命题 2

若间距序列为 $H = \{k = 2^p \cdot 3^q \mid p, q \in \mathbb{N}, k \le n\}$（从大到小），则希尔排序算法的时间复杂度为 $O(n \log^2 n)$.

为证明这两个命题，我们先给出一个重要的定理并证明它，这个定理反映了希尔排序的最主要特征.

定理 1

只要程序执行了一次 InsertionSort(h)，不管之后怎样调用 InsertionSort 函数，A 数组怎样变换，下列每个子序列均保持有序：

$$\begin{matrix} A_1, A_{1+h}, A_{1+2h}, \cdots \\ A_2, A_{2+h}, A_{2+2h}, \cdots \\ \vdots \\ A_h, A_{h+h}, A_{h+2h}, \cdots \end{matrix}$$

接下来我们证明定理 1.

我们先证明引理 1.

引理 1

对于非负整数 n, m、正整数 l 与两个数组 $X(x_1, x_2, ..., x_{n+l}), Y(y_1, y_2, ..., y_{m+l})$，满足如下要求：

$$y_1 \le x_{n+1}, y_2 \le x_{n+2}, ..., y_l \le x_{n+l}$$

则我们将两个数组分别升序排序后，上述要求依然成立.

引理 1 证明

设数组 X 排序完为数组 $X'(x'_1, ..., x'_{n+l})$，数组 Y 排序完为数组 $Y'(y'_1, ..., y'_{m+l})$.

对任意 $1 \le i \le l$，X 中严格大于 x'_{n+i} 的元素至多有 $l - i$ 个. 所以在选定的 l 个元素 $\{x_{n+1}, ..., x_{n+l}\}$ 中，至少有 i 个不大于 x'_{n+i}. 记为 $x_{n+k_1}, ..., x_{n+k_i}$，结合原配对不等式可得：

$$y_{k_1} \le x_{n+k_1} \le x'_{n+i}, y_{k_2} \le x_{n+k_2} \le x'_{n+i}, ..., y_{k_i} \le x_{n+k_i} \le x'_{n+i}$$

所以 x'_{n+i} 至少大于等于 Y 也即 Y' 中的 i 个元素，那么自然有 $y'_i \le x'_{n+i}\ (1 \le i \le l)$.

再回到原命题的证明. 设数组长度为 N，我们只需证明 InsertionSort(k) 会保持已有 h 的有序性，再对调用次数归纳即可. 这里 h, k 为任意正整数. 若 $h \ge N$，没有需要检查的相距 h 的元素对，结论显然成立. 以下设 $h < N$.

记调用前后的数组分别为 A 与 A'. 调用前有 $A_i \le A_{i+h}$，其中 $1 \le i \le N - h$. 对每个 $1 \le w \le \min(k, N - h)$，按带余除法写成

$$w + h = v + tk, \qquad 1 \le v \le k, \quad t \ge 0.$$

考虑从下标 w 和 v 开始、间距为 k 的两个完整子序列：

$$Y = (A_w, A_{w+k}, A_{w+2k}, ...), \qquad X = (A_v, A_{v+k}, A_{v+2k}, ...).$$

设满足 $w + h + jk \le N$ 的非负整数 j 共有 l 个. 于是 X 的长度为 $t + l$，Y 的长度至少为 l；由原来的 h 有序性，Y 的前 l 项与 X 的后 l 项满足

$$y_{j+1} = A_{w+jk} \le A_{w+h+jk} = x_{t+j+1}, \qquad 0 \le j < l.$$

InsertionSort(k) 恰好对每个这样的完整子序列进行升序排序. 由引理1, 排序后的对应关系仍成立, 即

$$A'_{w+jk} \le A'_{w+h+jk}, \qquad 0 \le j < l.$$

即使 $v=w$、两个子序列实际上相同，上述引理也仍然适用．任意下标 $1\le i\le N-h$ 都能唯一写为 $i=w+jk$，其中 $1\le w\le k$、$j\ge 0$，且必有 $w\le N-h$．因此上面的讨论覆盖全部 i，得到 $A'_i\le A'_{i+h}$，所以 h 有序性保持不变，定理 1 得证．

这个定理揭示了希尔排序在特定集合 H 下可以优化复杂度的关键，因为在整个过程中，它可以一直保持此前各子序列有序的性质，从而使后面的调用中，指针 i 的移动次数大大减少．

接下来我们单拎出来一个数论引理进行证明．这个定理在 OI 界因 小凯的疑惑[1] 一题而大为出名．而在希尔排序复杂度的证明中，它也使得定理 1 得到了很大的扩展．

引理 2

若 $a,b\ge 2$ 均为整数且互素，则不在集合 $\{ax+by\mid x,y\in\mathbb{N}\}$ 中的最大正整数为 $ab-a-b$.

引理 2 证明

分两步证明：

- 先证明方程 $ax+by=ab-a-b$ 没有 x,y 均为非负整数的解：

  若无非负整数的限制，容易得到两组解 $(b-1,-1),(-1,a-1)$.

  通过其通解形式 $x=x_0+tb,y=y_0-ta$，容易得到上面两组解是「相邻」的（因为 $b-1-b=-1$）.

  当 t 递增时，x 递增，y 递减，所以如果方程有非负整数解，必然会夹在这两组解中间，但这两组解「相邻」，中间没有别的解.

  故不可能有非负整数解.
- 再证明对任意整数 $c>ab-a-b$，方程 $ax+by=c$ 有非负整数解：

  我们找一组解 (x_0,y_0) 满足 $0\le x_0<b$（由通解的表达式，这可以做到）.

  则有：

  $by_0=c-ax_0\ge c-a(b-1)>ab-a-b-ab+a=-b$

  所以 $b(y_0+1)>0$，又因为 $b>0$，所以 $y_0+1>0$，所以 $y_0\ge 0$.

  所以 (x_0,y_0) 为一组非负整数解.

综上得证.

而下面这个定理则揭示了引理 2 是如何扩展定理 1 的.

定理 2

设 $h_{t+1}>h_t>h_{t-1}$ 为正整数．如果 $\gcd(h_{t+1},h_t)=1$，则程序先执行完 InsertionSort(h_{t+1}) 与 InsertionSort(h_t) 后，执行 InsertionSort(h_{t-1}) 的时间复杂度为 $O\left(\frac{nh_{t+1}h_t}{h_{t-1}}\right)$，且对于每个 j，其 i 的移动次数是 $O\left(\frac{h_{t+1}h_t}{h_{t-1}}\right)$ 级别的.

定理 2 证明

以下 A 表示调用 InsertionSort(h_{t-1}) 前的数组．固定外层循环的下标 j，待插入的值为 $v=A_j$.

对于 $j\le h_{t+1}h_t$ 的部分，i 的移动次数显然是 $O\left(\frac{h_{t+1}h_t}{h_{t-1}}\right)$ 级别的.

故以下假设 $j>h_{t+1}h_t$.

对于任意的正整数 k 满足 $1 \le k \le j - h_{t+1}h_t$，注意到：$h_{t+1}h_t - h_{t+1} - h_t < h_{t+1}h_t \le j - k \le j - 1$.

又因为 $\gcd(h_{t+1}, h_t) = 1$，故由引理 2，得存在非负整数 a, b，使得：$ah_{t+1} + bh_t = j - k$.

即得：

$$k = j - ah_{t+1} - bh_t$$

由定理 1，得：

$$A_{j-bh_t} \le A_{j-(b-1)h_t} \le ... \le A_{j-h_t} \le A_j$$

与

$$A_{j-bh_t-ah_{t+1}} \le A_{j-bh_t-(a-1)h_{t+1}} \le ... \le A_{j-bh_t-h_{t+1}} \le A_{j-bh_t}$$

综合以上即有：$A_k = A_{j-ah_{t+1}-bh_t} \le A_j$.

所以对于任何 $1 \le k \le j - h_{t+1}h_t$，有 $A_k \le A_j$.

在 v 所属子序列的已处理前缀中，只有原下标位于 $(j - h_{t+1}h_t, j)$ 内的元素可能大于 v，这样的元素至多有 $\left\lceil \frac{h_{t+1}h_t}{h_{t-1}} \right\rceil$ 个．此前的插入只将该子序列的前缀排序，因此，在上面的伪代码中，i 每次减 h_{t-1}，越过这些元素后就会遇到不大于 v 的元素或越过数组左端，从而退出 while 循环．移动次数为 $O\left(\frac{h_{t+1}h_t}{h_{t-1}}\right)$.

证明完对于每个 j 的移动复杂度后，即可得到总的时间复杂度：

$$\sum_{j=h_{t-1}+1}^{n} O\left(\frac{h_{t+1}h_t}{h_{t-1}}\right) = O\left(\frac{nh_{t+1}h_t}{h_{t-1}}\right)$$

得证.

认真观察定理 2 的证明过程，可以发现：定理 1 可以进行「线性组合」，即 A 以 h 为间隔有序，以 k 为间隔亦有序，则以 h 和 k 的非负系数线性组合仍是有序的．而这种「线性性」即是由引理 2 保证的.

有了这两个定理，我们可以证明命题 1 与 2.

命题 1 证明

将 H 写为序列的形式：

$$H(h_1 = 1, h_2 = 3, h_3 = 7, ..., h_{\lfloor \log_2 n \rfloor} = 2^{\lfloor \log_2 n \rfloor} - 1)$$

Shell-Sort 执行顺序为：$\text{InsertionSort}(h_{\lfloor \log_2 n \rfloor}), \text{InsertionSort}(h_{\lfloor \log_2 n \rfloor - 1}), ..., \text{InsertionSort}(h_2), \text{InsertionSort}(h_1)$.

以下设 $n \ge 4$，分两部分去分析复杂度：

- 对于前面的若干个满足 $h_t \ge \sqrt{n}$ 的 h_t，显然有 $\text{InsertionSort}(h_t)$ 的时间复杂度为 $O\left(\frac{n^2}{h_t}\right)$.

  取 $k = \min\{t : h_t \ge \sqrt{n}\}$，则 $h_k = \Theta(\sqrt{n})$，有：

  $$O\left(\frac{n^2}{h_k}\right) = O(n^{3/2})$$

  而对于 $i > k$ 的 h_i，因为有 $2h_i < h_{i+1}$，所以可得：

  $$O\left(\frac{n^2}{h_i}\right) = O(n^{3/2}/2^{i-k}) \, (i > k)$$

  所以大于等于 $\sqrt{n}$ 部分的总时间复杂度为：

$$\sum_{i=k}^{\lfloor \log_2 n \rfloor} O(n^{3/2}/2^{i-k}) = O(n^{3/2})$$

- 对于后面剩下的满足 $h_t < \sqrt{n}$ 的项，前两项的复杂度还是 $O(n^{3/2})$，而对于后面的项 h_t，由定理 2 可得时间复杂度为：

  $$O\left(\frac{nh_{t+2}h_{t+1}}{h_t}\right) = O\left(\frac{nh_{t+2} \cdot h_{t+2}/2}{h_{t+2}/4}\right) = O(nh_{t+2})$$

  再次利用 $2h_i < h_{i+1}$ 性质可得此部分总时间复杂度为（下式中 k 沿用了上一种情况中的含义）：

  $$2O(n^{3/2}) + \sum_{i=1}^{k-3} O(nh_{i+1}) = O(n^{3/2}) + \sum_{i=1}^{k-3} O(nh_{k-1}/2^{k-i-3}) = O(n^{3/2}) + O(nh_{k-1}) = O(n^{3/2})$$

综上可得总时间复杂度即为 $O(n^{3/2})$.

命题 2 证明

注意到一个事实：如果已经执行过了 InsertionSort(2) 与 InsertionSort(3)，那么因为 $2 \cdot 3 - 2 - 3 = 1$，所以由定理 2，每个元素只有与它相邻的前一个元素可能大于它，之前的元素全部都小于它. 于是 i 指针只需要最多两次就可以退出 while 循环. 也就是说，此时再执行 InsertionSort(1)，复杂度降为 $O(n)$.

更进一步：如果已经执行过了 InsertionSort(4) 与 InsertionSort(6)，我们考虑所有的下标为奇数的元素组成的子列与下标为偶数的元素组成的子列. 则这相当于把这两个子列分别执行 InsertionSort(2) 与 InsertionSort(3). 那么也是一样，这时候再执行 InsertionSort(2)，相当于对两个子列分别执行 InsertionSort(1)，也只需要两个序列和的级别，即 $O(n)$ 的复杂度就可以将数组变为 2 间隔有序.

不断归纳，就可以得到：如果已经执行过了 InsertionSort($2h$) 与 InsertionSort($3h$)，则执行 InsertionSort(h) 的复杂度也只有 $O(n)$.

接下来分为两部分分析复杂度：

- 对于 $h_t > n/3$ 的部分，则执行每个 InsertionSort(h_t) 的复杂度为 $O(n^2/h_t)$.

  而 $n^2/h_t < 3n$，所以单次插入排序复杂度为 $O(n)$.

  而这一部分元素个数是 $O(\log^2 n)$ 级别的，所以这一部分时间复杂度为 $O(n \log^2 n)$.
- 对于 $h_t \le n/3$ 的部分，因为 $3h_t \le n$，所以这之前已经执行了 InsertionSort($2h_t$) 与 InsertionSort($3h_t$)，于是执行 InsertionSort(h_t) 的时间复杂度是 $O(n)$.

  还是一样的，这一部分元素个数也是 $O(\log^2 n)$ 级别的，所以这一部分时间复杂度为 $O(n \log^2 n)$.

综上可得总时间复杂度即为 $O(n \log^2 n)$.

空间复杂度

希尔排序的空间复杂度为 $O(1)$.

实现

- C++[11]

```cpp
template <typename T>
void shell_sort(T array[], int length) {
```

[11]希尔排序 - 维基百科，自由的百科全书[2]

```
  int h = 1;
  while (h < length / 3) {
    h = 3 * h + 1;
  }
  while (h >= 1) {
    for (int i = h; i < length; i++) {
      for (int j = i; j >= h && array[j] < array[j - h]; j -= h) {
        std::swap(array[j], array[j - h]);
      }
    }
    h = h / 3;
  }
}
```

- Python

```
def shell_sort(array, length):
    h = 1
    while h < length / 3:
        h = int(3 * h + 1)
    while h >= 1:
        for i in range(h, length):
            j = i
            while j >= h and array[j] < array[j - h]:
                array[j], array[j - h] = array[j - h], array[j]
                j -= h
        h = int(h / 3)
```

参考资料与注释

[1] 小凯的疑惑

[2] 希尔排序 - 维基百科, 自由的百科全书

5.7.12 锦标赛排序

本页面将简要介绍锦标赛排序.

定义

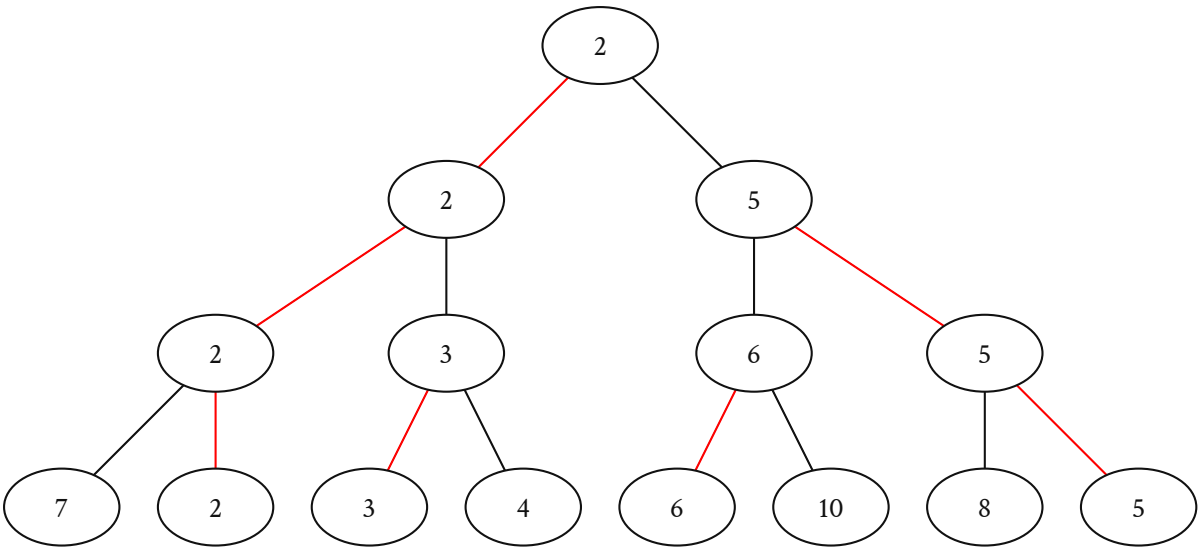
锦标赛排序（英文：Tournament sort），又被称为树形选择排序，是 选择排序→5.7.2@p568 的优化版本，堆排序→5.7.9@p596 的一种变体（均采用完全二叉树）. 它在选择排序的基础上使用优先队列查找下一个该选择的元素.

引入

锦标赛排序的名字来源于单败淘汰制的竞赛形式. 在这种赛制中有许多选手参与比赛，他们两两比较，胜者进入下一轮比赛. 这种淘汰方式能够决定最好的选手，但是在最后一轮比赛中被淘汰的选手不一定是第二好的——他可能不如先前被淘汰的选手.

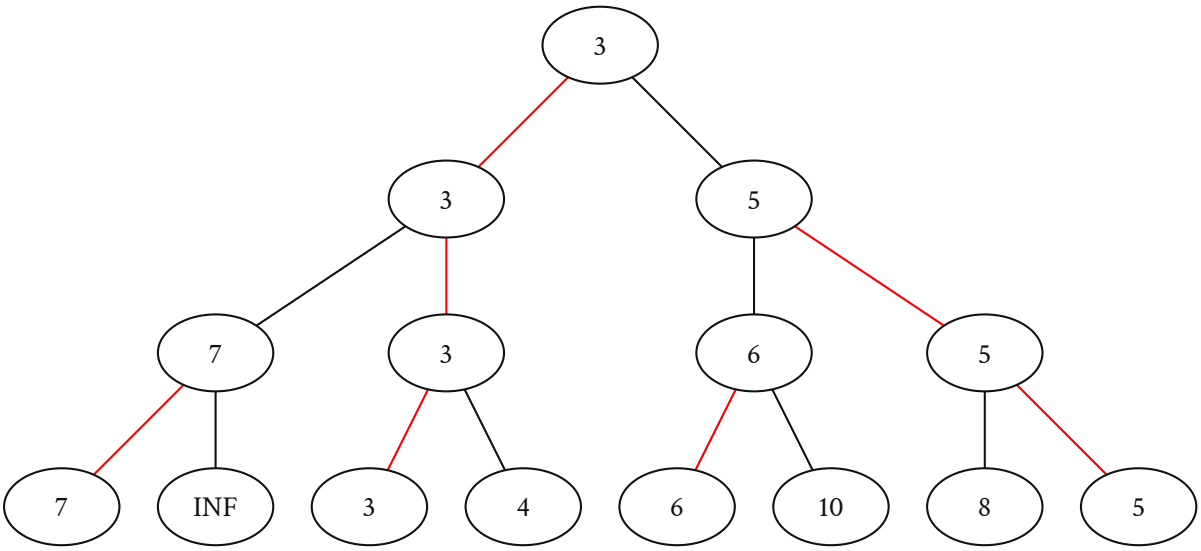
过程

以 **最小锦标赛排序树** 为例:



待排序元素是叶子节点显示的元素．红色边显示的是每一轮比较中较小的元素的胜出路径．显然，完成一次「锦标赛」可以选出一组元素中最小的那一个．

每一轮对 n 个元素进行比较后可以得到 $\left\lceil \frac{n}{2} \right\rceil$ 个「优胜者」，每一对中较小的元素进入下一轮比较．如果无法凑齐一对元素，那么这个元素直接进入下一轮的比较．



完成一次「锦标赛」后需要将被选出的元素去除．直接将其设置为 ∞（这个操作类似 堆排序 →5.7.9@p596），然后再次举行「锦标赛」选出次小的元素．

之后一直重复这个操作，直至所有元素有序．

性质

稳定性

锦标赛排序是一种不稳定的排序算法．

时间复杂度

锦标赛排序的最优时间复杂度、平均时间复杂度和最坏时间复杂度均为 $O(n \log n)$．它用 $O(n)$ 的时间初始化「锦标赛」，然后用 $O(\log n)$ 的时间从 n 个元素中选取一个元素．

空间复杂度

锦标赛排序的空间复杂度为 $O(n)$.

实现

- C++

```cpp
int n, a[MAXN], tmp[MAXN << 1];

int winner(int pos1, int pos2) {
  int u = pos1 >= n ? pos1 : tmp[pos1];
  int v = pos2 >= n ? pos2 : tmp[pos2];
  if (tmp[u] <= tmp[v]) return u;
  return v;
}

void create_tree(int& value) {
  for (int i = 0; i < n; i++) tmp[n + i] = a[i];
  for (int i = 2 * n - 1; i > 1; i -= 2) {
    int k = i / 2;
    int j = i - 1;
    tmp[k] = winner(i, j);
  }
  value = tmp[tmp[1]];
  tmp[tmp[1]] = INF;
}

void recreate(int& value) {
  int i = tmp[1];
  while (i > 1) {
    int j, k = i / 2;
    if (i % 2 == 0)
      j = i + 1;
    else
      j = i - 1;
    tmp[k] = winner(i, j);
    i = k;
  }
  value = tmp[tmp[1]];
  tmp[tmp[1]] = INF;
}

void tournament_sort() {
  if (n <= 1) return;
  int value;
  create_tree(value);
  for (int i = 0; i < n; i++) {
    a[i] = value;
    recreate(value);
  }
}

```

- Python

```python
n = 0
```

```
a = [0] * MAXN
tmp = [0] * MAXN * 2


def winner(pos1, pos2):
    u = pos1 if pos1 >= n else tmp[pos1]
    v = pos2 if pos2 >= n else tmp[pos2]
    if tmp[u] <= tmp[v]:
        return u
    return v


def create_tree():
    for i in range(0, n):
        tmp[n + i] = a[i]
    for i in range(2 * n - 1, 1, -2):
        k = i // 2
        j = i - 1
        tmp[k] = winner(i, j)
    value = tmp[tmp[1]]
    tmp[tmp[1]] = INF
    return value


def recreate():
    i = tmp[1]
    while i > 1:
        j = k = i // 2
        if i % 2 == 0:
            j = i + 1
        else:
            j = i - 1
        tmp[k] = winner(i, j)
        i = k
    value = tmp[tmp[1]]
    tmp[tmp[1]] = INF
    return value


def tournament_sort():
    if n <= 1:
        return
    value = create_tree()
    for i in range(0, n):
        a[i] = value
        value = recreate()


```

外部链接

- Tournament sort - Wikipedia[1]

外部参考资料

[1] Tournament sort - Wikipedia 

5.7.13 Tim 排序

本页面将介绍 Tim 排序（Timsort），一种混合的、稳定的排序算法.

引入

Timsort 由 Python 核心开发者 Tim Peters 于 2002 年设计，并应用于 Python 语言，其巧妙结合了插入排序和归并排序的优点，针对数据集中的有序性进行了精确的优化，尤其适合处理包含大量部分有序子序列的数据集. CPython 从 2.3 版本开始使用 Timsort，之后从 3.11 版本开始将其归并顺序策略替换为 Powersort[12]. Timsort 也被广泛应用于其他编程环境, 例如在 Java SE 7 中被用于对非原始对象数组进行排序.

本页介绍传统 Timsort，并以 CPython 3.6.5 中已修复栈不变量维护问题的归并规则为准[13].

步骤

Timsort 的核心思想是通过识别和利用数据集中已有的有序性, 提高排序效率, 其主要包括以下步骤:

1. **识别 Run**：扫描待排序数组，识别出有序的连续子序列（Run）;
2. **扩展 Run**：如果识别的 Run 长度小于 `MIN_RUN`，则使用插入排序对其进行扩展;
3. **归并 Run**：Timsort 维护一个特殊的栈，采用特定的归并策略将栈中已有的 Run 合并成更大的有序序列.

识别 Run

首先，Timsort 会从左向右扫描数组，识别出连续的有序序列，这些有序序列被称为 Run:

- **升序 Run**：如果后一个元素大于等于前一个元素，则继续扩展 Run.
- **降序 Run**：如果后一个元素小于前一个元素，则继续扩展 Run，随后将该 Run 反转为升序.

扩展 Run

为了提高小规模数据的排序效率，Timsort 引入了一个 Run 最小的长度 `MIN_RUN`. 其值一般根据待排序数组的长度动态计算，通常为 32 至 64 之间.

- 如果识别的 Run 长度大于等于 `MIN_RUN`，则不需要额外操作，直接将 Run 压入栈中.
- 如果识别的 Run 长度小于 `MIN_RUN`，则使用二分插入排序将该 Run 的后续元素插入到 Run 中，直到 Run 的长度达到 `MIN_RUN`，然后将其压入栈中.

归并 Run

在 Timsort 中，归并排序是通过 **栈** 来管理和控制的. 栈中保存了已经识别出的有序的 Run，并通过特定的归并规则控制栈中 Run 的合并，其目的是在合并时保持序列的平衡性和稳定性.

归并规则

Timsort 是一种稳定的排序算法，即相同元素在排序后仍然保持原有的相对顺序. 为确保这一点，Timsort 在归并时只会合并相邻的、连续的 Run，而不会直接合并非相邻的 Run. 因为非相邻的 Run 之间可能存在相同的元素，直接合并很有可能会打乱它们的相对顺序.

同时，为了控制合并的平衡性和待合并 Run 的数量，Timsort 要求栈中每组相邻的三个 Run 都满足以下不变量. 记三个 Run 从栈顶向栈底的方向依次为 X、Y 和 Z:

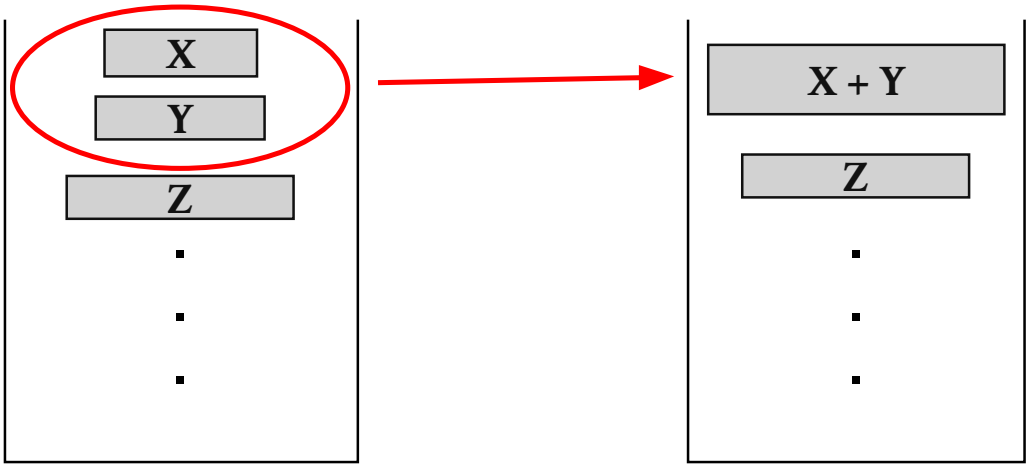
- **条件一**：`len(Z) > len(Y) + len(X)`
- **条件二**：`len(Y) > len(X)`

[12] CPython：采用 Powersort 归并顺序策略的提交[1]；CPython 3.11 的归并策略说明[2].
[13] CPython 3.6.5：`merge_collapse`[3].

栈中只有两个 Run 时也要求 `len(Y) > len(X)`. 每次压入新 Run 后，`mergeCollapse` 会检查栈顶至多四个 Run，记它们从栈顶向下依次为 X、Y、Z、W，并按以下顺序决定是否合并[14]：

1. 若 `len(Z) <= len(Y) + len(X)` 或 `len(W) <= len(Z) + len(Y)`，则将 Y 与 X、Z 中较短的一个合并；X 与 Z 等长时选择 X；
2. 否则，若 `len(Y) <= len(X)`，则合并 Y 与 X；
3. 否则，停止本轮合并，继续识别下一个 Run.

只检查所需 Run 均存在的条件；每次合并后重新确定 X、Y、Z、W 并重复上述过程. 下图示意合并 X 与 Y 时栈的变化.



为什么需要检查第四个 Run?

仅检查栈顶三个 Run，可能在合并后遗漏栈中更深处的不变量失效. 例如，依次压入长度为 $240, 160, 50, 40, 60$ 的 Run（从栈底到栈顶），旧规则会合并 50 和 40，得到 $240, 160, 90, 60$. 此时栈顶满足 $160 > 90 + 60$ 和 $90 > 60$，但更深处有 $240 \leq 160 + 90$.

这一缺陷会使根据全栈不变量推导的固定栈容量保证失效. CPython 在 2015 年修复了检查条件[15].

归并优化

为了在归并不同长度的 Run 时提高效率并减少空间开销，Timsort 在归并前会通过二分查找精确定位需要处理的元素范围，只对需要移动的部分进行归并，具体方式为：

1. **确定插入点**：使用二分查找，找到第二个 Run 的第一个元素在第一个 Run 中的插入位置，以及第一个 Run 的最后一个元素在第二个 Run 中的插入位置. 这样，可以缩小需要归并的范围，只对需要移动的元素进行处理；
2. **临时缓冲区**：传统的原地合并算法效率太低，需要大量的元素移动. 为了减少这种开销，Timsort 使用一个临时缓冲区，将长度较小的 Run 复制到缓冲区中，然后逐步将元素从缓冲区复制回原数组.

例如，假设存在两个 Run A 和 B，分别为：

- Run A:$[1, 2, 3, 6, 10]$
- Run B:$[4, 5, 7, 9, 12, 14, 17]$

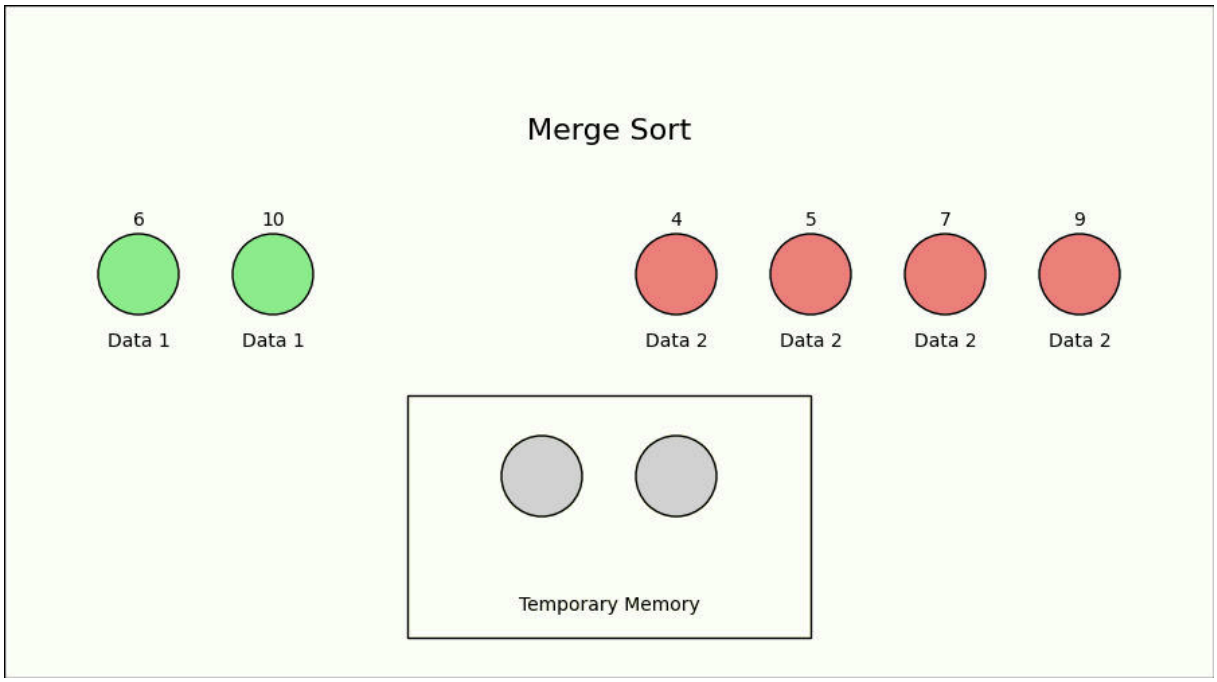
[14] CPython 3.6.5：`merge_collapse`[4].

[15] CPython Issue 23515：Bad logic in timsort's merge_collapse[5].

通过二分查找，可以确定：

- 元素 4 应插入到 Run A 的第四个位置.
- 元素 10 应插入到 Run B 的第五个位置.

因此，Run A 的前 3 个元素和 Run B 的后 3 个元素已经在正确位置，无需处理. 只需归并 Run A 的 $[6, 10]$ 和 Run B 的 $[4, 5, 7, 9]$，其归并过程如下图所示：



加速模式

为进一步提升归并效率，Timsort 引入了 **加速模式（Galloping Mode）**. 在标准的归并过程中，算法会逐一比较两个 Run 中的元素，将较小的元素放入结果数组. 然而，如果一侧的 Run 中有大量连续元素比另一侧的当前元素要小，逐一比较会造成不必要的开销.

为了解决这一问题，Timsort 设定了一个阈值 `Min_Gallop`（默认值为 7）. 当一侧 Run 中的元素连续比较胜利的次数达到 `Min_Gallop` 时，算法会进入加速模式，快速定位元素位置，其具体步骤如下：

1. **指数查找**：从当前位置开始，算法以指数增长的步长 $(1, 2, 4, 8, ...)$ 在一侧的 Run 中查找，直到找到一个区间，使得目标元素位于该区间内；
2. **二分查找**：一旦确定了包含目标元素的区间，算法会在该区间内使用二分查找，精确定位目标元素的位置.

通过这种方式，Timsort 可以跳过大量不必要的比较，快速处理一侧 Run 中连续的、较小（或较大）的元素，将它们批量移动到合并结果中.

然而，加速模式并非在所有情况下都更高效. 在某些数据分布下，加速模式可能导致更多的比较次数. 为此，Timsort 采用了动态调整策略：

- **阈值调整**：维护一个可变的 `Min_Gallop` 参数. 当加速模式表现良好（即连续多次从同一 Run 中选取元素）时，`Min_Gallop` 减 1，鼓励继续使用加速模式；当加速模式效果不佳（频繁在两个 Run 之间切换）时，`Min_Gallop` 加 1，降低加速模式的使用频率.

通过动态调整 `Min_Gallop` 的值，算法能够根据实际数据情况，在普通归并模式和加速模式之间取得平衡. 对于部分有序或高度有序的数据，加速模式可以显著提高效率，使 Timsort 的性能接近 $O(n)$；而对于随机数据，算法会逐渐倾向于使用普通归并，从而保证 $O(n \log n)$ 的时间复杂度.

复杂度

Timsort 的时间复杂度取决于数据的有序性：

- **最优情况**：$O(n)$
  - 当数据已经有序或近似有序时，算法识别出的 Run 长度接近 n，归并次数减少，复杂度趋近于 $O(n)$.
- **最坏情况**：$O(n \log n)$
  - 若有 r 个 Run，最终需要 $r-1$ 次二路合并；不能把合并次数写成 $O(\log n)$.

识别 Run 花费 $O(n)$，以有界的 `MIN_RUN` 扩展短 Run 也共需 $O(n)$. 归并成本则是每次合并的两个 Run 长度之和，等价于对每个元素参与合并的次数求和. 这个总和的上界依赖具体合并规则，对文献[16]所分析的 Python 3.6.5 TimSort 合并规则而言，均摊分析给出 $O(n + n \log r)$，从而最坏为 $O(n \log n)$.

而对于空间复杂度，由于 Timsort 大致需要额外的 $O(n)$ 空间用于存储栈和临时缓冲区，因此总的空间复杂度为 $O(n)$.

实现

伪代码实现

```
nRemaining ← 数组长度
minRun ← 选择合适的 MinRun 的值(nRemaining)
startIndex ← 0
while nRemaining > 0 do
    runLength ← 识别 Run (array, startIndex, nRemaining)
    if runLength < minRun then
        extendLength ← min(minRun, nRemaining)
        使用插入排序扩展区间 [startIndex, startIndex + extendLength − 1]
        runLength ← extendLength
    end if
    将 Run (startIndex, runLength) 压入栈中
    调用 mergeCollapse(栈) 检查并合并栈中的 Run
    startIndex ← startIndex + runLength 更新起始位置
    nRemaining ← nRemaining − runLength 更新剩余长度
end while
调用 mergeForceCollapse(栈) 对栈中所有 Run 进行最终的合并
```

参考资料与注释

1. Timsort[7]
2. Tim Peters 的设计说明（CPython 3.6.5，归并条件以对应版本代码为准）[8]
3. Java 实现[9]
4. C 语言实现（CPython 3.6.5）[10]

[1] CPython：采用 Powersort 归并顺序策略的提交

[2] CPython 3.11 的归并策略说明

[3] CPython 3.6.5：`merge_collapse`

[4] CPython 3.6.5：`merge_collapse`

[5] CPython Issue 23515：Bad logic in timsort's merge_collapse

[6] On the Worst-Case Complexity of TimSort

[7] Timsort

[8] Tim Peters 的设计说明（CPython 3.6.5，归并条件以对应版本代码为准）

[9] Java 实现

[10] C 语言实现（CPython 3.6.5）

[16] On the Worst-Case Complexity of TimSort[6].

5.7.14 排序相关 STL

本页面将简要介绍 C 和 C++ 标准库中实现的排序算法.

除已说明的函数外，本页所列函数默认定义于头文件 `<algorithm>` 中.

qsort

参见：`qsort`[1]，`std::qsort`[2]

该函数是 C 标准库的通用数组排序函数，具体算法由库实现决定，定义在 `<stdlib.h>` 中. 在 C++ 标准库里，该函数定义在 `<cstdlib>` 中.

> ⚠ **注意** [17]
>
> 虽然该函数名字叫 `qsort`，但 C 和 POSIX 标准都未要求此函数使用 快速排序→5.7.7@p585，也从未保证任何复杂度或稳定性.

qsort 与 bsearch 的比较函数

qsort 函数有四个参数：数组名、元素个数、元素大小、比较规则. 其中，比较规则通过指定比较函数来实现，指定不同的比较函数可以实现不同的排序规则.

比较函数的参数限定为两个类型为 `const void *` 的参数. 返回值规定为正数、负数和 0.

比较函数的一种示例写法为：

```c
int compare(const void* p1, const void* p2)  // int 类型数组的比较函数
{
  int* a = (int*)p1;
  int* b = (int*)p2;
  if (*a > *b)
    return 1;  // 返回正数表示 a 大于 b
  else if (*a < *b)
    return -1;  // 返回负数表示 a 小于 b
  else
    return 0;  // 返回 0 表示 a 与 b 等价
}
```

注意：返回值用两个元素相减代替正负数是一种典型的错误写法，因为这样可能会导致溢出错误.

以下是排序结构体的一个示例：

```c
struct eg  // 示例结构体
{
  int e;
  int g;
};

// struct eg 类型数组的比较函数：按成员 e 排序
int compare(const void* p1, const void* p2) {
  struct eg* a = (struct eg*)p1;
  struct eg* b = (struct eg*)p2;
  if (a->e > b->e)
    return 1;  // 返回正数表示 a 大于 b
  else if (a->e < b->e)
    return -1;  // 返回负数表示 a 小于 b
```

[17] qsort, qsort_s - cppreference.com[3]

```
  else
    return 0;  // 返回 0 表示 a 与 b 等价
}
```

这里也可以看出，等价不代表相等，只代表在此比较规则下两元素等价.

std::sort

参见：`std::sort`[4]

用法：

```
// a[0] .. a[n - 1] 为需要排序的数列
// 对 a 原地排序，将其按从小到大的顺序排列
std::sort(a, a + n);

// cmp 为自定义的比较函数
std::sort(a, a + n, cmp);
```

注意：sort 的比较函数的返回值是 `true` 和 `false`，用 `true` 和 `false` 表示两个元素的大小（先后）关系，这与 qsort 的三值比较函数的语义完全不同. 具体内容详见上方给出的 sort 的文档.

若 `cmp` 定义严格弱序，转换为 `qsort` 的三值比较规则时应返回 `cmp(a,b) ? -1 : cmp(b,a) ? 1 : 0`，等价元素必须返回零.

`std::sort` 函数是更常用的 C++ 库排序函数. 该函数的最后一个参数为二元比较函数，未指定 `cmp` 函数时，默认按从小到大的顺序排序.

旧版 C++ 标准中仅要求它的 **平均** 时间复杂度达到 $O(n \log n)$. C++11 标准以及后续标准要求它的 **最坏** 时间复杂度达到 $O(n \log n)$.

C++ 标准并未严格要求此函数的实现算法，具体实现取决于标准库. 例如，libstdc++ 14.2.0[5] 和 libc++ 20.1.8[6] 中的实现都使用了 内省排序→5.7.7@p585.

std::nth_element

参见：`std::nth_element`[7]

用法：

```
std::nth_element(first, nth, last);
std::nth_element(first, nth, last, cmp);
```

它重排 `[first, last)` 中的元素，使得 `nth` 所指向的元素被更改为 `[first, last)` 排好序后该位置会出现的元素. 这个新的 `nth` 元素前的所有元素小于或等于新的 `nth` 元素后的所有元素.

具体实现由标准库决定，常见实现采用 快速选择[8] 或 内省选择[9] 等算法，每次划分后只继续处理包含目标位置的一侧.

对于以上两种用法，C++ 标准要求它的平均时间复杂度为 $O(n)$，其中 n 为 `std::distance(first, last)`.

它常用于构建 K-D Tree→10.19@p2459.

std::stable_sort

参见：`std::stable_sort`[10]

用法：

```
std::stable_sort(first, last);
std::stable_sort(first, last, cmp);
```

稳定排序，保证相等元素排序后的相对位置与原序列相同.

时间复杂度为 $O(n \log^2 n)$，当额外内存可用时，复杂度为 $O(n \log n)$.

std::partial_sort

参见：`std::partial_sort`[11]

用法：

```
// mid = first + k
std::partial_sort(first, mid, last);
std::partial_sort(first, mid, last, cmp);
```

从 `[first,last)` 中选出按 `cmp` 排序最靠前的 `k` 个元素，将它们排序放入 `[first,mid)`；其余部分的顺序不保证. 未指定 `cmp` 函数时，默认按从小到大的顺序排序.

复杂度：约 $(last - first) \log(mid - first)$ 次应用 `cmp`.

原理：

`std::partial_sort` 的思想是：对原始容器内区间为 `[first, mid)` 的元素执行 `make_heap()` 操作，构造一个大根堆，然后将 `[mid, last)` 中的每个元素和 `first` 进行比较，保证 `first` 内的元素为堆内的最大值. 如果小于该最大值，则互换元素位置，并对 `[first, mid)` 内的元素进行调整，使其保持最大堆序. 比较完之后，再对 `[first, mid)` 内的元素做一次堆排序 `sort_heap()` 操作，使其按增序排列. 注意，堆序和增序是不同的.

自定义比较

参见：运算符重载[12]

内置类型（如 `int`）和用户定义的结构体允许定制调用 STL 排序函数时使用的比较函数. 可以在调用该函数时，在最后一个参数中传入一个实现二元比较的函数.

对于用户定义的结构体，对其使用 STL 排序函数前必须定义至少一种关系运算符，或是在使用函数时提供二元比较函数. 通常推荐定义 `operator<`. [18]

示例：

```
int a[1009], n = 10;
// ...
std::sort(a + 1, a + 1 + n);                    // 从小到大排序
std::sort(a + 1, a + 1 + n, greater<int>());  // 从大到小排序
```

```
struct data {
  int a, b;

  bool operator<(const data rhs) const {
    return (a == rhs.a) ? (b < rhs.b) : (a < rhs.a);
  }
} da[1009];

bool cmp(const data u1, const data u2) {
  return (u1.a == u2.a) ? (u1.b > u2.b) : (u1.a > u2.a);
}

// ...
std::sort(da + 1, da + 1 + 10);  // 使用结构体中定义的 < 运算符，从小到大排序
```

[18] 因为大部分标准算法默认使用 `operator<` 进行比较.

```
15 std::sort(da + 1, da + 1 + 10, cmp);  // 使用 cmp 函数进行比较，从大到小排序
```

严格弱序

另请参阅：C++ 中的应用 - 序理论→9.20@p1951

进行排序的运算符必须满足 严格弱序→9.20@p1951，否则会出现不可预料的情况（如运行时错误、无法正确排序）.

常见的错误做法：

- 使用 `<=` 来定义排序中的小于运算符.
- 在调用排序运算符时，读取外部数值可能会改变的数组（常见于最短路算法）.
- 将多个数的最大最小值进行比较的结果作为排序运算符（如皇后游戏/加工生产调度 中的经典错误）.

外部链接

- 浅谈邻项交换排序的应用以及需要注意的问题[13]

参考资料与注释

[1] `qsort` 

[2] `std::qsort` 

[3] qsort, qsort_s - cppreference.com 

[4] `std::sort` 

[5] libstdc++ 14.2.0 

[6] libc++ 20.1.8 

[7] `std::nth_element` 

[8] 快速选择 

[9] 内省选择 

[10] `std::stable_sort` 

[11] `std::partial_sort` 

[12] 运算符重载 

[13] 浅谈邻项交换排序的应用以及需要注意的问题 

5.7.15 排序应用

本页面将简要介绍排序的用法.

理解数据的特点

使用排序处理数据有利于理解数据的特点，方便我们之后的分析与视觉化. 像一些生活中的例子比如词典、菜单，如果不是按照一定顺序排列的话，人们想要找到自己需要的东西的时间就会大大增加.

计算机需要处理大规模的数据，排序后，人们可以根据数据的特点和需求来设计计算机的后续处理流程.

降低时间复杂度

使用排序预处理可以降低求解问题所需要的时间复杂度，通常是一个以空间换取时间的平衡. 如果一个排序好的列表需要被多次分析的话，只需要耗费一次排序所需要的资源是很划算的，因为之后的每次分析都可以减少很多时间.

示例：检查给定数列中是否有相等的元素

考虑一个数列，你需要检查其中是否有元素相等.

一个朴素的做法是检查每一个数对，并判断这一对数是否相等．时间复杂度是 $O(n^2)$.

我们不妨先对这一列数排序，之后不难发现：如果有相等的两个数，它们一定在新数列中处于相邻的位置上．这时，只需要 $O(n)$ 地扫一遍新数列了．

总的时间复杂度是排序的复杂度 $O(n \log n)$.

作为查找的预处理

排序是 二分查找→5.9@p629 所要做的预处理工作．在排序后使用二分查找，可以以 $O(\log n)$ 的时间在序列中查找指定的元素.

5.8 前缀和 & 差分

引入

前缀和与差分是算法竞赛中常用的技巧，前者用于快速求区间和，后者用于高效进行区间修改.

规定

为方便讨论，本文默认数组 $\{a_i\}$ 下标从 1 开始，并补充定义 $a_0 = 0$.

前缀和

前缀和可以简单理解为「数列的前 n 项的和」，是一种重要的预处理方式.

一维前缀和

对于长度为 n 的序列 $\{a_i\}$，如果要多次查询区间 $[l, r]$ 中序列数字的和，就可以考虑使用前缀和．序列的前缀和就是

$$S_i = \sum_{j=1}^{i} a_j.$$

它可以通过递推关系式

$$S_0 = 0,\ S_i = S_{i-1} + a_i$$

逐项计算得到．要询问区间 $[l, r]$ 内的序列的和，只需要计算差值

$$S([l, r]) = S_r - S_{l-1}.$$

就这样，通过 $O(n)$ 时间预处理，能够将单次查询区间和的复杂度降低到 $O(1)$.

参考实现

- C++

```cpp
int n;                // Array size.
std::vector<int> a;   // Array. (indexed from 1)
std::vector<int> ps;  // Prefix sum array.

// Calculate prefix sum.
void prefix_sum() {
  ps = a;
  // Or simply:
  // std::partial_sum(a.begin(), a.end(), ps.begin());
  for (int i = 1; i <= n; ++i) {
    ps[i] += ps[i - 1];
```

```
  }
}

// Query sum of elements in [l, r].
int query(int l, int r) { return ps[r] - ps[l - 1]; }

```

- Python

```
n = 0  # Array size
a = []  # Array (indexed from 1)
ps = []  # Prefix sum array


# Calculate prefix sum
def prefix_sum():
    global ps
    ps = a[:]
    # Or simply:
    # ps = list(itertools.accumulate(a))
    for i in range(1, n + 1):
        ps[i] += ps[i - 1]


# Query sum of elements in [l, r]
def query(l, r):
    return ps[r] - ps[l - 1]

```

C++ 标准库中实现了前缀和函数 `std::partial_sum`[1]，定义于头文件 `<numeric>` 中．从 C++17 开始，标准库还提供了一个功能相同的前缀和函数 `std::inclusive_scan`[2]，同样定义于头文件 `<numeric>` 中．

二维/多维前缀和

将一维前缀和拓展到多维的情形，就是多维前缀和．常见的多维前缀和的求解方法有两种．

基于容斥原理

这种方法多用于二维前缀和的情形．给定大小为 $m \times n$ 的二维数组 A，要求出其前缀和 S．那么，S 同样是大小为 $m \times n$ 的二维数组，且

$$S_{i,j} = \sum_{i' \le i} \sum_{j' \le j} A_{i',j'}.$$

类比一维的情形，$S_{i,j}$ 应该可以基于 $S_{i-1,j}$ 或 $S_{i,j-1}$ 计算，从而避免重复计算前面若干项的和．但是，如果直接将 $S_{i-1,j}$ 和 $S_{i,j-1}$ 相加，再加上 $A_{i,j}$，会导致重复计算 $S_{i-1,j-1}$ 这一重叠部分的前缀和，所以还需要再将这部分减掉．这就是 容斥原理→9.13.3@p1618．由此得到如下递推关系：

$$S_{i,j} = A_{i,j} + S_{i-1,j} + S_{i,j-1} - S_{i-1,j-1}.$$

实现时，直接遍历 (i, j) 求和即可．

示例

考虑一个具体的例子．

$$A = \begin{pmatrix} 1 & 2 & 4 & 3 \\ 5 & 1 & 2 & 4 \\ 6 & 3 & 5 & 9 \end{pmatrix} \qquad S = \begin{pmatrix} 1 & 3 & 7 & 10 \\ 6 & 9 & 15 & 22 \\ 12 & 18 & 29 & 45 \end{pmatrix}$$

其中，S 是矩阵 A 的前缀和．根据定义，$S_{3,3}$ 是左图中虚线方框中的子矩阵的和．而且，$S_{3,2}$ 是蓝色子矩阵的和，$S_{2,3}$ 是红色子矩阵的和，它们重叠部分的和是 $S_{2,2}$．由此可见，如果直接相加 $S_{3,2}$ 和 $S_{2,3}$，会重复计算 $S_{2,2}$，所以应该有

$$S_{3,3} = A_{3,3} + S_{2,3} + S_{3,2} - S_{2,2} = 5 + 18 + 15 - 9 = 29.$$

同样的道理，在已经预处理出二维前缀和后，要查询左上角为 (i_1, j_1)、右下角为 (i_2, j_2) 的子矩阵的和，可以计算

$$S_{i_2,j_2} - S_{i_1-1,j_2} - S_{i_2,j_1-1} + S_{i_1-1,j_1-1}.$$

这可以在 $O(1)$ 时间内完成．

在二维的情形，以上算法的时间复杂度可以简单认为是 $O(mn)$，即与给定数组的大小成线性关系．但是，当维度 k 增大时，由于容斥原理涉及的项数以指数级的速度增长，时间复杂度会成为 $O(2^k N)$，其中 k 是数组维度，而 N 是给定数组大小．因此，该算法不再适用．

洛谷 P1387 最大正方形[3]

在一个 $n \times m$ 的只包含 0 和 1 的矩阵里找出一个不包含 0 的最大正方形，输出边长．

参考代码

- C++

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

int n, m;
std::vector<std::vector<int>> a, ps;  // (n + 1) x (m + 1).

// Calculate the prefix sum of 2-d array.
void prefix_sum() {
  ps = a;
  for (int i = 1; i <= n; ++i)
    for (int j = 1; j <= m; ++j)
      ps[i][j] += ps[i - 1][j] + ps[i][j - 1] - ps[i - 1][j - 1];
}

// Find the sum of elements in submatrix [x1, y1] to [x2, y2].
int query(int x1, int y1, int x2, int y2) {
  return ps[x2][y2] - ps[x1 - 1][y2] - ps[x2][y1 - 1] + ps[x1 - 1][y1
- 1];
}

int main() {
  std::cin >> n >> m;
  a.assign(n + 1, std::vector<int>(m + 1));
```

```cpp

  for (int i = 1; i <= n; i++)
    for (int j = 1; j <= m; j++) std::cin >> a[i][j];

  prefix_sum();

  int ans = 0;
  for (int l = 1; l <= std::min(n, m); ++l)
    for (int i = l; i <= n; i++)
      for (int j = l; j <= m; j++)
        if (query(i - l + 1, j - l + 1, i, j) == l * l) ans =
std::max(ans, l);

  std::cout << ans << std::endl;
  return 0;
}

```

- Python

```python
n, m = 0, 0
a = []  # (n+1) x (m+1)
ps = []  # prefix sum array


# Calculate the prefix sum of 2D array.
def prefix_sum():
    global ps
    ps = [row[:] for row in a]  # Deep copy of a
    for i in range(1, n + 1):
        for j in range(1, m + 1):
            ps[i][j] += ps[i - 1][j] + ps[i][j - 1] - ps[i - 1][j - 1]


# Find the sum of elements in submatrix [x1, y1] to [x2, y2].
def query(x1, y1, x2, y2):
    return ps[x2][y2] - ps[x1 - 1][y2] - ps[x2][y1 - 1] + ps[x1 - 1][y1
- 1]


if __name__ == "__main__":
    n, m = map(int, input().split())

    # Initialize with zero padding for 1-based indexing
    a = [[0] * (m + 1)]
    for _ in range(n):
        row = list(map(int, input().split()))
        a.append([0] + row)

    prefix_sum()
    ans = 0

    for l in range(1, min(n, m) + 1):
```

```
        for i in range(l, n + 1):
            for j in range(l, m + 1):
                if query(i - l + 1, j - l + 1, i, j) == l * l:
                    ans = max(ans, l)

    print(ans)

```

逐维前缀和

对于一般的情形，给定 k 维数组 A，大小为 N，同样要求得其前缀和 S. 这里，

$$S_{i_1,\cdots,i_k} = \sum_{i'_1 \le i_1} \cdots \sum_{i'_k \le i_k} A_{i'_1,\cdots,i'_k}.$$

从上式可以看出，k 维前缀和就等于 k 次求和. 所以，一个显然的算法是，每次只考虑一个维度，固定所有其它维度，然后求若干个一维前缀和，这样对所有 k 个维度分别求和之后，得到的就是 k 维前缀和.

三维前缀和的参考实现

```
int N1, N2, N3;
std::vector<std::vector<std::vector<int>>> a,
    ps;  // (N1 + 1) x (N2 + 1) x (N3 + 1).

