d 对换，则边 1 和边 3 保持不动，同时，边 2 和边 4 对换，边 5 和边 6 对换，故对应单项式为 $3t_1^2t_2^2$；
- 轮换四顶点（6 种）：假设轮换是 $(abcd)$，则其中相邻顶点的连边 $1,2,3,4$ 也相应轮换，相对顶点的连边 $5,6$ 同时对换，故对应的单项式为 $6t_2t_4$.

所以，边置换群的轮换指标是

$$Z(S_4^{(2)})=\frac{1}{24}(t_1^6+9t_1^2t_2^2+8t_3^2+6t_2t_4).$$

根据 Pólya 计数原理，同构意义下有 4 个顶点的无向简单图的数目是

$$\frac{2^6+9\times 2^4+8\times 2^2+6\times 2^2}{24}=11.$$

多面体群

多面体群（polyhedral group）是正多面体的空间对称群．正多面体只有五种：正四面体、正方体、正八面体、正十二面体和正二十面体．如果保持点、棱、面之间的邻接关系，交换点和面，可以得到对偶的正多面体．其中，正四面体和它自身对偶，正方体和正八面体对偶，正十二面体和正二十面体对偶．利用对偶关系，可以简化它们的空间对称群的讨论．

只计三维空间中可以进行的旋转操作，它们的空间对称群只有三种．

- 四面体群（tetrahedral group），即正四面体的空间对称群：
  - 恒等变换；
  - 绕顶点和对面中心的连线旋转 120° 和 240°；
  - 绕对边的中点的连线旋转 180°.

  共计 $1+2\times 4+1\times 3=12$ 个对称操作.

  它对应的置换群的轮换指标如下.
  - 顶点置换群和面置换群：$\dfrac{1}{12}(t_1^4+8t_1t_3+3t_2^2)$；
  - 棱置换群：$\dfrac{1}{12}(t_1^6+8t_3^2+3t_1^2t_2^2)$.
- 八面体群（octahedral group），即正方体（和正八面体）的空间对称群：
  - 恒等变换；
  - 绕相对顶点的连线旋转 120° 和 240°；
  - 绕相对的棱的中点的连线旋转 180°；
  - 绕相对的面的中心的连线旋转 90°，180° 和 270°.

  共计 $1+2\times 4+1\times 6+3\times 3=24$ 个对称操作.

  它对应的正方体的置换群的轮换指标如下.
  - 顶点置换群：$\dfrac{1}{24}(t_1^8+8t_1^2t_3^2+9t_2^4+6t_4^2)$；
  - 棱置换群：$\dfrac{1}{24}(t_1^{12}+8t_3^4+6t_1^2t_2^5+6t_4^3+3t_2^6)$；
  - 面置换群：$\dfrac{1}{24}(t_1^6+8t_3^2+6t_2^3+6t_1^2t_4+3t_1^2t_2^2)$.

  正八面体的置换群类似，只是要将顶点和面的角色对换.
- 二十面体群（icosahedral group），即正十二面体（和正二十面体）的空间对称群：
  - 恒等变换；
  - 绕相对顶点的连线旋转 120° 和 240°；
  - 绕相对的棱的中点的连线旋转 180°；
  - 绕相对的面的中心的连线旋转 72°，144°，216° 和 288°.

  共计 $1+2\times 10+1\times 15+4\times 6=60$ 个对称操作.

它对应的正十二面体的置换群的轮换指标如下.

- 顶点置换群：$\frac{1}{60}(t_1^{20}+20t_1^2t_3^6+15t_2^{10}+24t_5^4)$；
- 棱置换群：$\frac{1}{60}(t_1^{30}+20t_3^{10}+15t_1^2t_2^{14}+24t_5^6)$；
- 面置换群：$\frac{1}{60}(t_1^{12}+20t_3^4+15t_2^6+24t_1^2t_5^2)$.

正二十面体的置换群类似，只是要将顶点和面的角色对换.

这里给出的都是对顶点、棱、面等单独的对象作用的置换群的轮换指标. 如果要对不同的对象同时染色，需要写出联合的轮换指标.

习题

染色问题

这些题目只需要分析置换群的结构，并应用 Pólya 计数原理.

- Luogu P4980【模板】Polya 定理[1]
- Luogu P2561 [AHOI2002] 黑白瓷砖[2]
- TRANSP - Transposing is Fun[3]
- TRANSP2 - Transposing is Even More Fun[4]
- Luogu P3307 [SDOI2013] 项链[5]

当可以使用的颜色组合受到限制时，需要通过背包 DP 或者组合方法求解对轮换染色的方法数目.

- Luogu P1446 [HNOI2008] Cards[6]
- UVA10601 Cubes[7]
- Luogu P4916 [MtOI2018] 魔力环[8]

图论计数

Pólya 计数原理可以用于 图论计数→9.13.15@p1693 问题，这类问题难点在于图的边置换群的枚举.

- SGU 282. Isomorphism[9]
- Luogu P4727 [HNOI2009] 图的同构计数[10]
- Luogu P4128 [SHOI2006] 有色图[11]

另一类可以应用 Pólya 计数原理的图论计数问题需要直接操纵生成函数.

- LOJ 6538 烷基计数 加强版 加强版[12]
- LOJ 6512「雅礼集训 2018」烷烃计数[13]
- Luogu P6597 烯烃计数[14]
- Luogu P5818 [JSOI2011] 同分异构体计数[15]

参考文献与注释

- Pólya enumeration theorem - Wikipedia[16]
- Notes on Pólya's Enumeration Theorem[17]
- Cycle index - Wikipedia[18]

[1] Luogu P4980【模板】Polya 定理

[2] Luogu P2561 [AHOI2002] 黑白瓷砖

[3] TRANSP - Transposing is Fun

[4] TRANSP2 - Transposing is Even More Fun

[5] Luogu P3307 [SDOI2013] 项链

[6] Luogu P1446 [HNOI2008] Cards

[7] UVA10601 Cubes

[8] Luogu P4916 [MtOI2018] 魔力环

[9] SGU 282. Isomorphism

[10] Luogu P4727 [HNOI2009] 图的同构计数

[11] Luogu P4128 [SHOI2006] 有色图

[12] LOJ 6538 烷基计数 加强版 加强版

[13] LOJ 6512「雅礼集训 2018」烷烃计数 

[14] Luogu P6597 烯烃计数 

[15] Luogu P5818 [JSOI2011] 同分异构体计数 

[16] Pólya enumeration theorem - Wikipedia 

[17] Notes on Pólya's Enumeration Theorem 

[18] Cycle index - Wikipedia 

9.13.15 图论计数

在组合数学中，图论计数（Graph Enumeration）是研究满足特定性质的图的计数问题的分支．生成函数→9.12.1@p1519、波利亚计数定理→9.13.14@p1680 与 符号化方法→9.12.12@p1580 和 OEIS[1] 是解决这类问题时最重要的数学工具．图论计数可分为有标号和无标号两大类问题，大多数情况下[5] 有标号版本的问题都比其对应的无标号问题更加简单，因此我们将先考察有标号问题的计数．

有标号树

即 Cayley 公式，参见 Prüfer 序列→11.25@p3033 一文，我们也可以使用 Kirchhoff 矩阵树定理→11.26@p3040 或 生成函数→9.12.1@p1519 和 拉格朗日定理[2] 得到这一结果．

习题

- Hihocoder 1047. Random Tree[3]

有标号连通图

例题「POJ 1737」Connected Graph

例题 「POJ 1737」Connected Graph[4]

题目大意：求有 n 个结点的有标号连通图的方案数（$n \leq 50$）．

这类问题最早出现于楼教主的男人八题系列中，我们设 g_n 为 n 个点有标号图的方案数，c_n 为待求序列．n 个点的图至多有 $\binom{n}{2}$ 条边，每条边根据其出现与否有两种状态，每种状态之间独立，因而有 $g_n = 2^{\binom{n}{2}}$．我们固定其中一个节点，枚举其所在连通块的大小，那么还需要从剩下的 $n-1$ 个节点中选择 $i-1$ 个节点组成一个连通块．连通块之外的节点可以任意连边，因而有如下递推关系：

$$\sum_{i=1}^{n} \binom{n-1}{i-1} c_i g_{n-i} = g_n$$

$$c_n = g_n - \sum_{i=1}^{n-1} \binom{n-1}{i-1} c_i g_{n-i}$$

移项得到 c_n 序列的 $O(n^2)$ 递推公式，可以通过此题．

例题「集训队作业 2013」城市规划

例题 「集训队作业 2013」城市规划[5]

题目大意：求有 n 个结点的有标号连通图的方案数（$n \leq 130000$）．

对于数据范围更大的序列问题，往往我们需要构造这些序列的生成函数，以使用高效的多项式算法．

方法一：分治 FFT

上述的递推式可以看作一种自卷积形式，因而可以使用分治 FFT 进行计算，复杂度 $O(n \log^2 n)$．

方法二：多项式求逆

我们将上述递推式中的组合数展开，并进行变形：

[5] 也许无标号二叉树是一个反例，在结构简单的情况下，对应的置换群是恒等群（Identity Group），此时有标号版本可以直接通过乘以 $n!$ 得到．

$$\sum_{i=1}^{n}\binom{n-1}{i-1}c_i g_{n-i}=g_n$$

$$\sum_{i=1}^{n}\frac{c_i}{(i-1)!}\frac{g_{n-i}}{(n-i)!}=\frac{g_n}{(n-1)!}$$

构造多项式：

$$C(x)=\sum_{n=1}\frac{c_n}{(n-1)!}x^n$$

$$G(x)=\sum_{n=0}\frac{g_n}{n!}x^n$$

$$H(x)=\sum_{n=1}\frac{g_n}{(n-1)!}x^n$$

代换进上式得到 $CG=H$，使用 多项式求逆→9.12.9@p1559 后再卷积解出 $C(x)$ 即可.

方法三：多项式 exp

另一种做法是使用 EGF 中多项式 exp 的组合意义→9.12.16@p1606，我们设有标号连通图和简单图序列的 EGF 分别为 $C(x)$ 和 $G(x)$，那么它们将有下列关系：

$$\exp(C(x))=G(x)$$

$$C(x)=\ln(G(x))$$

使用 多项式 ln→9.12.9@p1559 解出 $C(x)$ 即可.

有标号欧拉图、二分图

例题「SPOJ KPGRAPHS」Counting Graphs

例题 「SPOJ KPGRAPHS」Counting Graphs[6]

题目大意：求有 n 个结点的分别满足下列性质的有标号图的方案数（$n\le 1000$）.

- 连通图 A001187[7].
- 欧拉图 A033678[8].
- 二分图 A047864[9].

本题限制代码长度，因而无法直接使用多项式模板，但生成函数依然可以帮助我们进行分析.

连通图问题在之前的例题中已被解决，考虑欧拉图. 注意到上述对连通图计数的几种方法，均可以在满足任意性质的有标号连通图进行推广. 例如我们可以将连通图递推公式中的 g_n，从任意图替换成满足顶点度数均为偶数的图，此时得到的 c_n 即为欧拉图.

我们将 POJ 1737 的递推过程封装成连通化函数，

```
void ln(Int C[], Int G[]) {
  for (int i = 1; i <= n; ++i) {
    C[i] = G[i];
    for (int j = 1; j <= i - 1; ++j)
      C[i] -= binom[i - 1][j - 1] * C[j] * G[i - j];
  }
}
```

前两问即可轻松解决：

```
for (int i = 1; i <= n; ++i) G[i] = pow(2, binom[i][2]);
ln(C, G);
for (int i = 1; i <= n; ++i) G[i] = pow(2, binom[i - 1][2]);
ln(E, G);
```

注意到这里的连通化递推过程其实等价于对其 EGF 求多项式 ln，同理我们也可以写出逆连通化函数，它等价于对其 EGF 求多项式 exp.

```
void exp(Int G[], Int C[]) {
  for (int i = 1; i <= n; ++i) {
    G[i] = C[i];
    for (int j = 1; j <= i - 1; ++j)
      G[i] += binom[i - 1][j - 1] * C[j] * G[i - j];
  }
}
```

下面讨论有标号二分图计数，

我们设 b_n 表示 n 个结点的二分图方案数，g_n 表示 n 个结点对结点进行 2 染色，满足相同颜色的结点之间不存在边的图的方案数．枚举其中一种颜色节点的数量，有[6]：

$$g_n = \sum_{i=0}^{n} \binom{n}{i} 2^{i(n-i)}$$

接下来我们用两种不同的方法建立 g_n 与 b_n 之间的关系．

方法一：算两次

我们设 $c_{n,k}$ 表示有 k 个连通分量的二分图方案数，那么不难得到如下关系：

$$b_n = \sum_{i=1}^{n} c_{n,i}$$

$$g_n = \sum_{i=1}^{n} c_{n,i} 2^i$$

比较两种 g_n 的表达式，展开得：

$$\sum_{i=0}^{n} \binom{n}{i} 2^{i(n-i)} = \sum_{i=1}^{n} c_{n,i} 2^i$$

$$c_{n,i} = \sum_{i=0} n-1 \binom{n-1}{i-1} c_{n,1} c_{n-i,k-1}$$

不难得到 b_n 的递推关系，复杂度 $O(n^3)$，进一步使用容斥原理，可以优化到 $O(n^2)$ 通过本题．

方法二：连通化递推

方法二和方法三均使用连通二分图 $b1_n$ A001832[11] 来建立 g_n 与 b_n 之间的桥梁．

注意到对于每个连通二分图，我们恰好有两种不同的染色方法，对应到两组不同的连通 2 染色图，因而对 g_n 进行连通化，得到的序列恰好是 $b1_n$ 的两倍，而 b_n 则由 $b1_n$ 进行逆连通化得到．

因此：

```
for (int i = 1; i <= n; ++i) {
  G[i] = 0;
  for (int j = 0; j < i + 1; ++j) G[i] += binom[i][j] * pow(2, j * (i - j));
}
ln(B1, G);
for (int i = 1; i <= n; ++i) B1[i] /= 2;
exp(B, B1);
```

两种递推的过程复杂度均为 $O(n^2)$，可以通过本题．

方法三：多项式 exp

我们注意到也可以使用 EGF 理解上面的递推过程．

[6]粉兔的 blog[10] 告诉我们，这个序列也可以使用 Chirp Z-Transform→9.12.6@p1550 优化．

设 $G(x)$ 为 g_n 的 EGF，$B1(x)$ 为 $b1_n$ 的 EGF，$B(x)$ 为 b_n 的 EGF，应用做法二的方法，我们有：

$$G(x) = \exp(2B1(x))$$

$$\begin{aligned} B(x) &= \exp(B1(x)) \\ &= \exp(\frac{\ln G(x)}{2}) \\ &= \sqrt{G} \end{aligned}$$

我们可以对等式两边分别进行求导并比较两边系数，以得到易于编码的递推公式，通过此题．注意到做法二与做法三本质相同，且一般情况下做法三可以得到更优的时间复杂度．

$$B_n^2 = G$$

$$2B_nB_{(n)'} = G'$$

参考代码

```cpp
#include <iostream>
using namespace std;

using LL = long long;
constexpr int MOD = int(1e9) + 7;

// <<= '2. Number Theory .,//{
namespace NT {
void INC(int& a, int b) {
  a += b;
  if (a >= MOD) a -= MOD;
}

int sum(int a, int b) {
  a += b;
  if (a >= MOD) a -= MOD;
  return a;
}

void DEC(int& a, int b) {
  a -= b;
  if (a < 0) a += MOD;
}

int dff(int a, int b) {
  a -= b;
  if (a < 0) a += MOD;
  return a;
}

void MUL(int& a, int b) { a = (LL)a * b % MOD; }

int pdt(int a, int b) { return (LL)a * b % MOD; }

int _I(int b) {
  int a = MOD, x1 = 0, x2 = 1, q;
  while (1) {
    q = a / b, a %= b;
```

```
    if (!a) return x2;
    DEC(x1, pdt(q, x2));

    q = b / a, b %= a;
    if (!b) return x1;
    DEC(x2, pdt(q, x1));
  }
}

void DIV(int& a, int b) { MUL(a, _I(b)); }

int qtt(int a, int b) { return pdt(a, _I(b)); }

int pow(int a, LL b) {
  int c(1);
  while (b) {
    if (b & 1) MUL(c, a);
    MUL(a, a), b >>= 1;
  }
  return c;
}

template <class T>
T pow(T a, LL b) {
  T c(1);
  while (b) {
    if (b & 1) c *= a;
    a *= a, b >>= 1;
  }
  return c;
}

template <class T>
T pow(T a, int b) {
  return pow(a, (LL)b);
}

struct Int {
  int val;

  operator int() const { return val; }

  Int(int _val = 0) : val(_val) {
    val %= MOD;
    if (val < 0) val += MOD;
  }

  Int(LL _val) : val(_val) {
    _val %= MOD;
    if (_val < 0) _val += MOD;
    val = _val;
  }
```

```
91
92    Int& operator+=(const int& rhs) {
93      INC(val, rhs);
94      return *this;
95    }
96
97    Int operator+(const int& rhs) const { return sum(val, rhs); }
98
99    Int& operator-=(const int& rhs) {
100     DEC(val, rhs);
101     return *this;
102   }
103
104   Int operator-(const int& rhs) const { return dff(val, rhs); }
105
106   Int& operator*=(const int& rhs) {
107     MUL(val, rhs);
108     return *this;
109   }
110
111   Int operator*(const int& rhs) const { return pdt(val, rhs); }
112
113   Int& operator/=(const int& rhs) {
114     DIV(val, rhs);
115     return *this;
116   }
117
118   Int operator/(const int& rhs) const { return qtt(val, rhs); }
119
120   Int operator-() const { return MOD - *this; }
121 };
122
123 }  // namespace NT
124
125 using namespace NT;
126
127 constexpr int N = int(1e3) + 9;
128 Int binom[N][N], C[N], E[N], B[N], B1[N], G[N];
129 int n;
130
131 void ln(Int C[], Int G[]) {
132   for (int i = 1; i <= n; ++i) {
133     C[i] = G[i];
134     for (int j = 1; j <= i - 1; ++j)
135       C[i] -= binom[i - 1][j - 1] * C[j] * G[i - j];
136   }
137 }
138
139 void exp(Int G[], Int C[]) {
140   for (int i = 1; i <= n; ++i) {
141     G[i] = C[i];
142     for (int j = 1; j <= i - 1; ++j)
```

```
      G[i] += binom[i - 1][j - 1] * C[j] * G[i - j];
  }
}

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);

  n = 1000;
  for (int i = 0; i < n + 1; ++i) {
    binom[i][0] = 1;
    for (int j = 0; j < i; ++j)
      binom[i][j + 1] = binom[i - 1][j] + binom[i - 1][j + 1];
  }

  for (int i = 1; i <= n; ++i) G[i] = pow(2, binom[i][2]);
  ln(C, G);
  for (int i = 1; i <= n; ++i) G[i] = pow(2, binom[i - 1][2]);
  ln(E, G);
  for (int i = 1; i <= n; ++i) {
    G[i] = 0;
    for (int j = 0; j < i + 1; ++j) G[i] += binom[i][j] * pow(2, j * (i
- j));
  }
  ln(B1, G);
  for (int i = 1; i <= n; ++i) B1[i] /= 2;
  exp(B, B1);

  int T;
  cin >> T;
  while (T--) {
    cin >> n;
    cout << "Connected: " << C[n] << '\n'
         << "Eulerian: " << E[n] << '\n'
         << "Bipartite: " << B[n] << "\n\n";
  }
}

```

习题

- UOJ Goodbye Jihai D. 新年的追逐战[12]
- BZOJ 3864. 大朋友和多叉树[13]
- BZOJ 2863. 愤怒的元首[14]
- Luogu P6295. 有标号 DAG 计数[15]
- LOJ 6569. 仙人掌计数[16]
- LOJ 6570. 毛毛虫计数[17]
- Luogu P5434. 有标号荒漠计数[18]
- Luogu P3343. [ZJOI2015] 地震后的幻想乡[19]
- HDU 5279. YJC plays Minecraft[20]
- Luogu P7364. 有标号二分图计数[21]
- Luogu P5827. 点双连通图计数[22]

- Luogu P5827. 边双连通图计数[23]
- Luogu P6596. How Many of Them[24]
- Luogu U152448. 有标号强连通图计数[25]
- Project Euler 434. Rigid graphs[26]

Riddell's Formula

上述关于 EGF 的 exp 的用法，有时又被称作 Riddell's formula for labeled graphs，生成函数的 欧拉变换 →9.12.12@p1580，有时也被称为 Riddell's formula for unlabeled graphs，后者最早出现在欧拉对分拆数的研究中，除了解决图论计数问题之外，也在完全背包问题中出现.

对于给定序列 a_i，和对应的 OGF $A(x)$，定义 $A(x)$ 的欧拉变换为：

$$\begin{aligned}\mathcal{E}(A(x)) &= \prod_i (1-x^i)^{-a_i} \\ &= \exp(\sum_i \frac{A(x^i)}{i})\end{aligned}$$

设 $\mathcal{E}(A(x))$ 的各项系数为 b_i，定义辅助数组 $c_i = \sum_{d \mid n} d a_d$，则有递推公式

$$nb_n = c_n + \sum_{i=1}^{n-1} c_i b_{n-i}$$

无标号树

例题「SPOJ PT07D」Let us count 1 2 3

例题 「SPOJ PT07D」Let us count 1 2 3[27]

题目大意：求有 n 个结点的分别满足下列性质的树的方案数.

- 有标号有根树 A000169[28].
- 有标号无根树 A000272[29].
- 无标号有根树 A000081[30].
- 无标号无根树 A000055[31].

有根树

有标号情况以在前文中解决，下面考察无标号有根树，设其 OGF 为 $F(x)$，应用欧拉变换，可得：

$$F(x) = x\mathcal{E}(F(x))$$

取出系数即可.

无根树

考虑容斥，我们用有根树的方案中减去根不是重心的方案，并对 n 的奇偶性进行讨论.

当 n 是奇数时：

必然存在一棵子树大小 $\geq \lceil \frac{n}{2} \rceil$，枚举这棵子树的大小有.

$$g_n = f_n - \sum_{i=\lceil \frac{n}{2} \rceil}^{n-1} f_i f_{n-i}$$

当 n 是偶数时：

注意到当有两个重心的情况时，上面的过程只会减去一次，因此还需要减去

$$g_n = f_n - \sum_{i=\frac{n}{2}+1}^{n-1} f_i f_{n-i} - \binom{f_{\frac{n}{2}}}{2}$$

例题「Luogu P5900」无标号无根树计数

例题 「Luogu P5900」无标号无根树计数[32]

题目大意：求有 n 个结点的无标号无根树的方案数（$n \leq 200000$）.

对于数据范围更大的情况，做法同理，欧拉变换后使用多项式模板即可.

无标号简单图

例题「SGU 282. Isomorphism」Isomorphism

例题 「SGU 282. Isomorphism」Isomorphism[33]

题目大意：求有 n 个结点的无标号完全图的边进行 m 染色的方案数.

注意到当 m = 2 时，所求对象就是无标号简单图 A000088[34]，考察波利亚计数定理，

$$\frac{1}{|G|}\sum_{g\in G} m^{c(g)}$$

本题中置换群 G 为顶点的 n 阶对称群生成的边集置换群，但暴力做法的枚举量为 $O(n!)$，无法通过此题.

考虑根据按照置换的循环结构进行分类，每种循环结构对应一种数的分拆，我们用 dfs() 生成分拆，那么问题即转化为求每一种分拆 p 所对应的置换数目 $w(p)$ 和每一类置换中的循环个数 $c(p)$，答案为

$$\frac{1}{|G|}\sum_{p\in P} w(p)m^{c(p)}$$

考虑 $w(p)$，每一个分拆对应一个循环排列，同时同一种大小的分拆之间的顺序无关，因而我们有：

$$w(p)=\frac{n!}{\prod_i (p_i)\prod_i (q_{(i)!})}$$

这里 q_i 表示大小为 i 的分拆在 p 中出现的次数.

考虑 $c(p)$，p 所影响的点集的循环即为 $|p|$，但题目考察的是边染色，所以还需要考察点置换所生成的边置换，

如果一条边关联的顶点处在同一个循环内，设该循环大小为 p_i，那么边所生成的循环数恰好为 $\lfloor \frac{p_i}{2} \rfloor$.

如果一条边关联的顶点处在两个不同的循环中，设分别为 p_i,p_j，每个循环节的长度均为 $\mathrm{lcm}(p_i,p_j)$，因而边所生成的循环数恰好为 $\frac{p_ip_j}{\mathrm{lcm}(p_i,p_j)}=\gcd(p_i,p_j)$.

参考代码

```cpp
#include <iostream>
#include <vector>
using namespace std;

using LL = long long;
int MOD = int(1e9) + 7;

namespace NT {
void INC(int& a, int b) {
  a += b;
  if (a >= MOD) a -= MOD;
}
```

```
13
14 int sum(int a, int b) {
15   a += b;
16   if (a >= MOD) a -= MOD;
17   return a;
18 }
19
20 void DEC(int& a, int b) {
21   a -= b;
22   if (a < 0) a += MOD;
23 }
24
25 int dff(int a, int b) {
26   a -= b;
27   if (a < 0) a += MOD;
28   return a;
29 }
30
31 void MUL(int& a, int b) { a = (LL)a * b % MOD; }
32
33 int pdt(int a, int b) { return (LL)a * b % MOD; }
34
35 int _I(int b) {
36   int a = MOD, x1 = 0, x2 = 1, q;
37   while (1) {
38     q = a / b, a %= b;
39     if (!a) return x2;
40     DEC(x1, pdt(q, x2));
41
42     q = b / a, b %= a;
43     if (!b) return x1;
44     DEC(x2, pdt(q, x1));
45   }
46 }
47
48 void DIV(int& a, int b) { MUL(a, _I(b)); }
49
50 int qtt(int a, int b) { return pdt(a, _I(b)); }
51
52 int pow(int a, LL b) {
53   int c(1);
54   while (b) {
55     if (b & 1) MUL(c, a);
56     MUL(a, a), b >>= 1;
57   }
58   return c;
59 }
60
61 template <class T>
62 T pow(T a, LL b) {
63   T c(1);
64   while (b) {
```

```
    if (b & 1) c *= a;
    a *= a, b >>= 1;
  }
  return c;
}

template <class T>
T pow(T a, int b) {
  return pow(a, (LL)b);
}

struct Int {
  int val;

  operator int() const { return val; }

  Int(int _val = 0) : val(_val) {
    val %= MOD;
    if (val < 0) val += MOD;
  }

  Int(LL _val) : val(_val) {
    _val %= MOD;
    if (_val < 0) _val += MOD;
    val = _val;
  }

  Int& operator+=(const int& rhs) {
    INC(val, rhs);
    return *this;
  }

  Int operator+(const int& rhs) const { return sum(val, rhs); }

  Int& operator-=(const int& rhs) {
    DEC(val, rhs);
    return *this;
  }

  Int operator-(const int& rhs) const { return dff(val, rhs); }

  Int& operator*=(const int& rhs) {
    MUL(val, rhs);
    return *this;
  }

  Int operator*(const int& rhs) const { return pdt(val, rhs); }

  Int& operator/=(const int& rhs) {
    DIV(val, rhs);
    return *this;
  }
```

```
117
118   Int operator/(const int& rhs) const { return qtt(val, rhs); }
119
120   Int operator-() const { return MOD - *this; }
121 };
122
123 }  // namespace NT
124
125 using namespace NT;
126
127 constexpr int N = int(5e1) + 9;
128 Int Fact[N];
129 vector<vector<int>> Partition;
130 vector<int> cur;
131 int n, m;
132
133 void gen(int n = ::n, int s = 1) {
134   if (!n) {
135     Partition.push_back(cur);
136   } else if (n >= s) {
137     cur.push_back(s);
138     gen(n - s, s);
139     cur.pop_back();
140     gen(n, s + 1);
141   }
142 }
143
144 Int w(const vector<int> P) {
145   Int z = Fact[n];
146   int c = 0, l = P.front();
147
148   for (auto p : P) {
149     z /= p;
150     if (p != l) {
151       z /= Fact[c];
152       l = p;
153       c = 1;
154     } else {
155       ++c;
156     }
157   }
158
159   z /= Fact[c];
160   return z;
161 }
162
163 int gcd(int x, int y) { return y ? gcd(y, x % y) : x; }
164
165 int c(const vector<int> P) {
166   int z = 0;
167   for (int i = 0; i < P.size(); ++i) {
168     z += P[i] / 2;
```

```
169     for (int j = 0; j < i; ++j) z += gcd(P[i], P[j]);
170   }
171   return z;
172 }
173
174 int main() {
175   cin >> n >> m >> MOD;
176   Fact[0] = 1;
177   for (int i = 1; i <= n; ++i) Fact[i] = Fact[i - 1] * i;
178
179   gen();
180
181   Int res = 0;
182   for (auto P : Partition) {
183     res += w(P) * pow(m, c(P));
184   }
185   res /= Fact[n];
186   cout << res << endl;
187 }
188
```

习题

- CodeForces 438 E. The Child and Binary Tree[35]
- Luogu P5448. [THUPC2018] 好图计数[36]
- Luogu P5818. [JSOI2011] 同分异构体计数[37]
- Luogu P6597. 烯烃计数[38]
- Luogu P6598. 烷烃计数[39]
- Luogu P4128. [SHOI2006] 有色图[40]
- Luogu P4727. [HNOI2009] 图的同构计数[41]
- AtCoder Beginner Contest 222 H. Binary Tree[42]
- AtCoder Beginner Contest 284 Ex. Count Unlabeled Graphs[43]
- Luogu P4708. 画画[44]
- Luogu P7592. 数树（2021 CoE-II E）[45]
- Luogu P5206. [WC2019] 数树[46]

参考资料与注释

1. WC2015, 顾昱洲营员交流资料 Graphical Enumeration[47]
2. WC2019, 生成函数，多项式算法与图的计数[48]
3. Counting labeled graphs - Algorithms for Competitive Programming[49]
4. Graphical Enumeration Paperback, Frank Harary, Edgar M. Palmer[50]
5. The encyclopedia of integer sequences, N. J. A. Sloane, Simon Plouffe[51]
6. Combinatorial Problems and Exercises, László Lovász[52]
7. Graph Theory and Additive Combinatorics[53]

[1] OEIS

[2] 拉格朗日定理

[3] Hihocoder 1047. Random Tree

[4] 「POJ 1737」 Connected Graph

[5] 「集训队作业 2013」城市规划

[6] 「SPOJ KPGRAPHS」 Counting Graphs

[7] A001187 [8] A033678 [9] A047864

[10] 粉兔的 blog [11] A001832 [12] UOJ Goodbye Jihai D. 新年的追逐战

[13] BZOJ 3864. 大朋友和多叉树 [14] BZOJ 2863. 愤怒的元首 [15] Luogu P6295. 有标号 DAG 计数

[16] LOJ 6569. 仙人掌计数 [17] LOJ 6570. 毛毛虫计数 [18] Luogu P5434. 有标号荒漠计数

[19] Luogu P3343. [ZJOI2015] 地震后的幻想乡 [20] HDU 5279. YJC plays Minecraft [21] Luogu P7364. 有标号二分图计数

[22] Luogu P5827. 点双连通图计数 [23] Luogu P5827. 边双连通图计数 [24] Luogu P6596. How Many of Them

[25] Luogu U152448. 有标号强连通图计数 [26] Project Euler 434. Rigid graphs [27]「SPOJ PT07D」Let us count 1 2 3

[28] A000169 [29] A000272 [30] A000081

[31] A000055 [32]「Luogu P5900」无标号无根树计数 [33]「SGU 282. Isomorphism」Isomorphism

[34] A000088 [35] CodeForces 438 E. The Child and Binary Tree [36] Luogu P5448. [THUPC2018] 好图计数

[37] Luogu P5818. [JSOI2011] 同分异构体计数 [38] Luogu P6597. 烯烃计数 [39] Luogu P6598. 烷烃计数

[40] Luogu P4128. [SHOI2006] 有色图 [41] Luogu P4727. [HNOI2009] 图的同构计数 [42] AtCoder Beginner Contest 222 H. Binary Tree

[43] AtCoder Beginner Contest 284 Ex. Count Unlabeled Graphs [44] Luogu P4708. 画画 [45] Luogu P7592. 数树（2021 CoE-II E）

[46] Luogu P5206. [WC2019] 数树 [47] WC2015, 顾昱洲营员交流资料 Graphical Enumeration [48] WC2019, 生成函数，多项式算法与图的计数

[49] Counting labeled graphs - Algorithms for Competitive Programming [50] Graphical Enumeration Paperback, Frank Harary, Edgar M. Palmer [51] The encyclopedia of integer sequences, N. J. A. Sloane, Simon Plouffe

[52] Combinatorial Problems and Exercises, László Lovász [53] Graph Theory and Additive Combinatorics

9.14 线性代数

9.14.1 线性代数简介

Authors: codewasp942

提示

本篇与「线性代数」分类下的其他篇目关联不大．但笔者认为，讲讲线性代数的本质，追溯概念的根源与联系，让读者对于线性代数有一个初步但是成体系的认识，确实有其必要性．

早在几千年前，就有古人应用线性方程组解决问题，而如今，线性代数仍然应用广泛.

线性代数源于人们的观察. 人们发现，很多对象都拥有相似的性质，比如：

- 力可以被分解、合成.
- 对于任意的 k, x_0，$k\sin(x-x_0)$ 可以分解成 $k_1\sin x + k_2\cos x$.

这些性质与所描述对象的 **缩放**、**分解**、**叠加** 等有关. 线性代数把这些性质从具体对象中抽象出来，作为一个独立的学科来研究. 在 OI 中，线性代数的知识可以直接用来解决问题，也可以用于优化算法、数据结构等. 例如：

- 用树剖维护线性基求链上最大异或和
- 利用矩阵树定理把图的生成树计数问题转化为求矩阵的行列式
- 用矩阵快速幂优化递推

9.14.2 向量

在本文之前，特别说明一下翻译的相关问题. 由于历史原因，数学学科和物理学科关于「vector」一词的翻译不同.

在物理学科，一般翻译成「矢量」，并且与「标量」一词相对. 在数学学科，一般翻译成「向量」. 这种翻译的差别还有「本征」与「特征」、「幺正」与「酉」，等等.

在 OI Wiki，主要面向计算机等工程类相关学科，与数学学科关系更近一些，因此采用「向量」这个词汇.

定义及相关概念

向量: 既有大小又有方向的量称为向量. 数学上研究的向量为 **自由向量**, 即只要不改变它的大小和方向，起点和终点可以任意平行移动的向量. 记作 $\vec{a}$ 或 $\boldsymbol{a}$.

有向线段：带有方向的线段称为有向线段. 有向线段有三要素：**起点，方向，长度**，知道了三要素，终点就唯一确定. 一般使用有向线段表示向量.

向量的模：有向线段 $\overrightarrow{AB}$ 的长度称为向量的模，即为这个向量的大小. 记为：$|\overrightarrow{AB}|$ 或 $|\boldsymbol{a}|$.

零向量：模为 0 的向量. 零向量的方向任意. 记为：$\vec{0}$ 或 $\mathbf{0}$.

单位向量：模为 1 的向量称为该方向上的单位向量. 一般记为 $\vec{e}$ 或 $\boldsymbol{e}$.

平行向量：方向相同或相反的两个 **非零** 向量. 记作：$\boldsymbol{a} \parallel \boldsymbol{b}$. 对于多个互相平行的向量，可以任作一条直线与这些向量平行，那么任一组平行向量都可以平移到同一直线上，所以平行向量又叫 **共线向量**.

相等向量：模相等且方向相同的向量.

相反向量：模相等且方向相反的向量.

向量的夹角：已知两个非零向量 $\boldsymbol{a}, \boldsymbol{b}$，作 $\overrightarrow{OA} = \boldsymbol{a}, \overrightarrow{OB} = \boldsymbol{b}$，那么 $\theta = \angle AOB$ 就是向量 $\boldsymbol{a}$ 与向量 $\boldsymbol{b}$ 的夹角. 记作：$\langle \boldsymbol{a}, \boldsymbol{b} \rangle$. 显然当 $\theta = 0$ 时两向量同向，$\theta = \pi$ 时两向量反向，$\theta = \frac{\pi}{2}$ 时两向量垂直，记作 $\boldsymbol{a} \perp \boldsymbol{b}$，并且规定 $\theta \in [0, \pi]$.

注意到平面向量具有方向性，两个向量不能比较大小（但可以比较两向量的模长）. 但是两个向量可以相等.

向量的线性运算

向量的加减法

在定义了一种量之后，就希望让它具有运算. 向量的运算可以类比数的运算，从物理学的角度出发也可以研究向量的运算.

类比物理学中的位移概念，假如一个人从 A 经 B 走到 C，那么他经过的位移为 $\overrightarrow{AB}+\overrightarrow{BC}$，这其实等价于这个人直接从 A 走到 C，即 $\overrightarrow{AB}+\overrightarrow{BC}=\overrightarrow{AC}$.

注意到力的合成法则——平行四边形法则，同样也可以看做一些向量相加.

整理一下向量的加法法则：

1. **向量加法的三角形法则**：若要求和的向量首尾顺次相连，那么这些向量的和为第一个向量的起点指向最后一个向量的终点；
2. **向量加法的平行四边形法则**：若要求和的两个向量 **共起点**，那么它们的和向量为以这两个向量为邻边的平行四边形的对角线，起点为两个向量共有的起点，方向沿平行四边形对角线方向.

这样，向量的加法就具有了几何意义．并且可以验证，向量的加法满足 **交换律与结合律**.

因为实数的减法可以写成加上相反数的形式，考虑在向量做减法时也这么写．即：$\boldsymbol{a}-\boldsymbol{b}=\boldsymbol{a}+(-\boldsymbol{b})$.

这样，考虑共起点的向量，按照平行四边形法则做出它们的差，经过平移后可以发现 **「共起点向量的差向量」是由「减向量」指向「被减向量」的有向线段**．这也是向量减法的几何意义.

有时候有两点 A,B，想知道 $\overrightarrow{AB}$，可以利用减法运算 $\overrightarrow{AB}=\overrightarrow{OB}-\overrightarrow{OA}$ 获得.

向量的数乘

规定「实数 λ 与向量 $\boldsymbol{a}$ 的积」为一个向量，这种运算就是向量的 **数乘运算**，记作 $\lambda\boldsymbol{a}$，它的长度与方向规定如下：

1. $|\lambda\boldsymbol{a}|=|\lambda||\boldsymbol{a}|$；
2. 当 $\lambda>0$ 时，$\lambda\boldsymbol{a}$ 与 $\boldsymbol{a}$ 同向，当 $\lambda=0$ 时，$\lambda\boldsymbol{a}=\boldsymbol{0}$，当 $\lambda<0$ 时，$\lambda\boldsymbol{a}$ 与 $\boldsymbol{a}$ 方向相反.

根据数乘的定义，可以验证有如下运算律：

$$\lambda(\mu\boldsymbol{a})=(\lambda\mu)\boldsymbol{a}$$

$$(\lambda+\mu)\boldsymbol{a}=\lambda\boldsymbol{a}+\mu\boldsymbol{a}$$

$$\lambda(\boldsymbol{a}+\boldsymbol{b})=\lambda\boldsymbol{a}+\lambda\boldsymbol{b}$$

特别地：

$$(-\lambda)\boldsymbol{a}=-(\lambda\boldsymbol{a})=-\lambda(\boldsymbol{a})$$

$$\lambda(\boldsymbol{a}-\boldsymbol{b})=\lambda\boldsymbol{a}-\lambda\boldsymbol{b}$$

判定两向量共线

两个 **非零** 向量 $\boldsymbol{a}$ 与 $\boldsymbol{b}$ 共线 $\Longleftrightarrow$ 有唯一实数 λ，使得 $\boldsymbol{b}=\lambda\boldsymbol{a}$.

证明：由数乘的定义可知，对于 **非零** 向量 $\boldsymbol{a}$，如果存在实数 λ，使得 $\boldsymbol{b}=\lambda\boldsymbol{a}$，那么 $\boldsymbol{a}\parallel\boldsymbol{b}$.

反过来，如果 $\boldsymbol{a}\parallel\boldsymbol{b}$，$\boldsymbol{a}\neq\boldsymbol{0}$，且 $|\boldsymbol{b}|=\mu|\boldsymbol{a}|$，那么当 $\boldsymbol{a}$ 与 $\boldsymbol{b}$ 同向时，$\boldsymbol{b}=\mu\boldsymbol{a}$，反向时 $\boldsymbol{b}=-\mu\boldsymbol{a}$.

最后，向量的加，减，数乘统称为向量的线性运算.

平面向量的基本定理及坐标表示

平面向量基本定理

定理内容：如果两个向量 $\boldsymbol{e_1},\boldsymbol{e_2}$ 不共线，那么存在唯一实数对 (x,y)，使得与 $\boldsymbol{e_1},\boldsymbol{e_2}$ 共面的任意向量 $\boldsymbol{p}$ 满足 $\mathbf{p}=x\boldsymbol{e_1}+y\boldsymbol{e_2}$.

平面向量那么多，怎样用尽可能少的量表示出所有平面向量?

只用一个向量表示出所有向量显然是不可能的，最多只能表示出某条直线上的向量.

再加入一个向量，用两个 **不共线** 向量表示（两个共线向量在此可以看成同一个向量），这样可以把任意一个平面向量分解到这两个向量的方向上了.

在同一平面内的两个不共线的向量称为 **基底**．如果基底相互垂直，那么在分解的时候就是对向量 **正交分解**.

平面向量的坐标表示

如果取与横轴与纵轴方向相同的单位向量 i, j 作为一组基底，根据平面向量基本定理，平面上的所有向量与有序实数对 (x, y) 一一对应.

而有序实数对 (x, y) 与平面直角坐标系上的点一一对应，于是作 $\overrightarrow{OP} = \boldsymbol{p}$，那么终点 $P(x, y)$ 也是唯一确定的. 由于研究的对象是自由向量，可以自由平移起点，这样，在平面直角坐标系里，每一个向量都可以用有序实数对唯一表示.

平面向量的坐标运算

平面向量线性运算

由平面向量的线性运算可以推导其坐标运算，主要方法是将坐标全部化为用基底表示，然后利用运算律进行合并，之后表示出运算结果的坐标形式.

若两向量 $\boldsymbol{a} = (m, n)$，$\boldsymbol{b} = (p, q)$，则：

$$\begin{aligned} \boldsymbol{a} + \boldsymbol{b} &= (m + p, n + q) \\ \boldsymbol{a} - \boldsymbol{b} &= (m - p, n - q) \\ k\boldsymbol{a} &= (km, kn) \end{aligned}$$

求一个向量的坐标表示

已知两点 $A(a, b), B(c, d)$，易证 $\overrightarrow{AB} = (c - a, d - b)$.

平移一点

有时需要将一个点 P 沿一定方向平移某单位长度，这样把要平移的方向和距离组合成一个向量，利用向量加法的三角形法则，将 $\overrightarrow{OP}$ 加上这个向量，得到的向量终点即为平移后的点.

三点共线的判定

若 A, B, C 三点共线，则 $\overrightarrow{OB} = \lambda\overrightarrow{OA} + (1 - \lambda)\overrightarrow{OC}$.

三点共线判定的拓展

在三角形 ABC 中，若 D 为 BC 的 n 等分点（$n\ BD = k\ DC$），则有：$\overrightarrow{AD} = \frac{n}{k+n}\overrightarrow{AB} + \frac{k}{k+n}\overrightarrow{AC}$

在三维空间中的拓展（立体几何/空间向量）

在空间中，以上部分所述的所有内容均成立. 更有：

空间向量基本定理

定理内容：如果三个向量 $\boldsymbol{e_1}, \boldsymbol{e_2}, \boldsymbol{e_3}$ 不共面，那么存在唯一实数对 (x, y, z)，使得空间中任意向量 $\boldsymbol{p}$ 满足 $\mathbf{p} = x\boldsymbol{e_1} + y\boldsymbol{e_2} + z\boldsymbol{e_3}$. 根据空间向量基本定理，我们同样可以使用三个相互垂直的基底 $\boldsymbol{e_1}, \boldsymbol{e_2}, \boldsymbol{e_3}$ 作为正交基底，建立 **空间直角坐标系** 并用一个三元组 (x, y, z) 作为坐标表示空间向量.

共面向量基本定理

如果存在两个不共线的向量 $\boldsymbol{x}, \boldsymbol{y}$，则向量 $\boldsymbol{p}$ 与 $\boldsymbol{x}, \boldsymbol{y}$ 共面的充要条件是存在唯一实数对 (a, b) 使得 $\boldsymbol{p} = a\boldsymbol{x} + b\boldsymbol{y}$.

方向向量

空间直线的方向用一个与该直线平行的非零向量来表示，该向量称为这条直线的一个方向向量. 直线在空间中的位置，由它经过的空间一点及它的一个方向向量 **完全确定**.

注意，平面中的直线也有方向向量.

对于 **空间** 中的直线，对其方向向量有以下求法：

- 若有 $A(x_1, y_1, z_1), B(x_2, y_2, z_2)$，则 AB 所在直线的一个方向向量为 $\boldsymbol{s} = (x_2 - x_1, y_2 - y_1, z_2 - z_1)$.

- 若已知一个与所求直线 **垂直** 的平面，该平面一般方程为 $ax+by+cz+d=0$，那么垂直于该平面的直线的一个方向向量为 $\boldsymbol{s}=(a,b,c)$，该方向向量也是该平面的 **一个法向量**.

法向量

对于一个面 $ABCD$，其法向量 $\boldsymbol{n}$ 与这个面垂直.

计算方法：任取两个面内直线 $\overrightarrow{AB},\overrightarrow{AD}$，使得 $\overrightarrow{AB}\cdot\boldsymbol{n}=\boldsymbol{0}$ 且 $\overrightarrow{AD}\cdot\boldsymbol{n}=\boldsymbol{0}$，利用坐标法即可计算.

向量与矩阵

线性代数中，线性变换可以用矩阵表示. 令 T 表示一个将 $\mathbf{R}^n$ 映射到 $\mathbf{R}^m$ 的线性变换，$\mathbf{x}$ 表示一个 n 维列向量，则存在一个 $m\times n$ 矩阵 A，使得

$$T(\mathbf{x})=A\mathbf{x}.$$

矩阵 A 称为线性变换 T 的变换矩阵. 在算法问题中，一般情况下线性变换在相同维度下进行，因此 A 是一个方阵. 这样，对向量的线性变换问题可以转化为矩阵乘法问题.

接下来我们探讨三种竞赛中较为常见的变换与其对应的变换矩阵：放缩变换（变换矩阵用 S 表示）、旋转变换（变换矩阵用 R 表示）和平移变换（变换矩阵用 T 表示）.

放缩变换

对于 n 维列向量 $\boldsymbol{a}$，将其每一维放缩 $v_1,v_2,...,v_n$ 倍. 很容易发现放缩操作的变换矩阵 R 是 $n\times n$ 的对角矩阵，即 $S=\mathrm{diag}\{v_1,v_2,...,v_n\}$.

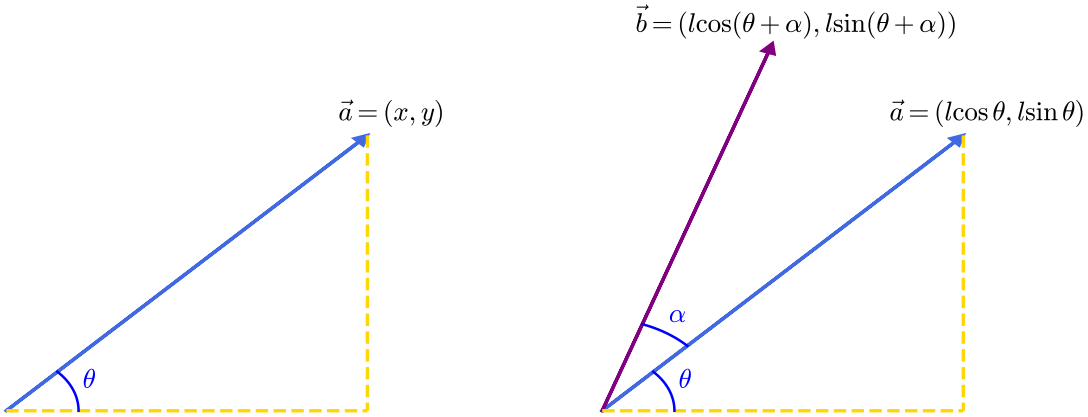
旋转变换

向量的旋转是相对复杂的操作，我们仅限于讨论二维和三维的情况.

向量绕点旋转

对于向量绕点旋转，一般指的是向量绕原点旋转. 对于某一点绕另一点 P 旋转，可以利用平移变换使得点 P 位于原点，进行向量旋转后再将坐标系平移回原位置即可. 设平移操作的变换矩阵为 T，绕原点旋转操作的变换矩阵为 R，则整个过程的变换矩阵为 TRT^{-1}. 根据几何意义，T^{-1} 一定存在.

对于二维空间，设 $\boldsymbol{a}=(x,y)$，倾角为 θ，长度为 $l=\sqrt{x^2+y^2}$. 则 $x=l\cos\theta,y=l\sin\theta$. 令其绕原点逆时针旋转 α 角，得到向量 $\boldsymbol{b}=(l\cos(\theta+\alpha),l\sin(\theta+\alpha))$.



由三角恒等变换得，

$$\boldsymbol{b}=(l(\cos\theta\cos\alpha-\sin\theta\sin\alpha),l(\sin\theta\cos\alpha+\cos\theta\sin\alpha))$$

化简，

$$\boldsymbol{b}=(l\cos\theta\cos\alpha-l\sin\theta\sin\alpha,l\sin\theta\cos\alpha+l\cos\theta\sin\alpha)$$

把上面的 x, y 代回来得

$$\boldsymbol{b} = (x\cos\alpha - y\sin\alpha, y\cos\alpha + x\sin\alpha)$$

因此二维空间下，变换矩阵 R 为

$$R = \begin{bmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{bmatrix}.$$

对于三维空间，向量旋转需要使用两个角度参量，即天顶角旋转角度与方向角旋转角度，可以利用 空间球坐标系→9.9@p1149 进行旋转操作.

向量绕直线旋转

对于三维向量，更常见的是绕某直线旋转．同样为了方便，此直线是过原点的．如果直线不过原点，我们仍可以平移坐标系进行转化.

取直线的方向向量 $\boldsymbol{u} = (u_x, u_y, u_z)$，设三维向量绕其逆时针旋转 θ 角．则对应的变换矩阵 R 为[1]

$$R = \begin{bmatrix} u_x^2(1-\cos\theta)+\cos\theta & u_xu_y(1-\cos\theta)-u_z\sin\theta & u_xu_z(1-\cos\theta)+u_y\sin\theta \\ u_xu_y(1-\cos\theta)+u_z\sin\theta & u_y^2(1-\cos\theta)+\cos\theta & u_yu_z(1-\cos\theta)-u_x\sin\theta \\ u_xu_z(1-\cos\theta)-u_y\sin\theta & u_yu_z(1-\cos\theta)+u_x\sin\theta & u_z^2(1-\cos\theta)+\cos\theta \end{bmatrix}.$$

平移变换

平移变换并非线性变换，而是仿射变换．但 $\mathbf{R}^n$ 下的仿射变换仍可以用 $\mathbf{R}^{n+1}$ 下的线性变换表示.

考虑 n 维向量 $\boldsymbol{a} = (a_1, a_2, ..., a_n)$，现在要将其沿向量 $\boldsymbol{t} = (t_1, t_2, ..., t_n)$ 平移．我们对列向量 $\boldsymbol{a}$ 添加一维并置为 1，得到新列向量 $\boldsymbol{a}' = (a_1, a_2, ..., a_n, 1)$．则变换矩阵 T 可以写作

$$T = \begin{bmatrix} 1 & & & t_1 \\ & 1 & & t_2 \\ & & \ddots & \vdots \\ & & 1 & t_n \\ & & & 1 \end{bmatrix}.$$

对于其他线性变换矩阵，在矩阵中增加一列与一行，除右下角的元素为 1 外其它部分填充为 0，通过这种方法，所有的线性变换矩阵都可以转换为仿射变换矩阵．例如，对于二维向量旋转，变换矩阵可以变为

$$R' = \begin{bmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

向量的更严格定义

上文中，向量被定义为了空间中的有向线段．但是严格来说，向量不仅是有向线段．要作出向量的更严格定义，需要先定义 线性空间→9.14.7@p1731，具体内容参见 线性空间→9.14.7@p1731 页面的介绍.

[1] Rotation matrix from axis and angle - Wikipedia

9.14.3 内积和外积

本文介绍向量之间的简单运算.

在本文之前，特别说明一下翻译的相关问题．由于历史原因，数学学科和物理学科关于「inner product」和「outer product」两个词汇有着五花八门的翻译.

在物理学科，一般翻译成「标积」和「矢积」，表示运算的结果为标量和矢量．高中数学课本上「数量积」和「向量积」也采用了这种意译的办法.

[1]参见 Rotation matrix from axis and angle - Wikipedia[1]

在数学学科，通常也可以翻译成「内积」和「外积」，是两个名词的直译.「点乘」和「叉乘」是根据运算符号得来的俗称，这种俗称也很常见.

在「点乘」运算中，经常省略运算的点符号，在线性代数中更是会直接看作矩阵乘法，不写点符号.

内积

内积的概念 **对于任意维数的向量都适用**.

定义

内积有不同但等价的定义方法，下面介绍其中一些.

几何定义

在 n 维欧氏空间 $\mathbf{R}^n$ 下，已知两个向量 $\boldsymbol{a},\boldsymbol{b}$，它们的夹角为 θ，那么：

$$\boldsymbol{a}\cdot\boldsymbol{b}=|\boldsymbol{a}||\boldsymbol{b}|\cos\theta$$

就是这两个向量的 **内积**，也叫 **点积** 或 **数量积**. 其中称 $|\boldsymbol{b}|\cos\theta$ 为 $\boldsymbol{b}$ 在 $\boldsymbol{a}$ 方向上的投影. 内积的几何意义即为：内积 $\boldsymbol{a}\cdot\boldsymbol{b}$ 等于 $\boldsymbol{a}$ 的模与 $\boldsymbol{b}$ 在 $\boldsymbol{a}$ 方向上的投影的乘积.

代数定义

在 n 维欧氏空间 $\mathbf{R}^n$ 下，已知两个向量 $\boldsymbol{a}=(a_1,a_2,...,a_n),\boldsymbol{b}=(b_1,b_2,...,b_n)$，那么：

$$\boldsymbol{a}\cdot\boldsymbol{b}=\sum_{i=1}^{n}a_ib_i$$

就是这两个向量的 **内积**，也叫 **点积** 或 **数量积**. 内积的几何定义与代数定义在欧氏空间下是等价的，而后者更方便使用.

在不引起混淆的情况下，内积的点号可以省略不写. 如果在向量的右上角有上角标 2，表示向量与自身内积的简写，即 **向量模长的平方**，省略模长记号. 该上角标 2 不可以理解为向量的平方，这是因为，向量内积的结果为标量，不存在除了 2 以外任何个数的向量的内积. 同理，向量模长平方的平方，不可以简写为上角标 4，而是必须将上角标 2 的结果视为一个整体，以此类推.

性质

可以发现，内积得到的结果是一个标量，其特别之处在于，它是关于两个向量分别都线性的双线性运算. 具体而言，内积满足：

$$(\boldsymbol{a}+\boldsymbol{b})\cdot\boldsymbol{c}=\boldsymbol{a}\cdot\boldsymbol{c}+\boldsymbol{b}\cdot\boldsymbol{c}$$

$$\boldsymbol{a}\cdot(\boldsymbol{b}+\boldsymbol{c})=\boldsymbol{a}\cdot\boldsymbol{b}+\boldsymbol{a}\cdot\boldsymbol{c}$$

$$(\lambda\boldsymbol{a})\cdot\boldsymbol{b}=\lambda(\boldsymbol{a}\cdot\boldsymbol{b})$$

$$\boldsymbol{a}\cdot(\lambda\boldsymbol{b})=\lambda(\boldsymbol{a}\cdot\boldsymbol{b})$$

内积还满足交换律，即：

$$\boldsymbol{a}\cdot\boldsymbol{b}=\boldsymbol{b}\cdot\boldsymbol{a}$$

应用

下面介绍内积运算的一些常见应用.

1. 判定两向量垂直：

   $\boldsymbol{a}\perp\boldsymbol{b}\iff\boldsymbol{a}\cdot\boldsymbol{b}=0$

   即互相垂直的两个向量的内积，结果为 0；向量与零向量内积，结果为 0. 如果使用内积为零作为垂直的定义，则可以得出零向量与任何向量都垂直.

2. 判定两向量共线：

   $\exists\lambda\in\mathbf{R}(\boldsymbol{a}=\lambda\boldsymbol{b})\iff|\boldsymbol{a}\cdot\boldsymbol{b}|=|\boldsymbol{a}||\boldsymbol{b}|$

3. 计算向量的模：

   $|\boldsymbol{a}|=\sqrt{\boldsymbol{a}\cdot\boldsymbol{a}}$

4. 计算两向量的夹角：

$$\theta = \arccos \frac{\boldsymbol{a} \cdot \boldsymbol{b}}{|\boldsymbol{a}|\,|\boldsymbol{b}|}$$

二阶与三阶行列式

二阶与三阶行列式，可以作为行列式的较为简单的情形特殊定义．在微积分的最后一个部分场论部分，格林公式用到了二阶行列式，高斯公式用到了点乘，斯托克斯公式用到了三阶行列式．

二阶行列式可以视为四元函数，其定义为：

$$\begin{vmatrix} a & b \\ c & d \end{vmatrix} = ad - bc$$

三阶行列式可以视为九元函数，其定义为：

$$\begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix} = aei + dhc + gbf - ahf - dbi - gec$$

一种特殊的记忆方法是采用「对角线法则」，对角线法则只适用于二阶与三阶行列式．

特别注意：四阶行列式展开后共有 24 项，并且副对角线一项的符号为正．如果强行应用三阶行列式的「对角线法则」，不仅项数不够，副对角线一项的符号也不正确，因此三阶行列式的「对角线法则」不适用于更高阶的行列式，更高阶的行列式也不适合使用直接展开法计算．

外积

外积是 **三维向量特有的运算**．

在物理学中，三维向量为默认与空间位置相关的向量，一律采用粗体表示．然而，物理学中与相对论相关的四维向量不会采用粗体，而是使用特殊的记号与下标．

在线性代数中，所有的向量都会用粗体表示，并且由于麻烦，并且线性代数中大多为向量与矩阵的运算，很难造成歧义，在手写时可以省略向量记号不写．

定义

外积有不同但等价的定义方法，下面介绍其中一些．

几何定义

在三维欧氏空间 $\mathbf{R}^3$ 下，定义向量 $\boldsymbol{a}, \boldsymbol{b}$ 的外积为一个向量，记为 $\boldsymbol{a} \times \boldsymbol{b}$，其模与方向定义如下：

1. $|\boldsymbol{a} \times \boldsymbol{b}| = |\boldsymbol{a}|\,|\boldsymbol{b}| \sin\langle \boldsymbol{a}, \boldsymbol{b} \rangle$；
2. $\boldsymbol{a} \times \boldsymbol{b}$ 与 $\boldsymbol{a}, \boldsymbol{b}$ 都垂直，且 $\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{a} \times \boldsymbol{b}$ 的方向符合右手法则．

注意到外积的模，联想到三角形面积计算公式 $S = \frac{1}{2} ab \sin C$，可以发现外积的几何意义是：$|\boldsymbol{a} \times \boldsymbol{b}|$ **是以 $\boldsymbol{a}, \boldsymbol{b}$ 为邻边的平行四边形的面积**．

代数定义

在三维欧氏空间 $\mathbf{R}^3$ 下，定义向量 $\boldsymbol{a} = (x_1, y_1, z_1), \boldsymbol{b} = (x_2, y_2, z_2)$ 的外积为一个向量 $\boldsymbol{c}$，记作 $\boldsymbol{c} = \boldsymbol{a} \times \boldsymbol{b}$，其结果可以使用三阶行列式表示：

$$\begin{vmatrix} \boldsymbol{i} & \boldsymbol{j} & \boldsymbol{k} \\ x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \end{vmatrix}$$

其中 $\boldsymbol{i}, \boldsymbol{j}, \boldsymbol{k}$ 表示朝向为坐标轴 x, y, z 的单位向量，并写在对应坐标处．展开得

$$\begin{aligned} \boldsymbol{c} &= \boldsymbol{a} \times \boldsymbol{b} \\ &= (y_1 z_2 - y_2 z_1)\boldsymbol{i} + (z_1 x_2 - z_2 x_1)\boldsymbol{j} + (x_1 y_2 - x_2 y_1)\boldsymbol{k} \\ &= (y_1 z_2 - y_2 z_1, z_1 x_2 - z_2 x_1, x_1 y_2 - x_2 y_1) \end{aligned}$$

性质

1. 外积是关于两个向量分别都线性的双线性运算．具体而言，外积满足：

$$
\begin{aligned}
(\boldsymbol{a}+\boldsymbol{b})\times\boldsymbol{c}&=\boldsymbol{a}\times\boldsymbol{c}+\boldsymbol{b}\times\boldsymbol{c}\\
\boldsymbol{a}\times(\boldsymbol{b}+\boldsymbol{c})&=\boldsymbol{a}\times\boldsymbol{b}+\boldsymbol{a}\times\boldsymbol{c}\\
(\lambda\boldsymbol{a})\times\boldsymbol{b}&=\lambda(\boldsymbol{a}\times\boldsymbol{b})\\
\boldsymbol{a}\times(\lambda\boldsymbol{b})&=\lambda(\boldsymbol{a}\times\boldsymbol{b})
\end{aligned}
$$

   前两行性质亦可称为分配律，即外积对于向量加法满足乘法分配律.

2. 外积满足反交换律，即：

   $\boldsymbol{a}\times\boldsymbol{b}=-\boldsymbol{b}\times\boldsymbol{a}$

3. 根据上文内积与外积的几何定义：

$$
\begin{aligned}
|\boldsymbol{a}\times\boldsymbol{b}|&=|\boldsymbol{a}|\,|\boldsymbol{b}|\sin\langle\boldsymbol{a},\boldsymbol{b}\rangle\\
\boldsymbol{a}\cdot\boldsymbol{b}&=|\boldsymbol{a}|\,|\boldsymbol{b}|\cos\theta\\
&=|\boldsymbol{a}|\,|\boldsymbol{b}|\cos\langle\boldsymbol{a},\boldsymbol{b}\rangle
\end{aligned}
$$

   可以写出恒等式：

   $(\boldsymbol{a}\times\boldsymbol{b})\cdot(\boldsymbol{a}\times\boldsymbol{b})=|\boldsymbol{a}|^2\,|\boldsymbol{b}|^2-(\boldsymbol{a}\cdot\boldsymbol{b})^2$

4. 外积满足 Jacobi 恒等式：

   $\boldsymbol{a}\times(\boldsymbol{b}\times\boldsymbol{c})+\boldsymbol{b}\times(\boldsymbol{c}\times\boldsymbol{a})+\boldsymbol{c}\times(\boldsymbol{a}\times\boldsymbol{b})=\boldsymbol{0}$

应用

下面介绍外积运算的一些常见应用.

1. 判定两向量是否共线：

   $\exists\lambda\in\mathbf{R}(\boldsymbol{a}=\lambda\boldsymbol{b})\iff\boldsymbol{a}\times\boldsymbol{b}=\boldsymbol{0}$

   即共线的两个三维向量的外积，结果为 $\boldsymbol{0}$；三维向量与自身外积，结果为 $\boldsymbol{0}$；三维向量与零向量外积，结果为 $\boldsymbol{0}$．若使用外积为零作为两向量共线的定义，则可以得出零向量与任何向量都共线.

2. 计算两向量张成的平行四边形面积：

   $S\langle\boldsymbol{a},\boldsymbol{b}\rangle=|\boldsymbol{a}\times\boldsymbol{b}|$

二维向量的情形

对于二维向量，无法计算外积，但是仍然可以计算两向量张成的平行四边形面积：

记 $\boldsymbol{a}=(m,n),\boldsymbol{b}=(p,q)$，将平面直角坐标系扩充为空间直角坐标系，原平面位于新坐标系的 xOy 平面，原本的坐标 (m,n) 和 (p,q) 变为 $(m,n,0)$ 和 $(p,q,0)$.

那么两个向量的外积为 $(0,0,mq-np)$，因此平行四边形的面积为 $|\,mq-np\,|$，可以视为二阶行列式运算结果的绝对值.

此时，根据右手法则和 z 坐标的符号，可以推断出 $\boldsymbol{b}$ 相对于 $\boldsymbol{a}$ 的方向，若在逆时针方向则 z 坐标为正值，反之为负值，简记为 **顺负逆正**.

混合积

与外积一样，向量的混合积是 **三维向量特有的运算**.

定义

设 $\boldsymbol{a},\boldsymbol{b},\boldsymbol{c}$ 是三维空间中的三个向量，则 $(\boldsymbol{a}\times\boldsymbol{b})\cdot\boldsymbol{c}$ 称为三个向量 $\boldsymbol{a},\boldsymbol{b},\boldsymbol{c}$ 的混合积，记作 $[\boldsymbol{abc}]$ 或 $(\boldsymbol{a},\boldsymbol{b},\boldsymbol{c})$ 或 $(\boldsymbol{abc})$ 或 $\det(\boldsymbol{a},\boldsymbol{b},\boldsymbol{c})$．混合积的绝对值 $|(\boldsymbol{a}\times\boldsymbol{b})\cdot\boldsymbol{c}|$ 的几何意义表示以 $\boldsymbol{a},\boldsymbol{b},\boldsymbol{c}$ 为棱的平行六面体的体积.

向量的混合积可以使用三阶行列式表示：

$$\begin{aligned}(\boldsymbol{a}\times\boldsymbol{b})\cdot\boldsymbol{c}&=\det(\boldsymbol{a},\boldsymbol{b},\boldsymbol{c})\\&=\begin{vmatrix}a_x & b_x & c_x\\ a_y & b_y & c_y\\ a_z & b_z & c_z\end{vmatrix}\\&=a_xb_yc_z+a_yb_zc_x+a_zb_xc_y-a_zb_yc_x-a_yb_xc_z-a_xb_zc_y\end{aligned}$$

性质

1. 混合积关于三个向量都分别线性，具体而言，有：

$$\begin{aligned}\det(\lambda\boldsymbol{u}+\mu\boldsymbol{v},\boldsymbol{b},\boldsymbol{c})&=\lambda\det(\boldsymbol{u},\boldsymbol{b},\boldsymbol{c})+\mu\det(\boldsymbol{v},\boldsymbol{b},\boldsymbol{c})\\\det(\boldsymbol{a},\lambda\boldsymbol{u}+\mu\boldsymbol{v},\boldsymbol{c})&=\lambda\det(\boldsymbol{a},\boldsymbol{u},\boldsymbol{c})+\mu\det(\boldsymbol{a},\boldsymbol{v},\boldsymbol{c})\\\det(\boldsymbol{a},\boldsymbol{b},\lambda\boldsymbol{u}+\mu\boldsymbol{v})&=\lambda\det(\boldsymbol{a},\boldsymbol{b},\boldsymbol{u})+\mu\det(\boldsymbol{a},\boldsymbol{b},\boldsymbol{v})\end{aligned}$$

2. 混合积具有反对称性，交换两个向量的位置会使混合积变成其相反数，因此有：

   $\det(\boldsymbol{a},\boldsymbol{b},\boldsymbol{c})=\det(\boldsymbol{b},\boldsymbol{c},\boldsymbol{a})=\det(\boldsymbol{c},\boldsymbol{a},\boldsymbol{b})=-\det(\boldsymbol{b},\boldsymbol{a},\boldsymbol{c})=-\det(\boldsymbol{a},\boldsymbol{c},\boldsymbol{b})=-\det(\boldsymbol{c},\boldsymbol{b},\boldsymbol{a})$

   据此还可以得到内积与外积有如下关系：

   $(\boldsymbol{a}\times\boldsymbol{b})\cdot\boldsymbol{c}=\boldsymbol{a}\cdot(\boldsymbol{b}\times\boldsymbol{c})$

应用

向量的混合积有如下常见应用.

1. 计算四面体 $ABCD$ 的体积：

   $V=\dfrac{1}{6}\left|\det(\overrightarrow{AB},\overrightarrow{AC},\overrightarrow{AD})\right|$

2. 判定 $\boldsymbol{a},\boldsymbol{b},\boldsymbol{c}$ 是否共面；

   三个三维向量 $\boldsymbol{a},\boldsymbol{b},\boldsymbol{c}$ 共面的充分必要条件是 $\det(\boldsymbol{a},\boldsymbol{b},\boldsymbol{c})=0$.

3. 判定 $\boldsymbol{a},\boldsymbol{b},\boldsymbol{c}$ 构成的坐标系的手性；

   混合积 $\det(\boldsymbol{a},\boldsymbol{b},\boldsymbol{c})$ 的符号是正还是负，取决于 $\boldsymbol{a}\times\boldsymbol{b}$ 与 $\boldsymbol{c}$ 形成的夹角是锐角还是钝角，即指向 $\boldsymbol{a}$ 与 $\boldsymbol{b}$ 张成平面的同侧还是异侧，这相当于 $\boldsymbol{a},\boldsymbol{b},\boldsymbol{c}$ 三个向量依序构成右手系还是左手系. 具体而言：

   - $\det(\boldsymbol{a},\boldsymbol{b},\boldsymbol{c})<0$ 等价于 $\boldsymbol{a},\boldsymbol{b},\boldsymbol{c}$ 依序构成左手系；
   - $\det(\boldsymbol{a},\boldsymbol{b},\boldsymbol{c})>0$ 等价于 $\boldsymbol{a},\boldsymbol{b},\boldsymbol{c}$ 依序构成右手系.

二重外积

三维向量的混合积是内积与外积的混搭，具有轮换对称性. 三维向量和三维向量的外积还是三维向量，那么外积的外积是否存在相关结论?

先证明一个引理.

$$(\boldsymbol{a}\times\boldsymbol{b})\times\boldsymbol{a}=(\boldsymbol{a}\cdot\boldsymbol{a})\boldsymbol{b}-(\boldsymbol{a}\cdot\boldsymbol{b})\boldsymbol{a}$$

证明：由右手定则，$\boldsymbol{a}\times\boldsymbol{b}$ 与 $\boldsymbol{a}$ 和 $\boldsymbol{b}$ 都垂直，待证等式左端与 $\boldsymbol{a}\times\boldsymbol{b}$ 垂直，因此待证等式左端与 $\boldsymbol{a}$ 和 $\boldsymbol{b}$ 共面.

因此可以假设：

$$(\boldsymbol{a}\times\boldsymbol{b})\times\boldsymbol{a}=\lambda\boldsymbol{a}+\mu\boldsymbol{b}$$

根据混合积的相关结论，上式两端同时对于 $\boldsymbol{a}$ 和 $\boldsymbol{b}$ 分别做内积，有：

$$\begin{aligned}\lambda(\boldsymbol{a}\cdot\boldsymbol{a})+\mu(\boldsymbol{a}\cdot\boldsymbol{b})&=0\\\lambda(\boldsymbol{a}\cdot\boldsymbol{b})+\mu(\boldsymbol{b}\cdot\boldsymbol{b})&=\det(\boldsymbol{b},\boldsymbol{a}\times\boldsymbol{b},\boldsymbol{a})\\&=(\boldsymbol{a}\times\boldsymbol{b})\cdot(\boldsymbol{a}\times\boldsymbol{b})\end{aligned}$$

由前文推出的恒等式：

$$(\boldsymbol{a}\times\boldsymbol{b})\cdot(\boldsymbol{a}\times\boldsymbol{b})=|\boldsymbol{a}|^2|\boldsymbol{b}|^2-(\boldsymbol{a}\cdot\boldsymbol{b})^2$$

可以解得：

$$\lambda = -\boldsymbol{a}\cdot\boldsymbol{b}$$
$$\mu = \boldsymbol{a}\cdot\boldsymbol{a}$$

证毕.

在上文的证明中提到，$\boldsymbol{a}\times\boldsymbol{b}$ 与任意向量叉乘，得到的向量与 $\boldsymbol{a}$ 和 $\boldsymbol{b}$ 共面．接下来证明 **二重外积** 的结论：

$$(\boldsymbol{a}\times\boldsymbol{b})\times\boldsymbol{c} = (\boldsymbol{a}\cdot\boldsymbol{c})\boldsymbol{b} - (\boldsymbol{b}\cdot\boldsymbol{c})\boldsymbol{a}$$

上述共面性有助于二重外积结论的记忆．可见，上文的引理为二重外积的特殊情况.

证明：这里只需考虑三个向量均为非零且不共线的情况，其他特例为显然的.

三维向量 $\boldsymbol{a}$，$\boldsymbol{b}$ 和 $\boldsymbol{a}\times\boldsymbol{b}$ 不共面，因此可以假设：

$$\boldsymbol{c} = \alpha\boldsymbol{a} + \beta\boldsymbol{b} + \gamma(\boldsymbol{a}\times\boldsymbol{b})$$

所以有：

$$\begin{aligned}(\boldsymbol{a}\times\boldsymbol{b})\times\boldsymbol{c} &= (\boldsymbol{a}\times\boldsymbol{b})\times(\alpha\boldsymbol{a} + \beta\boldsymbol{b} + \gamma(\boldsymbol{a}\times\boldsymbol{b}))\\ &= \alpha(\boldsymbol{a}\times\boldsymbol{b})\times\boldsymbol{a} + \beta(\boldsymbol{a}\times\boldsymbol{b})\times\boldsymbol{b}\end{aligned}$$

根据上文的引理有：

$$\begin{aligned}(\boldsymbol{a}\times\boldsymbol{b})\times\boldsymbol{a} &= (\boldsymbol{a}\cdot\boldsymbol{a})\boldsymbol{b} - (\boldsymbol{a}\cdot\boldsymbol{b})\boldsymbol{a}\\ (\boldsymbol{a}\times\boldsymbol{b})\times\boldsymbol{b} &= -(\boldsymbol{b}\times\boldsymbol{a})\times\boldsymbol{b}\\ &= -(\boldsymbol{b}\cdot\boldsymbol{b})\boldsymbol{a} + (\boldsymbol{a}\cdot\boldsymbol{b})\boldsymbol{b}\end{aligned}$$

因此有：

$$\begin{aligned}(\boldsymbol{a}\times\boldsymbol{b})\times\boldsymbol{c} &= \alpha((\boldsymbol{a}\cdot\boldsymbol{a})\boldsymbol{b} - (\boldsymbol{a}\cdot\boldsymbol{b})\boldsymbol{a}) + \beta((\boldsymbol{a}\cdot\boldsymbol{b})\boldsymbol{b} - (\boldsymbol{b}\cdot\boldsymbol{b})\boldsymbol{a})\\ &= (\alpha(-\boldsymbol{a}\cdot\boldsymbol{b}) + \beta(-\boldsymbol{b}\cdot\boldsymbol{b}))\boldsymbol{a} + (\alpha\boldsymbol{a}\cdot\boldsymbol{a} + \beta\boldsymbol{a}\cdot\boldsymbol{b})\boldsymbol{b}\\ &= (\boldsymbol{a}\cdot\boldsymbol{c})\boldsymbol{b} - (\boldsymbol{b}\cdot\boldsymbol{c})\boldsymbol{a}\end{aligned}$$

证毕.

根据外积的反交换性，可以得到二重外积的两个公式：

$$(\boldsymbol{a}\times\boldsymbol{b})\times\boldsymbol{c} = (\boldsymbol{a}\cdot\boldsymbol{c})\boldsymbol{b} - (\boldsymbol{b}\cdot\boldsymbol{c})\boldsymbol{a}$$
$$\boldsymbol{a}\times(\boldsymbol{b}\times\boldsymbol{c}) = (\boldsymbol{a}\cdot\boldsymbol{c})\boldsymbol{b} - (\boldsymbol{a}\cdot\boldsymbol{b})\boldsymbol{c}$$

可见，二重外积对于运算顺序有着严格的要求.

借助混合积与二重外积，还可以证明拉格朗日的恒等式.

$$(\boldsymbol{a}\times\boldsymbol{b})\cdot(\boldsymbol{c}\times\boldsymbol{d}) = (\boldsymbol{a}\cdot\boldsymbol{c})(\boldsymbol{b}\cdot\boldsymbol{d}) - (\boldsymbol{a}\cdot\boldsymbol{d})(\boldsymbol{b}\cdot\boldsymbol{c})$$

证明：

$$\begin{aligned}(\boldsymbol{a}\times\boldsymbol{b})\cdot(\boldsymbol{c}\times\boldsymbol{d}) &= \det(\boldsymbol{c},\boldsymbol{d},\boldsymbol{a}\times\boldsymbol{b})\\ &= \det(\boldsymbol{a}\times\boldsymbol{b},\boldsymbol{c},\boldsymbol{d})\\ &= ((\boldsymbol{a}\times\boldsymbol{b})\times\boldsymbol{c})\cdot\boldsymbol{d}\\ &= (\boldsymbol{b}(\boldsymbol{a}\cdot\boldsymbol{c}) - \boldsymbol{a}(\boldsymbol{b}\cdot\boldsymbol{c}))\cdot\boldsymbol{d}\\ &= (\boldsymbol{a}\cdot\boldsymbol{c})(\boldsymbol{b}\cdot\boldsymbol{d}) - (\boldsymbol{a}\cdot\boldsymbol{d})(\boldsymbol{b}\cdot\boldsymbol{c})\end{aligned}$$

可见，前文的恒等式

$$(\boldsymbol{a}\times\boldsymbol{b})\cdot(\boldsymbol{a}\times\boldsymbol{b}) = |\boldsymbol{a}|^2|\boldsymbol{b}|^2 - (\boldsymbol{a}\cdot\boldsymbol{b})^2$$

是拉格朗日的恒等式的特殊情形.

9.14.4 矩阵

本文介绍线性代数中一个非常重要的内容——矩阵（Matrix），主要讲解矩阵的性质、运算，以及矩阵乘法的一些应用.

向量与矩阵

在线性代数中，向量分为列向量和行向量.

> ⚠ **Warning**
>
> 在中国台湾地区关于「列」与「行」的翻译，恰好与中国大陆地区相反. 在 OI Wiki 按照中国大陆地区的习惯，采用列（column）与行（row）的翻译.

线性代数的主要研究对象是列向量，约定使用粗体小写字母表示列向量. 在用到大量向量与矩阵的线性代数中，不引起混淆的情况下，在手写时，字母上方的向量记号可以省略不写.

向量也是特殊的矩阵. 如果想要表示行向量，需要在粗体小写字母右上方写转置记号. 行向量在线性代数中一般表示方程.

引入

矩阵的引入来自于线性方程组. 与向量类似，矩阵体现了一种对数据「打包处理」的思想.

例如，将线性方程组：

$$\begin{cases} 7x_1 + 8x_2 + 9x_3 = 13 \\ 4x_1 + 5x_2 + 6x_3 = 12 \\ x_1 + 2x_2 + 3x_3 = 11 \end{cases}$$

一般用圆括号或方括号表示矩阵. 将上述系数抽出来，写成矩阵乘法的形式：

$$\begin{pmatrix} 7 & 8 & 9 \\ 4 & 5 & 6 \\ 1 & 2 & 3 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 13 \\ 12 \\ 11 \end{pmatrix}$$

简记为：

$$Ax = b$$

即未知数列向量 x，左乘一个矩阵 A，得到列向量 b. 这个式子可以认为是线性代数的基本形式.

线性代数主要研究的运算模型是内积. 内积是先相乘再相加，是行向量左乘列向量，得到一个数的过程.

矩阵乘法是内积的拓展. 矩阵乘法等价于左边矩阵抽出一行，与右边矩阵抽出一列进行内积，得到结果矩阵的对应元素，口诀「左行右列」.

当研究对象是右边的列向量时，矩阵乘法相当于对列向量进行左乘. 在左乘的观点下，矩阵就是对列向量的变换，将矩阵乘法中右边矩阵的每一个列向量进行变换，对应地得到结果矩阵中每一个列向量.

矩阵可以对一个列向量进行变换，也可以对一组列向量进行「打包」变换，甚至可以对整个空间——即全体列向量进行变换. 当矩阵被视为对整个空间变换的时候，也就脱离了空间，成为了纯粹变换的存在.

定义

对于矩阵 A，主对角线是指 $A_{i,i}$ 的元素.

一般用 I 来表示单位矩阵，就是主对角线上为 1，其余位置为 0.

同型矩阵

两个矩阵，行数与列数对应相同，称为同型矩阵.

方阵

行数等于列数的矩阵称为方阵. 方阵是一种特殊的矩阵. 对于「n 阶矩阵」的习惯表述，实际上讲的是 n 阶方阵. 阶数相同的方阵为同型矩阵.

研究方程组、向量组、矩阵的秩的时候，使用一般的矩阵. 研究特征值和特征向量、二次型的时候，使用方阵.

主对角线

方阵中行数等于列数的元素构成主对角线.

对称矩阵

如果方阵的元素关于主对角线对称，即对于任意的 i 和 j，i 行 j 列的元素与 j 行 i 列的元素相等，则将方阵称为对称矩阵.

对角矩阵

主对角线之外的元素均为 0 的方阵称为对角矩阵，一般记作:

$$\mathrm{diag}\{\lambda_1,\cdots,\lambda_n\}$$

式中的 $\lambda_1,\cdots,\lambda_n$ 是主对角线上的元素.

对角矩阵是对称矩阵.

如果对角矩阵的元素均为 1，称为单位矩阵，记为 I. 只要乘法可以进行，无论形状，任何矩阵乘单位矩阵仍然保持不变.

三角矩阵

如果方阵主对角线左下方的元素均为 0, 称为上三角矩阵. 如果方阵主对角线右上方的元素均为 0, 称为下三角矩阵.

两个上（下）三角矩阵的乘积仍然是上（下）三角矩阵. 如果对角线元素均非 0，则上（下）三角矩阵可逆，逆也是上（下）三角矩阵.

单位三角矩阵

如果上三角矩阵 A 的对角线全为 1，则称 A 是单位上三角矩阵. 如果下三角矩阵 A 的对角线全为 1，则称 A 是单位下三角矩阵.

两个单位上（下）三角矩阵的乘积仍然是单位上（下）三角矩阵，单位上（下）三角矩阵的逆也是单位上（下）三角矩阵.

运算

矩阵的线性运算

矩阵的线性运算分为加减法与数乘，它们均为逐个元素进行. 只有同型矩阵之间可以对应相加减.

矩阵的转置

矩阵的转置，就是在矩阵的右上角写上转置「T」记号，表示将矩阵的行与列互换.

对称矩阵转置前后保持不变.

矩阵乘法

矩阵的乘法是向量内积的推广.

矩阵相乘只有在第一个矩阵的列数和第二个矩阵的行数相同时才有意义.

设 A 为 $P\times M$ 的矩阵，B 为 $M\times Q$ 的矩阵，设矩阵 C 为矩阵 A 与 B 的乘积,

其中矩阵 C 中的第 i 行第 j 列元素可以表示为:

$$C_{i,j}=\sum_{k=1}^{M}A_{i,k}B_{k,j}$$

在矩阵乘法中，结果 C 矩阵的第 i 行第 j 列的数，就是由矩阵 A 第 i 行 M 个数与矩阵 B 第 j 列 M 个数分别 **相乘再相加** 得到的. 这里的 **相乘再相加**，就是向量的内积. 乘积矩阵中第 i 行第 j 列的数恰好是乘数矩阵 A 第 i 个行向量与乘数矩阵 B 第 j 个列向量的内积，口诀为 **左行右列**.

线性代数研究的向量多为列向量，根据这样的对矩阵乘法的定义方法，经常研究对列向量左乘一个矩阵的左乘运算，同时也可以在这里看出「打包处理」的思想，同时处理很多个向量内积.

矩阵乘法满足结合律，不满足一般的交换律.

利用结合律，矩阵乘法可以利用 快速幂→9.7@p1131 的思想来优化.

在比赛中，由于线性递推式可以表示成矩阵乘法的形式，也通常用矩阵快速幂来求线性递推数列的某一项.

优化

首先对于比较小的矩阵，可以考虑直接手动展开循环以减小常数.

可以重新排列循环以提高空间局部性，这样的优化不会改变矩阵乘法的时间复杂度，但是会得到常数级别的提升.

```
// 以下文的参考代码为例
mat operator*(const mat& T) const {
  mat res;
  for (int i = 0; i < sz; ++i)
    for (int j = 0; j < sz; ++j)
      for (int k = 0; k < sz; ++k) {
        res.a[i][j] += mul(a[i][k], T.a[k][j]);
        res.a[i][j] %= MOD;
      }
  return res;
}

// 不如
mat operator*(const mat& T) const {
  mat res;
  int r;
  for (int i = 0; i < sz; ++i)
    for (int k = 0; k < sz; ++k) {
      r = a[i][k];
      for (int j = 0; j < sz; ++j)
        res.a[i][j] += T.a[k][j] * r, res.a[i][j] %= MOD;
    }
  return res;
}
```

方阵的逆

方阵 A 的逆矩阵 P 是使得 $A \times P = I$ 的矩阵.

逆矩阵不一定存在. 如果存在，可以使用 高斯消元→9.19.3@p1944 进行求解.

方阵的行列式

行列式是方阵的一种运算.

参考代码

一般来说，可以用一个二维数组来模拟矩阵.

```
struct mat {
  LL a[sz][sz];

  mat() { memset(a, 0, sizeof a); }
```

```
5
6   mat operator-(const mat& T) const {
7     mat res;
8     for (int i = 0; i < sz; ++i)
9       for (int j = 0; j < sz; ++j) {
10        res.a[i][j] = (a[i][j] - T.a[i][j]) % MOD;
11      }
12    return res;
13  }
14
15  mat operator+(const mat& T) const {
16    mat res;
17    for (int i = 0; i < sz; ++i)
18      for (int j = 0; j < sz; ++j) {
19        res.a[i][j] = (a[i][j] + T.a[i][j]) % MOD;
20      }
21    return res;
22  }
23
24  mat operator*(const mat& T) const {
25    mat res;
26    int r;
27    for (int i = 0; i < sz; ++i)
28      for (int k = 0; k < sz; ++k) {
29        r = a[i][k];
30        for (int j = 0; j < sz; ++j)
31          res.a[i][j] += T.a[k][j] * r, res.a[i][j] %= MOD;
32      }
33    return res;
34  }
35
36  mat operator^(LL x) const {
37    mat res, bas;
38    for (int i = 0; i < sz; ++i) res.a[i][i] = 1;
39    for (int i = 0; i < sz; ++i)
40      for (int j = 0; j < sz; ++j) bas.a[i][j] = a[i][j] % MOD;
41    while (x) {
42      if (x & 1) res = res * bas;
43      bas = bas * bas;
44      x >>= 1;
45    }
46    return res;
47  }
48 };
```

看待线性方程组的两种视角

看待矩阵 A，或者变换 A，有两种视角.