// Calculate prefix sum of 3d array.
void prefix_sum() {
  ps = a;

  // Prefix-sum for 3rd dimension.
  for (int i = 1; i <= N1; ++i)
    for (int j = 1; j <= N2; ++j)
      for (int k = 1; k <= N3; ++k) ps[i][j][k] += ps[i][j][k - 1];

  // Prefix-sum for 2nd dimension.
  for (int i = 1; i <= N1; ++i)
    for (int j = 1; j <= N2; ++j)
      for (int k = 1; k <= N3; ++k) ps[i][j][k] += ps[i][j - 1][k];

  // Prefix-sum for 1st dimension.
  for (int i = 1; i <= N1; ++i)
    for (int j = 1; j <= N2; ++j)
      for (int k = 1; k <= N3; ++k) ps[i][j][k] += ps[i - 1][j][k];
}

```

因为考虑每一个维度的时候，都只遍历了整个数组一遍，这样的算法复杂度是 $O(kN)$ 的，通常可以接受.

特例：子集和 DP

维度比较大的情形，经常出现在一类叫做 **子集和**（sum over subsets, SOS）的问题中. 这是高维前缀和的特例.

问题描述如下. 考虑大小为 n 的集合的全体子集上面定义的函数 f，现在要求出其子集和函数 g，它满足

$$g(S) = \sum_{T \subseteq S} f(T).$$

即 $g(S)$ 等于其所有子集 $T \subseteq S$ 上的函数值 $f(T)$ 的和.

首先，子集和问题可以写成高维前缀和的形式. 注意到，任意子集 S 可以通过状态压缩的思想表示为长度为 n 的 0-1 字符串 s，子集 T 对应字符串 t. 将字符串的每一位都看作是数组下标的一个维度，那么 f 其实就是一个 n 维数组，且每个维度下标都一定在 $\{0, 1\}$ 中. 同时，子集的包含关系就等价于下标的大小关系，即

$$T \subseteq S \iff \forall i(t_i \leq s_i).$$

所以，对子集求和，就是求这个 n 维数组的前缀和.

现在，可以直接使用前文所述的逐维前缀和的方法求得子集和. 时间复杂度是 $O(n2^n)$.

参考实现

```cpp
int n;
std::vector<int> a, ps;  // length = 1 << n.

void sum_of_subsets() {
  ps = a;
  // Loop over dimensions.
  for (int i = 0; i < n; ++i) {
    // Loop over i-th dimension.
    for (int st = 0; st < (1 << n); ++st) {
      // This condition implies that i-th dimension is 1.
      if ((st >> i) & 1) {
        // ps[... 1 ...] += ps[... 0 ...]. (i-th dimension)
        ps[st] += ps[st ^ (1 << i)];
      }
    }
  }
}

```

子集和的逆操作需要通过 容斥原理→9.13.3@p1618 进行. 子集和问题也是快速莫比乌斯变换的必要步骤之一.

树上前缀和

一维前缀和还可以推广到有根树（树根为 1）的情形. 通过预处理前缀和，可以快速求解树上一段路径的权值和.

点权的情形

首先讨论权值存储在结点处的情形. 设结点 x 处有权值 a_x. 可以通过递推关系

$$S_1 = a_1,\ S_x = S_{\mathrm{fa}(x)} + a_x$$

求出从根结点到结点 x 的路径上的结点的权值和，其中，$\mathrm{fa}(x)$ 表示 x 的父结点. 预处理完前缀和后，就可以通过

$$S_x + S_y - S_{\mathrm{lca}(x,y)} - S_{\mathrm{fa}(\mathrm{lca}(x,y))}$$

计算连接结点 x 和 y 的路径上的结点权值和. 其中，$\mathrm{lca}(x, y)$ 表示结点 x 与 y 的 最近公共祖先→11.6.5@p2646.

边权的情形

权值储存在边上的情形几乎可以转化为点权的情形．对于所有非根结点 $x \neq 1$，记 $\mathrm{edge}(x)$ 表示连接结点 x 和它的父结点 $\mathrm{fa}(x)$ 的边．那么，可以假设边权存储在离根远的结点上．也就是说，结点 x 处存储的是边 $\mathrm{edge}(x)$ 上的边权．根结点处存储的权值是 0．那么，通过上一小节讨论过的递推关系，同样可以预处理出根结点到结点 x 的路径经过的所有边的权值和 S_x．

此时，连接结点 x 和 y 的路径上的边权和可以通过

$$S_x + S_y - 2S_{\mathrm{lca}(x,y)}$$

进行查询．注意与点权的情形不同，所查询的权值和不包括 $\mathrm{lca}(x,y)$ 处的权值，因为它存储的边权不在所求路径中．

子树和

和数组的情形不同，由于树的首尾不对称，所以自下而上（从树叶到树根）和自上而下（从树根到树叶）求「前缀和」得到的结果并不相同．一般情况下，「树上前缀和」指的是自上而下计算的前缀和．为方便讨论，本文将自下而上计算的「前缀和」称为 **子树和**．

以结点 x 为根的子树的点权权值和，即相应的子树和，就是

$$T_x = \sum_{y \in \mathrm{desc}(x)} a_y.$$

其中，$\mathrm{desc}(x)$ 表示 x 的所有子孙结点（包括其自身）的集合．

与树上前缀和不同，子树和并不能应用于 $O(1)$ 求路径权值和，但是它可以用于理解下文的树上差分．

差分

差分是一种与前缀和相对的策略，是前缀和的逆运算．相较于给定某一序列求它的差分，竞赛中更为常见的情景是，通过维护差分序列的信息，实现多次区间修改．在区间修改结束后，可以通过前缀和恢复原序列的信息，实现对原序列的查询．注意修改操作一定要在查询操作之前．

如果需要支持多次修改和查询的混合操作，需要使用 树状数组→10.12@p2078，但是它们的思想是共通的．

一维差分

对于序列 $\{a_i\}$，它的差分序列 $\{D_i\}$ 是指

$$D_i = a_i - a_{i-1},\ a_0 = 0.$$

C++ 标准库中实现了差分函数 `std::adjacent_difference`[4]，定义于头文件 `<numeric>` 中．

前缀和与差分的关系如下：

性质

设 $\{D_i\}$ 是 $\{a_i\}$ 的差分序列，那么，有

- 序列 $\{a_i\}$ 是序列 $\{D_i\}$ 的前缀和，即

$$a_i = \sum_{j=1}^{i} D_j.$$

- 序列 $\{a_i\}$ 的前缀和为

$$S_i = \sum_{j=1}^{i}\sum_{k=1}^{j} D_k = \sum_{j=1}^{i}(i-j+1)D_j.$$

差分信息常常用于维护多次对序列的一个区间加上一个数，并在之后一次或多次询问序列某一位的取值．

假设要将序列 $\{a_i\}$ 在区间 $[l,r]$ 中的每个数都加上一个 v. 可以在它的差分序列 $\{D_i\}$ 上做如下操作:

$$D_l \leftarrow D_l + v,\ D_{r+1} \leftarrow D_{r+1} - v.$$

在所有修改操作结束后, 可以通过前缀和操作恢复更新后的 $\{a_i\}$ 的值. 单次修改是 $O(1)$ 的. 查询时, 需要做一次 $O(n)$ 的前缀和操作, 随后每次查询都是 $O(1)$ 的.

参考代码

```cpp
int n;
std::vector<int> diff, a;

// Add v to each element in [l, r].
void add(int l, int r, int v) {
  diff[l] += v;
  if (r < n) diff[r + 1] -= v;
}

// Execute this after all modifications and before all queries.
void prefix_sum() {
  for (int i = 1; i <= n; ++i) a[i] = a[i - 1] + diff[i];
}

```

二维/多维差分

差分同样可以推广到多维的情形. 将多维差分看作多维前缀和的逆运算, 那么, 求多维差分数组的操作就相当于根据多维前缀和求它的原数组. 根据前文讨论, 可以利用容斥原理. 例如, 二维差分的定义是

$$D_{i,j} = a_{i,j} - a_{i-1,j} - a_{i,j-1} + a_{i-1,j-1}.$$

但是, 如果要计算整个差分数组, 更为简单高效的做法是逐维差分, 即穷举所有维度, 沿着每个维度都计算一遍数组的差分.

二维差分信息常用于维护二维数组的多次矩形加. 例如, 要对左上角为 (x_1,y_1)、右下角为 (x_2,y_2) 的矩阵中的每个数字都加上 v, 可以对它的差分数组 $\{D_{i,j}\}$ 做如下操作:

$$D_{x_1,y_1} \leftarrow D_{x_1,y_1} + v,$$

$$D_{x_1,y_2+1} \leftarrow D_{x_1,y_2+1} - v,$$

$$D_{x_2+1,y_1} \leftarrow D_{x_2+1,y_1} - v,$$

$$D_{x_2+1,y_2+1} \leftarrow D_{x_2+1,y_2+1} + v.$$

在所有修改操作结束后, 只需要执行一遍二维前缀和, 就可以快速查询更新后的数组的值.

参考代码

```cpp
int n, m;
std::vector<std::vector<int>> diff, a;

// Add v to each element from [x1, y1] to [x2, y2].
void add(int x1, int y1, int x2, int y2, int v) {
  diff[x1][y1] += v;
  if (x2 < n) diff[x2 + 1][y1] -= v;
  if (y2 < m) diff[x1][y2 + 1] -= v;
  if (x2 < n && y2 < m) diff[x2 + 1][y2 + 1] += v;
```

```
}

// Execute this after all modifications and before all queries.
void prefix_sum() {
  a = diff;

  for (int i = 1; i <= n; ++i)
    for (int j = 1; j <= m; ++j) a[i][j] += a[i - 1][j];

  for (int i = 1; i <= n; ++i)
    for (int j = 1; j <= m; ++j) a[i][j] += a[i][j - 1];
}

```

当然，类似的想法对于维度 $k > 2$ 也成立，但是单次修改操作需要的时间复杂度为 $O(2^k)$，随着 k 增大而不再实用.

树上差分

差分可以推广到有根树的情形，用于实现树上一段路径的区间加操作. 取决于维护的信息存储在结点上还是边上，树上差分可以分为 **点差分** 与 **边差分**，在实现上会稍有不同. 另外，相对于树上前缀和操作，更常用的是在所有修改操作后做子树和再查询. 本节讨论的就是这种情形.

点差分

如果要对结点 x 和 y 之间的路径上的所有点权都加 v，可以对它的差分序列 $\{D_x\}$ 做如下操作：

$$D_x \leftarrow D_x + v,$$

$$D_{\mathrm{lca}(x,y)} \leftarrow D_{\mathrm{lca}(x,y)} - v,$$

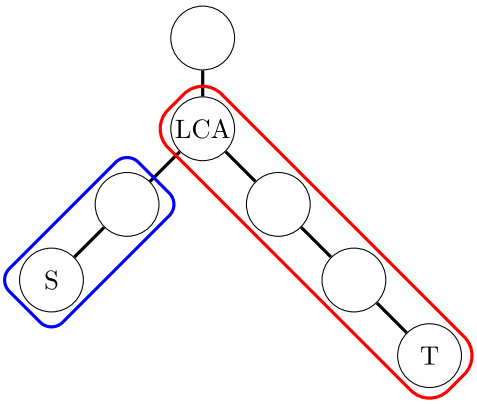
$$D_y \leftarrow D_y + v,$$

$$D_{\mathrm{fa}(\mathrm{lca}(x,y))} \leftarrow D_{\mathrm{fa}(\mathrm{lca}(x,y))} - v.$$

在所有修改操作完成后，可以计算一次子树和，就能得到更新后的点权.

示例

对结点 S 和 T 之间的路径上的点权做区间加操作时，上述公式中的前两条是对蓝色方框内的路径进行一维差分操作，后两条是对红色方框内的路径进行一维差分操作：



自下而上求和，就相当于对这两个区间从下向上计算前缀和. 由此，对比上文的一维差分操作，就能知道点差分操作的正确性.

边差分

如果要对结点 x 和 y 之间的路径上的所有边权都加 v，可以对它的差分序列 $\{D_x\}$ 做如下操作：

$$D_x \leftarrow D_x + v,$$

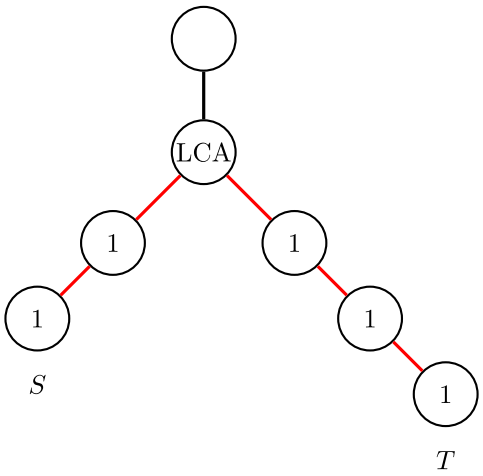
$$D_y \leftarrow D_y + v,$$

$$D_{\mathrm{lca}(x,y)} \leftarrow D_{\mathrm{lca}(x,y)} - 2v.$$

在所有修改操作完成后，可以计算一次子树和，就能得到更新后的边权（存储在相应边的子结点处）.

示例

如图所示，边差分操作可以用于解决红色路径上的边权区间加问题.



由于在边上直接进行差分比较困难，所以将本来应当累加到红色边上的值向下移动到相邻的结点里，操作起来就方便了．对比点差分的公式，就可以理解边差分的公式.

例题

「USACO15DEC」Max Flow[5]

FJ 给他的牛棚的 $N(2 \leq N \leq 50,000)$ 个隔间之间安装了 $N-1$ 根管道，隔间编号从 1 到 N．所有隔间都被管道连通了.

FJ 有 $K(1 \leq K \leq 100,000)$ 条运输牛奶的路线，第 i 条路线从隔间 s_i 运输到隔间 t_i．一条运输路线会给它的两个端点处的隔间以及中间途经的所有隔间带来一个单位的运输压力，你需要计算压力最大的隔间的压力是多少.

解题思路

需要统计每个点经过了多少次，那么就用树上差分将每一次的路径上的点加一，可以很快得到每个点经过的次数．这里采用倍增法计算 LCA，最后用 DFS 遍历整棵树，在回溯时对差分数组求和就能求得答案了.

参考代码

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

```cpp
using namespace std;

constexpr int MAXN = 50010;

struct node {
  int to, next;
} edge[MAXN << 1];

int fa[MAXN][30], head[MAXN << 1];
int power[MAXN];
int depth[MAXN], lg[MAXN];
int n, k, ans = 0, tot = 0;

void add(int x, int y) {  // 加边
  edge[++tot].to = y;
  edge[tot].next = head[x];
  head[x] = tot;
}

void dfs(int now, int father) {  // DFS 预处理结点深度和倍增祖先
  fa[now][0] = father;
  depth[now] = depth[father] + 1;
  for (int i = 1; i <= lg[depth[now]]; ++i)
    fa[now][i] = fa[fa[now][i - 1]][i - 1];
  for (int i = head[now]; i; i = edge[i].next)
    if (edge[i].to != father) dfs(edge[i].to, now);
}

int lca(int x, int y) {  // 求LCA，最近公共祖先
  if (depth[x] < depth[y]) swap(x, y);
  while (depth[x] > depth[y]) x = fa[x][lg[depth[x] - depth[y]] - 1];
  if (x == y) return x;
  for (int k = lg[depth[x]] - 1; k >= 0; k--) {
    if (fa[x][k] != fa[y][k]) x = fa[x][k], y = fa[y][k];
  }
  return fa[x][0];
}

// 用 DFS 累加子树差分，求出每个结点的压力并更新最大压力
void get_ans(int u, int father) {
  for (int i = head[u]; i; i = edge[i].next) {
    int to = edge[i].to;
    if (to == father) continue;
    get_ans(to, u);
    power[u] += power[to];
  }
  ans = max(ans, power[u]);
}

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  cin >> n >> k;
```

```
  int x, y;
  for (int i = 1; i <= n; i++) {
    lg[i] = lg[i - 1] + (1 << lg[i - 1] == i);
  }
  for (int i = 1; i <= n - 1; i++) {  // 建图
    cin >> x >> y;
    add(x, y);
    add(y, x);
  }
  dfs(1, 0);
  int s, t;
  for (int i = 1; i <= k; i++) {
    cin >> s >> t;
    int ancestor = lca(s, t);
    // 树上差分
    power[s]++;
    power[t]++;
    power[ancestor]--;
    power[fa[ancestor][0]]--;
  }
  get_ans(1, 0);
  cout << ans << '\n';
  return 0;
}

```

习题

前缀和：

- 洛谷 B3612【深进 1. 例 1】求区间和[6]
- 洛谷 U69096 前缀和的逆[7]
- AtCoder joi2007ho_a 最大の和[8]
- 「USACO16JAN」Subsequences Summing to Sevens[9]
- 「USACO05JAN」Moo Volume S[10]

二维/多维前缀和：

- HDU 6514 Monitor[11]
- 洛谷 P1387 最大正方形[12]
- 「HNOI2003」激光炸弹[13]
- CF 165E Compatible Numbers[14]
- CF 383E Vowels[15]
- ARC 100C Or Plus Max[16]

树上前缀和：

- LOJ 10134.Dis[17]
- LOJ 2491. 求和[18]

差分：

- 树状数组 3：区间修改，区间查询[19]
- 「Poetize6」IncDec Sequence[20]
- 洛谷 P4231 三步必杀[21]

二维/多维差分：

- 洛谷 P3397 地毯[22]
- 洛谷 P8228「Wdoi-5」模块化核熔炉[23]

树上差分：

- 「USACO15DEC」 Max Flow[24]
- JLOI2014 松鼠的新家[25]
- NOIP2015 运输计划[26]
- NOIP2016 天天爱跑步[27]

外部参考资料

[1] `std::partial_sum`

[2] `std::inclusive_scan`

[3] 洛谷 P1387 最大正方形

[4] `std::adjacent_differenc`

[5]「USACO15DEC」Max Flow

[6] 洛谷 B3612【深进 1. 例 1】求区间和

[7] 洛谷 U69096 前缀和的逆

[8] AtCoder joi2007ho_a 最大の和

[9] 「USACO16JAN」Subsequences Summing to Sevens

[10] 「USACO05JAN」 Moo Volume S

[11] HDU 6514 Monitor

[12] 洛谷 P1387 最大正方形

[13]「HNOI2003」激光炸弹

[14] CF 165E Compatible Numbers

[15] CF 383E Vowels

[16] ARC 100C Or Plus Max

[17] LOJ 10134.Dis

[18] LOJ 2491. 求和

[19] 树状数组 3：区间修改，区间查询

[20] 「Poetize6」 IncDec Sequence

[21] 洛谷 P4231 三步必杀

[22] 洛谷 P3397 地毯

[23] 洛谷 P8228「Wdoi-5」模块化核熔炉

[24] 「USACO15DEC」 Max Flow

[25] JLOI2014 松鼠的新家

[26] NOIP2015 运输计划

[27] NOIP2016 天天爱跑步

5.9 二分

本页面将简要介绍二分查找、二分答案，以及由二分法衍生的三分法.

二分查找

二分查找（binary search），也称折半搜索（half-interval search）、对数搜索（logarithmic search），是用来在一个有序序列中查找某一元素的算法.

过程

以在一个升序数组中查找一个数为例.

它每次访问数组当前部分的中间元素，如果中间元素刚好是要找的，就结束搜索过程；如果中间元素小于所查找的值，那么左侧的元素都不大于中间元素，不可能有所查找的元素，只需到右侧查找；如果中间元素大于所查找的值同理，只需到左侧查找.

具体地，设数组 a 下标从 1 开始，长度为 n，要查找数 x 在哪个位置. 令 l, r 表示当前只考虑数组中下标 i 满足 $l \le i \le r$ 的数. 一开始时，令 $l \leftarrow 1$，$r \leftarrow n$. 每次访问中间元素 a_{mid}，其中 $mid = \left\lfloor \dfrac{l+r}{2} \right\rfloor$，然后分情况讨论：

- 若 $a_{mid} < x$，由于数组升序，下标小于等于 mid 的数一定都比 x 小，因此一定不用查找这些数，只需要查找下标大于 mid 的数就可以了，也就是令 $l \leftarrow mid + 1$，r 不变.
- 若 $a_{mid} > x$，由于数组升序，下标大于等于 mid 的数一定都比 x 大，因此一定不用查找这些数，只需要查找下标小于 mid 的数就可以了，也就是令 l 不变，$r \leftarrow mid - 1$.
- 若 $a_{mid} = x$，我们就找到了数 x 的位置，算法结束.

如果直到考虑范围为空，即 $l > r$ 时都没有找到数 x 的位置，就说明 x 不在数组 a 中.

性质

时间复杂度

二分查找的最优时间复杂度为 $O(1)$.

二分查找的平均时间复杂度和最坏时间复杂度均为 $O(\log n)$. 因为在二分查找过程中，算法每次都把待查找的区间减半，所以对于一个长度为 n 的数组，至多会进行 $O(\log n)$ 次和目标元素的比较.

空间复杂度

迭代版本的二分查找的空间复杂度为 $O(1)$.

递归（无尾调用消除）版本的二分查找的空间复杂度为 $O(\log n)$.

实现

```cpp
int binary_search(int x, int l = 1, int r = n) {  // 在升序数组中查找数 x 的下标
  int ret = -1;                                   // 未找到时返回 -1
  while (l <= r) {
    int mid = (l + r) >> 1;  // l + r 可能溢出，详见下方 Note
    if (a[mid] < x)
      l = mid + 1;
    else if (a[mid] > x)
      r = mid - 1;
    else {  // 最后检测相等是因为多数搜索情况不是大于就是小于
      ret = mid;
      break;
    }
  }
  return ret;
}
```

Note

- 参考 编译优化 # 移位代替乘法→4.4.10@p456, 对于 s 是有符号数的情况, 当你可以保证 $s \geq 0$ 时, `s >> 1` 比 `s / 2` 指令数更少.
- 当 l 或 r 特别大时，$l + r$ 可能会溢出. 若此时 $r - l$ 不会溢出，可以将代码中的 `(l + r) >> 1` 更换为 `l + ((r - l) >> 1)`.

Warning

当 s 为负奇数时，`s >> 1` 和 `s / 2` 的结果相差 1，并不相同. 这是因为前者向负无穷取整（C++20 列入标准，此前为实现定义），后者向零取整. 详见 C++ 位操作符→4.2.4@p339. 因此 $l + r$ 可能为负时，`(l + r) / 2` 可能取到 r，在 `r = mid` 的写法中会导致 **死循环**. 例如 $l = -1$，$r = 0$ 时，`(l + r) / 2` 等于 0. 在实现代码时需要注意这点不同.

bsearch

`bsearch` 函数为 C 标准库实现的二分查找，定义在 `<stdlib.h>` 中．在 C++ 标准库里，该函数定义在 `<cstdlib>` 中．qsort 和 `bsearch` 是 C 语言标准库中仅有的两个算法类函数．

`bsearch` 函数相比 qsort（排序相关 STL→5.7.14@p613）的四个参数，在最左边增加了参数「待查元素的地址」．之所以按照地址的形式传入，是为了方便直接套用与 qsort 相同的比较函数，从而实现排序后的立即查找．因此这个参数不能直接传入具体值，而是要先将待查值用一个变量存储，再传入该变量地址．

于是 `bsearch` 函数总共有五个参数：待查元素的地址、数组首地址、元素个数、元素大小、比较规则．比较规则仍然通过指定比较函数实现，详见 排序相关 STL→5.7.14@p613．

`bsearch` 函数的返回值是查找到的元素的地址，返回类型为 `void *`．

注意：`bsearch` 与下文将要介绍的 `std::lower_bound` 和 `std::upper_bound` 有两点不同：

- 当有多个符合条件的元素时，返回其中哪一个未指定．
- 当查找不到相应的元素时，会返回 `NULL`．

用 `lower_bound` 可以实现与 `bsearch` 近乎相同的功能（只需要特别判断查找不到元素的情况即可做到完全相同），所以可以使用 `bsearch` 通过的题目，直接改写成 `lower_bound` 同样可以实现．

二分答案

二分答案是利用问题答案的广义有序性质（通常也被称为单调性），通过类似二分查找的方式快速求出答案的算法．

如果没有特殊说明，二分答案通常代指整数范围的二分答案，即已知答案必定是整数．

能够利用二分答案的问题通常形如「求满足条件 P 的最大（小）值」，并具有如下性质：

1. 如果给出任意数 x，容易判断 x 是否满足条件；
2. 问题的答案可以划定一个粗略的上下界，即可以确定 L 和 R 使得若答案 x 存在，则 x 必然满足 $L \le x \le R$；
3. 划定的粗略的上下界范围很大，一个个枚举并且判断会超时；
4. 条件 P 具有广义的有序性质．

假设存在一个函数 $f(x)$，当且仅当 x 满足条件 P 时 $f(x) = 1$，否则 $f(x) = 0$，则可以用下列方式定义条件 P 的广义有序性质：

- 若对于任意的 $L \le i < j \le R$，$f(i) \le f(j)$，即 $f(x)$ 单调不减．将 $i = L, ..., R$ 的 $f(i)$ 写成一个 01 序列时形如 `00...011...1`．此时二分答案可以求出满足条件 P 的 **最小值**．
- 若对于任意的 $L \le i < j \le R$，$f(i) \ge f(j)$，即 $f(x)$ 单调不增．对应的 01 序列形如 `11...100...0`．此时二分答案可以求出满足条件 P 的 **最大值**．

换言之，第一种有序性质是指：如果知道 x 满足条件 P，则所有比 x 大的数一定都满足条件 P．二分答案可以求出符合「比它小的数都不满足条件 P，它和比它大的数都满足条件 P」这一描述的 x．

而第二种有序性质是指：如果知道 x 满足条件 P，则所有比 x 小的数一定都满足条件 P．二分答案可以求出符合「它和比它小的数都满足条件 P，比它大的数都不满足条件 P」这一描述的 x．

过程

以利用二分答案求最小值为例（此时算法要求问题满足上文所述的第一种有序性质）．设答案的粗略上下界为 L 和 R．

令 l, r 表示当前可以确定答案 x 一定满足 $l \le x \le r$．和二分查找类似，一开始时，令 $l \leftarrow L$，$r \leftarrow R$．每次我们判断 $mid = \left\lfloor \dfrac{l+r}{2} \right\rfloor$ 是否满足条件 P，然后分情况讨论：

- 如果 mid 不满足条件（即 $f(mid)$ 为 0），根据问题的有序性质，此时小于等于 mid 的所有数都不满足条件，都无需考虑，因此令 $l \leftarrow mid + 1$，r 不变．

- 如果 mid 满足条件（即 $f(mid)$ 为 1），根据问题的有序性质，此时大于等于 mid 的所有数都满足条件，但是因为题目要求最小值，所以所有大于 mid 的数都没必要考虑了，只考虑小于等于 mid 的数就可以了．此时令 $r \leftarrow mid$，l 不变．

当答案范围满足 $l = r$（即不再满足 $l < r$）时，算法结束．此时 l 或 r 为答案．

需要注意的是：若上下界内答案不存在（即通常所说的无解），每次查询 $f(mid)$ 都会返回 0，因此算法结束后 $l = r = R$，且 $f(l) = 0$．因此如果需要判断无解，额外检查 $f(l)$ 是否等于 0 即可．

和二分查找类似，算法每次都把待搜索的区间减半，因此算法的时间复杂度为 $O(M \log(R - L + 1))$，其中 M 为每次判断 mid 是否满足条件 P 的时间复杂度．

实现

根据上面的算法描述，可以给出如下实现：

```
// 求 [L, R] 内满足条件 P 的最小整数 x，要求 L <= R
// 条件 P 需要满足第一种有序性质，即 f(x) 单调不减
// 在代码实现中，通常 check(x) = f(x)
// 若区间内无解，返回 -1
int binary_search_min(int L, int R) {
  if (L > R) return -1;
  int l = L, r = R;
  while (l < r) {
    int mid = (l + r) >> 1;
    if (check(mid))  // f(mid) = 1，满足条件 P
      r = mid;       // 答案在 [l, mid]
    else
      l = mid + 1;  // 答案在 [mid + 1, r]
  }
  // 此时 l == r
  if (!check(l)) return -1;  // 无解判断
  return l;
}
```

使用 `-1` 表示无解时，需要保证它不会与合法答案混淆．

实现细节

在查看题解时，我们可能会看到另一种实现方式：

```
// 求 [L, R] 内满足 check(x) 的最小整数 x
// 要求 check 在 [L, R] 上单调不减
// 若区间为空或区间内无解，返回 -1
int binary_search_min(int L, int R) {
  if (L > R) return -1;
  int l = L, r = R;
  while (l <= r) {
    int mid = (l + r) >> 1;
    if (check(mid))
      r = mid - 1;
    else
      l = mid + 1;
  }

  if (l > R) return -1;
  return l;
}
```

这两种写法求出的答案是相同的，但维护的循环不变量不同.

为什么两种写法都能找到最小可行值?

先假设 $[L,R]$ 非空，且区间内存在答案，记最小的满足条件 P 的数为 ans. 由 f 单调不减的性质，在原区间 $[L,R]$ 内有

$$f(x)=0 \quad (x<ans), \qquad f(x)=1 \quad (x\geq ans).$$

`l < r` 的写法将答案保留在闭区间内.

它维护的不变量是

$$l\leq ans\leq r.$$

每次取 $mid=\left\lfloor\dfrac{l+r}{2}\right\rfloor$. 若 $f(mid)=1$，则 $ans\leq mid$，令 $r\leftarrow mid$；若 $f(mid)=0$，则 $ans>mid$，令 $l\leftarrow mid+1$. 两种更新都保持不变量.

当 $l<r$ 时，有 $l\leq mid<r$，因此每次迭代都会使区间严格缩小. 循环结束时 $l=r$，由不变量可知 $ans=l$.

`l <= r` 的写法排除已经确定的部分，答案可能位于右边界的后一位.

它维护的不变量是：在原区间 $[L,R]$ 内，所有 $x<l$ 都满足 $f(x)=0$，所有 $x>r$ 都满足 $f(x)=1$. 在答案存在的前提下，这意味着

$$l\leq ans\leq r+1.$$

若 $f(mid)=1$，则原区间内所有 $x\geq mid$ 都满足 $f(x)=1$，可以令 $r\leftarrow mid-1$；若 $f(mid)=0$，则原区间内所有 $x\leq mid$ 都满足 $f(x)=0$，可以令 $l\leftarrow mid+1$.

每次迭代都会从待搜索区间中排除至少一个整数，因此循环最终会结束.

而循环执行时，有 $l\leq mid\leq r$. 若执行 $r\leftarrow mid-1$，则新的右边界满足 $r\geq l-1$；若执行 $l\leftarrow mid+1$，则新的左边界满足 $l\leq r+1$. 因此，无论采用哪种更新，更新后都有 $l\leq r+1$. 循环在 $l>r$ 时结束，结合整数边界的性质可知，此时必有 $l=r+1$. 再由答案位置不变量 $l\leq ans\leq r+1$，得到 $ans=l$.

这里需要注意：待搜索区间 $[l,r]$ 不一定始终包含答案. 例如，若本轮恰好取到 $mid=ans$，执行 $r\leftarrow mid-1$ 后，就有 $ans=r+1$. 此时答案虽然已经不在 $[l,r]$ 内，却仍然满足不变量 $l\leq ans\leq r+1$，因此不会影响上述正确性证明.

无解时，两种写法的终止位置不同.

对于 `l <= r` 的写法，若循环结束后有 $l\leq R$，由于左边界始终满足 $l\geq L$，可知 l 位于原区间内. 再由 $l=r+1$ 以及「原区间内所有 $x>r$ 都满足 $f(x)=1$」这一不变量，可知 $f(l)=1$，因此不必再次调用 `check(l)`.

若无解，则最终有 $l=R+1$，已经超出原搜索区间. 由于我们只要求 `check` 在 $[L,R]$ 内可用，因此应先检查 `l > R`，不能直接调用 `check(l)` 来判断是否有解.

而对于 `l < r` 的写法，回顾前面的描述可以得知，如果在区间内无解，则最终 $l=r=R$. 此时需要额外判断 `check(l)` 是否为 0 来区别到底是答案为 R 还是区间内无解.

综上，在 $L\leq R$ 且 f 在 $[L,R]$ 上单调不减的前提下，两种写法都能在有限次迭代后结束. 若区间内存在可行值，它们最终都得到 $l=ans$，返回区间内满足条件的最小整数；若区间内无解，`l < r` 的写法通过结束后的 `check(l)` 判定无解，`l <= r` 的写法通过结束后的 `l > R` 判定无解，两者均返回约定的无解标记. 对于 $L>R$ 的空区间，两段代码都会在进入循环前直接返回无解标记. 因此，两种实现虽然维护的循环不变量和终止位置不同，但都能正确完成所要求的最小可行值查询，并得到相同的返回结果.

这两种写法说明，二分实现的正确性取决于边界含义、循环不变量、循环条件和更新规则之间的配合．仅根据区间端点是否包含在内，或者循环使用 `l < r` 还是 `l <= r`，还不足以判断一种写法是否正确．

证明一种写法正确，需要确认：

1. 初始化满足循环不变量．
2. 每次更新都保持不变量，并使待搜索区间中的整数个数严格减少，从而保证循环最终结束．
3. 循环结束时，能够由不变量和终止条件确定返回值就是答案；若允许无解，还需说明如何识别无解情况．

在讨论实现方式时，需要特别区分「尚待搜索的区间」与「保证包含最终答案的区间」：由于边界条件不同，二者不一定相同，不能统一要求答案始终位于待搜索区间内．

下面列出几种常见写法．均假设 $L \le R$、问题满足第一种有序性质，且最小可行值 ans 存在．答案的位置以「答案位置不变量」一列为准．

写法	初始 l, r	循环条件	mid	$f(mid)=1$ 时	$f(mid)=0$ 时	答案位置不变量	结束时	返回
闭区间 $[l,r]$，排除已判定部分	$l=L, r=R$	$l \le r$	$\left\lfloor \frac{l+r}{2} \right\rfloor$	$r \leftarrow mid-1$	$l \leftarrow mid+1$	$l \le ans \le r+1$	$l=r+1$	l
闭区间 $[l,r]$，保留答案	$l=L, r=R$	$l < r$	$\left\lfloor \frac{l+r}{2} \right\rfloor$	$r \leftarrow mid$	$l \leftarrow mid+1$	$l \le ans \le r$	$l=r$	l
左闭右开区间 $[l,r)$，排除已判定部分	$l=L, r=R+1$	$l < r$	$\left\lfloor \frac{l+r}{2} \right\rfloor$	$r \leftarrow mid$	$l \leftarrow mid+1$	$l \le ans \le r$	$l=r$	l
左开右闭区间 $(l,r]$，保留答案	$l=L-1, r=R$	$l+1 < r$	$\left\lfloor \frac{l+r}{2} \right\rfloor$	$r \leftarrow mid$	$l \leftarrow mid$	$l < ans \le r$	$l+1=r$	r
开区间 (l,r)，维护两侧边界	$l=L-1, r=R+1$	$l+1 < r$	$\left\lfloor \frac{l+r}{2} \right\rfloor$	$r \leftarrow mid$	$l \leftarrow mid$	$l < ans \le r$	$l+1=r$	r

表中还有几点需要说明：

- 第 3 行的 $[l,r)$ 是待搜索区间，但答案可能恰好等于 r．
- 第 4 行将答案保留在 $(l,r]$ 内，依赖初始右边界 R 满足 $f(R)=1$．循环执行时 $r-l \ge 2$，因此第 4、5 行的中点改用上取整也正确．
- 第 5 行中的 (l,r) 表示两个边界之间尚待判断的位置．可以将初始的 $L-1$ 视为值为 0 的虚拟哨兵，将 $R+1$ 视为值为 1 的虚拟哨兵．循环只在原区间内调用判定函数，不需要实际计算哨兵处的函数值．
- 若要处理无解情况：第 1、3、5 行可以通过返回值是否为 $R+1$ 判断；第 2 行需要检查 $f(l)$；第 4 行需要先检查 $f(R)$．

最大值最小化与最小值最大化

最大值最小化与最小值最大化问题是典型的能够应用二分答案算法解决的问题．

以最大值最小化问题为例．通常每个方案对应一个需要考虑的集合 S．若要求在所有方案中，使得方案对应集合 S 中数最大值最小化，可以转化问题为：求最小的 k，使得存在方案满足 $\max(S) \le k$．

而这个问题具备广义有序性质：若存在方案满足 $\max(S) \le k$，则对于 $i \ge k$，该方案也满足 $\max(S) \le i$．因此对于 $i \ge k$，存在方案满足 $\max(S) \le i$，同样符合题目条件．于是可以通过二分答案解决．最小值最大化问题同理．

STL 的二分答案

std::lower_bound 与 std::upper_bound

C++ 标准库中实现了：

- 查找首个不小于给定值的元素的函数 `std::lower_bound`[1].
- 查找首个大于给定值的元素的函数 `std::upper_bound`[2].

二者均采用二分实现，所以调用前必须保证元素有序（注意这里的有序是指基于下面的比较函数而言的，不一定是数学意义上的有序），这样二者的问题才满足广义的有序性质（即对于 `std::lower_bound`，若 a_i 不小于给定值时，i 之后的数同样不小于给定值；对于 `std::upper_bound`，若 a_i 大于给定值时，i 之后的数同样大于给定值）.

`std::lower_bound` 和 `std::upper_bound` 均有四个参数，分别是：

- `first`：序列的起始 迭代器→4.3.2.2@p371，指向序列的首个元素.
- `last`：序列的终止迭代器，指向序列的最后一个元素的 **后一位置**. 换言之，如果 `last` 是双向迭代器，则 `--last` 指向的是序列的最后一个元素.
- `value`：给定值.
- `comp`（可选）：比较函数，参考 `sort` 函数的书写方式. 需要注意 `std::lower_bound` 以 `comp(元素, value)` 的形式调用，而 `std::upper_bound` 以 `comp(value, 元素)` 的形式调用.

二者的返回值均为满足条件的元素的迭代器，类型同传入的一致. 也就是说，如果传入数组指针，则返回满足条件的元素对应的数组指针. 如果找不到满足条件的元素，返回 `last`.

二者均定义于头文件 `<algorithm>` 中.

用法示例

- 在下标从 1 开始，长度为 n 的数组 a 中下标 l 到 r 的位置查找首个不小于 x 的数，并获取这个数的下标：`lower_bound(a+l,a+r+1,x)-a`.
- 在下标从 0 开始，长度为 n 的数组 a 中查找首个大于 x 的数（需保证这个数存在），并获取这个数的值：`*upper_bound(a,a+n,x)`.
- 在长度为 n 的 vector a 中查找首个不小于 x 的数并获取这个数的下标（注意 vector 下标从 0 开始）：`lower_bound(a.begin(),a.end(),x)-a.begin()`.

关于迭代器

上述起始、终止迭代器必须是 ForwardIterator. 数组指针以及 vector、set、map、string 的迭代器均符合这一迭代器要求.

关于算法的时间复杂度

在 GCC 使用的 libstdc++ 标准库实现中，二者均使用 `std::advance` 来访问中间元素. 这意味着当迭代器支持随机访问时（比如传入数组或是 vector 迭代器），函数的复杂度才是 $O(\log n)$ 的. 如果不支持随机访问（比如 set 或者 map），函数的复杂度为每次查询中间元素的时间复杂度之和（通常为线性）. 例如，在 set 或者 map 中，执行 `lower_bound(st.begin(),st.end(),val)` 类似的操作，时间复杂度为 $O(n)$. 此时应改用成员函数 `st.lower_bound(val)`.

利用 bsearch 实现 std::lower_bound 与 std::upper_bound

由于 bsearch 在查找不到元素时会返回 NULL（见 bsearch→5.9@p629），例如，在序列 1、2、4、5、6 中查找 3，`bsearch` 实现 `lower_bound` 的功能会变得困难.

利用 `bsearch` 实现 `std::lower_bound` 和 `std::upper_bound` 时，可以利用其比较函数的参数约定：第一个参数指向待查元素，第二个参数指向待查数组中的元素。所以只要比较函数能得到数组首地址即可实现。

```
int A[100005];  // 示例全局数组

// compare 比较两个 int 指针指向的值：*p1 > *p2 返回正数，相等返回
// 0，小于返回负数
int compare(const void*, const void*);

// 查找首个不小于待查元素的元素的地址
int lower(const void* p1, const void* p2) {
  int* a = (int*)p1;
  int* b = (int*)p2;
  if ((b == A || compare(a, b - 1) > 0) && compare(a, b) > 0)
    return 1;
  else if (b != A && compare(a, b - 1) <= 0)
    return -1;  // 用到地址的减法，因此必须指定元素类型
  else
    return 0;
}

// 查找首个大于待查元素的元素的地址
int upper(const void* p1, const void* p2) {
  int* a = (int*)p1;
  int* b = (int*)p2;
  if ((b == A || compare(a, b - 1) >= 0) && compare(a, b) >= 0)
    return 1;
  else if (b != A && compare(a, b - 1) < 0)
    return -1;  // 用到地址的减法，因此必须指定元素类型
  else
    return 0;
}
```

注意：若答案是尾后位置（如待查元素大于所有元素），上述方法仍会返回 `NULL`，需要单独处理。

因为现在的 OI 选手很少写纯 C，并且此方法作用有限，所以不是重点。对于新手而言，建议直接使用 C++ 中的 `std::lower_bound` 和 `std::upper_bound` 函数。

std::partition_point

C++11 引入了 `std::partition_point`[3]。其作用是在一个已分区的序列中，通过二分答案快速定位「分区点」。

`std::partition_point` 有三个参数，分别是：

- `first`、`last`：同上。
- `p`：一个一元 谓词→4.3.2.1@p370。这是一个可调用对象，支持传入一个参数 v，并返回一个布尔值表示 v 是否符合分区条件。

设传入的序列为 a，则该函数返回第一个不满足分区条件的元素的迭代器，即返回迭代器指向下标 x 最小的满足 $p(a_x)$ 等于 `false` 的元素。

该序列需要已分区，即需要满足上文所述第二种广义有序性质。换言之，将序列中每个元素 v 的 $p(v)$ 的结果列成一个 01 序列，则该序列形如 `11...100...0`，该函数返回第一个 0 对应位置的迭代器。

std::partition_point 与 std::lower_bound 和 std::upper_bound 的关系

实际上，`std::lower_bound` 和 `std::upper_bound` 是 `std::partition_point` 的特殊形式.

定义函数 `f`，其代码为：`bool f(int v) { return !(val <= v); }`. 将 `f` 作为谓词传入 `std::partition_point` 中，即可得到和 `std::lower_bound` 相同的结果. `std::upper_bound` 同理.

实数二分答案

实数二分答案，也称浮点二分，是二分答案在答案为实数时的一种形式.

与整数二分答案不同，实数二分答案通常不要求求出精确的答案，而是求一个满足给定精度要求的实数近似值.

过程

以求最小值为例. 设答案的粗略上下界为 L 和 R. 记 l, r 表示当前可以确定答案 x 一定满足 $l \le x \le r$. 一开始时，令 $l \leftarrow L$，$r \leftarrow R$. 每次取 $mid = \dfrac{l+r}{2}$（注意，这里是实数运算），判断 mid 是否满足条件 P:

- 若 mid 满足条件，与整数情形类似，令 $r \leftarrow mid$，l 不变.
- 若 mid 不满足条件，与整数情形类似，令 $l \leftarrow mid$，r 不变.

需要注意的是，与整数二分答案不同，实数二分答案不能通过 `mid + 1` 或 `mid - 1` 缩小区间，因为实数域中不存在相邻元素；只能令边界等于 mid，依赖区间长度不断减半来逼近答案.

当区间长度 $r - l$ 不超过给定精度 eps，或达到预设的迭代次数时，算法结束. 此时 l、r 或 $\dfrac{l+r}{2}$ 均可作为答案的近似值（若要求最终返回值满足条件 P，应返回 r）. 求最大值时，只需将上述两种情况的方向反过来：若 mid 满足条件，则令 $l \leftarrow mid$；否则令 $r \leftarrow mid$（相应地，若要求最终返回值满足条件 P，应返回 l）.

若采用 `while (r - l > eps)`，实数二分答案的时间复杂度为 $O(M \log((R-L)/eps))$. 若采用固定迭代次数 k，则时间复杂度为 $O(Mk)$. 其中 M 为每次判断 mid 是否满足条件 P 的时间复杂度. 由于实数运算存在浮点误差，实际实现中通常不直接判断 $l = r$，而是判断 $r - l < eps$，或直接循环固定次数，例如 60 至 100 次，以避免死循环并保证精度.

实现

```cpp
double binary_search_iter(double L, double R) {  // 固定迭代次数实现
  double l = L, r = R;
  for (int i = 0; i < 100; ++i) {
    double mid = (l + r) / 2;
    if (check(mid))
      r = mid;
    else
      l = mid;
  }
  return r;
}

const double eps = 1e-7;  // 精度要求，通常取题目要求精度的 1/100 或更小

double binary_search_eps(double L, double R) {  // eps 实现
  double l = L, r = R;
  while (r - l > eps) {
    double mid = (l + r) / 2;
```

```
    if (check(mid))
      r = mid;
    else
      l = mid;
  }
  return r;
}
```

Warning

eps 不宜过大，否则精度不足；也不宜过小，否则可能因浮点误差无法达到而死循环．若答案范围很大或精度要求很高，建议使用固定迭代次数而非 `while (r - l > eps)`．

例题

Luogu P1873 砍树[4]

伐木工人米尔科需要砍倒 M 米长的木材．这是一个对米尔科来说很容易的工作，因为他有一个漂亮的新伐木机，可以像野火一样砍倒森林．不过，米尔科只被允许砍倒单行树木．

米尔科的伐木机工作过程如下：米尔科设置一个高度参数 H（米），伐木机升起一个巨大的锯片到高度 H，并锯掉所有的树比 H 高的部分（当然，树木不高于 H 米的部分保持不变）．米尔科就得到树木被锯下的部分．

例如，如果一行树的高度分别为 20，15，10，17，米尔科把锯片升到 15 米的高度，切割后树木剩下的高度将是 15，15，10，15，而米尔科将从第一棵树得到 5 米木材，从第四棵树得到 2 米木材，共 7 米木材．

米尔科非常关注生态保护，所以他不会砍掉过多的木材．这正是他尽可能高地设定伐木机锯片的原因．你的任务是帮助米尔科找到伐木机锯片的最大的整数高度 H，使得他能得到木材至少为 M 米．即，如果再升高 1 米锯片，则他将得不到 M 米木材．

解题思路

我们可以在 0 到 10^9 中枚举答案，但是这种朴素写法肯定拿不到满分，因为从 0 枚举到 10^9 太耗时间．我们可以在 $[0,\ 10^9]$ 的区间上进行二分作为答案，然后检查各个答案的可行性（一般使用贪心法）．**这就是二分答案．**

参考代码

```
#include <iostream>
using namespace std;

int a[1000005];
int n, m;

bool check(int k) {  // 检查可行性，k 为锯片高度
  long long sum = 0;
  for (int i = 1; i <= n; i++)        // 检查每一棵树
    if (a[i] > k)                     // 如果树高于锯片高度
      sum += (long long)(a[i] - k);   // 累加树木长度
```

```
12    return sum >= m;                    // 如果满足最少长度代表可行
13  }
14
15  int find() {
16    int l = 0, r = 1e9 + 1;    // 因为是左闭右开的，所以 10^9 要加 1
17                               // 保持 check(l) 为真、check(r) 为假
18    while (l + 1 < r) {        // 如果两点不相邻
19      int mid = (l + r) / 2;   // 取中间值
20      if (check(mid))          // 如果可行
21        l = mid;               // 升高锯片高度
22      else
23        r = mid;   // 否则降低锯片高度
24    }
25    return l;   // 返回左边值
26  }
27
28  int main() {
29    cin >> n >> m;
30    for (int i = 1; i <= n; i++) cin >> a[i];
31    cout << find();
32    return 0;
33  }
34
```

看完了上面的代码，你肯定会有两个疑问：

1. 为何搜索区间是左闭右开的?

   因为搜到最后，会这样（以合法的最大值为例）：

合法			不合法			
最小值	L	MID	R	.....	.....	最大值

或者

合法				不合法		
最小值	.....	.....	L	MID	R	最大值

然后会

合法			不合法			
最小值	.....	L,MID	R	.....	.....	最大值

或者

合法				不合法		
最小值	.....	.....	L	MID,R	.....	最大值

合法的最小值恰恰相反.

2. 为何返回左边值?

同上. 循环结束时 $l+1=r$, `check(l)` 为真而 `check(r)` 为假, 因此 l 就是合法的最大值.

三分法

引入

二分法可以用于近似求出函数的零点. 如果需要求出单峰函数的极值点, 通常需要使用三分法 (ternary search).

本节采用如下严格单峰约定: 对于定义在 $[l,r]$ 上的函数 $f(x)$, 如果存在 $x^* \in [l,r]$, 使得 $f(x)$ 在 $[l,x^*]$ 上严格单调递增, 在 $[x^*,r]$ 上严格单调递减, 就称 $f(x)$ 为单峰函数 (unimodal function). 这里两个区间均包含 x^*, 因此 x^* 是唯一的最大值点, 而 $f(x^*)$ 是最大值.

为什么不通过求导函数的零点来求极值点?

首先, 单峰并不保证导数零点唯一, 导数为零的点也未必是最大值点. 例如

$$f(x)=\begin{cases}(x-1)^3+1, 0\le x<2,\\(3-x)^3+1, 2\le x\le 4.\end{cases}$$

$f'(x)$ 的零点为 $x=1$ 与 $x=3$, 而 $f(x)$ 的最大值在不可导的 $x=2$ 处取得.

其次, 对于一些函数, 求导的过程和结果比较复杂, 甚至无法写成 $y=f(x)$ 的形式.

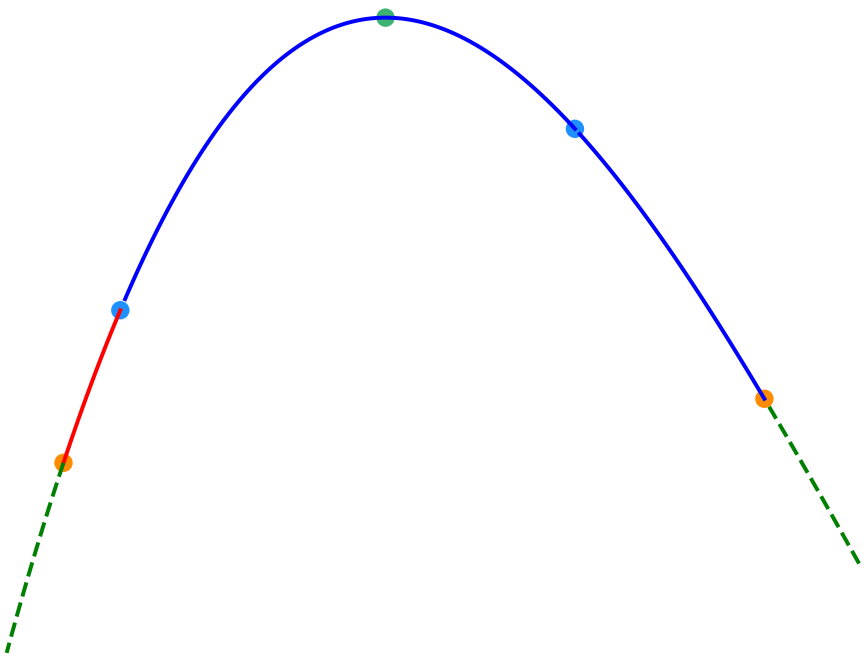
最后, 某些题中需要求极值点的单峰函数并非一个单独的函数, 而是多个函数进行特殊运算得到的函数 (如求多个单调性不完全相同的一次函数的最小值的最大值). 此时函数的导函数可能是分段函数, 且在函数某些点上可能不可导.

注意

三分法既可以求出单峰函数的最大值, 也可以求出「单谷函数」的最小值. 为行文方便, 除特殊说明外, 下文中均以求单峰函数的最大值为例.

过程

三分法与二分法的基本思想类似, 但每次操作需在当前区间 $[l,r]$ (下图中两个橙点之间) 内任取两点 $lmid<rmid$ (下图中的两个蓝点). 如下图所示, 如果 $f(lmid)<f(rmid)$, 则在 $[l,lmid)$ (下图中的红色部分) 中函数必然单调递增, 最大值点 (下图中的绿点) 必然不在这一区间内, 可舍去这一区间; 但是, 无法排除最大值点在 $rmid$ 右侧的可能性, 所以无法舍去更多区间. 反之亦然.



三分法的正确性并不依赖于 $lmid$ 和 $rmid$ 的具体选择，只需保证它们是区间内的两个不同点，通常可以取两个三等分点. 但是，它们的选择会影响三分法的效率. 每次操作都会舍去两侧区间中的一个，因此也可以取靠近中点的两个分点，以增大能舍去的区间. 若取 $mid \pm \delta$，其中 $\delta > 0$ 足够小，此时比较函数值相当于判断近似导数 $\dfrac{f(mid+\delta)-f(mid-\delta)}{2\delta}$ 的符号. 由于算法竞赛中遇到的函数往往都有良好的光滑性，所以我们可以基于此来粗略判定极值点位于 mid 哪一侧，从而达到接近二分法的效率.

实现

伪代码如下：

Algorithm TernarySearch(f, l, r) :
Input. A unimodal function $f(x)$ and its domain $[l, r]$.
Output. The maximizer x^*, up to an error of ε, and its value $f(x^*)$.
Method.

```
while r − l > ε
    mid ← (l + r)/2
    lmid ← mid − ε/3
    rmid ← mid + ε/3
    if f(lmid) < f(rmid)
        l ← lmid
    else
        r ← rmid
x* ← (l + r)/2
return x*, f(x*)
```

分割点的选取

代码中，分割点选取为 $mid \pm \varepsilon/3$ 是为了保证分割点总是在当前的 l 和 r 之间，进而避免陷入死循环.

整数的情形

如果函数 $f(x)$ 的定义域是整数，那么上述三分法和后文的黄金分割法都应该在 $r-l$ 很小时就终止．对于 $r-l$ 很小的情形，需要通过暴力遍历的方法求得最大值点．

优化：黄金分割法

如果单次调用 $f(x)$ 的成本很高，需要进一步减少 $f(x)$ 的调用次数，可以通过黄金分割法（golden-section search）进一步改进三分法的常数．这也是华罗庚提出的优选法的重要内容．

三分法中，每轮迭代需要两次函数调用，且单轮迭代后区间长度至多缩短到原来的 $1/2$．这意味着，要达到精度 ε，至少需要

$$2\log_2\frac{r-l}{\varepsilon}$$

次函数调用．这是三分法能够取得的最好的结果．如果选取其他分点，例如三等分点，那么调用次数会进一步增加，因为单轮迭代后区间缩短得更慢．

黄金分割法的改进思路是，复用前文已经计算过的分点．这样，除了第一轮迭代需要两次函数调用外，其余轮次的迭代只需要一次函数调用．设黄金分割比为

$$\phi=\frac{\sqrt{5}-1}{2}\approx 0.618.$$

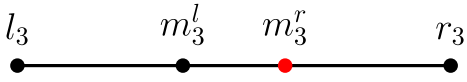
每轮迭代时，选取的分点是左右两个黄金分割点：

$$m^l=\phi l+(1-\phi)r,\ m^r=(1-\phi)l+\phi r.$$

黄金分割点分割线段具有自相似结构．也就是说，m^l 是线段 $[l,r]$ 的左黄金分割点，也是线段 $[l,m^r]$ 的右黄金分割点．这样选取分点的好处是，第 $k>1$ 轮迭代选取的分点中，一定有一个分点是之前已经计算过的，可以直接复用之前的计算结果．







这样选取分点后，要达到精度 ε，只需要

$$1+\log_{\phi^{-1}}\frac{r-l}{\varepsilon}\approx 1+1.44\log_2\frac{r-l}{\varepsilon}$$

次函数调用．渐近意义上，函数的调用次数更少．

伪代码如下：

Algorithm GoldenSectionSearch(f, l, r) :
Input. A unimodal function $f(x)$ and its domain $[l, r]$.
Output. The maximizer x^*, up to an error of ε, and its value $f(x^*)$.
Method.

$lmid \leftarrow \phi l + (1-\phi)r$
$rmid \leftarrow (1-\phi)l + \phi r$
$lval \leftarrow f(lmid)$
$rval \leftarrow f(rmid)$
while $r - l > \varepsilon$
    if $lval > rval$
        $r \leftarrow rmid$
        $rmid \leftarrow lmid$
        $rval \leftarrow lval$
        $lmid \leftarrow \phi l + (1-\phi)r$
        $lval \leftarrow f(lmid)$
    else
        $l \leftarrow lmid$
        $lmid \leftarrow rmid$
        $lval \leftarrow rval$
        $rmid \leftarrow (1-\phi)l + \phi r$
        $rval \leftarrow f(rmid)$
$x^* \leftarrow (l+r)/2$
return x^*, $f(x^*)$

例题

洛谷 P3382 - 三分[5]

给定一个 N 次函数和范围 $[l, r]$，求出使函数在 $[l, x]$ 上单调递增且在 $[x, r]$ 上单调递减的唯一的 x 的值.

解题思路

本题要求求 N 次函数在 $[l, r]$ 取最大值时自变量的值，显然可以使用三分法. 以下实现使用两个三等分点，并将区间端点更新到实际比较的分点；当区间长度足够小时，输出区间中点.

参考代码

- C++

```cpp
#include <cmath>
#include <iomanip>
#include <iostream>
using namespace std;

constexpr double eps = 1e-7;
int N;
double l, r, A[20], lmid, rmid;

double f(double x) {
  double res = (double)0;
  for (int i = N; i >= 0; i--) res += A[i] * pow(x, i);
  return res;
}

```

```cpp
int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  cin >> N >> l >> r;
  for (int i = N; i >= 0; i--) cin >> A[i];
  while (r - l > eps) {
    lmid = l + (r - l) / 3;
    rmid = r - (r - l) / 3;
    if (f(lmid) > f(rmid))
      r = rmid;
    else
      l = lmid;
  }
  cout << fixed << setprecision(6) << (l + r) / 2;
  return 0;
}

```

- Python

```python
eps = 1e-7
n, l, r = map(float, input().split())
a = tuple(map(float, input().split()))[::-1]


def f(x):
    return sum(x**i * j for i, j in enumerate(a))


while r - l > eps:
    lmid = l + (r - l) / 3
    rmid = r - (r - l) / 3
    if f(lmid) > f(rmid):
        r = rmid
    else:
        l = lmid
print(f"{(l + r) / 2:.6f}")

```

习题

- UVa 1476 - Error Curves[6]
- UVa 10385 - Duathlon[7]
- UOJ 162 -【清华集训 2015】灯泡测试[8]
- 洛谷 P7579 -「RdOI R2」称重（weigh）[9]

分数规划

参见：分数规划→13.5@p3245

分数规划通常描述为下列问题：每个物品有两个属性 c_i, d_i，要求通过某种方式选出若干个，使得 $\frac{\sum c_i}{\sum d_i}$ 最大或最小.

经典的例子有最优比率环、最优比率生成树等等.

分数规划可以用二分法来解决.

参考资料与注释

- Ternary search - Wikipedia[10]
- Golden-section search - Wikipedia[11]
- Ternary search - CP Algorithms[12]

外部参考资料

[1] `std::lower_bound` 

[2] `std::upper_bound` 

[3] `std::partition_point` 

[4] Luogu P1873 砍树 

[5] 洛谷 P3382 - 三分 

[6] UVa 1476 - Error Curves 

[7] UVa 10385 - Duathlon 

[8] UOJ 162 -【清华集训 2015】灯泡测试 

[9] 洛谷 P7579 -「RdOI R2」称重（weigh） 

[10] Ternary search - Wikipedia 

[11] Golden-section search - Wikipedia 

[12] Ternary search - CP Algorithms 

5.10 倍增

> **Authors:** Ir1d, ShadowsEpic, Fomalhauthmj, siger-young, MingqiHuang, Xeonacid, hsfzLZH1, orzAtalod, NachtgeistW

本页面将简要介绍倍增法.

定义

倍增法（binary lifting），顾名思义就是「成倍增长」. 我们在进行递推时，如果状态空间很大，通常的线性递推无法满足时间与空间复杂度的要求，那么我们可以通过成倍增长的方式，只递推状态空间中在 k 的整数次幂位置上的值作为代表. 当需要其他位置上的值时，我们通过「任意整数可以表示成若干个 k 的次幂项的和」这一性质，使用之前求出的代表值拼成所需的值. 所以使用倍增算法也要求我们递推的问题的状态空间关于 k 的次幂具有可划分性. 通常情况下 k 取 2. [1]

这个方法在很多算法中均有应用，其中最常用的是 RMQ 问题和求 LCA（最近公共祖先）→11.6.5@p2646.

应用

RMQ 问题

参见：RMQ 专题→14.1@p3342.

RMQ 是 Range Maximum/Minimum Query 的缩写，表示区间最大（最小）值查询. 使用倍增思想解决 RMQ 问题的方法是 ST 表→10.11@p2072.

树上倍增求 LCA

参见：最近公共祖先→11.6.5@p2646.

[1] 引用自李煜东《算法竞赛进阶指南》0x06. 倍增一节.

例题

题一

例题

如何用尽可能少的砝码称量出 $[0,31]$ 之间的所有重量？（只能在天平的一端放砝码）

解题思路

答案是使用 1 2 4 8 16 这五个砝码，可以称量出 $[0,31]$ 之间的所有重量．同样，如果要称量 $[0,127]$ 之间的所有重量，可以使用 1 2 4 8 16 32 64 这七个砝码．每次我们都选择 2 的非负整数次幂作砝码的重量，就可以用非常少的砝码称量出任意我们所需要的重量．

例如，如果我们要量出 $[0,1023]$ 之间的所有重量，只需要 10 个砝码，需要量出 $[0,1048575]$ 之间的所有重量，只需要 20 个．如果我们的目标重量翻倍，砝码个数只需要增加 1．这叫「对数级」的增长速度，因为砝码的所需个数与目标重量的范围的对数成正比．

题二

例题

给出一个长度为 n 的环和一个常数 k，每次会从第 i 个点跳到第 $(i+k) \bmod n+1$ 个点，总共跳了 m 次．每个点都有一个权值，记为 a_i，求 m 次跳跃的起点的权值之和对 10^9+7 取模的结果．

数据范围：$1 \le n \le 10^6$，$1 \le m \le 10^{18}$，$1 \le k \le n$，$0 \le a_i \le 10^9$．

解题思路

这里显然不能暴力模拟跳 m 次．因为 m 最大可到 10^{18} 级别，如果暴力模拟的话，时间承受不住．

所以就需要进行一些预处理，提前整合一些信息，以便于在查询的时候更快得出结果．如果记录下来每一个可能的跳跃次数的结果的话，不论是时间还是空间都难以承受．

那么应该如何预处理呢？看看第一道例题．有思路了吗？

回到本题．我们要预处理一些信息，然后用预处理的信息尽量快地整合出答案．同时预处理的信息也不能太多．所以可以预处理出以 2 的非负整数次幂为单位的信息，这样的话在预处理的时候只需要处理少量信息，在整合的时候也不需要大费周章．

在这题上，就是我们预处理出从每个点开始跳 1、2、4、8 等等步之后的结果（所处点和点权和），然后如果要跳 13 步，只需要跳 1+4+8 步就好了．也就是说先在起始点跳 1 步，然后再在跳了之后的终点跳 4 步，再接着跳 8 步，同时统计一下预先处理好的点权和，就可以知道跳 13 步的点权和了．

对于每一个点开始的 2^i 步，记录一个 `go[i][x]` 表示第 x 个点跳 2^i 步之后的终点，而 `sum[i][x]` 表示第 x 个点跳 2^i 步之后能获得的点权和．预处理的时候，开两重循环，对于跳 2^i 步的信息，我们可以看作是先跳了 2^{i-1} 步，再跳 2^{i-1} 步，因为显然有 $2^{i-1}+2^{i-1}=2^i$．即我们有 `sum[i][x] = sum[i-1][x]+sum[i-1][go[i-1][x]]`，且 `go[i][x] = go[i-1][go[i-1][x]]`．

当然还有一些实现细节需要注意．为了保证统计的时候不重不漏，我们一般预处理出「左闭右开」的点权和．亦即，对于跳 1 步的情况，我们只记录该点的点权和；对于跳 2 步的情况，我们只记录该点及其下一个点的点权和．相当于总是不将终点的点权和计入 `sum`．这样在预处理的时候，只需要将两部分的点权和直接相加就可以了，不需要担心第一段的终点和第二段的起点会被重复计算．

这题的 $m \le 10^{18}$，虽然看似恐怖，但是实际上只需要预处理第 0 至 59 层，就可以轻松解决，比起暴力枚举快了很多. 用行话讲，这个做法的 时间复杂度→5.2.1@p544 是预处理 $\Theta(n \log m)$，查询每次 $\Theta(\log m)$.

参考代码

```cpp
#include <cstdio>
using namespace std;

constexpr int mod = 1000000007;

int modadd(int a, int b) {
  if (a + b >= mod) return a + b - mod;  // 减法代替取模，加快运算
  return a + b;
}

int vi[1000005];

int go[60][1000005];  // 将数组稍微开大以避免越界，小的一维尽量定义在前面
int sum[60][1000005];

int main() {
  int n, k;
  scanf("%d%d", &n, &k);
  for (int i = 1; i <= n; ++i) {
    scanf("%d", vi + i);
  }

  for (int i = 1; i <= n; ++i) {
    go[0][i] = (i + k) % n + 1;
    sum[0][i] = vi[i];
  }

  // m <= 10^18，只需 0..59 层
  for (int i = 1; i < 60; ++i) {
    for (int j = 1; j <= n; ++j) {
      go[i][j] = go[i - 1][go[i - 1][j]];
      sum[i][j] = modadd(sum[i - 1][j], sum[i - 1][go[i - 1][j]]);
    }
  }

  long long m;
  scanf("%lld", &m);

  int ans = 0;
  int curx = 1;
  for (int i = 0; m; ++i) {
    if (m & (1ll << i)) {  // 参见位运算的相关内容，意为 m 的第 i 位是否为 1
      ans = modadd(ans, sum[i][curx]);
      curx = go[i][curx];
      m ^= 1ll << i;  // 将第 i 位置零
    }
```