第一种观点：按行看，观察 A 的每一行. 这样一来把 A 看作方程组. 于是就有了消元法解方程的过程.

第二种观点：按列看，观察 A 的每一列. A 本身也是由列向量构成的. 此时相当于把变换 A 本身看成了列向量组，而 x 是未知数系数，思考 A 当中的这组列向量能不能配上未知数，凑出列向量 b.

例如，文章开头的例子变为：

$$\begin{pmatrix}7\\4\\1\end{pmatrix}x_1+\begin{pmatrix}8\\5\\2\end{pmatrix}x_2+\begin{pmatrix}9\\6\\3\end{pmatrix}x_3=\begin{pmatrix}13\\12\\11\end{pmatrix}$$

解方程变为研究，是否可以通过调整三个系数 x，使得给定的三个基向量能够凑出结果的向量.

按列看比按行看更新颖. 在按列看的视角下，可以研究线性无关与线性相关.

矩阵乘法的应用

矩阵加速递推

以 斐波那契数列（Fibonacci Sequence）→9.13.4@p1629 为例. 在斐波那契数列当中，$F_1=F_2=1$，$F_i=F_{i-1}+F_{i-2}(i\geq 3)$.

如果有一道题目让你求斐波那契数列第 n 项的值，最简单的方法莫过于直接递推了. 但是如果 n 的范围达到了 10^{18} 级别，递推就不行了，此时我们可以考虑矩阵加速递推.

根据斐波那契数列 递推公式的矩阵形式→9.13.4@p1629:

$$\begin{bmatrix}F_{n-1} & F_{n-2}\end{bmatrix}\begin{bmatrix}1 & 1\\1 & 0\end{bmatrix}=\begin{bmatrix}F_n & F_{n-1}\end{bmatrix}$$

定义初始矩阵 $\text{ans}=\begin{bmatrix}F_2 & F_1\end{bmatrix}=\begin{bmatrix}1 & 1\end{bmatrix}$, $\text{base}=\begin{bmatrix}1 & 1\\1 & 0\end{bmatrix}$. 那么，$F_n$ 就等于 $\text{ans}\ \text{base}^{n-2}$ 这个矩阵的第一行第一列元素，也就是 $\begin{bmatrix}1 & 1\end{bmatrix}\begin{bmatrix}1 & 1\\1 & 0\end{bmatrix}^{n-2}$ 的第一行第一列元素.

> ⚠ **注意**
>
> 矩阵乘法不满足交换律，所以一定不能写成 $\begin{bmatrix}1 & 1\\1 & 0\end{bmatrix}^{n-2}\begin{bmatrix}1 & 1\end{bmatrix}$ 的第一行第一列元素. 另外，对于 $n\leq 2$ 的情况，直接输出 1 即可，不需要执行矩阵快速幂.

为什么要乘上 base 矩阵的 $n-2$ 次方而不是 n 次方呢? 因为 F_1,F_2 是不需要进行矩阵乘法就能求的. 也就是说，如果只进行一次乘法，就已经求出 F_3 了. 如果还不是很理解为什么幂是 $n-2$，建议手算一下.

下面是求斐波那契数列第 n 项对 10^9+7 取模的示例代码（核心部分）.

```
constexpr int mod = 1000000007;

struct Matrix {
  int a[3][3];

  Matrix() { memset(a, 0, sizeof a); }

  Matrix operator*(const Matrix &b) const {
    Matrix res;
    for (int i = 1; i <= 2; ++i)
      for (int j = 1; j <= 2; ++j)
        for (int k = 1; k <= 2; ++k)
          res.a[i][j] = (res.a[i][j] + a[i][k] * b.a[k][j]) % mod;
    return res;
  }
} ans, base;

void init() {
  base.a[1][1] = base.a[1][2] = base.a[2][1] = 1;
  ans.a[1][1] = ans.a[1][2] = 1;
}

void qpow(int b) {
```

```
  while (b) {
    if (b & 1) ans = ans * base;
    base = base * base;
    b >>= 1;
  }
}

int main() {
  int n = read();
  if (n <= 2) return puts("1"), 0;
  init();
  qpow(n - 2);
  println(ans.a[1][1] % mod);
}
```

这是一个稍微复杂一些的例子.

$$f_1 = f_2 = 0$$

$$f_n = 7f_{n-1} + 6f_{n-2} + 5n + 4 \times 3^n$$

我们发现，f_n 和 f_{n-1}, f_{n-2}, n 有关，于是考虑构造一个矩阵描述状态.

但是发现如果矩阵仅有这三个元素 $[f_n \ \ f_{n-1} \ \ n]$ 是难以构造出转移方程的，因为乘方运算和 $+1$ 无法用矩阵描述.

于是考虑构造一个更大的矩阵.

$$[f_n \ \ f_{n-1} \ \ n \ \ 3^n \ \ 1]$$

我们希望构造一个递推矩阵可以转移到

$$[f_{n+1} \ \ f_n \ \ n+1 \ \ 3^{n+1} \ \ 1]$$

转移矩阵即为

$$\begin{bmatrix} 7 & 1 & 0 & 0 & 0 \\ 6 & 0 & 0 & 0 & 0 \\ 5 & 0 & 1 & 0 & 0 \\ 12 & 0 & 0 & 3 & 0 \\ 5 & 0 & 1 & 0 & 1 \end{bmatrix}$$

矩阵表达修改

「THUSCH 2017」大魔法师[1]

大魔法师小 L 制作了 n 个魔力水晶球，每个水晶球有水、火、土三个属性的能量值. 小 L 把这 n 个水晶球在地上从前向后排成一行，然后开始今天的魔法表演.

我们用 A_i, B_i, C_i 分别表示从前向后第 i 个水晶球（下标从 1 开始）的水、火、土的能量值.

小 L 计划施展 m 次魔法. 每次，他会选择一个区间 $[l, r]$，然后施展以下 3 大类、7 种魔法之一：

1. 魔力激发：令区间里每个水晶球中 **特定属性** 的能量爆发，从而使另一个 **特定属性** 的能量增强. 具体来说，有以下三种可能的表现形式：
   - 火元素激发水元素能量：令 $A_i = A_i + B_i$.
   - 土元素激发火元素能量：令 $B_i = B_i + C_i$.
   - 水元素激发土元素能量：令 $C_i = C_i + A_i$.

   需要注意的是，增强一种属性的能量并不会改变另一种属性的能量，例如 $A_i = A_i + B_i$ 并不会使 B_i 增加或减少.

2. 魔力增强：小 L 挥舞法杖，消耗自身 v 点法力值，来改变区间里每个水晶球的 **特定属性** 的能量．具体来说，有以下三种可能的表现形式：
   - 火元素能量定值增强：令 $A_i = A_i + v$.
   - 水元素能量翻倍增强：令 $B_i = B_i \cdot v$.
   - 土元素能量吸收融合：令 $C_i = v$.
3. 魔力释放：小 L 将区间里所有水晶球的能量聚集在一起，融合成一个新的水晶球，然后送给场外观众. 生成的水晶球每种属性的能量值等于区间内所有水晶球对应能量值的代数和. **需要注意的是，魔力释放的过程不会真正改变区间内水晶球的能量.**

值得一提的是，小 L 制造和融合的水晶球的原材料都是定制版的 OI 工厂水晶，所以这些水晶球有一个能量阈值 998244353. 当水晶球中某种属性的能量值大于等于这个阈值时，能量值会自动对阈值取模，从而避免水晶球爆炸.

小 W 为小 L（唯一的）观众，围观了整个表演，并且收到了小 L 在表演中融合的每个水晶球．小 W 想知道，这些水晶球蕴涵的三种属性的能量值分别是多少.

由于矩阵的结合律和分配律成立，单点修改可以自然地推广到区间，即推出矩阵后直接用线段树维护区间矩阵乘积即可.

下面将举几个例子.

$A_i = A_i + v$ 的转移

$$[A\ \ B\ \ C\ \ 1]\begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&1&0\\v&0&0&1\end{bmatrix} = [A+v\ \ B\ \ C\ \ 1]$$

$B_i = B_i \cdot v$ 的转移

$$[A\ \ B\ \ C\ \ 1]\begin{bmatrix}1&0&0&0\\0&v&0&0\\0&0&1&0\\0&0&0&1\end{bmatrix} = [A\ \ B\cdot v\ \ C\ \ 1]$$

「LibreOJ 6208」树上询问[2]

有一棵 n 节点的树，根为 1 号节点．每个节点有两个权值 k_i, t_i，初始值均为 0.

给出三种操作：

1. Add(x, d) 操作：将 x 到根的路径上所有点的 $k_i \leftarrow k_i + d$
2. Mul(x, d) 操作：将 x 到根的路径上所有点的 $t_i \leftarrow t_i + d \times k_i$
3. Query(x) 操作：询问点 x 的权值 t_x

$n,\ m \le 100000,\ -10 \le d \le 10$

若直接思考，下放操作和维护信息并不是很好想．但是矩阵可以轻松地表达.

$$[k\ \ t\ \ 1]\begin{bmatrix}1&0&0\\0&1&0\\d&0&1\end{bmatrix} = [k+d\ \ t\ \ 1]$$

$$[k\ \ t\ \ 1]\begin{bmatrix}1&d&0\\0&1&0\\0&0&1\end{bmatrix} = [k\ \ t+d\times k\ \ 1]$$

定长路径统计

问题描述

给一个 n 阶有向图，每条边的边权均为 1，然后给一个整数 k，你的任务是对于所有点对 (u, v) 求出从 u 到 v 长度为 k 的路径的数量（不一定是简单路径，即路径上的点或者边可能走多次）.

我们将这个图用邻接矩阵 G（对于图中的边 $(u \to v)$，令 $G[u, v] = 1$，其余为 0 的矩阵；如果有重边，则设 $G[u, v]$ 为重边的数量）表示这个有向图. 下述算法同样适用于图有自环的情况.

显然，该邻接矩阵对应 $k = 1$ 时的答案.

假设我们知道长度为 k 的路径条数构成的矩阵，记为矩阵 C_k，我们想求 C_{k+1}. 显然有 DP 转移方程

$$C_{k+1}[i, j] = \sum_{p=1}^{n} C_k[i, p] \cdot G[p, j]$$

我们可以把它看作矩阵乘法的运算，于是上述转移可以描述为

$$C_{k+1} = C_k \cdot G$$

那么把这个递推式展开可以得到

$$C_k = \underbrace{G \cdot G \cdots G}_{k\text{次}} = G^k$$

要计算这个矩阵幂，我们可以使用快速幂（二进制取幂）的思想，在 $O(n^3 \log k)$ 的复杂度内计算结果.

定长最短路

问题描述

给你一个 n 阶加权有向图和一个整数 k. 对于每个点对 (u, v) 找到从 u 到 v 的恰好包含 k 条边的最短路的长度.（不一定是简单路径，即路径上的点或者边可能走多次）

我们仍构造这个图的邻接矩阵 G，$G[i, j]$ 表示从 i 到 j 的边权. 如果 i, j 两点之间没有边，那么 $G[i, j] = \infty$.（有重边的情况取边权的最小值）

显然上述矩阵对应 $k = 1$ 时问题的答案. 我们仍假设我们知道 k 的答案，记为矩阵 L_k. 现在我们想求 $k + 1$ 的答案. 显然有转移方程

$$L_{k+1}[i, j] = \min_{1 \le p \le n} \{L_k[i, p] + G[p, j]\}$$

事实上我们可以类比矩阵乘法，你发现上述转移只是把矩阵乘法的乘积求和变成相加取最小值，于是我们定义这个运算为 $\odot$，即

$$A \odot B = C \iff C[i, j] = \min_{1 \le p \le n} \{A[i, p] + B[p, j]\}$$

于是得到

$$L_{k+1} = L_k \odot G$$

展开递推式得到

$$L_k = \underbrace{G \odot \ldots \odot G}_{k\text{次}} = G^{\odot k}$$

我们仍然可以用矩阵快速幂的方法计算上式，因为它显然是具有结合律的. 时间复杂度 $O(n^3 \log k)$.

限长路径计数/最短路

上述算法只适用于边数固定的情况. 然而我们可以改进算法以解决边数小于等于 k 的情况. 具体地，考虑以下问题：

问题描述

给一个 n 阶有向图，边权为 1，然后给一个整数 k，你的任务是对于每个点对 (u,v) 找到从 u 到 v 长度小于等于 k 的路径的数量（不一定是简单路径，即路径上的点或者边可能走多次）.

我们对于每个点 v，建立一个虚点 v' 用于记录答案，并在图中加入 (v,v') 和 (v',v') 这两条边．那么对于点对 (u,v)，从 u 到 v 边数小于等于 k 的路径的数量，就和从 u 到 v' 边数恰好等于 $k+1$ 的路径的数量相等，这是因为对于任意一条边数为 $m(m\le k)$ 的路径 $(p_0=u)\to p_1\to p_2\to\ldots\to p_{m-1}\to(p_m=v)$，都存在一条边数为 $k+1$ 的路径 $(p_0=u)\to p_1\to p_2\to\ldots\to p_{m-1}\to(p_m=v)\to v'\to\ldots\to v'$ 与之一一对应.

对于求边数小于等于 k 的最短路，只需对每个点加一个边权为 0 的自环即可.

习题

- 洛谷 P1962 斐波那契数列[3]，即上面的例题，同题 POJ3070
- 洛谷 P1349 广义斐波那契数列[4]，base 矩阵需要变化一下
- 洛谷 P1939【模板】矩阵加速（数列）[5]，base 矩阵变成了 3×3 的矩阵，推导过程与上面差不多.

本页面部分内容译自博文 Кратчайшие пути фиксированной длины, количества путей фиксированной длины[6] 与其英文翻译版 Number of paths of fixed length/Shortest paths of fixed length[7]．其中俄文版版权协议为 Public Domain + Leave a Link；英文版版权协议为 CC-BY-SA 4.0．

外部参考资料

[1]「THUSCH 2017」大魔法师

[2]「LibreOJ 6208」树上询问

[3] 洛谷 P1962 斐波那契数列

[4] 洛谷 P1349 广义斐波那契数列

[5] 洛谷 P1939【模板】矩阵加速（数列）

[6] Кратчайшие пути фиксированной длины, количества путей фиксированной длины

[7] Number of paths of fixed length/Shortest paths of fixed length

9.14.5 初等变换

初等矩阵

以下三类方阵称为初等矩阵.

倍乘矩阵

倍乘矩阵是一种特殊的对角矩阵.

$$D_i(k)=\operatorname{diag}\{1,\cdots,1,k,1,\cdots,1\}$$

表示一个对角阵，主对角线上第 i 个元素为 k，并且规定 k 不能为 0，其余的元素全部为 1.

特别地，当 k 为 1 的时候，$D_i(1)$ 就是单位阵 I.

对换矩阵

对换矩阵是一种特殊的对称矩阵.

$$P_{ij}=\begin{pmatrix} I_{i-1} & & & & \\ & 0 & & 1 & \\ & & I_{j-i-1} & & \\ & 1 & & 0 & \\ & & & & I_{n-j} \end{pmatrix}$$

对换矩阵的元素全是 1 和 0，主对角线上其余元素均为 1，仅有第 i 个元素和第 j 个元素为 0，而在第 i 行第 j 列、第 j 行第 i 列上的两个元素为 1.

对换矩阵要求 i 与 j 不能相等.

倍加矩阵

倍加矩阵是在单位阵 I 的基础上，令第 i 行第 j 列为 k.

$$T_{ij}(k)=\begin{pmatrix} 1 & & & & & & \\ & \ddots & & & & & \\ & & 1 & \cdots & k & & \\ & & & \ddots & \vdots & & \\ & & & & 1 & & \\ & & & & & \ddots & \\ & & & & & & 1 \end{pmatrix}$$

倍加矩阵要求 i 与 j 不能相等. 如果 k 为 0，则 $T_{ij}(0)$ 退化为单位阵 I.

倍加矩阵是一种上三角矩阵或者下三角矩阵.

初等矩阵的行列式

三种初等矩阵具有行列式：

$$|D_i(k)|=k$$

$$|P_{ij}|=-1$$

$$|T_{ij}(k)|=1$$

由于方阵乘法的行列式等于行列式的乘法，借助下文初等变换与矩阵乘法的等价性，初等矩阵的这个性质可以用于行列式的计算.

初等变换

不仅限于方阵，对于一般的矩阵 A，可以进行初等行变换和初等列变换，统称为初等变换.

初等行变换与初等列变换一样，都有 3 种：倍乘（multiplication）、对换（switching）、倍加（addition）. 这里先介绍初等行变换：

- 第 i 行乘非零数 k：$B\mapsto D_i(k)B$.
- 第 i，j 行互换：$B\mapsto P_{ij}B$.
- 第 j 行乘 k 加到第 i 行：$B\mapsto T_{ij}(k)B$.

将上述操作的行改为列，即得到初等列变换.

在初等变换中，对换可以通过倍乘和倍加实现. 显然，倍加不能通过倍乘和对换实现. 借助行列式的知识，以及下文的初等变换与矩阵乘法的等价性，也能说明倍乘不能通过倍加和对换实现.

因此，相较对换而言，倍乘和倍加是更为本质的操作. 对换操作是为了在消元法中，保证消元的有序，而引入的辅助操作.

初等变换与矩阵乘法

可以发现，三类初等矩阵都是在单位阵 I 上进行一次相应的变换得到的结果. 在后文的线性变换中指出，线性变换与矩阵之间有对应关系，与这里的关系类似.

无论矩阵 A 是否方阵，对矩阵 A 进行初等行变换，等价于对矩阵 A 左乘初等矩阵. 对矩阵 A 进行初等列变换，等价于对矩阵 A 右乘初等矩阵.

倍乘操作

左乘一个倍乘矩阵 $D_i(k)$，等价于将第 i 行变为 k 倍. 右乘一个倍乘矩阵 $D_i(k)$，等价于将第 i 列变为 k 倍.

对角阵乘对角阵还是对角阵，对于对角阵的乘法，将主对角线上对应的元素相乘．由于单位阵是特殊的倍乘阵，而倍乘阵要求 k 不为 0，可以看出，只要对角阵主对角线上的元素均非 0，就可以拆分为倍乘阵的乘积．

对于一般的对角阵，无论元素是否为 0，也有相应的结论．左乘对角阵，等价于将对应的行变为原来的若干倍，倍数恰为对角阵主对角线上的相应元素．右乘对角阵，是对相应的列进行同样操作．

由于倍乘矩阵 $D_i(k)$ 的行列式为 k，对于方阵的行或列进行倍乘操作之后，方阵对应的行列式变为原来的 k 倍．对角阵的行列式为主对角线元素的乘积．

倍乘矩阵的乘法可以交换，对角阵的乘法也可以交换，在乘法只有对角阵时，顺序可以任意排列．

单位阵对应的倍乘操作为保持矩阵 A 不变，在实际应用中不进行这样的操作．

对换操作

左乘一个对换矩阵 P_{ij}，等价于将第 i 行与第 j 行交换．右乘一个对换矩阵 P_{ij}，等价于将第 i 列与第 j 列交换．

与倍乘阵和对角阵的关系类似，这里引入置换矩阵的概念．置换矩阵是一个方阵，每行每列均恰有一个 1，其余位置均为 0．单位阵 I 也是特殊的置换矩阵．

置换阵和对于单位阵 I 的行进行置换操作一致，也和对于单位阵 I 的列进行置换操作一致．单位阵 I 本身对应于恒等变换．

左乘一个置换矩阵等价于对原矩阵的行进行置换，右乘一个置换矩阵等价于对原矩阵的列进行置换，相应置换的方法和对于单位阵 I 的行或列进行置换操作一致．

置换矩阵与置换完全对应，置换矩阵构成的乘法群与置换群同构．由于有定理，在恒等变换视为零个对换的乘积的情形下，任何置换都可以拆为对换的乘积，因此任何置换矩阵也可以拆分为对换矩阵的乘积．

由于对换矩阵的行列式为 -1，对于方阵的行或列进行对换操作之后，方阵对应的行列式变为原来的 -1 倍．

对换阵的乘法不可交换，置换阵的乘法也不可交换．

置换矩阵的行列式为 $(-1)^p$，其中 p 为置换矩阵对应置换的逆序数，即置换拆分为对换乘积的个数．

倍加操作

左乘倍加矩阵 $T_{ij}(k)$ 等价于把第 j 行的 k 倍加到第 i 行上．右乘倍加矩阵 $T_{ij}(k)$ 等价于把第 i 列的 k 倍加到第 j 列上．

如果难以记忆，可以观察倍加阵 $T_{ij}(k)$ 是对单位阵 I 进行了怎样的操作，两者是对应的，左乘是对行的操作，右乘是对列的操作，符合口诀左行右列．

由于倍加矩阵的行列式为 1，对于方阵进行倍加操作之后，方阵对应的行列式不变．

倍加矩阵的乘法不可交换．

单位阵对应的倍加操作为保持矩阵 A 不变，在实际应用中不进行这样的操作．

上三角矩阵

倍加矩阵是一种上三角矩阵或者下三角矩阵．由于两种矩阵关于主对角线对称，这里讨论上三角矩阵．事实上在这个例子中，只需要进行初等行变换，而不需要列变换．

如果一个上三角矩阵的主对角线均为 1，则可拆分为一连串倍加矩阵的乘积．拆分的顺序为，先对单位矩阵 I 的第一行进行倍加操作，再对单位矩阵 I 的第二行进行倍加操作，以此类推，直到每一行均被操作完毕为止．

由于倍加矩阵的乘法不可交换，上述操作不可调换顺序．

如果一个上三角矩阵的主对角线均非 0，则可拆分为一连串倍加矩阵和倍乘矩阵的乘积. 可以在操作单位矩阵 I 的每一行时，先将该行进行倍乘操作，效果为主对角线元素变为指定非零值.

如果一个上三角矩阵的主对角线存在 0，则不可拆分为一连串初等矩阵的乘积.

无论上三角矩阵的主对角线上是否有 0, 上三角矩阵的行列式等于主对角线元素乘积, 与对角阵一致.

倍加操作将方阵转化为对角阵

只使用倍加操作可以使任意一个方阵变为对角阵，这个例子既需要初等行变换也需要初等列变换.

如果方阵的第一行和第一列存在非零元素, 则可以通过倍加办法将左上角元素变为非零, 进而借助初等行变换和初等列变换，将第一行和第一列除了左上角元素以外，均变为 0.

如果方阵的第一行和第一列已经均为 0，则直接看第二行和第二列即可.

借助这个办法，甚至可以规定对角阵的非零元素均在左上角.

如果方阵的第一行和第一列已经均为 0，则看剩余的行列是否有非零元素，只要有非零元素，则可以通过倍加操作将第一行和第一列中某个元素变为非 0，进而化归为一开始的情况，使得左上角元素非 0.

仅当剩余的行列也均没有非零元素时，左上角无法变为非零元素，此时剩余的方阵已经为零矩阵.

标准形矩阵

借助初等变换可以将任意的矩阵，无论形状，化归为标准形矩阵.

标准形矩阵拥有一个单位阵 I 作为子矩阵位于左上角，其余部分均为 0. 化归的办法与将方阵转化为对角阵的操作类似，并需要借助倍乘操作使左上角非零元素变为 1.

矩阵转化为标准形矩阵后，含有元素 1 的个数恰好为矩阵的秩.

可逆矩阵

设 A 是一个 n 阶矩阵. 如果存在一个 n 阶矩阵 B，使得 $AB = BA = I$，那么 A 叫做一个可逆矩阵或非奇异矩阵，B 叫做 A 的逆矩阵，并记为 A^{-1}.

如果矩阵 A 可逆，那么 A 的逆矩阵由 A 唯一确定.

可逆矩阵 A 的逆 A^{-1} 也可逆，并且 A^{-1} 的逆就是 A.

两个可逆矩阵 A 和 B 的乘积 AB 也可逆，并且逆为 $B^{-1}A^{-1}$.

可逆矩阵 A 的转置 A^T 也可逆，并且转置的逆等于逆的转置.

初等矩阵的逆

初等矩阵均可逆，并且逆为同类的初等矩阵:

$$D_i(k)^{-1} = D_i\left(\frac{1}{k}\right)$$

$$P_{ij}^{-1} = P_{ij}$$

$$T_{ij}(k)^{-1} = T_{ij}(-k)$$

显然单位阵 I 可逆，逆矩阵仍为 I.

初等变换保持矩阵的可逆性，变换前后矩阵要么同时可逆，要么同时不可逆.

矩阵 A 可逆，当且仅当矩阵 A 可以写成初等矩阵的乘积，即可以通过初等变换变为单位阵 I.

等到引入行列式之后可以知道:

矩阵 A 可逆，当且仅当矩阵 A 的秩为 n，当且仅当矩阵 A 的行列式非 0.

一种简单的记法为：记 E_{ij} 为第 i 行第 j 列的元素为 1、其余为零的 $n \times n$ 矩阵，那么

- $D_i(k) = I_n + (k-1)E_{ii}$

- $P_{ij}=I_n-E_{ii}-E_{jj}+E_{ij}+E_{ji}$
- $T_{ij}(k)=I_n+kE_{ij}$

这种记法也可以应用于它们的逆矩阵.

应用

线性方程组求解

对于一个线性方程组, 未知数前的系数构成系数矩阵, 如果在系数矩阵右端补上线性方程组的常数项则构成增广矩阵.

应用初等行变换，可以将线性方程组对应的增广矩阵先转化为行阶梯形矩阵，再转化为行最简形矩阵，进而完成线性方程组的求解. 这个方法叫做消元法解线性方程组，后文的 Gauss–Jordan 消元，是按照一定的顺序进行的消元算法.

行列式计算

由于方阵乘积的行列式等于方阵行列式的乘积, 初等矩阵的行列式便于计算, 以及初等变换等价于初等矩阵的乘法，在行列式计算中也会使用初等变换.

由于按照一定的顺序进行初等变换更加便于程序书写, 行列式计算也可以使用后文的 Gauss–Jordan 消元算法.

9.14.6 行列式

行列式，是方阵的一种运算. 对于方阵 A，$\det A$ 表示方阵 A 的行列式.

本文介绍行列式的三种定义. 可以证明，本文中的定义方法是等价的.

全排列方法定义

前置知识：置换→9.8@p1138、逆序数→9.8@p1138.

手动计算较低阶的行列式可以采用这种方法，它的时间复杂度为阶乘量级.

使用记号 $\pi(j_1j_2\cdots j_n)$ 表示排列 $j_1j_2\cdots j_n$ 的逆序数，S_n 为全体长度为 n 的排列构成的集合. 记号:

$$\begin{aligned}\det A&=\begin{vmatrix}a_{11}&a_{12}&\cdots&a_{1n}\\a_{21}&a_{22}&\cdots&a_{2n}\\\vdots&\vdots&&\vdots\\a_{n1}&a_{n2}&\cdots&a_{nn}\end{vmatrix}\\&=\sum_{(j_1j_2\cdots j_n)\in S_n}(-1)^{\pi(j_1j_2\cdots j_n)}a_{1j_1}a_{2j_2}\ldots a_{nj_n}\end{aligned}$$

表示的 n 阶行列式是指 $n!$ 项的代数和，这些项是一切可能的取自方阵 A 中不同的行与不同的列上的 n 个元素的乘积 $a_{1j_1}a_{2j_2}\cdots a_{nj_n}$.

项 $a_{1j_1}a_{2j_2}\cdots a_{nj_n}$ 前面的符号是 $(-1)^{\pi(j_1j_2\cdots j_n)}$，也就是说，当 $j_1j_2\cdots j_n$ 是偶排列时，符号为正，当 $j_1j_2\cdots j_n$ 是奇排列时，符号为负.

对于二三阶行列式的对角线法则, 事实上就是采用了全排列定义. 四阶以上行列式不再适用于对角线法则，也是同样的原因. 特别地，一阶行列式就是元素本身.

定理：从 n 阶行列式的第 $i_1,i_2,\cdots,i_n$ 行和第 $j_1,j_2,\cdots,j_n$ 列取出元素做乘积

$$a_{i_1j_1}a_{i_2j_2}\cdots a_{i_nj_n}$$

这里 $i_1,i_2,\cdots,i_n$ 和 $j_1,j_2,\cdots,j_n$ 都是 $1,2,\cdots,n$ 这 n 个数的排列. 那么这一项在行列式中的符号是 $(-1)^{s+t}$，其中

$$s=\pi(i_1i_2\cdots i_n)$$

$$t=\pi(j_1j_2\cdots j_n)$$

定理：行列式和它的转置行列式相等.

定理：设行列式 $\det A$ 的第 i 行的所有元素都可以表示成两项的和：

$$\begin{vmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ \vdots & \vdots & & \vdots \\ b_{i1}+c_{i1} & b_{i2}+c_{i2} & \cdots & b_{in}+c_{in} \\ \vdots & \vdots & & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{vmatrix}$$

那么该行列式等于两个行列式 $\det A_1$ 和 $\det A_2$ 的和．其中 A_1 的第 i 行是 $b_{i1}, b_{i2}, \cdots, b_{in}$，$A_2$ 的第 i 行是 $c_{i1}, c_{i2}, \cdots, c_{in}$，$A_1$ 和 A_2 的其余各行都和 A 相同．同样的性质对于列来说也成立.

归纳方法定义

这种方法只是描述了行列式的一种代数性质，时间复杂度也为阶乘量级，不适合用于计算.

代数余子式

在 n 阶行列式 $\det A$ 中，任意取定矩阵 A 的 k 行和 k 列．位于这些行列相交处的元素构成的 k 阶矩阵叫做 k 阶子矩阵，其行列式称为 k 阶子式.

对于 n 阶行列式 $\det A$，某一元素 a_{ij} 的余子阵 M_{ij} 指的是原矩阵 A 中，划去 a_{ij} 所在的行和列后，余下的 $n-1$ 阶子矩阵；其行列式 $\det M_{ij}$ 称为余子式.

对于 n 阶行列式 $\det A$，元素 a_{ij} 的余子式 $\det M_{ij}$ 附以符号 $(-1)^{i+j}$ 之后，叫做元素 a_{ij} 的代数余子式，用符号 A_{ij} 表示.

从上一节全排列方法的定义可以推出结论：

定理：若在一个 n 阶行列式 $\det A$ 中，第 i 行或第 j 列的元素除了 a_{ij} 都是 0，那么这个行列式等于 a_{ij} 和它的代数余子式 A_{ij} 的乘积.

行列式展开

由于方阵转置，行列式不变，只需介绍按行展开或按列展开之一即可.

行列式 $\det A$ 定义为它任意一行（或一列）的所有元素与它们的对应代数余子式乘积的和.

换句话说，行列式可以使用按行（或按列）的展开式递归定义：

$$\begin{aligned} \det A &= a_{i1}A_{i1} + a_{i2}A_{i2} + \cdots + a_{in}A_{in} \\ &= \sum_{j=1}^{n} a_{ij}A_{ij} \\ &= \sum_{j=1}^{n} (-1)^{i+j} a_{ij} \det M_{ij} \\ \det A &= a_{1j}A_{1j} + a_{2j}A_{2j} + \cdots + a_{nj}A_{nj} \\ &= \sum_{i=1}^{n} a_{ij}A_{ij} \\ &= \sum_{i=1}^{n} (-1)^{i+j} a_{ij} \det M_{ij} \end{aligned}$$

递归终点为一阶矩阵的行列式，其即为该矩阵包含的唯一一个元素.

于是有结论：

定理：行列式 $\det A$ 的某一行（或某一列）的元素与另外一行（或另外一列）对应元素的代数余子式的乘积之和等于 0.

换句话说，当 $i \neq j$ 时：

$$a_{i1}A_{j1}+a_{i2}A_{j2}+\cdots+a_{in}A_{jn}=0$$
$$a_{1i}A_{1j}+a_{2i}A_{2j}+\cdots+a_{ni}A_{nj}=0$$

公理化定义

公理化定义是说，满足了某些性质的运算只能是行列式.

前置知识：初等变换→9.14.5@p1725.

记 $D_i(k)$ 为 倍乘矩阵→9.14.5@p1725、P_{ij} 为 对换矩阵→9.14.5@p1725、$T_{ij}(k)$ 为 倍加矩阵→9.14.5@p1725.

对于一个 n 阶矩阵 A 的运算 det，如果满足以下四个性质，称为行列式：

- 把一个行列式的某一行或某一列的所有元素同时乘以一个数 k，等于用 k 乘这个行列式.
  $\det(D_i(k)A)=\det(AD_i(k))=k\det A$
- 交换一个行列式的两行或两列，行列式改变符号.
  $\det(P_{ij}A)=\det(AP_{ij})=-\det A$
- 把行列式的某一行或某一列的元素乘以同一数后加到另一行或另一列的对应元素上，行列式不变.
  $\det(T_{ij}(k)A)=\det(AT_{ij}(k))=\det A$
- 单位矩阵的行列式为 1.
  $\det I=1$

利用行列式有关初等变换的性质，可以方便手动计算更高阶的行列式. 「高斯消元」法计算行列式→9.19.3@p1944，也用到了这个性质，时间复杂度为 $O(n^3)$.

上述性质也有若干推论：

- 一个行列式中某一行或某一列的公因子可以提到行列式符号的外边.
- 如果一个行列式的某一行或某一列的元素全部是 0，那么这个行列式等于 0.
- 如果一个行列式有两行或两列的对应元素成比例，那么这个行列式等于 0.
- 如果一个行列式有两行或两列完全相同，那么这个行列式等于 0.

这些推论在手算行列式的时候非常常用.

9.14.7 线性空间

Authors: codewasp942, Tiphereth-A

线性空间是 d 维欧氏空间（$0\le d\le 3$）等的推广，相关概念的关系可参照 欧氏空间与线性空间的关系→9.14@p1706.

前置知识：阿贝尔群、域.

通俗地讲，一个集合关于某运算封闭，满足结合律、单位元与逆元则构成群. 如果还满足交换律，则构成阿贝尔群.

如果一个集合关于四则运算封闭，则构成域. 相关定义详见 抽象代数基本概念→9.16.1@p1805.

定义

线性空间（向量空间）是线性代数的基本概念与重要研究对象. 线性空间是由向量集合 V、域 $\mathbb{P}$、加法运算 $+$ 和标量乘法（数乘）组成的模类代数结构.

具体来说，设 $(V,+)$ 是一个阿贝尔群，$\mathbb{P}$ 是一个域.

定义 $\mathbb{P}$ 中的数与 V 中元素的一种代数运算，称为 **数乘**：$\cdot:\mathbb{P}\times V\mapsto V$，记为 $p\cdot v$ 或 pv，其中 p 在域 $\mathbb{P}$ 中，v 在阿贝尔群 V 中. 要求该数乘运算是封闭的，运算结果始终有意义，也在群 V 中.

且满足以下条件：

1. **数乘对向量加法分配律**：对于 $\mathbf{u},\mathbf{v}\in V,a\in\mathbb{P}$，$a(\mathbf{u}+\mathbf{v})=a\mathbf{u}+a\mathbf{v}$

2. **数乘对标量加法分配律**：对于 $a,b\in\mathbb{P},\mathbf{u}\in V$，$(a+b)\mathbf{u}=a\mathbf{u}+b\mathbf{u}$
3. **数乘结合律（一致于域乘法）**：对于 $a,b\in\mathbb{P},\mathbf{u}\in V$，$a(b\mathbf{u})=(ab)\mathbf{u}$
4. **标量乘法单位元**：令 $1\in\mathbb{P}$ 是 $\mathbb{P}$ 的乘法单位元，则对于 $u\in V$，$1\mathbf{u}=\mathbf{u}$

则称代数系统 $(V,+,\cdot,\mathbb{P})$ 是 V 关于 $+,\cdot$ 构成 $\mathbb{P}$ 上的一个 **线性空间**，$\mathbb{P}$ 为线性空间的 **基域**，V 中元素称为 **向量**，$\mathbb{P}$ 中元素称为 **标量**．当域 $\mathbb{P}$ 为实数域时，称为实线性空间．当域 $\mathbb{P}$ 为复数域时，称为复线性空间．

不管是一列数还是箭头，或是别的什么东西，只要满足上述公理，都可以认为是向量，也就都可以利用线性代数的理论来研究．

称加法群中的零元为零向量，记作 $\mathbf{0}$ 或 $\boldsymbol{\theta}$．

原阿贝尔群中向量的加减法，与线性空间新定义的数乘，统称为 **线性运算**．

> **Note**
>
> 为行文方便，下文中：
> 1. 对 V 中的元素不做加粗处理．
> 2. 将满足线性空间定义的代数系统 $(V,+,\cdot,\mathbb{P})$ 也称为线性空间．
>
> 请注意区分．

直观理解

不是很严谨地说，标量乘法对应着一种「**缩放**」，基域 $\mathbb{P}$ 中的元素就代表着缩放的「**比例**」，向量加法对应「**叠加**」．同时，$\mathbb{P}$ 中的元素还代表着向量的「**坐标**」的取值范围．

条件 1-4 描述的是「缩放」与「叠加」的关联．可以结合二维平面上的箭头来理解．

简单性质

> **Note**
>
> 以下性质可在群论等中找到．

对线性空间 $(V,+,\cdot,\mathbb{P})$，

1. θ 唯一
2. $\forall\alpha\in V$,$-\alpha$ 唯一
3. $\exists 0\in\mathbb{P}$,$\forall\alpha\in V$, 有 $0\alpha=\theta$
4. $\forall k\in\mathbb{P}$, 有 $k\theta=\theta$
5. $(-1)\alpha=-\alpha,\ \forall\alpha\in V$
6. 无零因子：$\forall\alpha\in V,k\in\mathbb{P}$, 有 $k\alpha=\theta\Longrightarrow k=0\vee\alpha=\theta$
7. 加法的消去律：$\forall\alpha,\beta,\gamma\in V$, 有 $\alpha+\beta=\alpha+\gamma\Longrightarrow\beta=\gamma$

> 实际上，加法的消去律是阿贝尔群的性质．

例子

1. $\mathbb{P}^n$ 关于数域 $\mathbb{P}$ 上的加法和乘法构成 $\mathbb{P}$ 上的一个线性空间．例如 $\mathbb{P}$ 可以是 $\mathbb{R}$,$\mathbb{C}$,$\mathbb{N}_p$（p 为素数）等．
2. 数域 $\mathbb{P}$ 上的 $n\times m$ 阶矩阵 $\mathbb{P}^{n\times m}$ 关于矩阵的加法和数乘构成 $\mathbb{P}$ 上的一个线性空间．
3. 数域 $\mathbb{P}$ 上的一元多项式环 $\mathbb{P}[x]$ 关于多项式的加法和数乘构成 $\mathbb{P}$ 上的一个线性空间．
4. 区间 $[a,b]$ 上的全体连续函数（记作 $C[a,b]$）关于「函数加法」和「值与连续函数的数乘」构成值域上的一个线性空间．

相关概念

线性相关、线性无关

对线性空间 $(V,+,\cdot,\mathbb{P})$：

1. 称 $a_1,a_2,...,a_n\in V$ 为 V 的一个 **向量组**.
2. 对于 $k_1,k_2,...,k_n\in\mathbb{P}$, 称 $\sum_{i=1}^n k_ia_i$ 为向量组 $a_1,a_2,...,a_n$ 的一个 **线性组合**.
3. 若向量 $\beta\in V$ 可以表示为向量组 $a_1,a_2,...,a_n$ 的一个线性组合，则称 β 能被向量组 $a_1,a_2,...,a_n$ **线性表出**.
4. 对于 $k_1,k_2,...,k_n\in\mathbb{P}$, 若向量组 $a_1,a_2,...,a_n$ 满足 $\sum_{i=1}^n k_ia_i=\theta\iff k_i=0,i=1,2,...,n$, 则称向量组 $a_1,a_2,...,a_n$ **线性无关**，否则称向量组 $a_1,a_2,...,a_n$ **线性相关**.

规定零向量与任意向量线性相关.

线性表示或线性相关的式子，可以写成矩阵乘法的形式：

$$\beta=k_1a_1+k_2a_2+\cdots+k_ra_r=(a_1,a_2,\cdots,a_r)\begin{pmatrix}k_1\\k_2\\\vdots\\k_r\end{pmatrix}$$

根据习惯，把向量 a 按顺序并排写在左边；把标量 k 按顺序竖着写在右边，构成一个「列向量」.

注意：这里标量构成的「列向量」只是方便的形式记号，不在空间 V 中，与左边的向量有着本质的区别. 左边的向量如果恰好是列向量，并排拼起来就可以形式上构成一个「矩阵」，上述乘积恰好是矩阵中常见的「矩阵左乘列向量」的形式.

下文指出，这里的线性表示也等价于，向量 β 落在矩阵 $(a_1,a_2\cdots,a_r)$ 的像空间里.

根据下文中的定义，零向量一定会落在像空间里. 如果用线性变换的观点看，线性相关等价于变换后多个向量变换到零向量，而线性无关等价于只有零向量本身变换到零向量.

性质

对线性空间 $(V,+,\cdot,\mathbb{P})$,

1. 若向量组的一部分线性相关，则向量组线性相关. 若向量组线性无关，则其任意非空部分均线性无关. 简记为：**「大无关、小无关」；「小相关、大相关」**.
2. 含 θ 的向量组线性相关.
3. 向量组线性相关当且仅当向量组的某个向量可以由其余向量线性表出.
4. 若向量 β 可被向量组 $a_1,a_2,...,a_n$ 线性表出，则表出方式唯一当且仅当向量组 $a_1,a_2,...,a_n$ 线性无关.
5. 若向量组 $a_1,a_2,...,a_n$ 线性无关，则向量 β 可被向量组 $a_1,a_2,...,a_n$ 线性表出当且仅当向量组 $a_1,a_2,...,a_n,\beta$ 线性相关.

极大线性无关组、秩

线性相关可以理解为「多余」，说明向量组内部有的向量可以被其他向量表出，可以删去. 删完了之后，将剩下极大线性无关组.

对线性空间 $(V,+,\cdot,\mathbb{P})$：

1. 对于向量组 $b_1,b_2,...,b_m$, 令 $\{a_1,a_2,...,a_n\}\subseteq\{b_1,b_2,...,b_m\}$, 若有：
   - 向量组 $a_1,a_2,...,a_n$ 线性无关.
   - $\forall\beta\in\{b_1,b_2,...,b_m\}\setminus\{a_1,a_2,...,a_n\}$, 向量组 $a_1,a_2,...,a_n,\beta$ 线性相关.

   则称向量组 $a_1,a_2,...,a_n$ 为向量组 $b_1,b_2,...,b_m$ 中的一个 **极大线性无关组**. 类似地，可定义线性空间 V 的极大线性无关组.

   规定向量组 $\theta,\theta,...,\theta$ 的极大线性无关组为空集，于是全 0 矩阵对应的向量组没有极大线性无关组.