```
47   }
48
49   printf("%d\n", ans);
50 }
51
```

参考资料与注释

5.11 构造

Authors: leoleoasd, yzxoi, Estrella-Explore

本页面将简要介绍构造题这类题型.

引入

构造题是比赛中常见的一类题型.

从形式上来看，问题的答案往往具有某种规律性，使得在问题规模迅速增大的时候，仍然有机会比较容易地得到答案.

这要求解题时要思考问题规模增长对答案的影响，这种影响是否可以推广. 例如，在设计动态规划方法的时候，要考虑从一个状态到后继状态的转移会造成什么影响.

特点

构造题一个很显著的特点就是高自由度，也就是说一道题的构造方式可能有很多种，但是会有一种较为简单的构造方式满足题意. 看起来是放宽了要求，让题目变得简单了，但很多时候，正是这种高自由度导致题目没有明确思路而无从下手.

构造题另一个特点就是形式灵活，变化多样. 并不存在一个通用解法或套路可以解决所有构造题，甚至很难找出解题思路的共性.

例题

下面将列举一些例题帮助读者体会构造题的一些思想内涵，给予思路上的启发. 建议大家深入思考后再查看题解，也欢迎大家参与分享有趣的构造题.

例题 1

Codeforces Round #384 (Div. 2) C.Vladik and fractions[1]

构造三个两两不同的正整数 x, y, z，使得对于给定的 n，满足 $\frac{1}{x}+\frac{1}{y}+\frac{1}{z}=\frac{2}{n}$.

解题思路

从样例二可以看出本题的构造方法.

可将 $\frac{2}{n}$ 分解为 $\frac{1}{n}+\frac{1}{n}$，再将后一项写为 $\frac{1}{n+1}+\frac{1}{n(n+1)}$. 所以 $n, n+1, n(n+1)$ 即为一组合法解. 特别地，$n=1$ 时无解，因为三个互异正整数的倒数和至多为 $1+\frac{1}{2}+\frac{1}{3}<2$.

例题 2

Luogu P3599 Koishi Loves Construction[2]

Task1：试判断能否构造并构造一个长度为 n 的 $1...n$ 的排列，满足其 n 个前缀和在模 n 的意义下互不相同.

Task2：试判断能否构造并构造一个长度为 n 的 $1...n$ 的排列，满足其 n 个前缀积在模 n 的意义下互不相同.

解题思路

两项任务在 $n=1$ 时均直接取排列 $[1]$. 以下设 $n>1$.

对于 Task1：

当 n 为奇数时，无法构造出合法解；当 n 为偶数时，可以构造一个形如 $n,1,n-2,3,\cdots$ 这样的数列.

首先，我们可以发现 n 必定出现在数列的第一位，否则 n 出现前后的两个前缀和必然会陷入模意义下相等的尴尬境地；

然后，我们考虑构造出整个序列的方式：

考虑通过构造前缀和序列的方式来获得原数列，可以发现相邻前缀和之差（再加第一项）在模意义下应互不相同，因为前缀和序列的差分序列对应着原来的排列.

因此我们尝试以前缀和数列在模意义下为

$$0,1,-1,2,-2,\cdots$$

这样的形式来构造这个序列，不难发现它完美地满足所有限制条件.

对于 Task2：

当 n 为除 4 以外的合数时，无法构造出合法解；当 $n=4$ 时单独取 $[1,3,2,4]$. 当 n 为质数时，可以构造一个形如 $1,\frac{2}{1},\frac{3}{2},\cdots,\frac{n-1}{n-2},n$ 这样的数列，其中除法表示乘模 n 的 逆元→9.11.10@p1229 后取 $[1,n]$ 中的代表元.

先考虑什么时候有解：

当 n 为除 4 以外的合数时无解. 因为对于一个合数来说，存在两个比它小的数 p,q 使得 $p\times q\equiv 0 \pmod n$，如 $(3\times 6)\bmod 9=0$. 那么，当 p,q 均出现过后，数列的前缀积将一直为 0，故除 4 以外的合数无解. 特殊地，我们可以发现 $4=2\times 2$，无满足条件的 p,q，因此存在合法解.

我们考虑如何构造这个数列：

和 task1 同样的思路，我们发现 1 必定出现在数列的第一位，否则 1 出现前后的两个前缀积必然相等；而 n 必定出现在数列的最后一位，因为 n 出现位置后的所有前缀积在模意义下都为 0. 分析题目给出的几组样例以后发现，所有样例中均有一组合法解满足前缀积在模意义下为 $1,2,3,\cdots,n$，因此我们可以构造出上文所述的数列来满足这个条件. 那么我们只需证明这 n 个数互不相同即可.

我们发现这些数均为 $1\cdots n-2$ 的逆元 $+1$，因此各不相同，此题得解.

例题 3

AtCoder Grand Contest 032 B[3]

给定一个整数 N，试构造一个节点数为 N 的无向图. 令节点编号为 $1...N$，要求其满足以下条件：

- 这是一个简单连通图.
- 存在一个整数 S 使得对于任意节点，与其相邻节点的下标和为 S.

保证输入数据有解.

解题思路

通过分析 $n=3,4,5$ 的情况，我们可以找到一个构造思路.

构造一个完全 k 分图，保证这 k 部分和相等. 则每个点的 S 均相等，为

$$S=\frac{(k-1)\sum_{i=1}^{n} i}{k}.$$

如果 n 为偶数，那么我们可以前后两两配对，即 $\{1,n\},\{2,n-1\}\cdots$.

如果 n 为奇数，那么我们可以把 n 单拿出来作为一组，剩余的 $n-1$ 个两两配对，即 $\{n\},\{1,n-1\},\{2,n-2\}\cdots$.

这样构造出的图在 $n\geq 3$ 时连通性易证，在此不加赘述.

此题得解.

例题 4

BZOJ 4971「Lydsy1708 月赛」记忆中的背包[4]

经过一天辛苦的工作，小 Q 进入了梦乡. 他脑海中浮现出了刚进大学时学 01 背包的情景，那时还是大一萌新的小 Q 解决了一道简单的 01 背包问题. 这个问题是这样的:

给定 n 个物品，每个物品的体积分别为 $v_1,v_2,...,v_n$，请计算从中选择一些物品（也可以不选），使得总体积恰好为 w 的方案数. 因为答案可能非常大，你只需要输出答案对 P 取模的结果.

因为长期熬夜刷题，他只看到样例输入中的 w 和 P，以及样例输出是 k，看不清到底有几个物品，也看不清每个物品的体积是多少. 直到梦醒，小 Q 也没有看清 n 和 v，请写一个程序，帮助小 Q 一起回忆曾经的样例输入.

多组数据，数据满足 $50\leq w\leq 20000$、$1\leq P\leq 2^{30}$、$0\leq k\leq \min(20000,P-1)$；要求输出 $1\leq n\leq 40$ 个物品，且 $1\leq v_i\leq 20000$.

解题思路

这道题是自由度最高的构造题之一了. 这就导致了没有头绪，难以入手的情况.

因为 $k<P$，所以我们可以考虑直接构造方案数恰为 k 的物品集合.

一种构造方式是选取 i 个体积为 1 的小物品和若干个体积为 $w-t$ 的大物品，其中 $1\leq i\leq 20$、$0\leq t\leq i$. 因为 $i<w$ 且 $w-t\geq w-20>w/2$，恰好装满背包的方案必须选择一个大物品.

令每个体积为 $w-t$ 的大物品对方案数的贡献为 C，则 $C=\binom{i}{t}$，不同大物品的贡献相加. 即使体积相同，不同物品仍分别计数.

令 $f_{i,j}$ 表示有 i 个 1，方案数为 j 的最小大物品数.

固定 i，令 $f_{i,0}=0$，其余状态初值为 $+\infty$. 按 j 递增计算完全背包转移:

$$f_{i,j}=1+\min_{0\leq t\leq i;\ C\leq j} f_{i,j-C},\qquad 1\leq j\leq 20000.$$

记录取得最小值的 t，即可回溯出每个大物品的体积 $w-t$. 对全部 $0 \le k \le 20000$ 计算可得 $\min_{1 \le i \le 20}(i + f_{i,k}) \le 29 < 40$，所以这段预处理范围足够.

参考实现

```cpp
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;

constexpr int K = 20000;
constexpr int INF = K + 100;
int c[21][21], take[21][K + 1];
int f[K + 1], best[K + 1], small[K + 1];

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  fill(best, best + K + 1, INF);
  c[0][0] = 1;
  for (int i = 1; i <= 20; ++i) {
    c[i][0] = c[i][i] = 1;
    for (int t = 1; t < i; ++t) c[i][t] = c[i - 1][t - 1] + c[i - 1][t];
    fill(f, f + K + 1, INF);
    f[0] = 0;
    // 每个大物品贡献 c[i][t] 种方案，按完全背包求最少物品数。
    for (int t = 0; t <= i; ++t) {
      for (int j = c[i][t]; j <= K; ++j) {
        if (f[j - c[i][t]] + 1 < f[j]) {
          f[j] = f[j - c[i][t]] + 1;
          take[i][j] = t;
        }
      }
    }
    for (int k = 0; k <= K; ++k) {
      if (i + f[k] < best[k]) {
        best[k] = i + f[k];
        small[k] = i;
      }
    }
  }

  int T;
  cin >> T;
  while (T--) {
    int w, P, k;
    cin >> w >> P >> k;
    int i = small[k];
    vector<int> volumes(i, 1);
    // 构造恰好 k 种方案；由于 k < P，无需另外处理取模。
    while (k > 0) {
      int t = take[i][k];
      volumes.push_back(w - t);
```

```
48        k -= c[i][t];
49      }
50      cout << volumes.size() << '\n';
51      for (size_t j = 0; j < volumes.size(); ++j)
52        cout << volumes[j] << (j + 1 == volumes.size() ? '\n' : ' ');
53    }
54    return 0;
55  }
56
```

外部参考资料

[1] Codeforces Round #384 (Div. 2) C.Vladik and fractions 

[2] Luogu P3599 Koishi Loves Construction 

[3] AtCoder Grand Contest 032 B 

[4] BZOJ 4971「Lydsy1708 月赛」记忆中的背包 

第六章

搜索

6.1 搜索部分简介

搜索，也就是对状态空间进行枚举，通过穷尽所有的可能来找到最优解，或者统计合法解的个数.

搜索有很多优化方式，如减小状态空间，更改搜索顺序，剪枝等.

搜索是一些高级算法的基础. 在 OI 中，纯粹的搜索往往也是得到部分分的手段，但可以通过纯粹的搜索拿到满分的题目非常少.

习题

- 「kuangbin 带你飞」专题一 简单搜索[1]
- 「kuangbin 带你飞」专题二 搜索进阶[2]
- 洛谷搜索题单[3]
- openjudge 搜索题单[4]

外部参考资料

[1]「kuangbin 带你飞」专题一 简单搜索

[2]「kuangbin 带你飞」专题二 搜索进阶

[3] 洛谷搜索题单

[4] openjudge 搜索题单

6.2 DFS（搜索）

引入

DFS 为图论中的概念，详见 DFS（图论）→11.4@p2612 页面. 在 **搜索算法** 中，该词常常指利用递归函数方便地实现暴力枚举的算法，与图论中的 DFS 算法有一定相似之处，但并不完全相同.

解释

考虑这个例子：

例题

把正整数 n 分解为 3 个正整数，如 $6=1+2+3$，排在后面的数必须大于等于前面的数，输出所有方案.

对于这个问题，如果不知道搜索，应该怎么办呢？当然是三重循环，参考代码如下：

实现

- C++

```cpp
for (int i = 1; i <= n; ++i)
  for (int j = i; j <= n; ++j)
    for (int k = j; k <= n; ++k)
      if (i + j + k == n) printf("%d = %d + %d + %d\n", n, i, j, k);
```

- Python

```python
for i in range(1, n + 1):
    for j in range(i, n + 1):
        for k in range(j, n + 1):
```

```
            if i + j + k == n:
                print("%d = %d + %d + %d" % (n, i, j, k))
```

- Java

```
for (int i = 1; i < n + 1; i++) {
    for (int j = i; j < n + 1; j++) {
        for (int k = j; k < n + 1; k++) {
            if (i + j + k == n) System.out.printf("%d = %d + %d + %d%n", n, i, j, k);
        }
    }
}
```

那如果是分解成四个整数呢？再加一重循环？那分解成小于等于 m 个整数呢？

这时候就需要用到递归搜索了．该类搜索算法的特点在于，将要搜索的目标分成若干「层」，每层基于前几层的状态进行决策，直到达到目标状态．

考虑上述问题，即将正整数 n 分解成不超过 m 个正整数之和，且排在后面的数必须大于等于前面的数，并输出所有方案．

设一组方案将正整数 n 分解成 k 个正整数 $a_1, a_2, ..., a_k$ 的和．将问题分层，第 i 层决定 a_i．则为了进行第 i 层决策，我们需要记录三个状态变量：$n - \sum_{j=1}^{i} a_j$，表示后面所有正整数的和；a_{i-1}，表示前一层的正整数，以确保正整数递增；以及 i，确保我们最多输出 m 个正整数．为了记录方案，我们用 `arr` 数组，第 i 项表示 a_i．注意到 `arr` 实际上是一个长度为 i 的栈．

代码如下：

实现

- C++

```
int m, arr[103];  // arr 用于记录方案

void dfs(int n, int i, int a) {
  if (n == 0) {
    for (int j = 1; j <= i - 1; ++j) printf("%d ", arr[j]);
    printf("\n");
  }
  if (i <= m) {
    for (int j = a; j <= n; ++j) {
      arr[i] = j;
      dfs(n - j, i + 1, j);  // 请仔细思考该行含义．
    }
  }
}

// 主函数
scanf("%d%d", &n, &m);
dfs(n, 1, 1);
```

- Python

```
arr = [0] * 103  # arr 用于记录方案
```

```
2
3
4 def dfs(n, i, a):
5     if n == 0:
6         print(arr[1:i])
7     if i <= m:
8         for j in range(a, n + 1):
9             arr[i] = j
10            dfs(n - j, i + 1, j)  # 请仔细思考该行含义.
11
12
13 # 主函数
14 n, m = map(int, input().split())
15 dfs(n, 1, 1)
```

- Java

```
1 static int m;
2
3 // arr 用于记录方案
4 static int[] arr = new int[103];
5
6 public static void dfs(int n, int i, int a) {
7     if (n == 0) {
8         for (int j = 1; j <= i - 1; j++) System.out.printf("%d
  ", arr[j]);
9         System.out.println();
10    }
11    if (i <= m) {
12        for (int j = a; j <= n; ++j) {
13            arr[i] = j;
14            dfs(n - j, i + 1, j); // 请仔细思考该行含义.
15        }
16    }
17 }
18
19 // 主函数
20 final int N = new Scanner(System.in).nextInt();
21 m = new Scanner(System.in).nextInt();
22 dfs(N, 1, 1);
```

例题

Luogu P1706 全排列问题[1]

```
1 #include <iomanip>
2 #include <iostream>
3 using namespace std;
4 int n;
5 bool vis[50];  // 访问标记数组
6 int a[50];     // 排列数组，按顺序储存当前搜索结果
```

```
7
8  void dfs(int step) {
9    if (step == n + 1) {  // 边界
10     for (int i = 1; i <= n; i++) {
11       cout << setw(5) << a[i];  // 保留5个场宽
12     }
13     cout << endl;
14     return;
15   }
16   for (int i = 1; i <= n; i++) {
17     if (!vis[i]) {  // 判断数字i是否在正在进行的全排列中
18       vis[i] = true;
19       a[step] = i;
20       dfs(step + 1);
21       vis[i] = false;  // 这一步不使用该数 置0后允许下一步使用
22     }
23   }
24   return;
25 }
26
27 int main() {
28   cin >> n;
29   dfs(1);
30   return 0;
31 }
32
```

外部参考资料

[1] Luogu P1706 全排列问题 

6.3 BFS（搜索）

引入

BFS（广度优先搜索）为图论中的基础算法，详见 BFS（图论）→11.5@p2616 页面. 在 **搜索算法** 中，该算法通常指利用队列结构逐层扩展状态的搜索方式，与图论中的 BFS 算法思想一致，特别适合求解 **最短路径** 或 **最少步骤** 类问题.

解释

BFS 的核心思想是 **按层扩展**，从起点开始逐层扫描可到达的位置. 首次遇到终点时的路径长度即为最短路径. 这种方式保证了搜索的层次性与最优性.

在实际执行中，BFS 会从起点出发，先访问起点的所有直接可到达结点，这些可到达结点构成了搜索的第一层；接着，再以这些可到达结点为新的起点，依次访问它们的邻居，形成第二层；以此类推，不断向外扩展，直至找到目标结点或遍历完所有可达结点. 这个过程中，算法会借助队列和访问数组，将每一层新发现的结点（访问数组中还没有记录过的）依次入队，确保同一层的结点按照访问顺序依次被处理，从而严格遵循「按层扩展」的逻辑.

BFS 非常擅于快速求解 **最短路径** 或 **最少步骤**．当算法在某一层首次遇到目标时，此时经过的路径长度（步骤数）必然是最短的．这是因为 BFS 算法的「按层扩展」机制保证了每个结点都是通过最少的步数被访问到：就像从起点出发，沿着最直接的路径不断搜索，直到抵达终点，不会出现绕路或走多余步骤的情况．在这类问题中，BFS 通常也比 DFS 的效率更高．

但是，相较于 DFS，BFS 也有其缺点．通常情况下，BFS 需要更大的内存，缺乏天然的回溯过程，且深度剪枝相对没有 DFS 灵活．

例题

例题 Luogu B3625 迷宫寻路[1]

在一个 $n \times m$ 的迷宫矩阵中，`.` 表示可通行区域，`#` 表示障碍物．从起点 $(1,1)$ 出发，每次可向上下左右四个方向移动，问是否能到达终点 (n,m)．

解答

实现时需要维护一个队列来存储待处理的坐标，并配合访问标记数组避免重复计算．一个结点扩展可到达结点的时候，需要向上下左右拓展，这四个方向分别为 $(x,y+1)$，$(x,y-1)$，$(x+1,y)$，$(x-1,y)$，在代码中使用了方向数组．注意判断不能拓展到有障碍物的位置．

参考实现

```cpp
#include <iostream>
#include <queue>
using namespace std;

char a[110][110];    // 存储迷宫地图
bool vis[110][110];  // 记录访问状态
int n, m;            // 迷宫尺寸

struct node {
  int x, y;
};  // 定义坐标结构体

int dx[] = {0, 0, 1, -1}, dy[] = {1, -1, 0, 0};  // 方向数组（右左上下）

// 检查坐标是否合法
bool chk(int x, int y) {
  return (x >= 1 && x <= n && y >= 1 && y <= m  // 边界检查
          && !vis[x][y]                         // 未访问过
          && a[x][y] != '#');                   // 不是障碍物
}

bool bfs() {
  queue<node> q;
  q.push({1, 1});  // 起点入队
  vis[1][1] = 1;   // 标记起点已访问
  while (!q.empty()) {
    node p = q.front();  // 取出队首坐标
    q.pop();
    int px = p.x, py = p.y;
```

```
    if (px == n && py == m) return true;  // 到达终点立即返回
    // 向四个方向扩展
    for (int i = 0; i < 4; ++i) {
      int nx = px + dx[i], ny = py + dy[i];
      if (chk(nx, ny)) {   // 合法性检查
        q.push({nx, ny});  // 新坐标入队
        vis[nx][ny] = 1;   // 标记已访问
      }
    }
  }
  return false;
}

int main() {
  cin >> n >> m;
  for (int i = 1; i <= n; ++i)
    for (int j = 1; j <= m; ++j) cin >> a[i][j];
  cout << (bfs() ? "Yes" : "No") << endl;
  return 0;
}

```

例题 Luogu P1135 奇怪的电梯[2]

有 n 层楼和一架电梯. 电梯位于第 i 层楼时，向上或向下移动的层数等于一个固定的数字 k_i. 如果到达的层数不合法，即不在 1 和 n 之间，相应的操作就无法进行. 问：从第 a 楼到第 b 楼至少操作几次电梯？如果无法到达，输出 -1.

解答

本题需要计算最短路径，这正是 BFS 擅长解决的问题. 实现时，需要在队列中同时维护需要处理的楼层位置和从起点 a 出发到达当前楼层的最短距离，并配合访问标记数组避免重复加入同一个元素. 一个结点 i 扩展可到达结点的时候，需要向 $i+k_i$ 和 $i-k_i$ 扩展，注意不能到达非法楼层. 当扩展到尚未到达的合法楼层时，需要将它加入队列，并记录到达该楼层的最短距离为到达当前所在楼层的最短距离加一. 当首次到达结点 b 时，记录的最短距离就是最终答案.

代码中，直接记录距离数组，并利用距离是否为默认值（即 -1）来判断结点是否尚未访问.

参考实现

```
#include <iostream>
#include <queue>
using namespace std;
const int N = 210;
int n, a, b, k[N], d[N];

int main() {
  cin >> n >> a >> b;
  for (int i = 1; i <= n; ++i) cin >> k[i];
  for (int i = 1; i <= n; ++i) d[i] = -1;
```

```
  queue<int> q;
  d[a] = 0;
  q.push(a);
  while (!q.empty()) {
    int x = q.front();
    q.pop();
    if (x == b) {
      break;
    }
    int y = x + k[x];
    if (y <= n && d[y] == -1) {
      d[y] = d[x] + 1;
      q.push(y);
    }
    y = x - k[x];
    if (y >= 1 && d[y] == -1) {
      d[y] = d[x] + 1;
      q.push(y);
    }
  }
  cout << d[b] << endl;
  return 0;
}

```

习题

- Luogu P1443 马的遍历[3]
- Luogu P3956 [NOIP 2017 普及组] 棋盘[4]
- Luogu P1126 机器人搬重物[5]

外部参考资料

[1] Luogu B3625 迷宫寻路 

[2] Luogu P1135 奇怪的电梯 

[3] Luogu P1443 马的遍历 

[4] Luogu P3956 [NOIP 2017 普及组] 棋盘 

[5] Luogu P1126 机器人搬重物 

6.4 双向搜索

Authors: FFjet, ChungZH, frank-xjh, hsfzLZH1, Xarfa, AndrewWayne, hcx1204

本页面将简要介绍两种双向搜索算法：「双向同时搜索」和「Meet in the middle」.

双向同时搜索

定义

双向同时搜索的基本思路是从状态图上的起点和终点同时开始进行 广搜→6.3@p657 或 深搜→6.2@p654.

如果发现搜索的两端相遇了，那么可以认为是获得了可行解.

过程

双向广搜的步骤：

```
将开始结点和目标结点加入队列 q
标记开始结点为 1
标记目标结点为 2
while (队列 q 不为空)
{
  从 q.front() 扩展出新的 s 个结点

  如果 新扩展出的结点已经被其他数字标记过
    那么 表示搜索的两端碰撞
    那么 循环结束

  如果 新的 s 个结点是从开始结点扩展来的
    那么 将这个 s 个结点标记为 1 并且入队 q

  如果 新的 s 个结点是从目标结点扩展来的
    那么 将这个 s 个结点标记为 2 并且入队 q
}
```

例题

例题 八数码难题[1]

在 3×3 的棋盘上，摆有八个棋子，每个棋子上标有 1 至 8 的某一数字. 棋盘中留有一个空格，空格用 0 来表示. 空格周围的棋子可以移到空格中. 要求解的问题是：给出一种初始布局（初始状态）和目标布局（为了使题目简单，设目标状态为 123804765），找到一种最少步骤的移动方法，实现从初始布局到目标布局的转变.

解题思路

很好想出暴力 bfs. 本题使用暴力 bfs 也不会超时. 但是这里把它作为双向同时搜索的例题. 我们可以使用两个 bfs，一个从起点状态开始正着搜，一个从终点状态开始反着搜，交替使用两个 bfs，搜索树的大小会大大减小. 当其中一个 bfs 搜出另一个 bfs 已经搜出的状态，即可得到答案.

参考代码

```cpp
#include <iostream>
#include <map>
#include <queue>
#include <string>
using namespace std;

struct State {
  int A[3][3];
  State() = default;

  State(string s) {
    for (int i = 0; i < 3; i++) {
      for (int j = 0; j < 3; j++) {
```

```
14          A[i][j] = s[i * 3 + j] - '0';
15        }
16      }
17    }
18
19    friend bool operator<(const State &a, const State &b) {
20      for (int i = 0; i < 3; i++) {
21        for (int j = 0; j < 3; j++) {
22          if (a.A[i][j] != b.A[i][j]) {
23            return a.A[i][j] < b.A[i][j];
24          }
25        }
26      }
27      return false;
28    }
29  };
30
31  int dir[4][2] = {{1, 0}, {-1, 0}, {0, 1}, {0, -1}};
32
33  void bfs(queue<State> &q, map<State, int> &m1, map<State, int> &m2) {
34    auto u = q.front();
35    q.pop();
36    int xx, yy;
37    for (int i = 0; i < 3; i++) {
38      for (int j = 0; j < 3; j++) {
39        if (u.A[i][j] == 0) {
40          xx = i;
41          yy = j;
42        }
43      }
44    }
45    for (int i = 0; i < 4; i++) {
46      int tx = dir[i][0] + xx, ty = dir[i][1] + yy;
47      if (tx >= 0 && tx < 3 && ty >= 0 && ty < 3) {
48        auto v = u;
49        swap(v.A[xx][yy], v.A[tx][ty]);
50        if (m2.count(v)) {
51          cout << m1[u] + m2[v] << endl;
52          exit(0);
53        }
54        if (!m1.count(v)) {
55          m1[v] = m1[u] + 1;
56          q.push(v);
57        }
58      }
59    }
60  }
61
62  int main() {
63    string I, O;
64    cin >> I;
65    O = "123804765";
```

```
  State in = I, ou = 0;
  queue<State> q1, q2;
  map<State, int> mp1, mp2;
  q1.push(in);
  mp1[in] = 0;
  q2.push(ou);
  mp2[ou] = 1;
  if (I == 0) {
    cout << 0;
    return 0;
  }
  while (1) {
    bfs(q1, mp1, mp2);
    bfs(q2, mp2, mp1);
  }
  return 0;
}

```

Meet in the middle

Warning

本节要介绍的不是 **二分搜索**→5.9@p629（二分搜索的另外一个译名为「折半搜索」）.

引入

Meet in the middle 算法没有正式译名，常见的翻译为「折半搜索」、「双向搜索」或「中途相遇」.

它适用于输入数据较小，但还没小到能直接使用暴力搜索的情况.

过程

Meet in the middle 算法的主要思想是将整个搜索过程分成两半，分别搜索，最后将两半的结果合并.

性质

暴力搜索的复杂度往往是指数级的，而改用 meet in the middle 算法后复杂度的指数可以减半，即让复杂度从 $O(a^b)$ 降到 $O(a^{b/2})$.

例题

例题 「USACO09NOV」灯 Lights[2]

有 n 盏灯，每盏灯与若干盏灯相连，每盏灯上都有一个开关，如果按下一盏灯上的开关，这盏灯以及与之相连的所有灯的开关状态都会改变. 一开始所有灯都是关着的，你需要将所有灯打开，求最小的按开关次数.

$1 \leq n \leq 35$.

解题思路

如果这道题暴力 DFS 找开关灯的状态，时间复杂度就是 $O(2^n)$，显然超时．不过，如果我们用 meet in middle 的话，时间复杂度可以优化至 $O(n2^{n/2})$．meet in middle 就是让我们先找一半的状态，也就是找出只使用编号为 1 到 mid 的开关能够到达的状态，再找出只使用另一半开关能到达的状态．如果前半段和后半段开启的灯互补，将这两段合并起来就得到了一种将所有灯打开的方案．具体实现时，可以把前半段的状态以及达到每种状态的最少按开关次数存储在 map 里面，搜索后半段时，每搜出一种方案，就把它与互补的第一段方案合并来更新答案．

参考代码

```cpp
#include <algorithm>
#include <iostream>
#include <map>
using namespace std;

int n, m, ans = 0x7fffffff;
map<long long, int> f;
long long a[40];

int main() {
  cin >> n >> m;
  a[0] = 1;
  for (int i = 1; i < n; ++i) a[i] = a[i - 1] * 2;  // 进行预处理

  for (int i = 1; i <= m; ++i) {  // 对输入的边的情况进行处理
    int u, v;
    cin >> u >> v;
    --u;
    --v;
    a[u] |= ((long long)1 << v);
    a[v] |= ((long long)1 << u);
  }

  for (int i = 0; i < (1 << (n / 2)); ++i) {  // 对前一半进行搜索
    long long t = 0;
    int cnt = 0;
    for (int j = 0; j < n / 2; ++j) {
      if ((i >> j) & 1) {
        t ^= a[j];
        ++cnt;
      }
    }
    if (!f.count(t))
      f[t] = cnt;
    else
      f[t] = min(f[t], cnt);
  }

  for (int i = 0; i < (1 << (n - n / 2)); ++i) {  // 对后一半进行搜索
    long long t = 0;
    int cnt = 0;
```

```
    for (int j = 0; j < (n - n / 2); ++j) {
      if ((i >> j) & 1) {
        t ^= a[n / 2 + j];
        ++cnt;
      }
    }
    if (f.count((((long long)1 << n) - 1) ^ t))
      ans = min(ans, cnt + f[(((long long)1 << n) - 1) ^ t]);
  }

  cout << ans;

  return 0;
}

```

外部链接

- What is meet in the middle algorithm w.r.t. competitive programming? - Quora[3]
- Meet in the Middle Algorithm - YouTube[4]

外部参考资料

[1] 八数码难题

[2]「USACO09NOV」灯 Lights

[3] What is meet in the middle algorithm w.r.t. competitive programming? - Quora

[4] Meet in the Middle Algorithm - YouTube

6.5 启发式搜索

本页面将简要介绍启发式搜索及其用法.

定义

启发式搜索（英文：heuristic search）是一种在普通搜索算法的基础上引入了启发式函数的搜索算法.

启发式函数的作用是基于已有的信息对搜索的每一个分支选择都做估价，进而选择分支．简单来说，启发式搜索就是对取和不取都做分析，从中选取更优解或删去无效解.

例题

由于概念过于抽象，这里使用例题讲解.

「NOIP2005 普及组」采药[1]

题目大意：有 N 种物品和一个容量为 W 的背包，每种物品有重量 w_i 和价值 v_i 两种属性，要求选若干个物品（每种物品只能选一次）放入背包，使背包中物品的总价值最大，且背包中物品的总重量不超过背包的容量.

解题思路

我们写一个估价函数 f，可以剪掉所有无效的 0 枝条（就是剪去大量无用不选枝条）.

估价函数 f 的运行过程如下：

我们在取的时候判断一下是不是超过了规定体积（可行性剪枝）；在不取的时候判断一下不取这个时，剩下的药所有的价值 + 现有的价值是否大于目前找到的最优解（最优性剪枝）.

示例代码

```cpp
#include <algorithm>
#include <iostream>
using namespace std;
constexpr int N = 105;
int n, m, ans;

struct Node {
  int a, b;  // a 代表时间，b 代表价值
  double f;
} node[N];

bool operator<(Node p, Node q) { return p.f > q.f; }

int f(int t, int v) {  // 计算在当前时间下，剩余物品的最大价值
  int tot = 0;
  for (int i = 1; t + i <= n; i++)
    if (v >= node[t + i].a) {
      v -= node[t + i].a;
      tot += node[t + i].b;
    } else
      return (int)(tot + v * node[t + i].f);
  return tot;
}

void work(int t, int p, int v) {
  ans = max(ans, v);
  if (t > n) return;                          // 边界条件：只有n种物品
  if (f(t, p) + v > ans) work(t + 1, p, v);  // 最优性剪枝
  if (node[t].a <= p) work(t + 1, p - node[t].a, v + node[t].b);  // 可行性剪枝
}

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  cin >> m >> n;
  for (int i = 1; i <= n; i++) {
    cin >> node[i].a >> node[i].b;
    node[i].f = 1.0 * node[i].b / node[i].a;  // f为性价比
  }
  sort(node + 1, node + n + 1);  // 根据性价比排序
  work(1, m, 0);
  cout << ans << '\n';
  return 0;
}

```

外部参考资料

[1] 「NOIP2005 普及组」采药 

6.6 A*

本文介绍 A* 搜索算法.

A* 搜索算法（A* search algorithm，A* 读作 A-star），简称 A* 算法，是一种在带权有向图上，找到给定起点与终点之间的最短路径的算法．它属于图遍历（graph traversal）和最佳优先搜索算法（best-first search），亦是 BFS→6.3@p657 的改进.

过程

A* 算法的目标是找到有向图上从起点 s 到终点 t 的最短路径．设 $d(x,y)$ 为结点 x 与 y 之间的距离，也就是它们之间最短路径的长度．记 $g(x)=d(s,x)$ 为从起点 s 到结点 x 的距离函数，$h^*(x)$ 为从结点 x 到终点 t 的距离函数，$h(x)$ 为 $h^*(x)$ 的一个估计[1]．最后，记从 s 出发经由 x 到达 t 的最短路径长度的估计为

$$f(x)=g(x)+h(x).$$

搜索时，A* 算法每次从优先队列中取出一个 f 最小的结点．然后，将它的所有后继结点 x 都推入优先队列中，并利用实际记录的 $g(x)$ 和估计的 $h(x)$ 更新 $f(x)$.

性质

由于 $h^*(x)$ 的实际值在搜索的时候是未知的，所以，需要使用容易计算的 $h(x)$ 作为它的估计．A* 搜索的实际复杂度就取决于这一估计函数 $h(x)$ 的性质．容易想象，如果 $h\equiv h^*$，也就是说，估计是精确的，那么，搜索过程就会严格按照最短路径前进．而如果 $h\equiv 0$，那么，A* 算法就退化为 Dijkstra 算法→11.9.1@p2735；当 $h\equiv 0$ 并且边权为 1 时，这就是 BFS→6.3@p657.

假设图没有负权边．如果估计 $h(x)$ 永远不超过实际距离 $h^*(x)$，即 $0\le h\le h^*$，那么，A* 算法就一定能够找到最优解．满足这一条件的估计函数 $h(x)$ 称为 **可采纳的**（admissible）．根据前文的讨论，h 越接近 h^*，相应的 A* 算法效率就越高．一般来说，在最差情形中，算法会经过所有满足

$$f(x)=g(x)+h(x)\le C^*$$

的结点，其中，C^* 是起点 s 和终点 t 之间的最短距离．直觉上，h 越接近 h^*，每次扩展时，能够满足该条件的后继结点就越少，因此，算法搜索到的分支就越少．所以，A* 算法可以看作是对搜索算法的一种「剪枝」优化.

如果 h 不仅是可采纳的，还是 **一致的**（consistent），即

$$h(x)\le h(y)+d(x,y),$$

那么，A* 算法不会将已经弹出队列的结点再次加入队列．一致性条件，可以理解为结点 x,y,t 之间的三角形不等式.

例题

A* 算法的一个经典应用是解决 k 短路问题．关于该问题的描述、A* 做法，以及复杂度更优的可持久化可并堆做法，请移步 k 短路问题→11.9.3@p2750 页面.

本节介绍一个可以用 A* 算法解决的经典问题.

[1]此处的 h 意为 heuristic．详见 启发式搜索 - 维基百科[1] 和 A* search algorithm - Wikipedia[2] 的 Bounded relaxation 一节.

八数码[3]

在 3×3 的棋盘上，摆有八个棋子，每个棋子上标有 1 至 8 的某一数字．棋盘中留有一个空格，空格用 0 来表示．空格周围的棋子可以移到空格中，这样原来的位置就会变成空格．给出一种初始布局和目标布局（为了使题目简单，设目标状态如下），找到一种从初始布局到目标布局最少步骤的移动方法．

123
804
765

解题思路

h 函数可以定义为，不在应该在的位置的棋子个数．容易发现，h 既是可采纳的，也是一致的．此题可以使用 A* 算法求解．

参考代码

```
#include <algorithm>
#include <cstring>
#include <iostream>
#include <queue>
#include <set>
using namespace std;
constexpr int dx[4] = {1, -1, 0, 0}, dy[4] = {0, 0, 1, -1};
int fx, fy;
char ch;

struct matrix {
  int a[5][5];

  bool operator<(matrix x) const {
    for (int i = 1; i <= 3; i++)
      for (int j = 1; j <= 3; j++)
        if (a[i][j] != x.a[i][j]) return a[i][j] < x.a[i][j];
    return false;
  }
} f, st;

int h(matrix a) {
  int ret = 0;
  for (int i = 1; i <= 3; i++)
    for (int j = 1; j <= 3; j++)
      if (a.a[i][j] != st.a[i][j] && a.a[i][j] != 0) ret++;
  return ret;
}

struct node {
  matrix a;
  int t;

  bool operator<(node x) const { return t + h(a) > x.t + h(x.a); }
```

```
} x;

priority_queue<node> q;  // 搜索队列
set<matrix> s;           // 防止搜索队列重复

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  st.a[1][1] = 1;  // 定义标准表
  st.a[1][2] = 2;
  st.a[1][3] = 3;
  st.a[2][1] = 8;
  st.a[2][2] = 0;
  st.a[2][3] = 4;
  st.a[3][1] = 7;
  st.a[3][2] = 6;
  st.a[3][3] = 5;
  for (int i = 1; i <= 3; i++)  // 输入
    for (int j = 1; j <= 3; j++) {
      cin >> ch;
      f.a[i][j] = ch - '0';
    }
  s.insert(f);
  q.push({f, 0});
  while (!q.empty()) {
    x = q.top();
    q.pop();
    if (!h(x.a)) {  // 判断是否与标准矩阵一致
      cout << x.t << '\n';
      return 0;
    }
    for (int i = 1; i <= 3; i++)
      for (int j = 1; j <= 3; j++)
        if (!x.a.a[i][j]) fx = i, fy = j;  // 查找空格子（0号点）的位置
    for (int i = 0; i < 4; i++) {  // 对四种移动方式分别进行搜索
      int xx = fx + dx[i], yy = fy + dy[i];
      if (1 <= xx && xx <= 3 && 1 <= yy && yy <= 3) {
        swap(x.a.a[fx][fy], x.a.a[xx][yy]);
        if (!s.count(x.a))
          s.insert(x.a),
              q.push({x.a, x.t + 1});  // 这样移动后，将新的情况放入搜索队列中
        swap(x.a.a[fx][fy], x.a.a[xx][yy]);  // 如果不这样移动的情况
      }
    }
  }
  return 0;
}
```

参考资料与注释

- A* search algorithm - Wikipedia[4]

[1] 启发式搜索 - 维基百科 [2] A* search algorithm - Wikipedia [3] 八数码

[4] A* search algorithm - Wikipedia

6.7 迭代加深搜索

定义

迭代加深是一种 **每次限制搜索深度的** 深度优先搜索.

解释

迭代加深搜索的本质还是深度优先搜索，只不过在搜索的同时带上了一个深度 d，当 d 达到设定的深度时就返回，一般用于找最优解. 如果一次搜索没有找到合法的解，就让设定的深度加一，重新从根开始.

既然是为了找最优解，为什么不用 BFS 呢？我们知道 BFS 的基础是一个队列，队列的空间复杂度很大，当状态比较多或者单个状态比较大时，使用队列的 BFS 就显出了劣势. 事实上，迭代加深就类似于用 DFS 方式实现的 BFS，它的空间复杂度相对较小.

当搜索树的分支比较多时，每增加一层的搜索复杂度会出现指数级爆炸式增长，这时前面重复进行的部分所带来的复杂度几乎可以忽略，这也就是为什么迭代加深是可以近似看成 BFS 的.

过程

首先设定一个较小的深度作为全局变量，进行 DFS. 每进入一次 DFS，将当前深度加一，当发现 d 大于设定的深度 $limit$ 就返回. 如果在搜索的途中发现了答案就可以回溯，同时在回溯的过程中可以记录路径. 如果没有发现答案，就返回到函数入口，增加设定深度，继续搜索.

实现（伪代码）

```
IDDFS(u,d)
    if d>limit
        return
    else
        for each edge (u,v)
            IDDFS(v,d+1)
return
```

注意事项

在大多数的题目中，广度优先搜索还是比较方便的，而且容易判重. 当发现广度优先搜索在空间上不够优秀，而且要找最优解的问题时，就应该考虑迭代加深.

6.8 IDA*

前置知识：A* 算法→6.6@p667、迭代加深搜索→6.7@p670

本页面将简要介绍 IDA* 算法. IDA* 就是采用了迭代加深算法的 A* 算法.

过程

IDA* 算法是迭代加深搜索的一种变形. 迭代加深搜索在每次 DFS 中限制搜索深度，而 IDA* 则限制单次 DFS 的路径成本.

在一次迭代中，算法从起点 s 开始进行 DFS，记录到达当前结点 x 的实际成本 $g(x)$，并利用它到终点的最小成本估计 $h(x)$ 进行剪枝. 如果沿着当前路径到达终点的总成本估计

$$f(x) = g(x) + h(x)$$

超过阈值 C，则停止对该分支的搜索.

阈值 C 在迭代间动态更新. 初始阈值取为起点的总成本估计值 $h(s)$. 在一次迭代中，每当因超过阈值而停止时，就记录所有尚未访问的后继结点的总成本估计的最小值. 迭代结束后，将阈值更新为这一最小值，继续下一轮搜索.

性质

由于使用了和 A* 算法一样的剪枝策略，所以对 A* 算法性质的讨论对 IDA* 算法也适用.

和 A* 算法相比，IDA* 算法有如下优点:

- 不需要判重，不需要排序，利于深度剪枝.
- 空间需求减少. 每次迭代都是一个深度优先搜索，但是对搜索中的路径成本有限制，使用 DFS 可以减小空间消耗.

同时，它也有缺点:

- 重复搜索. 即使前后两次搜索相差微小，每次放宽限制都要再次从头搜索.

实现

设 h 是一个合适的估价函数，s 为搜索起点. 完整的算法流程大致如下所示:

Algorithm. IdaStar() :
Output. The shortest path, $path$, and its cost, C, if a path exists, and NOT_ FOUND, otherwise.
Method.

```
C ← h(s)
path ← [s]
while true
    t ← Search(path, 0, C)
    if t = FOUND then return (path, C)
    if t = ∞ then return NOT_ FOUND
    C ← t
```

Sub-Algorithm. Search($path, g, C$) :
Input. The current path, $path$, its cost, g, and search limit C.
Output. FOUND, if the target node has been reached; ∞, if all reachable nodes have been explored; otherwise, the minimum total cost, t, among nodes not yet explored.
Method.

```
node ← the last element in path
f ← g + h(node)
if f > C then return f
if node is the target then return FOUND
min ← ∞
for each child of node do
    if child not in path then
        append child to path
        t ← Search(path, g + Cost(node, child), C)
        if t = FOUND then return FOUND
        if t < min then min ← t
        remove the last element of path
return min
```

例题

埃及分数[1]

在古埃及，人们使用互不相同的单位分数（即 $1/a$，$a \in \mathbf{N}_+$）的和表示一切有理数．例如，$\frac{2}{3} = \frac{1}{2} + \frac{1}{6}$，但不允许 $\frac{2}{3} = \frac{1}{3} + \frac{1}{3}$，因为在加数中不允许有相同的单位分数．

对于一个分数 $\frac{a}{b}$，表示方法有很多种．规定：同一个分数的不同表示方法中，加数少的比加数多的好；如果加数个数相同，则最小的分数越大越好．例如，$\frac{19}{45} = \frac{1}{5} + \frac{1}{6} + \frac{1}{18}$ 是最佳方案．

输入整数 a, b（$0 < a < b < 1000$），试编程计算最佳表达式．

解题思路

这道题目理论上可以用回溯法求解，但是解答树会非常「恐怖」——不仅深度没有明显的上界，而且加数的选择理论上也是无限的．换句话说，如果用宽度优先遍历，连一层都扩展不完，因为每一层都是无限大的．

解决方案是采用迭代加深搜索：从小到大枚举深度上限 C，每次搜索只考虑深度不超过 C 的结点．这样，只要解的深度有限，则一定可以在有限时间内枚举到．

深度上限 C 还可以用来剪枝．按照分母递增的顺序来进行扩展，如果扩展到 i 层时，前 i 个分数之和为 $\frac{c}{d}$，而第 i 个分数为 $\frac{1}{e}$，则接下来至少还需要

$$h = \left(\frac{a}{b} - \frac{c}{d}\right) / \left(\frac{1}{e+1}\right)$$

个分数，总和才能达到 $\frac{a}{b}$．例如，当前搜索到 $\frac{19}{45} = \frac{1}{5} + \frac{1}{100} + \cdots$，则后面的分数每个最大为 $\frac{1}{101}$，至少需要 $\left(\frac{19}{45} - \frac{1}{5}\right) / \left(\frac{1}{101}\right) = 23$ 项总和才能达到 $\frac{19}{45}$，因此前 22 次迭代是根本不会考虑这棵子树的．这里的关键在于：可以估计至少还要多少步才能出解．

注意，这里使用「至少」一词表示估计是「乐观的」．和 A* 算法一样，好的估计函数都需要是「乐观的」，也就是说，它不能高估实际成本．将迭代加深搜索中的深度限制 $g \le C$ 替换为更严格的限制 $g + h \le C$，就得到了本页面所讨论的 IDA* 算法．因为本文中的路径成本就是它的长度，所以，IDA* 算法同样是对路径长度进行限制，只是加上了对于还需要多少步的估计．更一般的问题中，根据具体要最小化的成本不同，还可以设计出其他的估计函数．

在实现中，对 IDA* 算法进一步剪枝优化：

1. 扩展结点时，下一个要考虑的分母至少是 $\left(\frac{a}{b} - \frac{c}{d}\right)^{-1}$，可以以此改进枚举 e 的起点．
2. IDA* 的路径成本限制可以变形为

   $$e \le \left(\frac{a}{b} - \frac{c}{d}\right)^{-1} (C - g) - 1.$$

   所以，不必枚举所有的后续分母再逐个判断，只需要枚举到这个上界即可．
3. 在搜索到最后两个分数时，直接利用二次方程计算是否可行，而非继续搜索．具体地，要找到 $e < x < y \le E_{\max}$ 使得

   $$\frac{1}{x} + \frac{1}{y} = \frac{p}{q} := \frac{a}{b} - \frac{c}{d},$$

   只需要求解二元二次方程组

   $$\begin{cases} x + y = kp, \\ xy = kq \end{cases}$$

   即可，其中，$k \in \mathbf{N}_+$．由二次方程的知识可知，方程组在

$$\Delta = k^2p^2 - 4kq > 0 \iff k > \frac{4q}{p^2}$$

时，才有两个不同的实根

$$x = \frac{kp - \sqrt{\Delta}}{2},\ y = \frac{kp + \sqrt{\Delta}}{2}.$$

因此，可以直接枚举所有可行的 k，判断是否存在这样一组整数解．枚举 k 时，上界通过 $y < E_{\max}$ 判断．

4. 每次得到一组答案时，都将分母的上界 M_e 调整到当前答案中的最大分母减一．

另外，实现中，直接记录了 $\frac{a}{b} - \frac{c}{d}$ 和 $C - g$ 的取值，前者的分子和分母分别存储在变量 `a` 和 `b` 中，后者则存储为变量 `d`.

示例代码

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

int max_e = 1e7;
std::vector<int> ans;
std::vector<int> current;

long long gcd(long long x, long long y) { return y ? gcd(y, x % y) : x; }

bool dfs(int d, long long a, long long b, int e) {
  long long _gcd = gcd(a, b);
  a /= _gcd;
  b /= _gcd;

  bool solved = false;
  if (d == 2) {
    for (int k = 4 * b / (a * a) + 1;; ++k) {
      long long delta = a * a * k * k - 4 * b * k;
      long long t = std::sqrt(delta + 0.5l);
      long long x = (a * k - t) / 2;
      long long y = (a * k + t) / 2;
      if (y > max_e) break;
      if (!t || t * t != delta || (a * k - t) % 2 != 0) continue;
      ans = current;
      ans.push_back(x);
      ans.push_back(y);
      max_e = y - 1;
      solved = true;
    }
  } else {
    int e1 = std::max(e + 1, int((b + a - 1) / a));
    for (; e1 <= d * b / a && e1 <= max_e; e1++) {
      current.push_back(e1);
      // a/b - 1/e
      solved |= dfs(d - 1, a * e1 - b, b * e1, e1);
```

```
      current.pop_back();
    }
  }
  return solved;
}

int solve(int a, int b) {
  if (b % a == 0) {
    ans.push_back(b / a);
    return 1;
  }
  for (int lim = 2; lim <= 100; lim++)
    if (dfs(lim, a, b, 1)) return lim;
  return -1;
}

int main() {
  int a, b;
  std::cin >> a >> b;
  solve(a, b);
  for (auto x : ans) std::cout << x << ' ';
  std::cout << std::endl;
  return 0;
}

```

习题

- UVa1343 旋转游戏[2]

外部参考资料

[1] 埃及分数  [2] UVa1343 旋转游戏 

6.9 回溯法

本页面将简要介绍回溯法的概念和应用.

简介

回溯法是一种经常被用在 深度优先搜索（DFS）→6.2@p654 和 广度优先搜索（BFS）→6.3@p657 的技巧.

其本质是：走不通就回头.

过程

1. 构造空间树；
2. 进行遍历；
3. 如遇到边界条件，即不再向下搜索，转而搜索另一条链；
4. 达到目标条件，输出结果.

例题

USACO 1.5.4 Checker Challenge[1]

现在有一个如下的 6×6 的跳棋棋盘，有六个棋子被放置在棋盘上，使得每行，每列，每条对角线（包括两条主对角线的所有对角线）上都至多有一个棋子.

```
0   1   2   3   4   5   6
  ------------------------
1 |   | 0 |   |   |   |   |
  ------------------------
2 |   |   |   | 0 |   |   |
  ------------------------
3 |   |   |   |   |   | 0 |
  ------------------------
4 | 0 |   |   |   |   |   |
  ------------------------
5 |   |   | 0 |   |   |   |
  ------------------------
6 |   |   |   |   | 0 |   |
  ------------------------
```

上面的布局可以用序列 $\{2,4,6,1,3,5\}$ 来描述，第 i 个数字表示在第 i 行的第 a_i 列有一个棋子，如下所示

行号 i：$\{1,2,3,4,5,6\}$

列号 a_i：$\{2,4,6,1,3,5\}$

这只是跳棋放置的一个方案. 请编一个程序找出所有方案并把它们以上面的序列化方法输出，按字典顺序排列. 你只需输出前 3 个解并在最后一行输出解的总个数. 特别注意：你需要优化你的程序以保证在更大棋盘尺寸下的程序效率.

参考代码

```cpp
// 该代码为回溯法的 DFS 实现
#include <iostream>
using namespace std;
int ans[14], check[3][28] = {0}, sum = 0, n;

void eq(int line) {
  if (line > n) {  // 如果已经搜索完n行
    sum++;
    if (sum > 3)
      return;
    else {
      for (int i = 1; i <= n; i++) cout << ans[i] << ' ';
      cout << '\n';
      return;
    }
  }
  for (int i = 1; i <= n; i++) {
    if ((!check[0][i]) && (!check[1][line + i]) &&
        (!check[2][line - i + n])) {  // 判断在某位置放置是否合法
```

```
      ans[line] = i;
      check[0][i] = 1;
      check[1][line + i] = 1;
      check[2][line - i + n] = 1;
      eq(line + 1);
      // 向下递归后进行回溯，方便下一轮递归
      check[0][i] = 0;
      check[1][line + i] = 0;
      check[2][line - i + n] = 0;
    }
  }
}

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  cin >> n;
  eq(1);
  cout << sum;
  return 0;
}

```

迷宫[2]

现有一个尺寸为 $N \times M$ 的迷宫，迷宫里有 T 处障碍，障碍处不可通过．给定起点坐标和终点坐标，且每个方格最多经过一次，问有多少种从起点坐标到终点坐标的方案．在迷宫中移动有上、下、左、右四种移动方式，每次只能移动一个方格．数据保证起点上没有障碍．

参考代码

```
// 该代码为回溯法的 BFS 实现
#include <cstring>
#include <iostream>
#include <queue>
using namespace std;
int n, m, k, x, y, a, b, ans;
int dx[4] = {0, 0, 1, -1}, dy[4] = {1, -1, 0, 0};  // 四个方向
bool vis[6][6];

struct oo {
  int x, y, used[6][6];
};

oo sa;

void bfs() {
  queue<oo> q;
  sa.x = x;
  sa.y = y;
  sa.used[x][y] = 1;
```

```
  q.push(sa);
  while (!q.empty()) {  // BFS队列
    oo now = q.front();
    q.pop();
    for (int i = 0; i < 4; i++) {  // 枚举向四个方向走
      int sx = now.x + dx[i];
      int sy = now.y + dy[i];
      if (now.used[sx][sy] || vis[sx][sy] || sx == 0 || sy == 0 || sx > n ||
          sy > m)
        continue;
      if (sx == a && sy == b) {
        ans++;
        continue;
      }
      sa.x = sx;
      sa.y = sy;
      memcpy(sa.used, now.used, sizeof(now.used));
      sa.used[sx][sy] = 1;
      q.push(sa);  // 假设向此方向走，放入BFS队列
    }
  }
}

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  cin >> n >> m >> k;
  cin >> x >> y >> a >> b;
  for (int i = 1, aa, bb; i <= k; i++) {
    cin >> aa >> bb;
    vis[aa][bb] = true;  // 障碍位置不可通过
  }
  bfs();
  cout << ans;
  return 0;
}

```

外部参考资料

[1] USACO 1.5.4 Checker Challenge  [2] 迷宫 

6.10 Dancing Links

Authors: LeverImmy, 383494

本页面将介绍精确覆盖问题、重复覆盖问题，解决这两个问题的算法「X 算法」，以及用来优化 X 算法的双向十字链表 Dancing Link．本页也将介绍如何在建模的配合下使用 DLX 解决一些搜索题．

精确覆盖问题

定义

精确覆盖问题（英文：Exact Cover Problem）是指给定许多集合 $S_i(1 \le i \le n)$ 以及一个集合 X，求满足以下条件的无序多元组 $(T_1, T_2, \cdots, T_m)$：

1. $\forall i, j \in [1, m], T_i \bigcap T_j = \emptyset (i \neq j)$
2. $X = \bigcup_{i=1}^{m} T_i$
3. $\forall i \in [1, m], T_i \in \{S_1, S_2, \cdots, S_n\}$

解释

例如，若给出

$$S_1 = \{5, 9, 17\}$$
$$S_2 = \{1, 8, 119\}$$
$$S_3 = \{3, 5, 17\}$$
$$S_4 = \{1, 8\}$$
$$S_5 = \{3, 119\}$$
$$S_6 = \{8, 9, 119\}$$
$$X = \{1, 3, 5, 8, 9, 17, 119\}$$

则 (S_1, S_4, S_5) 为一组合法解.

问题转化

将 $\bigcup_{i=1}^{n} S_i$ 中的所有数离散化，可以得到这么一个模型：

> 给定一个 01 矩阵，你可以选择一些行（row），使得最终每列（column）[1]都恰好有一个 1. 举个例子，我们对上文中的例子进行建模，可以得到这么一个矩阵：

$$\begin{pmatrix} 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{pmatrix}$$

> 其中第 i 行表示着 S_i，而这一行的每个数依次表示 $[1 \in S_i], [3 \in S_i], [5 \in S_i], \cdots, [119 \in S_i]$.

实现

暴力 1

一种方法是枚举选择哪些行，最后检查这个方案是否合法.

因为每一行都有选或者不选两种状态，所以枚举行的时间复杂度是 $O(2^n)$ 的；

而每次检查都需要 $O(nm)$ 的时间复杂度. 所以总的复杂度是 $O(nm \cdot 2^n)$.

实现

```
int ok = 0;
for (int state = 0; state < 1 << n; ++state) {  // 枚举每行是否被选
```

[1]（两岸用语差异）台灣：直行（column）、橫列（row）

```
  for (int i = 1; i <= n; ++i)
    if ((1 << i - 1) & state)
      for (int j = 1; j <= m; ++j) a[i][j] = 1;
  int flag = 1;
  for (int j = 1; j <= m; ++j)
    for (int i = 1, bo = 0; i <= n; ++i)
      if (a[i][j]) {
        if (bo)
          flag = 0;
        else
          bo = 1;
      }
  if (!flag)
    continue;
  else {
    ok = 1;
    for (int i = 1; i <= n; ++i)
      if ((1 << i - 1) & state) printf("%d ", i);
    puts("");
  }
  memset(a, 0, sizeof(a));
}
if (!ok) puts("No solution.");
```

暴力 2

考虑到 01 矩阵的特殊性质，每一行都可以看做一个 m 位二进制数.

因此原问题转化为

> 给定 n 个 m 位二进制数，要求选择一些数，使得任意两个数的与都为 0，且所有数的或为 2^m-1.
>
> `tmp` 表示的是截至目前被选中的二进制数的或.

因为每一行都有选或者不选两种状态，所以枚举行的时间复杂度为 $O(2^n)$;

而每次计算 `tmp` 都需要 $O(n)$ 的时间复杂度. 所以总的复杂度为 $O(n\cdot 2^n)$.

实现

```
int ok = 0;
for (int i = 1; i <= n; ++i)
  for (int j = m; j >= 1; --j) num[i] = num[i] << 1 | a[i][j];
for (int state = 0; state < 1 << n; ++state) {
  int tmp = 0;
  bool flag = true;
  for (int i = 1; i <= n; ++i)
    if ((1 << i - 1) & state) {
      if (tmp & num[i]) {
        flag = false;
        break;
      }
      tmp |= num[i];
    }
```

```
15   if (flag && tmp == (1 << m) - 1) {
16     ok = 1;
17     for (int i = 1; i <= n; ++i)
18       if ((1 << i - 1) & state) printf("%d ", i);
19     puts("");
20   }
21 }
22 if (!ok) puts("No solution.");
```

重复覆盖问题

重复覆盖问题与精确覆盖问题类似，但没有对元素相似性的限制．下文介绍的 X 算法→6@p653 原本针对精确覆盖问题，但经过一些修改和优化（已标注在其中）同样可以高效地解决重复覆盖问题．

X 算法

Donald E. Knuth 提出了 X 算法 (Algorithm X)，其思想与刚才的暴力差不多，但是方便优化．

过程

继续以上文中中提到的例子为载体，得到一个这样的 01 矩阵：

$$\begin{pmatrix} 0&0&1&0&1&1&0\\ 1&0&0&1&0&0&1\\ 0&1&1&0&0&1&0\\ 1&0&0&1&0&0&0\\ 0&1&0&0&0&0&1\\ 0&0&0&1&1&0&1 \end{pmatrix}$$

1. 此时第一行有 3 个 1，第二行有 3 个 1，第三行有 3 个 1，第四行有 2 个 1，第五行有 2 个 1，第六行有 3 个 1．选择第一行，将它删除，并将所有 1 所在的列打上标记；

$$\begin{pmatrix} 0&0&1&0&1&1&0\\ 1&0&0&1&0&0&1\\ 0&1&1&0&0&1&0\\ 1&0&0&1&0&0&0\\ 0&1&0&0&0&0&1\\ 0&0&0&1&1&0&1 \end{pmatrix}$$

2. 选择所有被标记的列，将它们删除，并将这些列中含 1 的行打上标记（重复覆盖问题无需打标记）；

$$\begin{pmatrix} 0&0&1&0&1&1&0\\ 1&0&0&1&0&0&1\\ 0&1&1&0&0&1&0\\ 1&0&0&1&0&0&0\\ 0&1&0&0&0&0&1\\ 0&0&0&1&1&0&1 \end{pmatrix}$$

3. 选择所有被标记的行，将它们删除；

$$\begin{pmatrix} 0&0&1&0&1&1&0\\ 1&0&0&1&0&0&1\\ 0&1&1&0&0&1&0\\ 1&0&0&1&0&0&0\\ 0&1&0&0&0&0&1\\ 0&0&0&1&1&0&1 \end{pmatrix}$$

这表示这一行已被选择，且这一行的所有 1 所在的列不能有其他 1 了．

于是得到一个新的小 01 矩阵：

$$\begin{pmatrix} 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix}$$

4. 此时第一行（原来的第二行）有 3 个 1，第二行（原来的第四行）有 2 个 1，第三行（原来的第五行）有 2 个 1．选择第一行（原来的第二行），将它删除，并将所有 1 所在的列打上标记；

$$\begin{pmatrix} 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix}$$

5. 选择所有被标记的列，将它们删除，并将这些列中含 1 的行打上标记；

$$\begin{pmatrix} 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix}$$

6. 选择所有被标记的行，将它们删除；

$$\begin{pmatrix} 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix}$$

这样就得到了一个空矩阵．但是上次删除的行 `1 0 1 1` 不是全 1 的，说明选择有误；

()

7. 回溯到步骤 4，考虑选择第二行（原来的第四行），将它删除，并将所有 1 所在的列打上标记；

$$\begin{pmatrix} 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix}$$

8. 选择所有被标记的列，将它们删除，并将这些列中含 1 的行打上标记；

$$\begin{pmatrix} 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix}$$

9. 选择所有被标记的行，将它们删除；

$$\begin{pmatrix} 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix}$$

于是我们得到了这样的一个矩阵：

$$\begin{pmatrix} 1 & 1 \end{pmatrix}$$

10. 此时第一行（原来的第五行）有 2 个 1，将它们全部删除，得到一个空矩阵：

()

11. 上一次删除的时候，删除的是全 1 的行，因此成功，算法结束．

答案即为被删除的三行：$1, 4, 5$.

强烈建议自己模拟一遍矩阵删除、还原与回溯的过程后，再接着阅读下文．

通过上述步骤，可将 X 算法的流程概括如下：

1. 对于现在的矩阵 M，选择并标记一行 r，将 r 添加至 S 中；
2. 如果尝试了所有的 r 却无解，则算法结束，输出无解；
3. 标记与 r 相关的行 r_i 和 c_i（相关的行和列与 X 算法→6@p653 中第 2 步定义相同，下同）；
4. 删除所有标记的行和列，得到新矩阵 M'；
5. 如果 M' 为空，且 r 为全 1，则算法结束，输出被删除的行组成的集合 S；

   如果 M' 为空，且 r 不全为 1，则恢复与 r 相关的行 r_i 以及列 c_i，跳转至步骤 1；

如果 M' 不为空，则跳转至步骤 1.

不难看出，X 算法需要大量的「删除行」、「删除列」和「恢复行」、「恢复列」的操作.

一个朴素的想法是，使用一个二维数组存放矩阵，再用四个数组分别存放每一行与之相邻的行编号，每次删除和恢复仅需更新四个数组中的元素. 但由于一般问题的矩阵中 0 的数量远多于 1 的数量，这样做的空间复杂度难以接受.

Donald E. Knuth 想到了用双向十字链表来维护这些操作.

而在双向十字链表上不断跳跃的过程被形象地比喻成「跳跃」，因此被用来优化 X 算法的双向十字链表也被称为「Dancing Links」.

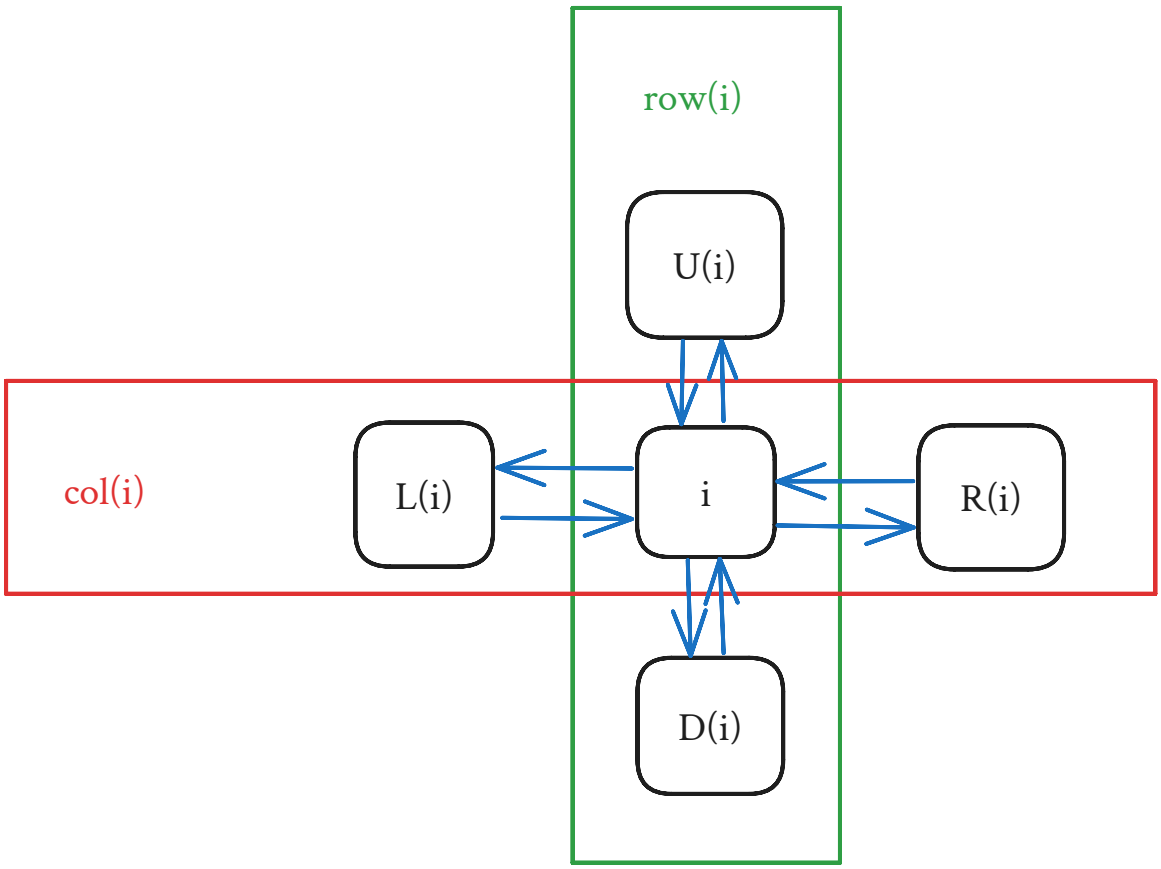
Dancing Links 优化的 X 算法

预编译命令

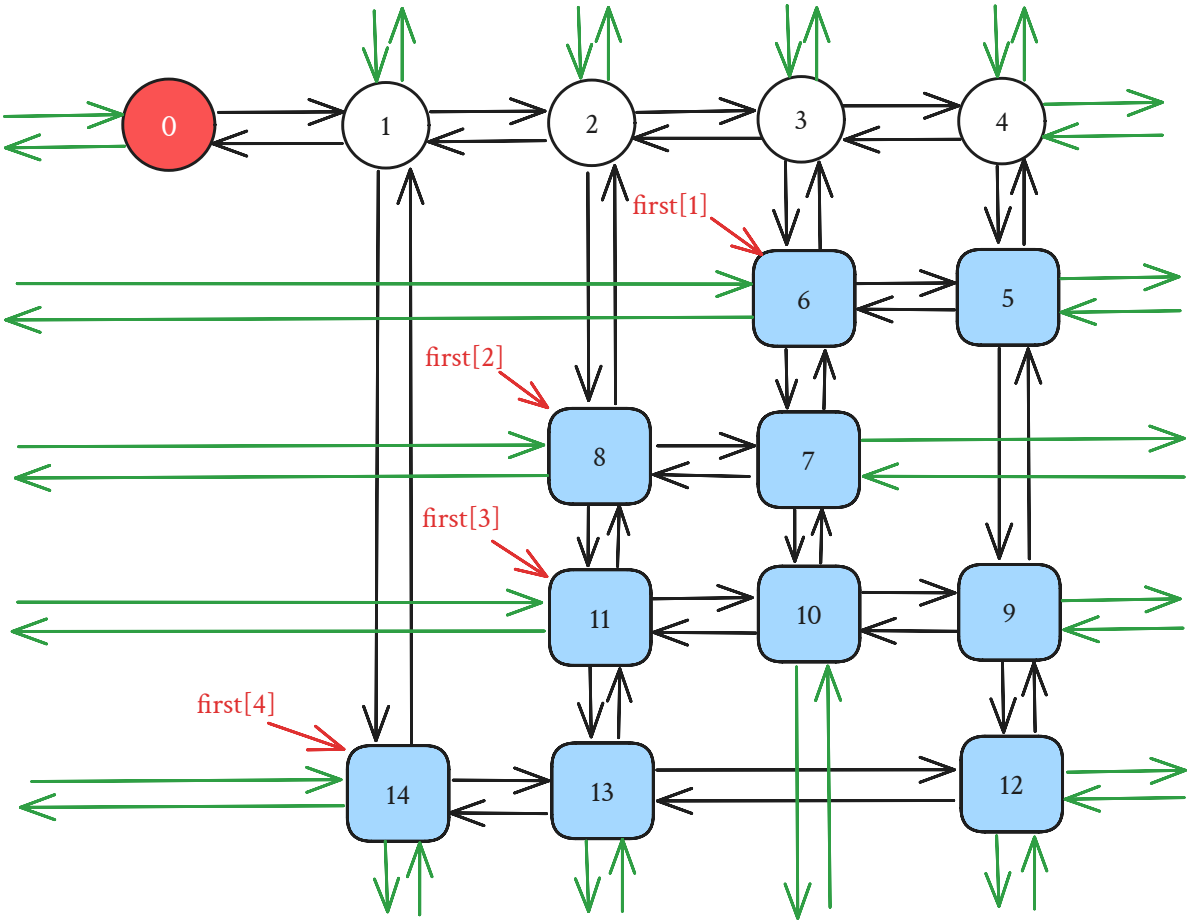
```
1 #define IT(i, A, x) for (i = A[x]; i != x; i = A[i])
```

定义

双向十字链表中存在四个指针域，分别指向上、下、左、右的元素；且每个元素 i 在整个双向十字链表系中都对应着一个格子，因此还要表示 i 所在的列和所在的行，如图所示：



大型的双向链表则更为复杂：



每一行都有一个行首指示，每一列都有一个列指示.

行首指示为 `first[]`，列指示是我们新建的 $c+1$ 个哨兵结点. 值得注意的是，**行首指示并非是链表中的哨兵结点**. 它是虚拟的，类似于邻接表中的 `first[]` 数组，**直接指向** 这一行中的首元素.

同时，每一列都有一个 `siz[]` 表示这一列的元素个数.

特殊地，0 号结点无右结点等价于这个 Dancing Links 为空.

```
constexpr int MS = 1e5 + 5;
int n, m, idx, first[MS], siz[MS];
int L[MS], R[MS], U[MS], D[MS];
int col[MS], row[MS];
```

过程

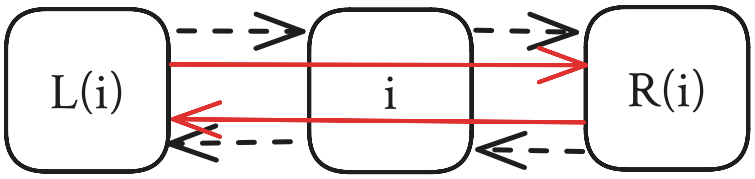
remove 操作

`remove(c)` 表示在 Dancing Links 中删除第 c 列以及与其相关的行和列.

先将 c 删除，此时：

- c 左侧的结点的右结点应为 c 的右结点.
- c 右侧的结点的左结点应为 c 的左结点.

即 `L[R[c]] = L[c], R[L[c]] = R[c];`.



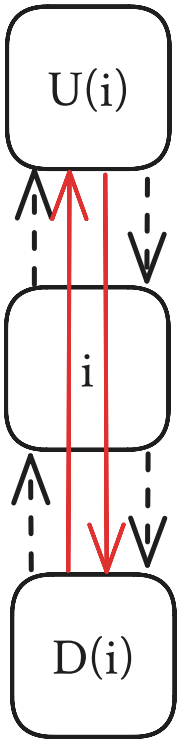
然后顺着这一列往下走，把走过的每一行都删掉.

如何删掉每一行呢？枚举当前行的指针 j，此时：

- j 上方的结点的下结点应为 j 的下结点.
- j 下方的结点的上结点应为 j 的上结点.

注意要修改每一列的元素个数.

即`U[D[j]] = U[j], D[U[j]] = D[j], --siz[col[j]];`.



`remove` 函数的代码实现如下：

实现

```
void remove(const int &c) {
  int i, j;
  L[R[c]] = L[c], R[L[c]] = R[c];
  // 顺着这一列从上往下遍历
  IT(i, D, c)
  // 顺着这一行从左往右遍历
  IT(j, R, i)
  U[D[j]] = U[j], D[U[j]] = D[j], --siz[col[j]];
}
```

recover 操作

`recover(c)` 表示在 Dancing Links 中还原第 c 列以及与其相关的行和列.

`recover(c)` 即 `remove(c)` 的逆操作，这里不再赘述.

值得注意的是，`recover(c)` 的所有操作的顺序与 `remove(c)` 的操作恰好相反.

`recover(c)` 的代码实现如下:

实现

```
void recover(const int &c) {
  int i, j;
  IT(i, U, c) IT(j, L, i) U[D[j]] = D[U[j]] = j, ++siz[col[j]];
  L[R[c]] = R[L[c]] = c;
}
```

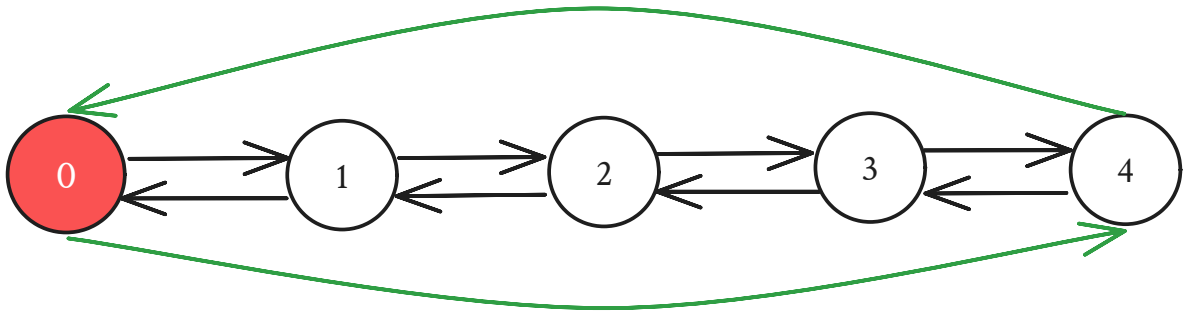
build 操作

`build(r, c)` 表示新建一个大小为 $r \times c$，即有 r 行，c 列的 Dancing Links.

新建 $c+1$ 个结点作为列指示.

第 i 个点的左结点为 $i-1$，右结点为 $i+1$，上结点为 i，下结点为 i. 特殊地，0 结点的左结点为 c，c 结点的右结点为 0.

于是我们得到了一个环状双向链表:



这样就初始化了一个 Dancing Links.

`build(r, c)` 的代码实现如下:

实现

```
void build(const int &r, const int &c) {
  n = r, m = c;
  for (int i = 0; i <= c; ++i) {
    L[i] = i - 1, R[i] = i + 1;
    U[i] = D[i] = i;
  }
  L[0] = c, R[c] = 0, idx = c;
  memset(first, 0, sizeof(first));
  memset(siz, 0, sizeof(siz));
}
```

insert 操作

`insert(r, c)` 表示在第 r 行，第 c 列插入一个结点.

插入操作分为两种情况：

- 如果第 r 行没有元素，那么直接插入一个元素，并使 `first[r]` 指向这个元素.

  这可以通过 `first[r] = L[idx] = R[idx] = idx;` 来实现.
- 如果第 r 行有元素，那么将这个新元素用一种特殊的方式与 c 和 $first(r)$ 连接起来.

  设这个新元素为 idx，然后：
  - 把 idx 插入到 c 的正下方，此时：
    - idx 下方的结点为原来 c 的下结点；
    - idx 下方的结点（即原来 c 的下结点）的上结点为 idx
    - idx 的上结点为 c；
    - c 的下结点为 idx.

    注意记录 idx 的所在列和所在行，以及更新这一列的元素个数.

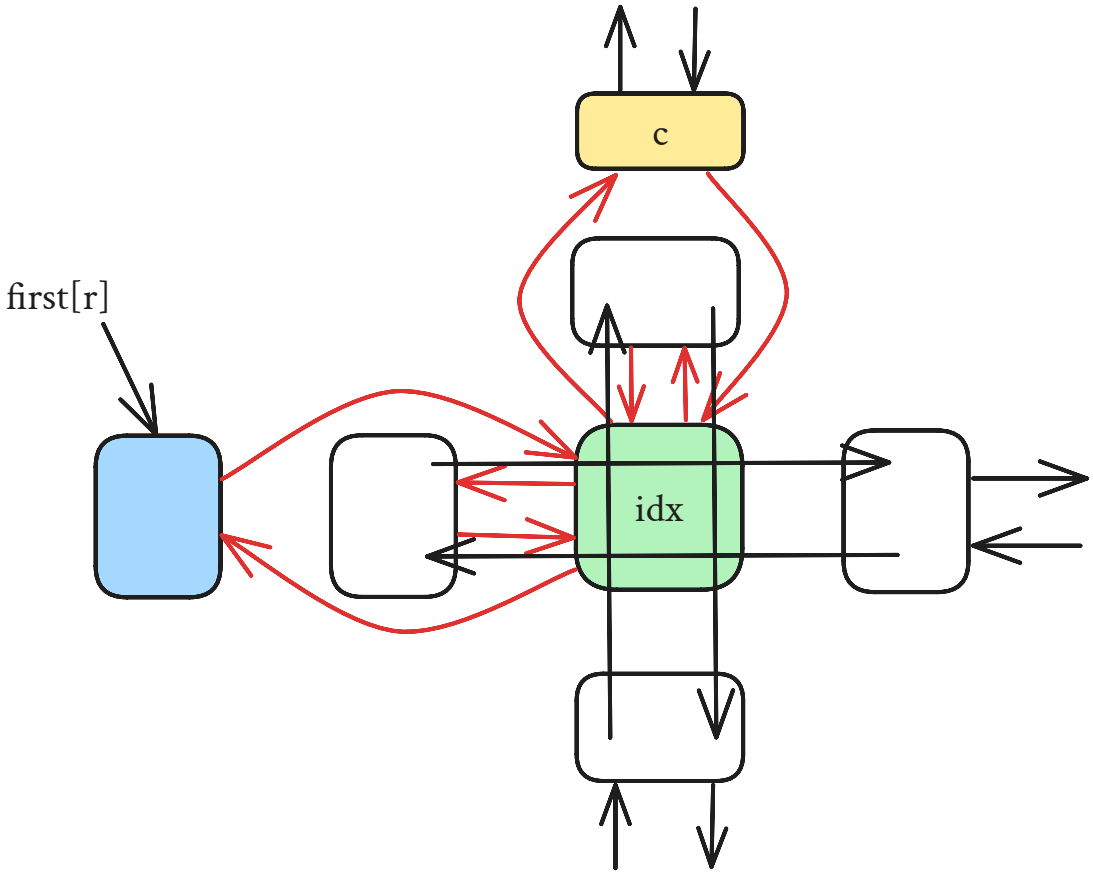
    ```
    col[++idx] = c, row[idx] = r, ++siz[c];
    U[idx] = c, D[idx] = D[c], U[D[c]] = idx, D[c] = idx;
    ```

    强烈建议读者完全掌握这几步的顺序后再继续阅读本文.
  - 把 idx 插入到 $first(r)$ 的正右方，此时：
    - idx 右侧的结点为原来 $first(r)$ 的右结点；
    - 原来 $first(r)$ 右侧的结点的左结点为 idx；
    - idx 的左结点为 $first(r)$；
    - $first(r)$ 的右结点为 idx.

    ```
    L[idx] = first[r], R[idx] = R[first[r]];
    L[R[first[r]]] = idx, R[first[r]] = idx;
    ```

    强烈建议读者完全掌握这几步的顺序后再继续阅读本文.

`insert(r, c)` 这个操作可以通过图片来辅助理解：



留心曲线箭头的方向.

`insert(r, c)` 的代码实现如下:

实现

```
void insert(const int &r, const int &c) {
  row[++idx] = r, col[idx] = c, ++siz[c];
  U[idx] = c, D[idx] = D[c], U[D[c]] = idx, D[c] = idx;
  if (!first[r])
    first[r] = L[idx] = R[idx] = idx;
  else {
    L[idx] = first[r], R[idx] = R[first[r]];
    L[R[first[r]]] = idx, R[first[r]] = idx;
  }
}
```

dance 操作

`dance()` 即为递归地删除以及还原各个行列的过程.

1. 如果 0 号结点没有右结点，那么矩阵为空，记录答案并返回;
2. 选择列元素个数最少的一列，并删掉这一列;
3. 遍历这一列所有有 1 的行，枚举它是否被选择;
4. 递归调用 `dance()`，如果可行，则返回;如果不可行，则恢复被选择的行;
5. 如果无解，则返回.

`dance()` 的代码实现如下:

实现

```
bool dance(int dep) {
  int i, j, c = R[0];
  if (!R[0]) {
    ans = dep;
    return true;
  }
  IT(i, R, 0) if (siz[i] < siz[c]) c = i;
  remove(c);
  IT(i, D, c) {
    stk[dep] = row[i];
    IT(j, R, i) remove(col[j]);
    if (dance(dep + 1)) return true;
    IT(j, L, i) recover(col[j]);
  }
  recover(c);
  return false;
}
```

其中 `stk[]` 用来记录答案.

注意我们每次优先选择列元素个数最少的一列进行删除, 这样能保证程序具有一定的启发性, 使搜索树分支最少.

对于重复覆盖问题，在搜索时可以用估价函数（与 A*→6.6@p667 中类似）进行剪枝：若当前最好情况下所选行数超过目前最优解，则可以直接返回.

模板

模板代码[1]

```
#include <cctype>
#include <cstring>
#include <iostream>
constexpr int N = 500 + 10;
int n, m, ans;
int stk[N];

struct DLX {
  static constexpr int MAXSIZE = 1e5 + 10;
  int n, m, tot, first[MAXSIZE + 10], siz[MAXSIZE + 10];
  int L[MAXSIZE + 10], R[MAXSIZE + 10], U[MAXSIZE + 10], D[MAXSIZE + 10];
  int col[MAXSIZE + 10], row[MAXSIZE + 10];

  void build(const int &r, const int &c) {  // 进行build操作
    n = r, m = c;
    for (int i = 0; i <= c; ++i) {
      L[i] = i - 1, R[i] = i + 1;
      U[i] = D[i] = i;
    }
    L[0] = c, R[c] = 0, tot = c;
```

```
21      memset(first, 0, sizeof(first));
22      memset(siz, 0, sizeof(siz));
23    }
24
25    void insert(const int &r, const int &c) {  // 进行insert操作
26      col[++tot] = c, row[tot] = r, ++siz[c];
27      D[tot] = D[c], U[D[c]] = tot, U[tot] = c, D[c] = tot;
28      if (!first[r])
29        first[r] = L[tot] = R[tot] = tot;
30      else {
31        R[tot] = R[first[r]], L[R[first[r]]] = tot;
32        L[tot] = first[r], R[first[r]] = tot;
33      }
34    }
35
36    void remove(const int &c) {  // 进行remove操作
37      int i, j;
38      L[R[c]] = L[c], R[L[c]] = R[c];
39      for (i = D[c]; i != c; i = D[i])
40        for (j = R[i]; j != i; j = R[j])
41          U[D[j]] = U[j], D[U[j]] = D[j], --siz[col[j]];
42    }
43
44    void recover(const int &c) {  // 进行recover操作
45      int i, j;
46      for (i = U[c]; i != c; i = U[i])
47        for (j = L[i]; j != i; j = L[j]) U[D[j]] = D[U[j]] = j, ++siz[col[j]];
48      L[R[c]] = R[L[c]] = c;
49    }
50
51    bool dance(int dep) {  // dance
52      if (!R[0]) {
53        ans = dep;
54        return true;
55      }
56      int i, j, c = R[0];
57      for (i = R[0]; i != 0; i = R[i])
58        if (siz[i] < siz[c]) c = i;
59      remove(c);
60      for (i = D[c]; i != c; i = D[i]) {
61        stk[dep] = row[i];
62        for (j = R[i]; j != i; j = R[j]) remove(col[j]);
63        if (dance(dep + 1)) return true;
64        for (j = L[i]; j != i; j = L[j]) recover(col[j]);
65      }
66      recover(c);
67      return false;
68    }
69  } solver;
70
71  using std::cin;
72  using std::cout;
```

```
73
74 int main() {
75   cin.tie(nullptr)->sync_with_stdio(false);
76   cin >> n >> m;
77   solver.build(n, m);
78   for (int i = 1; i <= n; ++i)
79     for (int j = 1; j <= m; ++j) {
80       int x;
81       cin >> x;
82       if (x) solver.insert(i, j);
83     }
84   solver.dance(1);
85   if (ans)
86     for (int i = 1; i < ans; ++i) cout << stk[i] << ' ';
87   else
88     cout << "No Solution!\n";
89   return 0;
90 }
91
```

性质

DLX 递归及回溯的次数与矩阵中 1 的个数有关，与矩阵的 r, c 等参数无关. 因此，它的时间复杂度是**指数级**的，理论复杂度大概在 $O(c^n)$ 左右，其中 c 为某个非常接近于 1 的常数，n 为矩阵中 1 的个数.

但实际情况下 DLX 表现良好，一般能解决大部分的问题.

建模

DLX 的难点，不全在于链表的建立，而在于建模.

请确保已经完全掌握 DLX 模板后再继续阅读本文.

我们每拿到一个题，应该考虑行和列所表示的意义：

- 行表示决策，因为每行对应着一个集合，也就对应着选/不选；
- 列表示状态，因为第 i 列对应着某个条件 P_i.

对于某一行而言，由于不同的列的值不尽相同，我们**由不同的状态，定义了一个决策**.

例题 1 P1784 数独[2]

解题思路

先考虑决策是什么.

在这一题中，每一个决策可以用形如 (r, c, w) 的有序三元组表示.

注意到「宫」并不是决策的参数，因为它**可以被每个确定的 (r, c) 表示**.

因此有 $9 \times 9 \times 9 = 729$ 行.

再考虑状态是什么.

我们思考一下 (r, c, w) 这个决策将会造成什么影响. 记 (r, c) 所在的宫为 b.

1. 第 r 行用了一个 w（用 $9 \times 9 = 81$ 列表示）；
2. 第 c 列用了一个 w（用 $9 \times 9 = 81$ 列表示）；

3. 第 b 宫用了一个 w（用 $9\times 9=81$ 列表示）；
4. (r,c) 中填入了一个数（用 $9\times 9=81$ 列表示）.

因此有 $81\times 4=324$ 列，共 $729\times 4=2916$ 个 1.

至此，我们成功地将 9×9 的数独问题转化成了一个**有 729 行，324 列，共 2916 个** 1 的精确覆盖问题.

参考代码

```cpp
#include <cctype>
#include <cstring>
#include <iostream>
constexpr int N = 1e6 + 10;
int ans[10][10], stk[N];

struct DLX {
  static constexpr int MAXSIZE = 1e5 + 10;
  int n, m, tot, first[MAXSIZE + 10], siz[MAXSIZE + 10];
  int L[MAXSIZE + 10], R[MAXSIZE + 10], U[MAXSIZE + 10], D[MAXSIZE + 10];
  int col[MAXSIZE + 10], row[MAXSIZE + 10];

  void build(const int &r, const int &c) {  // 进行build操作
    n = r, m = c;
    for (int i = 0; i <= c; ++i) {
      L[i] = i - 1, R[i] = i + 1;
      U[i] = D[i] = i;
    }
    L[0] = c, R[c] = 0, tot = c;
    memset(first, 0, sizeof(first));
    memset(siz, 0, sizeof(siz));
  }

  void insert(const int &r, const int &c) {  // 进行insert操作
    col[++tot] = c, row[tot] = r, ++siz[c];
    D[tot] = D[c], U[D[c]] = tot, U[tot] = c, D[c] = tot;
    if (!first[r])
      first[r] = L[tot] = R[tot] = tot;
    else {
      R[tot] = R[first[r]], L[R[first[r]]] = tot;
      L[tot] = first[r], R[first[r]] = tot;
    }
  }

  void remove(const int &c) {  // 进行remove操作
    int i, j;
    L[R[c]] = L[c], R[L[c]] = R[c];
    for (i = D[c]; i != c; i = D[i])
      for (j = R[i]; j != i; j = R[j])
        U[D[j]] = U[j], D[U[j]] = D[j], --siz[col[j]];
  }

  void recover(const int &c) {  // 进行recover操作
```

```
    int i, j;
    for (i = U[c]; i != c; i = U[i])
      for (j = L[i]; j != i; j = L[j]) U[D[j]] = D[U[j]] = j, ++siz[col[j]];
    L[R[c]] = R[L[c]] = c;
  }

  bool dance(int dep) {  // dance
    int i, j, c = R[0];
    if (!R[0]) {
      for (i = 1; i < dep; ++i) {
        int x = (stk[i] - 1) / 9 / 9 + 1;
        int y = (stk[i] - 1) / 9 % 9 + 1;
        int v = (stk[i] - 1) % 9 + 1;
        ans[x][y] = v;
      }
      return true;
    }
    for (i = R[0]; i != 0; i = R[i])
      if (siz[i] < siz[c]) c = i;
    remove(c);
    for (i = D[c]; i != c; i = D[i]) {
      stk[dep] = row[i];
      for (j = R[i]; j != i; j = R[j]) remove(col[j]);
      if (dance(dep + 1)) return true;
      for (j = L[i]; j != i; j = L[j]) recover(col[j]);
    }
    recover(c);
    return false;
  }
} solver;

int GetId(int row, int col, int num) {
  return (row - 1) * 9 * 9 + (col - 1) * 9 + num;
}

void Insert(int row, int col, int num) {
  int dx = (row - 1) / 3 + 1;
  int dy = (col - 1) / 3 + 1;
  int room = (dx - 1) * 3 + dy;
  int id = GetId(row, col, num);
  int f1 = (row - 1) * 9 + num;             // task 1
  int f2 = 81 + (col - 1) * 9 + num;        // task 2
  int f3 = 81 * 2 + (room - 1) * 9 + num;   // task 3
  int f4 = 81 * 3 + (row - 1) * 9 + col;    // task 4
  solver.insert(id, f1);
  solver.insert(id, f2);
  solver.insert(id, f3);
  solver.insert(id, f4);
}

using std::cin;
using std::cout;
```

```
96
97 int main() {
98   cin.tie(nullptr)->sync_with_stdio(false);
99   solver.build(729, 324);
100  for (int i = 1; i <= 9; ++i)
101    for (int j = 1; j <= 9; ++j) {
102      cin >> ans[i][j];
103      for (int v = 1; v <= 9; ++v) {
104        if (ans[i][j] && ans[i][j] != v) continue;
105        Insert(i, j, v);
106      }
107    }
108  solver.dance(1);
109  for (int i = 1; i <= 9; ++i, cout << '\n')
110    for (int j = 1; j <= 9; ++j, cout << ' ') cout << ans[i][j];
111  return 0;
112 }
113
```

例题 2 靶形数独[3]

解题思路

这一题与 数独[4] 的模型构建 **一模一样**，主要区别在于答案的更新.

这一题可以开一个权值数组，每次找到一组数独的解时，

每个位置上的数乘上对应的权值计入答案即可.

参考代码

```
1  #include <algorithm>
2  #include <cctype>
3  #include <cstring>
4  #include <iostream>
5  constexpr int oo = 0x3f3f3f3f;
6  constexpr int N = 1e5 + 10;
7  constexpr int e[] = {6, 6, 6, 6, 6, 6, 6,  6, 6, 6, 7, 7, 7, 7, 7, 7, 7,
8                       6, 6, 7, 8, 8, 8, 8,  8, 7, 6, 6, 7, 8, 9, 9, 9, 8,
9                       7, 6, 6, 7, 8, 9, 10, 9, 8, 7, 6, 6, 7, 8, 9, 9, 9,
10                      8, 7, 6, 6, 7, 8, 8,  8, 8, 8, 7, 6, 6, 7, 7, 7, 7,
11                      7, 7, 7, 6, 6, 6, 6,  6, 6, 6, 6, 6, 6};
12 int ans = -oo, a[10][10], stk[N];
13
14 int GetWeight(int row, int col, int num) {  // 求数乘上对应的权值
15   return num * e[(row - 1) * 9 + (col - 1)];
16 }
17
18 struct DLX {
19   static constexpr int MAXSIZE = 1e5 + 10;
20   int n, m, tot, first[MAXSIZE + 10], siz[MAXSIZE + 10];
21   int L[MAXSIZE + 10], R[MAXSIZE + 10], U[MAXSIZE + 10], D[MAXSIZE + 10];
```

```
int col[MAXSIZE + 10], row[MAXSIZE + 10];

void build(const int &r, const int &c) {  // 进行build操作
  n = r, m = c;
  for (int i = 0; i <= c; ++i) {
    L[i] = i - 1, R[i] = i + 1;
    U[i] = D[i] = i;
  }
  L[0] = c, R[c] = 0, tot = c;
  memset(first, 0, sizeof(first));
  memset(siz, 0, sizeof(siz));
}

void insert(const int &r, const int &c) {  // 进行insert操作
  col[++tot] = c, row[tot] = r, ++siz[c];
  D[tot] = D[c], U[D[c]] = tot, U[tot] = c, D[c] = tot;
  if (!first[r])
    first[r] = L[tot] = R[tot] = tot;
  else {
    R[tot] = R[first[r]], L[R[first[r]]] = tot;
    L[tot] = first[r], R[first[r]] = tot;
  }
}

void remove(const int &c) {  // 进行remove操作
  int i, j;
  L[R[c]] = L[c], R[L[c]] = R[c];
  for (i = D[c]; i != c; i = D[i])
    for (j = R[i]; j != i; j = R[j])
      U[D[j]] = U[j], D[U[j]] = D[j], --siz[col[j]];
}

void recover(const int &c) {  // 进行recover操作
  int i, j;
  for (i = U[c]; i != c; i = U[i])
    for (j = L[i]; j != i; j = L[j]) U[D[j]] = D[U[j]] = j, ++siz[col[j]];
  L[R[c]] = R[L[c]] = c;
}

void dance(int dep) {  // dance
  int i, j, c = R[0];
  if (!R[0]) {
    int cur_ans = 0;
    for (i = 1; i < dep; ++i) {
      int cur_row = (stk[i] - 1) / 9 / 9 + 1;
      int cur_col = (stk[i] - 1) / 9 % 9 + 1;
      int cur_num = (stk[i] - 1) % 9 + 1;
      cur_ans += GetWeight(cur_row, cur_col, cur_num);
    }
    ans = std::max(ans, cur_ans);
    return;
  }
```

```
    for (i = R[0]; i != 0; i = R[i])
      if (siz[i] < siz[c]) c = i;
    remove(c);
    for (i = D[c]; i != c; i = D[i]) {
      stk[dep] = row[i];
      for (j = R[i]; j != i; j = R[j]) remove(col[j]);
      dance(dep + 1);
      for (j = L[i]; j != i; j = L[j]) recover(col[j]);
    }
    recover(c);
  }
} solver;

int GetId(int row, int col, int num) {
  return (row - 1) * 9 * 9 + (col - 1) * 9 + num;
}

void Insert(int row, int col, int num) {
  int dx = (row - 1) / 3 + 1;    // r
  int dy = (col - 1) / 3 + 1;    // c
  int room = (dx - 1) * 3 + dy;  // room
  int id = GetId(row, col, num);
  int f1 = (row - 1) * 9 + num;            // task 1
  int f2 = 81 + (col - 1) * 9 + num;       // task 2
  int f3 = 81 * 2 + (room - 1) * 9 + num;  // task 3
  int f4 = 81 * 3 + (row - 1) * 9 + col;   // task 4
  solver.insert(id, f1);
  solver.insert(id, f2);
  solver.insert(id, f3);
  solver.insert(id, f4);
}

using std::cin;
using std::cout;

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  solver.build(729, 324);
  for (int i = 1; i <= 9; ++i)
    for (int j = 1; j <= 9; ++j) {
      cin >> a[i][j];
      for (int v = 1; v <= 9; ++v) {
        if (a[i][j] && v != a[i][j]) continue;
        Insert(i, j, v);
      }
    }
  solver.dance(1);
  cout << (ans == -oo ? -1 : ans);
  return 0;
}

```

例题 3 「NOI2005」智慧珠游戏[5]

解题思路

定义：题中给我们的智慧珠的形态，称为这个智慧珠的标准形态.

显然，我们可以通过改变两个参数 d（表示顺时针旋转 90° 的次数）和 f（是否水平翻转）来改变这个智慧珠的形态.

仍然，我们先考虑决策是什么.

在这一题中，每一个决策可以用形如 (v,d,f,i) 的有序五元组表示.

表示第 i 个智慧珠的标准形态的左上角的位置，序号为 v，经过了 d 次顺时针转 90°.

巧合的是，我们可以令 $f=1$ 时不水平翻转，$f=-1$ 时水平翻转，从而达到简化代码的目的.

因此有 $55\times 4\times 2\times 12=5280$ 行.

需要注意的是，因为一些不合法的填充，如 $(1,0,1,4)$，

所以 **在实际操作中，空的智慧珠棋盘也只需要建出 2730 行.**

再考虑状态是什么.

这一题的状态比较简单.

我们思考一下，(v,d,f,i) 这个决策会造成什么影响.

1. 某些格子被占了（用 55 列表示）;
2. 第 i 个智慧珠被用了（用 12 列表示）.

因此有 $55+12=67$ 列，共 $5280\times(5+1)=31680$ 个 1.

至此，我们成功地将智慧珠游戏转化成了一个 **有 5280 行，67 列，共 31680 个 1** 的精确覆盖问题.

参考代码

```cpp
#include <cstring>
#include <iostream>
#include <string>
int numcol, numrow;
int dfn[3000], tx[2], nxt[2], num[50][50], vis[50];
std::string ans[50];
constexpr int f[2] = {-1, 1};
constexpr int table[12][5][2] = {
    // directions of shapes
    {{0, 0}, {1, 0}, {0, 1}},                   // A
    {{0, 0}, {0, 1}, {0, 2}, {0, 3}},           // B
    {{0, 0}, {1, 0}, {0, 1}, {0, 2}},           // C
    {{0, 0}, {1, 0}, {0, 1}, {1, 1}},           // D
    {{0, 0}, {1, 0}, {2, 0}, {2, 1}, {2, 2}},   // E
    {{0, 0}, {0, 1}, {1, 1}, {0, 2}, {0, 3}},   // F
    {{0, 0}, {1, 0}, {0, 1}, {0, 2}, {1, 2}},   // G
    {{0, 0}, {1, 0}, {0, 1}, {1, 1}, {0, 2}},   // H
    {{0, 0}, {0, 1}, {0, 2}, {1, 2}, {1, 3}},   // I
    {{0, 0}, {-1, 1}, {0, 1}, {1, 1}, {0, 2}},  // J
    {{0, 0}, {1, 0}, {1, 1}, {2, 1}, {2, 2}},   // K
    {{0, 0}, {1, 0}, {0, 1}, {0, 2}, {0, 3}},   // L
};
```

```
constexpr int len[12] = {3, 4, 4, 4, 5, 5, 5, 5, 5, 5, 5, 5};
constexpr int getx[] = {
    0,  1,  2,  2,  3,  3,  3,  4,  4,  4,  4,  5,  5,  5,  5,  5,  6,
    6,  6,  6,  6,  6,  7,  7,  7,  7,  7,  7,  7,  8,  8,  8,  8,  8,
    8,  8,  8,  9,  9,  9,  9,  9,  9,  9,  9,  9,  10, 10, 10, 10, 10,
    10, 10, 10, 10, 10, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 12,
    12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 13, 13, 13, 13, 13, 13,
    13, 13, 13, 13, 13, 13, 13, 14, 14, 14, 14, 14, 14, 14, 14, 14};
constexpr int gety[] = {
    0, 1, 1,  2,  1,  2,  3,  1,  2, 3, 4, 1, 2, 3,  4,  5,  1, 2, 3, 4, 5,
    6, 1, 2,  3,  4,  5,  6,  7,  1, 2, 3, 4, 5, 6,  7,  8,  1, 2, 3, 4, 5,
    6, 7, 8,  9,  1,  2,  3,  4,  5, 6, 7, 8, 9, 10, 1,  2,  3, 4, 5, 6, 7,
    8, 9, 10, 11, 1,  2,  3,  4,  5, 6, 7, 8, 9, 10, 11, 12, 1, 2, 3, 4, 5,
    6, 7, 8,  9,  10, 11, 12, 13, 1, 2, 3, 4, 5, 6,  7,  8,  9};

struct DLX {
  static constexpr int MS = 1e5 + 10;
  int n, m, tot, first[MS], siz[MS];
  int L[MS], R[MS], U[MS], D[MS];
  int col[MS], row[MS];

  void build(const int &r, const int &c) {
    n = r, m = c;
    for (int i = 0; i <= c; ++i) {
      L[i] = i - 1, R[i] = i + 1;
      U[i] = D[i] = i;
    }
    L[0] = c, R[c] = 0, tot = c;
    memset(first, 0, sizeof(first));
    memset(siz, 0, sizeof(siz));
  }

  void insert(const int &r, const int &c) {  // insert
    col[++tot] = c, row[tot] = r, ++siz[c];
    D[tot] = D[c], U[D[c]] = tot, U[tot] = c, D[c] = tot;
    if (!first[r])
      first[r] = L[tot] = R[tot] = tot;
    else
      R[tot] = R[first[r]], L[R[first[r]]] = tot, L[tot] = first[r],
      R[first[r]] = tot;  // !
  }

  void remove(const int &c) {  // remove
    int i, j;
    L[R[c]] = L[c], R[L[c]] = R[c];
    for (i = D[c]; i != c; i = D[i])
      for (j = R[i]; j != i; j = R[j])
        U[D[j]] = U[j], D[U[j]] = D[j], --siz[col[j]];
  }

  void recover(const int &c) {  // recover
    int i, j;
```

```
    for (i = U[c]; i != c; i = U[i])
      for (j = L[i]; j != i; j = L[j]) U[D[j]] = D[U[j]] = j, ++siz[col[j]];
    L[R[c]] = R[L[c]] = c;
  }

  bool dance() {  // dance
    if (!R[0]) return true;
    int i, j, c = R[0];
    for (i = R[0]; i != 0; i = R[i])
      if (siz[i] < siz[c]) c = i;
    remove(c);
    for (i = D[c]; i != c; i = D[i]) {
      if (col[i] <= 55) ans[getx[col[i]]][gety[col[i]]] = dfn[row[i]] + 'A';
      for (j = R[i]; j != i; j = R[j]) {
        remove(col[j]);
        if (col[j] <= 55) ans[getx[col[j]]][gety[col[j]]] = dfn[row[j]]
+ 'A';
      }
      if (dance()) return true;
      for (j = L[i]; j != i; j = L[j]) recover(col[j]);
    }
    recover(c);
    return false;
  }
} solver;

using std::cin;
using std::cout;

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  for (int i = 1; i <= 10; ++i) {
    cin >> ans[i];
    ans[i] = " " + ans[i];
  }
  for (int i = 1; i <= 10; ++i)
    for (int j = 1; j <= i; ++j) {
      if (ans[i][j] != '.') vis[ans[i][j] - 'A'] = 1;
      num[i][j] = ++numcol;
    }
  solver.build(2730, numcol + 12);
  /*******build*******/
  for (int id = 0, op; id < 12; ++id) {  // every block
    for (++numcol, op = 0; op <= 1; ++op) {
      for (int dx = 0; dx <= 1; ++dx) {
        for (int dy = 0; dy <= 1; ++dy) {
          for (tx[0] = 1; tx[0] <= 10; ++tx[0]) {
            for (tx[1] = 1; tx[1] <= tx[0]; ++tx[1]) {
              bool flag = true;
              // Check if out of bound.
              for (int k = 0; k < len[id]; ++k) {
                nxt[op] = tx[op] + f[dx] * table[id][k][0];
```

```
126                 nxt[op ^ 1] = tx[op ^ 1] + f[dy] * table[id][k][1];
127                 if (nxt[1] < 1 || nxt[0] < nxt[1] || nxt[0] > 10) {
128                   flag = false;
129                   break;
130                 }
131               }
132               if (!flag) continue;
133               // Check if illegal.
134               for (int k = 0; k < len[id]; ++k) {
135                 nxt[op] = tx[op] + f[dx] * table[id][k][0];
136                 nxt[op ^ 1] = tx[op ^ 1] + f[dy] * table[id][k][1];
137                 if (vis[id]) {
138                   if (ans[nxt[0]][nxt[1]] != id + 'A') {
139                     flag = false;
140                     break;
141                   }
142                 } else if (ans[nxt[0]][nxt[1]] != '.') {
143                   flag = false;
144                   break;
145                 }
146               }
147               if (!flag) continue;
148               // Try to insert.
149               dfn[++numrow] = id;
150               solver.insert(numrow, numcol);
151               for (int k = 0; k < len[id]; ++k) {
152                 nxt[op] = tx[op] + f[dx] * table[id][k][0];
153                 nxt[op ^ 1] = tx[op ^ 1] + f[dy] * table[id][k][1];
154                 solver.insert(numrow, num[nxt[0]][nxt[1]]);
155               }
156             }
157           }
158         }
159       }
160     }
161   }
162   /********end********/
163   if (!solver.dance())
164     cout << "No solution\n";
165   else
166     for (int i = 1; i <= 10; ++i, cout << '\n')
167       for (int j = 1; j <= i; ++j) cout << ans[i][j];
168   return 0;
169 }
170
```

习题

- SUDOKU - Sudoku[6]
- 「kuangbin 带你飞」专题三 Dancing Links[7]

外部链接

- 跳跃的舞者，舞蹈链（Dancing Links）算法——求解精确覆盖问题 - 万仓一黍[8]
- 搜索：DLX 算法 - 静听风吟. [9]
- 《算法竞赛入门经典 - 训练指南》[10]

注释

[1] 模板代码

[2] P1784 数独

[3] 靶形数独

[4] 数独

[5] 「NOI2005」智慧珠游戏

[6] SUDOKU - Sudoku

[7]「kuangbin 带你飞」专题三 Dancing Links

[8] 跳跃的舞者，舞蹈链（Dancing Links）算法——求解精确覆盖问题 - 万仓一黍

[9] 搜索：DLX 算法 - 静听风吟.

[10] 《算法竞赛入门经典 - 训练指南》

6.11 Alpha–Beta 剪枝

本页面将简要介绍 Minimax 算法和 Alpha–Beta 剪枝.

Minimax 算法

Minimax 算法又叫极小化极大算法，是一种最小化最差（即最大损失）情境下的潜在损失的算法.

过程

在局面确定的双人零和对弈中，常需要进行对抗搜索，构建一棵每个节点都为一个确定状态的搜索树. 奇数层轮到己方行动，偶数层轮到对方行动. 搜索树上每个叶子节点都会被赋予一个估值，估值越大代表我方赢面越大. 我方追求更大的赢面，而对方会设法降低我方的赢面；体现在搜索树上就是，奇数层节点（我方节点）总是会选择赢面最大的子节点状态，而偶数层节点（对方节点）总是会选择（我方）赢面最小的子节点状态.

Minimax 算法中，会从上到下遍历搜索树，回溯时利用子树信息更新答案，最后得到根节点的值——这就是我方在双方都采取最优策略下能获得的最大分数.

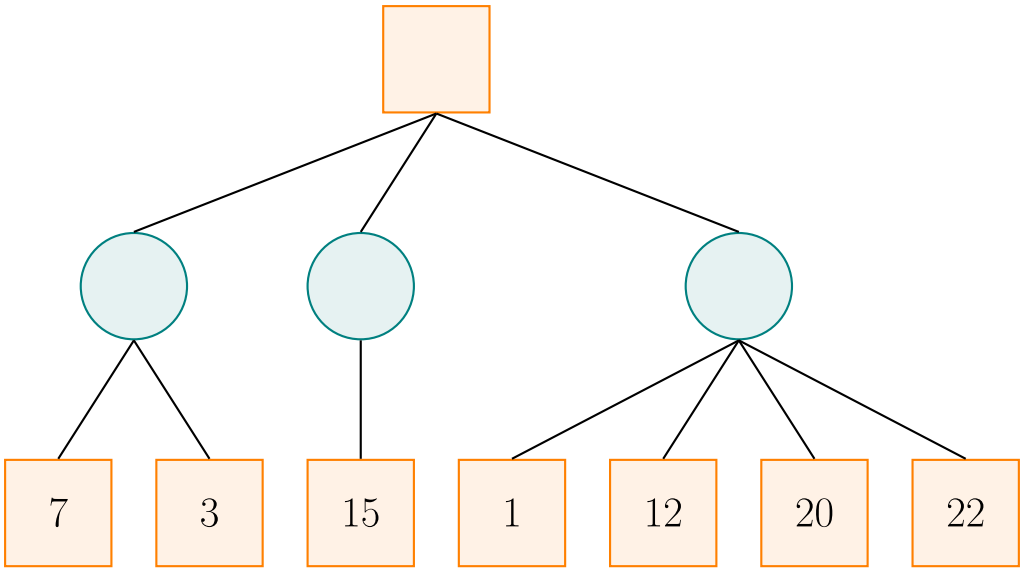
示例

来看一个简单的例子.

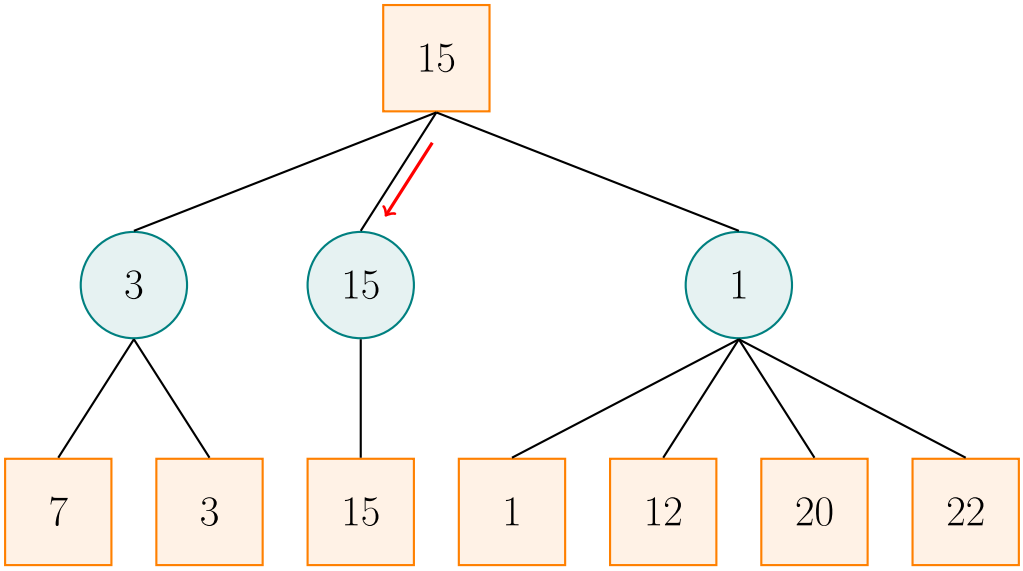
称我方为 MAX，对方为 MIN，图示如下：



例如，对于如下的局势，假设从左往右搜索，根节点的数值为我方赢面：



我方应选择中间的路线．因为，如果选择左边的路线，最差的赢面是 3；如果选择中间的路线，最差的赢面是 15；如果选择右边的路线，最差的赢面是 1．虽然选择右边的路线可能有 22 的赢面，但足够理性的对方将会使我方只有 1 的赢面．那么，经过权衡，显然选择中间的路线更优．



实际上，在看右边的路线时，当发现赢面可能为 1 后就不必再去看赢面为 12、20、22 的分支了．因为相较于左侧两条路线的赢面，已经可以确定右边的路线不是最好的．

朴素的 Minimax 算法常常需要构建一棵庞大的搜索树，时间和空间复杂度都将不能承受．而 Alpha–Beta 剪枝就是利用搜索树每个节点双方分数的上下界来对 Minimax 进行剪枝优化的一种方法．

需要注意的是，对于不同的问题，搜索树每个节点上的值有着不同的含义，它可以是估值、分数、赢的概率等等．为方便起见，下文统一用分数来称呼．

Alpha–Beta 剪枝

Alpha–Beta 剪枝是针对 Minimax 算法的搜索剪枝．

过程

Minimax 算法中，若已知某节点的所有子节点的分数，则可以算出该节点的分数：对于 MAX 节点，取最大分数；对于 MIN 节点，取最小分数．

在搜索进行到某节点但尚未完成时，虽然不能算出该节点的分数，但是可以算出**目前已经搜索过的节点中**，双方分数的取值范围．搜索时，维护两个变量 α 和 β，分别表示局面进行到该节点时，**考虑所有已经搜索过的节点**，Alpha 玩家（即寻求最大分数的一方）和 Beta 玩家（即寻求最小分数的一方）能够保证取得的分数的下界和上界．

Alpha–Beta 剪枝的剪枝策略依赖于搜索当前节点时 α 和 β 的取值．如果当前节点是 MAX 节点，那么，Alpha 玩家可以继续搜索它的子节点来提高分数下界 α．但是，如果某次搜索后已经有 $\alpha \geq \beta$ 了，那么继续搜索这个节点就不会影响对弈的结果：只要到达该节点处，Alpha 玩家就能够保证分数至少是 α；可是 Beta 玩家已经知道存在一种（偏离当前路径的）策略，能够保证分数不超过 $\beta \leq \alpha$，那么，Beta 玩家没有必要让局面发展到**当前节点**处．同理，如果当前节点是 MIN 节点，且搜索它的某个子节点后已经发现该节点处有 $\beta \leq \alpha$ 成立，那么，同样无需继续搜索其他子节点，因为 Alpha 玩家没有必要让局面进入**当前节点**．总结两种情形可以发现：当 $\alpha \geq \beta$ 时，该节点剩余的分支就不必继续搜索了（也就是可以进行剪枝了）．注意，当 $\alpha = \beta$ 时，也可以剪枝，这是因为对方已经有一种能够保证分数对其不劣于 α（即 β）的备选方案，继续搜索剩余分支不会改变上层节点的结果．

搜索过程中，无需维护节点分数，只需要维护 α 和 β 即可．初始时，令 $\alpha = -\infty$，$\beta = +\infty$．向下搜索时，需要一并下传 α 和 β 的信息，以记录两名玩家的备选方案．

搜索完子节点时，需要更新当前节点处的信息．不妨假设当前节点 X 是 MAX 节点，且刚刚搜索完它的子节点 Y．那么，节点 X 处的 β 值不会改变，只有 α 值需要与子节点 Y 的分数取最大值．如果子节点 Y 是叶子节点，直接用子节点 Y 的分数更新当前节点 X 处的 α 值；否则，只需要用子节点 Y 的 β 值更新当前节点 X 的 α 值．此时，有三种可能性：

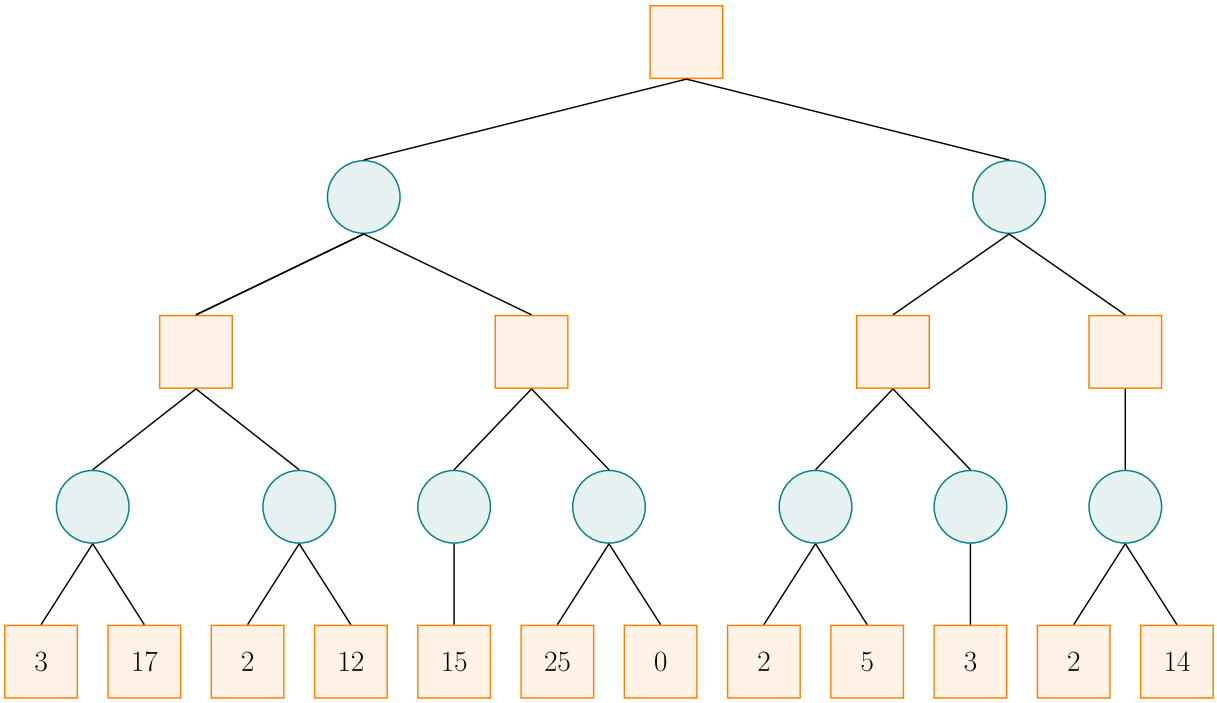
1. 子节点 Y 的 β 值严格位于节点 X 的 α 值和 β 值之间．因为子节点 Y 继承了节点 X 的 α 值且不会更新它，所以，搜索完子节点 Y 后仍然有 $\beta > \alpha$，就说明搜索子节点 Y 时没有发生剪枝．子节点 Y 最终的 β 值，就等于它继承的节点 X 的 β 值和它（指子节点 Y）的所有子节点的分数中，最小的那个．既然这个最小值严格小于节点 X 的 β 值，就说明它一定是子节点 Y 的所有子节点的分数最小值．因此，作为 MIN 节点，子节点 Y 的分数就是这个 β 值．用它更新节点 X 的 α 值是合理的．
2. 子节点 Y 的 β 值就等于节点 X 的 β 值．如上文所述，这说明子节点 Y 的所有子节点的分数均不小于节点 X 的 β 值．这进一步说明 Beta 玩家没有必要让局面进入节点 X：因为 Alpha 玩家只要选择了子节点 Y，Beta 玩家就不能取得比 β 更低的分数．因此，此时使用子节点 Y 的 β 值更新节点 X 的 α 值，是为了使得节点 X 处 $\alpha = \beta$，以触发剪枝条件．它的效果与使用 Y 处实际分数——一个大于等于节点 X 处 β 值的数字——更新节点 X 的 α 值的效果是一样的．
3. 子节点 Y 的 β 值小于等于节点 X 的 α 值．此时，子节点 Y 触发了剪枝条件，它的实际分数不会超过子节点 Y 的 β 值，更不会超过节点 X 的 α 值．用子节点 Y 的实际分数更新节点 X 的 α 值不会改变 α 值．这与使用子节点 Y 的 β 值更新节点 X 的 α 值的效果是一样的．

这一分析说明，当某个子节点搜索完成后，只有处于第一种情形时，α（或 β）才准确记录了这个子节点作为一个 MAX 节点（或 MIN 节点）的实际分数．对于其他情形，虽然它未必是准确的分数，但是它提供的信息足以保证剪枝的正确进行，从而不影响根节点处的分数记录．

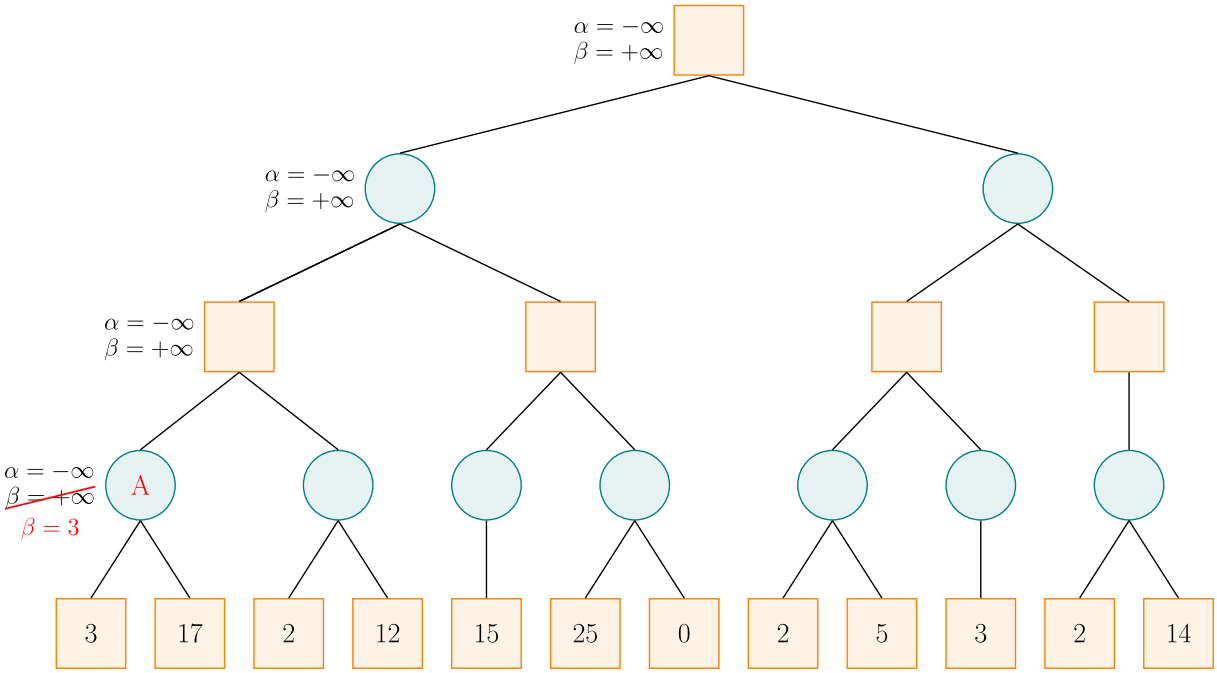
示例

本节通过分析一个例子，来展示如何在搜索过程中更新各个节点处的 α 和 β 值．过程中，也一并计算了所涉及的节点处的分数．由此，就可以观察每个节点处的实际分数与所记录的 α 和 β 值的关系．但应注意，实现这一算法时，并不会计算这些节点的实际分数．

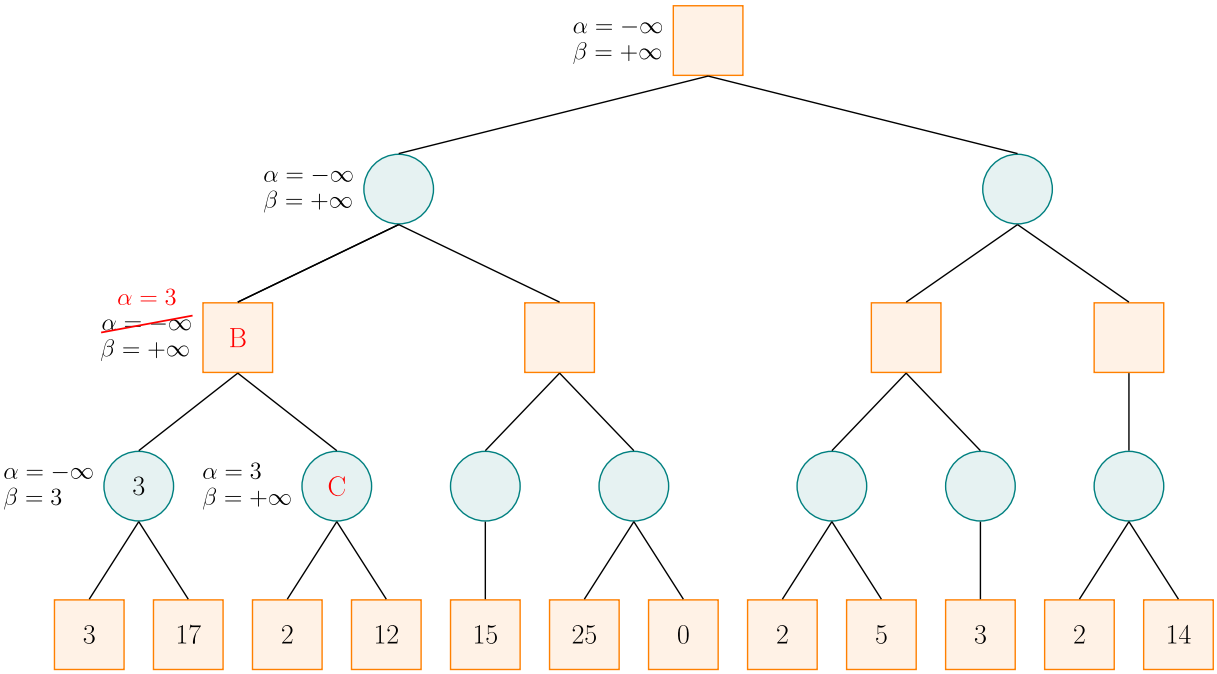
对于如下的局势，假设从左往右搜索：



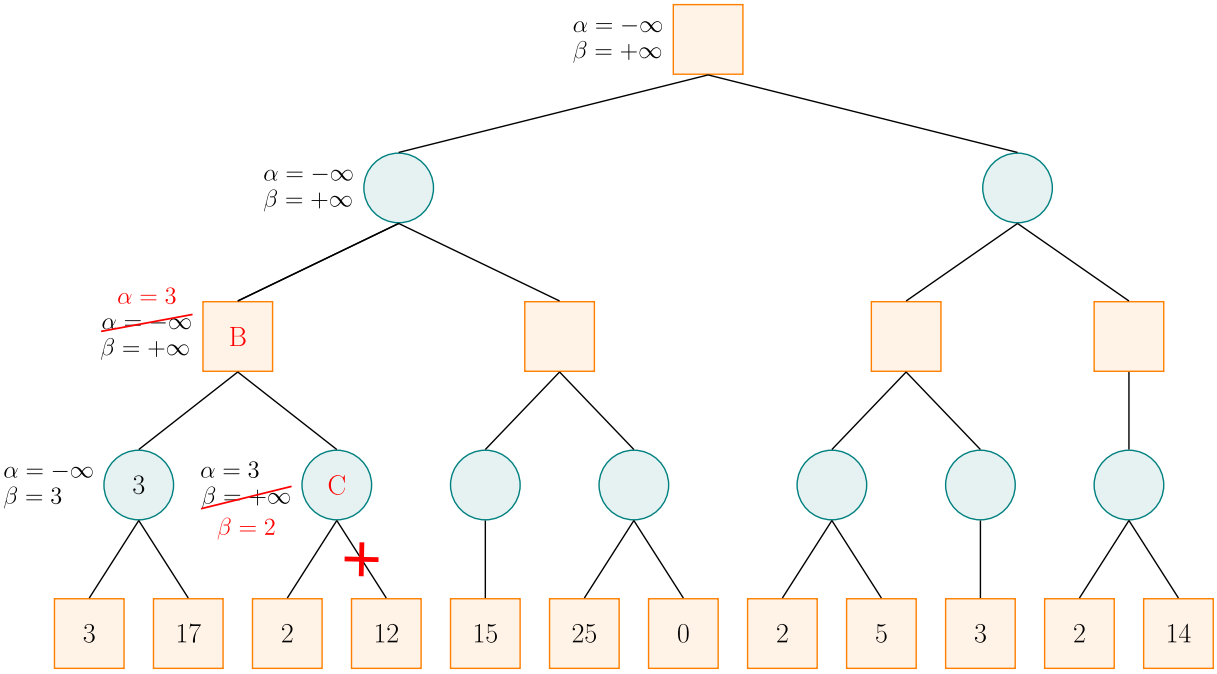
初始化时，令 $\alpha = -\infty$，$\beta = +\infty$，并将这一信息沿着搜索路径下传.