从向量组删向量的删法不唯一，因此极大线性无关组也不唯一．习惯上从左到右按顺序删．

很巧的是，按顺序删，留下的向量，恰好就是「按行看」观点里面，高斯消元法剩下的行最简形矩阵中，元素 1 所在的列．

称向量组 $b_1, b_2, ..., b_m$ 的极大线性无关组的大小为向量组的 **秩**，记作 $\text{rank}\{b_1, b_2, ..., b_m\}$，规定 $\text{rank}\{\theta, \theta, ..., \theta\} = 0$．

于是，向量组的秩的定义与矩阵的秩的定义完全一致．

2. 若向量组 $a_1, a_2, ..., a_n$ 能线性表出向量组 $b_1, b_2, ..., b_m$ 中的所有向量，称向量组 $b_1, b_2, ..., b_m$ 能被向量组 $a_1, a_2, ..., a_n$ 线性表出．
3. 若向量组 $a_1, a_2, ..., a_n$ 能被向量组 $b_1, b_2, ..., b_m$ 线性表出，且向量组 $b_1, b_2, ..., b_m$ 能被向量组 $a_1, a_2, ..., a_n$ 线性表出，则称两向量组 **等价**，记作 $\{a_1, a_2, ..., a_n\} \cong \{b_1, b_2, ..., b_m\}$．

   向量组的 **等价** 就是向量组张成的空间相同．张成空间相同的向量组相互等价，张成空间不同的向量组不等价．

   向量组等价比矩阵等价条件更强，不仅要求秩相同，还要求空间完全一样．因此，把两个矩阵 **横向** 拼在一起，秩不能发生变化．

   矩阵等价仅要求秩相同，因此矩阵等价表示前一个矩阵或空间，可以通过可逆变换，到达后一个矩阵或空间．

性质

对线性空间 $(V, +, \cdot, \mathbb{P})$，

1. 设向量组 $a_1, a_2, ..., a_n$ 能被线性表出向量组 $b_1, b_2, ..., b_m$ 线性表出．
   - 若 $n > m$，则向量组 $a_1, a_2, ..., a_n$ 线性相关．
   - 若向量组 $a_1, a_2, ..., a_n$ 线性无关，则 $n \leq m$．
2. 等价的线性无关向量组的大小相等．

   向量组的任意极大线性无关组的大小均相等．
3. 向量组线性无关当且仅当其秩等于其大小．
4. 若向量组 $a_1, a_2, ..., a_n$ 能被线性表出向量组 $b_1, b_2, ..., b_m$ 线性表出，则 $\text{rank}\{a_1, a_2, ..., a_n\} \leq \text{rank}\{b_1, b_2, ..., b_m\}$．
5. 等价的向量组的秩相等．

线性包

对于线性空间 $(V, +, \cdot, \mathbb{P})$，$\left\{v = \sum_{i=1}^{n} k_i a_i : a_i \in V, k_i \in \mathbb{P}, i = 1, 2, ..., n\right\}$ 也构成一个线性空间，称为由向量组 $a_1, a_2, ..., a_n$ **张成** 的线性空间（或 **线性包**），记作 $\text{span}\{a_1, a_2, ..., a_n\}$．

这里的 n 个向量 a 不一定线性无关．

线性子空间

对线性空间 $(V, +, \cdot, \mathbb{P})$，若代数系统 $(V_1, +, \cdot, \mathbb{P})$ 满足：

1. $\emptyset \neq V_1$
2. $V_1 \subseteq V$
3. V_1 关于 $+, \cdot$ 构成 $\mathbb{P}$ 上的线性空间

则称 V_1 为 V 的线性子空间，简称子空间，记作 $V_1 \leq V$．

任何空间 V 都有两个 **平凡子空间**：它本身 V 与零子空间．零子空间只含零向量，不含有线性无关的向量．

若第 2 条中的 $\subseteq$ 换为 $\subset$，则称 V_1 为 V 的线性真子空间，记作 $V_1 < V$．

不难证明：线性空间 V 的非空子集 V_1 是其线性子空间当且仅当线性运算在 V_1 上封闭，即：

1. $\forall u,v\in V_1$,$u+v\in V_1$
2. $\forall v\in V_1$,$\forall k\in\mathbb{P}$,$kv\in V_1$

交、和与直和、直积

对线性空间 $(V_1,+,\cdot,\mathbb{P})$ 与 $(V_2,+,\cdot,\mathbb{P})$：

1. 不难验证：加法和数乘在 $V_1\cap V_2$ 上封闭，故可称 $V_1\cap V_2$ 为线性空间 V_1 和 V_2 的 **交**.

   类似地，可定义多个线性空间的交 $\bigcap_{i=1}^m V_i$.
2. 若线性空间 V 满足 $V=\{u+v\mid u\in V_1,v\in V_2\}$，则称 V 为线性空间 V_1 和 V_2 的 **和**，记为 $V=V_1+V_2$.

   可以验证：V_1+V_2 是包含 $V_1\cup V_2$ 的最小子空间.

   类似地，可定义多个线性空间的和 $\sum_{i=1}^m V_i$.
3. 设 $V=V_1+V_2$，若线性空间 V 中的任意元素 v，均只能找到唯一一组向量 v_1,v_2 满足 $v=v_1+v_2$，则称 V 为线性空间 V_1 和 V_2 的 **直和**(direct sum)，记为 $V_1\oplus V_2$.

   类似地，可定义多个线性空间的直和 $\bigoplus_{i=1}^m V_i$.
4. V_1 与 V_2 的 **直积** $V_1\times V_2$ 定义为二者的笛卡儿积关于如下的加法和数乘构成 $\mathbb{P}$ 上的线性空间：
   1. $+:(V_1\times V_2)\times(V_1\times V_2)\mapsto V_1\times V_2;((u_1,v_1),(u_2,v_2))\to(u_1+u_2,v_1+v_2)$
   2. $\cdot:\mathbb{P}\times(V_1\times V_2)\mapsto V_1\times V_2;(k,(u,v))\to(ku,kv)$

   类似地，可定义多个线性空间的直积 $\prod_{i=1}^m V_i$.

例子

对于线性空间 $V=\mathbb{R}^3$，设线性空间：

- $V_1:=\{(x,0,0)|\ x\in\mathbb{R}\}$
- $V_2:=\{(x,y,0)|\ x,y\in\mathbb{R}\}$
- $V_3:=\{(0,y,z)|\ y,z\in\mathbb{R}\}$
- $V_4:=\{(x,0,z)|\ x,z\in\mathbb{R}\}$

则

1. $V_1<V_2<V$,$V_3<V$
2. $V_2=V_1+V_2$
3. $V=V_1\oplus V_3=V_2+V_3$
4. $V_2+V_3\leq V$

性质

1. 令 V_1,V_2,V_3 是关于 $\mathbb{P}$ 的线性空间，和集合的交一样，线性空间的交适用如下法则：
   1. 交换律：$V_1\cap V_2=V_2\cap V_1$
   2. 结合律：$V_1\cap(V_2\cap V_3)=(V_1\cap V_2)\cap V_3$
2. 令 V_1,V_2,V_3 是关于 $\mathbb{P}$ 的线性空间，类似于集合的并，线性空间的和适用如下法则：
   1. 交换律：$V_1+V_2=V_2+V_1$
   2. 结合律：$V_1+(V_2+V_3)=(V_1+V_2)+V_3$
3. 令 V_1,V_2,V_3 是关于 $\mathbb{P}$ 的线性空间，线性空间的交与并有如下关系：
   1. $V_1\cap(V_2+V_3)\supseteq(V_1\cap V_2)+(V_1\cap V_3)$
   2. $V_1+(V_2\cap V_3)\subseteq(V_1+V_2)\cap(V_1+V_3)$
4. $\mathrm{span}\{a_1,a_2,...,a_n\}+\mathrm{span}\{b_1,b_2,...,b_m\}=\mathrm{span}\{a_1,a_2,...,a_n,b_1,b_2,...,b_m\}$
5. 令 V_1,V_2 是关于 $\mathbb{P}$ 的线性空间，则下列诸款等价：
   1. $V_1+V_2=V_1\oplus V_2$
   2. $\exists\beta\in V_1+V_2$，使得拆分为 V_1 和 V_2 中的向量和的方式唯一（任意 $\to$ 存在）
   3. θ 拆分为 V_1 和 V_2 中向量的和的方式唯一
   4. $V_1\cap V_2=\{\theta\}$

证明

$1 \Longrightarrow 2$：由定义立得.

$2 \Longrightarrow 3$：

令 $\beta = \beta_1 + \beta_2$, 其中 $\beta_1 \in V_1, \beta_2 \in V_2$, 若 $\theta = \alpha_1 + \alpha_2, \theta \neq \alpha_1 \in V_1, \alpha_2 \in V_2$, 则 $\beta = \beta + \theta = (\beta_1 + \alpha_1) + (\beta_2 + \alpha_2)$.

而 $\beta_1 \neq \beta_1 + \alpha_1$, 与条件矛盾.

$3 \Longrightarrow 4$：

在 V_1 和 V_2 中取一非零向量 α, 则 $\theta = \alpha + (-\alpha) = (-\alpha) + \alpha$, 这与条件矛盾.

$4 \Longrightarrow 1$：

若 $V_1 + V_2$ 不是直和，则存在 $\beta \in V_1 + V_2$ 使得 $\beta = \beta_1 + \beta_2 = \gamma_1 + \gamma_2$, 其中 $\beta_1, \gamma_1 \in V_1, \beta_2, \gamma_2 \in V_2$ 且 $\beta_1, \beta_2, \gamma_1, \gamma_2$ 互不相同.

进而 $\theta \neq \beta_1 - \gamma_1 = \gamma_2 - \beta_2 \in V_1 \cap V_2$, 与条件矛盾.

同构

设 V, V' 均为域 $\mathbb{P}$ 上的线性空间，若存在双射 $\sigma : V \mapsto V'$ 且保持加法与数乘，即 $\forall u, v \in V, \forall k \in \mathbb{P}$ 满足：

1. $\sigma(u + v) = \sigma(u) + \sigma(v)$
2. $\sigma(ku) = k\sigma(u)$

则称 σ 是 V 到 V' 的 **同构映射**，此时称 V 与 V' **同构**，记为 $V \cong V'$.

Note

若 σ 是单射，则可定义 **单同态**；若 σ 是满射，则可定义 **满同态**.

性质

1. 域 $\mathbb{P}$ 上的两线性空间同构当且仅当其维数相等.（维数的定义参见 线性基→9.14.8@p1738.）
2. （1 的推论）域 $\mathbb{P}$ 上的 n 维线性空间与线性空间 $\mathbb{P}^n$ 同构.

Note

本性质说明我们基本上可以将坐标和向量等同看待.

欧氏空间与线性空间的关系

以我们最熟悉的三维欧氏空间为例，其部分相关概念在线性空间中的对应关系如下表：

三维欧氏空间	线性空间
向量	向量
垂直	正交（即内积为 0）
三向量共线/共面	k 个向量线性相关
三向量不共面	k 个向量线性无关
基向量	线性基→9.14.8@p1738
空间的维数	空间的维数

应用

从本节开始主要讲述对于线性方程组「按列看」的观点.

矩阵 A 本身也是由列向量构成的. 把 A 本身看成了列向量组，而 x 是未知数系数，思考 A 当中的这组列向量能不能配上未知数，凑出列向量 b. 此时列向量 x 是完全未知的.

此时研究的等式 $Ax=b$ 整理为:

$$\alpha_1x_1+\alpha_2x_2+\cdots+\alpha_nx_n=b$$

这时，矩阵乘法中，位于左边的矩阵 A 可以看作向量组，即一组列向量. 这组列向量作为一组基，张成一个空间，探讨列向量 b 是否落在这个空间里.

按列看待线性方程组的解

秩是极大线性无关组中向量的个数，代表了「约束」. 那么其余的向量将赋予解的自由度，即允许在其他方向赋予冗余的向量.

如果记 n 是矩阵 A 的列数，即含有的列向量个数，记 $r(A)$ 为矩阵 A 的秩，则有自由度 S:

$$S=n-r(A)$$

方程组的全体解也构成向量组，自由度 S 就是 $Ax=0$ 解向量组的秩，即下文核空间的维数.

方程组的同解

两个方程组的公共解定义为两组解的交集.

方程组的 **同解** 就是方程组的解的集合相等. 解的集合相等的方程组同解，解的集合不相等的方程组不同解.

方程组同解也比矩阵等价条件强，不仅要求秩相等，还要求把两个矩阵 **纵向** 拼在一起之后，秩仍然不改变.

这里与向量组等价对比，向量组等价要求矩阵横向拼接，秩不改变. 因此，有如下关系:

矩阵等价，不一定有对应的向量组等价或者方程组同解，但是若有向量组等价或者方程组同解，必然有对应的矩阵等价（秩相同）.

如果矩阵对应的向量组等价，那么将矩阵转置后，对应的方程组同解，反之亦然.

矩阵的核空间与像空间

这部分的核空间与像空间是站在线性空间的角度上叙述的.

对于矩阵 A，令 W 为方程 $Ax=0$ 的全体解 x 构成的集合，则 W 是一个线性空间，W 的标量域与 A 的元素所在的域相同.

称此时的 W 为矩阵 A 的 **核空间**，记作 $N(A)$.

矩阵 A 的核空间 $N(A)$ 就是方程 $Ax=0$ 的 **解空间**. 根据后文基的定义，该方程的 **基础解系** 就是核空间的基.

如果矩阵 A 是可逆矩阵，则 A 的核空间 $N(A)$ 只含零向量.

对于矩阵 A，它的 n 个列为向量 α，称 n 个列向量 α 张成的空间为 A 的 **像空间**，或者记作 **列空间**，记作:

$$R(A)=\text{span}\{\alpha_1,\alpha_2,\cdots,\alpha_n\}$$

根据后文维数的定义，像空间的维数等于矩阵 A 的秩.

由定义，对于像空间 $R(A)$ 中的每一个元素 y，均有相应的表示:

$$y = k_1\alpha_1 + k_2\alpha_2 + \cdots + k_n\alpha_n = (\alpha_1, \alpha_2, \cdots, \alpha_n)\begin{pmatrix} k_1 \\ k_2 \\ \vdots \\ k_n \end{pmatrix} = A\begin{pmatrix} k_1 \\ k_2 \\ \vdots \\ k_n \end{pmatrix}$$

因此像空间 $R(A)$ 就是对于任意向量 x，Ax 的 **值域**.

同理可以定义 A 的 **行空间**，即 A 的转置的值域 $R(A^T)$.

由于矩阵的行秩等于列秩，行空间的维数也为矩阵的秩，因此转置改变像空间，而不改变像空间的维数.

在这里可以与前文建立对应关系：

向量组等价，等价于对应矩阵的像空间 $R(A)$ 相同.

方程组同解，等价于对应矩阵的行空间 $R(A^T)$ 相同.

参考资料与注释

1. 丘维声，高等代数（下）. 清华大学出版社.
2. Vector space[1]. *Wikipedia, The Free Encyclopedia*.

外部参考资料

[1] Vector space 

9.14.8 线性基

> **Authors:** cesonic, Enter-tainer, Great-designer, Ir1d, ksyx, lychees, MegaOwIer, RUIN-RISE, wjy-yy, rsdbkhusky, ouuan, Menci, Tiphereth-A

回想高中数学立体几何中基向量的概念，我们可以在三维欧氏空间中找到一组基向量 $\boldsymbol{i}$，$\boldsymbol{j}$，$\boldsymbol{k}$，之后空间中任意一个向量都可以由这组基向量表示. 换句话说，我们可以 **通过有限的基向量来描述无限的三维空间**，这足以体现基向量的重要性.

三维欧氏空间是特殊的 线性空间→9.14.7@p1731，三维欧氏空间的基向量在线性空间中就被推广为了线性基.

OI 中有关线性基的应用一般只涉及两类线性空间：n 维实线性空间 $\mathbf{R}^n$ 和 n 维 布尔域[1] 线性空间 $\mathbf{Z}_2^n$，我们会在 应用→9.14@p1706 一节中详细介绍. 若您不熟悉线性代数，则推荐从应用部分开始阅读.

以下会从一般的线性空间出发来介绍线性基，并给出线性基的常见性质.

前置知识：线性空间→9.14.7@p1731.

线性基是线性空间的一组基，是研究线性空间的重要工具.

定义

称线性空间 V 的一个极大线性无关组为 V 的一组 **Hamel 基** 或 **线性基**，简称 **基**.

规定线性空间 $\{\theta\}$ 的基为空集.

可以证明任意线性空间均存在线性基[2]，我们定义线性空间 V 的 **维数** 为线性基的元素个数（或势），记作 $\dim V$.

性质

1. 对于有限维线性空间 V，设其维数为 n，则：

[2] Proof that every vector space has a basis[2]

1. V 中的任意 $n+1$ 个向量线性相关.
2. V 中的任意 n 个线性无关的向量均为 V 的基.
3. 若 V 中的任意向量均可被向量组 $a_1, a_2, ..., a_n$ 线性表出，则其是 V 的一个基.

   证明

   任取 V 中的一组基 $b_1, b_2, ..., b_n$, 由已知条件, 向量组 $b_1, b_2, ..., b_n$ 可被 $a_1, a_2, ..., a_n$ 线性表出，故

   $n = \text{rank}\{b_1, b_2, ..., b_n\} \leq \text{rank}\{a_1, a_2, ..., a_n\} \leq n$

   因此 $\text{rank}\{a_1, a_2, ..., a_n\} = n$

4. V 中任意线性无关向量组 $a_1, a_2, ..., a_m$ 均可通过插入一些向量使得其变为 V 的一个基.

2. （子空间维数公式）令 V_1, V_2 是关于 $\mathbb{P}$ 的有限维线性空间，且 $V_1 + V_2$ 和 $V_1 \cap V_2$ 也是有限维的，则 $\dim V_1 + \dim V_2 = \dim(V_1 + V_2) + \dim(V_1 \cap V_2)$

   证明

   设 $\dim V_1 = n_1$,$\dim V_2 = n_2$,$\dim(V_1 \cap V_2) = m$.

   取 $V_1 \cap V_2$ 的一组基 $a_1, a_2, ..., a_m$，将其分别扩充为 V_1 和 V_2 中的基：$a_1, a_2, ..., a_m, b_1, b_2, ..., b_{n_1-m}$ 和 $a_1, a_2, ..., a_m, c_1, c_2, ..., c_{n_2-m}$.

   接下来只需证明向量组 $a_1, a_2, ..., a_m, b_1, b_2, ..., b_{n_1-m}, c_1, c_2, ..., c_{n_2-m}$ 线性无关即可.

   设 $\sum_{i=1}^{m} r_i a_i + \sum_{i=1}^{n_1-m} s_i b_i + \sum_{i=1}^{n_2-m} t_i c_i = \theta$.

   则 $\sum_{i=1}^{n_2-m} t_i c_i = -\sum_{i=1}^{m} r_i a_i - \sum_{i=1}^{n_1-m} s_i b_i$.

   注意到上式左边在 V_2 中，右边在 V_1 中，故两边均在 $V_1 \cap V_2$ 中，因此 $\sum_{i=1}^{n_2-m} t_i c_i = \sum_{i=1}^{m} k_i a_i$

   故 $t_1 = t_2 = ... = t_{n_2-m} = k_1 = k_2 = ... = k_m = 0$, 进而 $r_1 = r_2 = ... = r_m = s_1 = s_2 = ... = s_{n_1-m} = t_1 = t_2 = ... = t_{n_2-m} = 0$

3. 令 V_1, V_2 是关于 $\mathbb{P}$ 的有限维线性空间，且 $V_1 + V_2$ 和 $V_1 \cap V_2$ 也是有限维的，则下列诸款等价：
   1. $V_1 + V_2 = V_1 \oplus V_2$.
   2. $\dim V_1 + \dim V_2 = \dim(V_1 + V_2)$.
   3. 若 $a_1, a_2, ..., a_n$ 是 V_1 的一组基，$b_1, b_2, ..., b_m$ 是 V_2 的一组基，则 $a_1, a_2, ..., a_n, b_1, b_2, ..., b_m$ 是 $V_1 + V_2$ 的一组基.

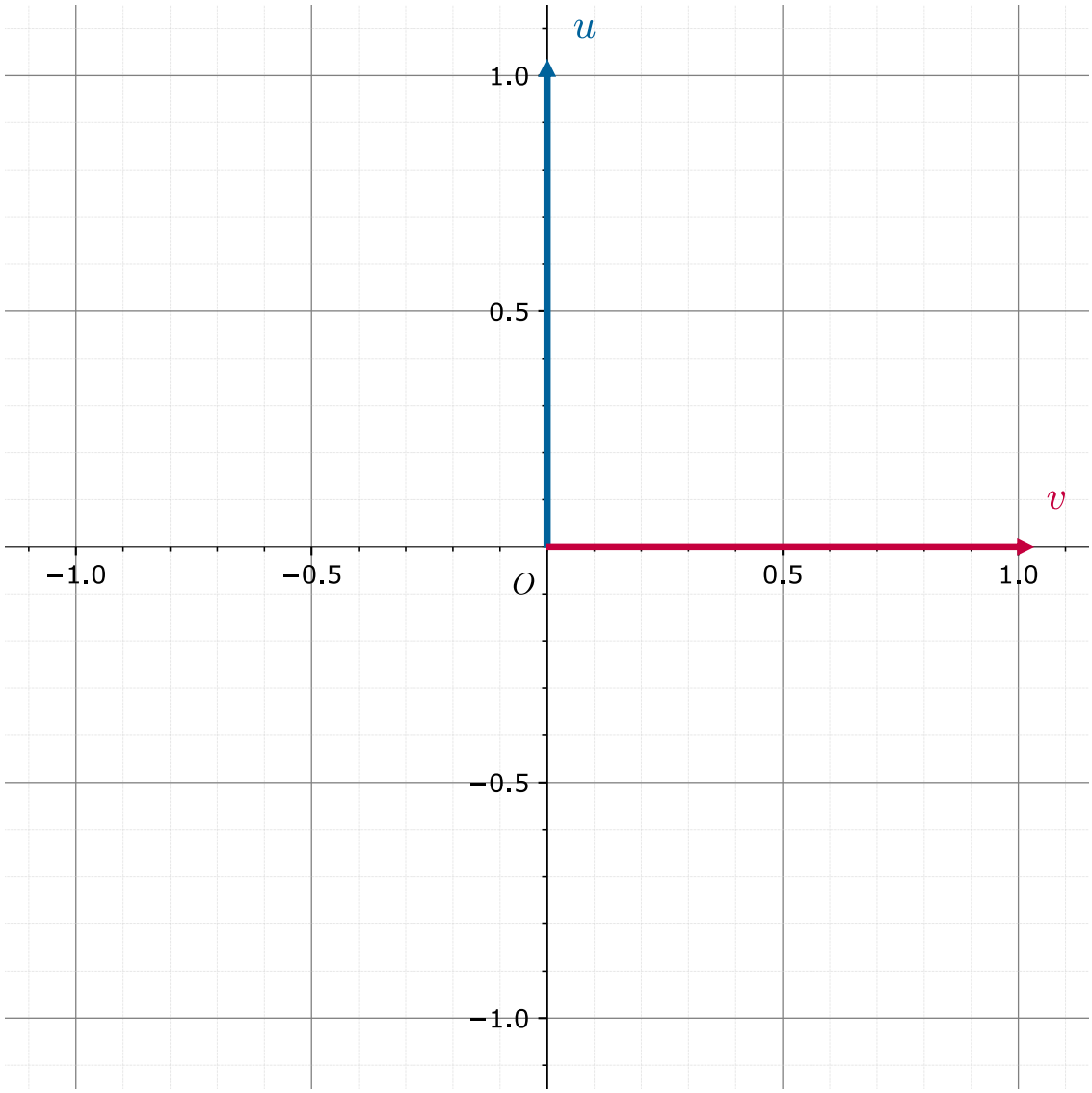
   Note

   1,3 两条可推广到无限维线性空间中

例子

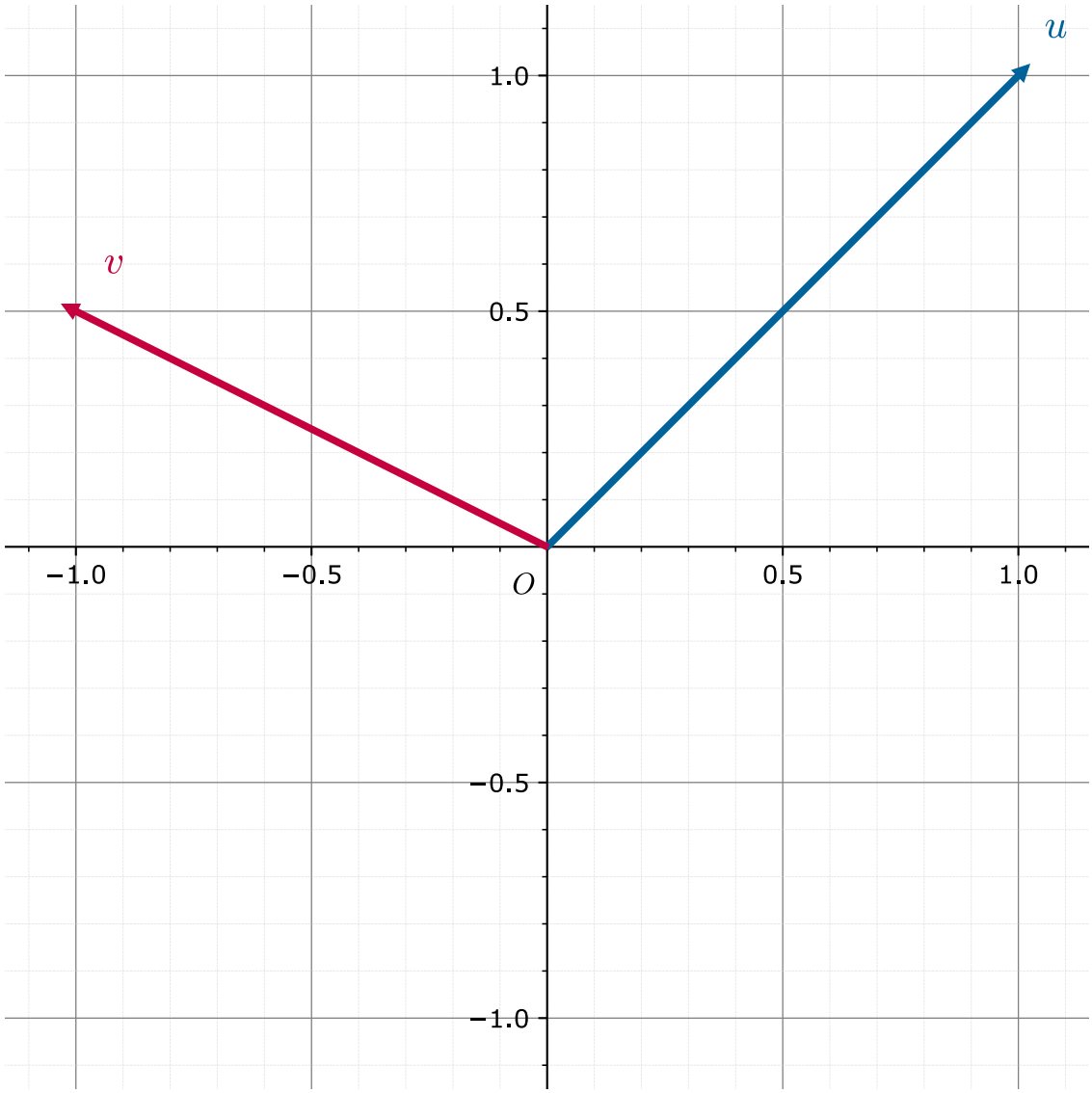
考虑 $\mathbb{R}^2$ 的基.

1. 如图



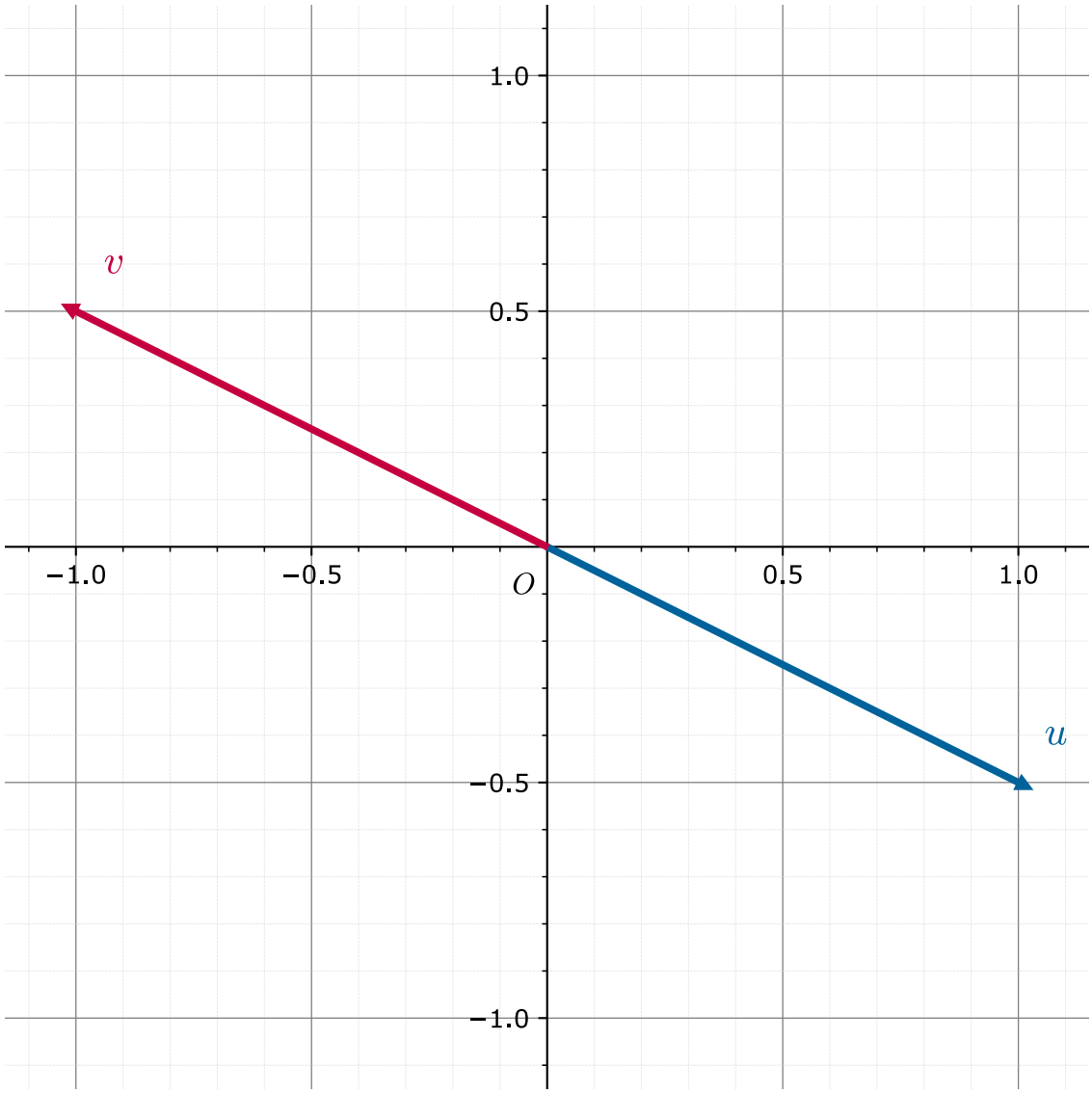
u, v 是一组基.

2. 如图



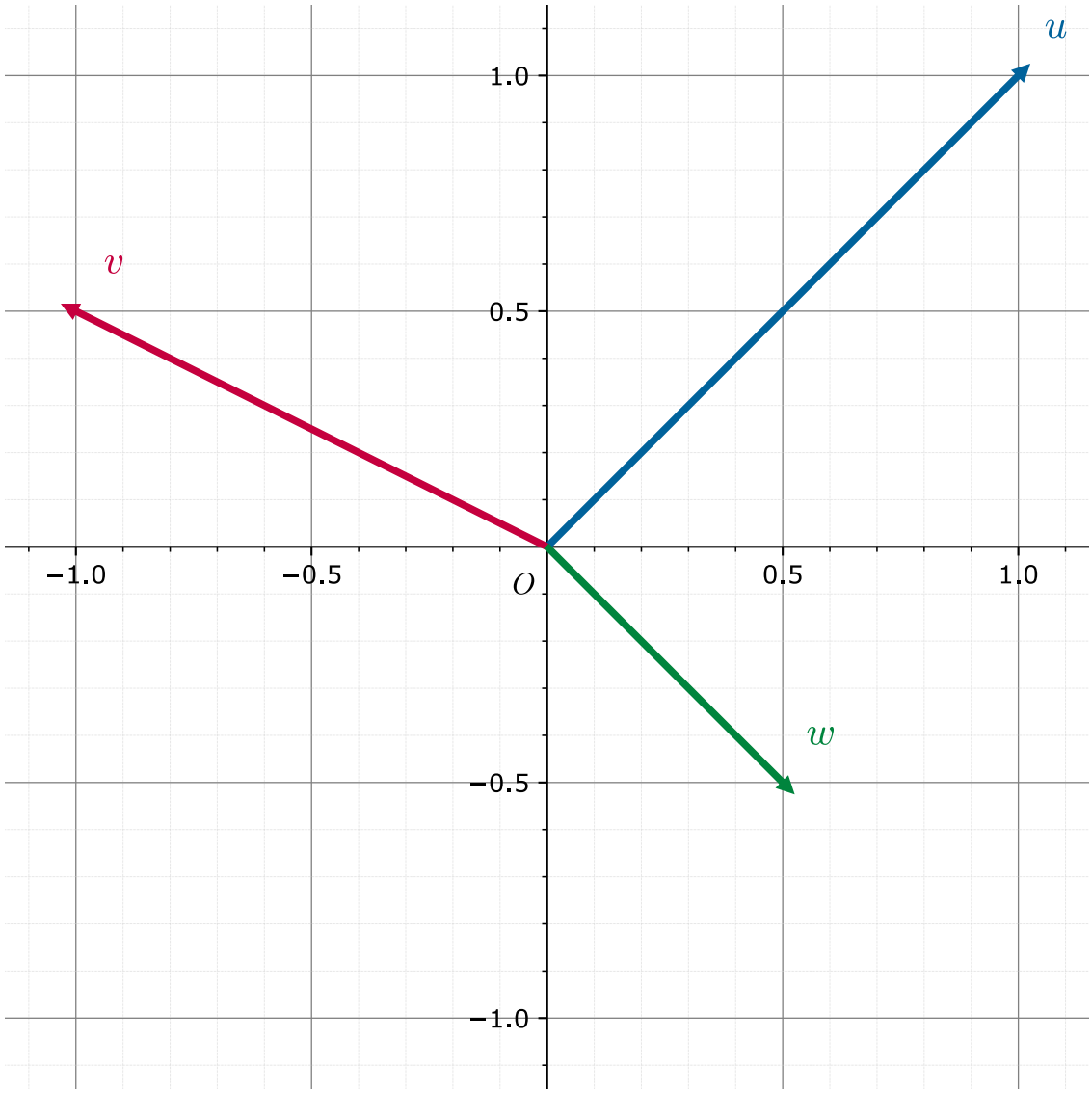
u, v 是一组基.

3. 如图



u, v 不是一组基，因为 $u = -v$.

4. 如图



u, v, w 不是一组基，因为 $u + 4v + 6w = \theta$.

正交基与单位正交基

若线性空间 V 的一组基 B 满足 $\forall b, b' \in B,\ (b, b') \neq 0 \iff b = b'$（即两两正交），则称这组基是 **正交基**.

若线性空间 V 的一组正交基 B 还满足 $\forall b \in B,\ |\ b\ | = \sqrt{(b, b)} = 1$，则称这组基是 **单位正交基**.

任意有限维线性空间 V 的基都可以通过 Schmidt 正交化[3] 变换为正交基.

应用

根据前文内容，我们可以利用线性基实现：

1. 求给定向量组的秩；
2. 对给定的向量组，找到一组极大线性无关组（或其张成的线性空间的一组基）；
3. 向给定的向量组插入某些向量，在插入操作后的向量组中找到一组极大线性无关组（或其张成的线性空间的一组基）；
4. 对找到的一组极大线性无关组（或基），判断某向量能否被其线性表出；
5. 对找到的一组极大线性无关组（或基），求其张成的线性空间中的特殊元素（如最大元、最小元等）.

在 OI 中，我们一般将 n 维实线性空间 $\mathbf{R}^n$ 下的线性基称为 **实数线性基**，n 维布尔域线性空间 $\mathbf{Z}_2^n$ 下的线性基称为 **异或线性基**.

Tip

$\mathbf{Z}_2$ 中的加法为异或，乘法为与，可以证明 $\mathbf{Z}_2$ 是域.

可以证明代数系统 $(\mathbf{Z}_2^n, +, \cdot, \mathbf{Z}_2)$ 是线性空间，其中：

$$(a_1, ..., a_n) + (b_1, ..., b_n) := (a_1 + b_1, ..., a_n + b_n),$$
$$k \cdot (a_1, ..., a_n) := (ka_1, ..., ka_n).$$

即加法是异或，数乘是与.

以异或线性基为例，我们可以根据给定的一组布尔序列 $X = \{x_1, ..., x_m\}$ 构造出一组异或线性基 $B = \{b_1, ..., b_n\}$，这组基有如下性质：

1. B 中任意非空子集的异或和不为 0；
2. 对 X 中的任意元素 x，都可在 B 中取出若干元素使其异或和为 x；
3. 对任意满足上两条的集合 B'，其元素个数不会小于 B 的元素个数.

我们可以利用异或线性基实现：

1. 判断一个数能否表示成某数集子集的异或和；
2. 求一个数表示成某数集子集异或和的方案数；
3. 求某数集子集的最大/最小/第 k 大/第 k 小异或和；
4. 求一个数在某数集子集异或和中的排名.

构造方法

因为异或线性基与实数线性基没有本质差别，所以接下来以异或线性基为例，实数线性基版本的代码只需做一点简单修改即可.

贪心法

对原集合的每个数 p 转为二进制，从高位向低位扫，对于第 x 位是 1 的，如果 a_x 不存在，那么令 $a_x \leftarrow p$ 并结束扫描，如果存在，令 $p \leftarrow p \text{ xor } a_x$.

查询原集合内任意几个元素 xor 的最大值，只需将线性基从高位向低位扫，若 xor 上当前扫到的 a_x 答案变大，就把答案异或上 a_x.

为什么能行呢？因为从高往低位扫，若当前扫到第 i 位，意味着可以保证答案的第 i 位为 1，且后面没有机会改变第 i 位.

查询原集合内任意几个元素 xor 的最小值，就是线性基集合所有元素中最小的那个.

查询某个数是否能被异或出来，类似于插入，如果最后插入的数 p 被异或成了 0，则能被异或出来.

代码（洛谷 P3812 【模板】线性基[4]）

```cpp
#include <algorithm>
#include <iostream>
using ull = unsigned long long;

ull p[64];

void insert(ull x) {
  for (int i = 63; ~i; --i) {
    if (!(x >> i))  // x 的第 i 位是 0
      continue;
    if (!p[i]) {
      p[i] = x;
```

```
      break;
    }
    x ^= p[i];
  }
}

using std::cin;
using std::cout;

int main() {
  int n;
  cin >> n;
  ull a;
  for (int i = 1; i <= n; ++i) {
    cin >> a;
    insert(a);
  }
  ull ans = 0;
  for (int i = 63; ~i; --i) {
    ans = std::max(ans, ans ^ p[i]);
  }
  cout << ans << '\n';
  return 0;
}

```

高斯消元法

高斯消元法相当于从线性方程组的角度去构造线性基，正确性显然.

代码（洛谷 P3812 【模板】线性基[5]）

```
#include <iostream>
using ull = unsigned long long;
constexpr int MAXN = 1e5 + 5;

ull deg(ull num, int deg) { return num & (1ull << deg); }

ull a[MAXN];
using std::cin;
using std::cout;

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  int n;
  cin >> n;
  for (int i = 1; i <= n; ++i) cin >> a[i];
  int row = 1;
  for (int col = 63; ~col && row <= n; --col) {
    for (int i = row; i <= n; ++i) {
      if (deg(a[i], col)) {
        std::swap(a[row], a[i]);
```

```
        break;
      }
    }
    if (!deg(a[row], col)) continue;
    for (int i = 1; i <= n; ++i) {
      if (i == row) continue;
      if (deg(a[i], col)) {
        a[i] ^= a[row];
      }
    }
    ++row;
  }
  ull ans = 0;
  for (int i = 1; i < row; ++i) {
    ans ^= a[i];
  }
  cout << ans << '\n';
  return 0;
}

```

性质

贪心法构造的线性基具有如下性质：

- 线性基没有异或和为 0 的子集.
- 线性基中各数二进制最高位不同.

高斯消元法构造出的线性基满足如下性质：

- 高斯消元后的矩阵是一个行简化阶梯形矩阵.

> 该性质包含了贪心法构造的线性基满足的两条性质

如果不理解这条性质的正确性，可以跳转 高斯消元→9.19.3@p1944.

提供一组样例：

```
5
633 211 169 841 1008
```

二进制表示：

```
1001111001
0011010011
0010101001
1101001001
1111110000
```

贪心法生成的线性基：

```
1001111001
0100110000
0011010011
0001111010
0000000000
0000010000
0000000000
```

```
0000000000
0000000000
0000000000
```

高斯消元法生成的线性基：

```
1000000011
0100100000
0010101001
0001101010
0000010000
0000000000
0000000000
0000000000
0000000000
0000000000
```

这是一条非常好的性质，能帮我们更方便的解决很多问题. 比如：给定一些数，选其中一些异或起来，求异或最大值，如果用贪心法构造线性基，需要再做一遍贪心，如果 `ans` 的当前位是 `0`，那么异或一定会更优，否则当前位如果为 `1`，则一定不会更优；而使用高斯消元法构造线性基后直接将线性基中所有元素都异或起来输出即可.

对于其他比较经典的问题（查询一个数能否被异或得到，查询能被异或得到的第 k 大数等），高斯消元法得到的线性基也能更加方便地解决.

时间复杂度

设向量长度为 n，总数为 m，则时间复杂度为 $O(nm)$. 其中高斯消元法的常数略大.

若是实数线性基，则时间复杂度为 $O(n^2m)$.

线性基合并

线性基的合并只需要暴力处理，即将要合并的一组线性基暴力地插入到另一组线性基即可. 单次合并的时间复杂度是 $O(n^2)$（异或线性基）或 $O(n^3)$（实数线性基）.

线性基求交

线性基求交，严格地说就是求它们张成的两个线性空间的交空间的一组线性基. 本节介绍两种算法. 这两种算法，单次求交的时间复杂度都是 $O(n^2)$（异或线性基）或 $O(n^3)$（实数线性基）.

朴素算法

设要求交的线性基分别是 α 和 β. 线性基求交的算法只需要对线性基暴力合并的算法做如下调整：（以异或线性基为例）

- 将线性基 β 中的向量 β_j 利用 贪心法→9.14@p1706 尝试插入到 α 中，并初始化线性基的交 γ 为空集；
- 在插入时，需要记录要插入的向量中，线性基 β 中元素的贡献. 具体地，维持一个新向量 b，初始化为 β_j，而且，如果正在插入的向量与线性基中第 x 位的向量取了异或，那么贡献 b 也要与第 x 位记录的贡献 b_x 取一次异或；
- 如果插入成功，在线性基的第 x 位插入了向量 $\beta_{(j)'}$，就将第 x 位记录的 b_x 改为得到 $\beta_{(j)'}$ 的过程中线性基 β 中元素的贡献 b；
- 如果插入不成功，就将过程中记录到的线性基 β 中元素的贡献 b 插入到 γ 中.

这样得到的线性基 γ 就是所求的交，当然，该算法同时也求出了线性基的并.