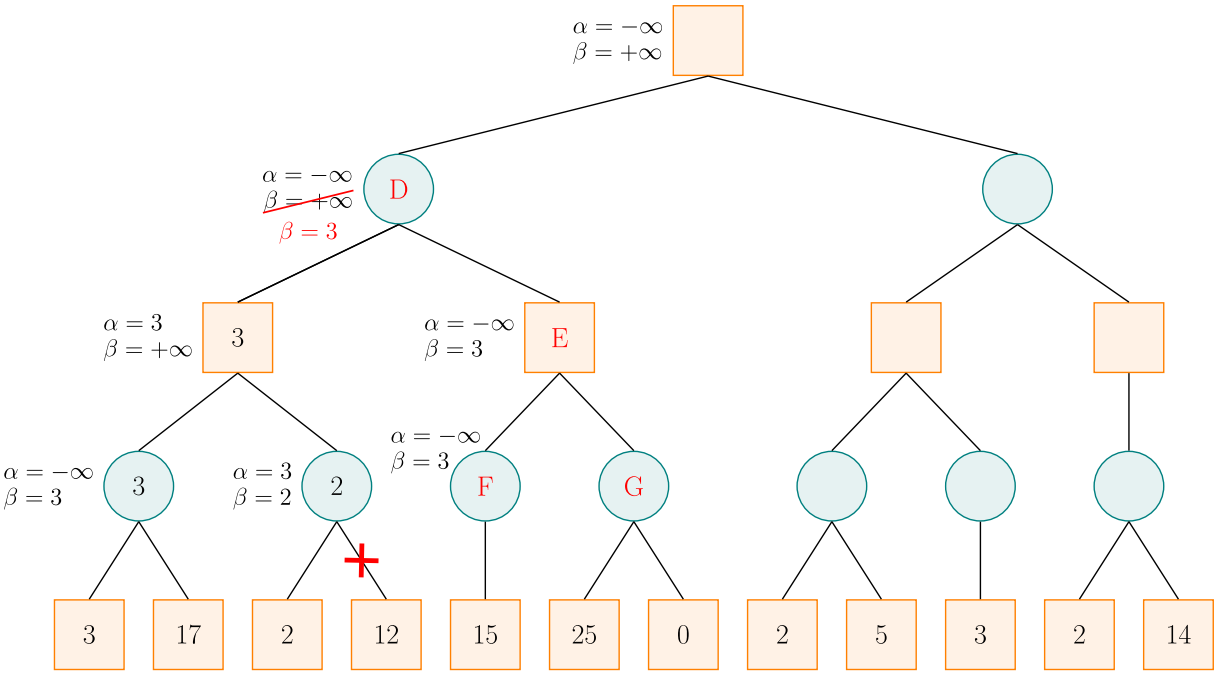
搜索到节点 A 时，由于左子节点的分数为 3，而节点 A 是 MIN 节点，试图找分数小的走法，于是将 β 值修改为 3，这是因为 3 小于当前的 β 值（$\beta = +\infty$）. 然后节点 A 的右子节点的分数为 17，此时不修改节点 A 的 β 值，这是因为 17 大于当前的 β 值（$\beta = 3$）. 此时，节点 A 的所有子节点已搜索完毕，即可计算出节点 A 的分数为 3，这与该节点处记录的 β 值一致（前文的情形 1）.



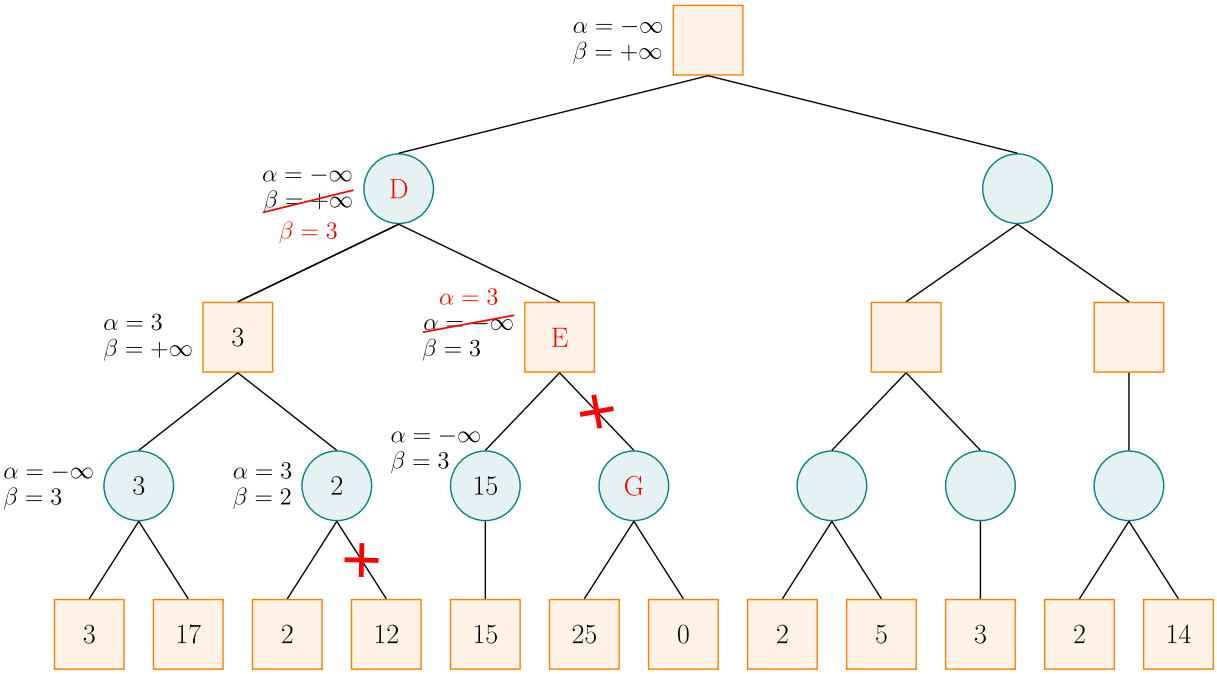
节点 A 是节点 B 的子节点，计算出节点 A 的分数后，可以更新节点 B 的 α 和 β 值. 由于节点 B 是 MAX 节点，试图找分数大的走法，于是将 α 值修改为 3，这是因为子节点 A 处的 β 值（$\beta = 3$）大于当前的 α 值（$\alpha = -\infty$）. 之后，搜索节点 B 的右子节点 C，并将节点 B 的 α 和 β 值传递给节点 C.



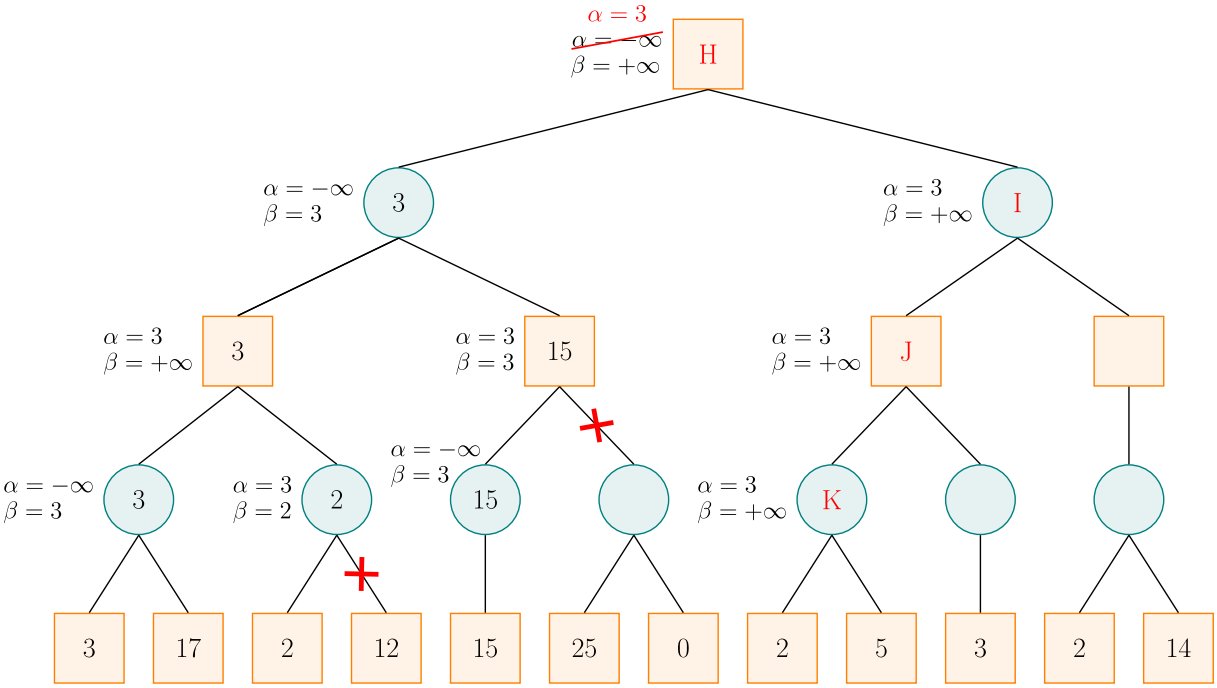
对于节点 C，由于左子节点的分数为 2，而节点 C 是 MIN 节点，于是将 β 值修改为 2. 此时 $\alpha \geq \beta$，故节点 C 的剩余子节点就不必搜索了，因为可以确定，Alpha 玩家不会允许局面发展到节点 C. 此时，节点 C 是 MIN 节点，它的分数就是 2，不超过记录的 β 值（前文的情形 3）. 由于节点 B 的所有子节点搜索完毕，即可计算出节点 B 的分数为 3，与记录的 α 值相同（前文的情形 1）.



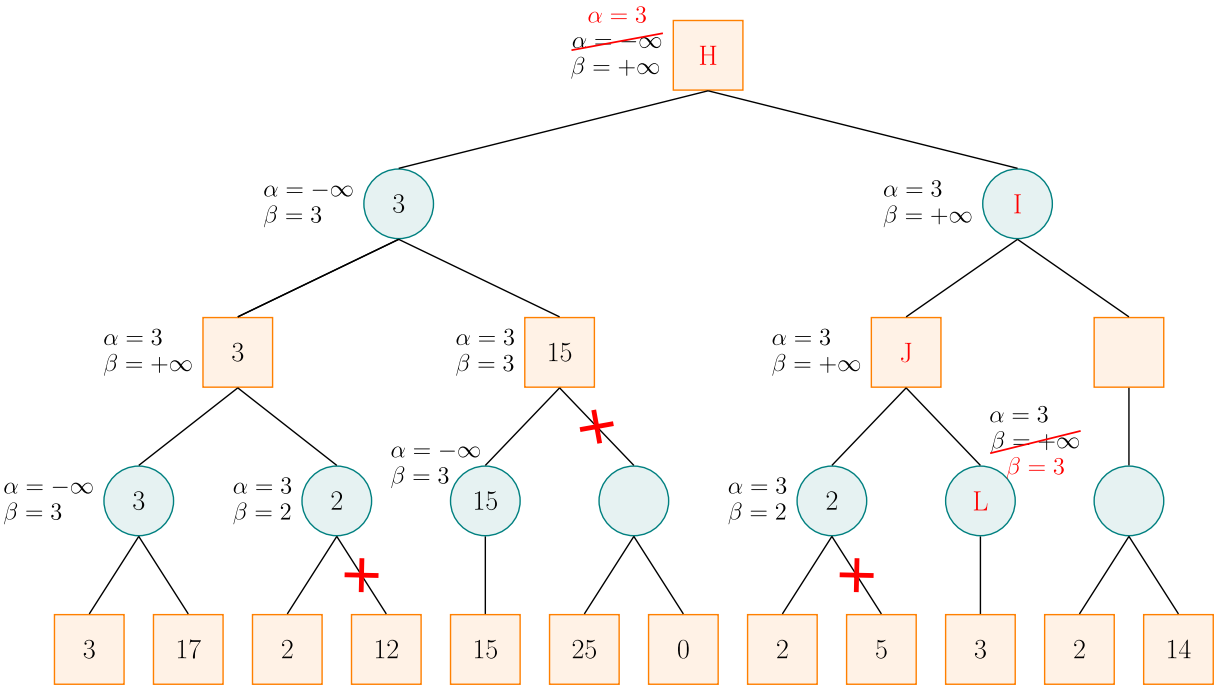
计算出节点 B 的分数后，节点 B 是节点 D 的一个子节点，故可以更新节点 D 的 α 和 β 值．由于节点 D 是 MIN 节点，于是将 β 值修改为 3．然后节点 D 将 α 和 β 值传递给节点 E，节点 E 又传递给节点 F．对于节点 F，它只有一个分数为 15 的子节点，由于 15 大于当前的 β 值，而节点 F 为 MIN 节点，所以不更新其 β 值，然后可以计算出节点 F 的分数为 15，大于记录的 β 值（前文的情形 2）．



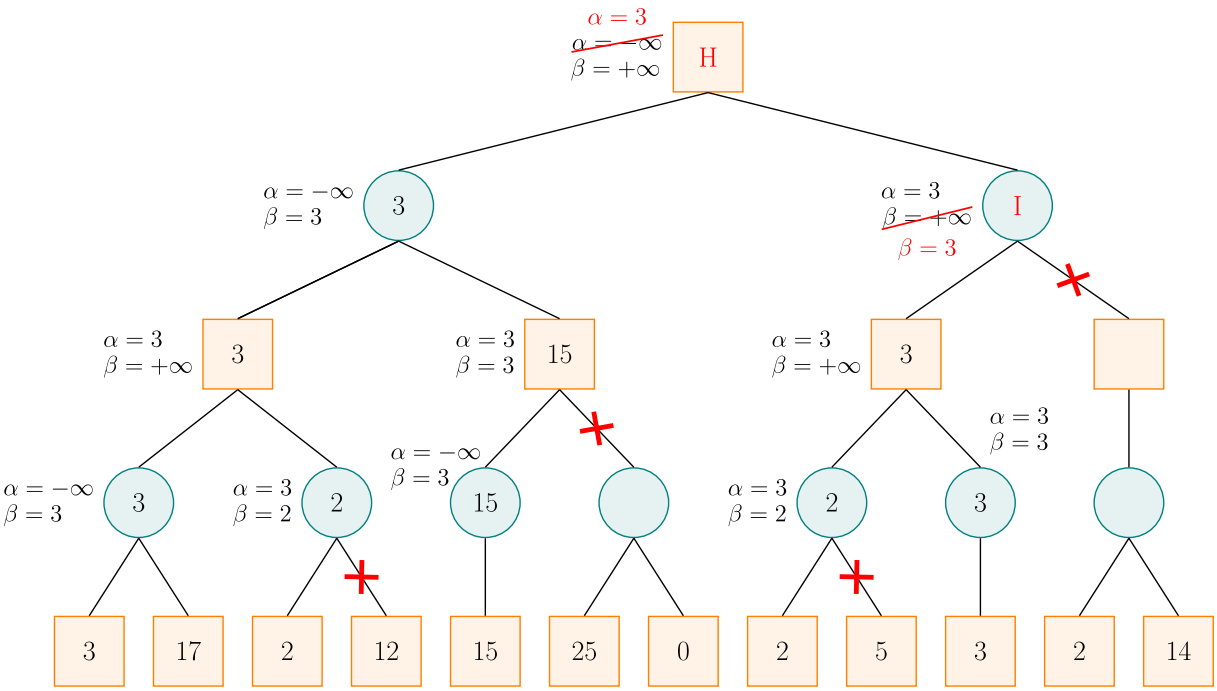
计算出节点 F 的分数后，节点 F 是节点 E 的一个子节点，故可以更新节点 E 的 α 和 β 值．节点 E 是 MAX 节点，更新 α 值，此时 $\alpha \geq \beta$，故可以剪去节点 E 的余下分支（即节点 G）．然后，节点 E 是 MAX 节点，将节点 E 的分数设为 15，严格大于记录的 α 值（前文的情形 3）．利用节点 E 的 α 值更新节点 D 的 β 值，仍然是 3．此时，节点 D 的所有子节点搜索完毕，即可计算出节点 D 的分数为 3，等于记录的 β 值（前文的情形 1）．



计算出节点 D 的分数后，节点 D 是节点 H 的一个子节点，故可以更新节点 H 的 α 和 β 值．节点 H 是 MAX 节点，更新 α．然后，按搜索顺序，将节点 H 的 α 和 β 值依次传递给节点 I、J、K．对于节点 K，其左子节点的分数为 2，而节点 K 是 MIN 节点，更新 β，此时 $\alpha \geq \beta$，故可以剪去节点 K 的余下分支．然后，将节点 K 的分数设为 2，小于等于记录的 β 值（前文的情形 3）．



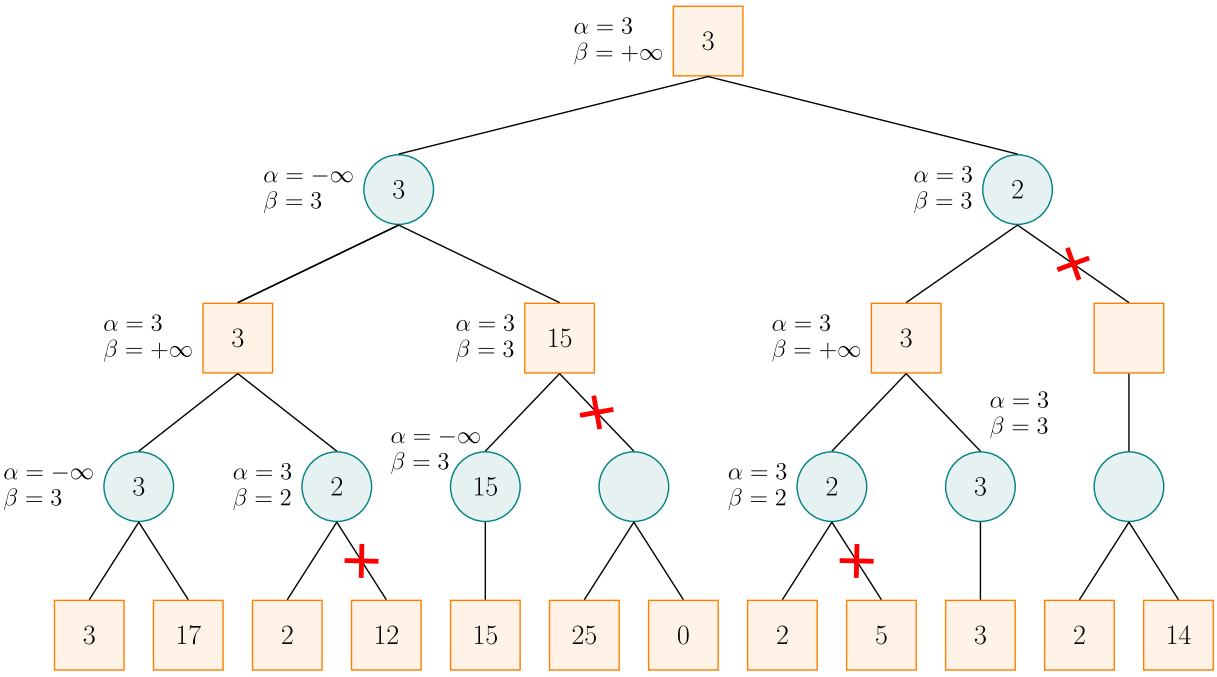
计算出节点 K 的分数后，节点 K 是节点 J 的一个子节点，故可以更新节点 J 的 α 和 β 值．节点 J 是 MAX 节点，更新 α，但是，由于节点 K 的分数小于 α，所以节点 J 的 α 值维持 3 不变．然后，将节点 J 的 α 和 β 值传递给节点 L．由于节点 L 是 MIN 节点，更新 $\beta = 3$，此时 $\alpha \geq \beta$，故可以剪去节点 L 的余下分支．由于节点 L 没有余下分支，所以此处并没有实际剪枝．然后，将节点 L 的分数设为 3，它小于等于记录的 β 值（前文的情形 3）．



计算出节点 L 的分数后，节点 L 是节点 J 的一个子节点，故可以更新节点 J 的 α 和 β 值．节点 J 是 MAX 节点，更新 α，但是，由于节点 L 的分数小于等于 α，所以节点 J 的 α 值维持 3 不变．此时，节点 J 的所有子节点搜索完毕，即可计算出节点 J 的分数为 3，它等于记录的 α 值（前文的情形 2）．

计算出节点 J 的分数后，节点 J 是节点 I 的一个子节点，故可以更新节点 I 的 α 和 β 值．节点 I 是 MIN 节点，更新 β，此时 $\alpha \geq \beta$，故可以剪去节点 I 的余下分支．值得注意的是，由于右子节点的存在，节点 I 的实际分数是 2，小于记录的 β 值（前文的情形 3）．

计算出节点 I 的分数后，节点 I 是节点 H 的一个子节点，故可以更新节点 H 的 α 和 β 值．节点 H 是 MAX 节点，更新 α，但是，由于节点 I 的分数小于等于 α，所以节点 H 的 α 值维持 3 不变．此时，节点 H 的所有子节点搜索完毕，即可计算出节点 H 的分数为 3，它等于记录的 α 值（前文的情形 1）．



这就是最终结果．

实现

参考代码

```
int alpha_beta(int u, int alph, int beta, bool is_max) {
  if (!son_num[u]) return val[u];
  if (is_max) {
    for (int i = 0; i < son_num[u]; ++i) {
      int d = son[u][i];
      alph = max(alph, alpha_beta(d, alph, beta, !is_max));
      if (alph >= beta) break;
    }
    return alph;
  } else {
    for (int i = 0; i < son_num[u]; ++i) {
      int d = son[u][i];
      beta = min(beta, alpha_beta(d, alph, beta, !is_max));
      if (alph >= beta) break;
    }
    return beta;
  }
}
```

参考资料与注释

- Minimax Algorithm - Wikipedia[1]
- Alpha–beta pruning - Wikipedia[2]

本文部分引用自博文 详解 Minimax 算法与 α-β 剪枝_文剑木然[3]，遵循 CC 4.0 BY-SA 版权协议．内容有改动．

外部参考资料

[1] Minimax Algorithm - Wikipedia

[2] Alpha–beta pruning - Wikipedia

[3] 详解 Minimax 算法与 α-β 剪枝_文剑木然

6.12 优化

Authors: CBW2007, ChungZH, Marcythm, abc1763613206, Ir1d

前言

DFS（深度优先搜索）是一种常见的算法，大部分的题目都可以用 DFS 解决，但是大部分情况下，这都是骗分算法，很少会有爆搜为正解的题目．因为 DFS 的时间复杂度特别高．（没学过 DFS 的请自行补上这一课）

既然不能成为正解，那就多骗一点分吧．那么这一篇文章将介绍一些实用的优化算法（俗称「剪枝」）．

先来一段深搜模板，之后的模板将在此基础上进行修改．

```
int ans = 最坏情况, now;  // now 为当前答案

void dfs(传入数值) {
  if (到达目的地) ans = 从当前解与已有解中选最优;
  for (遍历所有可能性)
```

```
    if (可行) {
      进行操作;
      dfs(缩小规模);
      撤回操作;
    }
}
```

其中的 ans 可以是解的记录，那么从当前解与已有解中选最优就变成了输出解.

剪枝方法

最常用的剪枝有三种，记忆化搜索、最优性剪枝、可行性剪枝.

记忆化搜索

因为在搜索中，相同的传入值往往会带来相同的解，那我们就可以用数组来记忆，详见 记忆化搜索→7.3@p720.

模板：

```
int g[MAXN];  // 定义记忆化数组
int ans = 最坏情况, now;

void dfs f(传入数值) {
  if (g[规模] != 无效数值) return;  // 或记录解，视情况而定
  if (到达目的地) ans = 从当前解与已有解中选最优;  // 输出解，视情况而定
  for (遍历所有可能性)
    if (可行) {
      进行操作;
      dfs(缩小规模);
      撤回操作;
    }
}

int main() {
  // ...
  memset(g, 无效数值, sizeof(g));  // 初始化记忆化数组
  // ...
}
```

最优性剪枝

在搜索中导致运行慢的原因还有一种，就是在当前解已经比已有解差时仍然在搜索，那么我们只需要判断一下当前解是否已经差于已有解.

模板：

```
int ans = 最坏情况, now;

void dfs(传入数值) {
  if (now比ans的答案还要差) return;
  if (到达目的地) ans = 从当前解与已有解中选最优;
  for (遍历所有可能性)
    if (可行) {
      进行操作;
      dfs(缩小规模);
      撤回操作;
```

```
    }
}
```

可行性剪枝

在搜索过程中当前解已经不可用了还继续搜索下去也是运行慢的原因.

模板:

```
int ans = 最坏情况, now;

void dfs(传入数值) {
  if (当前解已不可用) return;
  if (到达目的地) ans = 从当前解与已有解中选最优;
  for (遍历所有可能性)
    if (可行) {
      进行操作;
      dfs(缩小规模);
      撤回操作;
    }
}
```

剪枝思路

剪枝思路有很多种，大多需要对于具体问题来分析，在此简要介绍几种常见的剪枝思路.

- 极端法：考虑极端情况，如果最极端（最理想）的情况都无法满足，那么肯定实际情况搜出来的结果不会更优了.
- 调整法：通过对子树的比较剪掉重复子树和明显不是最有「前途」的子树.
- 数学方法：比如在图论中借助连通分量，数论中借助模方程的分析，借助不等式的放缩来估计下界等等.

例题

工作分配问题

有 n（$1 \le n \le 15$）份工作要分配给 n 个人来完成，每个人完成一份．第 i 个人完成第 k 份工作所用的时间为一个正整数 $t_{i,k}$（$1 \le t_{i,k} \le 10^4$），其中 $1 \le i,k \le n$．试确定一个分配方案，使得完成这 n 份工作的时间总和最小.

由于每个人都必须分配到工作，在这里可以建一个二维数组 `time[i][j]`，用以表示 i 个人完成 j 号工作所花费的时间. 给定一个循环，从第 1 个人开始循环分配工作，直到所有人都分配到. 为第 i 个人分配工作时，再循环检查每个工作是否已被分配，没有则分配给 i 个人，否则检查下一个工作. 可以用一个一维数组 `is_working[j]` 来表示第 j 号工作是否已被分配，未分配则 `is_working[j]=0`，否则 `is_working[j]=1`. 利用回溯思想，在工人循环结束后回到上一工人，取消此次分配的工作，而去分配下一工作直到可以分配为止．这样，一直回溯到第 1 个工人后，就能得到所有的可行解.

检查工作分配，其实就是判断取得可行解时的二维数组的第一维下标各不相同并且第二维下标各不相同. 而我们是要得到完成这 n 份工作的最小时间总和，即可行解中时间总和最小的一个，故需要再定义一个全局变量 `cost_time_total_min` 表示目前找到的解中最小的时间总和，初始 `cost_time_total_min` 为 `time[i][i]` 之和，即对角线工作时间相加之和．在所有人分配完工作时，比较 `count` 与 `cost_time_total_min` 的大小，如果 `count` 小于 `cost_time_total_min`，说明找到了一个最优解，此时就把 `count` 赋给 `cost_time_total_min`.

但考虑到算法的效率，这里还有一个剪枝优化的工作可以做. 就是在每次计算局部费用变量 `count` 的值时，如果判断 `count` 已经大于 `cost_time_total_min`，就没必要再往下分配了，因为这时得到的解必然不是最优解.

参考代码

```cpp
#include <iostream>
constexpr int N = 16;
int is_working[N] = {0};  // 某项工作是否被分配
int tm[N][N];             // 完成某项工作所需的时间
int cost_time_total_min;  // 完成 n 份工作的最小时间总和

// i 表示第几个人，count 表示工作费用总和
void work(int i, int count, int n) {
  // 如果 i 超出了所能分配的最大工作件数，表示分配完成，并且 count 比原来
  // cost_time_total_min 花费少，则更新 cost_time_total_min 的值
  if (i > n && count < cost_time_total_min) {
    cost_time_total_min = count;
    return;
  }
  // 回溯思想
  if (count < cost_time_total_min) {
    // j 表示第几件工作
    for (int j = 1; j <= n; j++) {
      // 如果工作未被分配 is_working = 0
      if (is_working[j] == 0) {
        // 分配工作 is_working = 1
        is_working[j] = 1;
        // 工作交给第 i + 1 个人
        work(i + 1, count + tm[i][j], n);
        // 在一轮迭代完成之后，返回到上一个人，要对此次的工作进行重新分配
        // 将 is_working[j] 重设为 0
        is_working[j] = 0;
      }
    }
  }
}

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++) {
    for (int j = 1; j <= n; j++) {
      cin >> tm[i][j];
    }
    cost_time_total_min += tm[i][i];
  }
  work(1, 0, n);
  cout << cost_time_total_min << '\n';
```

```
48   return 0;
49 }
50
```

第七章

动态规划

7.1 动态规划部分简介

本章将介绍动态规划（Dynamic Programming, DP）及其解决的问题、根据其设计的算法及优化.

动态规划是一种通过把原问题分解为相对简单的子问题的方式求解复杂问题的方法.

由于动态规划并不是某种具体的算法，而是一种解决特定问题的方法，因此它会出现在各式各样的数据结构中，与之相关的题目种类也更为繁杂.

在 OI 中，计数等非最优化问题的递推解法也常被不规范地称作 DP，因此本章将它们一并列出. 事实上，动态规划与其它类型的递推的确有很多相似之处，学习时可以注意它们之间的异同.

参考资料

动态规划 - 维基百科，自由的百科全书[1]

外部参考资料

[1] 动态规划 - 维基百科，自由的百科全书

7.2 动态规划基础

> **Authors:** Ir1d, CBW2007, ChungZH, xhn16729, Xeonacid, tptpp, hsfzLZH1, ouuan, Marcythm, HeRaNO, greyqz, Chrogeek, partychicken, zhb2000, xyf007, Persdre, XiaoSuan250, hhc0001, ZhangZhanhaoxiang, Taoran_01

本页面主要介绍了动态规划的基本思想，以及动态规划中状态及状态转移方程的设计思路，帮助各位初学者对动态规划有一个初步的了解.

本部分的其他页面，将介绍各种类型问题中动态规划模型的建立方法，以及一些动态规划的优化技巧.

引入

[IOI1994] 数字三角形[1]

给定一个 r 行的数字三角形（$r \leq 1000$），需要找到一条从最高点到底部任意处结束的路径，使路径经过数字的和最大. 每一步可以走到当前点左下方的点或右下方的点.

```
1         7
2       3   8
3     8   1   0
4   2   7   4   4
5 4   5   2   6   5
```

在上面这个例子中，最优路径是 $7 \to 3 \to 8 \to 7 \to 5$.

最简单粗暴的思路是尝试所有的路径. 因为路径条数是 $O(2^r)$ 级别的，这样的做法无法接受.

注意到这样一个事实，一条最优的路径，它的每一步决策都是最优的.

以例题里提到的最优路径为例，只考虑前四步 $7 \to 3 \to 8 \to 7$，不存在一条从最顶端到 4 行第 2 个数的权值更大的路径.

而对于每一个点，它的下一步决策只有两种：往左下角或者往右下角（如果存在）. 因此只需要记录当前点的最大权值，用这个最大权值执行下一步决策，来更新后续点的最大权值.

这样做还有一个好处：我们成功缩小了问题的规模，将一个问题分成了多个规模更小的问题．要想得到从顶端到第 r 行的最优方案，只需要知道从顶端到第 $r-1$ 行的最优方案的信息就可以了．

这时候还存在一个问题：子问题间重叠的部分会有很多，同一个子问题可能会被重复访问多次，效率还是不高．解决这个问题的方法是把每个子问题的解存储下来，通过记忆化的方式限制访问顺序，确保每个子问题只被访问一次．

上面就是动态规划的一些基本思路．下面将会更系统地介绍动态规划的思想．

动态规划原理

能用动态规划解决的问题，需要满足三个条件：最优子结构，无后效性和子问题重叠．

最优子结构

具有最优子结构也可能是适合用贪心的方法求解．

注意要确保我们考察了最优解中用到的所有子问题．

1. 证明问题最优解的第一个组成部分是做出一个选择；
2. 对于一个给定问题，在其可能的第一步选择中，假定你已经知道哪种选择才会得到最优解．你现在并不关心这种选择具体是如何得到的，只是假定已经知道了这种选择；
3. 给定可获得的最优解的选择后，确定这次选择会产生哪些子问题，以及如何最好地刻画子问题空间；
4. 证明作为构成原问题最优解的组成部分，每个子问题的解就是它本身的最优解．方法是反证法，考虑加入某个子问题的解不是其自身的最优解，那么就可以从原问题的解中用该子问题的最优解替换掉当前的非最优解，从而得到原问题的一个更优的解，从而与原问题最优解的假设矛盾．

要保持子问题空间尽量简单，只在必要时扩展．

最优子结构的不同体现在两个方面：

1. 原问题的最优解中涉及多少个子问题；
2. 确定最优解使用哪些子问题时，需要考察多少种选择．

子问题图中每个定点对应一个子问题，而需要考察的选择对应关联至子问题顶点的边．

无后效性

已经求解的子问题，不会再受到后续决策的影响．

子问题重叠

如果有大量的重叠子问题，我们可以用空间将这些子问题的解存储下来，避免重复求解相同的子问题，从而提升效率．

基本思路

对于一个能用动态规划解决的问题，一般采用如下思路解决：

1. 将原问题划分为若干 **阶段**，每个阶段对应若干个子问题，提取这些子问题的特征（称之为 **状态**）；
2. 寻找每一个状态的可能 **决策**，或者说是各状态间的相互转移方式（用数学的语言描述就是 **状态转移方程**）．
3. 按顺序求解每一个阶段的问题．

如果用图论的思想理解，我们建立一个 有向无环图→11.7@p2727，每个状态对应图上一个节点，决策对应节点间的连边．这样问题就转变为了一个在 DAG 上寻找最长（短）路的问题（参见：DAG 上的 DP→7.6@p735）．

最长公共子序列

最长公共子序列问题

给定一个长度为 n 的序列 A 和一个 长度为 m 的序列 B（$n, m \leq 5000$），求出一个最长的序列，使得该序列既是 A 的子序列，也是 B 的子序列.

子序列的定义可以参考 子序列→8.2@p890. 一个简要的例子：字符串 `abcde` 与字符串 `acde` 的公共子序列有 `a`、`c`、`d`、`e`、`ac`、`ad`、`ae`、`cd`、`ce`、`de`、`acd`、`ade`、`ace`、`cde`、`acde`，最长公共子序列的长度是 4.

设 $f(i, j)$ 表示只考虑 A 的前 i 个元素，B 的前 j 个元素时的最长公共子序列的长度，求这时的最长公共子序列的长度就是 **子问题**. $f(i, j)$ 就是我们所说的 **状态**，则 $f(n, m)$ 是最终要达到的状态，即为所求结果.

对于每个 $f(i, j)$，存在三种决策：如果 $A_i = B_j$，则可以将它接到公共子序列的末尾；另外两种决策分别是跳过 A_i 或者 B_j. 状态转移方程如下：

$$f(i,j) = \begin{cases} f(i-1,j-1)+1 & A_i = B_j \\ \max(f(i-1,j), f(i,j-1)) & A_i \neq B_j \end{cases}$$

可参考 SourceForge 的 LCS 交互网页[2] 来更好地理解 LCS 的实现过程.

参考实现

- C++

```cpp
int n, m, a[MAXN], b[MAXM], f[MAXN][MAXM];

int dp() {
  for (int i = 1; i <= n; i++)
    for (int j = 1; j <= m; j++)
      if (a[i] == b[j])
        f[i][j] = f[i - 1][j - 1] + 1;
      else
        f[i][j] = std::max(f[i - 1][j], f[i][j - 1]);
  return f[n][m];
}

```

- Python

```python
def dp(n, m, a, b):
    f = [[0] * (m + 1) for _ in range(n + 1)]
    for i in range(1, n + 1):
        for j in range(1, m + 1):
            if a[i] == b[j]:
                f[i][j] = f[i - 1][j - 1] + 1
            else:
                f[i][j] = max(f[i - 1][j], f[i][j - 1])
    return f[n][m]


```

该做法的时间复杂度为 $O(nm)$.

另外，本题存在 $O\left(\frac{nm}{w}\right)$ 的算法[1]．有兴趣的同学可以自行探索．

最长不下降子序列

最长不下降子序列问题

给定一个长度为 n 的序列 a（$n \leq 5000$），求出一个最长的 a 的子序列，满足该子序列的后一个元素不小于前一个元素．

算法一

设 $f(i)$ 表示以 a_i 为结尾的最长不下降子序列的长度，则所求为 $\max_{1\leq i\leq n} f(i)$．

计算 $f(i)$ 时，尝试将 a_i 接到其他的最长不下降子序列后面，以更新答案．于是可以写出这样的状态转移方程：$f(i) = \max_{1\leq j<i,\ a_j\leq a_i}(f(j)+1)$．

参考实现

- C++

```cpp
int n, a[MAXN], d[MAXN];

int dp() {
  d[1] = 1;
  int ans = 1;
  for (int i = 2; i <= n; i++) {
    d[i] = 1;
    for (int j = 1; j < i; j++)
      if (a[j] <= a[i]) {
        d[i] = std::max(d[i], d[j] + 1);
        ans = std::max(ans, d[i]);
      }
  }
  return ans;
}

```

- Python

```python
def dp(n, a):
    d = [0] * (n + 1)
    d[1] = 1
    ans = 1
    for i in range(2, n + 1):
        d[i] = 1
        for j in range(1, i):
            if a[j] <= a[i]:
                d[i] = max(d[i], d[j] + 1)
                ans = max(ans, d[i])
    return ans


```

容易发现该算法的时间复杂度为 $O(n^2)$．

[1]位运算求最长公共子序列 - -Wallace- - 博客园[3]

算法二

当 n 的范围扩大到 $n \le 10^5$ 时，第一种做法就不够快了，下面给出了一个 $O(n \log n)$ 的做法.

考虑之前定义的状态 (i, l)，表示序列以第 i 个元素结尾的不下降子序列最长为 l. 不同于以往按固定 i 处理状态的方法，这里直接判断 (i, l) 是否合法：

- 初始状态 $(1, 1)$ 必然合法.
- 对于任意 (i, l)，如果存在 $j < i$ 且 $(j, l-1)$ 合法，同时 $a_j \le a_i$，则 (i, l) 合法.

最终，只需要找到合法状态中 l 最大的 (i, l)，即可得到最长不下降子序列的长度.

设原序列为 $a_1, \cdots, a_n$，定义数组 d，其中第 x 位表示长度为 x 的不下降子序列末尾元素的最小值. 初始时序列为空. 令 i 从 1 到 n 遍历，依次求出前 i 个元素的最长不下降子序列的长度. 对于当前元素 a_i：

- 如果 a_i 大于等于序列 d 中最后一个元素，直接将元素 a_i 插入到序列 d 的末尾.
  - 解释：若 a_i 大于等于当前最长子序列的末尾元素，说明存在一个不下降子序列可以接上 a_i. 不插入将破坏最优性.
- 如果 a_i 严格小于 d 中最后一个元素，找到 **第一个** 大于它的元素，并用 a_i 替换它.
  - 解释：若直接插在末尾，会破坏 d 的单调性；替换操作可以保证每个长度的末尾元素尽可能小，从而为后续元素保留更多可能性.
  - 优化：因为 d 单调不减，可用二分查找直接找到元素的插入位置，将整体复杂度降低到 $O(n \log n)$ 而非暴力查找的 $O(n^2)$.

如果还要输出具体的最长不下降子序列，可以额外维护数组 d'_x，表示长度为 x 的不下降子序列中末尾最小元素的位置（有多个可任选一个）. 具体维护时，只需要在插入元素 a_i 到 d_x 时，同时更新 d'_x 为 i 即可. 同时，需要记录 i 的最优前驱 p_i 为 d'_{x-1}. 最终，从任意最大长度状态出发，沿前驱 p_i 回溯，即可得到完整子序列.

参考实现

- C++

```cpp
int n, a[MAXN], d[MAXN], di[MAXN], pre[MAXN], res[MAXN];

int dp() {
  int ans = 0;
  for (int i = 1; i <= n; ++i) {
    int tmp = std::upper_bound(d, d + ans, a[i]) - d;
    pre[i] = tmp ? di[tmp - 1] : -1;
    d[tmp] = a[i];
    di[tmp] = i;
    if (tmp == ans) ++ans;
  }
  // Construct the subsequence.
  for (int k = ans, i = di[ans - 1]; k; --k) {
    res[k] = a[i];
    i = pre[i];
  }
  return ans;
}

```

- Python

```python
n = 0
a = [0] * MAXN

```

```
d = [0] * MAXN
di = [0] * MAXN
pre = [0] * MAXN
res = [0] * MAXN


def dp():
    ans = 0
    for i in range(1, n + 1):
        tmp = bisect.bisect_right(d, a[i], 0, ans)
        pre[i] = di[tmp - 1] if tmp else -1
        d[tmp] = a[i]
        di[tmp] = i
        if tmp == ans:
            ans += 1

    # Construct the subsequence
    k = ans
    i = di[ans - 1]
    while k:
        res[k] = a[i]
        i = pre[i]
        k -= 1

    return ans


```

该算法的时间复杂度为 $O(n \log n)$. 输出答案的时间复杂度为 $O(ans)$.

注意

对于最长 **上升** 子序列问题，类似地，可以令 d_i 表示所有长度为 i 的最长上升子序列的末尾元素的最小值.

需要注意的是，在步骤 2 中，若 $a_i \leq d_{len}$，由于最长上升子序列中相邻元素不能相等，需要在 d 序列中找到 **第一个 不小于** a_i 的元素，用 a_i 替换之.

在实现上（以 C++ 为例），需要将 `upper_bound` 函数改为 `lower_bound`.

参考资料与注释

- 最长不下降子序列 nlogn 算法详解 - lvmememe - 博客园[4]

[1] [IOI1994] 数字三角形

[2] SourceForge 的 LCS 交互网页

[3] 位运算求最长公共子序列 - -Wallace- - 博客园

[4] 最长不下降子序列 nlogn 算法详解 - lvmememe - 博客园

7.3 记忆化搜索

定义

记忆化搜索是一种通过记录已经遍历过的状态的信息，从而避免对同一状态重复遍历的搜索实现方式.

因为记忆化搜索确保了每个状态只访问一次，它也是一种常见的动态规划实现方式.

引入

[NOIP2005] 采药[1]

山洞里有 M 株不同的草药，采每一株都需要一些时间 t_i，每一株也有它自身的价值 v_i. 给你一段时间 T，在这段时间里，你可以采到一些草药. 让采到的草药的总价值最大.

$1 \le T \le 10^3$，$1 \le t_i, v_i, M \le 100$

朴素的 DFS→6.2@p654 做法

很容易实现这样一个朴素的搜索做法：在搜索时记录下当前准备选第几个物品、剩余的时间是多少、已经获得的价值是多少这三个参数，然后枚举当前物品是否被选，转移到相应的状态.

实现

- C++

```cpp
int n, t;
int tcost[103], mget[103];
int ans = 0;

void dfs(int pos, int tleft, int tans) {
  if (tleft < 0) return;
  if (pos == n + 1) {
    ans = max(ans, tans);
    return;
  }
  dfs(pos + 1, tleft, tans);
  dfs(pos + 1, tleft - tcost[pos], tans + mget[pos]);
}

int main() {
  cin >> t >> n;
  for (int i = 1; i <= n; i++) cin >> tcost[i] >> mget[i];
  dfs(1, t, 0);
  cout << ans << endl;
  return 0;
}
```

- Python

```python
tcost = [0] * 103
mget = [0] * 103
ans = 0

```

```
5
6  def dfs(pos, tleft, tans):
7      global ans
8      if tleft < 0:
9          return
10     if pos == n + 1:
11         ans = max(ans, tans)
12         return
13     dfs(pos + 1, tleft, tans)
14     dfs(pos + 1, tleft - tcost[pos], tans + mget[pos])
15
16
17 t, n = map(lambda x: int(x), input().split())
18 for i in range(1, n + 1):
19     tcost[i], mget[i] = map(lambda x: int(x), input().split())
20 dfs(1, t, 0)
21 print(ans)
```

这种做法的时间复杂度是指数级别的，并不能通过本题.

优化

上面的做法为什么效率低下呢？因为同一个状态会被访问多次.

如果我们每查询完一个状态后将该状态的信息存储下来，再次需要访问这个状态就可以直接使用之前计算得到的信息，从而避免重复计算. 这充分利用了动态规划中很多问题具有大量重叠子问题的特点，属于用空间换时间的「记忆化」思想.

具体到本题上，我们在朴素的 DFS 的基础上，增加一个数组 `mem` 来记录每个 `dfs(pos,tleft)` 的返回值. 刚开始把 `mem` 中每个值都设成 `-1`（代表没求解过）. 每次需要访问一个状态时，如果相应状态的值在 `mem` 中为 `-1`，则递归访问该状态. 否则我们直接使用 `mem` 中已经存储过的值即可.

通过这样的处理，我们确保了每个状态只会被访问一次，因此该算法的时间复杂度为 $O(TM)$.

实现

- C++

```
1  int n, t;
2  int tcost[103], mget[103];
3  int mem[103][1003];
4
5  int dfs(int pos, int tleft) {
6    if (mem[pos][tleft] != -1)
7      return mem[pos][tleft];  // 已经访问过的状态，直接返回之前记录的值
8    if (pos == n + 1) return mem[pos][tleft] = 0;
9    int dfs1, dfs2 = -INF;
10   dfs1 = dfs(pos + 1, tleft);
11   if (tleft >= tcost[pos])
12     dfs2 = dfs(pos + 1, tleft - tcost[pos]) + mget[pos];  // 状态转移
13   return mem[pos][tleft] = max(dfs1, dfs2);  // 最后将当前状态的值存下来
14 }
15
16 int main() {
```

```
  memset(mem, -1, sizeof(mem));
  cin >> t >> n;
  for (int i = 1; i <= n; i++) cin >> tcost[i] >> mget[i];
  cout << dfs(1, t) << endl;
  return 0;
}
```

- Python

```
tcost = [0] * 103
mget = [0] * 103
mem = [[-1 for i in range(1003)] for j in range(103)]


def dfs(pos, tleft):
    if mem[pos][tleft] != -1:
        return mem[pos][tleft]
    if pos == n + 1:
        mem[pos][tleft] = 0
        return mem[pos][tleft]
    dfs1 = dfs2 = -INF
    dfs1 = dfs(pos + 1, tleft)
    if tleft >= tcost[pos]:
        dfs2 = dfs(pos + 1, tleft - tcost[pos]) + mget[pos]
    mem[pos][tleft] = max(dfs1, dfs2)
    return mem[pos][tleft]


t, n = map(lambda x: int(x), input().split())
for i in range(1, n + 1):
    tcost[i], mget[i] = map(lambda x: int(x), input().split())
print(dfs(1, t))
```

与递推的联系与区别

在求解动态规划的问题时，记忆化搜索与递推的代码，在形式上是高度类似的．这是由于它们使用了相同的状态表示方式和类似的状态转移．也正因为如此，一般来说两种实现的时间复杂度是一样的．

下面给出的是递推实现的代码（为了方便对比，没有添加滚动数组优化），通过对比可以发现二者在形式上的类似性．

```
int n, t, w[105], v[105], f[105][1005];

int main() {
  cin >> n >> t;
  for (int i = 1; i <= n; i++) cin >> w[i] >> v[i];
  for (int i = 1; i <= n; i++)
    for (int j = 0; j <= t; j++) {
      f[i][j] = f[i - 1][j];
      if (j >= w[i])
        f[i][j] = max(f[i][j], f[i - 1][j - w[i]] + v[i]);  // 状态转移方程
    }
  cout << f[n][t];
```

```
  return 0;
}
```

在求解动态规划的问题时，记忆化搜索和递推，都确保了同一状态至多只被求解一次．而它们实现这一点的方式则略有不同：递推通过设置明确的访问顺序来避免重复访问，记忆化搜索虽然没有明确规定访问顺序，但通过给已经访问过的状态打标记的方式，也达到了同样的目的．

与递推相比，记忆化搜索因为不用明确规定访问顺序，在实现难度上有时低于递推，且能比较方便地处理边界情况，这是记忆化搜索的一大优势．但与此同时，记忆化搜索难以使用滚动数组等优化，且由于存在递归，运行效率会低于递推．因此应该视题目选择更适合的实现方式．

如何写记忆化搜索

方法一

1. 把这道题的 dp 状态和方程写出来
2. 根据它们写出 dfs 函数
3. 添加记忆化数组

举例：

$dp_i = \max\{dp_j + 1\}$ $(1 \le j < i \wedge a_j < a_i)$（最长上升子序列）

转为

- C++

```cpp
int dfs(int i) {
  if (mem[i] != -1) return mem[i];
  int ret = 1;
  for (int j = 1; j < i; j++)
    if (a[j] < a[i]) ret = max(ret, dfs(j) + 1);
  return mem[i] = ret;
}

int main() {
  memset(mem, -1, sizeof(mem));
  // 读入部分略去
  int ret = 0;
  for (int j = 1; j <= n; j++) {
    ret = max(ret, dfs(j));
  }
  cout << ret << endl;
}
```

- Python

```python
def dfs(i):
    if mem[i] != -1:
        return mem[i]
    ret = 1
    for j in range(1, i):
        if a[j] < a[i]:
            ret = max(ret, dfs(j) + 1)
    mem[i] = ret
    return mem[i]
```

方法二

1. 写出这道题的暴搜程序（最好是 dfs→6.2@p654）
2. 将这个 dfs 改成「无需外部变量」的 dfs
3. 添加记忆化数组

举例：本文中「采药」的例子

外部参考资料

[1] [NOIP2005] 采药 

7.4 背包 DP

7.4.1 背包 DP 基础

Authors: Ir1d, sshwy, StudyingFather, Marcythm, partychicken, H-J-Granger, NachtgeistW, countercurrent-time, Enter-tainer, Tiphereth-A, weiranfu, greyqz, iamtwz, Konano, ksyx, ouuan, paigeman, wolfdan666, AngelKitty, CCXXXI, cjsoft, diauweb, Early0v0, ezoixx130, GekkaSaori, GoodCoder666, Henry-ZHR, HeRaNO, Link-cute, LovelyBuggies, LuoshuiTianyi, Makkiy, mgt, minghu6, odeinjul, oldoldtea, P-Y-Y, PotassiumWings, SamZhangQingChuan, shenshuaijie, Suyun514, weiyong1024, Xeonacid, xyf007, Alisahhh, Alphnia, c-forrest, cbw2007, dhbloo, fps5283, GavinZhengOI, Gesrua, hsfzLZH1, hydingsy, kenlig, kxccc, lychees, Menci, Peanut-Tang, PlanariaIce, sbofgayschool, shawlleyw, Siyuan, SukkaW, TianKong-y, tLLWtG, WAAutoMaton, x4Cx58x54, xk2013, zhb2000, zhufengning, hhc0001

前置知识：动态规划部分简介→7.1@p714.

引入

在具体讲何为「背包 dp」前，先来看如下的例题：

「USACO07 DEC」Charm Bracelet[1]

题意概要：有 n 个物品和一个容量为 W 的背包，每个物品有重量 w_i 和价值 v_i 两种属性，要求选若干物品放入背包使背包中物品的总价值最大且背包中物品的总重量不超过背包的容量.

在上述例题中，由于每个物体只有两种可能的状态（取与不取），对应二进制中的 0 和 1，这类问题便被称为「0-1 背包问题」.

0-1 背包

解释

例题中已知条件有第 i 个物品的重量 w_i，价值 v_i，以及背包的总容量 W.

设 DP 状态 $f_{i,j}$ 为在只能放前 i 个物品的情况下，容量为 j 的背包所能达到的最大总价值.

考虑转移. 假设当前已经处理好了前 $i-1$ 个物品的所有状态，那么对于第 i 个物品，当其不放入背包时，背包的剩余容量不变，背包中物品的总价值也不变，故这种情况的最大价值为 $f_{i-1,j}$；当其放入背包时，背包的剩余容量会减小 w_i，背包中物品的总价值会增大 v_i，故这种情况的最大价值为 $f_{i-1,j-w_i}+v_i$.

由此可以得出状态转移方程：

$$f_{i,j}=\max(f_{i-1,j},f_{i-1,j-w_i}+v_i)$$

这里如果直接采用二维数组对状态进行记录，会出现 MLE. 可以考虑改用滚动数组的形式来优化.

由于对 f_i 有影响的只有 f_{i-1}，可以去掉第一维，直接用 f_i 来表示处理到当前物品时背包容量为 i 的最大价值，得出以下方程：

$$f_j = \max\left(f_j, f_{j-w_i} + v_i\right)$$

务必牢记并理解这个转移方程，因为大部分背包问题的转移方程都是在此基础上推导出来的.

实现

还有一点需要注意的是，很容易写出这样的 **错误核心代码**：

- C++

```cpp
for (int i = 1; i <= n; i++)
  for (int l = 0; l <= W - w[i]; l++)
    f[l + w[i]] = max(f[l] + v[i], f[l + w[i]]);
// 由 f[i][l + w[i]] = max(max(f[i - 1][l + w[i]], f[i - 1][l] + v[i]),
// f[i][l + w[i]]); 简化而来
```

- Python

```python
for i in range(1, n + 1):
    for l in range(0, W - w[i] + 1):
        f[l + w[i]] = max(f[l] + v[i], f[l + w[i]])
# 由 f[i][l + w[i]] = max(max(f[i - 1][l + w[i]], f[i - 1][l] + v[i]),
# f[i][l + w[i]]) 简化而来
```

这段代码哪里错了呢？枚举顺序错了.

仔细观察代码可以发现：对于当前处理的物品 i 和当前状态 $f_{i,j}$，在 $j \geqslant w_i$ 时，$f_{i,j}$ 是会被 $f_{i,j-w_i}$ 所影响的. 这就相当于物品 i 可以多次被放入背包，与题意不符.（事实上，这正是完全背包问题的解法）

为了避免这种情况发生，我们可以改变枚举的顺序，从 W 枚举到 w_i，这样就不会出现上述的错误，因为 $f_{i,j}$ 总是在 $f_{i,j-w_i}$ 前被更新.

因此实际核心代码为

- C++

```cpp
for (int i = 1; i <= n; i++)
  for (int l = W; l >= w[i]; l--) f[l] = max(f[l], f[l - w[i]] + v[i]);
```

- Python

```python
for i in range(1, n + 1):
    for l in range(W, w[i] - 1, -1):
        f[l] = max(f[l], f[l - w[i]] + v[i])
```

例题代码

```cpp
#include <iostream>
using namespace std;
constexpr int MAXN = 13010;
int n, W, w[MAXN], v[MAXN], f[MAXN];

int main() {
  cin >> n >> W;
  for (int i = 1; i <= n; i++) cin >> w[i] >> v[i];  // 读入数据
  for (int i = 1; i <= n; i++)
    for (int l = W; l >= w[i]; l--)
```

```
11        if (f[l - w[i]] + v[i] > f[l]) f[l] = f[l - w[i]] + v[i];  // 状态方程
12    cout << f[W];
13    return 0;
14 }
```

完全背包

解释

完全背包模型与 0-1 背包类似，与 0-1 背包的区别仅在于一个物品可以选取无限次，而非仅能选取一次.

我们可以借鉴 0-1 背包的思路，进行状态定义：设 $f_{i,j}$ 为只能选前 i 个物品时，容量为 j 的背包可以达到的最大价值.

需要注意的是，虽然定义与 0-1 背包类似，但是其状态转移方程与 0-1 背包并不相同.

过程

可以考虑一个朴素的做法：对于第 i 件物品，枚举其选了多少个来转移. 这样做的时间复杂度是 $O(nW^2)$ 的，其中 n 为物品总数，W 为背包容量.

状态转移方程如下：

$$f_{i,j}=\max_{k=0}^{\lfloor j/w_i \rfloor}(f_{i-1,j-k\times w_i}+v_i\times k)$$

考虑做一个简单的优化. 可以发现，对于 $f_{i,j}$，只要通过 $f_{i,j-w_i}$ 转移就可以了. 因此状态转移方程为：

$$f_{i,j}=\max(f_{i-1,j},f_{i,j-w_i}+v_i)$$

理由是当我们这样转移时，$f_{i,j-w_i}$ 已经由 $f_{i,j-2\times w_i}$ 更新过，那么 $f_{i,j-w_i}$ 就是充分考虑了第 i 件物品所选次数后得到的最优结果. 换言之，我们通过局部最优子结构的性质重复使用了之前的枚举过程，优化了枚举的复杂度.

与 0-1 背包相同，我们可以将第一维去掉来优化空间复杂度. 如果理解了 0-1 背包的优化方式，就不难明白压缩后的循环是正向的（也就是上文中提到的错误优化）.

「Luogu P1616」疯狂的采药[2]

题意概要：有 n 种物品和一个容量为 W 的背包，每种物品有重量 w_i 和价值 v_i 两种属性，要求选若干个物品放入背包使背包中物品的总价值最大且背包中物品的总重量不超过背包的容量.

例题代码

```
1  #include <iostream>
2  using namespace std;
3  constexpr int MAXN = 1e4 + 5;
4  constexpr int MAXW = 1e7 + 5;
5  int n, W, w[MAXN], v[MAXN];
6  long long f[MAXW];
7
8  int main() {
9    cin >> W >> n;
10   for (int i = 1; i <= n; i++) cin >> w[i] >> v[i];
11   for (int i = 1; i <= n; i++)
```

```
    for (int l = w[i]; l <= W; l++)
      if (f[l - w[i]] + v[i] > f[l]) f[l] = f[l - w[i]] + v[i];  // 核心状态方程
  cout << f[W];
  return 0;
}
```

多重背包

多重背包也是 0-1 背包的一个变式. 与 0-1 背包的区别在于每种物品有 k_i 个，而非一个.

一个很朴素的想法就是：把「每种物品选 k_i 次」等价转换为「有 k_i 个相同的物品，每个物品选一次」. 这样就转换成了一个 0-1 背包模型，套用上文所述的方法就可已解决. 状态转移方程如下：

$$f_{i,j}=\max_{k=0}^{k_i}(f_{i-1,j-k\times w_i}+v_i\times k)$$

时间复杂度 $O(W\sum_{i=1}^{n}k_i)$.

核心代码

```
for (int i = 1; i <= n; i++) {
  for (int weight = W; weight >= w[i]; weight--) {
    // 多遍历一层物品数量
    for (int k = 1; k * w[i] <= weight && k <= cnt[i]; k++) {
      dp[weight] = max(dp[weight], dp[weight - k * w[i]] + k * v[i]);
    }
  }
}
```

二进制分组优化

考虑优化. 我们仍考虑把多重背包转化成 0-1 背包模型来求解.

解释

显然，复杂度中的 $O(nW)$ 部分无法再优化了，我们只能从 $O(\sum k_i)$ 处入手. 为了表述方便，我们用 $A_{i,j}$ 代表第 i 种物品拆分出的第 j 个物品.

在朴素的做法中，$\forall j\le k_i$，$A_{i,j}$ 均表示相同物品. 那么我们效率低的原因主要在于我们进行了大量重复性的工作. 举例来说，我们考虑了「同时选 $A_{i,1},A_{i,2}$」与「同时选 $A_{i,2},A_{i,3}$」这两个完全等效的情况. 这样的重复性工作我们进行了许多次. 那么优化拆分方式就成为了解决问题的突破口.

过程

我们可以通过「二进制分组」的方式使拆分方式更加高效.

具体地说就是令 $A_{i,j}(j\in[0,\lfloor\log_2(k_i+1)\rfloor-1])$ 分别表示由 2^j 个单个物品「捆绑」而成的大物品. 特殊地，若 k_i+1 不是 2 的整数次幂，则需要在最后添加一个由 $k_i-2^{\lfloor\log_2(k_i+1)\rfloor}+1$ 个单个物品「捆绑」而成的大物品用于补足.

举几个例子：

- $6=1+2+3$
- $8=1+2+4+1$
- $18=1+2+4+8+3$
- $31=1+2+4+8+16$

显然，通过上述拆分方式，可以表示任意 $\leq k_i$ 个物品的等效选择方式．将每种物品按照上述方式拆分后，使用 0-1 背包的方法解决即可．

时间复杂度 $O(W\sum_{i=1}^{n}\log_2 k_i)$

实现

二进制分组代码

- C++

```cpp
index = 0;
for (int i = 1; i <= m; i++) {
  int c = 1, p, h, k;
  cin >> p >> h >> k;
  while (k > c) {
    k -= c;
    list[++index].w = c * p;
    list[index].v = c * h;
    c *= 2;
  }
  list[++index].w = p * k;
  list[index].v = h * k;
}
```

- Python

```python
index = 0
for i in range(1, m + 1):
    c = 1
    p, h, k = map(int, input().split())
    while k > c:
        k -= c
        index += 1
        list[index].w = c * p
        list[index].v = c * h
        c *= 2
    index += 1
    list[index].w = p * k
    list[index].v = h * k
```

单调队列优化

见 单调队列/单调栈优化→7.15.2@p814.

习题：「Luogu P1776」宝物筛选_NOI 导刊 2010 提高（02）[3]

求方案数

对于给定的一个背包容量、物品费用、其他关系等的问题，求装到一定容量的方案总数．

这种问题就是把求最大值换成求和即可．

例如 0-1 背包问题的转移方程就变成了：

$$dp_j \leftarrow dp_j + dp_{j-c_i} \qquad (j \geq c_i)$$

初始条件：$dp_0 = 1$

因为当容量为 0 时也有一个方案，即什么都不装．

参考资料与注释

- 背包问题九讲 - 崔添翼[4].

外部参考资料

[1] 「USACO07 DEC」Charm Bracelet 

[2]「Luogu P1616」疯狂的采药 

[3] 「Luogu P1776」宝物筛选_NOI 导刊 2010 提高（02） 

[4] 背包问题九讲 - 崔添翼 

7.4.2 背包 DP 进阶

> **Authors:** Ir1d, sshwy, StudyingFather, Marcythm, partychicken, H-J-Granger, NachtgeistW, countercurrent-time, Enter-tainer, Tiphereth-A, weiranfu, greyqz, iamtwz, Konano, ksyx, ouuan, paigeman, wolfdan666, AngelKitty, CCXXXI, cjsoft, diauweb, Early0v0, ezoixx130, GekkaSaori, GoodCoder666, Henry-ZHR, HeRaNO, Link-cute, LovelyBuggies, LuoshuiTianyi, Makkiy, mgt, minghu6, odeinjul, oldoldtea, P-Y-Y, PotassiumWings, SamZhangQingChuan, shenshuaijie, Suyun514, weiyong1024, Xeonacid, xyf007, Alisahhh, Alphnia, c-forrest, cbw2007, dhbloo, fps5283, GavinZhengOI, Gesrua, hsfzLZH1, hydingsy, kenlig, kxccc, lychees, Menci, Peanut-Tang, PlanariaIce, sbofgayschool, shawlleyw, Siyuan, SukkaW, TianKong-y, tLLWtG, WAAutoMaton, x4Cx58x54, xk2013, zhb2000, zhufengning, hhc0001

背包相关 trick

泛化物品的背包

这种背包，单个物品 i 没有固定的费用和价值，它的价值是随着分配给它的费用而定．在背包容量为 V 的背包问题中，当分配给物品 i 的费用为 v_i 时，能得到的价值就是 $h_i(v_i)$.

那么，我们枚举分配给第 i 个物品的重量 w'，这时的物品价值将会是 $h_i(w')$，那么现在的价值就是 $f_{i-1,w-w'}+h_i(w')$．所以此时状态转移的方程为 $f_{i,j}=\max\limits_{0\le k\le j}(f_{i-1,j-k}+h_i(k))$．实际上上面这一堆东西讲的就是 $(\max,+)$ 卷积.

分组背包

> **「Luogu P1757」通天之分组背包**[1]
>
> 有 n 件物品和一个大小为 m 的背包，第 i 个物品的价值为 w_i，体积为 v_i．同时，每个物品属于一个组，同组内最多只能选择一个物品．求背包能装载物品的最大总价值.

这种题其实只是从「在所有物品中选择一件」变成了「从当前组中选择一件」，于是就对每一组进行一次 0-1 背包就可以了.

再说一说如何进行存储．我们可以将 $t_{k,i}$ 表示第 k 组的第 i 件物品的编号是多少，再用 cnt_k 表示第 k 组物品有多少个.

实现

- C++

```cpp
for (int k = 1; k <= ts; k++)           // 循环每一组
  for (int i = m; i >= 0; i--)          // 循环背包容量
    for (int j = 1; j <= cnt[k]; j++)   // 循环该组的每一个物品
      if (i >= w[t[k][j]])              // 背包容量充足
        dp[i] =
            max(dp[i],
```

```
                dp[i - w[t[k][j]]] + c[t[k][j]]);  // 像0-1背包一样状态转移
```

- Python

```python
for k in range(1, ts + 1):  # 循环每一组
    for i in range(m, -1, -1):  # 循环背包容量
        for j in range(1, cnt[k] + 1):  # 循环该组的每一个物品
            if i >= w[t[k][j]]:  # 背包容量充足
                dp[i] = max(
                    dp[i], dp[i - w[t[k][j]]] + c[t[k][j]]
                )  # 像0-1背包一样状态转移
```

这里要注意：**一定不能搞错循环顺序**，这样才能保证正确性.

回退背包

普通的 0-1 背包求方案数只需要直接 dp 即可，但是有时会遇到形如「其他物品都能选，只有几个物品不能选」的情况，而且通常是在同一组物品中多次出现不同的物品不能选，比如部分复杂的树上背包. 这时直接 dp 可能会 TLE，所以需要引入回退背包来处理这种情况.

注意到背包中物品是无序的：对于两个物品，先放哪个不会当前情况造成任何影响，可以认为 **每一个物品都是最后被放入的那个**. 所以可以先把所有东西的 dp 预处理出来，然后把某一个物品的贡献撤销即可. 即：

$$dp_j \leftarrow dp_j - dp_{j-w_i}$$

注意循环时从小到大执行，否则它对应的贡献方式就是完全背包的方式了.

背包杂项

输出方案

输出方案其实就是记录下来背包中的某一个状态是怎么推出来的. 我们可以用 $g_{i,v}$ 表示第 i 件物品占用空间为 v 的时候是否选择了此物品. 然后在转移时记录是选用了哪一种策略（选或不选）. 输出时的伪代码：

```
int v = V;  // 记录当前的存储空间

// 因为最后一件物品存储的是最终状态，所以从最后一件物品进行循环
for (从最后一件循环至第一件) {
  if (g[i][v]) {
    选了第 i 项物品;
    v -= 第 i 项物品的重量;
  } else {
    未选第 i 项物品;
  }
}
```

求最优方案总数

要求最优方案总数，我们要对 0-1 背包里的 dp 数组的定义稍作修改，DP 状态 $f_{i,j}$ 为在只能放前 i 个物品的情况下，容量为 j 的背包「正好装满」所能达到的最大总价值.

这样修改之后，每一种 DP 状态都可以用一个 $g_{i,j}$ 来表示方案数.

$f_{i,j}$ 表示只考虑前 i 个物品时背包体积「正好」是 j 时的最大价值.

$g_{i,j}$ 表示只考虑前 i 个物品时背包体积「正好」是 j 时的方案数.

转移方程：

如果 $f_{i,j}=f_{i-1,j}$ 且 $f_{i,j}\neq f_{i-1,j-v}+w$ 说明我们此时不选择把物品放入背包更优，方案数由 $g_{i-1,j}$ 转移过来，

如果 $f_{i,j}\neq f_{i-1,j}$ 且 $f_{i,j}=f_{i-1,j-v}+w$ 说明我们此时选择把物品放入背包更优，方案数由 $g_{i-1,j-v}$ 转移过来，

如果 $f_{i,j}=f_{i-1,j}$ 且 $f_{i,j}=f_{i-1,j-v}+w$ 说明放入或不放入都能取得最优解，方案数由 $g_{i-1,j}$ 和 $g_{i-1,j-v}$ 转移过来.

初始条件：

```
memset(f, 0xcf, sizeof(f));
// 因为是求最大值，初始化为负无穷，避免没有装满而进行了转移
// 若求最小值，则初始化为正无穷0x3f
f[0] = 0;
g[0] = 1;  // 什么都不装是一种方案
```

因为背包体积最大值有可能装不满，所以最优解不一定是 f_m.

最后我们通过找到最优解的价值，把 g_j 数组里取到最优解的所有方案数相加即可.

实现

```
for (int i = 0; i < N; i++) {
  for (int j = V; j >= v[i]; j--) {
    int tmp = std::max(dp[j], dp[j - v[i]] + w[i]);
    int c = 0;
    if (tmp == dp[j]) c += cnt[j];                      // 如果从dp[j]转移
    if (tmp == dp[j - v[i]] + w[i]) c += cnt[j - v[i]];  // 如果从dp[j-v[i]]转移
    dp[j] = tmp;
    cnt[j] = c;
  }
}
int max = 0;  // 寻找最优解
for (int i = 0; i <= V; i++) {
  max = std::max(max, dp[i]);
}
int res = 0;
for (int i = 0; i <= V; i++) {
  if (dp[i] == max) {
    res += cnt[i];  // 求和最优解方案数
  }
}
```

背包的第 k 优解

普通的 0-1 背包是要求最优解，在普通的背包 DP 方法上稍作改动，增加一维用于记录当前状态下的前 k 优解，即可得到求 0-1 背包第 k 优解的算法. 具体来讲：$f_{i,j,k}$ 记录了前 i 个物品中，选择的物品总体积为 j 时，能够得到的第 k 大的价值和. 这个状态可以理解为将普通 0-1 背包只用记录一个数据的 $f_{i,j}$ 扩展为记录一个有序的优解序列. 转移时，普通背包最优解的求法是 $f_{i,j}=\max(f_{i-1,j},f_{i-1,j-v_i}+w_i)$，现在我们则是要合并 $f_{i-1,j}$，$f_{i-1,j-v_i}+w_i$ 这两个大小为 k 的递减序列，并保留合并后前 k 大的价值记在 $f_{i,j}$ 里，这一步利用双指针法，复杂度是 $O(k)$ 的，整体时间复杂度为 $O(nmk)$. 空间上，此方法与普通背包一样可以压缩掉第一维，复杂度是 $O(mk)$ 的.

例题 HDU 2639 Bone Collector II[2]

求 0-1 背包的严格第 k 优解. $n \leq 100, v \leq 1000, k \leq 30$

实现

```
const int kMaxM = 1010, kMaxK = 33;

int w[kMaxM], c[kMaxM], dp[kMaxM][kMaxK];

void solve(int n, int m, int K) {
  for (int i = 0; i <= m; i++) {
    for (int j = 1; j <= K; j++) dp[i][j] = 0;
  }
  int i, j, p, x, y, z;
  int a[kMaxK], b[kMaxK];
  for (i = 0; i < n; i++) {
    for (j = m; j >= c[i]; j--) {
      for (p = 1; p <= K; p++) {
        a[p] = dp[j - c[i]][p] + w[i];
        b[p] = dp[j][p];
      }
      a[p] = b[p] = -1;
      x = y = z = 1;
      while (z <= K && (a[x] != -1 || b[y] != -1)) {
        if (a[x] > b[y])
          dp[j][z] = a[x++];
        else
          dp[j][z] = b[y++];
        if (dp[j][z] != dp[j][z - 1]) z++;
      }
    }
  }
}

```

背包相关问题

混合背包

混合背包就是将 01 背包、完全背包和多重背包混合起来，有的只能取一次，有的能取无限次，有的只能取 k 次.

这种题目看起来很难，但是每一种物品的选择依然是独立的，因此可以判断当前物品是哪一种背包，然后使用这种背包的解决方案即可.

例题

「Luogu P1833」樱花[3]

有 n 种樱花树和长度为 T 的时间，有的樱花树只能看一遍，有的樱花树最多看 A_i 遍，有的樱花树可以看无数遍. 每棵樱花树都有一个美学值 C_i，求在 T 的时间内看哪些樱花树能使美学值最高.

核心代码

```
for (int i = 1; i <= n; i++) {
  if (cnt[i] == 0) {  // 如果数量没有限制使用完全背包的核心代码
    for (int weight = w[i]; weight <= W; weight++) {
      dp[weight] = max(dp[weight], dp[weight - w[i]] + v[i]);
    }
  } else {  // 物品有限使用多重背包的核心代码，它也可以处理0-1背包问题
    for (int weight = W; weight >= w[i]; weight--) {
      for (int k = 1; k * w[i] <= weight && k <= cnt[i]; k++) {
        dp[weight] = max(dp[weight], dp[weight - k * w[i]] + k * v[i]);
      }
    }
  }
}
```

习题：HDU 5410 CRB and His Birthday[4]

二维费用背包

「Luogu P1855」榨取 kkksc03[5]

有 n 个任务需要完成，完成第 i 个任务需要花费 t_i 分钟，产生 c_i 元的开支.

现在有 T 分钟时间，W 元钱来处理这些任务，求最多能完成多少任务.

这道题是很明显的 0-1 背包问题，可是不同的是选一个物品会消耗两种费用（经费、时间），只需在状态中增加一维存放第二种费用即可．这时状态转移方程变为 $f_{i,j,k}=\max(f_{i-1,j,k},f_{i-1,j-t_i,k-c_i}+w_i)$．本题的 w_i 均是 1.

这时候就要注意，再开一维存放物品编号就不合适了，因为容易 MLE.

实现

- C++

```
for (int k = 1; k <= n; k++) {
  cin >> mi >> ti;
  for (int i = m; i >= mi; i--)    // 对经费进行一层枚举
    for (int j = t; j >= ti; j--)  // 对时间进行一层枚举
      dp[i][j] = max(dp[i][j], dp[i - mi][j - ti] + 1);
}
```

- Python

```
for k in range(1, n + 1):
    for i in range(m, mi - 1, -1):  # 对经费进行一层枚举
        for j in range(t, ti - 1, -1):  # 对时间进行一层枚举
            dp[i][j] = max(dp[i][j], dp[i - mi][j - ti] + 1)
```

有依赖的背包

「Luogu P1064」金明的预算方案[6]

金明有 n 元钱，想要买 m 个物品，第 i 件物品的价格为 v_i，重要度为 p_i．有些物品是从属于某个主件物品的附件，要买这个物品，必须购买它的主件.

> 目标是让所有购买的物品的 $v_i \times p_i$ 之和最大.

直接当成 树上背包→7.7@p739 处理即可. 注意在最后将所有背包合并在一起.

参考资料与注释

- 背包问题九讲 - 崔添翼[7].

外部参考资料

[1]「Luogu P1757」通天之分组背包 

[2] HDU 2639 Bone Collector II 

[3]「Luogu P1833」樱花 

[4] HDU 5410 CRB and His Birthday 

[5]「Luogu P1855」榨取 kkksc03 

[6]「Luogu P1064」金明的预算方案 

[7] 背包问题九讲 - 崔添翼 

7.5 区间 DP

定义

区间类动态规划是线性动态规划的扩展, 它在分阶段地划分问题时, 与阶段中元素出现的顺序和由前一阶段的哪些元素合并而来有很大的关系.

令状态 $f(i,j)$ 表示将下标位置 i 到 j 的所有元素合并能获得的价值的最大值, 那么 $f(i,j)=\max\{f(i,k)+f(k+1,j)+cost\}$, $cost$ 为将这两组元素合并起来的价值.

性质

区间 DP 有以下特点:

合并: 即将两个或多个部分进行整合, 当然也可以反过来;

特征: 能将问题分解为能两两合并的形式;

求解: 对整个问题设最优值, 枚举合并点, 将问题分解为左右两个部分, 最后合并两个部分的最优值得到原问题的最优值.

解释

例题

> **「NOI1995」石子合并**[1]
>
> 题目大意: 在一个环上有 n 个数 $a_1,a_2,\ldots,a_n$, 进行 $n-1$ 次合并操作, 每次操作将相邻的两堆合并成一堆, 能获得新的一堆中的石子数量的和的得分. 你需要最大化你的得分.

需要考虑不在环上, 而在一条链上的情况.

令 $f(i,j)$ 表示将区间 $[i,j]$ 内的所有石子合并到一起的最大得分.

写出 **状态转移方程**: $f(i,j)=\max\{f(i,k)+f(k+1,j)+\sum_{t=i}^{j}a_t\}$ $(i\le k<j)$

令 sum_i 表示 a 数组的前缀和, 状态转移方程变形为 $f(i,j)=\max\{f(i,k)+f(k+1,j)+sum_j-sum_{i-1}\}$.

怎样进行状态转移

由于计算 $f(i,j)$ 的值时需要知道所有 $f(i,k)$ 和 $f(k+1,j)$ 的值，而这两个中包含的元素的数量都小于 $f(i,j)$，所以我们以 $len=j-i+1$ 作为 DP 的阶段．首先从小到大枚举 len，然后枚举 i 的值，根据 len 和 i 用公式计算出 j 的值，然后枚举 k，时间复杂度为 $O(n^3)$

怎样处理环

题目中石子围成一个环，而不是一条链，怎么办呢?

方法一：由于石子围成一个环，我们可以枚举分开的位置，将这个环转化成一个链，由于要枚举 n 次，最终的时间复杂度为 $O(n^4)$.

方法二：我们将这条链延长两倍，变成 $2\times n$ 堆，其中第 i 堆与第 $n+i$ 堆相同，用动态规划求解后，取 $f(1,n),f(2,n+1),...,f(n,2n-1)$ 中的最优值，即为最后的答案．时间复杂度 $O(n^3)$.

实现

- C++

```cpp
for (len = 2; len <= n; len++)
  for (i = 1; i <= 2 * n - len; i++) {
    int j = len + i - 1;
    for (k = i; k < j; k++)
      f[i][j] = max(f[i][j], f[i][k] + f[k + 1][j] + sum[j] - sum[i - 1]);
  }
```

- Python

```python
for len in range(2, n + 1):
    for i in range(1, 2 * n - len + 1):
        j = len + i - 1
        for k in range(i, j):
            f[i][j] = max(f[i][j], f[i][k] + f[k + 1][j] + sum[j] - sum[i
- 1])
```

几道练习题

NOIP 2006 能量项链[2]

NOIP 2007 矩阵取数游戏[3]

「IOI2000」邮局[4]

外部参考资料

[1] 「NOI1995」石子合并 

[2] NOIP 2006 能量项链 

[3] NOIP 2007 矩阵取数游戏 

[4] 「IOI2000」邮局 

7.6 DAG 上的 DP

定义

DAG 即 有向无环图→11.7@p2727，一些实际问题中的二元关系都可使用 DAG 来建模，从而将这些问题转化为 DAG 上的最长（短）路问题.

解释

以这道题为例，来分析一下 DAG 建模的过程.

> **例题 UVa 437 巴比伦塔 The Tower of Babylon**[1]
>
> 有 $n(n \leqslant 30)$ 种砖块，已知三条边长，每种都有无穷多个. 要求选一些立方体摞成一根尽量高的柱子（每个砖块可以自行选择一条边作为高），使得每个砖块的底面长宽分别严格小于它下方砖块的底面长宽，求塔的最大高度.

过程

建立 DAG

由于每个砖块的底面长宽分别严格小于它下方砖块的底面长宽，因此不难将这样一种关系作为建图的依据，而本题也就转化为最长路问题.

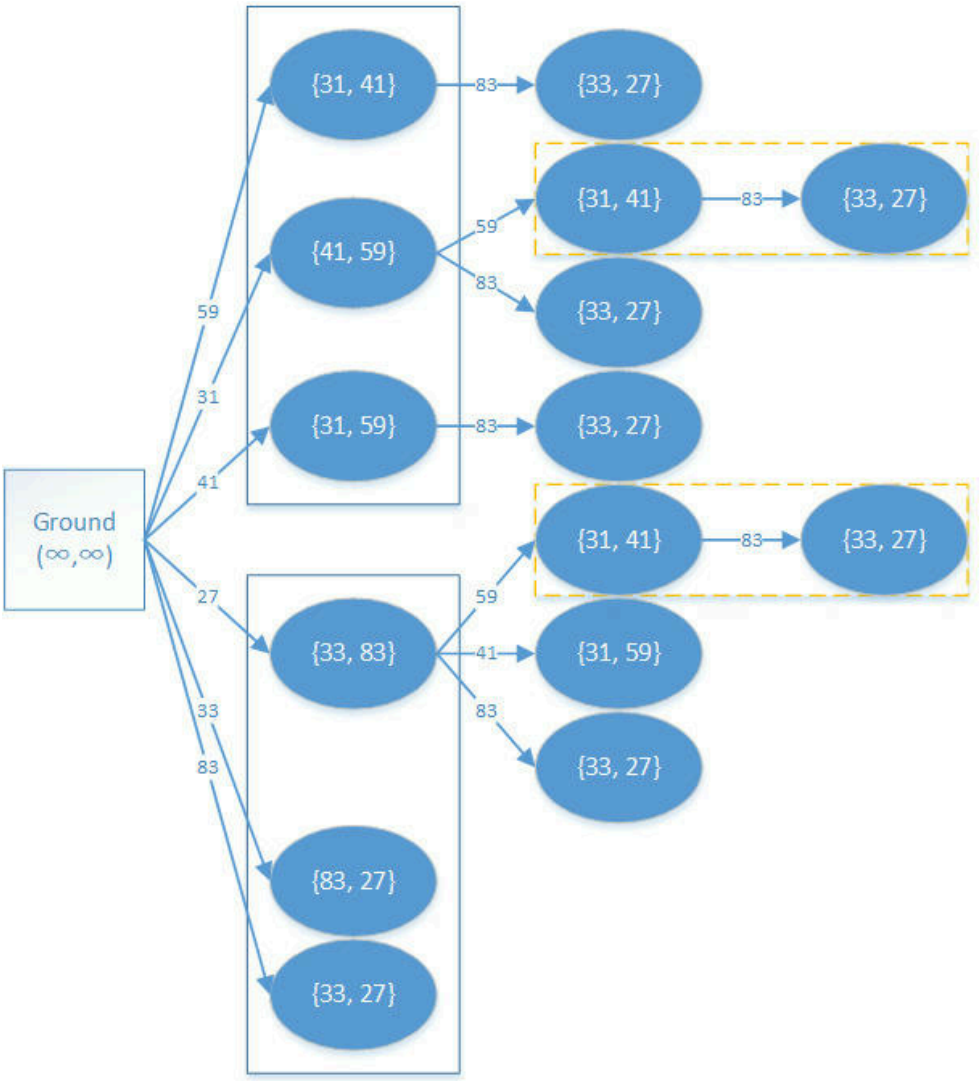
也就是说如果砖块 j 能放在砖块 i 上，那么 i 和 j 之间存在一条边 (i, j)，且边权就是砖块 j 所选取的高.

本题的另一个问题在于每个砖块的高有三种选法，怎样建图更合适呢?

不妨将每个砖块拆解为三种堆叠方式，即将一个砖块分解为三个砖块，每一个拆解得到的砖块都选取不同的高.

初始的起点是大地，大地的底面是无穷大的，则大地可达任意砖块，当然我们写程序时不必特意写上无穷大.

假设有两个砖块，三条边分别为 $31, 41, 59$ 和 $33, 83, 27$，那么整张 DAG 应该如下图所示.



图中蓝色实线框所表示的是一个砖块拆解得到的一组砖块，之所以用 {} 表示底面边长，是因为砖块一旦选取了高，底面边长就是无序的.

图中黄色虚线框表示的是重复计算部分，可以采用 记忆化搜索→7.3@p720 的方法来避免重复计算.

转移

题目要求的是塔的最大高度，已经转化为最长路问题，其起点上文已指出是大地，那么终点呢？显然终点已经自然确定，那就是某砖块上不能再搭别的砖块的时候.

下面我们开始考虑转移方程.

设 $d(i,r)$ 表示第 i 块砖块在最下面，且采取第 r 种堆叠方式时的最大高度. 那么有如下转移方程:

$$d(i,r)=\max\{d(j,r')+h\}$$

其中 j 是所有那些在砖块 i 以 r 方式堆叠时可放上的砖块，r' 对应 j 此时的摆放方式，h 对应砖块 i 采用第 r 种堆叠方式时的高度.

实现

```cpp
#include <cmath>
#include <cstring>
#include <iostream>
```

```cpp
using namespace std;
constexpr int MAXN = 30 + 5;
constexpr int MAXV = 500 + 5;
int d[MAXN][3];
int x[MAXN], y[MAXN], z[MAXN];

int babylon_sub(int c, int rot, int n) {
  if (d[c][rot] != -1) {
    return d[c][rot];
  }
  d[c][rot] = 0;
  int base1, base2;
  if (rot == 0) {  // 处理三个方向
    base1 = x[c];
    base2 = y[c];
  }
  if (rot == 1) {
    base1 = y[c];
    base2 = z[c];
  }
  if (rot == 2) {
    base1 = x[c];
    base2 = z[c];
  }
  for (int i = 0; i < n; i++) {  // 根据不同条件，分别调用不同的递归
    if ((x[i] < base1 && y[i] < base2) || (y[i] < base1 && x[i] < base2))
      d[c][rot] = max(d[c][rot], babylon_sub(i, 0, n) + z[i]);
    if ((y[i] < base1 && z[i] < base2) || (z[i] < base1 && y[i] < base2))
      d[c][rot] = max(d[c][rot], babylon_sub(i, 1, n) + x[i]);
    if ((x[i] < base1 && z[i] < base2) || (z[i] < base1 && x[i] < base2))
      d[c][rot] = max(d[c][rot], babylon_sub(i, 2, n) + y[i]);
  }
  return d[c][rot];
}

int babylon(int n) {
  for (int i = 0; i < n; i++) {
    d[i][0] = -1;
    d[i][1] = -1;
    d[i][2] = -1;
  }
  int r = 0;
  for (int i = 0; i < n; i++) {  // 三种建法
    r = max(r, babylon_sub(i, 0, n) + z[i]);
    r = max(r, babylon_sub(i, 1, n) + x[i]);
    r = max(r, babylon_sub(i, 2, n) + y[i]);
  }
  return r;
}

int main() {
  int t = 0;
```

```
  while (true) {  // 死循环求答案
    int n;
    cin >> n;
    if (n == 0) break;  // 没有砖头了就停止
    t++;
    for (int i = 0; i < n; i++) {
      cin >> x[i] >> y[i] >> z[i];
    }
    cout << "Case " << t << ":"
         << " maximum height = " << babylon(n);  // 递归
    cout << endl;
  }
  return 0;
}

```

外部参考资料

[1] UVa 437 巴比伦塔 The Tower of Babylon 

7.7 树形 DP

Authors: aaron20100919

树形 DP，即在树上进行的 DP．由于树固有的递归性质，树形 DP 一般都是递归进行的．

基础

以下面这道题为例，介绍一下树形 DP 的一般过程．

例题 洛谷 P1352 没有上司的舞会[1]

某大学有 n 个职员，编号为 $1\sim N$．他们之间有从属关系，也就是说他们的关系就像一棵以校长为根的树，父结点就是子结点的直接上司．现在有个周年庆宴会，宴会每邀请来一个职员都会增加一定的快乐指数 a_i，但是呢，如果某个职员的直接上司来参加舞会了，那么这个职员就无论如何也不肯来参加舞会了．所以，请你编程计算，邀请哪些职员可以使快乐指数最大，求最大的快乐指数．

我们设 $f(i,0/1)$ 代表以 i 为根的子树的最优解（第二维的值为 0 代表 i 不参加舞会的情况，1 代表 i 参加舞会的情况）．

对于每个状态，都存在两种决策（其中下面的 x 都是 i 的儿子）：

- 上司不参加舞会时，下属可以参加，也可以不参加，此时有 $f(i,0)=\sum\max\{f(x,1),f(x,0)\}$；
- 上司参加舞会时，下属都不会参加，此时有 $f(i,1)=\sum f(x,0)+a_i$．

我们可以通过 DFS，在返回上一层时更新当前结点的最优解．

```
#include <algorithm>
#include <iostream>
using namespace std;

struct edge {
  int v, next;
```

```
} e[6005];

int head[6005], n, cnt, f[6005][2], ans, is_h[6005], vis[6005];

void addedge(int u, int v) {  // 建图
  e[++cnt].v = v;
  e[cnt].next = head[u];
  head[u] = cnt;
}

void calc(int k) {
  vis[k] = 1;
  for (int i = head[k]; i; i = e[i].next) {  // 枚举该结点的每个子结点
    if (vis[e[i].v]) continue;
    calc(e[i].v);
    f[k][1] += f[e[i].v][0];
    f[k][0] += max(f[e[i].v][0], f[e[i].v][1]);  // 转移方程
  }
  return;
}

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  cin >> n;
  for (int i = 1; i <= n; i++) cin >> f[i][1];
  for (int i = 1; i < n; i++) {
    int l, k;
    cin >> l >> k;
    is_h[l] = 1;
    addedge(k, l);
  }
  for (int i = 1; i <= n; i++)
    if (!is_h[i]) {  // 从根结点开始DFS
      calc(i);
      cout << max(f[i][1], f[i][0]);
      return 0;
    }
}
```

通常，树形 DP 状态一般都为当前节点的最优解. 先 DFS 遍历子树的所有最优解，然后向上传递给子树的父节点来转移，最终根节点的值即为所求的最优解.

习题

- HDU 2196 Computer[2]
- POJ 1463 Strategic game[3]
- [POI2014]FAR-FarmCraft[4]

树上背包

树上的背包问题，简单来说就是背包问题与树形 DP 的结合.

例题 洛谷 P2014 CTSC1997 选课[5]

现在有 n 门课程，第 i 门课程的学分为 a_i，每门课程有零门或一门先修课，有先修课的课程需要先学完其先修课，才能学习该课程.

一位学生要学习 m 门课程，求其能获得的最多学分数.

$n, m \leq 300$

每门课最多只有一门先修课的特点，与有根树中一个点最多只有一个父亲结点的特点类似.

因此可以想到根据这一性质建树，从而所有课程组成了一个森林的结构. 为了方便起见，我们可以新增一门 0 学分的课程（设这个课程的编号为 0），作为所有无先修课课程的先修课，这样我们就将森林变成了一棵以 0 号课程为根的树.

我们设 $f(u, i, j)$ 表示以 u 号点为根的子树中，已经遍历了 u 号点的前 i 棵子树，选了 j 门课程的最大学分.

转移的过程结合了树形 DP 和 背包 DP→7.4.1@p724 的特点，我们枚举 u 点的每个子结点 v，同时枚举以 v 为根的子树选了几门课程，将子树的结果合并到 u 上.

记点 x 的儿子个数为 s_x，以 x 为根的子树大小为 siz_x，可以写出下面的状态转移方程：

$$f(u, i, j) = \max_{v, k \leq j, k \leq siz_v} f(u, i-1, j-k) + f(v, s_v, k)$$

注意上面状态转移方程中的几个限制条件，这些限制条件确保了一些无意义的状态不会被访问到.

f 的第二维可以很轻松地用滚动数组的方式省略掉，注意这时需要倒序枚举 j 的值.

可以证明，该做法的时间复杂度为 $O(nm)$[1].

参考代码

```cpp
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;
int f[305][305], s[305], n, m;
vector<int> e[305];

int dfs(int u) {
  int p = 1;
  f[u][1] = s[u];
  for (auto v : e[u]) {
    int siz = dfs(v);
    // 注意下面两重循环的上界和下界
    // 只考虑已经合并过的子树，以及选的课程数超过 m+1 的状态没有意义
    for (int i = min(p, m + 1); i; i--)
      for (int j = 1; j <= siz && i + j <= m + 1; j++)
        f[u][i + j] = max(f[u][i + j], f[u][i] + f[v][j]);  // 转移方程
    p += siz;
  }
  return p;
}

```

[1]子树合并背包类型的 dp 的复杂度证明 - LYD729 的 CSDN 博客[6]

```cpp
int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  cin >> n >> m;
  for (int i = 1; i <= n; i++) {
    int k;
    cin >> k >> s[i];
    e[k].push_back(i);
  }
  dfs(0);
  cout << f[0][m + 1];
  return 0;
}
```

习题

- 「CTSC1997」选课[7]
- 「JSOI2018」潜入行动[8]
- 「SDOI2017」苹果树[9]
- 「Codeforces Round 875 Div. 1」Problem D. Mex Tree[10]

换根 DP

树形 DP 中的换根 DP 问题又被称为二次扫描，通常不会指定根结点，并且根结点的变化会对一些值，例如子结点深度和、点权和等产生影响.

通常需要两次 DFS，第一次 DFS 预处理诸如深度，点权和之类的信息，在第二次 DFS 开始运行换根动态规划.

接下来以一些例题来带大家熟悉这个内容.

例题 [POI2008]STA-Station[11]

给定一个 n 个点的树，请求出一个结点，使得以这个结点为根时，所有结点的深度之和最大.

不妨令 u 为当前结点，v 为当前结点的子结点. 首先需要用 s_i 来表示以 i 为根的子树中的结点个数，并且有 $s_u = 1 + \sum s_v$. 显然需要一次 DFS 来计算所有的 s_i，这次的 DFS 就是预处理，我们得到了以某个结点为根时其子树中的结点总数.

考虑状态转移，这里就是体现 " 换根 " 的地方了. 令 f_u 为以 u 为根时，所有结点的深度之和.

$f_v \leftarrow f_u$ 可以体现换根，即以 u 为根转移到以 v 为根. 显然在换根的转移过程中，以 v 为根或以 u 为根会导致其子树中的结点的深度产生改变. 具体表现为:

- 所有在 v 的子树上的结点深度都减少了一，那么总深度和就减少了 s_v;
- 所有不在 v 的子树上的结点深度都增加了一，那么总深度和就增加了 $n - s_v$;

根据这两个条件就可以推出状态转移方程 $f_v = f_u - s_v + n - s_v = f_u + n - 2 \times s_v$.

于是在第二次 DFS 遍历整棵树并状态转移 $f_v = f_u + n - 2 \times s_v$，那么就能求出以每个结点为根时的深度和了. 最后只需要遍历一次所有根结点深度和就可以求出答案.

参考代码

```cpp
#include <iostream>
using namespace std;
```

```
3
4  int head[1000010 << 1], tot;
5  long long n, sz[1000010], dep[1000010];
6  long long f[1000010];
7
8  struct node {
9    int to, next;
10 } e[1000010 << 1];
11
12 void add(int u, int v) {  // 建图
13   e[++tot] = {v, head[u]};
14   head[u] = tot;
15 }
16
17 void dfs(int u, int fa) {  // 预处理dfs
18   sz[u] = 1;
19   dep[u] = dep[fa] + 1;
20   for (int i = head[u]; i; i = e[i].next) {
21     int v = e[i].to;
22     if (v != fa) {
23       dfs(v, u);
24       sz[u] += sz[v];
25     }
26   }
27 }
28
29 void get_ans(int u, int fa) {  // 第二次dfs换根dp
30   for (int i = head[u]; i; i = e[i].next) {
31     int v = e[i].to;
32     if (v != fa) {
33       f[v] = f[u] - sz[v] * 2 + n;
34       get_ans(v, u);
35     }
36   }
37 }
38
39 int main() {
40   cin.tie(nullptr)->sync_with_stdio(false);
41   cin >> n;
42   int u, v;
43   for (int i = 1; i <= n - 1; i++) {
44     cin >> u >> v;
45     add(u, v);
46     add(v, u);
47   }
48   dfs(1, 1);
49   for (int i = 1; i <= n; i++) f[1] += dep[i];
50   get_ans(1, 1);
51   long long int ans = -1;
52   int id;
53   for (int i = 1; i <= n; i++) {  // 统计答案
54     if (f[i] > ans) {
```

```
55        ans = f[i];
56        id = i;
57      }
58    }
59    cout << id << '\n';
60    return 0;
61  }
```

习题

- Atcoder Educational DP Contest, Problem V, Subtree[12]
- Educational Codeforces Round 67, Problem E, Tree Painting[13]
- POJ 3585 Accumulation Degree[14]
- [USACO10MAR]Great Cow Gathering G[15]
- CodeForce 708C Centroids[16]

参考资料与注释

[1] 洛谷 P1352 没有上司的舞会

[2] HDU 2196 Computer

[3] POJ 1463 Strategic game

[4] [POI2014]FAR-FarmCraft

[5] 洛谷 P2014 CTSC1997 选课

[6] 子树合并背包类型的 dp 的复杂度证明 - LYD729 的 CSDN 博客

[7] 「CTSC1997」选课

[8] 「JSOI2018」潜入行动

[9] 「SDOI2017」苹果树

[10] 「Codeforces Round 875 Div. 1」Problem D. Mex Tree

[11] [POI2008]STA-Station

[12] Atcoder Educational DP Contest, Problem V, Subtree

[13] Educational Codeforces Round 67, Problem E, Tree Painting

[14] POJ 3585 Accumulation Degree

[15] [USACO10MAR]Great Cow Gathering G

[16] CodeForce 708C Centroids

7.8 状压 DP

简介

状压 DP 是动态规划的一种，通过将状态集合转化为整数记录在 DP 状态中来实现状态转移的目的.

为了达到更低的时间复杂度，通常需要寻找更低状态数的状态. 大部分题目中会利用二元状态，用 n 位二进制数表示 n 个独立二元状态的情况.

使用状态压缩通常涉及位操作，关于基础位操作详见 位操作→9.4@p1099 页面.

例题 1

「SCOI2005」互不侵犯[1]

在 $N \times N$ 的棋盘里面放 K 个国王（$1 \leq N \leq 9, 1 \leq K \leq N \times N$），使他们互不攻击，共有多少种摆放方案.

> 国王能攻击到它上下左右，以及左上左下右上右下八个方向上附近的各一个格子，共 8 个格子.

解释

设 $f(i,j,l)$ 表示前 i 行，第 i 行的状态为 j，且棋盘上已经放置 l 个国王时的合法方案数.

对于编号为 j 的状态，我们用二进制整数 $sit(j)$ 表示国王的放置情况，$sit(j)$ 的某个二进制位为 0 表示对应位置不放国王，为 1 表示在对应位置上放置国王；用 $sta(j)$ 表示该状态的国王个数，即二进制数 $sit(j)$ 中 1 的个数. 例如，如下图所示的状态可用二进制数 100101 来表示（棋盘左边对应二进制低位），则有 $sit(j)=100101_{(2)}=37, sta(j)=3$.



设当前行的状态为 j，上一行的状态为 x，可以得到下面的状态转移方程：$f(i,j,l)=\sum f(i-1,x,l-sta(j))$.

设上一行的状态编号为 x，在保证当前行和上一行不冲突的前提下，枚举所有可能的 x 进行转移，转移方程：

$$f(i,j,l)=\sum f(i-1,x,l-sta(j))$$

实现

参考代码

```cpp
#include <algorithm>
#include <iostream>
using namespace std;
long long sta[2005], sit[2005], f[15][2005][105];
int n, k, cnt;

void dfs(int x, int num, int cur) {
  if (cur >= n) {  // 有新的合法状态
    sit[++cnt] = x;
    sta[cnt] = num;
    return;
  }
  dfs(x, num, cur + 1);  // cur位置不放国王
  dfs(x + (1 << cur), num + 1,
      cur + 2);  // cur位置放国王，与它相邻的位置不能再放国王
}

bool compatible(int j, int x) {
  if (sit[j] & sit[x]) return false;
  if ((sit[j] << 1) & sit[x]) return false;
  if (sit[j] & (sit[x] << 1)) return false;
  return true;
}

int main() {
  cin >> n >> k;
```

```
  dfs(0, 0, 0);  // 先预处理一行的所有合法状态
  for (int j = 1; j <= cnt; j++) f[1][j][sta[j]] = 1;
  for (int i = 2; i <= n; i++)
    for (int j = 1; j <= cnt; j++)
      for (int x = 1; x <= cnt; x++) {
        if (!compatible(j, x)) continue;  // 排除不合法转移
        for (int l = sta[j]; l <= k; l++) f[i][j][l] += f[i - 1][x][l - sta[j]];
      }
  long long ans = 0;
  for (int i = 1; i <= cnt; i++) ans += f[n][i][k];  // 累加答案
  cout << ans << endl;
  return 0;
}
```

例题 2

[POI2004] PRZ[2]

有 n 个人需要过桥，第 i 的人的重量为 w_i，过桥用时为 t_i. 这些人过桥时会分成若干组，只有在某一组的所有人全部过桥后，其余的组才能过桥. 桥最大承重为 W，问这些人全部过桥的最短时间.

$100 \le W \le 400$，$1 \le n \le 16$，$1 \le t_i \le 50$，$10 \le w_i \le 100$.

解释

我们用 S 表示所有人构成集合的一个子集，设 $t(S)$ 表示 S 中人的最长过桥时间，$w(S)$ 表示 S 中所有人的总重量，$f(S)$ 表示 S 中所有人全部过桥的最短时间，则：

$$\begin{cases} f(\emptyset) = 0, \\ f(S) = \min\limits_{T \subseteq S;\ w(T) \le W} \{t(T) + f(S \setminus T)\}. \end{cases}$$

需要注意的是这里不能直接枚举集合再判断是否为子集，而应使用 子集枚举→9.5@p1107，从而使时间复杂度为 $O(3^n)$.

实现

参考代码

```
#include <iostream>
#include <limits>
#include <vector>
using namespace std;

int main() {
  ios::sync_with_stdio(false);
  cin.tie(nullptr);
  int W, n;
  cin >> W >> n;
  const int S = (1 << n) - 1;
  vector<int> ts(S + 1), ws(S + 1);
```

```
  for (int j = 0, t, w; j < n; ++j) {
    cin >> t >> w;
    for (int i = 0; i <= S; ++i)
      if (i & (1 << j)) {
        ts[i] = max(ts[i], t);
        ws[i] += w;
      }
  }
  vector<int> dp(S + 1, numeric_limits<int>::max() / 2);
  for (int i = 0; i <= S; ++i) {
    if (ws[i] <= W) dp[i] = ts[i];
    for (int j = i; j; j = i & (j - 1))
      if (ws[i ^ j] <= W) dp[i] = min(dp[i], dp[j] + ts[i ^ j]);
  }
  cout << dp[S] << '\n';
  return 0;
}

```

习题

- 「NOI2001」炮兵阵地[3]
- 「USACO06NOV」玉米田 Corn Fields[4]
- 「九省联考 2018」一双木棋[5]

外部参考资料

[1] 「SCOI2005」互不侵犯 [2] [POI2004] PRZ [3] 「NOI2001」炮兵阵地

[4] 「USACO06NOV」玉米田 Corn Fields [5]「九省联考 2018」一双木棋

7.9 数位 DP

本页面将简要介绍数位 DP.

引入

数位是指把一个数字按照个、十、百、千等等一位一位地拆开，关注它每一位上的数字．如果拆的是十进制数，那么每一位数字都是 0~9，其他进制可类比十进制.

数位 DP：用来解决一类特定问题，这种问题比较好辨认，一般具有这几个特征：

1. 要求统计满足一定条件的数的数量（即，最终目的为计数）；
2. 这些条件经过转化后可以使用「数位」的思想去理解和判断；
3. 输入会提供一个数字区间（有时也只提供上界）来作为统计的限制；
4. 上界很大（比如 10^{18}），暴力枚举验证会超时.

数位 DP 的基本原理：

考虑人类计数的方式，最朴素的计数就是从小到大开始依次加一．但我们发现对于位数比较多的数，这样的过程中有许多重复的部分．例如，从 7000 数到 7999、从 8000 数到 8999、和从 9000 数到 9999 的过程非常相似，它们都是后三位从 000 变到 999，不一样的地方只有千位这一位，所以我们可以把这些过程归

并起来，将这些过程中产生的计数答案也都存在一个通用的数组里．此数组根据题目具体要求设置状态，用递推或 DP 的方式进行状态转移．

数位 DP 中通常会利用常规计数问题技巧，比如把一个区间内的答案拆成两部分相减（即 $ans_{[l,r]} = ans_{[0,r]} - ans_{[0,l-1]}$）

那么有了通用答案数组，接下来就是统计答案．统计答案可以选择记忆化搜索，也可以选择循环迭代递推．为了不重不漏地统计所有不超过上限的答案，要从高到低枚举每一位，再考虑每一位都可以填哪些数字，最后利用通用答案数组统计答案．

接下来我们具体看几道题目．

例题一

例 1 Luogu P2602 数字计数[1]

题目大意：给定两个正整数 a, b，求在 $[a, b]$ 中的所有整数中，每个数码（digit）各出现了多少次．

方法一

解释

发现对于满 i 位的数，所有数字出现的次数都是相同的，故设数组 dp_i 为满 i 位的数中每个数字出现的次数，此时暂时不处理前导零．则有 $dp_i = 10 \times dp_{i-1} + 10^{i-1}$，这两部分前一个是来自前 $i-1$ 位数字的贡献，后一个是来自第 i 位的数字的贡献．

有了 dp 数组，我们来考虑如何统计答案．将上界按位分开，从高到低枚举，不贴着上界时，后面可以随便取值．贴着上界时，后面就只能取 0 到上界，分两部分分别计算贡献．最后考虑下前导零，第 i 位为前导 0 时，此时 1 到 $i-1$ 位也都是 0，也就是多算了将 $i-1$ 位填满的答案，需要额外减去．

实现

参考代码

```cpp
#include <cstdio>
using namespace std;
constexpr int N = 15;
using ll = long long;
ll l, r, dp[N], mi[N];
ll ans1[N], ans2[N];
int a[N];

void solve(ll n, ll *ans) {
  ll tmp = n;
  int len = 0;
  while (n) a[++len] = n % 10, n /= 10;
  for (int i = len; i >= 1; --i) {
    for (int j = 0; j < 10; j++) ans[j] += dp[i - 1] * a[i];
    for (int j = 0; j < a[i]; j++) ans[j] += mi[i - 1];
    tmp -= mi[i - 1] * a[i], ans[a[i]] += tmp + 1;
    ans[0] -= mi[i - 1];
  }
}

```

```
int main() {
  scanf("%lld%lld", &l, &r);
  mi[0] = 1ll;
  for (int i = 1; i <= 13; ++i) {
    dp[i] = dp[i - 1] * 10 + mi[i - 1];
    mi[i] = 10ll * mi[i - 1];
  }
  solve(r, ans1), solve(l - 1, ans2);
  for (int i = 0; i < 10; ++i) printf("%lld ", ans1[i] - ans2[i]);
  return 0;
}
```

方法二

解释

此题也可以使用记忆化搜索. dp_i 表示不贴上限、无前导零时，位数为 i 的答案.

详见代码注释

过程

参考代码

```
#include <cstdio>
#include <cstring>
#include <iostream>
using namespace std;
using ll = long long;
constexpr int N = 50005;
ll a, b;
ll f[15], ksm[15], p[15], now[15];

ll dfs(int u, int x, bool f0,
       bool lim) {  // u 表示位数，f0 是否有前导零，lim 是否都贴在上限上
  if (!u) {
    if (f0) f0 = false;
    return 0;
  }
  if (!lim && !f0 && (~f[u])) return f[u];
  ll cnt = 0;
  int lst = lim ? p[u] : 9;
  for (int i = 0; i <= lst; i++) {  // 枚举这位要填的数字
    if (f0 && i == 0)
      cnt += dfs(u - 1, x, 1, lim && i == lst);  // 处理前导零
    else if (i == x && lim && i == lst)
      cnt += now[u - 1] + 1 +
             dfs(u - 1, x, 0,
                 lim && i == lst);  // 此时枚举的前几位都贴在给定的上限上.
    else if (i == x)
      cnt += ksm[u - 1] + dfs(u - 1, x, 0, lim && i == lst);
    else
```

```
29        cnt += dfs(u - 1, x, 0, lim && i == lst);
30    }
31    if ((!lim) && (!f0)) f[u] = cnt;  // 只有不贴着上限和没有前导零才能记忆
32    return cnt;
33  }
34
35  ll gans(ll d, int dig) {
36    int len = 0;
37    memset(f, -1, sizeof(f));
38    while (d) {
39      p[++len] = d % 10;
40      d /= 10;
41      now[len] = now[len - 1] + p[len] * ksm[len - 1];
42    }
43    return dfs(len, dig, 1, 1);
44  }
45
46  int main() {
47    scanf("%lld%lld", &a, &b);
48    ksm[0] = 1;
49    for (int i = 1; i <= 12; i++) ksm[i] = ksm[i - 1] * 10ll;
50    for (int i = 0; i < 9; i++) printf("%lld ", gans(b, i) - gans(a - 1, i));
51    printf("%lld\n", gans(b, 9) - gans(a - 1, 9));
52    return 0;
53  }
```

例题二

例 2 HDU 2089 不要 62[2]

题面大意：统计一个区间内数位上不能有 4 也不能有连续的 62 的数有多少.

解释

没有 4 的话在枚举的时候判断一下，不枚举 4 就可以保证状态合法了，所以这个约束没有记忆化的必要，而对于 62 的话，涉及到两位，当前一位是 6 或者不是 6 这两种不同情况计数是不相同的，所以要用状态来记录不同的方案数. $dp_{pos,sta}$ 表示当前第 pos 位，前一位是否是 6 的状态，这里 sta 只需要取 0 和 1 两种状态就可以了，不是 6 的情况可视为同种，不会影响计数.

实现

参考代码

```
1  #include <cstdio>
2  #include <cstring>
3  #include <iostream>
4  using namespace std;
5  int x, y, dp[15][3], p[50];
6
7  void pre() {
```

```
  memset(dp, 0, sizeof(dp));
  dp[0][0] = 1;
  for (int i = 1; i <= 10; i++) {
    dp[i][0] = dp[i - 1][0] * 9 - dp[i - 1][1];
    dp[i][1] = dp[i - 1][0];
    dp[i][2] = dp[i - 1][2] * 10 + dp[i - 1][1] + dp[i - 1][0];
  }
}

int cal(int x) {
  int cnt = 0, ans = 0, tmp = x;
  while (x) {
    p[++cnt] = x % 10;
    x /= 10;
  }
  bool flag = false;
  p[cnt + 1] = 0;
  for (int i = cnt; i; i--) {  // 从高到低枚举数位
    ans += p[i] * dp[i - 1][2];
    if (flag)
      ans += p[i] * dp[i - 1][0];
    else {
      if (p[i] > 4) ans += dp[i - 1][0];
      if (p[i] > 6) ans += dp[i - 1][1];
      if (p[i] > 2 && p[i + 1] == 6) ans += dp[i][1];
      if (p[i] == 4 || (p[i] == 2 && p[i + 1] == 6)) flag = true;
    }
  }
  return tmp - ans;
}

int main() {
  pre();
  while (~scanf("%d%d", &x, &y)) {
    if (!x && !y) break;
    if (x > y) swap(x, y);
    printf("%d\n", cal(y + 1) - cal(x));
  }
  return 0;
}
```

例题三

例 3 SCOI2009 windy 数[3]

题目大意：给定一个区间 $[l,r]$，求其中满足条件 **不含前导 0 且相邻两个数字相差至少为 2** 的数字个数.

解释

首先我们将问题转化成更加简单的形式. 设 ans_i 表示在区间 $[1,i]$ 中满足条件的数的数量，那么所求的答案就是 $ans_r - ans_{l-1}$.

对于一个小于 n 的数，它从高到低肯定出现某一位，使得这一位上的数值小于 n 这一位上对应的数值．而之前的所有位都和 n 上的位相等．

有了这个性质，我们可以定义 $f(i, st, op)$ 表示当前将要考虑的是从高到低的第 i 位，当前该前缀的状态为 st 且前缀和当前求解的数字的大小关系是 op（$op = 1$ 表示等于，$op = 0$ 表示小于）时的数字个数．在本题中，这个前缀的状态就是上一位的值，因为当前将要确定的位不能取哪些数只和上一位有关．在其他题目中，这个值可以是：前缀的数字和，前缀所有数字的 gcd，该前缀取模某个数的余数，也有两种或多种合用的情况．

写出 **状态转移方程**：$f(i, st, op) = \sum_{k=1}^{maxx} f(i+1, k, op = 1 \text{ and } k = maxx) \quad (|\ st - k\ | \geq 2)$

这里的 k 就是当前枚举的下一位的值，而 $maxx$ 就是当前能取到的最高位．因为如果 $op = 1$，那么你在这一位上取的值一定不能大于求解的数字上该位的值，否则没有限制．

我们发现，尽管前缀所选择的状态不同，而 f 的三个参数相同，答案就是一样的．为了防止这个答案被计算多次，可以使用 记忆化搜索→7.3@p720 的方式实现．

实现

参考代码

```
int dfs(int x, int st, int op)  // op=1 =; op=0 <
{
  if (!x) return 1;
  if (!op && ~f[x][st]) return f[x][st];
  int maxx = op ? dim[x] : 9, ret = 0;
  for (int i = 0; i <= maxx; i++) {
    if (abs(st - i) < 2) continue;
    if (st == 11 && i == 0)
      ret += dfs(x - 1, 11, op & (i == maxx));
    else
      ret += dfs(x - 1, i, op & (i == maxx));
  }
  if (!op) f[x][st] = ret;
  return ret;
}

int solve(int x) {
  memset(f, -1, sizeof f);
  dim.clear();
  dim.push_back(-1);
  int t = x;
  while (x) {
    dim.push_back(x % 10);
    x /= 10;
  }
  return dfs(dim.size() - 1, 11, 1);
}
```

例题四

例 4.SPOJMYQ10[4]

题面大意：假如手写下 $[n,m]$ 之间所有整数，会有多少数看起来和在镜子里看起来一模一样？（$n,m<10^{44},T<10^5$）

解释

注：由于这里考虑到的镜像，只有 $0,1,8$ 的镜像是自己本身．所以，这里的「一模一样」并不是传统意义上的回文串，而是只含有 $0,1,8$ 的回文串．

首先，在数位 DP 过程中，显然只有 $0,1,8$ 能被选中．

其次，由于数值超过 long long 范围，所以 $[n,m]=[1,m]-[1,n-1]$ 不再适用（高精度比较繁琐），而是需要对 n 是否合法进行判断，得出：$[n,m]=[1,m]-[1,n]+\text{check}(n)$．

镜像解决了，如何判断回文？

我们需要用一个小数组记录一下之前的值．在未超过一半的长度时，只要不超上限就行；在超过一半的长度时，还需要判断是否和与之「镜面对称」的位相等．

需要额外注意的是，这道题的记忆化部分，不能用 `memset`，否则会导致超时．

实现

参考代码

```cpp
int check(char cc[]) {  // n 的特判
  int strc = strlen(cc);
  for (int i = 0; i < strc; ++i) {
    if (!(cc[i] == cc[strc - i - 1] &&
          (cc[i] == '1' || cc[i] == '8' || cc[i] == '0')))
      return 0ll;
  }
  return 1ll;
}

// now: 当前位, eff: 有效位, fulc: 是否全顶格, ful0: 是否全0
int dfs(int now, int eff, bool ful0, bool fulc) {
  if (now == 0) return 1ll;
  if (!fulc && f[now][eff][ful0] != -1)  // 记忆化
    return f[now][eff][ful0];

  int res = 0, maxk = fulc ? dig[now] : 9;
  for (int i = 0; i <= maxk; ++i) {
    if (i != 0 && i != 1 && i != 8) continue;
    b[now] = i;
    if (ful0 && i == 0)  // 全前导 0
      res += dfs(now - 1, eff - 1, 1, 0);
    else if (now > eff / 2)                                  // 未过半程
      res += dfs(now - 1, eff, 0, fulc && (dig[now] == i));  // 已过半程
    else if (b[now] == b[eff - now + 1])
      res += dfs(now - 1, eff, 0, fulc && (dig[now] == i));
  }
```

```
  if (!fulc) f[now][eff][ful0] = res;
  return res;
}

char cc1[100], cc2[100];
int strc, ansm, ansn;

int get(char cc[]) {  // 处理封装
  strc = strlen(cc);
  for (int i = 0; i < strc; ++i) dig[strc - i] = cc[i] - '0';
  return dfs(strc, strc, 1, 1);
}

scanf("%s%s", cc1, cc2);
printf("%lld\n", get(cc2) - get(cc1) + check(cc1));
```

例题五

例 5.P3311 数数[5]

题面：我们称一个正整数 x 是幸运数，当且仅当它的十进制表示中不包含数字串集合 S 中任意一个元素作为其子串．例如当 $S=\{22,333,0233\}$ 时，233233 是幸运数，23332333、2023320233、32233223 不是幸运数．给定 n 和 S，计算不大于 n 的幸运数个数．答案对 10^9+7 取模．

$1\le n<10^{1201}, 1\le m\le 100, 1\le \sum_{i=1}^{m}|s_i|\le 1500$，$\min_{i=1}^{m}|s_i|\ge 1$，其中 $|s_i|$ 表示字符串 s_i 的长度．n 没有前导 0，但是 s_i 可能有前导 0.