对算法的解释

设合并后的线性基为 $\{\alpha_1,\cdots,\alpha_m,\beta'_{j_1},\cdots,\beta'_{j_\ell}\}$，其中，$\beta'_{j_k}$ 是插入 β_{j_k} 时最后得到的向量．那么，$\{\alpha_1,\cdots,\alpha_m,\beta_{j_1},\cdots,\beta_{j_\ell}\}$ 同样是一组合并后的线性基．记 β^+ 为集合 $\{\beta_{j_1},\cdots,\beta_{j_\ell}\}$，则合并后的基可以写作 $\alpha\cup\beta^+$．而且，和空间中的每个向量 c 都可以唯一地表示成

$$c=a\oplus b$$

的形式，其中，$a\in\operatorname{span}\alpha$ 且 $b\in\operatorname{span}\beta^+$．这个分解中的 b 就是前文算法所 **试图** 记录的「线性基 β 中元素的贡献」．严格地说，只是 β 中最后成功插入的那些向量的贡献．

对于成功的插入，最后记录的 b 就是该分解中的 b 项．设 $\beta_j\in\beta^+$．初始时，$\beta_j=0\oplus\beta_j$，已经是 β_j 在基 $\alpha\cup\beta^+$ 上的正确的分解．在更新 $\beta'_j=a\oplus b$ 为 $\beta'_j\oplus c_x$ 时，因为 $\beta_{(j)'}\oplus c_x=(a\oplus a_x)\oplus(b\oplus b_x)$，所以，只需要更新 b 为 $b\oplus b_x$，就可以保证分解依然正确．因此，归纳可知，最后插入 β'_j 到合并后的线性基中时，记录的贡献 b 就是上述分解中的 b 项．

对于不成功的插入，最后要插入的变量一定会变成 0，而此时的贡献 b 要插入到 γ 中．此时，如果重复上面的论证，会发现仍然能够保证在插入过程中总是有 $\beta_{(j)'}=a\oplus b$，且 $a\in\operatorname{span}\alpha$，只是 b 不再属于 $\operatorname{span}\beta^+$．这是因为初始化时，$\beta_j=0\oplus\beta_j$ 中的 $\beta_j\notin\beta^+$．除此之外，贡献更新时异或的项都属于 $\operatorname{span}\beta^+$．所以，实际上，有 $b\oplus\beta_j\in\operatorname{span}\beta^+$．

那么，为什么将这些插入不成功时的 b 都插入到 γ 中，就能得到交空间的线性基呢？首先，插入 β_j 不成功，最后一定会得到 $0=a\oplus b$，其中，$a\in\operatorname{span}\alpha$ 且 $b\in\operatorname{span}(\beta^+\cup\{\beta_j\})\subseteq\operatorname{span}\beta$．因此，$b=a$ 必然位于交空间 $\operatorname{span}\alpha\cap\operatorname{span}\beta$ 中．反过来，设 c 是交空间中的任意元素，因为 $c\in\operatorname{span}\beta$，所以 c 可以表示为 β 中元素的线性组合（异或和）：

$$c=\bigoplus_{\beta_j\in\beta}\lambda_j\beta_j,$$

其中，$\lambda_j\in\{0,1\}$．对于每一个 $\beta_j\notin\beta^+$，记相应的插入到 γ 中的贡献为 b_j，就有

$$c\oplus\bigoplus_{\beta_j\notin\beta^+}\lambda_jb_j=\bigoplus_{\beta_j\in\beta^+}\lambda_j\beta_j+\bigoplus_{\beta_j\notin\beta^+}\lambda_j(\beta_j\oplus b_j),$$

注意到，b_j 和 c 都位于交空间中，因而左侧必然也位于交空间中，故而左侧可以写成 α 中元素的线性组合；同时，右侧所有项，要么 $\beta_j\in\beta^+$，要么 $\beta_j\notin\beta^+$ 且 $\beta_j\oplus b_j\in\beta^+$，故而，右侧实际上是 β^+ 中元素的线性组合．但是，$\alpha\cup\beta^+$ 线性无关，故而所有的系数都是 0，也就是说 $c=\bigoplus_{\beta_j\notin\beta^+}\lambda_jb_j\in\operatorname{span}\{b_1,\cdots,b_j\}$．这就说明了，这些无法插入的向量的贡献 b 共同张成了交空间．

根据这一解释，过程中维护贡献 b 的目的，实际上是为了维护分解 $a\oplus b$；而且，最后向 γ 插入贡献时也总有 $a=b$．所以，无论维护 α 还是 β 中元素的贡献（即无论维护 a 还是 b），得到的结果都是正确的．如果要维护线性基 α 中元素的贡献，只需要修改初始化时相应贡献的取值：每个 α 中的向量 α_i 初始就有贡献 α_i，而插入的 β_j 初始贡献为 0．

模板题代码如下：

代码（Library Checker Intersection of $\mathbf{F}_2$ vector spaces[6]）

```cpp
#include <array>
#include <iostream>

class LinearBasis {
  static constexpr int K = 30;
  std::array<int, K> a;

  // Size of basis.
```

```
 9   int size() const {
10     int res = 0;
11     for (auto x : a) {
12       if (x) {
13         ++res;
14       }
15     }
16     return res;
17   }
18
19 public:
20   LinearBasis() : a{} {}
21
22   // Insert vector x.
23   void insert(int x) {
24     for (int k = K - 1; ~k && x; --k) {
25       if ((x >> k) & 1) {
26         if (!a[k]) {
27           a[k] = x;
28         }
29         x ^= a[k];
30       }
31     }
32   }
33
34   // Return a basis for *THIS intersecting RHS.
35   LinearBasis intersect(const LinearBasis& rhs) const {
36     LinearBasis res;
37     std::array<int, K> c = a, b_parts = {};
38     for (int i = K - 1; ~i; --i) {
39       int x = rhs.a[i], b_part = x;
40       for (int k = i; ~k && x; --k) {
41         if ((x >> k) & 1) {
42           if (!c[k]) {
43             c[k] = x;
44             b_parts[k] = b_part;
45           }
46           x ^= c[k];
47           b_part ^= b_parts[k];
48         }
49       }
50       res.insert(b_part);
51     }
52     return res;
53   }
54
55   // Output.
56   void print() const {
57     std::cout << size();
58     for (auto x : a) {
59       if (x) {
60         std::cout << ' ' << x;
```

```
61        }
62      }
63      std::cout << '\n';
64    }
65  };
66
67  int main() {
68    int t;
69    std::cin >> t;
70    for (; t; --t) {
71      LinearBasis a, b;
72      int n;
73      std::cin >> n;
74      for (; n; --n) {
75        int x;
76        std::cin >> x;
77        a.insert(x);
78      }
79      int m;
80      std::cin >> m;
81      for (; m; --m) {
82        int x;
83        std::cin >> x;
84        b.insert(x);
85      }
86      a.intersect(b).print();
87    }
88    return 0;
89  }
90
```

Zassenhaus 算法

另一种等价的做法是 Zassenhaus 算法，它同样可以同时计算出两个线性基的并和交．复杂度和上文完全一致．

具体步骤如下：

- 初始化一个向量长度为 $2n$ 的线性基 γ 为空，其中的向量写成 (a, b) 的形式，且 a 和 b 长度均为 n；
- 将 α 中的元素 α_i 以 (α_i, α_i) 的形式插入 γ 中；
- 将 β 中的元素 β_j 以 $(\beta_j, 0)$ 的形式插入 γ 中；
- 最后得到的线性基 γ 中的所有非零元素 (c_k, d_k) 中，c_k 非零的那些向量中项 c_k 的全体组成了 α 和 β 的并的线性基，c_k 为零的那些向量中项 d_k 的全体组成了 α 和 β 的交的线性基．

算法中的构造线性基的方法可以是 贪心法→9.14@p1706 或 高斯消元法→9.14@p1706，只要保证 γ 中的线性基组成行阶梯型矩阵即可．

将 Zassenhaus 算法中的消元的步骤与上面的朴素算法相比较，很容易发现，基于贪心法的 Zassenhaus 算法相当于维护 α 中元素的贡献的朴素算法．如果转而先插入所有 $(\alpha_i, 0)$，再插入所有 (β_j, β_j)，那么基于贪心法的 Zassenhaus 算法就相当于维护 β 中元素贡献的朴素算法．根据消元步骤的等价性，Zassenhaus 算法的正确性也是成立的．

除此之外，还可以再提供一个独立且更为一般的代数证明：

正确性证明

设 V 为一线性空间，且有子空间 $U = \operatorname{span}\alpha$ 和 $W = \operatorname{span}\beta$. 算法本身相当于通过化简为行阶梯型来求子空间

$$H = \operatorname{span}(\{(\alpha_i, \alpha_i) : \alpha_i \in \alpha\} \cup \{(\beta_j, 0) : \beta_j \in \beta\})$$

的一组基 γ. 算法最后，γ 中的元素 (c_k, d_k) 根据 $c_k \neq 0$ 与否需要分为两类，所以不妨考察投影映射 $\pi : H \to V$ 且 $\pi(a, b) = a$. 于是，$\pi(H) = U + W$ 且容易验证

$$\ker \pi = H \cap (\{0\} \times V) = \{0\} \times (U \times W).$$

根据 线性映射的相关定理→9.14.9@p1755，有 $\dim H = \dim \pi(H) + \dim \ker \pi = \dim(U + W) + \dim(U \cap W)$.

行阶梯型矩阵的前几列仍然是行阶梯型矩阵，因而 $c_k \neq 0$ 的行的数目，恰好等于 $\alpha \cup \beta$ 的行秩，亦即 $\dim(U + W)$；而且，这些行中 c_k 的集合就形成了 $U + W$ 的一组基. 剩下的非零行恰好有 $\dim(U \cap W)$ 个，且都满足 $c_k = 0$. 对于这些行中的 d_k，因为有 $(0, d_k) \in \ker \pi$，所以 $d_k \in U \cap W$；而且，$(0, d_k)$ 作为行阶梯型矩阵的行，必然线性无关，这就说明这些 d_k 都线性无关. 综合起来，这些 d_k 是交空间 $U \cap W$ 中大小为 $\dim(U \cap W)$ 的线性无关组，所以也必然是该空间的一组基.

模板题代码如下：

代码（Library Checker Intersection of $\mathbf{F}_2$ vector spaces[7]）

```cpp
#include <iostream>
#include <vector>

class LinearBasis {
  int K;
  std::vector<long long> a;

 public:
  LinearBasis(int K) : K(K), a(K) {}

  // Insert vector x.
  void insert(long long x) {
    for (int k = K - 1; ~k && x; --k) {
      if ((x >> k) & 1) {
        if (!a[k]) {
          a[k] = x;
        }
        x ^= a[k];
      }
    }
  }

  // Output those not exceeding 2^k.
  void print(int k) const {
    int sz = 0;
    for (int i = 0; i < k; ++i) {
      if (a[i]) {
        ++sz;
      }
    }
```

```
    std::cout << sz;
    for (int i = 0; i < k; ++i) {
      if (a[i]) {
        std::cout << ' ' << a[i];
      }
    }
    std::cout << '\n';
  }
};

int main() {
  constexpr int K = 30;
  int t;
  std::cin >> t;
  for (; t; --t) {
    LinearBasis c(K << 1);
    int n;
    std::cin >> n;
    for (; n; --n) {
      int x;
      std::cin >> x;
      c.insert(((long long)x << K) | x);
    }
    int m;
    std::cin >> m;
    for (; m; --m) {
      int x;
      std::cin >> x;
      c.insert((long long)x << K);
    }
    c.print(K);
  }
  return 0;
}

```

注意，输出时只需要考虑前 n 位均为零的向量即可.

拓展：前缀线性基

本节只讨论异或线性基的情形，并假设单个向量可以存储在 $O(1)$ 的空间内，且单次操作复杂度总是 $O(1)$ 的.

对于需要多次查询区间异或最大值的情形，一种常见的做法是 猫树→10.13.4@p2187 配合线性基，时间复杂度为 $O(nm\log m+n^2q)$，其中，n 是向量长度，m 是序列长度，q 是询问次数. 另一种可行的做法是利用前缀线性基（或称时间戳线性基），可以将复杂度降低到 $O(n(m+q))$.

前缀线性基允许对于序列的每个前缀，都维护该前缀的所有后缀的线性基，这样就可以支持查询每个区间的线性基. 注意到序列的某个前缀 $[1,i]$ 的所有后缀 $[j,i]$ 的线性基是相互包含的，即 $[j,i]$ 的线性基总是包含着 $[j+1,i]$ 的线性基，所以，这些后缀的线性基中互不相同的至多只有 n 种，而且总是可以通过向空集中逐步添加新的向量来得到自 $[i,i]$ 到 $[1,i]$ 所有这些后缀的线性基. 因此，利用这个单调性，只需要为添加的每个向量 v，都标记它出现的最大下标 t，就可以在 $O(n)$ 的空间内存储所有后缀的线性基. 而且，查询区间 $[j,i]$ 对应的线性基时，只需要在 i 处的前缀线性基中仅保留标记 $t\geq j$ 的那些向量即可.

不妨将每个向量 v 的标记 t 称为它的时间戳. 线性基中的向量 v 总是可以表示为原序列中某些元素的异或和，比如 $v_{i_1} \oplus v_{i_2} \oplus \cdots \oplus v_{i_k}$. 而在所有这样的可能的表示中，最小下标的最大值就是 t，即

$$t(v) = \max\{j : \exists i_1, \cdots, i_k \in [j, i] \text{ s.t. } v = v_{i_1} \oplus v_{i_2} \oplus \cdots \oplus v_{i_k}\}.$$

这个表达式不过是将上一段的叙述用形式的语言写出来而已. 它给我们带来的启发是, 要维护线性基中每个向量 v 的时间戳，只需要贪心地选取尽可能新的向量替换掉旧的向量即可.

基于上文提到的 贪心法→9.14@p1706 构造线性基，前缀线性基在构造过程中做了如下调整:

- 为线性基中保留的每个向量 a_x 都保存一个时间戳 t_x，初始时均设为 0;
- 要添加序列中第 i 个向量 v，仍然从高位向低位扫，但同时需要记录当前时间 i;
- 如果 v 的第 x 位是一，就比较线性基中已有的向量 a_x 的时间戳 t_x 和当前时间 i:
  - 如果 $i > t_x$，即要添加的向量时间更晚，就将 a_x 设为 v，并更新时间戳为 i，并将旧的 a_x 异或 v 的结果 $a_x \oplus v$ 按照之前记录的时间 t_x 继续添加过程;
  - 如果 $i < t_x$，即要添加的向量时间更早，不更新 a_x 和 t_x，将 v 异或 a_x 后继续添加即可.

也就是说, 如果当前位可以通过较新的向量表示, 就直接用较新的向量; 否则, 保留原来的向量. 在更新位置 x 的向量时, 不能将异或的结果 $a_x \oplus v$ 存入位置 x, 因为异或的结果 $a_x \oplus v$ 的时间戳为 $\min\{t(a_x) = t(v)\} = t(a_x)$，小于要添加的变量 v 的时间戳 $t(v)$. 同样的原因，高斯消元法→9.14@p1706 构造线性基的过程中向上更新时可能会破坏时间戳的性质，所以不再适用于构造前缀线性基.

模板题代码如下:

代码（Codeforces 1100F Ivan and Burgers[8]）

```cpp
#include <algorithm>
#include <array>
#include <iostream>
#include <numeric>
#include <vector>

class LinearBasis {
  static constexpr int K = 20;
  std::array<int, K> a, t;

 public:
  LinearBasis() : a{}, t{} {}

  // Insert vector x at time i.
  void insert(int x, int i) {
    for (int k = K - 1; ~k && x; --k) {
      if (((x >> k) & 1)) {
        if (i > t[k]) {
          std::swap(a[k], x);
          std::swap(t[k], i);
        }
        x ^= a[k];
      }
    }
  }

  // Find max xor of subsets of elements from time i till now.
  int query(int i) const {
    int res = 0;
```

```
    for (int k = K - 1; ~k; --k) {
      if (t[k] >= i && (res ^ a[k]) > res) {
        res ^= a[k];
      }
    }
    return res;
  }
};

int main() {
  int n;
  std::cin >> n;
  std::vector<int> c(n + 1);
  for (int i = 1; i <= n; ++i) std::cin >> c[i];
  int q;
  std::cin >> q;
  std::vector<std::array<int, 2>> qu(q);
  for (auto& v : qu) std::cin >> v[0] >> v[1];
  std::vector<int> ids(q);
  std::iota(ids.begin(), ids.end(), 0);
  std::sort(ids.begin(), ids.end(),
            [&](int l, int r) -> bool { return qu[l][1] < qu[r][1]; });
  LinearBasis lb;
  std::vector<int> res(q);
  for (int i = 1, j = 0; i <= n; ++i) {
    lb.insert(c[i], i);
    for (; j < q && qu[ids[j]][1] == i; ++j) {
      res[ids[j]] = lb.query(qu[ids[j]][0]);
    }
  }
  for (int x : res) std::cout << x << '\n';
  return 0;
}

```

如果需要在线询问，也可以用 $O(mn)$ 的空间将每个前缀处的前缀线性基都存下来再查询，这可以看作是一种「可持久化」线性基．如果需要用到高斯消元法得到的线性基的性质，可以在查询时另行处理．

练习题

- Luogu P3812【模板】线性基[9]
- Acwing 3164. 线性基[10]
- SGU 275 to xor or not xor[11]
- HDU 3949 XOR[12]
- HDU 6579 Operation[13]
- Luogu P4151 [WC2011] 最大 XOR 和路径[14]
- Library Checker - Intersection of $\mathbf{F}_2$ vector spaces[15]
- AtCoder Grand Contest 045 A - Xor Battle[16]
- Codeforces 1100F Ivan and Burgers[17]
- Luogu P3292 [SCOI2016] 幸运数字[18]

参考资料与注释

1. 丘维声，高等代数（下）. 清华大学出版社.
2. Basis (linear algebra) - Wikipedia[19]
3. Vector Basis -- from Wolfram MathWorld[20]
4. Zassenhaus algorithm - Wikipedia[21]

[1] 布尔域

[2] Proof that every vector space has a basis

[3] Schmidt 正交化

[4] 【模板】线性基

[5] 【模板】线性基

[6] Intersection of $\mathbf{F}_2$ vector spaces

[7] Intersection of $\mathbf{F}_2$ vector spaces

[8] 1100F Ivan and Burgers

[9] Luogu P3812【模板】线性基

[10] Acwing 3164. 线性基

[11] SGU 275 to xor or not xor

[12] HDU 3949 XOR

[13] HDU 6579 Operation

[14] Luogu P4151 [WC2011] 最大 XOR 和路径

[15] Library Checker - Intersection of $\mathbf{F}_2$ vector spaces

[16] AtCoder Grand Contest 045 A - Xor Battle

[17] Codeforces 1100F Ivan and Burgers

[18] Luogu P3292 [SCOI2016] 幸运数字

[19] Basis (linear algebra) - Wikipedia

[20] Vector Basis -- from Wolfram MathWorld

[21] Zassenhaus algorithm - Wikipedia

9.14.9 线性映射

研究线性映射是研究线性空间之间的映射.

线性映射可以表示为矩阵的形式，所以在线性映射中矩阵中的大量概念都可以找到对应关系.

线性映射与线性变换

设 V 和 W 是域 F 上的两个线性空间，T 是 V 到 W 的一个映射.

如果对于 W 中任意的向量 x 和 y，域 F 中任意的标量 k 和 l，有:

$$T(kx+ly)=kTx+lTy$$

称 T 是 V 到 W 的一个线性映射. 如果 $W=V$，则称 T 是 V 上的一个线性变换.

例如，恒等变换 T_e 保持空间不变，零变换 T_0 将空间映射至零空间.

可以记 $L(V,W)$ 为所有 V 到 W 的线性映射构成的集合. 对于全体线性变换 $L(V,V)$，也记为 $L(V)$.

性质

- 线性映射将零向量映射到零向量.
- 线性映射保持线性运算形式不变，即，线性运算的线性映射，等于线性映射的线性运算.
- 线性映射保持线性相关性，即，映射前线性相关，映射后也线性相关.

但是线性映射不保持线性无关性. 映射前线性无关，映射后不一定线性无关.

线性映射的矩阵表示

设 V 的维数是 n，V 的一组基为 $\alpha_1,\cdots,\alpha_n$，W 的维数是 m，W 的一组基为 $\beta_1,\cdots,\beta_m$，T 是 V 到 W 的一个线性映射.

将每个 α 经由 T 映射后的向量用 β 表示:

$$T\alpha_j=a_{1j}\beta_1+\cdots+a_{mj}\beta_m$$

采用矩阵记法:

$$T(\alpha_1,\cdots,\alpha_n)=(T\alpha_1,\cdots,T\alpha_n)=(\beta_1,\cdots,\beta_m)A$$

称矩阵 A 为线性映射 T 在这两组基下的矩阵表示.

线性映射的核空间与像空间

这里的核空间与像空间是站在线性映射的视角下叙述的. 借助矩阵表示可以看出, 线性映射的核空间与像空间与矩阵的核空间与像空间是一致的.

设 T 是由空间 V 到空间 W 的线性映射，令:

$$N(T)=\{x\in V\mid Tx=0\}$$

$$R(T)=Im(T)=\{y\in W\mid y=Tx,Vx\in V\}$$

易验证 $N(T)$ 为 V 的子空间，$R(T)$ 为 W 的子空间，称 $N(T)$ 及 $R(T)$ 为 V 的核空间和像空间，并称 $N(T)$ 的维数为 T 的 **零度** 或 **亏**，$R(T)$ 的维数为 T 的 **秩**.

定理: 设 T 是由空间 V 到空间 W 的线性映射，V 的维数有限，则 $N(T)$ 及 $R(T)$ 均为有限维，且有:

$$\dim N(T)+\dim R(T)=\dim V$$

即 T 的亏加秩等于其定义域 V 的维数.

线性变换的矩阵表示

设 V 的维数是 n，V 的一组基为 $\alpha_1,\cdots,\alpha_n$，T 是 V 上的一个线性变换，则有:

$$T\alpha_j=a_{1j}\alpha_1+\cdots+a_{nj}\alpha_n$$

采用矩阵记法:

$$T(\alpha_1,\cdots,\alpha_n)=(T\alpha_1,\cdots,T\alpha_n)=(\alpha_1,\cdots,\alpha_n)A$$

称矩阵 A 为线性变换 T 在这组基下的矩阵表示.

由空间结构和 T 的线性性质，T 由 $T\alpha_1,\cdots,T\alpha_n$ 完全确定，故由 T 唯一确定一个矩阵 A.

定理: 设 V 的维数是 n，$\alpha_1,\cdots,\alpha_n$ 为 V 的一组基，任取 n 阶方阵 A，有且仅有一个从 V 到 V 的线性变换 T，使得 T 的矩阵恰好为 A.

推论: 在 $L(V,V)$ 和全体 n 阶方阵之间存在一一对应关系.

例如: 零变换对应零矩阵，恒等变换对应单位矩阵.

线性变换构成的空间

定理: $L(V)$ 也可以构成线性空间，引入 $L(V)$ 中的运算: 对于 $L(V)$ 中任意的 T_1 与 T_2，V 中任意的 x，域 F 中任意的 k，有:

$$(T_1+T_2)x=T_1x+T_2x$$

$$(kT_1)x=k(T_1x)$$

容易验证 $L(V)$ 是 F 上的一个线性空间，即线性变换空间.

对于 $L(V)$ 中的线性变换 T_1 与 T_2，定义 T_1 与 T_2 的乘积 T_1T_2 为:

$$(T_1T_2)x=T_2(T_1x)$$

可以验证 (T_1T_2) 也是 $L(V)$ 中的线性变换，并且线性变换的乘积满足结合律，而不满足交换律，与矩阵的乘积类似.

对于 $L(V)$ 中的线性变换 T_1，如果 $L(V)$ 中的线性变换 T_2，使得对于 V 中任意的向量 x，有:

$$(T_1T_2)x=T_1(T_2x)=x$$

则称 T_2 是 T_1 的逆变换，记作:

$$T_2=T_1^{-1}$$

且有：

$$T_1T_2 = T_2T_1 = T_e$$

定理：设 V 的维数为 n，$\alpha_1, \cdots, \alpha_n$ 为 V 的一组基，在这组基下线性变换 T_1 的矩阵为 A，T_2 的矩阵为 B，则：

- 线性变换 $T_1 + T_2$ 的矩阵为 $A + B$
- 线性变换的数乘 kT_1 的矩阵为 kA
- 线性变换的乘积 T_1T_2 的矩阵为 AB
- 线性变换 T_1 的逆变换若存在，矩阵为 A^{-1}

坐标

设 n 个向量 x 是 n 维空间 V 的一个基，对于 V 中任意的向量 y，令 y 为：

$$y = a_1x_1 + a_2x_2 + \cdots + a_nx_n = (x_1, x_2, \cdots, x_n)\begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix}$$

称列向量：

$$\begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix}$$

为向量 y 在基 $x_1, x_2, \cdots, x_n$ 下的 **坐标**.

可见，坐标是由域中的标量构成的列向量，与阿贝尔群中的向量应当进行区分.

坐标变换公式

设 V 的维数为 n，$L(V)$ 中有变换 T，T 在基 $\alpha_1, \cdots, \alpha_n$ 下的矩阵为 A. 设：

$$\xi = (\alpha_1, \cdots, \alpha_n)\begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}$$

且有：

$$T\xi = T(\alpha_1, \cdots, \alpha_n)\begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}$$

则有：

$$T\xi = T(\alpha_1, \cdots, \alpha_n)\begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} = (\alpha_1, \cdots, \alpha_n)A\begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}$$

空间 V 中的列向量点本质上都是「基乘坐标」的形式. 空间 V 中的列向量点 x，本身用了单位阵 I 作为基，即 $x = Ix$.

只有同一个基，基不动的时候，单纯的线性变换 T，就是坐标左乘普通矩阵.

把线性变换 T 看成对于空间 V 的一个观测滤镜. 线性变换 T 的作用对象是空间 V，将空间 V 扭曲了. 加了滤镜之后，点本身的位置没有变.

这个定理也说明，对于列向量基的线性变换 T，等价于对于基右乘一个过渡矩阵.

于是，在不同的基之间，坐标关系是左乘过渡矩阵的逆矩阵.

过渡矩阵

设 n 个向量 x 与 n 个向量 y 是空间 V 的两组基．对于 $1 \le i \le n$，令每个向量 y_i 在基 $x_1, x_2, \cdots, x_n$ 下的坐标为：

$$y_i = (x_1, x_2, \cdots, x_n) \begin{pmatrix} a_{1i} \\ a_{2i} \\ \vdots \\ a_{ni} \end{pmatrix}$$

于是 n 个向量 y 排成等式左边的矩阵，n 个坐标排成等式右边的矩阵 A：

$$(y_1, y_2, \cdots, y_n) = (x_1, x_2, \cdots, x_n)A$$

矩阵 A 称为由基 $x_1, x_2 \cdots, x_n$ 到基 $y_1, y_2 \cdots, y_n$ 的 **过渡矩阵**，也称为变换矩阵．

显然过渡矩阵可逆．对于上式，由基 $y_1, y_2 \cdots, y_n$ 到基 $x_1, x_2 \cdots, x_n$ 的过渡矩阵为 A^{-1}．

可见，过渡矩阵是由域中的标量构成的矩阵，并非阿贝尔群中的向量排成的矩阵，应当予以区分．

设 n 个向量 x 与 n 个向量 y 是空间 V 的两组基．对于空间 V 中的同一个向量 z，有：

$$z = (x_1, x_2, \cdots, x_n) \begin{pmatrix} \xi_1 \\ \xi_2 \\ \vdots \\ \xi_n \end{pmatrix} = (y_1, y_2 \cdots, y_n) \begin{pmatrix} \eta_1 \\ \eta_2 \\ \vdots \\ \eta_n \end{pmatrix}$$

代入上文的

$$(y_1, y_2 \cdots, y_n) = (x_1, x_2 \cdots, x_n)A$$

由唯一性，得到：

$$\begin{pmatrix} \xi_1 \\ \xi_2 \\ \vdots \\ \xi_n \end{pmatrix} = A \begin{pmatrix} \eta_1 \\ \eta_2 \\ \vdots \\ \eta_n \end{pmatrix}$$

或者

$$\begin{pmatrix} \eta_1 \\ \eta_2 \\ \vdots \\ \eta_n \end{pmatrix} = A^{-1} \begin{pmatrix} \xi_1 \\ \xi_2 \\ \vdots \\ \xi_n \end{pmatrix}$$

这是纯粹坐标之间的变换，坐标变换公式均在标量域中．由于前文做了区分，线性空间与阿贝尔群中的向量是「抽象的向量」，而坐标与过渡矩阵的元素均在标量域中，视为「具体的向量」，两种向量应当视为「不同的东西」．

矩阵可以对整个空间，即全体坐标进行变换，列向量 x 作为坐标遍布整个空间．

单位矩阵 I 由单位向量构成．矩阵 A 会将单位矩阵 I 变换到矩阵 A 的每个列向量，即将单位向量变换到矩阵 A 的每个列向量．因此左乘矩阵 A，也可以视为将空间做了这样的变换．

向量左乘矩阵，也可以视为坐标左乘向量组．用坐标的观点看待就是：

$$Iy = Xa$$

同一个列向量 y，在「正常」的空间，单位矩阵 I 代表的空间下，坐标为 y，在变换后新的空间里，坐标将记为 a．这样一来，矩阵 X 不仅是正常空间下的一组基，也是从向量组 I 到向量组 X 的过渡矩阵．

线性变换 T 会将一个基映射为另一个基，于是坐标也被映射为另一个坐标．

如果将基 α 映射到 β 对应的线性变换 T 的过渡矩阵是 A，那么对应的基矩阵就有 $\beta = \alpha A$．

于是坐标的关系恰好反过来．假设线性变换 T 映射后的坐标是 b，即加滤镜后观察到坐标 b，于是点在 V 的表示就是 βb．还原的办法就是用过渡矩阵，把点在 V 的表示写成 $\alpha A b$．于是坐标变换为左乘过渡矩阵的逆矩阵的看法就明显了．

线性变换与矩阵相似

在空间 V 中的一个线性变换 T 对于空间 V 的基 α 的关系：

线性变换 T 作用于基 α，将基 α 映射到了 $T(\alpha)$，相当于在基 α 右乘一个 A，即 $T(\alpha)=\alpha A$.

矩阵相似考虑的问题是：同一个线性变换 T，在基 β 的空间 V 中描述为矩阵 B，在基 α 的空间 V 中描述为矩阵 A.

如果过渡矩阵为 C，即 $\beta=\alpha C$，那么两个描述 B 和 A 之间有怎样的联系.

由于是同一个变换 T，可以发现一个事实，变换前后的过渡矩阵关系始终成立，即：

$$T(\beta)=T(\alpha)C=\alpha AC$$

线性变换 T 在基 β 视角下仍旧为右乘，基 β 转化到基 α 再右乘一个 C，变换前后保持过渡矩阵 C 的关系：

$$T(\beta)=\beta B=\alpha CB$$

于是问题得到解决：

$$B=C^{-1}AC$$

定理：设 $L(V)$ 中有变换 T，则 T 在不同基下的矩阵 **相似**.

对于方阵 A 和方阵 B，如果存在可逆矩阵 C 使得 $B=C^{-1}AC$，则 A 和 B 相似.

矩阵相似保持秩不变，因此矩阵相似可以推出矩阵等价．但是，等价的两个矩阵未必相似.

由于矩阵相似与形状密切相关，因此矩阵相似和向量组等价、方程组同解之间没有关系.

回过头来，矩阵相似的解释就是 4 个等式：$\beta=\alpha C$、$T(\alpha)=\alpha A$、$T(\beta)=\beta B$、$T(\beta)=T(\alpha)C$.

参考资料

- 【官方双语/合集】线性代数的本质 - 系列合集 P13 09 - 基变换[1]

外部参考资料

[1] 【官方双语/合集】线性代数的本质 - 系列合集 P13 09 - 基变换



9.14.10 特征多项式

特征的这部分只研究方阵，即矩阵 A 对应的线性变换将 n 个向量映射到 n 个向量.

由于在实际问题中，经常要考虑连续进行重复的变换，如果只用「矩阵 A 对应的线性变换将单位阵 I 变换为 A」的描述，就会很抽象．此时最好的办法是找「不动点」，即变换当中不动的部分.

然而事实上，矩阵 A 对应的线性变换很可能没有不动点，于是退而求其次，寻找共线或者类似于简单变形的部分.

特征值与特征向量

在矩阵 A 对应的线性变换作用下，一些向量的方向不改变，只是伸缩了.

设 V 是 F 上的线性空间，T 是 V 上的线性变换．若存在 F 中的 λ 与 V 中的 **非零向量** ξ，使得：

$$T\xi=\lambda\xi$$

则称 λ 为 T 的一个 **特征值**，而 ξ 为 T 的 **属于特征值 λ 的一个特征向量**.

特征向量在同一直线上，在线性变换作用下保持方向不改变（压缩到零也认为是方向不改变）．特征向量不唯一，与特征向量共线的向量都是特征向量，但是规定零向量不是特征向量，拥有方向的向量自然是非零向量．特征向量的特征值就是它伸缩的倍数.

在实际应用中，一般对于拥有相同特征值的特征向量，会选取一组基作为它们全体的代表.

设 $\alpha_1,\alpha_2,\cdots,\alpha_n$ 是 V 的一组基，T 在这组基下的矩阵为 A，即：

$$T(\alpha_1,\alpha_2,\cdots,\alpha_n)=(\alpha_1,\alpha_2,\cdots,\alpha_n)A$$

设 λ_0 是 T 的一个特征值，ξ 为 T 的属于特征值 λ_0 的一个特征向量，且有非零向量 X 满足：

$$\xi=(\alpha_1,\alpha_2,\cdots,\alpha_n)X$$

于是有：

$$T\xi=\lambda_0\xi$$
$$T(\alpha_1,\alpha_2,\cdots,\alpha_n)X=\lambda_0(\alpha_1,\alpha_2,\cdots,\alpha_n)X$$
$$(\alpha_1,\alpha_2,\cdots,\alpha_n)AX=\lambda_0(\alpha_1,\alpha_2,\cdots,\alpha_n)X$$
$$AX=\lambda_0X$$
$$(A-\lambda_0I)X=0$$

所以相应的行列式也为 0.

特征多项式

考虑一个 $n\times n$ 的矩阵 A，其中 $n\geq 0\wedge n\in\mathbb{Z}$. 设 λ 为一个参量，矩阵 $\lambda I-A$ 称为 A 的 **特征矩阵**.

特征矩阵的行列式称为 A 的 **特征多项式**，展开为一个 n 次多项式，根为 A 的特征值，记为 $p_A(\lambda)$：

$$p_A(\lambda)=\det(\lambda I_n-A)=\begin{vmatrix}\lambda-a_{11} & -a_{12} & \cdots & -a_{1n}\\ -a_{21} & \lambda-a_{22} & \cdots & -a_{2n}\\ \vdots & \vdots & & \vdots\\ -a_{n1} & -a_{n2} & \cdots & \lambda-a_{nn}\end{vmatrix}$$

其中 I_n 为一个 $n\times n$ 的单位矩阵. 一些地方会定义为 $p_A(\lambda)=\det(A-\lambda I_n)$ 与我们的定义仅相差了一个符号 $(-1)^n$，但采用这种定义得到的 $p_A(\lambda)$ 一定为首一多项式，而另外的定义则仅当 n 为偶数时才是首一多项式. 需要注意的是 0×0 的矩阵行列式为 1 是良定义的.

相应于 $(\lambda_0I-A)X=0$ 的非零解向量 X，称为 A 的属于 λ_0 的特征向量.

线性变换 T 有特征值 λ_0 等价于矩阵 A 有特征值 λ_0.

线性变换 T 有特征向量 ξ 等价于矩阵 A 有特征向量 X，其中有：

$$\xi=(\alpha_1,\cdots,\alpha_n)X$$

根据代数基本定理，特征多项式可以分解为：

$$f(\lambda)=|\ \lambda I-A\ |=(\lambda-\lambda_1)^{d_1}\cdots(\lambda-\lambda_m)^{d_m}$$

称 d_i 为特征值 λ_i 的 **代数重数**. 全体代数重数的和为空间维数 n.

求解矩阵的全部特征值及特征向量

分为以下步骤：

- 计算行列式 $|\ \lambda I-A\ |$.
- 求出多项式 $f(\lambda)=|\ \lambda I-A\ |$ 在域 F 中的全部根，即 A 的特征值.
- 对 A 的每个特征值 λ，解齐次线性方程组 $(\lambda I-A)X=0$，求出它的一组基础解系 $X_1,\cdots,X_t$，则 A 的属于 λ 的全部特征向量为：

  $$k_1X_1+k_2X_2+\cdots+k_tX_t$$

  该表达式中的 k_i 不全为零.
- 线性变换 T 的属于 λ 的特征向量为：

  $$\xi_i=(\alpha_1,\cdots,\alpha_n)X_i$$

  因此，属于 λ 的全部特征向量为：

  $$k_1\xi_1+k_2\xi_2+\cdots+k_t\xi_t$$

  该表达式中的 k_i 不全为零.

特征值与特征向量是否存在，依赖于 V 所在的域.

相似变换

引入

若 $n \times n$ 的矩阵 A 为上三角矩阵如

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ & a_{2,2} & \cdots & a_{2,n} \\ & & \ddots & \vdots \\ & & & a_{n,n} \end{bmatrix}$$

那么

$$\begin{aligned} p_A(x) &= \det(xI_n - A) \\ &= \begin{bmatrix} x - a_{1,1} & -a_{1,2} & \cdots & -a_{1,n} \\ & x - a_{2,2} & \cdots & -a_{2,n} \\ & & \ddots & \vdots \\ & & & x - a_{n,n} \end{bmatrix} \\ &= \prod_{i=1}^{n} (x - a_{i,i}) \end{aligned}$$

可轻松求得，下三角矩阵也是类似的. 但如果 A 不属于这两种矩阵，则需要使用相似变换，使得矩阵变为容易求得特征多项式的形式.

定义

对于 $n \times n$ 的矩阵 A 和 B，当存在 $n \times n$ 的可逆矩阵 P 满足

$$B = P^{-1}AP$$

则矩阵 A 和 B 相似，记变换 $A \mapsto P^{-1}AP$ 为相似变换. 且 A 和 $P^{-1}AP$ 有相同的特征多项式.

考虑

$$\begin{aligned} \det(xI_n - P^{-1}AP) &= \det(xP^{-1}I_nP - P^{-1}AP) \\ &= \det(P^{-1}xI_nP - P^{-1}AP) \\ &= \det(P^{-1}) \cdot \det(P) \cdot \det(xI_n - A) \\ &= \det(xI_n - A) \\ &= p_A(x) \end{aligned}$$

得证，对于 $A \mapsto PAP^{-1}$ 也是一样的. 另外 $p_A(0) = (-1)^n \cdot \det(A)$，因为 $p_A(0) = \det(-1 \cdot I_nA) = \det(-1 \cdot I_n) \cdot \det(A)$ 故 $\det(A) = \det(P^{-1}AP)$.

定理：相似矩阵有相同的特征多项式及特征值，反之不然.

定理表明，线性变换的矩阵的特征多项式与基的选取无关，而直接由线性变换决定，故可称之为线性变换的特征多项式.

矩阵 A 的特征多项式 $f(\lambda) = |\ \lambda I - A\ |$ 是一个首一的多项式. 根据韦达定理，它的 $n-1$ 次系数为:

$$-(\lambda_1 + \cdots + \lambda_n) = -(a_{11} + \cdots + a_{nn}) = -trA$$

其中 trA 称为 A 的迹，为 A 的主对角线元素之和.

根据韦达定理，特征多项式的常数项为:

$$(-1)^n|\ A\ | = (-1)^n(\lambda_1 \cdots \lambda_n)$$

定理：相似的矩阵有相同的迹.

换位公式

定理：无论矩阵 A 和矩阵 B 是否方阵，只要乘法能进行，则矩阵 AB 的迹等于矩阵 BA 的迹.

一种证法是直接展开，即证毕. 另一种证法用到换位公式.

定理：设 A 为 m 行 n 列矩阵，设 B 为 n 行 m 列矩阵，则有：

$$\lambda^n|\ \lambda I_m - AB\ | = \lambda^m|\ \lambda I_n - BA\ |$$

该公式表明 AB 与 BA 有相同的非零特征值.

舒尔（Schur）引理

任意的 n 阶矩阵 A 都相似于一个上三角阵，即存在满秩阵 P，使得 $P^{-1}AP$ 为上三角阵，它的主对角线上元素为 A 的全部特征值.

推论：设 A 的 n 个特征值为 $\lambda_1, \cdots, \lambda_n$，$\phi(x)$ 为任一多项式，则矩阵多项式 $\phi(A)$ 的 n 个特征值为：

$$\phi(\lambda_1), \cdots, \phi(\lambda_n)$$

特别地，kA 的特征值为 $k\lambda_1, \cdots, k\lambda_n$，$A^m$ 的特征值为 $\lambda_1^m, \cdots, \lambda_n^m$.

使用高斯消元进行相似变换

对 $n \times n$ 的矩阵 B 可以进行高斯消元，其基本操作为初等行变换.

在对矩阵使用上述操作（左乘初等矩阵）后再右乘其逆矩阵即相似变换，左乘为行变换，易发现右乘即列变换.

若能将矩阵通过相似变换变为上三角或下三角的形式，那么可以轻松求出其特征多项式. 但若对主对角线上的元素应用变换 $A \mapsto T_{ij}(k)AT_{ij}(-k)$ 后会导致原本通过 $A \mapsto T_{ij}(k)A$ 将第 i 行第 j 列的元素消为零后右乘 $T_{ij}(-k)$ 即将 A 的第 i 列的 $-k$ 倍加到第 j 列这一操作使得之前消为零的元素现在可能不为零，可能不能将其变为上三角或下三角形式.

后文将说明对次对角线上的元素应用变换后得到的矩阵依然可以轻松得到其特征多项式.

上 Hessenberg 矩阵

对于 $n > 2$ 的形如

$$H = \begin{bmatrix} \alpha_1 & h_{12} & \cdots & \cdots & h_{1n} \\ \beta_2 & \alpha_2 & h_{23} & \cdots & \vdots \\ & \ddots & \ddots & \ddots & \vdots \\ & & \ddots & \ddots & h_{(n-1)n} \\ & & & \beta_n & \alpha_n \end{bmatrix}$$

的矩阵我们称为上 Hessenberg 矩阵，其中 β 为次对角线.

我们使用相似变换将次对角线以下的元素消为零后即能得到上 Hessenberg 矩阵，而求出一个 $n \times n$ 上 Hessenberg 矩阵的特征多项式则可在 $O(n^3)$ 时间完成.