解释

阅读题面发现，如果将数字看成字符串，那么这就是需要完成一个多模匹配，自然而然就想到 AC 自动机．普通数位 DP 中，先从高到低枚举数位，再枚举每一位都填什么，在这道题中，我们也就自然地转化为枚举已经填好的位数，再枚举此时停在 AC 自动机上的哪个节点，然后从当前节点转移到它在 AC 自动机上的子节点．

设 $f(i,j,0/1)$ 表示当前从高到低已经填了 i 位（即在 AC 自动机上走过了 i 条边），此时停在标号为 j 的节点上，当前是否正好贴着上界．

至于题目中的「不包含」条件，只需在 AC 自动机上给每个模式串的结尾节点都打上标记，DP 过程中一旦遇上这些结尾节点就跳过即可．

转移很好想，详见代码主函数部分．

实现

参考代码

```
#include <cstdio>
#include <cstring>
#include <queue>
using namespace std;
using ll = long long;
```

```cpp
constexpr int N = 1505;
constexpr int mod = 1000000007;
int n, m;
char s[N], c[N];
int ch[N][10], fail[N], ed[N], tot, len;

void insert() {
  int now = 0;
  int L = strlen(s);
  for (int i = 0; i < L; ++i) {
    if (!ch[now][s[i] - '0']) ch[now][s[i] - '0'] = ++tot;
    now = ch[now][s[i] - '0'];
  }
  ed[now] = 1;
}

queue<int> q;

void build() {
  for (int i = 0; i < 10; ++i)
    if (ch[0][i]) q.push(ch[0][i]);
  while (!q.empty()) {
    int u = q.front();
    q.pop();
    for (int i = 0; i < 10; ++i) {
      if (ch[u][i]) {
        fail[ch[u][i]] = ch[fail[u]][i], q.push(ch[u][i]),
        ed[ch[u][i]] |= ed[fail[ch[u][i]]];
      } else
        ch[u][i] = ch[fail[u]][i];
    }
  }
  ch[0][0] = 0;
}

ll f[N][N][2], ans;

void add(ll &x, ll y) { x = (x + y) % mod; }

int main() {
  scanf("%s", c);
  n = strlen(c);
  scanf("%d", &m);
  for (int i = 1; i <= m; ++i) scanf("%s", s), insert();
  build();
  f[0][0][1] = 1;
  for (int i = 0; i < n; ++i) {
    for (int j = 0; j <= tot; ++j) {
      if (ed[j]) continue;
      for (int k = 0; k < 10; ++k) {
        if (ed[ch[j][k]]) continue;
        add(f[i + 1][ch[j][k]][0], f[i][j][0]);
```

```
58          if (k < c[i] - '0') add(f[i + 1][ch[j][k]][0], f[i][j][1]);
59          if (k == c[i] - '0') add(f[i + 1][ch[j][k]][1], f[i][j][1]);
60        }
61      }
62    }
63    for (int j = 0; j <= tot; ++j) {
64      if (ed[j]) continue;
65      add(ans, f[n][j][0]);
66      add(ans, f[n][j][1]);
67    }
68    printf("%lld\n", ans - 1);
69    return 0;
70  }
```

此题可以很好地帮助理解数位 DP 的原理.

习题

Ahoi2009 self 同类分布[6]

洛谷 P3413 SAC#1 - 萌数[7]

HDU 6148 Valley Number[8]

CF55D Beautiful numbers[9]

CF628D Magic Numbers[10]

外部参考资料

[1] Luogu P2602 数字计数 

[2] HDU 2089 不要 62 

[3] SCOI2009 windy 数 

[4] SPOJMYQ10 

[5] P3311 数数 

[6] Ahoi2009 self 同类分布 

[7] 洛谷 P3413 SAC#1 - 萌数 

[8] HDU 6148 Valley Number 

[9] CF55D Beautiful numbers 

[10] CF628D Magic Numbers 

7.10 插头 DP

定义

有些 状压 DP→7.8@p744 问题要求我们记录状态的连通性信息，这类问题一般被形象的称为插头 DP 或连通性状态压缩 DP. 例如格点图的哈密顿路径计数，求棋盘的黑白染色方案满足相同颜色之间形成一个连通块的方案数，以及特定图的生成树计数等等. 这些问题通常需要我们对状态的连通性进行编码，讨论状态转移过程中连通性的变化.

引入

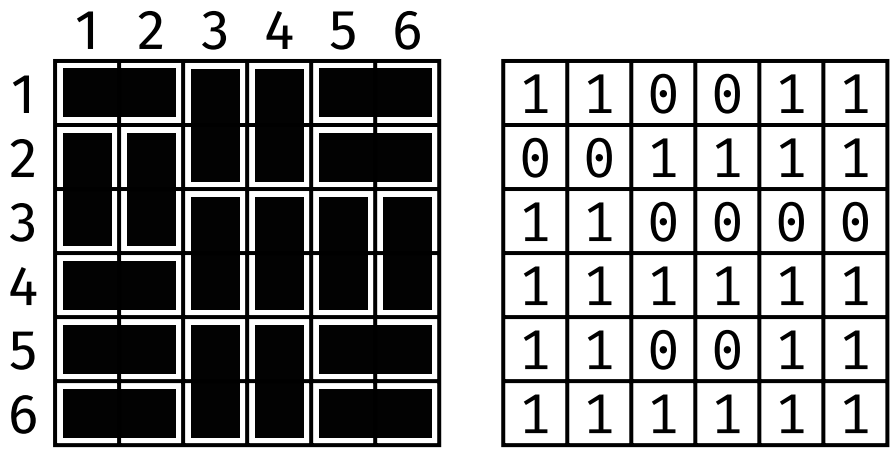
骨牌覆盖与轮廓线 DP

温故而知新，在开始学习插头 DP 之前，不妨先让我们回顾一个经典问题.

例题 「HDU 1400」Mondriaan's Dream[1]

题目大意：在 $N \times M$ 的棋盘内铺满 1×2 或 2×1 的多米诺骨牌，求方案数.

当 n 或 m 规模不大的时候，这类问题可以使用 状压 DP→7.8@p744 解决. 逐行划分阶段，设 $dp(i,s)$ 表示当前已考虑过前 i 行，且第 i 行的状态为 s 的方案数. 这里的状态 s 的每一位可以表示这个位置是否已被上一行覆盖.



另一种划分阶段的方法是逐格 DP，或者称之为轮廓线 DP. $dp(i,j,s)$ 表示已经考虑到第 i 行第 j 列，且当前轮廓线上的状态为 s 的方案数.

虽然逐格 DP 中我们的状态增加了一个维度，但是转移的时间复杂度减少为 $O(1)$，所以时间复杂度未变. 我们用 f_0 表示当前阶段的状态，用 f_1 表示下一阶段的状态，$u = f_0(s)$ 表示当前枚举的函数值，那么有如下的状态转移方程：

```
if (s >> j & 1) {        // 如果已被覆盖
  f1[s ^ 1 << j] += u;   // 不放
} else {                 // 如果未被覆盖
  if (j != m - 1 && (!(s >> j + 1 & 1))) f1[s ^ 1 << j + 1] += u;  // 横放
  f1[s ^ 1 << j] += u;                                             // 竖放
}
```

观察到这里不放和竖放的方程可以合并.

实现

```
#include <algorithm>
#include <iostream>
using namespace std;
constexpr int N = 11;
long long f[2][1 << N], *f0, *f1;
int n, m;

int main() {
  while (cin >> n >> m && n) {
    f0 = f[0];
    f1 = f[1];
    fill(f1, f1 + (1 << m), 0);
```

```
    f1[0] = 1;
    for (int i = 0; i < n; ++i) {
      for (int j = 0; j < m; ++j) {
        swap(f0, f1);
        fill(f1, f1 + (1 << m), 0);
#define u f0[s]
        for (int s = 0; s < 1 << m; ++s)
          if (u) {
            if (j != m - 1 && (!(s >> j & 3))) f1[s ^ 1 << j + 1] += u;  // 横放
            f1[s ^ 1 << j] += u;  // 竖放或不放
          }
      }
    }
    cout << f1[0] << endl;
  }
}
```

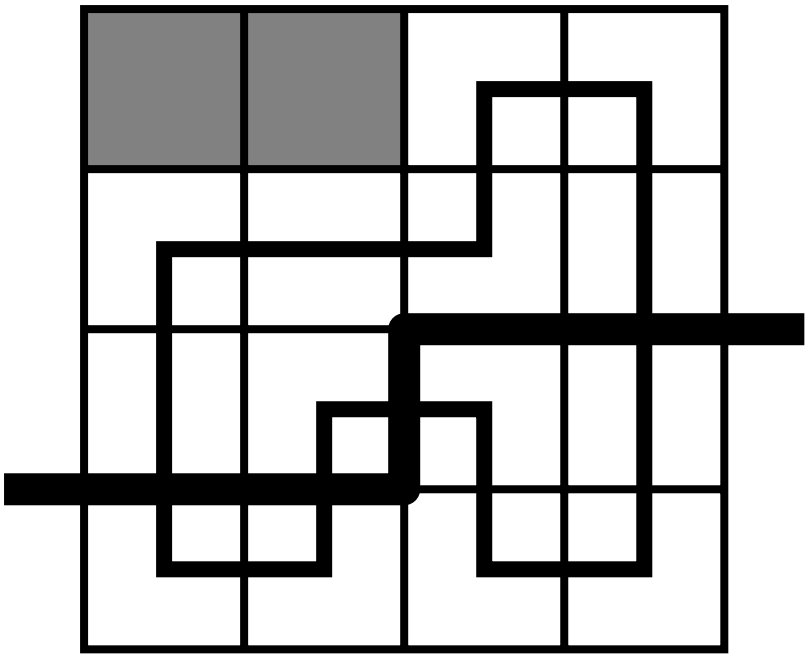
习题 「SRM 671. Div 1 900」BearDestroys[2]

题目大意：给定 $n \times m$ 的矩阵，每个格子有 `E` 或 `S`. 对于一个矩阵，有一个计分方案. 按照行优先的规则扫描每个格子，如果这个格子之前被骨牌占据，则 skip. 否则尝试放多米诺骨牌. 如果放骨牌的方向在矩阵外或被其他骨牌占据，则放置失败，切换另一种方案或 skip. 如果是 `E` 则优先放一个 1×2 的骨牌，如果是 `S` 则优先放一个 2×1 的骨牌. 一个矩阵的得分为最后放的骨牌数. 问所有 2^{nm} 种矩阵的得分的和.

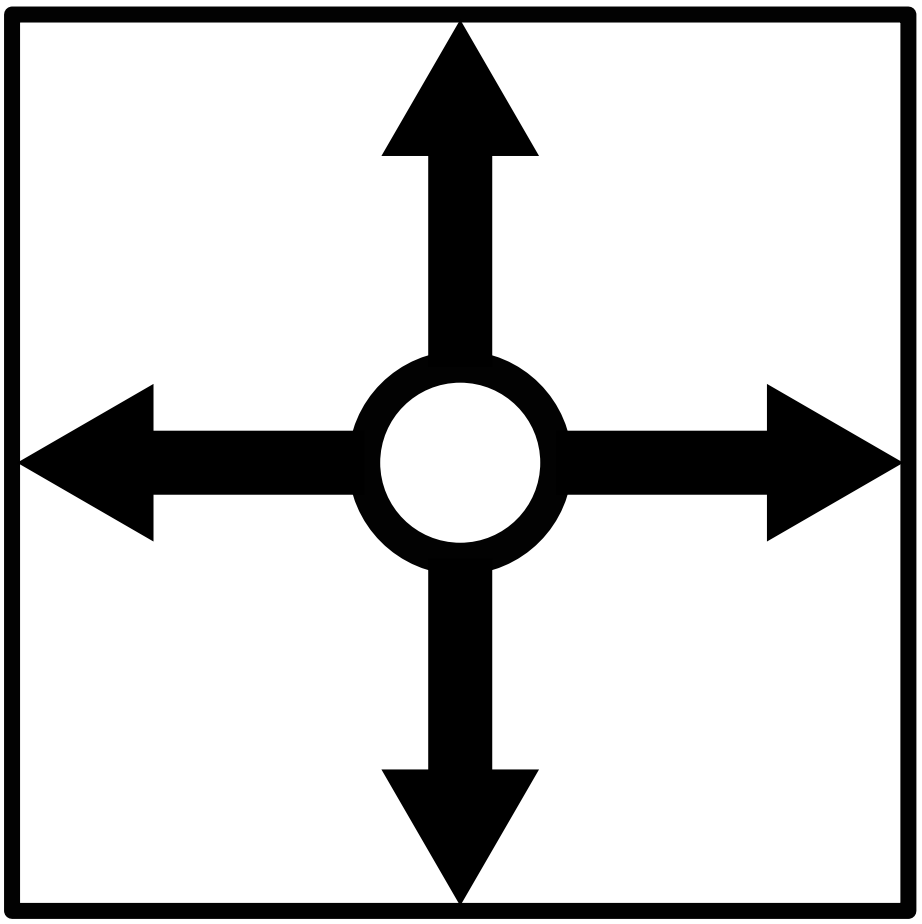
术语

阶段：动态规划执行的顺序，后续阶段的结果只与前序阶段的结果有关（无后效性）. 很多 DP 问题可以有多种划分阶段的方式. 例如在背包问题中，我们通常既可以按照物品划分阶段，也可以按照背包容量划分阶段（外层循环先枚举什么）. 而在多米诺骨牌问题中，我们可以按照行、列、格子以及对角线等特征划分阶段.

轮廓线：已决策状态和未决策状态的分界线.



插头：一个格子某个方向的插头存在，表示这个格子在这个方向与相邻格子相连.



路径模型

多条回路

例题

例题 「HDU 1693」Eat the Trees[3]

题目大意：求用若干条回路覆盖 $N \times M$ 棋盘的方案数，有些位置有障碍.

严格来说，多条回路问题并不属于插头 DP，因为我们只需要和上面的骨牌覆盖问题一样，记录插头是否存在，然后成对的合并和生成插头就可以了.

注意对于一个宽度为 m 的棋盘，轮廓线的宽度为 $m+1$，因为包含 m 个上插头，和 1 个左插头. 注意，当一行迭代完成之后，最右边的左插头通常是不合法的状态，同时我们需要补上下一行第一个左插头，这需要我们调整当前轮廓线的状态，通常是所有状态进行左移，我们把这个操作称为滚动 `roll()`.

例题代码

```cpp
#include <iostream>
using namespace std;
constexpr int N = 11;
long long f[2][1 << (N + 1)], *f0, *f1;
int n, m;

int main() {
  int T;
  cin >> T;
  for (int Case = 1; Case <= T; ++Case) {
    cin >> n >> m;
    f0 = f[0];
    f1 = f[1];
    fill(f1, f1 + (1 << (m + 1)), 0);
    f1[0] = 1;  // 初始化
    for (int i = 0; i < n; ++i) {
      for (int j = 0; j < m; ++j) {
        bool bad;
        cin >> bad;
        bad ^= 1;
        swap(f0, f1);
        fill(f1, f1 + (1 << (m + 1)), 0);
        for (int s = 0; s < 1 << (m + 1); ++s)  // 具体的dp转移，上面都是初始化
          if (f0[s]) {
            bool lt = s >> j & 1, up = s >> (j + 1) & 1;
            if (bad) {
              if (!lt && !up) f1[s] += f0[s];
            } else {
              f1[s ^ 3 << j] += f0[s];
              if (lt != up) f1[s] += f0[s];
            }
          }
      }
      swap(f0, f1);
      fill(f1, f1 + (1 << (m + 1)), 0);
```

```
36        for (int s = 0; s < 1 << m; ++s) f1[s << 1] = f0[s];
37      }
38      cout << "Case " << Case << ": There are " << f1[0]
39           << " ways to eat the trees.\n";
40    }
41  }
```

习题

习题「ZOJ 3466」The Hive II[4]

题目大意：同上题，但格子变成了六边形.

一条回路

例题

例题「Andrew Stankevich Contest 16 - Problem F」Pipe Layout[5]

题目大意：求用一条回路覆盖 $N \times M$ 棋盘的方案数.

在上面的状态表示中我们每合并一组连通的插头，就会生成一条独立的回路，因而在本题中，我们还需要区分插头之间的连通性（出现了!）. 这需要我们对状态进行额外的编码.

状态编码

通常的编码方案有括号表示和最小表示，这里着重介绍泛用性更好的最小表示. 我们用长度 $m+1$ 的整形数组，记录轮廓线上每个插头的状态，0 表示没有插头，并约定连通的插头用相同的数字进行标记.

那么下面两组编码方式表示的是相同的状态:

- `0 3 1 0 1 3`
- `0 1 2 0 2 1`

我们将相同的状态都映射成字典序最小表示，例如在上例中的 `0 1 2 0 2 1` 就是一组最小表示.

我们用 `b[]` 数组表示轮廓线上插头的状态. `bb[]` 表示在最小表示的编码的过程中，每个数字被映射到的最小数字. 注意 0 表示插头不存在，不能被映射成其他值.

代码实现

```
int b[M + 1], bb[M + 1];

int encode() {
  int s = 0;
  memset(bb, -1, sizeof(bb));
  int bn = 1;
  bb[0] = 0;
  for (int i = m; i >= 0; --i) {
#define bi bb[b[i]]
    if (!~bi) bi = bn++;
    s <<= offset;
    s |= bi;
  }
```

```
  return s;
}

void decode(int s) {
  REP(i, m + 1) {
    b[i] = s & mask;
    s >>= offset;
  }
}
```

我们注意到插头总是成对出现，成对消失的．因而 `0 1 2 0 1 2` 这样的状态是不合法的．合法的状态构成一组括号序列，实际中合法状态可能是非常稀疏的．

手写哈希

在一些 状压 DP→7.8@p744 的问题中，合法的状态可能是稀疏的（例如本题），为了优化时空复杂度，我们可以使用哈希表存储合法的 DP 状态．对于 C++ 选手，我们可以使用 std::unordered_map[6]，当然也可以直接手写，这样可以灵活的将状态转移函数也封装于其中．

代码实现

```
constexpr int MaxSZ = 16796, Prime = 9973;

struct hashTable {
  int head[Prime], next[MaxSZ], sz;
  int state[MaxSZ];
  long long key[MaxSZ];

  void clear() {
    sz = 0;
    memset(head, -1, sizeof(head));
  }

  void push(int s) {
    int x = s % Prime;
    for (int i = head[x]; ~i; i = next[i]) {
      if (state[i] == s) {
        key[i] += d;
        return;
      }
    }
    state[sz] = s, key[sz] = d;
    next[sz] = head[x];
    head[x] = sz++;
  }

  void roll() { REP(i, sz) state[i] <<= offset; }
} H[2], *H0, *H1;
```

上面的代码中：

- `MaxSZ` 表示合法状态的上界，可以估计，也可以预处理出较为精确的值．
- `Prime` 一个小于 `MaxSZ` 的大素数．

- `head[]` 表头节点的指针.
- `next[]` 后续状态的指针.
- `state[]` 节点的状态.
- `key[]` 节点的关键字，在本题中是方案数.
- `clear()` 初始化函数，和手写邻接表类似，我们只需要初始化表头节点的指针.
- `push()` 状态转移函数，其中 `d` 是一个全局变量（偷懒），表示每次状态转移所带来的增量. 如果找到的话就 `+=`，否则就创建一个状态为 `s`，关键字为 `d` 的新节点.
- `roll()` 迭代完一整行之后，滚动轮廓线.

关于哈希表的复杂度分析，以及开哈希和闭哈希的不同，可以参见《算法导论》→2.5@p96 中关于散列表的相关章节.

状态转移

代码实现

```
REP(ii, H0->sz) {
  decode(H0->state[ii]);                    // 取出状态，并解码
  d = H0->key[ii];                          // 得到增量 delta
  int lt = b[j], up = b[j + 1];             // 左插头，上插头
  bool dn = i != n - 1, rt = j != m - 1;    // 下插头，右插头
  if (lt && up) {                           // 如果左、上均有插头
    if (lt == up) {                         // 来自同一个连通块
      if (i == n - 1 &&
          j == m - 1) {  // 只有在最后一个格子时，才能合并，封闭回路.
        push(j, 0, 0);
      }
    } else {  // 否则，必须合并这两个连通块，因为本题中需要回路覆盖
      REP(i, m + 1) if (b[i] == lt) b[i] = up;
      push(j, 0, 0);
    }
  } else if (lt || up) {  // 如果左、上之中有一个插头
    int t = lt | up;      // 得到这个插头
    if (dn) {             // 如果可以向下延伸
      push(j, t, 0);
    }
    if (rt) {  // 如果可以向右延伸
      push(j, 0, t);
    }
  } else {           // 如果左、上均没有插头
    if (dn && rt) {  // 生成一对新插头
      push(j, m, m);
    }
  }
}
```

例题代码

```
#include <cassert>
#include <cstring>
#include <iostream>
using namespace std;
```

```
constexpr int M = 10;
constexpr int offset = 3, mask = (1 << offset) - 1;
int n, m;
long long ans, d;
constexpr int MaxSZ = 16796, Prime = 9973;

struct hashTable {
  int head[Prime], next[MaxSZ], sz;
  int state[MaxSZ];
  long long key[MaxSZ];

  void clear() {
    sz = 0;
    memset(head, -1, sizeof(head));
  }

  void push(int s) {
    int x = s % Prime;
    for (int i = head[x]; ~i; i = next[i]) {
      if (state[i] == s) {
        key[i] += d;
        return;
      }
    }
    state[sz] = s, key[sz] = d;
    next[sz] = head[x];
    head[x] = sz++;
  }

  void roll() {
    for (int i = 0; i < sz; i++) state[i] <<= offset;
  }
} H[2], *H0, *H1;

int b[M + 1], bb[M + 1];

int encode() {
  int s = 0;
  memset(bb, -1, sizeof(bb));
  int bn = 1;
  bb[0] = 0;
  for (int i = m; i >= 0; --i) {
    if (!~bb[b[i]]) bb[b[i]] = bn++;
    s <<= offset;
    s |= bb[b[i]];
  }
  return s;
}

void decode(int s) {
  for (int i = 0; i < m + 1; i++) {
    b[i] = s & mask;
```

```
    s >>= offset;
  }
}

void push(int j, int dn, int rt) {
  b[j] = dn;
  b[j + 1] = rt;
  H1->push(encode());
}

int main() {
  cin >> n >> m;
  if (m > n) swap(n, m);
  H0 = H, H1 = H + 1;
  H1->clear();
  d = 1;
  H1->push(0);
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
      swap(H0, H1);
      H1->clear();
      for (int ii = 0; ii < (H0->sz); ii++) {
        decode(H0->state[ii]);
        d = H0->key[ii];
        int lt = b[j], up = b[j + 1];
        bool dn = i != n - 1, rt = j != m - 1;
        if (lt && up) {
          if (lt == up) {
            if (i == n - 1 && j == m - 1) {
              push(j, 0, 0);
            }
          } else {
            for (int i = 0; i < m + 1; i++)
              if (b[i] == lt) b[i] = up;
            push(j, 0, 0);
          }
        } else if (lt || up) {
          int t = lt | up;
          if (dn) {
            push(j, t, 0);
          }
          if (rt) {
            push(j, 0, t);
          }
        } else {
          if (dn && rt) {
            push(j, m, m);
          }
        }
      }
    }
    H1->roll();
```

```
109   }
110   assert(H1->sz <= 1);
111   cout << (H1->sz == 1 ? H1->key[0] : 0) << endl;
112 }
```

习题

习题 「Ural 1519」 Formula 1[7]

题目大意：求用一条回路覆盖 $N \times M$ 棋盘的方案数，有些位置有障碍.

习题 「USACO 5.4.4」 Betsy's Tours[8]

题目大意：一个 $N \times N$ 的方阵（$N \leq 7$），求从左上角出发到左下角结束经过每个格子的路径总数. 虽然是一条路径，但因为起点和终点固定，可以转化为一条回路问题.

习题 「POJ 1739」 Tony's Tour[9]

题目大意：一个 $N \times M$ 的棋盘，求从左下角出发到右下角结束经过每个格子的路径总数，有些位置有障碍.

习题 「USACO 6.1.1」 Postal Vans[10]

题目大意：求用一条有向回路覆盖 $4 \times N$ 的棋盘的方案数，需要高精度.

习题 「HNOI 2007」 神奇游乐园[11]

题目大意：给定一个 $n \times m$ 的网格图，每格内有一个权值，求一个任意一个回路，最大化经过的权值和.

习题 「ProjectEuler 393」 Migrating ants[12]

题目大意：用多条回路覆盖 $n \times n$ 的方阵，每个有 m 条回路的方案对答案的贡献是 2^m，求所有方案的贡献和.

一条路径

例题

例题 「ZOJ 3213」 Beautiful Meadow[13]

题目大意：一个 $N \times M$ 的方阵（$N, M \leq 8$），每个格点有一个权值，求一段路径，最大化路径覆盖的格点的权值和.

本题是标准的一条路径问题，在一条路径问题中，编码的状态中还会存在不能配对的独立插头. 需要在状态转移函数中，额外讨论独立插头的生成、合并与消失的情况. 独立插头的生成和消失对应着路径的一端，因而这类事件不会发生超过两次（一次生成一次消失，或者两次生成一次合并），否则最终结果一定会出现多个连通块.

我们需要在状态中额外记录这类事件发生的总次数，可以将这个信息编码进状态里（注意，类似这样的额外信息在调整轮廓线的时候，不需要跟着滚动），当然也可以在 `hashTable` 数组的外面加维．下面的范例程序中我们选择后者．

状态转移

代码实现

```
REP(i, n) {
  REP(j, m) {
    checkMax(ans, A[i][j]);  // 需要单独处理一个格子的情况
    if (!A[i][j]) continue;  // 如果有障碍，则跳过，注意这时状态数组不需要滚动
    swap(H0, H1);
    REP(c, 3)
    H1[c].clear();  // c 表示生成和消失事件发生的总次数，最多不超过 2 次
    REP(c, 3) REP(ii, H0[c].sz) {
      decode(H0[c].state[ii]);
      d = H0[c].key[ii] + A[i][j];
      int lt = b[j], up = b[j + 1];
      bool dn = A[i + 1][j], rt = A[i][j + 1];
      if (lt && up) {
        if (lt == up) {  // 在一条路径问题中，我们不能合并相同的插头．
          // Cannot deploy here...
        } else {  // 有可能参与合并的两者中有独立插头，但是也可以用同样的代码片段处理
          REP(i, m + 1) if (b[i] == lt) b[i] = up;
          push(c, j, 0, 0);
        }
      } else if (lt || up) {
        int t = lt | up;
        if (dn) {
          push(c, j, t, 0);
        }
        if (rt) {
          push(c, j, 0, t);
        }
        // 一个插头消失的情况，如果是独立插头则意味着消失，如果是成对出现的插头则相当于生成了一个独立插头，
        // 无论哪一类事件都需要将 c + 1.
        if (c < 2) {
          push(c + 1, j, 0, 0);
        }
      } else {
        d -= A[i][j];
        H1[c].push(H0[c].state[ii]);
        d += A[i][j];    // 跳过插头生成，本题中不要求全部覆盖
        if (dn && rt) {  // 生成一对插头
          push(c, j, m, m);
        }
        if (c < 2) {  // 生成一个独立插头
          if (dn) {
            push(c + 1, j, m, 0);
          }
          if (rt) {
```

```
            push(c + 1, j, 0, m);
          }
        }
      }
    }
  }
  REP(c, 3) H1[c].roll();  // 一行结束，调整轮廓线
}
```

例题代码

```
#include <cstring>
#include <iostream>
using namespace std;

template <class T>
bool checkMax(T &a, const T b) {
  return a < b ? a = b, true : false;
}

constexpr int N = 8, M = 8;
constexpr int offset = 3, mask = (1 << offset) - 1;
int A[N + 1][M + 1];
int n, m;
int ans, d;
constexpr int MaxSZ = 16796, Prime = 9973;

struct hashTable {
  int head[Prime], next[MaxSZ], sz;
  int state[MaxSZ];
  int key[MaxSZ];

  void clear() {
    sz = 0;
    memset(head, -1, sizeof(head));
  }

  void push(int s) {
    int x = s % Prime;
    for (int i = head[x]; ~i; i = next[i]) {
      if (state[i] == s) {
        checkMax(key[i], d);
        return;
      }
    }
    state[sz] = s, key[sz] = d;
    next[sz] = head[x];
    head[x] = sz++;
  }

  void roll() {
```

```
    for (int i = 0; i < sz; i++) state[i] <<= offset;
  }
} H[2][3], *H0, *H1;

int b[M + 1], bb[M + 1];

int encode() {
  int s = 0;
  memset(bb, -1, sizeof(bb));
  int bn = 1;
  bb[0] = 0;
  for (int i = m; i >= 0; --i) {
    if (!~bb[b[i]]) bb[b[i]] = bn++;
    s <<= offset;
    s |= bb[b[i]];
  }
  return s;
}

void decode(int s) {
  for (int i = 0; i < m + 1; i++) {
    b[i] = s & mask;
    s >>= offset;
  }
}

void push(int c, int j, int dn, int rt) {
  b[j] = dn;
  b[j + 1] = rt;
  H1[c].push(encode());
}

void init() {
  cin >> n >> m;
  H0 = H[0], H1 = H[1];
  for (int c = 0; c < 3; c++) H1[c].clear();
  d = 0;
  H1[0].push(0);
  memset(A, 0, sizeof(A));
  for (int i = 0; i < n; i++)
    for (int j = 0; j < m; j++) cin >> A[i][j];
}

void solve() {
  ans = 0;
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
      checkMax(ans, A[i][j]);  // 需要单独处理一个格子的情况
      if (!A[i][j]) continue;  // 如果有障碍，则跳过，注意这时状态数组不需要滚动
      swap(H0, H1);
      for (int c = 0; c < 3; c++)
        H1[c].clear();  // c 表示生成和消失事件发生的总次数，最多不超过 2 次
```

```
93        for (int c = 0; c < 3; c++)
94          for (int ii = 0; ii < H0[c].sz; ii++) {
95            decode(H0[c].state[ii]);
96            d = H0[c].key[ii] + A[i][j];
97            int lt = b[j], up = b[j + 1];
98            bool dn = A[i + 1][j], rt = A[i][j + 1];
99            if (lt && up) {
100             if (lt == up) {  // 在一条路径问题中，我们不能合并相同的插头。
101               // Cannot deploy here...
102             } else {  // 有可能参与合并的两者中有独立插头，但是也可以用同样的代码片段处理
103               for (int i = 0; i < m + 1; i++)
104                 if (b[i] == lt) b[i] = up;
105               push(c, j, 0, 0);
106             }
107           } else if (lt || up) {
108             int t = lt | up;
109             if (dn) {
110               push(c, j, t, 0);
111             }
112             if (rt) {
113               push(c, j, 0, t);
114             }
115             // 一个插头消失的情况，如果是独立插头则意味着消失，如果是成对出现的插头则相当于生成了一个独立插头，
116             // 无论哪一类事件都需要将 c + 1。
117             if (c < 2) {
118               push(c + 1, j, 0, 0);
119             }
120           } else {
121             d -= A[i][j];
122             H1[c].push(H0[c].state[ii]);
123             d += A[i][j];  // 跳过插头生成，本题中不要求全部覆盖
124             if (dn && rt) {  // 生成一对插头
125               push(c, j, m, m);
126             }
127             if (c < 2) {  // 生成一个独立插头
128               if (dn) {
129                 push(c + 1, j, m, 0);
130               }
131               if (rt) {
132                 push(c + 1, j, 0, m);
133               }
134             }
135           }
136         }
137     }
138     for (int c = 0; c < 3; c++) H1[c].roll();  // 一行结束，调整轮廓线
139   }
140   for (int ii = 0; ii < H1[2].sz; ii++) checkMax(ans, H1[2].key[ii]);
141   cout << ans << endl;
142 }
```

```
143
144 int main() {
145   int T;
146   cin >> T;
147   while (T--) {
148     init();
149     solve();
150   }
151 }
```

习题

习题「BZOJ 2310」ParkII[14]

题目大意：$m \times n$ 的棋盘，每个格点有一个权值，求一条路径覆盖，最大化路径经过的点的权值和.

习题「NOI 2010 Day2」旅行路线[15]

题目大意：$n \times m$ 的棋盘，棋盘的每个格子有一个 01 权值 T[x][y]，要求寻找一个路径覆盖，满足：

- 第 i 个参观的格点 (x, y)，满足 T[x][y]= L[i]
- 路径的一端在棋盘的边界上

求可行的方案数.

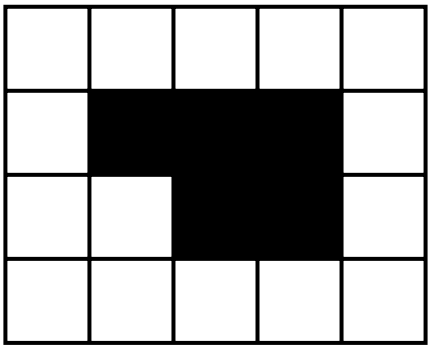
染色模型

除了路径模型之外，还有一类常见的模型，需要我们对棋盘进行染色，相邻的相同颜色节点被视为连通．在路径类问题中，状态转移的时候我们枚举当前路径的方向，而在染色类问题中，我们枚举当前节点染何种颜色．在染色模型中，状态中处在相同连通性的节点可能不止两个．但总体来说依然大同小异．我们不妨来看一个经典的例题.

例题「UVa 10572」Black & White

例题「UVa 10572」Black & White[16]

题目大意：在 $N \times M$ 的棋盘内对未染色的格点进行黑白染色，要求所有黑色区域和白色区域连通，且任意一个 2×2 的子矩形内的颜色不能完全相同（例如下图中的情况非法），求合法的方案数，并构造一组合法的方案.



状态编码

我们先考虑状态编码．不考虑连通性，那么就是 SGU 197. Nice Patterns Strike Back[17]，不难用 状压DP→7.8@p744 直接解决．现在我们需要在状态中同时体现颜色和连通性的信息，考察轮廓线上每个位置的状态，二进制的每 `Offset` 位描述轮廓线上的一个位置，因为只有黑白两种颜色，我们用最低位的奇偶性表示颜色，其余部分示连通性．

考虑第一行上面的节点，和第一列左侧节点，如果要避免特判的话，可以考虑引入第三种颜色区分它们，这里我们观察到这些边界状态的连通性信息一定为 0，所以不需要对第三种颜色再进行额外编码．

在路径问题中我们的轮廓线是由 m 个上插头与 1 个左插头组成的．本题中，由于我们还需要判断当前格点为右下角的 2×2 子矩形是否合法，所以需要记录左上角格子的颜色，因此轮廓线的长度依然是 $m+1$．

这样的编码方案中依然保留了很多冗余信息，（连通的区域颜色一定相同，且左上角的格子只需要颜色信息不需要连通性），但是因为已经用了哈希表和最小表示，对时间复杂度的影响不大，为了降低编程压力，就不再细化了．

在最多情况下（例如第一行黑白相间），每个插头的连通性信息都不一样，因此我们需要 4 位二进制位记录连通性，再加上颜色信息，本题的 `Offset` 为 5 位．

代码实现

```
constexpr int Offset = 5, Mask = (1 << Offset) - 1;
int c[N + 2];
int b[N + 2], bb[N + 3];

T_state encode() {
  T_state s = 0;
  memset(bb, -1, sizeof(bb));
  int bn = 1;
  bb[0] = 0;
  for (int i = m; i >= 0; --i) {
#define bi bb[b[i]]
    if (!~bi) bi = bn++;
    s <<= Offset;
    s |= (bi << 1) | c[i];
  }
  return s;
```

```
}

void decode(T_state s) {
  REP(i, m + 1) {
    b[i] = s & Mask;
    c[i] = b[i] & 1;
    b[i] >>= 1;
    s >>= Offset;
  }
}
```

手写哈希

因为需要构造任意一组方案，这里的哈希表我们需要添加一组域 `pre[]` 来记录每个状态在上一阶段的任意一个前驱.

代码实现

```
constexpr int Prime = 9979, MaxSZ = 1 << 20;

template <class T_state, class T_key>
struct hashTable {
  int head[Prime];
  int next[MaxSZ], sz;
  T_state state[MaxSZ];
  T_key key[MaxSZ];
  int pre[MaxSZ];

  void clear() {
    sz = 0;
    memset(head, -1, sizeof(head));
  }

  void push(T_state s, T_key d, T_state u) {
    int x = s % Prime;
    for (int i = head[x]; ~i; i = next[i]) {
      if (state[i] == s) {
        key[i] += d;
        return;
      }
    }
    state[sz] = s, key[sz] = d, pre[sz] = u;
    next[sz] = head[x], head[x] = sz++;
  }

  void roll() { REP(ii, sz) state[ii] <<= Offset; }
};

hashTable<T_state, T_key> _H, H[N][N], *H0, *H1;
```

方案构造

有了上面的信息，我们就可以容易的构造方案了．首先遍历当前哈希表中的状态，如果连通块数目不超过 2，那么统计进方案数．如果方案数不为 0，我们倒序用 `pre` 数组构造出方案，注意每一行的末尾因为我们执行了 `Roll()` 操作，颜色需要取 `c[j+1]`．

代码实现

```
void print() {
  T_key z = 0;
  int u;
  REP(i, H1->sz) {
    decode(H1->state[i]);
    if (*max_element(b + 1, b + m + 1) <= 2) {
      z += H1->key[i];
      u = i;
    }
  }
  cout << z << endl;
  if (z) {
    DWN(i, n, 0) {
      B[i][m] = 0;
      DWN(j, m, 0) {
        decode(H[i][j].state[u]);
        int cc = j == m - 1 ? c[j + 1] : c[j];
        B[i][j] = cc ? 'o' : '#';
        u = H[i][j].pre[u];
      }
    }
    REP(i, n) puts(B[i]);
  }
  puts("");
}
```

状态转移

我们记：

- `cc` 当前正在染色的格子的颜色
- `lf` 左边格子的颜色
- `up` 上边格子的颜色
- `lu` 左上格子的颜色

我们用 -1 表示颜色不存在．接下来讨论状态转移，一共有三种情况，合并，继承与生成：

状态转移 - 代码

```
void trans(int i, int j, int u, int cc) {
  decode(H0->state[u]);
  int lf = j ? c[j - 1] : -1, lu = b[j] ? c[j] : -1,
      up = b[j + 1] ? c[j + 1] : -1;  // 没有颜色也是颜色的一种!
  if (lf == cc && up == cc) {         // 合并
    if (lu == cc) return;             // 2x2 子矩形相同的情况
    int lf_b = b[j - 1], up_b = b[j + 1];
```

```
    REP(i, m + 1) if (b[i] == up_b) { b[i] = lf_b; }
    b[j] = lf_b;
  } else if (lf == cc || up == cc) {  // 继承
    if (lf == cc)
      b[j] = b[j - 1];
    else
      b[j] = b[j + 1];
  } else {                                                  // 生成
    if (i == n - 1 && j == m - 1 && lu == cc) return;  // 特判
    b[j] = m + 2;
  }
  c[j] = cc;
  if (!ok(i, j, cc)) return;  // 判断是否会因生成封闭的连通块导致不合法
  H1->push(encode(), H0->key[u], u);
}
```

对于最后一种情况需要注意的是，如果已经生成了一个封闭的连通区域，那么我们不能再使用她的颜色染色，否则这种颜色会出现两个连通块. 我们似乎需要额度记录这种事件，可以参考「ZOJ 3213」Beautiful Meadow→7@p713 中的做法，再开一维记录这个事件. 不过利用本题的特殊性，我们也可以特判掉.

特判 - 代码

```
bool ok(int i, int j, int cc) {
  if (cc == c[j + 1]) return true;
  int up = b[j + 1];
  if (!up) return true;
  int c1 = 0, c2 = 0;
  REP(i, m + 1) if (i != j + 1) {
    if (b[i] == b[j + 1]) {  // 连通性相同，颜色一定相同
      assert(c[i] == c[j + 1]);
    }
    if (c[i] == c[j + 1] && b[i] == b[j + 1]) ++c1;
    if (c[i] == c[j + 1]) ++c2;
  }
  if (!c1) {                // 如果会生成新的封闭连通块
    if (c2) return false;  // 如果轮廓线上还有相同的颜色
    if (i < n - 1 || j < m - 2) return false;
  }
  return true;
}
```

进一步讨论连通块消失的情况. 每当我们对一个格子进行染色后，如果没有其他格子与其上侧的格子连通，那么会形成一个封闭的连通块. 这个事件仅在最后一行的最后两列时可以发生，否则后续为了不出现 2×2 的同色连通块，这个颜色一定会再次出现，除了下面的情况：

```
2 2
o#
#o
```

我们特判掉这种情况，这样在本题中，就可以偷懒不用记录之前是否已经生成了封闭的连通块了.

例题代码

```cpp
#include <algorithm>
#include <cassert>
#include <cstring>
#include <iostream>
using namespace std;
using T_state = long long;
using T_key = int;
constexpr int N = 8;
int n, m;
char A[N + 1][N + 1], B[N + 1][N + 1];
constexpr int Offset = 5, Mask = (1 << Offset) - 1;
int c[N + 2];
int b[N + 2], bb[N + 3];

T_state encode() {
  T_state s = 0;
  memset(bb, -1, sizeof(bb));
  int bn = 1;
  bb[0] = 0;
  for (int i = m; i >= 0; --i) {
    if (!~bb[b[i]]) bb[b[i]] = bn++;
    s <<= Offset;
    s |= (bb[b[i]] << 1) | c[i];
  }
  return s;
}

void decode(T_state s) {
  for (int i = 0; i < m + 1; i++) {
    b[i] = s & Mask;
    c[i] = b[i] & 1;
    b[i] >>= 1;
    s >>= Offset;
  }
}

constexpr int Prime = 9979, MaxSZ = 1 << 20;

template <class T_state, class T_key>
struct hashTable {
  int head[Prime];
  int next[MaxSZ], sz;
  T_state state[MaxSZ];
  T_key key[MaxSZ];
  int pre[MaxSZ];

  void clear() {
    sz = 0;
    memset(head, -1, sizeof(head));
  }

```

```
  void push(T_state s, T_key d, T_state u) {
    int x = s % Prime;
    for (int i = head[x]; ~i; i = next[i]) {
      if (state[i] == s) {
        key[i] += d;
        return;
      }
    }
    state[sz] = s, key[sz] = d, pre[sz] = u;
    next[sz] = head[x], head[x] = sz++;
  }

  void roll() {
    for (int ii = 0; ii < sz; ii++) state[ii] <<= Offset;
  }
};

hashTable<T_state, T_key> _H, H[N][N], *H0, *H1;

bool ok(int i, int j, int cc) {
  if (cc == c[j + 1]) return true;
  int up = b[j + 1];
  if (!up) return true;
  int c1 = 0, c2 = 0;
  for (int i = 0; i < m + 1; i++)
    if (i != j + 1) {
      if (b[i] == b[j + 1]) {  // 连通性相同，颜色一定相同
        assert(c[i] == c[j + 1]);
      }
      if (c[i] == c[j + 1] && b[i] == b[j + 1]) ++c1;
      if (c[i] == c[j + 1]) ++c2;
    }
  if (!c1) {                // 如果会生成新的封闭连通块
    if (c2) return false;  // 如果轮廓线上还有相同的颜色
    if (i < n - 1 || j < m - 2) return false;
  }
  return true;
}

void trans(int i, int j, int u, int cc) {
  decode(H0->state[u]);
  int lf = j ? c[j - 1] : -1, lu = b[j] ? c[j] : -1,
      up = b[j + 1] ? c[j + 1] : -1;  // 没有颜色也是颜色的一种!
  if (lf == cc && up == cc) {         // 合并
    if (lu == cc) return;             // 2x2 子矩形相同的情况
    int lf_b = b[j - 1], up_b = b[j + 1];
    for (int i = 0; i < m + 1; i++)
      if (b[i] == up_b) {
        b[i] = lf_b;
      }
    b[j] = lf_b;
  } else if (lf == cc || up == cc) {  // 继承
```

```
104     if (lf == cc)
105       b[j] = b[j - 1];
106     else
107       b[j] = b[j + 1];
108   } else {                                              // 生成
109     if (i == n - 1 && j == m - 1 && lu == cc) return;  // 特判
110     b[j] = m + 2;
111   }
112   c[j] = cc;
113   if (!ok(i, j, cc)) return;  // 判断是否会因生成封闭的连通块导致不合法
114   H1->push(encode(), H0->key[u], u);
115 }
116
117 void init() {
118   cin >> n >> m;
119   for (int i = 0; i < n; i++) cin >> A[i];
120 }
121
122 void solve() {
123   H1 = &_H, H1->clear(), H1->push(0, 1, 0);
124   for (int i = 0; i < n; i++) {
125     for (int j = 0; j < m; j++) {
126       H0 = H1, H1 = &H[i][j], H1->clear();
127       for (int u = 0; u < H0->sz; u++) {
128         if (A[i][j] == '.' || A[i][j] == '#') trans(i, j, u, 0);
129         if (A[i][j] == '.' || A[i][j] == 'o') trans(i, j, u, 1);
130       }
131     }
132     H1->roll();
133   }
134 }
135
136 void print() {
137   T_key z = 0;
138   int u;
139   for (int i = 0; i < H1->sz; i++) {
140     decode(H1->state[i]);
141     if (*max_element(b + 1, b + m + 1) <= 2) {
142       z += H1->key[i];
143       u = i;
144     }
145   }
146   cout << z << endl;
147   if (z) {
148     for (int i = n - 1; i >= 0; i--) {
149       B[i][m] = 0;
150       for (int j = m - 1; j >= 0; j--) {
151         decode(H[i][j].state[u]);
152         int cc = j == m - 1 ? c[j + 1] : c[j];
153         B[i][j] = cc ? 'o' : '#';
154         u = H[i][j].pre[u];
155       }
```

```
156     }
157     for (int i = 0; i < n; i++) cout << B[i] << '\n';
158   }
159   cout << '\n';
160 }
161
162 int main() {
163   cin.tie(nullptr)->sync_with_stdio(false);
164   int T;
165   cin >> T;
166   while (T--) {
167     init();
168     solve();
169     print();
170   }
171 }
```

习题

习题 「Topcoder SRM 312. Div1 Hard」 CheapestIsland[18]

题目大意：给一个棋盘图，每个格子有权值，求权值之和最小的连通块.

习题 「JLOI 2009」神秘的生物[19]

题目大意：给一个棋盘图，每个格子有权值，求权值之和最大的连通块.

习题 「AtCoder Beginner Contest 211. Problem E」 Red Polyomino[20]

题目大意：给一个 $N \times N$ 大小的棋盘图，每个格子初始为黑色或白色．你可以从白色格子中挑选恰好 K 个并将之染成红色，问有多少种染色方案满足红色格子形成一个连通块.

图论模型

例题 「NOI 2007 Day2」生成树计数[21]

题目大意：某类特殊图的生成树计数，每个节点恰好与其前 k 个节点之间有边相连.

例题 「2015 ACM-ICPC Asia Shenyang Regional Contest - Problem E」 Efficient Tree[22]

题目大意: 给出一个 $N \times M$ 的网格图, 以及相邻四连通格子之间的边权. 对于一颗生成树, 每个节点的得分为 1+[有一条连向上的边]+[有一条连向左的边]. 生成树的得分为所有节点的得分之积.

你需要求出：最小生成树的边权和，以及所有最小生成树的得分之和. （$n \leq 800, m \leq 7$）

实战篇

例题

例题 「HDU 4113」Construct the Great Wall[23]

题目大意：在 $N \times M$ 的棋盘内构造一组回路，分割所有的 `x` 和 `o`.

有一类插头 DP 问题要求我们在棋盘上构造一组墙，以分割棋盘上的某些元素. 不妨称之为修墙问题，这类问题既可视作染色模型，也可视作路径模型.

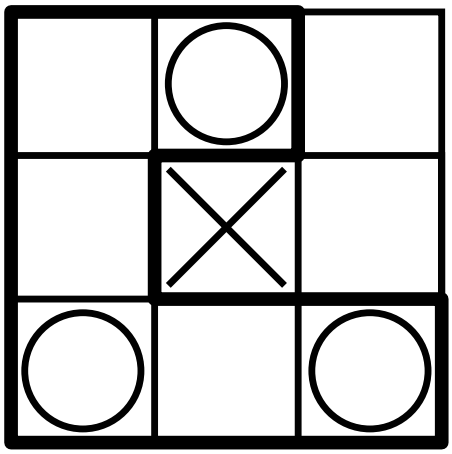


Figure.1

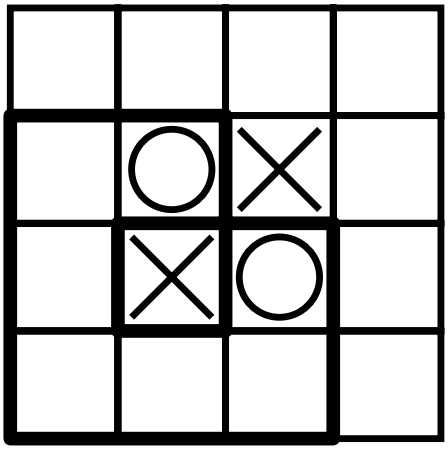


Figure.2

在本题中，如果视作染色模型的话，不仅需要额外讨论染色区域的周长，还要判断在角上触碰而导致不合法的情况（图 2）. 另外与 「UVa 10572」Black & White[24] 不同的是，本题中要求围墙为简单多边形，因而对于下面的回字形的情况，在本题中是不合法的.

```
3 3
ooo
oxo
ooo
```

因而我们使用路径模型，转化为 一条回路→7@p713 来处理.

我们沿着棋盘的交叉点进行 DP（因而长宽需要增加 1），每次转移时，需要保证所有的 `x` 在回路之外，`o` 在回路之内. 因此我们还需要维护当前位置是否在回路内部. 对于这个信息我们可以加维，也可以直接统计轮廓线上到这个位置之前出现下插头次数的奇偶性（射线法）.

例题代码

```cpp
#include <cstring>
#include <iostream>
using namespace std;
#define REP(i, n) for (int i = 0; i < n; ++i)

template <class T>
bool checkMin(T &a, const T b) {
  return b < a ? a = b, true : false;
}
```

```
10
11 constexpr int N = 10, M = N;
12 constexpr int offset = 3, mask = (1 << offset) - 1;
13 int n, m;
14 int d;
15 constexpr int INF = 0x3f3f3f3f;
16 int b[M + 1], bb[M + 1];
17
18 int encode() {
19   int s = 0;
20   memset(bb, -1, sizeof(bb));
21   int bn = 1;
22   bb[0] = 0;
23   for (int i = m; i >= 0; --i) {
24 #define bi bb[b[i]]
25     if (!~bi) bi = bn++;
26     s <<= offset;
27     s |= bi;
28   }
29   return s;
30 }
31
32 void decode(int s) {
33   REP(i, m + 1) {
34     b[i] = s & mask;
35     s >>= offset;
36   }
37 }
38
39 constexpr int MaxSZ = 16796, Prime = 9973;
40
41 struct hashTable {
42   int head[Prime], next[MaxSZ], sz;
43   int state[MaxSZ];
44   int key[MaxSZ];
45
46   void clear() {
47     sz = 0;
48     memset(head, -1, sizeof(head));
49   }
50
51   void push(int s) {
52     int x = s % Prime;
53     for (int i = head[x]; ~i; i = next[i]) {
54       if (state[i] == s) {
55         checkMin(key[i], d);
56         return;
57       }
58     }
59     state[sz] = s, key[sz] = d;
60     next[sz] = head[x];
61     head[x] = sz++;
```

```
  }

  void roll() { REP(i, sz) state[i] <<= offset; }
} H[2], *H0, *H1;

char A[N + 1][M + 1];

void push(int i, int j, int dn, int rt) {
  b[j] = dn;
  b[j + 1] = rt;
  if (A[i][j] != '.') {
    bool bad = A[i][j] == 'o';
    REP(jj, j + 1) if (b[jj]) bad ^= 1;
    if (bad) return;
  }
  H1->push(encode());
}

int solve() {
  cin >> n >> m;
  int ti, tj;
  REP(i, n) {
    scanf("%s", A[i]);
    REP(j, m) if (A[i][j] == 'o') ti = i, tj = j;
    A[i][m] = '.';
  }
  REP(j, m + 1) A[n][j] = '.';
  ++n, ++m, ++ti, ++tj;
  H0 = H, H1 = H + 1;
  H1->clear();
  d = 0;
  H1->push(0);
  int z = INF;
  REP(i, n) {
    REP(j, m) {
      swap(H0, H1);
      H1->clear();
      REP(ii, H0->sz) {
        decode(H0->state[ii]);
        d = H0->key[ii] + 1;
        int lt = b[j], up = b[j + 1];
        bool dn = i != n - 1, rt = j != m - 1;
        if (lt && up) {
          if (lt == up) {
            int cnt = 0;
            REP(i, m + 1) if (b[i])++ cnt;
            if (cnt == 2 && i == ti && j == tj) {
              checkMin(z, d);
            }
          } else {
            REP(i, m + 1) if (b[i] == lt) b[i] = up;
            push(i, j, 0, 0);
```

```
114           }
115         } else if (lt || up) {
116           int t = lt | up;
117           if (dn) {
118             push(i, j, t, 0);
119           }
120           if (rt) {
121             push(i, j, 0, t);
122           }
123         } else {
124           --d;
125           push(i, j, 0, 0);
126           ++d;
127           if (dn && rt) {
128             push(i, j, m, m);
129           }
130         }
131       }
132     }
133     H1->roll();
134   }
135   if (z == INF) z = -1;
136   return z;
137 }
138
139 int main() {
140   int T;
141   cin >> T;
142   for (int Case = 1; Case <= T; ++Case) {
143     printf("Case #%d: %d\n", Case, solve());
144   }
145 }
```

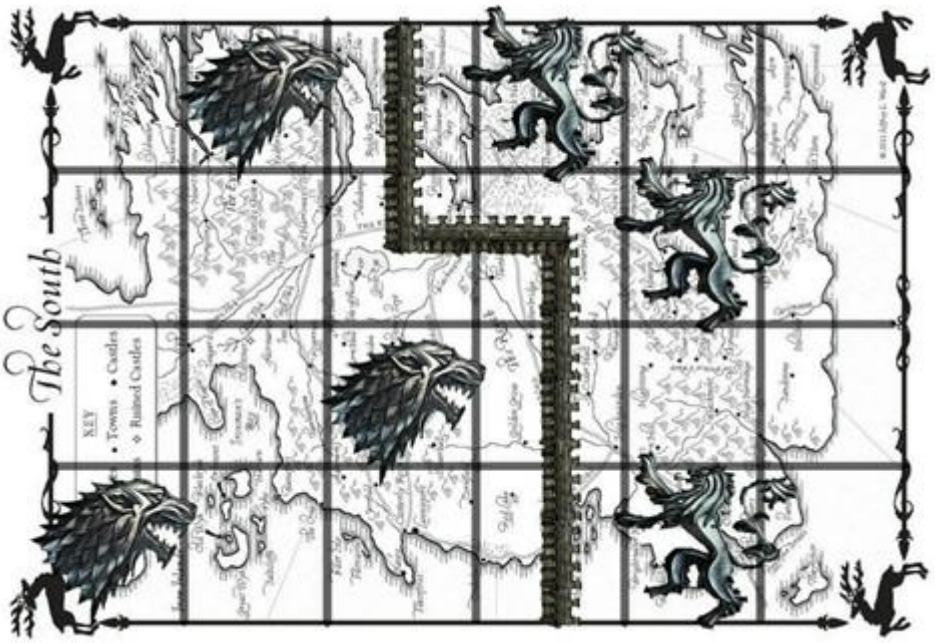
习题

习题 「SCOI 2011」地板[25]

题目大意：$r \times c$ 的棋盘上有一些位置设置障碍，问使用 L 型的瓷砖铺满所有没有障碍的格子，有多少种方案.

习题 「HDU 4796」Winter's Coming[26]

题目大意：在 $N \times M$ 的棋盘内对未染色的格点进行黑白灰染色，要求所有黑色区域和白色区域连通，且黑色区域与白色区域分别与棋盘的上下边界连通，且其中黑色区域与白色区域不能相邻. 每个格子有对应的代价，求一组染色方案，最小化灰色区域的代价.



习题 「ZOJ 2125」 Rocket Mania[27]

题目大意：9×6 的地图上每个格子里是一种管道（`-`,`T`,`L`,`+` 型或没有），可以把管道旋转 0°,90°,180°,270°, 问地图最多能有几行的右边界与第 X 行的左边界通过管道相连.

习题 「ZOJ 2126」 Rocket Mania Plus[28]

题目大意：9×6 的地图上每个格子里是一种管道（`-`,`T`,`L`,`+` 型或没有），可以把管道旋转 0°,90°,180°,270°, 问地图最多能有几行的右边界与左边界通过管道相连.

习题 「World Finals 2009/2010 Harbin」 Channel[29]

题目大意：一张方格地图上用 `.` 表示空地、`#` 表示石头，找到最长的一条路径满足：

1. 起点在左上角，终点在右下角.
2. 不能经过石头.
3. 路径自身不能在八连通的意义下成环.（即包括拐角处也不能接触）

习题 「HDU 3958」 Tower Defence[30]

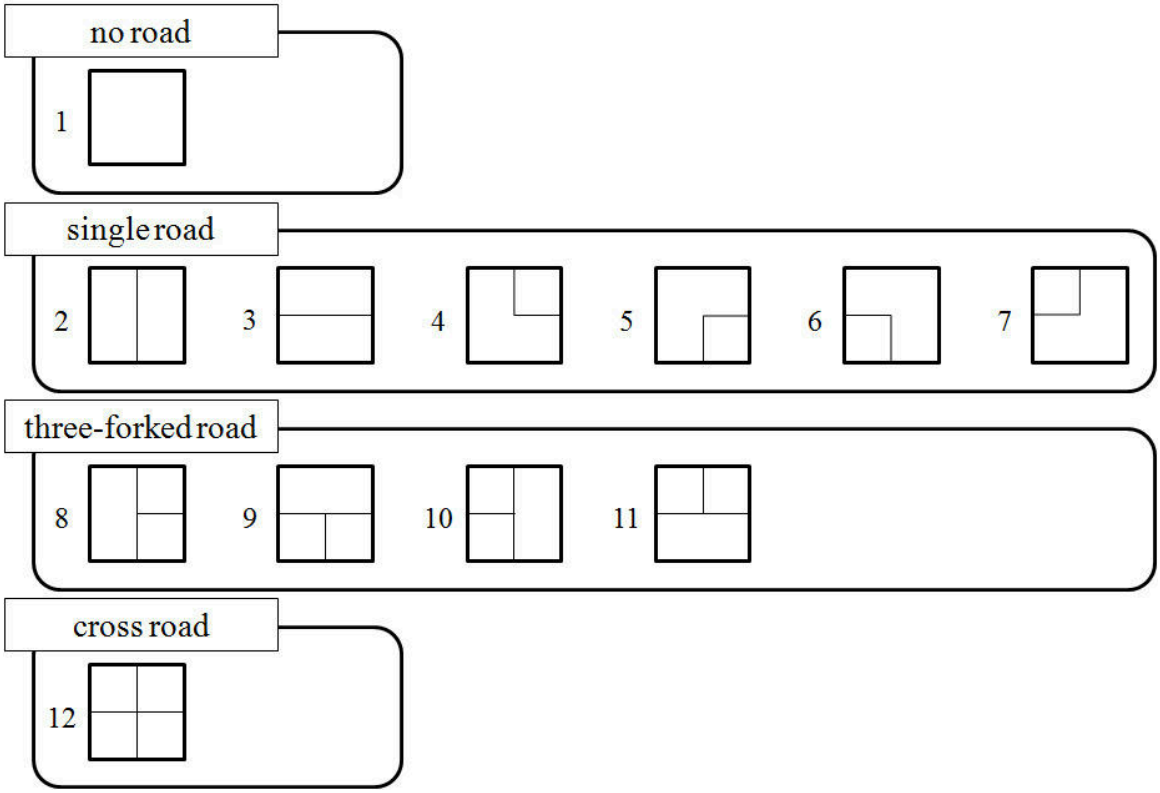
题目大意：可以转化为求解一条从 S 到 T 的不能接触的最长路径，拐角处可以接触.