我们记 H_i 为只保留 H 的前 i 行和前 i 列的矩阵，记 $p_i(x) = \det(xI_i - H_i)$ 那么

$$H_0 = [], \quad p_0(x) = 1$$

$$H_1 = [\alpha_1], \quad p_1(x) = \det(xI_1 - H_1) = x - \alpha_1$$

$$H_2 = \begin{bmatrix} \alpha_1 & h_{12} \\ \beta_2 & \alpha_2 \end{bmatrix}, \quad p_2(x) = \det(xI_2 - H_2) = (x - \alpha_2)p_1(x) - \beta_2 h_{12} p_0(x)$$

在计算行列式时我们一般选择按零最多的行或列余子式展开，余子式即删除了当前选择的元素所在行和列之后的矩阵，在这里我们选择按最后一行进行展开，有

$$
\begin{aligned}
p_3(x) &= \det(xI_3 - H_3) \\
&= \begin{vmatrix} x-\alpha_1 & -h_{12} & -h_{13} \\ -\beta_2 & x-\alpha_2 & -h_{23} \\ & -\beta_3 & x-\alpha_3 \end{vmatrix} \\
&= (x-\alpha_3)\cdot(-1)^{3+3}p_2(x) - \beta_3 \cdot (-1)^{3+2} \begin{vmatrix} x-\alpha_1 & -h_{13} \\ -\beta_2 & -h_{23} \end{vmatrix} \\
&= (x-\alpha_3)p_2(x) - \beta_3(h_{23}p_1(x) + \beta_2 h_{13}p_0(x))
\end{aligned}
$$

观察并归纳，对 $2 \le i \le n$ 有

$$
p_i(x) = (x-\alpha_i)p_{i-1}(x) - \sum_{m=1}^{i-1} h_{i-m,i}\left(\prod_{j=i-m+1}^{i}\beta_j\right)p_{i-m-1}(x)
$$

至此完成了整个算法，该算法一般被称为 Hessenberg 算法.

Cayley–Hamilton 定理

对于任意的 n 阶矩阵 A，特征多项式为 $f(\lambda) = |\lambda I - A|$，则必有 $f(A) = 0$.

对于线性变换 T 有平行的结果：如果 $f(\lambda)$ 为 T 的特征多项式，则 $f(T)$ 为零变换.

由本定理可知，对于任意的矩阵 A，必有可以使其零化的多项式.

最小多项式

设 V 是一个 n 维向量空间, 由于线性变换对应的矩阵有 n^2 个元素, 一切线性变换构成 n^2 维线性空间.

对于一个特定的线性变换 T，从作用 0 次到作用 n 次，总共 n^2+1 个线性变换，它们对应的矩阵一定线性相关. 于是存在非零多项式 f，使得 $f(T)$ 为零变换，称变换 T 满足多项式 f. 在 T 满足的所有多项式 f 中，存在次数最低的.

可以将矩阵 A 零化的最小次数的首一多项式称为 A 的最小多项式，记为 $m_A(\lambda)$.

根据多项式的辗转相除法，最小多项式是唯一的，且可整除任一 A 的零化多项式. 特别地，最小多项式整除特征多项式.

定理：在不计重数的情况下，矩阵 A 的特征多项式 $f(\lambda)$ 与最小多项式 $m_A(\lambda)$ 有相同的根.

定理：矩阵 A 的属于不同特征值的特征向量线性无关.

应用

在信息学中我们一般考虑 $(\mathbb{Z}/m\mathbb{Z})^{n\times n}$ 上的矩阵，通常 m 为素数，进行上述相似变换是简单的，当 m 为合数时，我们可以考虑类似辗转相除的方法来进行.

实现

```cpp
#include <cassert>
#include <iostream>
#include <random>
#include <vector>

using Matrix = std::vector<std::vector<int>>;
using i64 = int64_t;

Matrix to_upper_Hessenberg(const Matrix &M, int mod) {
  Matrix H(M);
  int n = H.size();
```

```
  for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j) {
      if ((H[i][j] %= mod) < 0) H[i][j] += mod;
    }
  }
  for (int i = 0; i < n - 1; ++i) {
    int pivot = i + 1;
    for (; pivot < n; ++pivot) {
      if (H[pivot][i] != 0) break;
    }
    if (pivot == n) continue;
    if (pivot != i + 1) {
      for (int j = i; j < n; ++j) std::swap(H[i + 1][j], H[pivot][j]);
      for (int j = 0; j < n; ++j) std::swap(H[j][i + 1], H[j][pivot]);
    }
    for (int j = i + 2; j < n; ++j) {
      for (;;) {
        if (H[j][i] == 0) break;
        if (H[i + 1][i] == 0) {
          for (int k = i; k < n; ++k) std::swap(H[i + 1][k], H[j][k]);
          for (int k = 0; k < n; ++k) std::swap(H[k][i + 1], H[k][j]);
          break;
        }
        if (H[j][i] >= H[i + 1][i]) {
          int q = H[j][i] / H[i + 1][i], mq = mod - q;
          for (int k = i; k < n; ++k)
            H[j][k] = (H[j][k] + i64(mq) * H[i + 1][k]) % mod;
          for (int k = 0; k < n; ++k)
            H[k][i + 1] = (H[k][i + 1] + i64(q) * H[k][j]) % mod;
        } else {
          int q = H[i + 1][i] / H[j][i], mq = mod - q;
          for (int k = i; k < n; ++k)
            H[i + 1][k] = (H[i + 1][k] + i64(mq) * H[j][k]) % mod;
          for (int k = 0; k < n; ++k)
            H[k][j] = (H[k][j] + i64(q) * H[k][i + 1]) % mod;
        }
      }
    }
  }
  return H;
}

std::vector<int> get_charpoly(const Matrix &M, int mod) {
  Matrix H(to_upper_Hessenberg(M, mod));
  int n = H.size();
  std::vector<std::vector<int>> p(n + 1);
  p[0] = {1 % mod};
  for (int i = 1; i <= n; ++i) {
    const std::vector<int> &pi_1 = p[i - 1];
    std::vector<int> &pi = p[i];
    pi.resize(i + 1, 0);
    int v = mod - H[i - 1][i - 1];
```

```
64      if (v == mod) v -= mod;
65      for (int j = 0; j < i; ++j) {
66        pi[j] = (pi[j] + i64(v) * pi_1[j]) % mod;
67        if ((pi[j + 1] += pi_1[j]) >= mod) pi[j + 1] -= mod;
68      }
69      int t = 1;
70      for (int j = 1; j < i; ++j) {
71        t = i64(t) * H[i - j][i - j - 1] % mod;
72        int prod = i64(t) * H[i - j - 1][i - 1] % mod;
73        if (prod == 0) continue;
74        prod = mod - prod;
75        for (int k = 0; k <= i - j - 1; ++k)
76          pi[k] = (pi[k] + i64(prod) * p[i - j - 1][k]) % mod;
77      }
78    }
79    return p[n];
80  }
81
82  bool verify(const Matrix &M, const std::vector<int> &charpoly, int mod) {
83    if (mod == 1) return true;
84    int n = M.size();
85    std::vector<int> randvec(n), sum(n, 0);
86    std::mt19937 gen(std::random_device{}());
87    std::uniform_int_distribution<int> dis(1, mod - 1);
88    for (int i = 0; i < n; ++i) randvec[i] = dis(gen);
89    for (int i = 0; i <= n; ++i) {
90      int v = charpoly[i];
91      for (int j = 0; j < n; ++j) sum[j] = (sum[j] + i64(v) * randvec[j])
  % mod;
92      std::vector<int> prod(n, 0);
93      for (int j = 0; j < n; ++j) {
94        for (int k = 0; k < n; ++k) {
95          prod[j] = (prod[j] + i64(M[j][k]) * randvec[k]) % mod;
96        }
97      }
98      randvec.swap(prod);
99    }
100   for (int i = 0; i < n; ++i)
101     if (sum[i] != 0) return false;
102   return true;
103 }
104
105 int main() {
106   std::ios::sync_with_stdio(false);
107   std::cin.tie(nullptr);
108   int n, mod;
109   std::cin >> n >> mod;
110   Matrix M(n, std::vector<int>(n));
111   for (int i = 0; i < n; ++i)
112     for (int j = 0; j < n; ++j) std::cin >> M[i][j];
113   std::vector<int> charpoly(get_charpoly(M, mod));
114   for (int i = 0; i <= n; ++i) std::cout << charpoly[i] << ' ';
```

```
115   assert(verify(M, charpoly, mod));
116   return 0;
117 }
```

上述 Hessenberg 算法不具有数值的稳定性，所以 $\mathbb{R}^{n\times n}$ 上的矩阵在使用前需要其他算法进行调整或改用其他具有数值稳定性的算法.

我们可以将特征多项式与常系数齐次线性递推联系起来，也可结合 Cayley–Hamilton 定理、多项式取模加速一些域上求矩阵幂次的算法.

Cayley–Hamilton 定理指出

$$\begin{aligned} p_A(A) &= A^n + c_1A^{n-1} + \cdots + c_{n-1}A + c_nI \\ &= O \end{aligned}$$

其中 O 为 $n\times n$ 的零矩阵，$A\in\mathbb{C}^{n\times n}$ 且 $p_A(x)=x^n+\sum_{i=1}^{n}c_ix^{n-i}\in\mathbb{C}[x]$ 为 A 的特征多项式.

若我们要求 A^K 其中 K 较大，那么可以求出 $f(x)=x^K \bmod p_A(x)$ 后利用 $f(A)=A^K$.

而 $\deg(f(x))<n$ 显然. 我们令 $f(x)=\sum_{i=0}^{n-1}f_ix^i$ 且 $n=km$ 那么

$$\begin{aligned} f_{km-1}x^{km-1}+\cdots+f_1x+f_0 = (\cdots(&f_{km-1}x^{k-1}+\cdots+f_{k(m-1)})x^k \\ &+f_{k(m-1)-1}x^{k-1}+\cdots+f_{k(m-2)})x^k \\ &+\cdots \\ &+f_{k-1}x^{k-1}+\cdots+f_1x+f_0 \end{aligned}$$

令 $k=\sqrt{n}$ 可以发现计算 $f(A)$ 大约需要 $O(\sqrt{n})$ 次矩阵与矩阵的乘法.

参考文献

- Rizwana Rehman, Ilse C.F. Ipsen.La Budde's Method for Computing Characteristic Polynomials[1].
- Marshall Law.Computing Characteristic Polynomials of Matrices of Structured Polynomials[2].
- Mike Paterson.On the Number of Nonscalar Multiplications Necessary to Evaluate Polynomials[3].

外部参考资料

[1] La Budde's Method for Computing Characteristic Polynomials

[2] Computing Characteristic Polynomials of Matrices of Structured Polynomials

[3] On the Number of Nonscalar Multiplications Necessary to Evaluate Polynomials

9.14.11 对角化

特征子空间

矩阵 A 的属于 λ_0 的全部特征向量，再添上零向量，构成一个线性空间，称为矩阵 A 的一个特征子空间，记为 $E(\lambda_0)$. 它是齐次线性方程组：

$$(\lambda_0 I - A)X = 0$$

的解空间.

对于特征子空间 $E(\lambda_i)=N(\lambda_iI-A)$，由亏加秩定理有：

$$r(\lambda_iI-A)+\dim N(\lambda_iI-A)=n$$

因此，特征子空间 $E(\lambda_i)$ 的维数为：

$$\dim E(\lambda_i)=n-r(\lambda_iI-A)$$

也称为 λ_i 的 **几何重数**.

不变子空间

在研究线性变换 T 的时候，常常希望选取空间 V 的一个基，使得线性变换 T 对于这个基的矩阵具有尽可能简单的形状.

设 V 是数域 F 上的线性空间，W 是 V 的一个子空间，T 是 V 上的一个线性变换. 如果对于 W 中任意的向量 x，都有 $T(x)$ 也在 W 中（也称为空间在变换下不变或稳定），称 W 是 T 的一个不变子空间.

空间在变换下不变，并不是说坐标在变换下真的「不变」，有可能是进行了一个拉伸等变形，只是变形后还落在空间里.

- 线性空间 V 的任意一个子空间都是数乘变换的不变子空间.
- 对于 V 中任意的线性变换 T，空间 V 和零子空间都是 T 的不变子空间，称为平凡不变子空间.
- 不变子空间的交与和也是不变子空间.

设 W 是线性变换 T 的一个不变子空间. 只考虑 T 在不变子空间 W 上的作用，就得到子空间 W 本身的线性变换，称为 T 在子空间 W 上的限制，记作 $T\mid_W$.

对于 V 中任意的线性变换 T，像空间 $R(T)$ 与核空间 $N(T)$ 是 T 的不变子空间. 这两种情况的含义是，空间 V 在变换前后，完成了自身的压缩（像空间），或者压缩到 0（核空间）.

对于 V 中任意的线性变换 T，T 的特征子空间是 T 的不变子空间.

准素分解

根据代数基本定理，最小多项式可以分解为:

$$m_A(\lambda)=(\lambda-\lambda_1)^{r_1}\cdots(\lambda-\lambda_S)^{r_S}$$

考虑最小多项式代入变元 λ 为矩阵 A 后，各个因式的核空间，构成矩阵 A 的一系列不变子空间:

$$W_i=N((\lambda_iI-A)^{r_i})$$

定理：该不变子空间 W_i 的维数，恰好为特征值 λ_i 的代数重数.

回顾一下, 代数重数是指特征多项式各个因式的次数, 几何重数是指特征子空间 $E(\lambda_i)=N(\lambda_iI-A)$ 的维数. 这个不变子空间 W_i 与特征子空间 $E(\lambda_i)$，两者都是矩阵的核空间，并且两个矩阵构成最小多项式 r_i 次幂的关系. 也就是说，特征子空间的维数是几何重数,「特征子空间」经过最小多项式 r_i 次幂后到达一个「不变子空间」，不变子空间的维数到达了特征多项式的代数重数.

该定理其实是下面准素分解定理的推论.

记矩阵 A 对应的线性变换 T，在每个子空间 W_i 上的限制 $T_i=T\mid_{W_i}$. 于是 T_i 的最小多项式是 $(x-\lambda_i)^{r_i}$.

定理：设 V 是域 F 上的线性空间，T 是 V 上的一个线性变换. 那么空间 V 可以关于线性变换 T 进行准素分解，拆成若干不变子空间 W_i 的直和.

$$V=W_1\oplus W_2\oplus\cdots\oplus W_S$$

这意味着，T 在某组基下的矩阵是准对角阵:

$$\mathrm{diag}\{A_1,A_2,\cdots,A_S\}$$

其中，A_i 是 T_i 在对应基下的矩阵.

该定理表明，可以使用不变子空间简化线性变换的矩阵.

可对角化矩阵

对于 n 阶方阵 A，如果相似于一个对角阵，则称 A 为可对角化矩阵，或称单纯矩阵.

- 对角阵的和、积、逆，如果存在，仍然是对角阵，其对角线上的元素就是它的特征值.
- 线性变换 T 的矩阵为可对角化矩阵，等价于 T 在某组基下的矩阵为对角阵.

定理：设矩阵 A 的全部互异特征根为 $\lambda_1,\cdots,\lambda_m$，则以下命题等价:

- 矩阵 A 可对角化.

- 矩阵 A 有 n 个线性无关的特征向量.
- 以下公式成立:

$$\dim E(\lambda_1) + \cdots + \dim E(\lambda_m) = n$$

前文已经指出，特征多项式的分解式中特征值的次数称为代数重数，特征子空间的维数称为几何重数．这个定理也表明，矩阵 A 可对角化，等价于 A 的每个特征值 λ 的代数重数都等于它的几何重数.

推论：如果 n 阶方阵 A 恰有 n 个互异特征值，则它必可对角化．反之则不一定.

定理：矩阵 A 可对角化当且仅当 A 的最小多项式没有重根.

矩阵的相似也会保持特征向量之间的线性相关关系不变.

特征向量完全可能不是实数，也完全可能找不到 n 个线性无关的特征向量.

对于重特征值而言，特征向量张成空间．为了描述这个空间，需要从其中选择代表．一般会选择线性无关的代表，代表的个数就是空间的维数.

选取代表时，常常将它们正交化与单位化．最终得到的就是一套单位正交的代表.

特征向量不一定正交，不同特征值的特征向量，可能无法正交．因此正交化只能对于重特征值的特征向量进行．但是单位化可以对任意特征向量进行.

幂零矩阵

设 T 是空间 V 的一个线性变换．如果存在一个正整数 r，使得 T^r 为零变换，称 T 是空间 V 的一个幂零变换.

对于某一个正整数 r，满足条件 $N^r = 0$ 的矩阵称为幂零矩阵.

一般可以进一步假定 r 是使 T^r 为零变换的最小正整数，于是 T 的最小多项式是 x^r．于是存在一个向量 ξ_0，使得:

- $T^r(\xi_0) = 0$
- $T^{r-1}(\xi_0) \neq 0$

循环子空间

定理：设 T 是空间 V 的一个线性变换，ξ 是空间 V 的一个向量．如果存在一个正整数 s，使得:

- $T^s(\xi) = 0$
- $T^{s-1}(\xi) \neq 0$

那么向量 $\xi, T(\xi), \cdots, T^{s-1}(\xi)$ 线性无关.

由这个定理可以给出一个定义:

设 T 是空间 V 的一个线性变换，W 是 V 的一个子空间．如果存在一个向量 ξ_0 和一个正整数 r，使得:

- 向量 $\xi_0, T(\xi_0), \cdots, T^{r-1}(\xi_0)$ 构成 W 的一个基.
- 如下等式成立:

  $T^r(\xi_0) = 0$

那么子空间 W 称为关于 T 的一个循环子空间，简称 T 循环子空间．此时 ξ_0 称为循环子空间 W 的一个生成向量，向量 $\xi_0, T(\xi_0), \cdots, T^{r-1}(\xi_0)$ 称为 W 的一个循环基.

显然，一个 T 循环子空间 W 在 T 作用下不变，并且对于循环子空间 W 中的任意向量 ξ，均有 $T^r(\xi) = 0$，这里 r 为循环子空间的维数.

幂零 Jordan 块

如果空间 W 是变换 T 的循环子空间，那么 T 在 W 上的限制 $T\mid_W$ 是 W 的一个幂零变换，并且 $T\mid_W$ 关于 W 的倒序排列的循环基 $T^{r-1}(\xi_0), T^{r-2}(\xi_0), \cdots, \xi_0$ 的矩阵是如下形状的 r 阶上三角矩阵:

$$N_r = \begin{pmatrix} 0 & 1 & 0 & \cdots & 0 & 0 \\ 0 & 0 & 1 & \cdots & 0 & 0 \\ 0 & 0 & 0 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 0 & 1 \\ 0 & 0 & 0 & \cdots & 0 & 0 \end{pmatrix}$$

矩阵 N_r 称为一个 r 阶幂零 Jordan 矩阵，或者 r 阶幂零 Jordan 块.

设 T 是 n 维空间 V 的一个幂零变换，把出现在 V 关于 T 的循环子空间的分解中，唯一确定的一组正整数 $r_1 \geq \cdots \geq r_S$ 叫做 T 的不变指数.

对于 n 阶幂零矩阵 A，A 与一个上述形状的矩阵 N 相似，也唯一确定一个正整数序列 $r_1 \geq \cdots \geq r_S$，称为矩阵 A 的不变指数.

幂零阵虽然不能和对角阵相似，但是可以相似于这样的标准形式. 在 Jordan 标准型，将相似对角化与幂零阵的标准形式，二者结合起来，给出一般的矩阵通过相似变换可以达到的标准形式.

一些定理

1. 设 T 是空间 V 的一个幂零变换，而
   $h(x) = a_0 + a_1 x + \cdots + a_m x^m$
   是一个多项式，那么当且仅当 $a_0 \neq 0$ 时，线性变换 $h(T)$ 有逆变换. 当 $h(T)$ 可逆时，$h(T)$ 的逆变换也是 T 的一个多项式.
2. 设 T 是空间 V 的一个幂零变换，W 是一个 r 维 T 循环子空间，ξ 是 W 中的向量. 如果存在一个整数 k，使得
   $T^{r-k}(\xi) = 0$
   那么存在 W 中的向量 η，使得
   $\xi = T^k(\eta)$
3. 设 T 是 n 维空间 V 的一个幂零变换，x^r 是 T 的最小多项式，令 W_1 是一个 r 维 T 循环子空间，那么存在 W_1 的一个余子空间 W_2，使得:
   $V = W_1 \oplus W_2$
   并且 W_2 也在 T 作用下不变.
4. 设 T 是 n 维空间 V 的一个幂零变换，那么 V 可以分解为 T 循环子空间的直和:
   $V = W_1 \oplus W_2 \oplus \cdots \oplus W_S$
5. 每一个 n 阶幂零矩阵都与一个形如:
   $$N = \begin{pmatrix} N_{r_1} & & & 0 \\ & N_{r_2} & & \\ & & \cdots & \\ 0 & & & N_{r_S} \end{pmatrix}$$
   的矩阵相似，这里的每一个 N_{r_i} 是一个 r_i 阶幂零 Jordan 块.
6. 如果规定 T 循环子空间 W_i 按照维数 r_i 降序排列 $r_1 \geq \cdots \geq r_S$，那么将 V 分解为 T 循环子空间的方法是由 T 唯一确定的.

9.14.12 Jordan 标准型

Jordan 分解

设 T 是 n 维空间 V 上的一个线性变换. 如果 T 的最小多项式为:

$$m_A(\lambda) = (\lambda - \lambda_1)^{r_1}(\lambda - \lambda_2)^{r_2} \cdots (\lambda - \lambda_k)^{r_k}$$

那么由准素分解可知，空间 V 可以分解为子空间的直和:

$$V = V_1 \oplus V_2 \oplus \cdots \oplus V_k$$

其中 $V_i = N((A - \lambda_i I)^{r_i})$，式中 A 为 T 对应的矩阵，这些子空间都在 T 作用下不变.

令变换 T_i 为 V 在子空间 V_i 上的射影，即构造多项式 $u_i(T)$ 使得：

- $T_i = u_i(T)\dfrac{m_A(T)}{(T - \lambda_i T_e)^{r_i}}$
- $T_1 + T_2 + \cdots + T_k = T_e$

式中 T_e 表示空间 V 的恒等变换. 于是有性质：

- 变换 T_i 在空间 V_i 上的限制 $T_i|_{V_i}$ 为空间 V_i 的恒等变换.
- 如果 i 与 j 不相等，变换 T_i 在空间 V_j 上的限制 $T_i|_{V_j}$ 为空间 V_j 的零变换.

于是变换 T_i 将空间 V 的每一个向量 ξ 映射为它在空间 V_i 中的分量 ξ_i.

构造变换：

$$T_D = \lambda_1 T_1 + \lambda_2 T_2 + \cdots + \lambda_k T_k$$

由于每一个变换 T_i 都是变换 T 的一个多项式，所以变换 T_D 也是变换 T 的一个多项式，于是每一个子空间 V_i 在变换 T_D 下不变.

由上述等式可知，变换 T_D 在子空间 V_i 上的限制 $T_D|_{V_i}$ 是子空间 V_i 的一个位似，位似系数为 λ_i. 因此，变换 T_D 可以对角化.

构造：

$$T_N = T - T_D$$

于是变换 T_N 也是变换 T 的一个多项式，所以每一个子空间 V_i 在变换 T_N 下不变. 对于子空间 V_i 中的任意向量 ξ_i，有：

$$T_N^{r_i}(\xi_i) = T - T_D^{r_i}(\xi_i) = T - \lambda_i T_i^{r_i}(\xi_i) = 0$$

令 r 为全体 r_i 的最大值，那么对于空间 V 中的任意向量 ξ，变换 T_N 的 r 次方将向量 ξ 映射至零向量. 因此变换 T_N 是一个幂零变换.

这样，空间 V 的每一个变换 T 都可以写成：

$$T = T_D + T_N$$

其中 T_D 可以对角化，而 T_N 是一个幂零变换. 因为 T_D 和 T_N 都是变换 T 的多项式，所以它们的乘积可交换：

$$T_D T_N = T_N T_D$$

定理：设 T_1 和 T_2 是空间 V 的两个可对角化变换，且 $T_1 T_2 = T_2 T_1$，那么存在一个基，使得 T_1 和 T_2 关于这同一个基的矩阵是对角形式.

定理：设 T 是 n 维空间 V 上的一个线性变换，那么存在一个可对角化变换 T_D 和一个幂零变换 T_N，使得：

- $T = T_D + T_N$
- $T_D T_N = T_N T_D$

它们都是变换 T 的多项式，并且它们由变换 T 唯一确定.

该定理给出关于变换 T 的分解，称为 T 的若尔当（Jordan）分解，T_D 叫做 T 的可对角化部分，T_N 叫做 T 的幂零部分.

同样地，有矩阵的 Jordan 分解：

定理：设 A 是一个 n 阶矩阵，那么存在一个可对角化矩阵 D 和一个幂零矩阵 N，使得：

- $A = D + N$
- $DN = ND$

它们都是矩阵 A 的多项式，并且它们由矩阵 A 唯一确定.

该定理给出关于矩阵 A 的分解，称为 A 的若尔当（Jordan）分解，D 叫做 A 的可对角化部分，N 叫做 A 的幂零部分.

lambda 矩阵

接下来引入的部分是含有变元参量 λ 的更广义的矩阵，不仅仅是一个数表. 这部分讨论相较单纯由数构成的矩阵而言，更加广泛一些.

对于 λ 矩阵，对应空间相应的域，变为含有一个变元 λ 的有理式域.

以 λ 的多项式为元素的矩阵称为 λ 矩阵，记为 $A(\lambda)$.

由于多项式域包含数域，数字矩阵是特殊的 λ 矩阵，数字矩阵 A 的特征矩阵 $\lambda I - A$ 是一种 λ 矩阵.

lambda 矩阵的初等变换

对于 λ 矩阵，同样可以定义加减法、乘法、初等变换、秩. 对于 λ 方阵，同样可以定义行列式、余子式、代数余子式.

对于 λ 矩阵，初等变换与数阵大多相同，仅将倍加变换改为（这里以行变换为例）：

- 用 λ 的多项式 $\varphi(\lambda)$ 乘某行并加到另一行上.

注意倍乘变换不进行修改. 这是因为倍加变换不改变行列式，而倍乘变换改变行列式. 为了保持多项式域的秩的性质，行列式只能在数域上进行改变.

相应的初等矩阵也一并进行修改.

易见三种初等阵的行列式均为非零常数，因此均为满秩. 所以它们左乘或右乘，不改变 λ 矩阵的秩.

若 $A(\lambda)$ 经过有限次初等变换变为 $B(\lambda)$，则称 $A(\lambda)$ 和 $B(\lambda)$ 等价.

对于 λ 矩阵，如果等价，则秩相同. 反之则不然，这与数字矩阵有区别.

Smith 标准型

定理：设 λ 矩阵的秩是 r，则 $A(\lambda)$ 一定等价于：

$$\begin{pmatrix} D(\lambda) & 0 \\ 0 & 0 \end{pmatrix}$$

其中：

$$D(\lambda) = \begin{pmatrix} d_1(\lambda) & & \\ & \ddots & \\ & & d_r(\lambda) \end{pmatrix}$$

每一个 $d_i(\lambda)$ 是一个首 1 多项式，并且相邻两个多项式有整除关系 $d_i(\lambda) \mid d_{i+1}(\lambda)$.

称此标准型为 Smith 标准型，称 $d_i(\lambda)$ 为不变因子.

具体求解 Smith 标准型的办法是，从左上角到右下角进行消元，每次左上角的元素是右下方剩余的全体多项式的最大公因式，并借助左上角的元素将该行该列全部消为 0.

定理：条件 $A(\lambda)$ 和 $B(\lambda)$ 等价，等价于条件 $A(\lambda)$ 和 $B(\lambda)$ 拥有完全一样的不变因子.

初等因子

由代数基本定理，设 $A(\lambda)$ 的不变因子 $d_1(\lambda), d_2(\lambda), \cdots, d_m(\lambda)$ 的分解为：

$$d_i(\lambda) = (\lambda - \lambda_1)^{e_{i1}} (\lambda - \lambda_2)^{e_{i2}} \cdots (\lambda - \lambda_S)^{e_{iS}}$$

其中 $\lambda_1, \cdots, \lambda_S$ 互不相同. 由于：

$$d_i(\lambda) \mid d_{i+1}(\lambda)$$

因此指数 $e_{1j}, e_{2j}, \cdots, e_{mj}$ 递增，并且最后一项 $d_m(\lambda)$ 的各项指数均非零.

上式中指数大于零的全部因子，统称为 $A(\lambda)$ 的初等因子.

注意，初等因子计重数．如果对于某个 j，指数 e_{ij} 出现了若干次，则对应的初等因子 $(\lambda-\lambda_j)^{e_{ij}}$ 也应当出现相应次数．

之前的定理说明，$A(\lambda)$ 与 $B(\lambda)$ 等价，等价于他们两个拥有完全一致的不变因子．不变因子完全相同，自然初等因子也完全相同，但是反之则不然．事实上有结论：

定理：$A(\lambda)$ 与 $B(\lambda)$ 不变因子完全相同，等价于初等因子和秩均完全相同．

于是「初等因子和秩均完全相同」也成为判断 λ 矩阵等价性的条件．

在初等变换的时候，也可以先将 $A(\lambda)$ 变换为对角阵，再求出初等因子和秩，再求出不变因子得到标准型．有结论：

定理：设 $A(\lambda)$ 等价于对角阵：

$$\operatorname{diag}\{f_1(\lambda),f_2(\lambda),\cdots,f_r(\lambda),0,\cdots,0\}$$

那么有 $f_1(\lambda),f_2(\lambda),\cdots,f_r(\lambda)$ 的全体一次因子的幂 $(\lambda-\lambda_j)^{e_{ij}}$，构成 $A(\lambda)$ 的初等因子．

由初等因子和秩构造不变因子的具体方法为：先将初等因子按照因式分类，排成表格，把同类因式进行降幂排列放到同一行，各类因式的最高次幂放到一列，把列数用 1 补齐至秩 r，那么每一列的乘积构成一个不变因子．

在特征矩阵中的应用

如果 A 与 B 是数阵，那么它们的特征矩阵是 λ 矩阵．有结论：

定理：条件数阵 A 与 B 相似，等价于条件特征矩阵 $\lambda I-A$ 和 $\lambda I-B$ 等价．

由于特征矩阵 $\lambda I-A$ 只在主对角线含有 n 个 λ，所以秩为 n．由上述推理，同型的数阵的特征矩阵的秩始终相等，于是有等价性：

数阵 A 与 B 相似，等价于特征矩阵 $\lambda I-A$ 和 $\lambda I-B$ 有完全相同的初等因子．

对于特征矩阵 $\lambda I-A$，初等变换保持等价性，所以不改变秩．

观察三种初等变换，由于唯一被改写的倍加变换不改变行列式，事实上三种初等变换仅对行列式的结果多项式改变常数倍，因此不改变行列式的结果多项式的因式分解与次数．

因此特征矩阵 $\lambda I-A$ 的行列式为 n 次多项式，初等变换化为 Smith 标准型后，由于秩为 n，行列式就是主对角线全体不变因子的乘积，也等于全体初等因子的乘积．因此，特征矩阵 $\lambda I-A$ 的全体初等因子的次数之和等于 n．

Jordan 标准型

矩阵

$$\begin{pmatrix}\lambda&1&0&\cdots&0&0\\0&\lambda&1&\cdots&0&0\\0&0&\lambda&\cdots&0&0\\\vdots&\vdots&\vdots&&\vdots&\vdots\\0&0&0&\cdots&\lambda&1\\0&0&0&\cdots&0&\lambda\end{pmatrix}$$

主对角线上的元素都是 λ，紧邻主对角线上方的元素都是 1，其余位置都是 0，叫做属于 λ 的一个 Jordan 矩阵，或称 Jordan 块．

显然，幂零 Jordan 矩阵是 Jordan 矩阵的特例，即 λ 为 0 的情形．

定理：设 T 是 n 维空间 V 的一个变换，$\lambda_1,\cdots,\lambda_k$ 是 T 的一切互不相同的特征值，那么存在一个基，使得 T 关于这个基的矩阵有形状：

$$\begin{pmatrix} B_1 & & 0 \\ & B_2 & \\ & & \ddots \\ 0 & & B_k \end{pmatrix}$$

其中

$$B_i = \begin{pmatrix} J_{i1} & & 0 \\ & J_{i2} & \\ & & \ddots \\ 0 & & J_{is_i} \end{pmatrix}$$

其中 $J_{i1}, \cdots, J_{is_i}$ 都是属于 λ_i 的 Jordan 块.

这是因为，首先根据最小多项式：

$$m_A(\lambda) = (\lambda - \lambda_1)^{r_1}(\lambda - \lambda_2)^{r_2} \cdots (\lambda - \lambda_k)^{r_k}$$

有准素分解：

$$V = V_1 \oplus V_2 \oplus \cdots \oplus V_k$$

其中：

$$V_i = N((A - \lambda_i I)^{r_i})$$

式中 A 为 T 对应的矩阵.

令变换 S_i 为 T 在 V_i 上的限制 $T\mid_{V_i}$，接下来试图对每一个 S_i 进行 Jordan 分解.

记 T_e 为 V 上的恒等变换. 与前文的 Jordan 分解不同，记 T_i 为 S_i 的 Jordan 分解中的幂零部分：

$$S_i = \lambda_i T_e + T_i$$

于是 T_i 为子空间 V_i 的一个幂零变换，事实上也是 $T - \lambda_i T_e$ 在 V_i 上的限制 $(T - \lambda_i T_e)|_{V_i}$.

子空间 V_i 可以分解为幂零变换 T_i 循环子空间的直和：

$$V_i = W_{i1} \oplus W_{i2} \oplus \cdots \oplus W_{is_i}$$

在每一个循环子空间 W_{ij} 里，取一个循环基并倒序排列，凑成 V_i 的一个基，于是 T_i 关于这个基的矩阵有形状：

$$N_i = \begin{pmatrix} N_{i1} & & 0 \\ & N_{i2} & \\ & & \ddots \\ 0 & & N_{is_i} \end{pmatrix}$$

全体 N_{ij} 均为幂零 Jordan 块. 于是对于 V_i 上述选取的基，S_i 对应的矩阵是：

$$B_i = \begin{pmatrix} \lambda_i & & 0 \\ & \lambda_i & \\ & & \ddots \\ 0 & & \lambda_i \end{pmatrix} + \begin{pmatrix} N_{i1} & & 0 \\ & N_{i2} & \\ & & \ddots \\ 0 & & N_{is_i} \end{pmatrix} = \begin{pmatrix} J_{i1} & & 0 \\ & J_{i2} & \\ & & \ddots \\ 0 & & J_{is_i} \end{pmatrix}$$

这里 $J_{i1}, J_{i2}, \cdots, J_{is_i}$ 都是属于 λ_i 的 Jordan 块.

对于每一个子空间 V_i，按照以上方式选取一个基，凑起来成为 V 的基，那么 T 关于这个基的矩阵即构成定理规定的形式.

形如：

$$\begin{pmatrix} J_1 & & 0 \\ & J_2 & \\ & & \ddots \\ 0 & & J_m \end{pmatrix}$$

的 n 阶矩阵，其中每一个 J_i 都是一个 Jordan 块，叫做一个 Jordan 标准型.

定理：每一个 n 阶矩阵 A 都与一个 Jordan 标准型相似．除了各个 Jordan 块排列的次序以外，与 A 相似的 Jordan 标准型是由 A 唯一确定的．

注意在上述构造的矩阵 B_i 中，第一项是一个单位阵的若干倍，自然可以和第二项交换．因此，第一项就是 B_i 的 Jordan 分解的可对角化部分，第二项就是 B_i 的 Jordan 分解的幂零部分．

在一个矩阵对应的 Jordan 标准型里面，主对角线上的元素构成的对角阵是这个矩阵对应的 Jordan 标准型的可对角化部分，把主对角线上的元素换成 0 就得到这个矩阵对应的 Jordan 标准型的幂零部分．

定理：对于矩阵 A 的 Jordan 标准型中，每一个 Jordan 块：

$$J_i = \begin{pmatrix} \lambda_i & 1 & & & \\ & \lambda_i & 1 & & \\ & & \ddots & \ddots & \\ & & & \ddots & 1 \\ & & & & \lambda_i \end{pmatrix}$$

对应于特征矩阵 $\lambda I - A$ 的一个初等因子 $(\lambda - \lambda_i)^{n_i}$，特征矩阵 $\lambda I - A$ 的全体初等因子对应于矩阵 A 的 Jordan 标准型中的全体 Jordan 块．

这是因为，矩阵 A 相似于它的 Jordan 标准型，因此两者的特征矩阵也等价，将 Jordan 标准型的特征矩阵化为 Smith 标准型即可看出．

由这个定理，借助特征矩阵 $\lambda I - A$ 的初等因子，可以写出矩阵 A 的 Jordan 标准型．

一个推论是，矩阵 A 可对角化，等价于特征矩阵 $\lambda I - A$ 的初等因子均为一次的．

弗罗贝尼乌斯（Forbenious）定理

上文指出，n 阶特征矩阵的 Smith 标准形的秩为 n．

定理：设矩阵 A 的特征矩阵 $\lambda I - A$ 的 Smith 标准形为：

$$\mathrm{diag}\{d_1(\lambda), d_2(\lambda), \cdots, d_n(\lambda)\}$$

则最后一个不变因子 $d_n(\lambda)$ 恰好为矩阵 A 的最小多项式 $m_A(\lambda)$．

推论：矩阵 A 可对角化的等价条件为：

- 最小多项式 $m_A(\lambda)$ 无重根．
- 特征矩阵 $\lambda I - A$ 的不变因子无重根．
- 特征矩阵 $\lambda I - A$ 的初等因子均为一次的．

9.15 线性规划

9.15.1 线性规划基础

> **Authors:** Ir1d, YZircon, huhaoo, QAQAutoMaton, Enter-tainer, Marcythm, sshwy, partychicken, Konano, H-J-Granger, baker221, isdanni, ksyx

引入

线性规划（linear programming, LP）是研究线性约束条件下线性目标函数最值问题的方法总称，是运筹学的一个分支，在多方面均有应用．线性规划的某些特殊情况，如网络流、多商品流量等问题都有可能在算法竞赛题目中出现．算法竞赛很少会出现只能用线性规划算法解决的问题，绝大多数这类问题可以通过网络流建模等方法更高效地解决．

一个简单的例子

一个问题能够写成线性规划的形式，既要有若干个线性约束条件，又要有线性的目标函数．

考虑下面的例子：

例子

早点师傅每天可以制作一定数量的包子和油条，这两种早餐深受顾客喜爱．为了最大化利润，师傅希望尽可能多地制作早点，但在实际操作中受到食材、时间等多种资源的限制．为此，师傅统计了制作每份早点所需的食材用量、制作时间及其对应的利润，具体如下表所示：

早点	植物油	面粉	时间	利润
包子	4	7	8	5
油条	7	3	6	6

假设师傅每天最多可以购入 66 单位的植物油和 60 单位的面粉，并且最多可以投入 96 单位的制作时间．那么，师傅应如何合理安排包子和油条的生产数量，才能使每天的利润最大化？

用数学语言描述，可以设 x_1 和 x_2 分别是师傅制作包子和油条的数量．那么，「总共需要的植物油不超过 66 单位」就可以表示为

$$4x_1 + 7x_2 \leq 66.$$

类似地，「总共需要的面粉不超过 60 单位」和「总共需要的时间不超过 96 单位」可以表示为

$$7x_1 + 3x_2 \leq 60,$$

$$8x_1 + 6x_2 \leq 96.$$

另外，师傅不可能生产出负数单位的早点，所以，还有条件

$$x_1, x_2 \geq 0.$$

师傅就是要在这些限制下，最大化利润：

$$z = 5x_1 + 6x_2.$$

这就是一个典型的线性规划问题．它的目标函数是关于决策变量的线性函数，约束条件则由决策变量构成的线性等式或不等式组成．

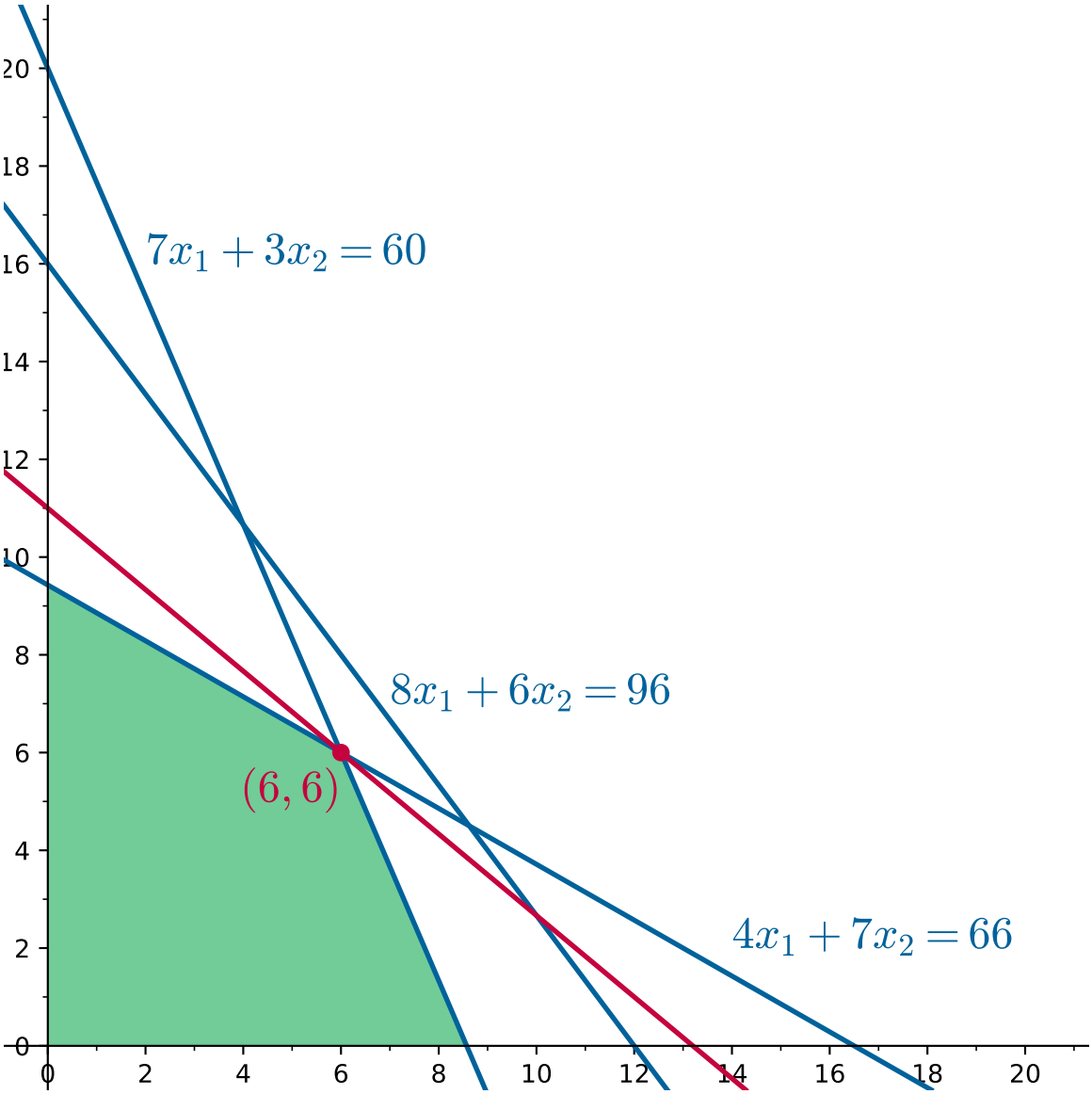
图解法

对于只有两个决策变量的线性规划问题，可以通过图解法直观地解决问题．

考虑本节的问题

$$\begin{aligned} \max_{x_1, x_2} \ & z = 5x_1 + 6x_2 \\ \text{subject to } & 4x_1 + 7x_2 \leq 66, \\ & 7x_1 + 3x_2 \leq 60, \\ & 8x_1 + 6x_2 \leq 96, \\ & x_1, x_2 \geq 0 \end{aligned}$$

对应的几何图像．最后一行约束表示可选的点 (x_1, x_2) 都出现在第一象限，另外的三个约束则表示可选的点一定在直线 $4x_1 + 7x_2 = 66$、直线 $7x_1 + 3x_2 = 60$ 和直线 $8x_1 + 6x_2 = 96$ 的下方．这些区域的交集（如下图绿色区域所示）就是所有可供选择的点的集合：



接下来要最大化 $z = 5x_1 + 6x_2$ 的取值．如果将该等式视作直线 $5x_1 + 6x_2 = z$ 的方程，则随着 z 的变化，将得到一族平行直线，且 z 越大，直线就越靠近右上方．因此，只需要不断移动直线直至达到某一临界位置，使得再向右上移动一点点，直线就不再和图中所示区域相交，此时直线对应的 z 就是所求的最大值．

如图所示，这样的情形发生在红点所示位置．它是直线 $4x_1 + 7x_2 = 66$ 和直线 $7x_1 + 3x_2 = 60$ 的交点．联立两直线方程可知，它的坐标是 $(6, 6)$．这是本问题唯一的最优解．早点师傅的最大利润是 $z = 66$．

当问题涉及多于两个决策变量时，图解法不再适用．但是，本节的例子中的一些观察仍然有效．线性规划问题中的每个不等式约束都描述了一个「半平面」，所有可行的解的集合就是这些「半平面」的交集，因此，总是一个「凸多边形」．规划问题的最优解总是可以在该「凸多边形」的某个「顶点」处取得．这些「顶点」的坐标可以通过联立这些「半平面」的「边界」的方程求得．将这些观察拓展到高维空间，就发展出了一个高效的求解线性规划问题的方法——单纯形法．这也是算法竞赛中最常应用的方法．

另外一个值得注意的问题是，原则上，早点师傅制作的包子和油条都不是无限可分的，应当是某个整数．虽然本题求解过程中没有明确地限制这一点，但是由于最终的最优解的确是整数，所以，即使加上整数限制，本题的答案仍然是可行的．但是对于很多规划问题，最优解可能无法在整点处取得，这些问题实际上是一类整数规划问题，而非简单的线性规划问题．本文的结尾简单地讨论了这一类问题．

基本概念

本节介绍线性规划问题的基本概念.

线性规划问题

一个线性规划问题 P 通常由如下两部分组成:

- 线性目标函数, 即形如
  $f(x_1, x_2, \cdots, x_n) = c_1x_1 + c_2x_2 + \cdots + c_nx_n$
  的函数, 其中, $c_i \in \mathbf{R}$ 是常数;
- 线性约束, 即形如
  $g_j(x_1, x_2, \cdots, x_n) = a_{j1}x_1 + a_{j2}x_2 + \cdots + a_{jn}x_n \leq (=, \geq) b_j$
  的不等式或等式约束, 其中, $a_{ji}, b_j \in \mathbf{R}$ 都是常数.

线性规划问题, 就是要在满足所给约束的前提下, 最大化或者最小化目标函数. 满足所给约束的解 $(x_1, x_2, \cdots, x_n) \in \mathbf{R}^n$ 称为 **可行解** (feasible solution); 在所有可行解中, 使得目标函数取得最值的解称为 **最优解** (optimal solution).

标准形式

为了方便描述和进一步处理, 通常需要指定一个线性规划问题的标准形式. 不同文献可能有不同的规定方式, 本文规定线性规划的标准形式如下:

$$\min_{\{x_i\}} \sum_{i=1}^{n} c_i x_i$$

$$\text{subject to} \sum_{i=1}^{n} a_{ji} x_i = b_j \geq 0,\ j = 1, \cdots, m,$$

$$x_i \geq 0,\ i = 1, \cdots, n.$$

也就是说, 线性规划问题是最小化问题, 所有决策变量都有非负约束, 且除此之外只包含若干右侧常量非负的等式约束. 利用 矩阵→9.14.4@p1716 可以更为简洁地表达这一问题:

$$\min\{c^T x : Ax = b \geq 0,\ x \geq 0\}.$$

其中, $x = (x_i) \in \mathbf{R}^n$ 是决策变量, $b = (b_j) \in \mathbf{R}^m$ 和 $A = (a_{ji}) \in \mathbf{R}^{m \times n}$ 是约束中涉及的常量. 线性规划问题的规模就是指它的决策变量的数目和它的约束的个数.

> **向量不等式**
>
> 本文中会多次出现像 $b \geq 0$ 这样的向量不等式. 一般地, 对于向量 $x, y \in \mathbf{R}^n$, 不等式 $x \leq y$ 表示 $\forall i(x_i \leq y_i)$, 即逐维地进行实数意义上的比较. 这一关系是向量空间上的 偏序关系→9.20@p1951, 也就是说, 存在两个向量无法比较的情形.

标准形式的选取只是为了行文方便, 而并没有任何特别之处, 因为任何线性规划问题都可以等价地写成下面的六种形式:

$$\min\{c^T x : Ax = b,\ x \geq 0\},$$

$$\min\{c^T x : Ax \geq b\},$$

$$\min\{c^T x : Ax \geq b,\ x \geq 0\},$$

$$\max\{c^T x : Ax = b,\ x \geq 0\},$$

$$\max\{c^T x : Ax \leq b\},$$

$$\max\{c^T x : Ax \leq b,\ x \geq 0\}.$$

下列操作可以将所有线性规划问题都等价地转化为这六种形式之一:

1. 通过添加负号, 即将 c 变为 $-c$, 就可以完成最大化问题和最小化问题的相互转化.

2. 通过添加负号，即将 $a_j^T x \lessgtr b_j$ 替换成 $-a_j^T x \gtrless -b_j$，就可以完成不等式约束的两种方向的相互转化，或将等式约束的右侧常量变为非负数.
3. 所有的等式约束 $a_j^T x = b_j$ 都可以替换成两个相反方向的不等式约束 $a_j^T x \geq b_j$ 和 $a_j^T x \leq b_j$.
4. 所有的不等式约束 $a_j^T x \leq (\geq) b_j$ 都可以通过添加非负松弛变量 s_j 的方式，转化为等式约束 $a_j^T x + (-) s_j = b_j$ 以及相应的非负约束 $s_j \geq 0$.
5. 如果某个决策变量 x_i 没有非负约束，那么，可以将它替换成两个非负变量的差值，即 $x_i = x_i^+ - x_i^-$ 且 $x_i^+, x_i^- \geq 0$.

通过这些操作转化得到的线性规划问题的规模不超过原问题的规模的二倍，而且这些问题的可行解和最优解都很容易相互转化. 因此, 对于一般形式的线性规划问题, 总是可以首先将它转化为标准形式（或上述六种形式之一）再进行求解.

例子

考虑线性规划问题

$$\begin{aligned} \max\ & 3x_1 - 2x_2 + x_3 \\ \text{subject to}\ & 2x_1 + 3x_2 + 4x_3 \geq 1, \\ & 3x_1 + 4x_2 \leq 5, \\ & 5x_2 - x_3 = -1, \\ & x_1, x_2 \geq 0. \end{aligned}$$

通过操作 1、2 和 3 可以将它转化为形式 $\min\{c^T x : Ax \geq b\}$，即

$$\begin{aligned} \min\ & -3x_1 + 2x_2 - x_3 \\ \text{subject to}\ & 2x_1 + 3x_2 + 4x_3 \geq 1, \\ & -3x_1 - 4x_2 \geq -5, \\ & 5x_2 - x_3 \geq -1, \\ & -5x_2 + x_3 \geq 1, \\ & x_1 \geq 0, \\ & x_2 \geq 0. \end{aligned}$$

通过操作 4 和 5 可以将它转化为形式 $\max\{c^T x : Ax = b,\ x \geq 0\}$，即

$$\begin{aligned} \max\ & 3x_1 - 2x_2 + x_3^+ - x_3^- \\ \text{subject to}\ & 2x_1 + 3x_2 + 4x_3^+ - 4x_3^- - x_4 = 1, \\ & 3x_1 + 4x_2 + x_5 = 5, \\ & 5x_2 - x_3^+ + x_3^- = -1, \\ & x_1, x_2, x_3^+, x_3^-, x_4, x_5 \geq 0. \end{aligned}$$

可行域与问题的解

所有可行解的集合 $\mathcal{D} \subseteq \mathbf{R}^n$ 称为线性规划问题 P 的 **可行域**（feasible region）. 从几何角度看，每个不等式约束 $a_j^T x \leq b_j$ 都描述了一个半空间 $\{x \in \mathbf{R}^n : a_j^T x \leq b_j\}$，每个等式约束 $a_j^T x = b_j$ 都描述了一个超平面 $\{x \in \mathbf{R}^n : a_j^T x = b_j\}$，所以，可行域一定是有限多个半空间和超平面的交集. 在优化领域[1]，这样的几何体通常称为 $\mathbf{R}^n$ 中的 **多面体**（polyhedron）. 多面体一定是闭凸集，但未必是有界的. 有界的多面体也称为 **多胞形**（polytope）. 多胞形可以看作是平面上的多边形在高维空间的推广，而多面体将它进一步推广到可能无界的情形.

[1]不同文献可能对这两个名词有着不同的定义: 有些文献会将有界的情形称作「多面体」, 而将无界的情形称作「多胞形」; 有些文献不会假定它们一定是凸集; 有些文献会用「多面体」称呼三维空间中的多胞形. 本文采取了与 Schrijver (1998) 和 Boyd and Vandenberghe (2004) 等文献一致的定义.

多面体的例子

此处列举了一些常见的多面体：

1. 空集 $\emptyset$，又称为 **零胞形**（nullitope），维度规定为 -1.
2. **仿射子空间**（affine subspace），即若干超平面的交集 $\{x \in \mathbf{R}^n : Ax = b\}$. 它相当于线性方程组 $Ax = b$ 的解集：当方程组无解时，它就是空集；否则，它总是可以写成 $x_0 + V$ 的形式，其中，$x_0 \in \mathbf{R}^n$ 且 $V \subseteq \mathbf{R}^n$ 是 $n - \operatorname{rank} A$ 维线性子空间. 特别地，超平面也是仿射子空间.
3. **多面体锥**（polyhedral cone），即空间中有限多个点 $\{x_i\}$ 的全体非负线性组合 $\{\sum_i \alpha_i x_i : \alpha_i \geq 0\}$. 它是顶点位于原点的凸锥体. 等价地，它可以看作是由若干个经过原点的超平面围成的多面体，即 $\{x \in \mathbf{R}^n : Ax \leq 0\}$. 特别地，半空间也是多面体锥.
4. 多胞形，即有界的多面体. 特别地，-1、0、1、2、3 维的多胞形就是常见的空集、点、线段、多边形和（通常意义下的）多面体. 一个集合是多胞形，当且仅当它是有限多个点 $\{x_i\}$ 的凸包 $\{\sum_i \alpha_i x_i : \alpha_i \geq 0,\ \sum_i \alpha_i = 1\}$. 一个 k 维的多胞形至少是由 $k+1$ 个点生成的凸包.
5. **单纯形**（simplex），即恰由 $k+1$ 个点生成的 k 维多胞形. 它是最简单的 k 维多胞形. 特别地，-1、0、1、2、3 维的单纯形分别是空集、点、线段、三角形和四面体. 最简单的 k 维单纯形的例子，就是 $\{x \in \mathbf{R}^{k+1} : x_i \geq 0,\ \sum_i x_i = 1\}$. 实际上，任何 k 维单纯形都可以通过仿射变换（即平移和伸缩）变为这样一种特殊情形. 值得注意的是，单纯形法并不是真的在单纯形上进行的.

任何多面体，都可以看作是一个多面体锥和一个多胞形的 Minkowski 和→12.7@p3113：前者描述了多面体无界的部分，后者描述了多面体有界部分的形状. 这个多面体锥是唯一的：多面体 $\{x \in \mathbf{R}^n : Ax \leq b\}$ 分解得到的多面体锥一定是 $\{x \in \mathbf{R}^n : Ax \leq 0\}$.

线性规划的解与多面体的结构紧密相关. 对于多面体 $\mathcal{D} \subseteq \mathbf{R}^n$ 和向量 $c \in \mathbf{R}^n \setminus \{0\}$，考虑如下的线性规划问题 P：（对最小化的情形也可以类似地讨论）

$$\max\{c^T x : x \in \mathcal{D}\}.$$

从几何角度看，这相当于在超平面 $H : c^T x = z$ 与可行域 $\mathcal{D}$ 至少有一个交点的前提下，沿着向量 c 的方向移动超平面 H，使得 z 尽可能大. 这就存在三种可能性：

- 可行域 $\mathcal{D}$ 是空集. 这说明问题 P 没有可行解，它的某些约束是相互矛盾的. 此时，称问题 P 是 **不可行的**（infeasible），它的最优价值规定为 $-\infty$.
- 可行域 $\mathcal{D}$ 非空，但是它包含一条方向向量 d 满足 $c^T d > 0$ 的射线，即存在 $x_0 \in \mathbf{R}^n$ 使得 $x_0 + td \in \mathcal{D}$ 对于所有 $t \geq 0$ 都成立. 因为沿着向量 c 的方向可以不断地移动超平面 H，而且移动过程中，集合 $H \cap \mathcal{D}$ 至少含有这条射线中的某个点，一定是非空的，所以，目标函数 $c^T x = c^T x_0 + t c^T d$ 可以取得任意大的值. 此时，称问题 P 是 **无界的**（unbounded），它的最优价值规定为 $+\infty$.
- 可行域 $\mathcal{D}$ 非空，且不含有任何这样的射线. 此时，问题 P 称为 **有界的**（bounded）. 记 $z^* \in \mathbf{R}$ 为问题 P 的最优价值. 超平面 $H^* : c^T x = z^*$ 处于一种临界位置：它与多面体 $\mathcal{D}$ 相交，且 $\mathcal{D}$ 包含于半空间 $\{x : c^T x \leq z^*\}$ 中. 这样的超平面称为多面体 $\mathcal{D}$ 的一个 **支撑超平面**（supporting hyperplane）. 问题 P 的最优解集就是 $H^* \cap \mathcal{D}$. 作为支撑超平面和多面体的交集，集合 $H^* \cap \mathcal{D}$ 一定是多面体，且包含在 $\mathcal{D}$ 的边界中. 它称为多面体 $\mathcal{D}$ 的一个 **面**（face）. 形象地说，多面体就是由这些面围成的. 除了这些由支撑超平面和多面体相交形成的面之外，一般来说，多面体还有两个面：空集和多面体本身. 多面体的所有面在集合的包含关系下，形成了 格→9.20@p1951 的结构.

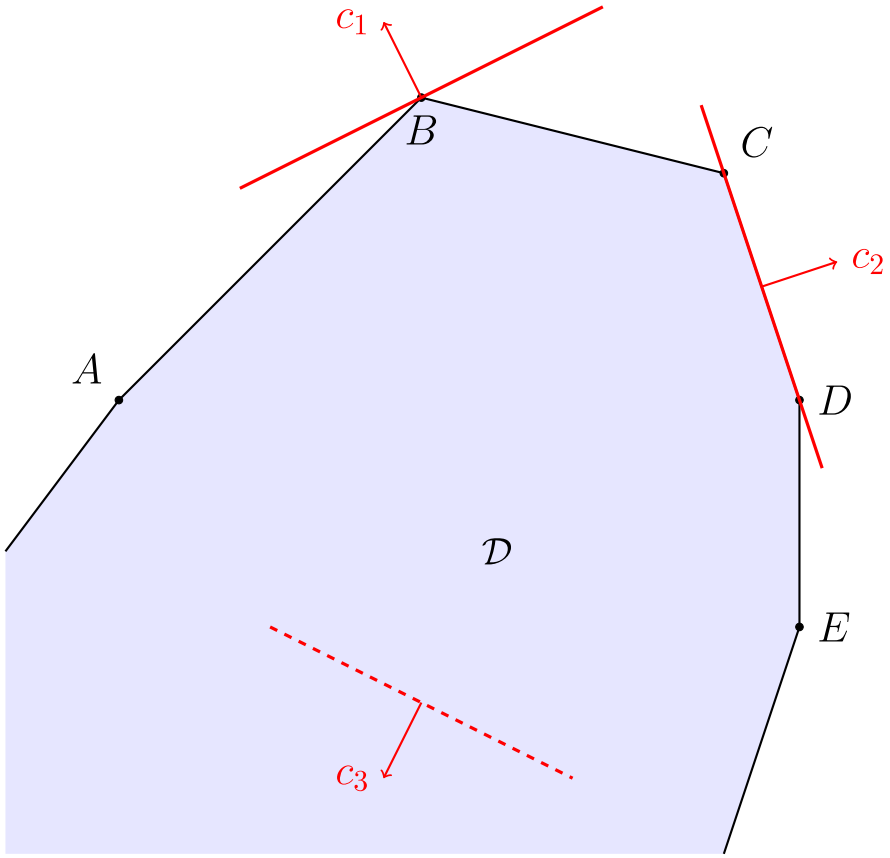
  一个 d 维的多面体的面的维度一定是 -1 和 d 之间的整数. 维度为 0 的面（即一个点）称为多面体 $\mathcal{D}$ 的 **顶点**（vertex）或 **角点**（corner point），维度为 1 的面称为多面体 $\mathcal{D}$ 的 **边**（edge），维度为 $d-1$ 的面则称为多面体 $\mathcal{D}$ 的 **维面**（facet）. 但是，并非所有多面体都有顶点. 因为多面体的面的面仍然是多面体的面，而只有仿射子空间才没有严格更小的非空面，所以，多面体 $\mathcal{D}$ 的所有极小面都是仿射子空间. 而且，同一个多面体的极小面的维度是相同的；特别地，多面体 $\mathcal{D} = \{x \in \mathbf{R}^n : Ax \leq b\}$ 的极小面的维度是 $n - \operatorname{rank} A$.

因为多面体的面就是有界线性规划问题的解集，所以，需要搞清楚如何确定多面体的面的方程．设多面体 $\mathcal{D}$ 由若干个约束 $a_j^T x \lessgtr b_j$ 描述，且 F 是 $\mathcal{D}$ 的一个面．如果某个约束在所有 $x \in F$ 处都取得等号，就称该约束在面 F 上是 **紧的**（tight）．面 F 上的点显然满足这些紧约束取等号得到的方程组，而这个方程组确定的仿射子空间和多面体 $\mathcal{D}$ 的交集，就是面 F．反过来，任意选取多面体 $\mathcal{D}$ 的约束的一个子集，将这些约束取等、联立、求解得到的仿射子空间和多面体的交集，就是 $\mathcal{D}$ 的一个面．而且，选取的紧约束越多，得到的面（在包含意义下）就越小．

特别地，标准形式的线性规划的可行域 $\mathcal{D} = \{x \in \mathbf{R}^n : Ax = b,\ x \geq 0\}$ 的系数矩阵 $\begin{pmatrix} A \\ I \end{pmatrix}$ 的秩是 n，因此，它的极小面就是它的顶点．也就是说，如果问题有界，那么它的最优解一定可以选取为某个顶点．而且，这个顶点可以通过选取 n 个线性独立的紧约束联立得到．这正是线性规划的标准形式的方便之处．

例子

下图中，$\mathcal{D}$ 为可行域．目标函数中的系数是 c_1, c_2, c_3 时，分别对应着唯一最优解、多组最优解和无界三种情形．对于前两种情形，相应的红色粗实线就是解集对应的支撑超平面（之一），最优解集分别是多面体 $\mathcal{D}$ 的顶点 B 和边 $\overline{CD}$．对于第三种情形，因为可行域 $\mathcal{D}$ 中包含方向为 c_3 的射线，所以，以 c_3 为法向量的超平面可以不断沿着 c_3 方向移动，进而问题是无界的．



这些讨论忽略了 $c = 0$ 的情形．此时，线性规划问题显然不能是无界的，所以要么问题本身是不可行的，要么最优价值等于 0，且最优解集就是 $\mathcal{D}$ 本身．这类特殊的线性规划也称为 **可行性线性规划**（feasibility linear programming）．

值得指出的是，判定线性规划问题是否可行、是否有界，以及求出不等式组的可行解等问题，都和解线性规划问题本身同样困难[2]．比如说，下文中强对偶定理的证明就说明，解一个有界的线性规划问题，就

[2]更严格的表述是，它们之间可以在多项式时间内相互归约．

相当于寻找一组不等式的可行解. 因此，对于判断不等式组是否有解和判断方程组是否有非负解等任务，最有效的方式就是求解相应的可行性线性规划[3].

另外，如果线性规划问题的一个约束，在可行域的所有面上都不是紧的，那么这个约束就是 **冗余的**（redundant）. 本文开头早点师傅的例子中，工作时间的约束就是一个冗余约束. 在给定的不等式组中判定某个不等式 $a_j^T x \le b_j$ 是否冗余这一问题，可以通过求解线性规划问题 $\max\{a_j^T x : x \in \mathcal{D}\}$，并将它与 b_j 相比较来解决.

常见算法

算法竞赛中，很少有问题只能通过线性规划的算法解决. 大多数可以用线性规划方法求解的题目，通常也可以通过网络流等更为专门也更为高效的算法来解决.

解决线性规划问题的常见算法如下：

- 单纯形法→9.15.2@p1787
- 椭球法
- 内点法

尽管单纯形法的最差情形复杂度是指数级的，而内点法的复杂度是多项式的，但这两类算法在大多数实际问题中的表现都非常出色. 相比之下，虽然椭球法的理论复杂度是多项式级别的，但是通常运行缓慢，并不实用.

目前尚不清楚线性规划问题是否存在强多项式复杂度的算法.

对偶问题

每个线性规划问题都对应着一个对偶问题. 原问题和对偶问题的解有着紧密的联系. 通过对偶问题，不仅有助于更深入地理解问题的结构，还常常可以提升原问题的求解效率.

对于线性规划问题 P（所涉小写字母变量均为向量）

$$\begin{aligned}
&\min_{x_1,x_2,x_3} \quad c_1^T x_1 + c_2^T x_2 + c_3^T x_3 \\
&\text{subject to } A_{11}x_1 + A_{12}x_2 + A_{13}x_3 \ge b_1, \\
&\qquad\qquad A_{21}x_1 + A_{22}x_2 + A_{23}x_3 = b_2, \\
&\qquad\qquad A_{31}x_1 + A_{32}x_2 + A_{33}x_3 \le b_3, \\
&\qquad\qquad x_1 \ge 0,\ x_3 \le 0,
\end{aligned}$$

它的对偶问题 D 是指线性规划问题

$$\begin{aligned}
&\max_{y_1,y_2,y_3} \quad b_1^T y_1 + b_2^T y_2 + b_3^T y_3 \\
&\text{subject to } A_{11}^T y_1 + A_{21}^T y_2 + A_{31}^T y_3 \le c_1, \\
&\qquad\qquad A_{12}^T y_1 + A_{22}^T y_2 + A_{32}^T y_3 = c_2, \\
&\qquad\qquad A_{13}^T y_1 + A_{23}^T y_2 + A_{33}^T y_3 \ge c_3, \\
&\qquad\qquad y_1 \ge 0,\ y_3 \le 0.
\end{aligned}$$

其中，对偶问题的决策变量 y_1, y_2, y_3 分别是原问题的三类约束的 Lagrange 乘子；反过来，原问题的决策变量 x_1, x_2, x_3 也分别是对偶问题的三类约束的 Lagrange 乘子. 容易验证，对偶问题的对偶问题就是原问题.

原问题 P 和对偶问题 D 的对应关系如下：

最小化问题	最大化问题
大于等于约束	非负变量

[3]其它用于解决不等式组的方法还包括 Fourier–Motzkin 消元法和 Agmon–Motzkin–Schoenberg 松弛法等. 它们更为直接，但是效率往往不高.

小于等于约束	非正变量
等式约束	无约束变量
非负变量	小于等于约束
非正变量	大于等于约束
无约束变量	等式约束
目标函数系数	约束右侧常量
约束右侧常量	目标函数系数

特别地，标准形式的线性规划问题

$$\min\{c^T x : Ax = b,\ x \geq 0\}$$

的对偶问题是

$$\max\{b^T y : A^T y \leq c\}.$$

对偶原理

原问题和对偶问题不仅在形式上互为镜像，而且两者的解也紧密相关．这称为 **对偶原理**（duality principle）．为表述方便，本节在叙述和证明定理时，将采用标准形式的原问题．

首先，**弱对偶定理**（weak duality theorem）说明，对偶问题的最大值不超过原问题的最小值．

弱对偶定理

对于所有 $A \in \mathbf{R}^{m\times n}$，$b \in \mathbf{R}^m$ 和 $c \in \mathbf{R}^n$，总有

$$\max\{b^T y : A^T y \leq c\} \leq \min\{c^T x : Ax = b,\ x \geq 0\}.$$

证明

如果原问题和对偶问题中的任何一个不可行，那么该不等式就是平凡的．假设两个问题都是可行的．那么，对于所有可行的 x 和 y，都有

$$b^T y = x^T A^T y \leq x^T c.$$

因此，将两侧取最值，就得到弱对偶定理成立．

要进一步讨论原问题和对偶问题的解之间的关系，还需要 Farkas 引理（和它的各种变体）．它给出了线性方程组存在非负解的充分必要条件．

Farkas 引理

对于 $A \in \mathbf{R}^{m\times n}$ 和 $b \in \mathbf{R}^m$，下列情形中恰有一种成立：

1. 存在 $x \in \mathbf{R}^n$，使得 $Ax = b$ 且 $x \geq 0$；
2. 存在 $y \in \mathbf{R}^m$，使得 $A^T y \geq 0$ 且 $b^T y < 0$.

证明

如果两种情形同时成立，那么就有 $0 \leq (A^T y)^T x = y^T (Ax) = b^T y < 0$，矛盾．所以，至多有一种情形成立．接下来说明至少有一种情形成立．记 $C = \{Ax : x \geq 0\}$ 为 A 的列向量生成的多面体锥．如果 $b \in C$，那么情形 1 成立；否则，需要说明情形 2 成立．因为多面体锥 C 也可以写成有限多个经过原点的半空间的交集，所以它是闭凸集．又因为 $b \notin C$，所以，由超平面分离定理可知，存在 $y \in \mathbf{R}^m$ 和 $\alpha \in \mathbf{R}$，使得 $b^T y < \alpha$，且对所有 $z \in C$ 都有 $z^T y > \alpha$．因为 $0 \in C$，所以 $\alpha < 0$，进而 $b^T y < 0$．又因为对所有 $x \geq 0$ 和 $t > 0$ 都有 $t(A^T y)^T x > \alpha$，令 $t \to +\infty$ 可知 $(A^T y)^T x \geq 0$ 对所有 $x \geq 0$ 都成立，即 $A^T y \geq 0$．这就是情形 2.

从几何角度看，情形 1 是在说，点 b 位于 A 的列向量生成的多面体锥 C 里；情形 2 则是在说，存在经过原点且法向量为 y 的超平面 $H: y^Tx=0$ 强分离了点 b 和多面体锥 C. 因此，Farkas 引理相当于一种超平面分离定理[1].

利用 Farkas 引理，就可以得到根据原问题和对偶问题的可行性判定原问题无界的方法.

推论

线性规划问题无界，当且仅当它可行，且它的对偶问题不可行.

证明

仍然以标准形式的原问题为例. 对于一般形式的线性规划问题, 只需要先将它转化为标准形式即可，因为 前文→9@p1079 所述的转化并不改变问题及其对偶问题的可行性和有界性.

一方面，如果原问题无界，那么它显然是可行的，且它的最优价值为 $-\infty$. 此时，对偶问题一定不可行，否则它的最优价值不小于某个实数，与弱对偶定理矛盾.

另一方面，设原问题可行，且对偶问题不可行. 将约束 $A^Ty\le c$ 中的自由变量写作两个非负变量的差 $y=y^+-y^-$，并添加非负松弛变量 s，它就等价于约束 $Mz=c$ 且 $z\ge 0$，其中，$M=(A^T,-A^T,I)$ 且 $z=(y^+,y^-,s)$. 因为 $M^Td=(Ad,-Ad,d)$，所以，对 M 和 c 应用 Farkas 引理就说明，要么存在 y 使得 $A^Ty\le c$，要么存在 $d\ge 0$ 使得 $Ad=0$ 且 $c^Td<0$，二者必择其一. 由假设可知，后者成立. 此时，取原问题的任意一组可行解 x_0，就有 $A(x_0+td)=b$ 且 $x_0+td\ge 0$ 对所有 $t\ge 0$ 都成立，而 $c^T(x_0+td)=c^Tx_0+tc^Td\to-\infty\ (t\to+\infty)$. 这说明原问题无界.

结合弱对偶定理和这一推论，原问题和对偶问题的解的情况只能有下面四种情形:

1. 原问题和对偶问题均不可行，即 $-\infty\le+\infty$;
2. 原问题不可行，对偶问题无界，即 $+\infty\le+\infty$;
3. 原问题无界，对偶问题不可行，即 $-\infty\le-\infty$;
4. 原问题和对偶问题均有界.

对于上述第四种情形，有更强的结论成立: 原问题和对偶问题的最优值是相等的. 将后三种情形合在一起，就得到 **强对偶定理**（strong duality theorem）: 只要原问题或对偶问题之一是可行的，它们的最优值就必然相等.

强对偶定理

对于所有 $A\in\mathbf{R}^{m\times n}$，$b\in\mathbf{R}^m$ 和 $c\in\mathbf{R}^n$，都有

$$\max\{b^Ty: A^Ty\le c\}=\min\{c^Tx: Ax=b,\ x\ge 0\}.$$

只要两个集合之一非空.

证明

前述讨论中，唯一没有确定最优价值的，就是原问题和对偶问题都可行的情形. 此时，考虑如下可行性线性规划问题 Q:

$$\max\{0: c^Tx\le b^Ty,\ Ax=b,\ x\ge 0,\ A^Ty\le c\}.$$

如果问题 Q 有可行解 $(x^*,y^*)\in\mathbf{R}^n\times\mathbf{R}^m$，那么，由弱对偶定理和最优性可知

$$b^Ty^*\le\max\{b^Ty: A^Ty\le c\}\le\min\{c^Tx: Ax=b,\ x\ge 0\}\le c^Tx^*,$$

但是 $c^Tx^*\le b^Ty^*$，故而所有这些不等式都可以取得等号，也就是说，不仅强对偶成立，而且 x^* 和 y^* 分别是原问题和对偶问题的最优解.

因此，只需要证明问题 Q 是可行的．假设不然．仿照上述推论的证明，可以考虑问题 Q 的对偶问题 DQ：

$$\min\{c^T\mu - b^T\lambda : ct - A^T\lambda \geq 0,\ -bt + A\mu = 0,\ t \geq 0,\ \mu \geq 0\}.$$

因为 $(t, \lambda, \mu) = (0, 0, 0)$ 是对偶问题 DQ 的一组可行解，所以由上述推论可知，问题 Q 不可行，就意味着对偶问题 DQ 无界，即存在一组 (t^*, λ^*, μ^*) 使得

$$c^T\mu^* - b^T\lambda^* < 0,\ ct^* - A^T\lambda^* \geq 0,\ -bt^* + A\mu^* = 0,\ t^* \geq 0,\ \mu^* \geq 0.$$

此时，如果 $t^* > 0$，那么这些不等式实际说明 $(x, y) = (\mu^*/t^*, \lambda^*/t^*)$ 是前述问题的一组可行解，与假设矛盾．所以，只能有 $t^* = 0$．这说明

$$c^T\mu^* < b^T\lambda^*,\ A^T\lambda^* \leq 0,\ A\mu^* = 0,\ \mu^* \geq 0.$$

但是，因为已经假设定理中的原问题和对偶问题都可行，也就是说，存在 (x_0, y_0) 使得

$$Ax_0 = b,\ x_0 \geq 0,\ A^Ty_0 \leq c$$

成立，所以，有

$$0 = (A\mu^*)^Ty_0 = (A^Ty_0)^T\mu^* \leq c^T\mu^* < b^T\lambda^* = x_0^TA^T\lambda^* \leq 0.$$

这显然矛盾．这一矛盾说明问题 Q 是可行的，进而说明强对偶成立．

从强对偶定理的证明过程还能得到如下推论：

推论

设原问题和对偶问题的一组可行解 x^* 和 y^* 满足强对偶性，即 $c^Tx^* = b^Ty^*$．那么，它们同样分别是原问题和对偶问题的最优解．

强对偶定理说明，对于可行的线性规划问题，只需要求解它的对偶问题，就能够得到原问题的最优价值．

互补松弛条件

和其它的优化问题一样，互补松弛条件是线性规划问题的最优性条件的一部分．而且，因为目标函数是线性的，所以对于线性规划问题来说，互补松弛条件是可行解成为最优解的充分必要条件．

所谓 **互补松弛**（complementary slackness）条件，就是指只有在原问题（对偶问题）中的约束取得等号（即约束是紧的）的时候，对偶问题（原问题）中与之对应的变量才能取非零值．如果将变量取非零值也当成一条松弛的约束，那么这就相当于说，原问题和对偶问题中相对应的变量和约束不能同时是松弛的．因此，这一条件称为互补松弛条件．

以标准形式的线性规划问题为例，如下结论成立：

定理

假设 x^* 和 y^* 分别是原问题 $\min\{c^Tx : Ax = b,\ x \geq 0\}$ 和对偶问题 $\max\{b^Ty : A^Ty \leq c\}$ 的可行解．那么，当且仅当互补松弛条件成立，即

$$(x^*)^T(A^Ty^* - c) = 0$$

时，x^* 和 y^* 也分别是原问题和对偶问题的最优解．

证明

因为 x^* 和 y^* 都是可行解，所以，有

$$b^Ty^* - c^Tx^* = (x^*)^T(A^Ty^* - c).$$

因此，互补松弛条件成立，当且仅当 $b^Ty^*=c^Tx^*$．根据强对偶定理的推论，这一条件成立，当且仅当 x^* 和 y^* 分别是原问题和对偶问题的最优解．

标准形式可能太过特殊．该定理的稍微一般的形式如下：

定理

假设 x^* 和 y^* 分别是原问题 $\min\{c^Tx : Ax\geq b,\ x\geq 0\}$ 和对偶问题 $\max\{b^Ty : A^Ty\leq c,\ y\geq 0\}$ 的可行解．那么，当且仅当互补松弛条件成立，即

$$(x^*)^T(A^Ty^*-c)=(y^*)^T(Ax^*-b)=0$$

时，x^* 和 y^* 也分别是原问题和对偶问题的最优解．

证明

证明基本同上，只是这次要将差值写成

$$b^Ty^*-c^Tx^*=(x^*)^T(A^Ty^*-c)-(y^*)^T(Ax^*-b).$$

互补松弛条件提供了判断线性规划问题的可行解的最优性的简单条件．

原始-对偶方法

对偶问题可以辅助原问题的求解．在解决线性规划问题时，常常会用到的一种方法是 **原始-对偶方法**（primal-dual method）．它通过求解一系列相对简单的辅助问题，逐步改进对偶问题的解，进而获得原问题的最优解．

对于标准形式的原问题

$$(P)\qquad \min\{c^Tx : Ax=b\geq 0,\ x\geq 0\}$$

和它的对偶问题

$$(D)\qquad \max\{b^Ty : A^Ty\leq c\},$$

上一节已经说明，要找到它们的最优解，只需要找到问题 (P) 和 (D) 的一组可行解，使得它们满足互补松弛条件 $x^T(A^Ty-c)=0$．那不妨考虑如下流程：

1. 从对偶问题 (D) 的一个可行解 y 出发，计算对偶问题的紧约束的集合
   $I=\{i : (A^Ty-c)_i=0\}$.
2. 根据互补松弛条件，如果存在问题 (P) 的可行解 x 使得 $x_i>0$ 仅在 $i\in I$ 上成立，就意味着已经找到一组最优解．因此，考虑线性规划问题

$$(RP)\qquad \begin{aligned}&\min_{x,s}\ 1^Ts\\ &\text{subject to } Ax+s=b,\\ &\qquad x_i\geq 0,\ \forall i\in I,\\ &\qquad x_i=0,\ \forall i\notin I,\\ &\qquad s\geq 0.\end{aligned}$$

3. 如果问题 (RP) 的最小值是 0，那么最优解 $(x^*,0)$ 中的 x^* 就是原问题 (P) 的最优解．否则，可以求出它的对偶问题 (DRP) 的解 $\bar{y}$：

$$(DRP)\qquad \begin{aligned}&\max_{y}\ b^Ty\\ &\text{subject to}\sum_j a_{ji}y_j\leq 0,\ \forall i\in I,\\ &\qquad y\leq 1.\end{aligned}$$

根据强对偶定理可知，$b^T\bar{y}=1^Ts^*>0$.

4. 根据问题 (DRP) 的解改进对偶问题 (D) 的可行解. 设 $y'=y+\varepsilon\bar{y}$, 其中, $\varepsilon>0$, 则一定有 $b^Ty'=b^Ty+\varepsilon b^T\bar{y}>b^Ty$. 因此, 只要保证 y' 仍然是对偶问题 (D) 的可行解, 就要尽可能大地选取 ε 的值.

   对于 $i\in I$, 有

   $$\sum_j a_{ji}y'_j=\sum_j a_{ji}y_j+\varepsilon\sum_j a_{ji}\bar{y}_j\le c_i,$$

   所以, 问题 (D) 的这些约束总是可以满足的.

   对于剩下的约束, 即 $i\notin I$ 时, 只需要取

   $$\varepsilon=\min\left\{\frac{c_i-\sum_j a_{ji}y_j}{\sum_j a_{ji}\bar{y}_j}:i\notin I,\ \sum_j a_{ji}\bar{y}_j>0\right\}$$

   就可以在保证可行性的前提下, 尽可能大地改进对偶问题的解, 然后回到步骤 1 继续迭代. 特别地, 如果上式中的集合为空集, 即 $\varepsilon=+\infty$, 那么, 对偶问题 (D) 无界, 原问题 (P) 不可行.

这个过程中其实只有问题 (DRP) 是确实需要求解的, 它与问题 (RP) 通过强对偶定理相互联系. 问题 (DRP) 提供了一个改进对偶问题解的方向, 而且相对于对偶问题 (D) 本身, 问题 (DRP) 的形式更加简单. 问题 (DRP) 总是可行的 ($y=0$ 就是一组可行解), 而约束 $y\le 1$ 只是一组规范化条件, 保证了问题 (DRP) 有界.

算法竞赛中, 原始-对偶方法广泛地应用于各类组合优化问题. 例如二分图最大权匹配的 匈牙利算法→11.24.3@p2983、最小费用流的 消圈算法→11.23.4@p2935 和 SSP 算法 (原始-对偶算法) →11.23.4@p2935、最短路的 Dijkstra 算法→11.9.1@p2735、最大流的 Ford–Fulkerson 增广算法→11.23.2@p2907 等, 都可以看作是原始-对偶方法的直接应用.

整数规划

整数规划 (integer programming) 通常指 **整数线性规划** (integer linear programming, ILP). 标准形式的整数线性规划如下:

$$\begin{aligned}\min_x\ & c^Tx\\ \text{subject to }\ & Ax=b\ge 0,\\ & x\ge 0,\\ & x\in\mathbf{Z}^n,\end{aligned}$$

其中, $A\in\mathbf{R}^{m\times n}$, $b\in\mathbf{R}^m$, $c\in\mathbf{R}^n$. 也就是说, 整数线性规划是在线性规划问题上添加决策变量必须为整数这一约束条件所得到的问题.

整数约束显著增加了整数规划问题的复杂性. 许多组合优化问题, 例如背包问题、可满足性问题以及众多图论中的优化问题, 都可以表示为整数规划模型, 而这些问题中的多数被证明是 NP 困难的.

全幺模矩阵

正因如此, 对于很多大规模的整数优化问题, 有时候会考虑将它的整数约束松弛掉, 转而求解一个线性规划问题. 通常来说, 松弛后的线性规划问题的最优价值只是原来的整数规划问题的一个下界估计 (假设问题是最小化问题). 但是, 如果松弛后的线性规划问题的最优解恰好是整数解, 那么, 它也一定是原来的整数规划问题的最优解.

一个自然的问题是, 是否存在条件, 能够保证线性规划问题的最优解都是整数解? 全幺模矩阵的概念就提供了这样的一个条件.

自 全幺模矩阵

如果矩阵 $A\in\mathbf{R}^{m\times n}$ 的所有子方阵的行列式都是 0 或 ±1, 那么, 矩阵 A 就称为一个 **全幺模矩阵** (totally unimodular matrix).

特别地，全幺模矩阵的所有元素都是 0 或 ± 1．利用全幺模矩阵的概念，可以叙述如下结论：

定理

对于全幺模矩阵 $A \in \mathbf{Z}^{m \times n}$，$b \in \mathbf{Z}^m$ 且 $c \in \mathbf{Z}^n$，线性规划问题及其对偶问题

$$\min\{c^T x : Ax = b, x \geq 0\} = \max\{b^T y : A^T y \leq c\}$$

都有整数最优解，只要它们都有界．

证明

前文已经说明，线性规划问题的最优解集可以取作它的一个极小面，而后者是由若干线性独立的紧约束作为等式联立得到的方程组的解：

$$\{x \in \mathbf{R}^n : a_j^T x = b_j,\ \forall j \in J\}.$$

记这个方程组为 $A_J x = b_J$，且 $A_J = (A_1, A_2)$，其中，A_1 是满秩的方阵，行列式为 ± 1．那么，由 Cramer 法则，解

$$x = \begin{pmatrix} A_1^{-1} b_J \\ 0 \end{pmatrix}$$

就是极小面上的一个整数解．

常见的图论模型中，网络流、最短路、二分图等对应的线性规划问题的系数矩阵都是全幺模矩阵．因此，只需要这些问题仅涉及整数参数，它们的最优解就可以取作整数，而不用担心线性规划问题的解对应着分数流、分数匹配等情形．所以，最大流→11.23.2@p2907、最小割→11.23.3@p2932、最小费用流→11.23.4@p2935、最短路→11.9.1@p2735、差分约束→11.9.2@p2747、二分图最大（权）匹配和最小点覆盖→11.24.2@p2959 等问题，都可以转化为线性规划问题求解．而且，最大流与最小割、最短路与差分约束、二分图最大匹配和最小点覆盖，两两互为对偶问题．

除此之外，还有一些常见的图论模型，它所有的可行解恰巧是某个顶点均为整点的多胞形的全体顶点．因此，可以通过巧妙地选取约束，使得相应的组合优化问题的解，恰为某个线性规划问题的最优解．例如，一般图匹配和生成树等图论模型都属于这种情况，因此 一般图最大（权）匹配→11.24.5@p3012 和 最小生成树→11.10.1@p2764 等问题同样可以转化为线性规划问题．

参考文献与注释

- Schrijver, Alexander. Theory of linear and integer programming. John Wiley & Sons, 1998.
- Papadimitriou, Christos H., and Kenneth Steiglitz. Combinatorial optimization: algorithms and complexity. Courier Corporation, 1998.
- Duality in linear programming. Part 1—definition and construction. by adamant - Codeforces blog[2]
- Duality in linear programming. Part 2—in competitive programming. by adamant - Codeforces blog[3]

[1] 超平面分离定理

[2] Duality in linear programming. Part 1—definition and construction. by adamant - Codeforces blog

[3] Duality in linear programming. Part 2—in competitive programming. by adamant - Codeforces blog

9.15.2 单纯形法

前置知识：线性规划基础→9.15.1@p1774

引入

算法竞赛中，经常使用单纯形法解决线性规划问题．但是，由于算法竞赛中遇到的线性规划问题大多有着更特殊的结构，常常可以转化为网络流问题，因此，单纯形法并不常用，效率也不如专门为网络流问题设计的算法．

基本概念

假设要求解如下一个有 n 个决策变量和 $m+n$ 个约束的 标准形式→9.15.1@p1774 线性规划问题：

$$\begin{aligned}\min_x \ & z = c^T x \\ \text{subject to } & Ax = b, \\ & x \geq 0.\end{aligned}$$

不妨假设这 m 个等式约束确定的线性方程组有解，且 A 满秩，则 $\operatorname{rank} A = m \leq n$.

一个例子

在严格叙述单纯形法的步骤之前，本节首先考察一个具体的例子，以方便理解.

例子

考虑线性规划问题

$$\begin{aligned}\max \ & 10x_1 + 12x_2 + 12x_3 \\ \text{subject to } & x_1 + 2x_2 + 2x_3 \leq 20, \\ & 2x_1 + x_2 + 2x_3 \leq 20, \\ & 2x_1 + 2x_2 + x_3 \leq 20, \\ & x_1, x_2, x_3 \geq 0.\end{aligned}$$

通过添加松弛变量，就得到它的标准形式：

$$\begin{aligned}\min \ & -10x_1 - 12x_2 - 12x_3 \\ \text{subject to } & x_1 + 2x_2 + 2x_3 + x_4 = 20, \\ & 2x_1 + x_2 + 2x_3 + x_5 = 20, \\ & 2x_1 + 2x_2 + x_3 + x_6 = 20, \\ & x_1, x_2, x_3, x_4, x_5, x_6 \geq 0.\end{aligned}$$

观察该问题的等式约束，它们其实相当于将变量 x_4, x_5, x_6 由变量 x_1, x_2, x_3 表示. 将原问题稍微整理一下，就有

$$\begin{array}{rrrrrr} \min_{x_i \geq 0} & z = & 0 & -10x_1 & -12x_2 & -12x_3 \\ \text{subject to} & x_4 = & 20 & -x_1 & -2x_2 & -2x_3, \\ & x_5 = & 20 & -2x_1 & -x_2 & -2x_3, \\ & x_6 = & 20 & -2x_1 & -2x_2 & -x_3. \end{array}$$

从这个形式中，可以清楚地看到，如果令 $x_1 = x_2 = x_3 = 0$，就可以得到原问题的一组可行解

$$x = (0, 0, 0, 20, 20, 20)^T.$$

且它对应的价值为 $z = 0$. 为方便叙述，称那些设为零的变量 x_1, x_2, x_3 为非基变量，剩下的变量 x_4, x_5, x_6 为基变量.

这组可行解显然不是最优解. 只要适当地增加 x_1, x_2, x_3 的值，使得 x_4, x_5, x_6 仍然是非负数，就可以保持解仍然可行. 而且，因为目标函数中，x_1, x_2, x_3 的系数都是严格的负数，所以，增加它们的值一定会降低目标函数的值. 比如说，可以选择增加 x_1 的值. 为了尽可能多地降低目标函数的值，需要尽可能多地增加 x_1 的值. 但是，为了保证解仍然是可行的，就需要保证 $x_4, x_5, x_6 \geq 0$. 因此，x_1 最多可以增加到

$$\min\left\{\frac{20}{1}, \frac{20}{2}, \frac{20}{2}\right\} = 10.$$

此时，可行解变为

$$x = (10, 0, 0, 10, 0, 0)^T.$$

因为 x_1 成为了基变量，为了回到最初的情形（即三个基变量由三个非基变量表示），需要选择一个新的非基变量．因为 x_5, x_6 都是零，所以，可以选择它们其中的任何一个作为非基变量，设为零．不妨选择 x_5 作为非基变量．而且，将

$$x_1 = 10 - 0.5x_5 - 0.5x_2 - x_3$$

代入到原来的问题中，就可以将原问题改写作

$$\begin{array}{rrrrrr} \min_{x_i \geq 0} & z = & -100 & +5x_5 & -7x_2 & -2x_3 \\ \text{subject to} & x_4 = & 10 & +0.5x_5 & -1.5x_2 & -x_3, \\ & x_1 = & 10 & -0.5x_5 & -0.5x_2 & -x_3, \\ & x_6 = & 0 & +x_5 & -x_2 & +x_3. \end{array}$$

这就回到了初始的情形．

继续观察当前的目标函数．非基变量 x_3 的系数仍然是负数．可以考虑增加 x_3 的值．为了保证 $x_4, x_1, x_6 \geq 0$，变量 x_3 最多只能增加到

$$\min\left\{\frac{10}{1}, \frac{10}{1}\right\} = 10.$$

注意到，因为 x_6 的表达式中，x_3 的系数是正数，所以，无论怎么增加 x_3 的值，都不会使得 x_6 变为负数．这就是这次大括号中只有两项的原因．因为 x_3 增加到 10 的时候，x_1, x_4 都变为零，所以，可以任选其中一个作为新的非基变量．不妨选择 x_4．就可以将

$$x_3 = 10 + 0.5x_5 - 1.5x_2 - x_4$$

代入上述问题中，问题变形为

$$\begin{array}{rrrrrr} \min_{x_i \geq 0} & z = & -120 & +4x_5 & -4x_2 & +2x_4 \\ \text{subject to} & x_3 = & 10 & +0.5x_5 & -1.5x_2 & -x_4, \\ & x_1 = & 0 & -x_5 & +x_2 & +x_4, \\ & x_6 = & 10 & +1.5x_5 & -2.5x_2 & -x_4. \end{array}$$

只需要代入 $x_5 = x_2 = x_4 = 0$，就能从这个形式中读出当前的可行解是

$$x = (0, 0, 10, 0, 0, 10)^T,$$

以及它对应的价值为 $z = -120$.

重复之前的操作．因为 x_2 的系数是负数，可以增加它的值；但为了保持 x_3, x_6 仍然是非负数，只能增加到

$$\min\left\{\frac{10}{1.5}, \frac{10}{2.5}\right\} = 4.$$

因为大括号中的最小值出现在变量 x_6 的表达式中，所以，它将在 $x_2 = 4$ 时变为零．将表达式

$$x_2 = 4 + 0.6x_5 - 0.4x_6 - 0.4x_4$$

代入上述问题，就可以将原问题改写为

$$\begin{array}{rrrrrr} \min_{x_i \geq 0} & z = & -136 & +1.6x_5 & +1.6x_6 & +3.6x_4 \\ \text{subject to} & x_3 = & 4 & -0.4x_5 & +0.6x_6 & -0.4x_4, \\ & x_1 = & 4 & -0.4x_5 & -0.4x_6 & +0.6x_4, \\ & x_2 = & 4 & +0.6x_5 & -0.4x_6 & -0.4x_4. \end{array}$$

仍然令非基变量 x_5, x_6, x_4 为零，就可以得到当前的可行解为

$$x = (4, 4, 4, 0, 0, 0)^T.$$

对应的价值为 $z = -136$.

因为目标函数中所有非基变量的系数都是正数，无法继续前文所述过程以改进目标函数，所以，当前的可行解就是最优解．算法终止．

这个例子中，算法从一组可行解出发，不断地改进目标函数，直到无法继续改进．这就是单纯形法的基本思想．

基本可行解

由于 A 是满秩的，所以，总是可以选取大小为 m 的子集 $B \subseteq \{1,2,\cdots,n\}$ 使得 A_B 是可逆方阵．由此，可以将 x_B 由剩下的变量 x_N 表示：

$$x_B = A_B^{-1}b - A_B^{-1}A_N x_N.$$

其中，$N = \{1,2,\cdots,n\} \setminus B$，矩阵 A_B, A_N 分别为矩阵 A 中标号 $i \in B$ 和标号 $i \in N$ 的列组成的子矩阵，向量 x_B, x_N 分别为向量 x 中标号 $i \in B$ 和标号 $i \in N$ 的分量组成的子向量．如果 $i \in B$，那么，称 x_i 为 **基变量**（basic variable）；否则，称 x_i 为 **非基变量**（non-basic variable）．基变量的全体称为一组 **基**（basis），本文用对应的标号集合 B 表示一组基．

> **「基」**
>
> 「基」这个名称，可以从线性代数的角度理解．设 A 的全体列向量张成的线性空间为 V．那么，基 B 对应的列向量就是空间 V 的一组基．

在基变量 x_B 的表达式中，令 $x_N = 0$，就得到全体等式约束的一组解[4]

$$x = (x_B, x_N) = (A_B^{-1}b, 0).$$

这样得到的解称为线性规划问题的一个 **基本解**（basic solution）．如果它还满足所有非负约束，即 $x \geq 0$，那么，它也是原问题的一个可行解，也称为 **基本可行解**（basic feasible solution, BFS）．在单纯形法的迭代过程中，需要始终保持当前的解为一组基本可行解．

转轴

单纯形法的每次迭代就称为一次 **转轴**（pivoting）．从结果上看，每次转轴总是移除一个旧的基变量，再添加一个新的基变量，进而改进目标函数的值．

> **「转轴」**
>
> 「转轴」这个名称，同样可以从线性代数的角度理解．如上文所述，基 B 对应的列向量是空间 V 的一组基，它们也就对应着对应基的表示下空间 V 的一组坐标轴．因此，转轴的过程，就是将某条坐标轴旋转到新的位置的过程．

为了确定需要添加的基变量，可以将目标函数利用非基变量表示为

$$\begin{aligned} c^T x &= c_B^T x_B + c_N^T x_N \\ &= c_B^T A_B^{-1} b + (c_N^T - c_B^T A_B^{-1} A_N) x_N. \end{aligned}$$

令 $x_N = 0$，就得到目标函数在当前基本可行解处的价值 $z = c_B^T A_B^{-1} b$．表达式中的第二项的系数则表示 x_N 改变时，目标函数的改变为

$$\tilde{c}_N = \frac{\partial z}{\partial x_N} = c_N - A_N^T (A_B^{-1})^T c_B.$$

注意到 $c_B - A_B^T (A_B^{-1})^T c_B = 0$，所以，可以记向量

$$\tilde{c} = (\tilde{c}_B^T, \tilde{c}_N^T)^T = c - A^T (A_B^{-1})^T c_B$$

为线性规划问题在基本可行解 x 处的 **约化成本**（reduced cost）．分量 $\tilde{c}_i < 0$ 说明增加变量 x_i 的值可以改进原问题的目标函数. 这样的变量只能是一个非基变量, 它称为本次转轴的 **入基变量**（entering variable）．因为在转轴后，x_i 将变为基变量，不再恒设为零（但依然有可能等于零）．

选择完入基变量后，还需要选择需要移除的旧的基变量．为此，只需要确定，在增加 x_i 的过程中，哪个现有的基变量最先变为零．将 $x_N = (x_i, x_{N\setminus\{i\}}) = (x_i, 0)$ 代入 x_B 的表达式，就有

$$x_B = A_B^{-1}b - A_B^{-1}A_i x_i.$$

[4]原则上，因为所有向量默认为列向量，(x_B, x_N) 应该写作 $(x_B^T, x_N^T)^T$．但是为了简化记号，本文所有类似的情形都直接写作 (x_B, x_N)，而省略转置符号．

因此，x_i 可以增加的最大量就等于

$$\theta = \min\left\{ \frac{(A_B^{-1}b)_j}{(A_B^{-1}A_i)_j} : (A_B^{-1}A_i)_j > 0 \right\}.$$

最先变为零的变量就是使得该表达式取得最小值的下标 j 对应的基变量 x_{B_j}．它也是增加 x_i 的过程中的「瓶颈」——继续增加 x_i 将使得 x_{B_j} 成为负值．这一变量就是本次转轴的 **出基变量**（leaving variable）．确定出基变量的方法称为 **最小比值检验**（minimum ratio test）．

设入基变量为 x_i，出基变量为 $x_{i'}$．转轴之后，基变量就是 $x_{B\setminus\{i'\}\cup\{i\}}$，非基变量就是 $x_{N\setminus\{i\}\cup\{i'\}}$．

终止条件

单纯形法，就是从一组基本可行解出发，不断进行转轴的过程．上一节对转轴的讨论并不是完整的，它忽略了一些特殊的情形．有些特殊情形对应着算法的终止，有些则需要额外的处理．

首先，入基变量未必存在，即 $\tilde{c} \geq 0$．此时，没有进一步改进最优价值的方法，这说明当前基本可行解就是最优解，算法终止．要严格地证明这一点，需要用到 互补松弛条件→9.15.1@p1774．令 $y = (A_B^{-1})^T c_B$．注意到，在算法的整个过程中，始终保持 x 是可行解，且互补松弛条件成立，即

$$x^T(c - A^T y) = \tilde{c}^T x = \tilde{c}_B^T x_B + \tilde{c}_N^T x_N = 0.$$

因此，只需要 y 是对偶问题的可行解，即 $A^T y \leq c$，就能得到 x 和 y 分别是原问题和对偶问题的最优解这一结论．这个条件就是 $\tilde{c} \geq 0$，即不存在入基变量．

> **「影子价格」**
>
> 向量 $y = (A_B^{-1})^T c_B$ 常称为 **对偶向量**（dual vector）．当 B 对应的基本可行解是原问题的最优解时，向量 y 是对偶问题的最优解．因此，利用单纯形法求解原问题的最优解时，也会获得对偶问题的最优解．因为向量 y 是当前价值关于约束常量的偏导数，即
>
> $$\frac{\partial(c^T x)}{\partial b} = (A_B^{-1})^T c_B = y,$$
>
> 它也称为 **影子价格**（shadow price）．

其次，出基变量未必存在，即 $A_B^{-1}A_i \leq 0$．此时，转轴的过程不存在任何「瓶颈」，也就是说，可以不断地通过增加 x_i 的值来改进目标函数，使它趋于 $-\infty$．这说明给定的线性规划问题是无界的，算法终止．

最后，入基变量和出基变量的选择可能并非是唯一的．不恰当的选择方式可能会导致过多地转轴，甚至使得算法陷入循环，无法正常终止．这类情形的处理稍微复杂，需要应用一些 转轴规则→9@p1079 以防止陷入循环并减少转轴次数．

单纯形表

具体实现转轴过程时，只需要维护每次转轴之后，线性规划问题的系数矩阵：

$$\tilde{T}_B = \begin{pmatrix} -z_B & \tilde{c}_N^T \\ x_B & A_B^{-1}A_N \end{pmatrix} = \begin{pmatrix} -c_B^T A_B^{-1} b & c_N^T - c_B^T A_B^{-1} A_N \\ A_B^{-1} b & A_B^{-1} A_N \end{pmatrix}.$$

它对应着线性规划问题：

$$\begin{aligned} \min_{x\geq 0} \quad & c_B^T A_B^{-1} b & + \tilde{c}_N^T x_N \\ \text{subject to} \quad x_B = \quad & A_B^{-1} b & - A_B^{-1} A_N x_N. \end{aligned}$$

矩阵 $\tilde{T}_B$ 称为该线性规划问题相对于基 B 的 **压缩单纯形表**（condensed simplex tableau）．表的左上角 $(\tilde{T}_B)_{00}$ 为当前解的价值（的相反数），第 0 行、第 i 列的量 $(\tilde{T}_B)_{0i}$ 为第 i 个非基变量 x_{N_i} 的约化成本，第 j 行、第 0 列的量 $(\tilde{T}_B)_{j0}$ 为第 j 个基变量 x_{B_j} 的值，而 $A_B^{-1}A_N$ 就是用非基变量 x_N 表示基变量 x_B 得到的表达式中的系数．

容易看出，转轴需要的所有信息都可以从压缩单纯形表中直接获得．具体地，利用压缩单纯形表，单次转轴包括如下操作：

1. 选取列 $i = 1, \cdots, n-m$，使得 $(\tilde{T}_B)_{0i} < 0$. 如果不存在这样的 i，那么当前解就是最优解，量 $-(\tilde{T}_B)_{00}$ 就是最优价值.
2. 选取行 $j = 1, \cdots, m$，使得 $(\tilde{T}_B)_{ji} > 0$ 且 $(\tilde{T}_B)_{j0}/(\tilde{T}_B)_{ji}$ 最小. 如果不存在这样的 j，那么原问题无界.
3. 令变量 x_{N_i} 入基，变量 x_{B_j} 出基，并更新单纯形表.

现在，具体地讨论一下如何更新单纯形表. 在更新单纯形表之前，第 j 行表示等式

$$x_{B_j} = (\tilde{T}_B)_{j0} - \sum_{i=1}^{n-m} (\tilde{T}_B)_{ji} x_{N_i}.$$

为了更新单纯形表，需要用 $x_{N \setminus \{N_i\} \cup \{B_j\}}$ 表示 x_{N_i}，也就是：

$$x_{N_i} = \frac{(\tilde{T}_B)_{j0}}{(\tilde{T}_B)_{ji}} - \frac{1}{(\tilde{T}_B)_{ji}} x_{B_j} - \sum_{i' \neq i} \frac{(\tilde{T}_B)_{ji'}}{(\tilde{T}_B)_{ji}} x_{N_{i'}}.$$

将它代入其余的式子中，就得到

$$x_{B_{j'}} = \left((\tilde{T}_B)_{j'0} - (\tilde{T}_B)_{j'i} \frac{(\tilde{T}_B)_{j0}}{(\tilde{T}_B)_{ji}} \right) + \frac{(\tilde{T}_B)_{j'i}}{(\tilde{T}_B)_{ji}} x_{B_j} - \sum_{i' \neq i} \left((\tilde{T}_B)_{j'i'} - (\tilde{T}_B)_{j'i} \frac{(\tilde{T}_B)_{ji'}}{(\tilde{T}_B)_{ji}} \right) x_{N_{i'}}.$$

第 0 行类似，只是等式左侧变为 $-z_B$. 虽然式子看起来复杂，但是实现时，只需要分两步：

1. 更新第 j 行，即令 $\alpha = (\tilde{T}_B)_{ji}$，再令第 i 列数字为 1，然后将整行所有数字同除以 α；
2. 更新第 $j' \neq j$ 行，即令 $\beta = (\tilde{T}_B)_{j'i}$，再令第 i 列数字为 0，然后将整行数字同时减去 β 倍的第 j 行数字.

「单纯形表」

单纯形表（simplex tableau）指矩阵

$$T_B = \begin{pmatrix} -z_B & \tilde{c}^T \\ x_B & A_B^{-1} A \end{pmatrix} = \begin{pmatrix} -c_B^T A_B^{-1} b & c^T - c_B^T A_B^{-1} A \\ A_B^{-1} b & A_B^{-1} A \end{pmatrix}.$$

相较于压缩单纯形表，它多了 m 列，分别对应 m 个基变量；而且，对应着第 j 个基变量的列一定是 e_j，即该向量只在第 j 行处取值为 1，其余行均为 0. 因为这些列并没有提供多余的信息，所以在实现单纯形法时，常常省略这些列，这就得到了压缩单纯形表.

利用单纯形表可以更方便地理解更新单纯形表的步骤. 因为所有的单纯形表 T_B 都可以由同一个矩阵 T_0 左乘一个与基有关的可逆矩阵 L_B 得到，即

$$T_B = \begin{pmatrix} -c_B^T A_B^{-1} b & c^T - c_B^T A_B^{-1} A \\ A_B^{-1} b & A_B^{-1} A \end{pmatrix} = \begin{pmatrix} 1 & -c_B^T A_B^{-1} \\ O & A_B^{-1} \end{pmatrix} \begin{pmatrix} 0 & c^T \\ b & A \end{pmatrix} = L_B T_0,$$

所以，这些单纯形表和 T_0 之间可以通过若干次 初等行变换→9.14.5@p1725 相互转化. 因此，在更新单纯形表时，只需要施行初等行变换，使得入基变量对应列变为 e_j 即可. 由此，所需要的操作迁移到压缩单纯形表上，就是上文给出的步骤.

更新压缩单纯形表的参考实现如下：

参考实现

```
// Pivot on (N[x], B[y]).
void pivot(int x, int y) {
  std::swap(N[x], B[y]);
  long double v = -1 / tab[x][y];
  for (int j = 0; j <= m + 1; ++j) {
    tab[x][j] = j == y ? -v : v * tab[x][j];
  }
  for (int i = 0; i <= n; ++i) {
```

```
 9     if (i == x) continue;
10     v = tab[i][y];
11     tab[i][y] = 0;
12     for (int j = 0; j <= m + 1; ++j) {
13       tab[i][j] += v * tab[x][j];
14     }
15   }
16 }
17
```

由该实现可知, 单次更新单纯形表的时间复杂度是 $O(mn)$ 的. 稍后讨论 转轴规则→9@p1079 时会说明, 确定出基变量和入基变量的复杂度同样不会超过 $O(mn)$, 因此, 单次转轴的时间复杂度就是 $O(mn)$ 的.

为方便理解, 此处列出前文所示例子中, 利用压缩单纯形表计算的详细步骤.

例子（续）

初始时, 压缩单纯形表如下所示:

$$\begin{array}{rcccc} & & x_1 & x_2 & x_3 \\ & 0 & -10 & -12 & -12 \\ x_4 = & 20 & 1 & 2 & 2 \\ x_5 = & 20 & 2 & 1 & 2 \\ x_6 = & 20 & 2 & 2 & 1 \end{array}$$

根据第 0 行的约化成本, 可以选择 x_1, x_2, x_3 入基. 令 x_1 入基. 再根据最小比值检验, 可以选择 x_5, x_6 出基. 令 x_5 出基. 相应地, 更新压缩单纯形表如下:

$$\begin{array}{rcccc} & & x_5 & x_2 & x_3 \\ & 100 & 5 & -7 & -2 \\ x_4 = & 10 & -0.5 & 1.5 & 1 \\ x_1 = & 10 & 0.5 & 0.5 & 1 \\ x_6 = & 0 & -1 & 1 & -1 \end{array}$$

根据第 0 行的约化成本, 可以选择 x_2, x_3 入基. 令 x_3 入基. 再根据最小比值检验, 可以选择 x_4, x_1 出基. 令 x_4 出基. 相应地, 更新压缩单纯形表如下:

$$\begin{array}{rcccc} & & x_5 & x_2 & x_4 \\ & 120 & 4 & -4 & 2 \\ x_3 = & 10 & -0.5 & 1.5 & 1 \\ x_1 = & 0 & 1 & -1 & -1 \\ x_6 = & 10 & -1.5 & 2.5 & 1 \end{array}$$

根据第 0 行的约化成本, 只能选择 x_2 入基. 令 x_2 入基. 再根据最小比值检验, 只能选择 x_6 出基. 令 x_6 出基. 利用前述初等行变换更新单纯形表如下:

$$\begin{array}{rcccc} & & x_5 & x_6 & x_4 \\ & 136 & 1.6 & 1.6 & 3.6 \\ x_3 = & 4 & 0.4 & -0.6 & 0.4 \\ x_1 = & 4 & 0.4 & 0.4 & -0.6 \\ x_2 = & 4 & -0.6 & 0.4 & 0.4 \end{array}$$

根据第 0 行的约化成本, 不存在入基变量. 因此, 当前解

$$x = (4, 4, 4, 0, 0, 0)^T$$

就是最优解, (最小化问题的) 最优价值为 -136.

除了利用单纯形表实现单纯形法之外, 还可以使用修正单纯形法 (revised simplex method), 它进一步改进了算法的时空复杂度, 将单次更新的复杂度进一步降低到了 $O(m^2)$, 在 $m \ll n$ 或 A 是稀疏矩阵的情形下尤为高效.

几何背景

本节介绍单纯形法的几何背景.

对线性规划问题的可行域

$$\mathcal{D} = \{x \in \mathbf{R}^n : Ax = b,\ x \geq 0\}$$

的 分析→9.15.1@p1774 指出:

- 线性规划问题的最优解（如果存在）必然可以选取为可行域 $\mathcal{D}$ 的顶点. 求解线性规划问题，就转化为在所有顶点解里找到价值函数最优的那个.
- 每一个顶点的坐标，都可以通过 n 个紧约束联立得到的方程组求解得到. 对于标准形式的约束，所有 m 个等式约束一定是紧的，剩下的 $n-m$ 个约束只能从非负约束中选取. 选取这些非负约束作为紧约束，就相当于将相应的决策变量 x_N 设为 0；相应地，方程组 $Ax = b$ 退化为关于剩余 m 个决策变量 x_B 的线性方程组 $A_B x_B = b$. 只要 A_B 可逆，就可以解得 $x_B = A_B^{-1} b$. 这就得到了一个解 $(x_B, x_N) = (A_B^{-1} b, 0)$；如果 $x_B \geq 0$，这就是 $\mathcal{D}$ 的一个顶点坐标.

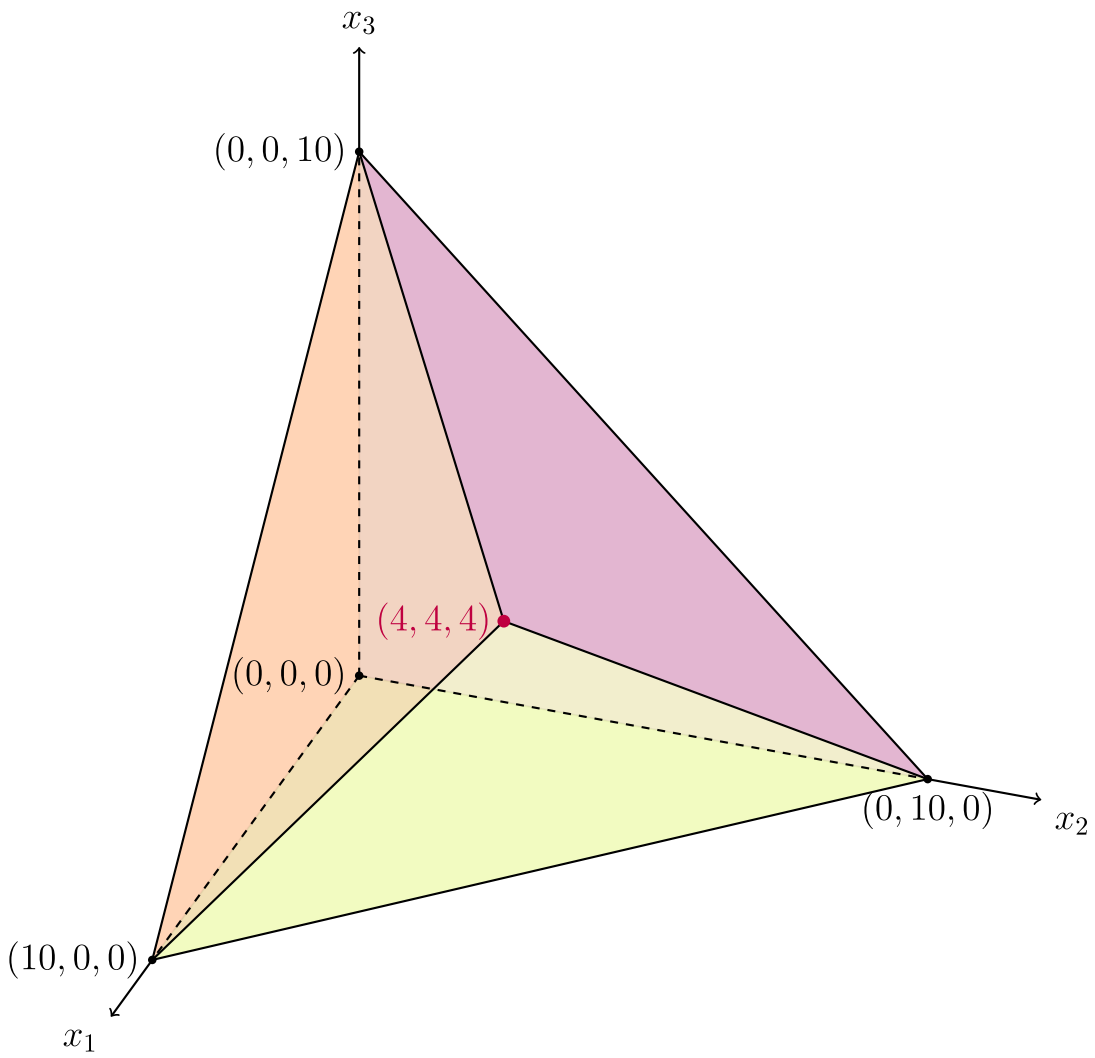
容易看出，顶点解的概念，和前文定义的基本可行解是一致的. 因此，只要在所有基本可行解内找到最优的那个，就能获得原问题的最优解. 虽然这大幅简化了问题，但是，可行域的顶点的个数可能是指数级的，穷举并不现实.

为了解决这一困难，可以考虑沿着可行域的 边→9.15.1@p1774 移动，从一个顶点移动到与之相邻的顶点. 因为相邻的顶点必定位于同一条边上，所以，它们至少满足 $n-1$ 条相同的紧约束. 也就是说，相邻的顶点对应的紧约束能且仅能相差一个. 因此，对于一个基本可行解 x，只要将它的一个基变量换成一个非基变量，就能得到一个 **相邻的**（adjacent）基本可行解 x'. 这正是转轴操作.

因此，单纯形法从一个基本可行解出发，不断进行转轴，改进目标函数的过程，其实就是在相应的可行域上，从一个顶点出发，不断向相邻顶点移动，进而改进目标函数的过程.

例子（续）

本文讨论的例子中，可行域是一个有五个顶点的三维多面体，如下图所示:



前述求解过程，从几何直观上看，就对应着多面体的顶点间的如下路径：

$$(0,0,0) \to (10,0,0) \to (0,0,10) \to (4,4,4).$$

实现细节

利用单纯形表，已经能够求解许多线性规划问题．然而，对于最一般的情形，单纯形法中仍有许多细节值得深入探讨．

松弛形式

将一般形式的线性规划问题转化为标准形式的 方法→9.15.1@p1774 已经讨论过了．但是，为了方便利用单纯形法求解，还需要保证系数矩阵 A 满秩．虽然先转换为标准形式再消去线性相关的约束的方法是可行的，但是为了求解简便，通常采用如下策略：

1. 将线性规划问题转化为 **不等式形式**（inequality form），即 $\min\{c^T x : Ax \le b,\ x \ge 0\}$ 的形式；
2. 通过添加松弛变量 s，将问题转化为标准形式：$\min\{c^T x : Ax + s = b,\ x \ge 0,\ s \ge 0\}$．

这样做的好处是，最终得到的标准形式的系数矩阵 (A, I) 总是满秩的，且总是存在（未必可行的）基本解 $(x, s) = (0, b)$．这种特殊的标准形式也称为 **松弛形式**（slack form）．

初始基本可行解

前文描述的单纯形法总是假定已知一组基本可行解．有些时候，很容易找到一组基本可行解．例如，如果上述松弛形式中 $b \geq 0$，那么，$(x, s) = (0, b)$ 就是一组基本可行解．这就是前文的数值例子里遇到的情形．

对于一般的情形，可以采用 **两阶段法**（two-phase method）．两阶段法中，需要做两次单纯形法．第一阶段求解一个可行性线性规划问题，获得原问题的一个基本可行解．第二阶段从这个基本可行解出发，应用单纯形法求解原问题．

假设有标准形式的问题 $\min\{c^T x : Ax = b \geq 0,\ x \geq 0\}$．在第一阶段中，需要求解问题

$$\min\{1^T x_a : Ax + x_a = b,\ x \geq 0,\ x_a \geq 0\}.$$

这本质是可行性线性规划问题，其中，新添加的变量 x_a 也称为 **人工变量**（artificial variable）．它一定有基本可行解 $(x, x_a) = (0, b)$，所以可以直接用单纯形法求解．如果该问题的最优价值严格大于 0，那么，就不存在 $x \geq 0$ 使得 $Ax = b$，也就是说，原问题不可行．如果该问题的最优价值等于 0，那么，它的最优解中人工变量只能是零．如果仍有一些人工变量是基变量，可以通过若干次转轴将它们出基．最后，当所有人工变量都是非基变量时，第一阶段得到的基本解就可以用作第二阶段的初始基本可行解．

不显式引入人工变量的第一阶段实现

实现第一阶段时，没有必要显式地引入人工变量．对于任意选取的初始基 B，有

$$x_B + A_B^{-1} A_N x_N = A_B^{-1} b.$$

如果 $(A_B^{-1}b)_j \geq 0$，那么无需引入人工变量；否则，需要额外引入人工变量 $x_{B_j}^-$，即

$$x_{B_j} - x_{B_j}^- + (A_B^{-1} A_N)_{(j)} x_N = (A_B^{-1} b)_j.$$

记下标集合 $L := \{j : (A_B^{-1}b)_j < 0\}$，那么一阶段的单纯形表都是如下单纯形表经过若干初等行变换得到的：

$$\begin{array}{rcccccc} & & & x_N & x_{B_{\sim L}} & x_{B_L} & x_{B_L}^- \\ & 0 & & 0^T & 0^T & 0^T & 1^T \\ x_{B_{\sim L}} = & (A_B^{-1}b)_{\sim L} & (A_B^{-1}A_N)_{(\sim L)} & I & O & O \\ x_{B_L}^- = & (A_B^{-1}b)_L & (A_B^{-1}A_N)_{(L)} & O & I & -I \end{array}$$

类似于将单纯形表简化为压缩单纯形表，可以将其适当地简化：（将 $x_{B_L}^-$ 行左乘以 1^T 然后加到第 0 行上，然后略去后三列）

$$\begin{array}{rcc} & & x_N \\ & 1^T(A_B^{-1}b)_L & 1^T(A_B^{-1}A_N)_{(L)} \\ x_{B_{\sim L}} = & (A_B^{-1}b)_{\sim L} & (A_B^{-1}A_N)_{(\sim L)} \\ -x_{B_L} = & (A_B^{-1}b)_L & (A_B^{-1}A_N)_{(L)} \end{array}$$

这与正常的压缩单纯形表大体一致，只是最后一行的变量上标记了负号，表示该行仍然含有人工变量，即原来的松弛变量仍然不可行．利用该表，转轴过程如下：

1. 如果 $L = \emptyset$，算法终止．
2. 否则，根据第 0 行的约化成本为负这一条件选取入基变量 x_{N_i}．如果不存在，则原问题不可行，算法终止．
3. 再根据第 i 列选取出基变量 x_{B_j}．仍然利用最小比值检验，但是要求同时保证当前的可行变量可行和当前的不可行变量不可行，即选取

   $$\arg\min_j \left\{ \frac{(\tilde{T}_B)_{j0}}{(\tilde{T}_B)_{ji}} : (j \notin L \wedge (\tilde{T}_B)_{ji} > 0) \vee (j \in L \wedge (\tilde{T}_B)_{ji} < 0) \right\}$$

   作为出基变量所在行．如果存在多个这样的出基变量，优先选取不可行的出基变量．
4. 令 x_{N_i} 入基，x_{B_j} 出基，并更新单纯形表．
5. 如果 $j \in L$，那么将 j 移出 L（即取消该行的负号标记），并且将 $(\tilde{T}_B)_{0i}$ 加一．

之所以可以略去人工变量所在列，是因为如果它们仍然是基变量，那么它们对应的列就是 e_j，无需记录，而如果它们不再是基变量，那么它们就不会再次入基，也无需记录．人工变量出基时，需要替换成相应的非人工变量，这正是上述过程中最后一步的目的．

参考实现如下：

参考实现

```
// First phase: find an initial BFS.
// Return false if no feasible solution.
bool initialize() {
  int neg_count = 0;
  for (int j = 0; j < m; ++j) {
    if (tab[n][j] < -eps) {
      for (int i = 0; i <= n; ++i) {
        tab[i][m + 1] += tab[i][j];
      }
      B[j] = ~B[j];
      ++neg_count;
    }
  }
  while (neg_count) {
    int x = -1;
    long double mi = -eps;
    for (int i = 0; i < n; ++i) {
      if (tab[i][m + 1] < mi) {
        x = i;
        mi = tab[i][m + 1];
      }
    }
    if (x == -1) return false;
    int y = -1;
    mi = INFINITY;
    for (int j = 0; j < m; ++j) {
      if ((B[j] < 0 && tab[x][j] < -eps) || (B[j] >= 0 && tab[x][j] >
eps)) {
        auto tmp = tab[n][j] / tab[x][j] + (B[j] < 0 ? -eps : eps);
        if (tmp < mi) {
          y = j;
          mi = tmp;
        }
      }
    }
    if (B[y] < 0) {
      --neg_count;
      B[y] = ~B[y];
      pivot(x, y);
      tab[x][m + 1] += 1;
    } else {
      pivot(x, y);
    }
  }
```

```
  return true;
}

```

在一阶段开始前，额外添加一行用于记录一阶段的目标函数．转轴时，对整个表进行转轴，包括第二阶段的目标函数．这样，在第一阶段完成时，第二阶段的目标函数也一并相应地更新，可以直接开始二阶段的单纯形法．

两阶段法也可以通过一次单纯形法实现．只需要取充分大的正数 M，就可以通过直接求解问题

$$\min\{c^T x + M 1^T x_a : Ax + x_a = b,\ x \geq 0,\ x_a \geq 0\}$$

得到原问题的最优解．实现时，并不会赋予 M 具体的数值，而是将它视为一个未知的充分大的正数进行运算．这种方法称为 **大 M 法**（big M method）．

⚠ 朴素算法的实际效率是指数级的

因为松弛形式总是存在初始基本解，只是未必是可行的，所以，一种简单的寻找初始基本可行解的想法是，从一个不可行的基本解出发，反复利用转轴操作，将不可行的基变量出基，并选择对应行中同样为负的数字对应列的非基变量入基，直到所有基变量都是非负数为止．参考实现如下：

参考实现

```
// First phase: find an initial BFS.
// Return false if no feasible solution.
bool initialize() {
  while (true) {
    int y = -1;
    long double mi = -eps;
    for (int j = 0; j < m; ++j) {
      if (tab[n][j] <= mi) {
        y = j;
        mi = tab[n][j];
      }
    }
    // No row with a negative basic variable => Feasible.
    if (y == -1) break;
    int x = -1;
    for (int i = 0; i < n; ++i) {
      if (tab[i][y] < -eps && (x == -1 || N[i] > N[x])) {
        x = i;
      }
    }
    // No column with a negative entry => Infeasible.
    if (x == -1) return false;
    pivot(x, y);
  }
  return true;
}

```

> 这样做虽然简单，但是相较于两阶段法，它没有一个描述当前基的不可行程度的目标函数，所以缺乏明确的改进方向．实际测试可以发现，相较于通常只需要 $O(m)$ 的转轴次数的两阶段法或大 M 法，这一朴素算法通常需要 $O(2^m)$ 的转轴次数，且当 n, m 较大时容易陷入循环．虽然朴素算法的转轴次数中的常数很小，但是仅仅适用于 $n, m < 50$ 的情形．

转轴规则

转轴时，如果出现多个可选的入基变量或出基变量，就需要用到 **转轴规则**（pivot rule）来决定选择哪个变量入基或出基．利用单纯形表，本节讨论的所有规则都能够在 $O(mn)$ 时间内找到入基和出基变量，因此，单次转轴的时间复杂度仍然是 $O(mn)$ 的．

入基变量的选择往往决定了算法终止前转轴的次数．常见的规则如下：

- 选择最先找到的入基变量；
- 选择标号最小的入基变量；（Bland 规则的一部分）
- 选择约化成本绝对值（即 $|\tilde{c}_i|$）最大的入基变量；（Dantzig 规则）
- 选择单次转轴价值函数改进（即 $|\tilde{c}_i|\,\theta_i$）最大的入基变量；
- 选择对应着最陡的边的入基变量，即沿着边移动单位长度引起的价值函数改进（即 $|\tilde{c}_i| \,/\sqrt{1+\|A_B^{-1}A_i\|^2}$）最大；
- 随机选择一个入基变量．

实践中，最陡边规则的效率最高[5]．通常认为，适当的转轴规则可以在大致 $2m$ 次转轴内得到大多数问题的最优解．但是，对于目前所有已知的转轴规则，都存在特殊构造的例子[6]，能够将转轴次数卡到指数级．这也正是单纯形法实践中运行效率相当优秀，但是理论最差复杂度是指数级的原因．

出基变量的选择往往决定了算法是否会陷入循环．如果存在多组基对应着同一个基本可行解（即退化情形），那么，算法就有可能一直在这些基之间循环．这些情形并不常见，因此很多单纯形法的实现并不会指定出基变量的选择规则．常见的避免循环的规则有两种：

- Bland 规则：总是选择标号最小的入基变量和出基变量．
- 字典序规则：总是选择 $(A_B^{-1}A_i)_j > 0$ 且

  $$\left(\frac{(A_B^{-1}b)_j}{(A_B^{-1}A_i)_j}, \frac{(A_B^{-1})_{j1}}{(A_B^{-1}A_i)_j}, \ldots, \frac{(A_B^{-1})_{jm}}{(A_B^{-1}A_i)_j}\right)$$

  字典序最小的行号 j 对应的出基变量 x_{B_j}．入基变量的选择不重要．

  注意，如果线性规划问题是松弛形式的，那么这些量都可以从前文所述形式的单纯形表 T_B 中直接找到；否则，可以在找到一组初始基本解（未必可行）后，利用这些初始基中的基变量对应列（顺序保持固定）的系数作为 A_B^{-1} 的系数．

Bland 规则效率很低，因为规则本身对入基和出基变量的选择方法是一样的，这很容易造成同一个变量反复入基再出基．相对来说，字典序规则更为实用．字典序规则相当于对线性规划问题中的参数进行微扰[7]，使得不存在退化的基本可行解，也就不存在循环的可能性．

参考实现

本节提供一个基于压缩单纯形表的两阶段单纯形法的参考实现．

---

[5] 测试结果参见 Forrest, John J., and Donald Goldfarb. "Steepest-edge simplex algorithms for linear programming." Mathematical programming 57, no. 1 (1992): 341-374.

[6] 一个经典的反例可以参见 Klee, Victor, and George J. Minty. "How good is the simplex algorithm." Inequalities 3, no. 3 (1972): 159-175.

[7] 详细的解释可以看 这份讲义[1].

Luogu P13337【模板】线性规划[2]

```cpp
#include <algorithm>
#include <climits>
#include <cmath>
#include <iomanip>
#include <iostream>
#include <numeric>
#include <vector>

int m, n;  // Number of constraints and variables.
std::vector<std::vector<long double>>
    tab;                                // Compressed tableau (transposed) with
                                        // first-phase objective attached.
std::vector<int> B, N;                  // Basic and nonbasic variables.
constexpr long double eps = 1e-12l;  // Precision.

// Pivot on (N[x], B[y]).
void pivot(int x, int y) {
  std::swap(N[x], B[y]);
  long double v = -1 / tab[x][y];
  for (int j = 0; j <= m + 1; ++j) {
    tab[x][j] = j == y ? -v : v * tab[x][j];
  }
  for (int i = 0; i <= n; ++i) {
    if (i == x) continue;
    v = tab[i][y];
    tab[i][y] = 0;
    for (int j = 0; j <= m + 1; ++j) {
      tab[i][j] += v * tab[x][j];
    }
  }
}

// First phase: find an initial BFS.
// Return false if no feasible solution.
bool initialize() {
  int neg_count = 0;
  for (int j = 0; j < m; ++j) {
    if (tab[n][j] < -eps) {
      for (int i = 0; i <= n; ++i) {
        tab[i][m + 1] += tab[i][j];
      }
      B[j] = ~B[j];
      ++neg_count;
    }
  }
  while (neg_count) {
    int x = -1;
    long double mi = -eps;
    for (int i = 0; i < n; ++i) {
      if (tab[i][m + 1] < mi) {
        x = i;
```