习题 「UVa 10531」 Maze Statistics[31]

题目大意：有一个 $N \times M$ 的图，每个格子有独立概率 p 变成障碍物. 你要从迷宫左上角走到迷宫右下角. 求每个格子成为一个 **有解迷宫（即起点终点四联通）** 中的障碍物的概率.（$N \leq 5$, $M \leq 6$）

习题 「Aizu 2452」Pipeline Plans[32]

题目大意：现有一共 12 种图案的瓷砖，每种瓷砖数量给定．要求铺到一块可视为 $R \times C$ 网格图的矩形地板上，一个格子铺一块瓷砖，且左上角格子的中心与右下角格子的中心通过瓷砖图案上的线联通．$(2 \leq R \times C \leq 15)$



习题 「SDOI 2014」电路板[33]

题目大意：一块 $N \times M$ 的电路板，上面有些位置是电线不能走的障碍，给定 K 个格子对，要求每对格子都有电线相连，且电线之间互不相交（允许一条电路线从上边界进入当前格子，从左边界离开这个格子，另外一条电路线可以从下边界进入格子，从右边界出去）．视电线为无向边，求满足要求的最短电线长度和方案数．

习题 「SPOJ CAKE3」Delicious Cake[34]

题目大意：一块可视为 $N \times M$ 网格的蛋糕，现沿着格线将蛋糕切成数块，问有多少种不同的切割方法．切法相同当且仅当切成的每块蛋糕都形状相同且在同一位置上．$(\min(N, M) \leq 5, \max(N, M) \leq 130)$

本章注记

插头 DP 问题通常编码难度较大，讨论复杂，因而属于 OI/ACM 中相对较为偏门的领域[35]．这方面最为经典的资料，当属 2008 年 陈丹琦[36] 的集训队论文——基于连通性状态压缩的动态规划问题[37]．其次，HDU 的 notonlysuccess 2011 年曾经在博客中连续写过两篇由浅入深的专题，也是不可多得的好资料，不过现在需要在 Web Archive 里考古．

- notonlysuccess，【专辑】插头 DP[38]

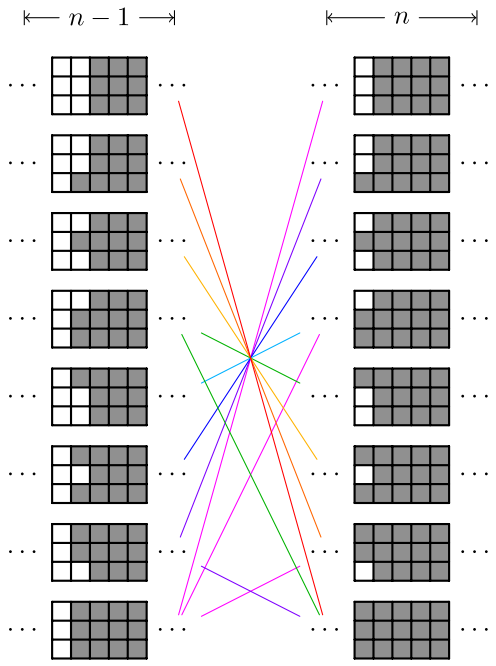
- notonlysuccess，【完全版】插头 DP[39]

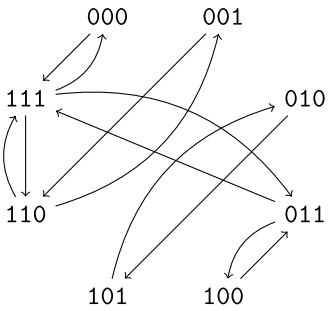
多米诺骨牌覆盖

「HDU 1400」Mondriaan's Dream[40] 也出现在《算法竞赛入门经典训练指南》→2.5@p96 中，并作为《轮廓线上的动态规划》一节的例题．多米诺骨牌覆盖（Domino tiling）[41] 是一组非常经典的数学问题，稍微修改其数据范围就可以得到不同难度，需要应用不同的算法解决的子问题．

当限定 $m = 2$ 时，多米诺骨牌覆盖等价于斐波那契数列．《具体数学》[42] 中使用了该问题以引出斐波那契数列，并使用了多种方法得到其解析解．

当 $m \leq 10, n \leq 10^9$ 时，可以将转移方程预处理成矩阵形式，并使用 矩阵乘法进行加速[43]．





当 $n, m \leq 100$，可以用 FKT Algorithm[44] 计算其所对应平面图的完美匹配数．

- 「51nod 1031」骨牌覆盖[45]
- 「51nod 1033」骨牌覆盖 V2[46]|「Vijos 1194」Domino[47]
- 「51nod 1034」骨牌覆盖 V3[48]|「Ural 1594」Aztec Treasure[49]
- Wolfram MathWorld, Chebyshev Polynomial of the Second Kind[50]

一条路径

「一条路径」是 哈密顿路径（Hamiltonian Path）[51] 问题在 格点图（Grid Graph）[52] 中的一种特殊情况．哈密顿路径的判定性问题是 NP-complete[53] 家族中的重要成员．

外部参考资料

[1] 「HDU 1400」Mondriaan's Dream

[2] 「SRM 671. Div 1 900」BearDestroys

[3] 「HDU 1693」Eat the Trees

[4] 「ZOJ 3466」The Hive II

[5] 「Andrew Stankevich Contest 16 - Problem F」Pipe Layout

[6] std::unordered_map

[7] 「Ural 1519」Formula 1

[8] 「USACO 5.4.4」Betsy's Tours

[9] 「POJ 1739」Tony's Tour

[10]「USACO 6.1.1」Postal Vans 

[11]「HNOI 2007」神奇游乐园 

[12]「ProjectEuler 393」Migrating ants 

[13]「ZOJ 3213」Beautiful Meadow 

[14]「BZOJ 2310」ParkII 

[15]「NOI 2010 Day2」旅行路线 

[16]「UVa 10572」Black & White 

[17] SGU 197. Nice Patterns Strike Back 

[18]「Topcoder SRM 312. Div1 Hard」CheapestIsland 

[19]「JLOI 2009」神秘的生物 

[20]「AtCoder Beginner Contest 211. Problem E」Red Polyomino 

[21]「NOI 2007 Day2」生成树计数 

[22]「2015 ACM-ICPC Asia Shenyang Regional Contest - Problem E」Efficient Tree 

[23]「HDU 4113」Construct the Great Wall 

[24]「UVa 10572」Black & White 

[25]「SCOI 2011」地板 

[26]「HDU 4796」Winter's Coming 

[27]「ZOJ 2125」Rocket Mania 

[28]「ZOJ 2126」Rocket Mania Plus 

[29]「World Finals 2009/2010 Harbin」Channel 

[30]「HDU 3958」Tower Defence 

[31]「UVa 10531」Maze Statistics 

[32]「Aizu 2452」Pipeline Plans 

[33]「SDOI 2014」电路板 

[34]「SPOJ CAKE3」Delicious Cake 

[35] 偏门的领域 

[36] 陈丹琦 

[37] 基于连通性状态压缩的动态规划问题 

[38] notonlysuccess,【专辑】插头 DP 

[39] notonlysuccess,【完全版】插头 DP 

[40]「HDU 1400」Mondriaan's Dream 

[41] 多米诺骨牌覆盖（Domino tiling） 

[42]《具体数学》 

[43] 矩阵乘法进行加速 

[44] FKT Algorithm 

[45]「51nod 1031」骨牌覆盖 

[46]「51nod 1033」骨牌覆盖 V2 

[47]「Vijos 1194」Domino 

[48]「51nod 1034」骨牌覆盖 V3 

[49]「Ural 1594」Aztec Treasure 

[50] Wolfram MathWorld, Chebyshev Polynomial of the Second Kind

[51] 哈密顿路径（Hamiltonian Path）

[52] 格点图（Grid Graph）

[53] NP-complete

7.11 计数 DP

计数 DP 是一种利用类似 DP 的记忆化搜索方法（与在狭义上的 DP，即最优化问题有一定区别），用于解决计数（以及求和）问题.

基础

基本思想

计数问题一般指求一个集合 S 的大小，在 OI 中，S 的大小有时会达到 $\Theta(n^n)$ 甚至 $\Theta(2^{n!})$ 的级别（当然，一般会对某一个固定的数取模），其中 n 是问题规模，所以我们不能逐一求出 S 的元素.

如果我们能够将 S 分成若干无交的子集，那么 S 的元素个数就等于这些部分的元素个数的和. 如果这些子集的计数恰好与原问题类似，那么我们就可以通过类似动态规划的方法来解决.

例题

例题

给定一个正整数 n，求有多少个把 n 划分成 k 个正整数的和的方案，位置调换视为不同的划分方案.

需集合 $S_{n,k}$ 为形如 $(a_1,...,a_k)$ 的正整数组组成的集合，其中 $a_1+...+a_k=n$. 如果 a_k 固定，则有如下推导：因为 $a_1+a_2+...+a_{k-1}+a_k=n$，所以 $a_1+a_2+...+a_{k-1}=n-a_k$. 根据 $S_{n,k}$ 的定义，$(a_1,a_2,...,a_{k-1})\in S_{n-a_k,k-1}$.

由于 $a_1,a_2,...,a_k$ 是正整数，所以 a_k 的取值范围是 $[1,n-k+1]\cap\mathbb{Z}$. 因此，$S_{n,k}$ 可以按照 a_k 被划分，分成 $n-k+1$ 个子集，其中当 $a_k=i$ 时，这个子集为：

$$\{(L,i)\mid L\in S_{n-i,k-1}\}.$$

这个子集的元素个数显然等于 $S_{n-i,k-1}$，由于 i 的不同，这些子集两两无交. 所以：

$$|S_{n,k}|=\sum_{i=1}^{n-k+1}|S_{n-i,k-1}|.$$

这样我们就可以使用类似 DP 的方法处理它：设 $f_{n,k}$ 为 $|S_{n,k}|$，则有状态转移方程：

$$f_{n,k}=\sum_{i=1}^{n-k+1}f_{n-i,k-1}.$$

这样就可以使用 DP 的方法求解了.

与最优化 DP 的异同

可以发现，计数 DP 和最优化 DP 都是在一个范围 Ω 内求一个值（大小值、最优值），这个值通过将 Ω 中的所有元素做一次处理，再对处理值做一次整合得到.

例如，对于 0-1 背包问题，Ω 中的元素为背包内的所有物品组成的集合，对于 Ω 中的一个方案 S，我们对 S 做一次处理，处理得到的结果 $w(S)$ 为 S 中物品的总价值，对所有得到的处理值，我们取最大值，得到问题的答案.

对于计数问题，Ω 中的元素为要计算元素个数的集合 S，它的处理是把所有的 S 中元素变为 1，然后将这些 1 通过加法的方式汇总起来，因为每一个 S 中元素都对应一个 1，所以这样得到的值就是 S 中元素个数.

当汇总操作为最大/最小值时，我们可以将 Ω 分成任意若干个部分，只需这些部分的并为 Ω 即可，无需无交的条件. 而计数问题由于不满足这个条件，所以我们需要将 Ω 分成若干个部分，这些部分两两无交，这就是与最优化 DP 的区别.

例题

例题

给定一个正整数 n，求有多少个把 n 划分成任意多个正整数的和的方案，位置调换视为 **相同** 的划分方案.

解法 1

需要计算的集合的元素为满足其和为 n 的正整数多重集. 但是这样显然不好推.

若一个多重集 T 只包含 $\le M$ 的正整数，且 T 中所有元素的和为 n，则称 $T\in S_{n,M}$. 考虑 M 出现的个数. 可能为 $k\in\left[0,\left\lfloor\frac{n}{M}\right\rfloor\right]\cap\mathbb{Z}$. 于是它可以被转移到 $S_{n-kM,M-1}$. 求和一下即可. 复杂度是 $\Theta(n^2\log n)$（log 来自于 k 的范围导致的调和级数）.

但是这样还不够优秀．考虑下面所示的一个例子：

$$f_{8,3} = f_{8,2} + f_{5,2} + f_{2,2}$$
$$f_{9,3} = f_{9,2} + f_{6,2} + f_{3,2} + f_{0,2}$$
$$f_{10,3} = f_{10,2} + f_{7,2} + f_{4,2} + f_{1,2}$$
$$f_{11,3} = f_{11,2} + f_{8,2} + f_{5,2} + f_{2,2}$$
$$f_{12,3} = f_{12,2} + f_{9,2} + f_{6,2} + f_{3,2} + f_{0,2}$$
$$f_{13,3} = f_{13,2} + f_{10,2} + f_{7,2} + f_{4,2} + f_{1,2}$$

等量代换得 $f_{11,3} = f_{11,2} + f_{8,3}$，$f_{12,3} = f_{12,2} + f_{9,3}$，$f_{13,3} = f_{13,2} + f_{10,3}$．同理我们可以得到一个通用的状态转移方程：

$$f_{n,M} = f_{n,M-1} + \begin{cases} f_{n-M,M} & n \geq M, \\ 0 & \text{otherwise.} \end{cases}$$

此时，时间复杂度为 $\Theta(n^2)$．

解法 2

考虑到某一个正整数组成的多重集 T 必然可以通过「将 T 中每一个元素自增」、「在 T 中加一个值为 1 的元素」两个操作得到，并且不同的操作序列得到的结果是不同的．

这样对 T 的转移可以变为对操作序列的转移．考虑将 n 划分成 m 个数的操作序列（所有的这些操作序列记作 $B_{n,m}$）中的最后一次操作，如果是 1 操作，那么不会增加数，但是 $\sum T$ 增加了 m．为了使最终的 $\sum T = n$，原来的 T（记作 T'）的和需要为 $n - m$．所以 $B_{n,m} \to B_{n-m,m}$；如果是 2 操作，那么会增加一个数，$\sum T$ 增加了 1．所以 $B_{n,m} \to B_{n-1,m-1}$．

这样做的时间复杂度依旧是 $\Theta(n^2)$．

解法 3

考虑将 T 分为大于 $\sqrt{n}$ 的部分 T_1 和小于等于 $\sqrt{n}$ 的部分 T_2．T_2 可以使用解法 1 求出，而 T_1 的数量可以通过略微修改解法 2 求出：考虑将两个操作变为「将 T_1 中每一个元素自增」、「在 T_1 中加一个值为 $\lfloor\sqrt{n}\rfloor + 1$ 的元素」．容易列出状态转移方程．

将 n 拆为 A 和 B 两部分．枚举其中一个即可得出另一个．将满足 $\sum T_1 = A$ 的 T_1 个数和 $\sum T_2 = B$ 的 T_2 个数求出，乘起来，对所有的 A 求和便是最终结果．

由于在计算 T_1 个数的过程中，$M \leq \sqrt{n}$，所以我们利用解法 1 计算 T_1 的时间复杂度为 $\Theta(n^{3/2})$．同样地，由于在计算 T_2 个数的过程中，$|T_2| \leq \dfrac{\sum T_2}{\sqrt{n}} \leq \dfrac{n}{\sqrt{n}} = \sqrt{n}$，所以我们利用解法 2 计算 T_2 的时间复杂度也是 $\Theta(n^{3/2})$．所以总时间复杂度为 $\Theta(n^{3/2})$．

7.12 动态 DP

前置知识：矩阵→9.14.4@p1716，树链剖分→11.6.6@p2658．

动态 DP 问题是猫锟在 WC2018 讲的黑科技，一般用来解决树上的带有点权（边权）修改操作的 DP 问题．

例子

以这道模板题为例子讲解一下动态 DP 的过程．

例题 洛谷 P4719【模板】动态 DP[1]

给定一棵 n 个点的树，点带点权．有 m 次操作，每次操作给定 x, y 表示修改点 x 的权值为 y．你需要在每次操作之后求出这棵树的最大权独立集的权值大小．

广义矩阵乘法

定义广义矩阵乘法 $A \times B = C$ 为：

$$C_{i,j} = \max_{k=1}^{n}(A_{i,k} + B_{k,j})$$

相当于将普通的矩阵乘法中的乘变为加，加变为 max 操作．

同时广义矩阵乘法满足结合律，所以可以使用矩阵快速幂．

不带修改操作

令 $f_{i,0}$ 表示不选择 i 的最大答案，$f_{i,1}$ 表示选择 i 的最大答案．

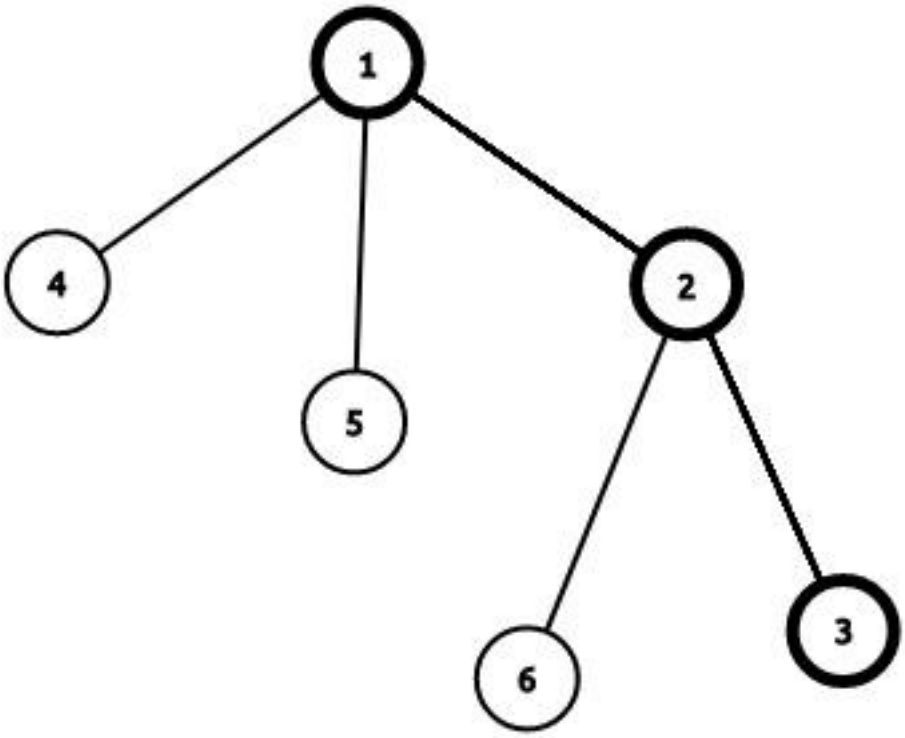
则有 DP 方程：

$$\begin{cases} f_{i,0} = \sum_{son} \max(f_{son,0}, f_{son,1}) \\ f_{i,1} = w_i + \sum_{son} f_{son,0} \end{cases}$$

答案就是 $\max(f_{root,0}, f_{root,1})$．

带修改操作

首先将这棵树进行树链剖分，假设有这样一条重链：



设 $g_{i,0}$ 表示不选择 i 且只允许选择 i 的轻儿子所在子树的最大答案，$g_{i,1}$ 表示不考虑 son_i 的情况下选择 i 的最大答案，son_i 表示 i 的重儿子．

假设我们已知 $g_{i,0/1}$ 那么有 DP 方程：

$$\begin{cases} f_{i,0} = g_{i,0} + \max(f_{son_i,0}, f_{son_i,1}) \\ f_{i,1} = g_{i,1} + f_{son_i,0} \end{cases}$$

答案是 $\max(f_{root,0}, f_{root,1})$.

可以构造出矩阵：

$$\begin{bmatrix} g_{i,0} & g_{i,0} \\ g_{i,1} & -\infty \end{bmatrix} \times \begin{bmatrix} f_{son_i,0} \\ f_{son_i,1} \end{bmatrix} = \begin{bmatrix} f_{i,0} \\ f_{i,1} \end{bmatrix}$$

注意，我们这里使用的是广义乘法规则.

可以发现，修改操作时只需要修改 $g_{i,1}$ 和每条往上的重链即可.

具体思路

1. DFS 预处理求出 $f_{i,0/1}$ 和 $g_{i,0/1}$.
2. 对这棵树进行树链剖分（注意，因为我们对一个点进行询问需要计算从该点到该点所在的重链末尾的区间矩阵乘，所以对于每一个点记录 End_i 表示 i 所在的重链末尾节点编号），每一条重链建立线段树，线段树维护 g 矩阵和 g 矩阵区间乘积.
3. 修改时首先修改 $g_{i,1}$ 和线段树中 i 节点的矩阵，计算 top_i 矩阵的变化量，修改到 fa_{top_i} 矩阵.
4. 查询时就是 1 到其所在的重链末尾的区间乘，最后取一个 max 即可.

代码实现

```cpp
#include <algorithm>
#include <cstring>
#include <iostream>

using namespace std;

constexpr int MAXN = 500010;
constexpr int INF = 0x3f3f3f3f;

int Begin[MAXN], Next[MAXN], To[MAXN], e, n, m;
int sz[MAXN], son[MAXN], top[MAXN], fa[MAXN], dis[MAXN], p[MAXN], id[MAXN],
    End[MAXN];
// p[i]表示i树剖后的编号，id[p[i]] = i
int cnt, tot, a[MAXN], f[MAXN][2];

struct matrix {
  int g[2][2];

  matrix() { memset(g, 0, sizeof(g)); }

  matrix operator*(const matrix &b) const  // 重载矩阵乘
  {
    matrix c;
    for (int i = 0; i <= 1; i++)
      for (int j = 0; j <= 1; j++)
        for (int k = 0; k <= 1; k++)
          c.g[i][j] = max(c.g[i][j], g[i][k] + b.g[k][j]);
    return c;
  }
} Tree[MAXN], g[MAXN];  // Tree[]是建出来的线段树，g[]是维护的每个点的矩阵
```

```
31
32 void PushUp(int root) { Tree[root] = Tree[root << 1] * Tree[root << 1 |
   1]; }
33
34 void Build(int root, int l, int r) {
35   if (l == r) {
36     Tree[root] = g[id[l]];
37     return;
38   }
39   int Mid = (l + r) >> 1;
40   Build(root << 1, l, Mid);
41   Build(root << 1 | 1, Mid + 1, r);
42   PushUp(root);
43 }
44
45 matrix Query(int root, int l, int r, int L, int R) {
46   if (L <= l && r <= R) return Tree[root];
47   int Mid = (l + r) >> 1;
48   if (R <= Mid) return Query(root << 1, l, Mid, L, R);
49   if (Mid < L) return Query(root << 1 | 1, Mid + 1, r, L, R);
50   return Query(root << 1, l, Mid, L, R) *
51          Query(root << 1 | 1, Mid + 1, r, L, R);
52   // 注意查询操作的书写
53 }
54
55 void Modify(int root, int l, int r, int pos) {
56   if (l == r) {
57     Tree[root] = g[id[l]];
58     return;
59   }
60   int Mid = (l + r) >> 1;
61   if (pos <= Mid)
62     Modify(root << 1, l, Mid, pos);
63   else
64     Modify(root << 1 | 1, Mid + 1, r, pos);
65   PushUp(root);
66 }
67
68 void Update(int x, int val) {
69   g[x].g[1][0] += val - a[x];
70   a[x] = val;
71   // 首先修改x的g矩阵
72   while (x) {
73     matrix last = Query(1, 1, n, p[top[x]], End[top[x]]);
74     // 查询top[x]的原本g矩阵
75     Modify(1, 1, n,
76            p[x]);  // 进行修改(x点的g矩阵已经进行修改但线段树上的未进行修改)
77     matrix now = Query(1, 1, n, p[top[x]], End[top[x]]);
78     // 查询top[x]的新g矩阵
79     x = fa[top[x]];
80     g[x].g[0][0] +=
81         max(now.g[0][0], now.g[1][0]) - max(last.g[0][0], last.g[1][0]);
```

```
82      g[x].g[0][1] = g[x].g[0][0];
83      g[x].g[1][0] += now.g[0][0] - last.g[0][0];
84      // 根据变化量修改fa[top[x]]的g矩阵
85    }
86  }
87
88  void add(int u, int v) {
89    To[++e] = v;
90    Next[e] = Begin[u];
91    Begin[u] = e;
92  }
93
94  void DFS1(int u) {
95    sz[u] = 1;
96    int Max = 0;
97    f[u][1] = a[u];
98    for (int i = Begin[u]; i; i = Next[i]) {
99      int v = To[i];
100     if (v == fa[u]) continue;
101     dis[v] = dis[u] + 1;
102     fa[v] = u;
103     DFS1(v);
104     sz[u] += sz[v];
105     if (sz[v] > Max) {
106       Max = sz[v];
107       son[u] = v;
108     }
109     f[u][1] += f[v][0];
110     f[u][0] += max(f[v][0], f[v][1]);
111     // DFS1过程中同时求出f[i][0/1]
112   }
113 }
114
115 void DFS2(int u, int t) {
116   top[u] = t;
117   p[u] = ++cnt;
118   id[cnt] = u;
119   End[t] = cnt;
120   g[u].g[1][0] = a[u];
121   g[u].g[1][1] = -INF;
122   if (!son[u]) return;
123   DFS2(son[u], t);
124   for (int i = Begin[u]; i; i = Next[i]) {
125     int v = To[i];
126     if (v == fa[u] || v == son[u]) continue;
127     DFS2(v, v);
128     g[u].g[0][0] += max(f[v][0], f[v][1]);
129     g[u].g[1][0] += f[v][0];
130     // g矩阵根据f[i][0/1]求出
131   }
132   g[u].g[0][1] = g[u].g[0][0];
133 }
```

```
134
135 int main() {
136   cin.tie(nullptr)->sync_with_stdio(false);
137   cin >> n >> m;
138   for (int i = 1; i <= n; i++) cin >> a[i];
139   for (int i = 1; i <= n - 1; i++) {
140     int u, v;
141     cin >> u >> v;
142     add(u, v);
143     add(v, u);
144   }
145   dis[1] = 1;
146   DFS1(1);
147   DFS2(1, 1);
148   Build(1, 1, n);
149   for (int i = 1; i <= m; i++) {
150     int x, val;
151     cin >> x >> val;
152     Update(x, val);
153     matrix ans = Query(1, 1, n, 1, End[1]);  // 查询1所在重链的矩阵乘
154     cout << max(ans.g[0][0], ans.g[1][0]) << '\n';
155   }
156   return 0;
157 }
158
```

习题

- SPOJ GSS3 - Can you answer these queries III[2]
- 「NOIP2018」保卫王国[3]
- 「SDOI2017」切树游戏[4]

外部参考资料

[1] 洛谷 P4719【模板】动态 DP 

[2] SPOJ GSS3 - Can you answer these queries III 

[3] 「NOIP2018」保卫王国 

[4] 「SDOI2017」切树游戏 

7.13 概率 DP

Authors: Tiphereth-A, ShaoChenHeng, Enter-tainer, ksyx, c-forrest, StudyingFather, H-J-Granger, iamtwz, imp2002, Ir1d, kenlig, LeBronGod, Marcythm, MegaOwIer, NachtgeistW, ouuan, Patchouliys, Soohti, TianKong-y, sun2snow

引入

概率 DP 用于解决概率问题与期望问题，建议先对 概率 & 期望→9.17.4@p1894 的内容有一定了解．一般情况下，解决概率问题需要顺序循环，而解决期望问题使用逆序循环．如果定义的状态转移方程存在后效性问题，还需要用到 高斯消元→9.19.3@p1944 来优化．概率 DP 也会结合其他知识进行考察，例如 状态压缩→7.8@p744、树上进行 DP 转移等．

概率 DP

这类题目采用顺推，也就是从初始状态推向结果．同一般的 DP 类似，难点依然是对状态转移方程的刻画，只是这类题目经过了概率论知识的包装．

例题

Codeforces 148D Bag of mice[1]

袋子里有 w 只白鼠和 b 只黑鼠，公主和龙轮流从袋子里抓老鼠．谁先抓到白色老鼠谁就赢，如果袋子里没有老鼠了并且没有谁抓到白色老鼠，那么算龙赢．公主每次抓一只老鼠，龙每次抓完一只老鼠之后会有一只老鼠跑出来．每次抓的老鼠和跑出来的老鼠都是随机的．公主先抓．问公主赢的概率．

解答

设 $f_{i,j}$ 为轮到公主时袋子里有 i 只白鼠，j 只黑鼠，公主赢的概率．初始化边界，$f_{0,j}=0$ 因为没有白鼠了算龙赢，$f_{i,0}=1$ 因为抓一只就是白鼠，公主赢． 考虑 $f_{i,j}$ 的转移：

- 公主抓到一只白鼠，公主赢了．概率为 $\frac{i}{i+j}$．
- 公主抓到一只黑鼠，龙抓到一只白鼠，龙赢了．概率为 $\frac{j}{i+j}\cdot\frac{i}{i+j-1}$．
- 公主抓到一只黑鼠，龙抓到一只黑鼠，跑出来一只黑鼠，转移到 $f_{i,j-3}$．概率为 $\frac{j}{i+j}\cdot\frac{j-1}{i+j-1}\cdot\frac{j-2}{i+j-2}$．
- 公主抓到一只黑鼠，龙抓到一只黑鼠，跑出来一只白鼠，转移到 $f_{i-1,j-2}$．概率为 $\frac{j}{i+j}\cdot\frac{j-1}{i+j-1}\cdot\frac{i}{i+j-2}$．

考虑公主赢的概率，第二种情况不参与计算．并且要保证后两种情况合法，所以还要判断 i,j 的大小，满足第三种情况至少要有 3 只黑鼠，满足第四种情况要有 1 只白鼠和 2 只黑鼠．

参考代码

```cpp
#include <cstring>
#include <iomanip>
#include <iostream>
using namespace std;

using ll = long long;
int w, b;
double dp[1010][1010];

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  cin >> w >> b;
  memset(dp, 0, sizeof(dp));
  for (int i = 1; i <= w; i++) dp[i][0] = 1;  // 初始化
  for (int i = 1; i <= b; i++) dp[0][i] = 0;
  for (int i = 1; i <= w; i++) {
    for (int j = 1; j <= b; j++) {  // 以下为题面概率转移
      dp[i][j] += (double)i / (i + j);
      if (j >= 3) {
        dp[i][j] += (double)j / (i + j) * (j - 1) / (i + j - 1) * (j - 2) /
                    (i + j - 2) * dp[i][j - 3];
```

```
      }
      if (i >= 1 && j >= 2) {
        dp[i][j] += (double)j / (i + j) * (j - 1) / (i + j - 1) * i /
                    (i + j - 2) * dp[i - 1][j - 2];
      }
    }
  }
  cout << fixed << setprecision(9) << dp[w][b] << '\n';
  return 0;
}
```

习题

- POJ3071 Football[2]
- CodeForces 768D Jon and Orbs[3]

期望 DP

例题

POJ2096 Collecting Bugs[4]

一个软件有 s 个子系统，会产生 n 种 bug．某人一天发现一个 bug，这个 bug 属于某种 bug 分类，也属于某个子系统．每个 bug 属于某个子系统的概率是 $\frac{1}{s}$，属于某种 bug 分类的概率是 $\frac{1}{n}$．求发现 n 种 bug，且 s 个子系统都找到 bug 的期望天数．

解答

令 $f_{i,j}$ 为已经找到 i 种 bug 分类，j 个子系统的 bug，达到目标状态的期望天数．这里的目标状态是找到 n 种 bug 分类，s 个子系统的 bug．那么就有 $f_{n,s}=0$，因为已经达到了目标状态，不需要用更多的天数去发现 bug 了，于是就以目标状态为起点开始递推，答案是 $f_{0,0}$．

考虑 $f_{i,j}$ 的状态转移：

- $f_{i,j}$，发现一个 bug 属于已经发现的 i 种 bug 分类，j 个子系统，概率为 $p_1=\frac{i}{n}\cdot\frac{j}{s}$．
- $f_{i,j+1}$，发现一个 bug 属于已经发现的 i 种 bug 分类，不属于已经发现的子系统，概率为 $p_2=\frac{i}{n}\cdot(1-\frac{j}{s})$．
- $f_{i+1,j}$，发现一个 bug 不属于已经发现 bug 分类，属于 j 个子系统，概率为 $p_3=(1-\frac{i}{n})\cdot\frac{j}{s}$．
- $f_{i+1,j+1}$，发现一个 bug 不属于已经发现 bug 分类，不属于已经发现的子系统，概率为 $p_4=(1-\frac{i}{n})\cdot(1-\frac{j}{s})$．

再根据期望的线性性质，就可以得到状态转移方程：

$$f_{i,j}=p_1\cdot f_{i,j}+p_2\cdot f_{i,j+1}+p_3\cdot f_{i+1,j}+p_4\cdot f_{i+1,j+1}+1$$

$$=\frac{p_2\cdot f_{i,j+1}+p_3\cdot f_{i+1,j}+p_4\cdot f_{i+1,j+1}+1}{1-p_1}$$

参考代码

```
#include <iomanip>
#include <iostream>
```

```cpp
using namespace std;
int n, s;
double dp[1010][1010];

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  cin >> n >> s;
  dp[n][s] = 0;
  for (int i = n; i >= 0; i--) {
    for (int j = s; j >= 0; j--) {
      if (i == n && s == j) continue;
      dp[i][j] = (dp[i][j + 1] * i * (s - j) + dp[i + 1][j] * (n - i) * j +
                  dp[i + 1][j + 1] * (n - i) * (s - j) + n * s) /
                 (n * s - i * j);  // 概率转移
    }
  }
  cout << fixed << setprecision(4) << dp[0][0] << '\n';
  return 0;
}
```

「NOIP2016」换教室[5]

牛牛要上 n 个时间段的课，第 i 个时间段在 c_i 号教室，可以申请换到 d_i 号教室，申请成功的概率为 p_i，至多可以申请 m 节课进行交换．第 i 个时间段的课上完后要走到第 $i+1$ 个时间段的教室，给出一张图 v 个教室 e 条路，移动会消耗体力，申请哪几门课程可以使他因在教室间移动耗费的体力值的总和的期望值最小，也就是求出最小的期望路程和．

解答

对于这个无向连通图，先用 Floyd 求出最短路，为后续的状态转移带来便利．以移动一步为一个阶段（从第 i 个时间段到达第 $i+1$ 个时间段就是移动了一步），那么每一步就有 p_i 的概率到 d_i，不过在所有的 d_i 中只能选 m 个，有 $1-p_i$ 的概率到 c_i，求出在 n 个阶段走完后的最小期望路程和．

定义 $f_{i,j,0/1}$ 为在第 i 个时间段，连同这一个时间段已经用了 j 次换教室的机会，在这个时间段换（1）或者不换（0）教室的最小期望路程和，那么答案就是 $\min\{f_{n,i,0}, f_{n,i,1}\}, i \in [0,m]$．注意边界 $f_{1,0,0} = f_{1,1,1} = 0$．

考虑 $f_{i,j,0/1}$ 的状态转移：

- 如果这一阶段不换，即 $f_{i,j,0}$．可能是由上一次不换的状态转移来的，那么就是 $f_{i-1,j,0} + w_{c_{i-1},c_i}$，也有可能是由上一次交换的状态转移来的，这里结合条件概率和全概率的知识分析可以得到 $f_{i-1,j,1} + w_{d_{i-1},c_i} \cdot p_{i-1} + w_{c_{i-1},c_i} \cdot (1-p_{i-1})$，状态转移方程就有：

$$f_{i,j,0} = min(f_{i-1,j,0} + w_{c_{i-1},c_i}, f_{i-1,j,1} + w_{d_{i-1},c_i} \cdot p_{i-1} + w_{c_{i-1},c_i} \cdot (1-p_{i-1}))$$

- 如果这一阶段交换，即 $f_{i,j,1}$．类似地，可能由上一次不换的状态转移来，也可能由上一次交换的状态转移来．那么遇到不换的就乘上 $(1-p_i)$，遇到交换的就乘上 p_i，将所有会出现的情况都枚举一遍出进行计算就好了．这里不再赘述各种转移情况，相信通过上一种阶段例子，这里的状态转移应该能够很容易写出来．

参考代码

```cpp
#include <algorithm>
#include <iomanip>
#include <iostream>

using namespace std;

constexpr int MAXN = 2010;
int n, m, v, e;
int f[MAXN][MAXN], c[MAXN], d[MAXN];
double dp[MAXN][MAXN][2], p[MAXN];

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  cin >> n >> m >> v >> e;
  for (int i = 1; i <= n; i++) cin >> c[i];
  for (int i = 1; i <= n; i++) cin >> d[i];
  for (int i = 1; i <= n; i++) cin >> p[i];
  for (int i = 1; i <= v; i++)
    for (int j = 1; j < i; j++) f[i][j] = f[j][i] = 1e9;

  int u, V, w;
  for (int i = 1; i <= e; i++) {
    cin >> u >> V >> w;
    f[u][V] = f[V][u] = min(w, f[u][V]);
  }

  for (int k = 1; k <= v; k++)
    for (int i = 1; i <= v; i++)  // 前面的，按照前面的题解进行一个状态转移
      for (int j = 1; j < i; j++)
        if (f[i][k] + f[k][j] < f[i][j]) f[i][j] = f[j][i] = f[i][k] +
f[k][j];

  for (int i = 1; i <= n; i++)
    for (int j = 0; j <= m; j++) dp[i][j][0] = dp[i][j][1] = 1e9;

  dp[1][0][0] = dp[1][1][1] = 0;
  for (int i = 2; i <= n; i++)  // 有后效性方程
    for (int j = 0; j <= min(i, m); j++) {
      dp[i][j][0] = min(dp[i - 1][j][0] + f[c[i - 1]][c[i]],
                        dp[i - 1][j][1] + f[c[i - 1]][c[i]] * (1 - p[i -
1]) +
                            f[d[i - 1]][c[i]] * p[i - 1]);
      if (j != 0) {
        dp[i][j][1] = min(dp[i - 1][j - 1][0] + f[c[i - 1]][d[i]] * p[i] +
                              f[c[i - 1]][c[i]] * (1 - p[i]),
                          dp[i - 1][j - 1][1] +
                              f[c[i - 1]][c[i]] * (1 - p[i - 1]) * (1 -
p[i]) +
                              f[c[i - 1]][d[i]] * (1 - p[i - 1]) * p[i] +
                              f[d[i - 1]][c[i]] * (1 - p[i]) * p[i - 1] +
                              f[d[i - 1]][d[i]] * p[i - 1] * p[i]);
      }
```

```
    }

  double ans = 1e9;
  for (int i = 0; i <= m; i++) ans = min(dp[n][i][0], min(dp[n][i][1],
ans));
  cout << fixed << setprecision(2) << ans;

  return 0;
}
```

比较这两个问题可以发现，DP 求期望题目在对具体是求一个值或是最优化问题上会对方程得到转移方式有一些影响，但无论是 DP 求概率还是 DP 求期望，总是离不开概率知识和列出、化简计算公式的步骤，在写状态转移方程时需要思考的细节也类似.

习题

- HDU3853 LOOPS[6]
- HDU4035 Maze[7]
- 「SCOI2008」奖励关[8]

有后效性 DP

例题

CodeForces 24D Broken robot[9]

给出一个 $n \times m$ 的矩阵区域. 一个机器人初始在第 x 行第 y 列，每一步机器人会等概率地选择停在原地、左移一步、右移一步、下移一步. 如果机器人在边界则不会往区域外移动，问机器人到达最后一行的期望步数.

解答

在 $m = 1$ 时每次有 $\frac{1}{2}$ 的概率不动，有 $\frac{1}{2}$ 的概率向下移动一格，答案为 $2 \cdot (n - x)$. 设 $f_{i,j}$ 为机器人从第 i 行第 j 列出发到达第 n 行的期望步数，最终状态为 $f_{n,j} = 0$. 由于机器人会等概率地选择停在原地，左移一步，右移一步，下移一步，考虑 $f_{i,j}$ 的状态转移:

- $f_{i,1} = \frac{1}{3} \cdot (f_{i+1,1} + f_{i,2} + f_{i,1}) + 1$
- $f_{i,j} = \frac{1}{4} \cdot (f_{i,j} + f_{i,j-1} + f_{i,j+1} + f_{i+1,j}) + 1$
- $f_{i,m} = \frac{1}{3} \cdot (f_{i,m} + f_{i,m-1} + f_{i+1,m}) + 1$

在行之间由于只能向下移动，是满足无后效性的. 在列之间可以左右移动，在移动过程中可能产生环，不满足无后效性. 将方程变换后可以得到:

- $2f_{i,1} - f_{i,2} = 3 + f_{i+1,1}$
- $3f_{i,j} - f_{i,j-1} - f_{i,j+1} = 4 + f_{i+1,j}$
- $2f_{i,m} - f_{i,m-1} = 3 + f_{i+1,m}$

由于是逆序的递推，所以每一个 $f_{i+1,j}$ 是已知的. 由于有 m 列，所以右边相当于是一个 m 行的列向量，那么左边就是 m 行 m 列的矩阵. 使用增广矩阵，就变成了 m 行 $m+1$ 列的矩阵，然后进行 高斯消元→9.19.3@p1944 即可解出答案.

参考代码

```cpp
#include <cstdio>
#include <cstring>
using namespace std;

constexpr int MAXN = 1e3 + 10;

double a[MAXN][MAXN], f[MAXN];
int n, m;

void solve(int x) {
  for (int i = 1; i <= m; i++) {
    if (i == 1) {
      a[i][i] = 2;
      a[i][i + 1] = -1;
      a[i][m + 1] = 3 + f[i];
      continue;
    } else if (i == m) {
      a[i][i] = 2;
      a[i][i - 1] = -1;
      a[i][m + 1] = 3 + f[i];
      continue;
    }
    a[i][i] = 3;
    a[i][i + 1] = -1;
    a[i][i - 1] = -1;
    a[i][m + 1] = 4 + f[i];
  }

  for (int i = 1; i < m; i++) {
    double p = a[i + 1][i] / a[i][i];
    a[i + 1][i] = 0;
    a[i + 1][i + 1] -= a[i][i + 1] * p;
    a[i + 1][m + 1] -= a[i][m + 1] * p;
  }

  f[m] = a[m][m + 1] / a[m][m];
  for (int i = m - 1; i >= 1; i--)
    f[i] = (a[i][m + 1] - f[i + 1] * a[i][i + 1]) / a[i][i];
}

int main() {
  scanf("%d %d", &n, &m);
  int st, ed;
  scanf("%d %d", &st, &ed);
  if (m == 1) {
    printf("%.10f\n", 2.0 * (n - st));
    return 0;
  }
  for (int i = n - 1; i >= st; i--) {
    solve(i);
  }
```

```
52   printf("%.10f\n", f[ed]);
53   return 0;
54 }
55
```

习题

- HDU 4418 Time Travel[10]
- 「HNOI2013」游走[11]

参考文献

kuangbin 概率 DP 总结[12]

外部参考资料

[1] Codeforces 148D Bag of mice 
[2] POJ3071 Football 
[3] CodeForces 768D Jon and Orbs 
[4] POJ2096 Collecting Bugs 
[5] 「NOIP2016」换教室 
[6] HDU3853 LOOPS 
[7] HDU4035 Maze 
[8] 「SCOI2008」奖励关 
[9] CodeForces 24D Broken robot 
[10] HDU 4418 Time Travel 
[11] 「HNOI2013」游走 
[12] kuangbin 概率 DP 总结 

7.14 DP 套 DP

Authors: Hope666666

引入

本文将介绍 DP 套 DP 的思想，并通过两道例题展示它如何应用到具体问题中.

思想

所谓「DP 套 DP」，实际上指的是在动态规划的过程中，把一个子问题的求解过程（通常是一个 DP）抽象成一个自动机（DFA），然后在这个自动机的基础上再设计一层新的 DP 的方法.

这种技巧主要应用于一类 **序列计数**、**概率** 或 **期望** 问题. 一个典型的问题具有如下结构：

- 给定字符集 Σ 和它上面一个长度为 n 的「合法序列」的集合 $A \subseteq \Sigma^n$. 根据字符集不同，序列可以是二进制串、数字串、状态序列等.
- 对于每一个具体的序列 $s \in \Sigma^n$，可以通过动态规划来判断它是否合法（即 $s \in A$），计算它的权值或求出相关值.
- 最终，我们希望统计集合 A 中所有序列的数目、总权值、期望值等.

这时，枚举所有序列是不可行的. 于是我们考虑将「判断一个序列是否合法」的过程（即内层 DP）抽象为一个 确定有限状态自动机→13.8@p3284（DFA）. 一般地，对于固定的序列 $s \in \Sigma^n$，内层 DP 的状态函数可以表示为 $g(i, x; s)$，即已经处理完序列 s 的长度为 i 的前缀，且其他状态分量为 x 时某个量的取值. 相应地，内层 DP 的状态转移方程为

$$g(i, \cdot; s) = G(g(i-1, \cdot; s), s_i).$$

也就是说，函数 $g(i,\cdot;s)$ 由之前的函数 $g(i-1,\cdot;s)$ 和当前的字符 s_i 唯一确定. 如果将函数 $g(i,\cdot;s)$ 看作是自动机的一个状态，那么，内层 DP 的状态转移方程就给出了自动机的一个转移. 因此，内层 DP 对应的自动机 (Q,Σ,δ,q_0,F) 结构如下：

- 状态集合 Q 就是所有可能的 $s\in\Sigma^n$ 和 $i=0,1,\cdots,n$ 对应的函数 $g(i,\cdot;s)$ 的集合；
- 转移函数 $\delta:Q\times\Sigma\to Q$ 就是内层 DP 的状态转移方程中的 G；
- 起始状态 q_0 通常是显然的，即内层 DP 的初始状态；
- 接受状态集合 F 就对应着所有的合法序列 $s\in A$.

函数 $g(i,\cdot;s)$ 本身可能相当复杂，因此，在处理具体问题时，通常需要进行 状态压缩→7.8@p744 或结合 DFA 最小化的技巧来压缩状态空间. 这也是 DP 套 DP 相较于暴力 DP 能够显著降低时空复杂度的主要原因.

将内层 DP 抽象为 DFA 后，就可以在这个 DFA 上设计一个新的 DP 用于求解原问题，即外层 DP. 为方便表述，以简单的计数问题为例. 外层 DP 的状态函数定义为 $f(i,q)$，即处理到长度为 i 的前缀时，到达 DFA 中状态 $q\in Q$ 的前缀的数目. 它的状态转移方程为

$$f(i,q)=\sum_{c\in\Sigma}\sum_{q'\in Q:\delta(q',c)=q}f(i-1,q').$$

初始值当然是 $f(0,q_0)=1$，而最终要求的答案通常可以根据 $\{f(n,q):q\in F\}$ 简单计算得到. 外层 DP 实际上是 DAG 上 DP→7.6@p735 的特殊情形.

例题

接下来的两个例题会详细说明 DP 套 DP 的一般做法.

例一

Hero meet devil[1]

给定一个字符集为 `ACGT` 的字符串 S，且 $|S|\le 15$. 对于每个 $0\le i\le|S|$，求有多少个长度为 m，字符集 `ACGT` 的字符串 T，满足它与 S 的最长公共子序列长度为 i.

题解

我们首先会想到一个 DP：设 $f_{i,j}$ 表示长度为 i 的 T 中，和 S 的最长公共子序列的长度为 j 的方案数. 但是这样无法转移，我们发现主要的问题是，我们不知道这个最长公共子序列对应的是哪些字符.

考虑朴素求最长公共子序列的过程. 设 $g_{i,j}$ 表示 T 的前 i 位和 S 的前 j 位，它们的最长公共子序列的长度，就有

$$g_{i,j}=\max\{g_{i-1,j},g_{i,j-1},g_{i-1,j-1}+[T_i=S_j]\}.$$

我们发现，对于一个 i，只需要记录 g_i 这个一维数组每一位的值，就能准确维护当前 S 与 T 前 i 位最长公共子序列的状态. 因为 S 长度只有 15，我们发现这个思想是可行的.

于是重新设状态 $f_{i,x}$ 表示对于长度为 i 的 T，与 S 的 DP 数组（就是 g_i）状态为 x 的方案数. 这个 DP 看起来状态数很多，然而我们发现 $g_{i,j}-g_{i,j-1}\in\{0,1\}$，就可以维护 g_i 的差分数组，状态数是 $2^{|S|}$ 的.

现在思考怎么转移. 容易发现，如果我们知道了 g_i 这个数组，也知道了 T_{i+1}，就能通过朴素 LCS 转移（即前文的 DP 方程）求出 g_{i+1}. 于是朴素的 LCS 就成为了帮助 f 转移的内层 DP.

因此，我们枚举 T_{i+1}，计算出 x 转移后的状态 x'，再将 $f_{i+1,x'}$ 加上 $f_{i,x}$ 就可以完成外层 DP 的状态转移．最后，我们记录 ans_i 为 LCS 长度为 i 的答案，枚举每个状态 x，$ans_{\text{popcount}(x)}$ 加上 $f_{m,x}$ 即可．

参考代码

```cpp
#include <cstring>
#include <iostream>
#include <string>
using namespace std;

const int MOD = 1e9 + 7;
const int MAXN = 100;

int T, a[MAXN + 1], h[MAXN + 1];
int g[MAXN + 1], m, n;
int nxt[1 << 16]
       [4];  // 状态转移表：nxt[mask][ch] 表示 mask 状态下追加 ch 后的新状态
int f[1001]
     [1 << 16];  // DP 数组：f[i][mask] 表示长度为 i，处于状态 mask 的方案数
int ans[MAXN +
        1];  // 最终答案：ans[i] 表示长度为 m 时，包含 i 个匹配位置的方案数

void add(int &x, int y) { x = (x + y) % MOD; }

int calc(int mask, int x) {  // 给定当前 mask 状态和当前字符
                             // x（0~3），返回新状态
  int res = 0;
  for (int i = 0; i < n; ++i) g[i + 1] = g[i] + ((mask >> i) & 1);

  for (int i = 1; i <= n; ++i) {
    if (a[i] == x)
      h[i] = g[i - 1] + 1;
    else
      h[i] = max(h[i - 1], g[i]);
  }

  for (int i = 1; i <= n; ++i)
    if (h[i] > h[i - 1]) res |= (1 << (i - 1));

  return res;
}

int popcount(int x) {  // 计算 mask 中有多少个 1（即匹配了多少位置）
  int res = 0;
  while (x) {
    res += x & 1;
    x >>= 1;
  }
  return res;
}

```

```cpp
int main() {
  ios::sync_with_stdio(false);
  cin.tie(nullptr);

  cin >> T;
  while (T--) {
    string S;
    cin >> S >> m;
    n = S.length();

    memset(f, 0, sizeof(f));
    memset(ans, 0, sizeof(ans));
    memset(a, 0, sizeof(a));

    for (int i = 0; i < n; ++i) {
      if (S[i] == 'A') a[i + 1] = 0;
      if (S[i] == 'C') a[i + 1] = 1;
      if (S[i] == 'G') a[i + 1] = 2;
      if (S[i] == 'T') a[i + 1] = 3;
    }

    int lim = (1 << n);
    for (int i = 0; i < lim; ++i)
      for (int j = 0; j < 4; ++j) nxt[i][j] = calc(i, j);

    f[0][0] = 1;
    for (int i = 0; i < m; ++i)
      for (int j = 0; j < lim; ++j)
        if (f[i][j])
          for (int k = 0; k < 4; ++k) add(f[i + 1][nxt[j][k]], f[i][j]);

    for (int i = 0; i < lim; ++i) add(ans[popcount(i)], f[m][i]);

    for (int i = 0; i <= n; ++i) cout << ans[i] << '\n';
  }

  return 0;
}

```

例二

[ZJOI2019] 麻将[2]

假设麻将牌有 n 种大小的牌，每种大小有 4 张牌. 定义面子为三张相邻大小的麻将牌 $i, i+1, i+2$（顺子）或三张相同大小的麻将牌 i, i, i（刻子），对子为两张相同大小的麻将牌 i, i. 定义一个麻将牌的序列是胡的，当且仅当它（看作多重集合）可以拆成四个面子和一个对子，或者七个不同的对子. 给定初始的 13 张麻将牌，剩余的 $4n-13$ 张牌等概率随机打乱后依次摸取，问期望再摸多少张牌可以满足存在一个胡牌的子序列的条件，答案对 998244353 取模.

题解

首先，对于一副牌，我们只需考虑每种牌的数量，而不必关心它们的顺序．因此，对于任何一副牌的任意前缀，我们都可以将它转化为一个长度为 n，每个位置上为 $0\sim 4$ 的序列．初始时，第 i 种牌的数量为 a_i，就相当于限制了序列中第 i 个数字 x_i 的取值为 $[a_i,4]$ 内的整数．注意，转化后的序列没有考虑摸牌的顺序，但是题目中的麻将牌的序列是考虑顺序的．

设 X 表示可以胡牌的最小摸牌次数．直接计算期望 $\mathbf{E}[X]$ 较为困难，可以考虑进行如下转化．设 h_i 表示从剩余的牌中选出 i 张（同种牌的不同张视为不同）后 **没有胡** 的方案数．因为它对应的麻将牌序列中，这 i 张牌必然排在剩下的 $(4n-13-i)$ 张牌前方，但是这 i 张牌和剩下的 $(4n-13-i)$ 张牌的顺序是任意的，所以，只摸前 i 张牌无法胡牌的麻将牌序列的数目就是

$$h_i\cdot i!(4n-13-i)!.$$

因为麻将牌序列的总数为 $(4n-13)!$，所以，只摸前 i 张牌无法胡牌的概率为

$$\mathbf{P}[X>i]=\frac{h_i\cdot i!(4n-13-i)!}{(4n-13)!}.$$

利用尾求和公式，就可以得到所求期望

$$\mathbf{E}[X]=\sum_{i=0}^{\infty}\mathbf{P}[X>i]=1+\sum_{i=1}^{4n-13}\frac{h_i\cdot i!(4n-13-i)!}{(4n-13)!}.$$

至此，问题转化为如何计算 h_i．我们采用 DP 套 DP 的方法来解决这一问题．

首先，考虑内层 DP，也就是要用动态规划判断一个（转化后的）序列是否对应一副能胡的牌．七对子的情形较为容易，重点讨论第一种胡牌的形式．为此，设 $g_{0/1,i,j,k}$ 表示处理完前 i 种牌，还剩 j 组 $(i-1,i)$ 以及 k 张 i，且不存在/存在对子（即 $0/1$）时最多的面子数．如果对于一个序列进行 DP，最后得到的 $g_{1,n}$ 中包括一个大于等于 4 的数字，那么这个序列就是能胡的．

这个 DP 的状态转移较为复杂．我们分两步讨论．第一步，考虑 $g_{0/1,i}$ 的转移．这相当于说，如果要向现在的牌型中，添加 x_i 张大小为 i 的牌，但是不组成新的对子时，面子数如何转移．显然，如果希望在添加 x_i 张大小为 i 的牌后，要得到 ℓ 个顺子，j 个 $(i-1,i)$ 和 k 个单独的 i，那么，就应该从 $(g_{0/1,i-1})_{\ell,j}$ 中转移过来（这里的选择最大程度避免了浪费），并将剩余的牌 $(x_i-\ell-j-k)$ 用于组成尽可能多的刻子．穷举所有的可能性，就得到如下转移方程：

$$\tilde{G}(g_{0/1,i-1},x_i)_{j,k}=\max\left\{(g_{0/1,i-1})_{\ell,j}+\ell+\left\lfloor\frac{x_i-\ell-j-k}{3}\right\rfloor:\ell+j+k\le x_i\right\}.$$

第二步，再考虑需要组成对子的情形．加入 x_i 张大小为 i 的牌时，有如下三种转移：

- 将 $g_{0,i-1}$ 加 x_i 张牌转移到 $g_{0,i}$；
- 将 $g_{1,i-1}$ 加 x_i 张牌转移到 $g_{1,i}$；
- 若 $x_i\ge 2$，将 $g_{0,i-1}$ 加 x_i-2 张牌转移到 $g_{1,i}$．

由此，就得到了从 g_{i-1} 添加 x_i 张牌得到 g_i 的全部转移．

解决了内层 DP 的状态转移，就可以构建 **胡牌自动机**．自动机的转移就是上述内层 DP 的转移，还需要考虑如何表示自动机的每个状态．每个状态都对应 g_i 的一种可能的取值．它有三个维度 $(0/1,j,k)$．因为 j 和 k 对应的维度中保留的 $(i-1,i)$ 和 i 都是用于将来组成顺子的，而三个相同顺子总是可以重组为三个刻子，所以，只需要考虑组成不超过 2 个相同顺子的需求就行，每种牌型也就只要保留不超过 2 个，即 $j,k\in\{0,1,2\}$．因此，g_i 可以表示为一个 $2\times 3\times 3$ 的数组．另外，为了维护七对子的胡牌牌型，还需要为每个状态添加一个计数器，用于表示当前最多可组成的对子数目．

数组 g_i 中每个元素的取值范围可能是 $\{-\infty\}\cup\mathbf{N}$，但是，因为面子数目大于等于 4 都是胡牌，所以，可以限制每个元素取值不超过 4．由于胡牌序列再添加任何牌都是胡牌序列，所以，可以利用 DFA

最小化的思想，将全体胡牌状态压缩为一个状态. 因此，对于非胡牌状态，g_1 中每个位置的取值只要考虑 $\{-\infty\}\cup\{0,1,2,3\}$，而 g_0 中每个位置的取值为 $\{-\infty\}\cup\{0,1,2,3,4\}$. 实现时，$-\infty$ 用 -1 表示.

尽管如此，所有可能的状态依然相当地多，共有 $1+7\times5^9\times6^9$ 种. 穷举它们并不现实. 实际上，绝大多数这些可能性都不会真的出现在一个胡牌自动机中. 为了避免考虑实际不存在的状态，可以利用 BFS 的思想，从初始状态开始，一步一步扩展状态，直到胡牌状态处停止. 这样得到的自动机中有 $N=2092$ 个状态.

最后，考虑如何在胡牌自动机上 DP（即外层 DP）. 设 $f_{i,j,k}$ 表示处理完前 i 种牌，共摸了 j 张牌，走到了胡牌自动机上的 k 号状态的方案数. 转移时，枚举摸牌数 $0\le t\le 4-a_i$，其中 a_i 为初始 13 张牌中大小为 i 的牌的张数，将之前的方案数乘以 $4-a_i$ 张牌中选 t 张牌的方案数 $\binom{4-a_i}{t}$，再累加到一起. 形式化地，有：

$$f_{i,j+t,k'}\leftarrow f_{i,j+t,k'}+\binom{4-a_i}{t}f_{i-1,j,k}.$$

其中，$k'=\delta(k,a_i+t)$，表示向自动机的状态 k 中加入 a_i+t 张牌后的状态. 外层 DP 结束后，就可以计算出摸了 i 张牌仍然没有胡牌的方案数，即

$$h_i=\sum_{k\notin F}f_{n,i,k},$$

其中，F 为胡牌状态的集合. 代入前文所述表达式，就可以得到所要求的期望.

参考代码

```cpp
#include <algorithm>
#include <cstdio>
#include <iostream>
#include <map>
#include <unordered_map>
using namespace std;

const int N = 100, M = 400, mod = 998244353;

int n, m, a[N + 5], fact[M + 5], inv[M + 5];

inline int C(int x, int y) {
  return 1LL * fact[x] * inv[y] % mod * inv[x - y] % mod;
}

class HuAutomation {  // 胡牌自动机
 private:
  class Mat {
   private:
    int f[3][3];

   public:
    Mat() {
      for (int i = 0; i <= 2; i++)
        for (int j = 0; j <= 2; j++) f[i][j] = -1;
    }

    int* operator[](const int& x) { return f[x]; }
```

```
29
30      inline bool operator==(Mat x) const {
31        for (int i = 0; i <= 2; i++)
32          for (int j = 0; j <= 2; j++)
33            if (f[i][j] != x[i][j]) return 0;
34        return 1;
35      }
36
37      inline bool operator<(Mat x) const {
38        for (int i = 0; i <= 2; i++)
39          for (int j = 0; j <= 2; j++)
40            if (f[i][j] != x[i][j]) return f[i][j] < x[i][j];
41        return 0;
42      }
43
44      inline bool Check() {
45        for (int i = 0; i <= 2; i++)
46          for (int j = 0; j <= 2; j++)
47            if (f[i][j] > 3) return 1;
48        return 0;
49      }
50
51      inline void Upd(Mat x, int t) {
52        // 将已有矩阵 x 的值更新到当前矩阵中，模拟加入 t 张当前牌后的状态
53        for (int i = 0; i <= 2; i++)
54          for (int j = 0; j <= 2; j++)
55            if (x[i][j] != -1)
56              for (int k = 0; k < 3 && i + j + k <= t; k++)
57                f[j][k] = max(f[j][k], min(i + x[i][j] + (t - i - j - k) /
   3, 4));
58        // i,j,k,(t-i-j-k)
59        // 表示分别枚举用于拼面子、用于保留(i-1,i)、用于保留i和直接拼面子的牌数
60        //  更新时限制最多 4 个面子
61      }
62    };
63
64    struct node {
65      int t, state[5];
66      Mat F[2];
67
68      node() {
69        for (int i = 0; i <= 4; i++) state[i] = 0;
70        t = 0;
71        for (int i = 0; i <= 1; i++) F[i] = Mat();
72      }
73
74      inline bool Check() { return t == -1 || t >= 7 || F[1].Check(); }
75
76      node Hu() {
77        node x;
78        x.t = -1;
79        return x;
```

```
    }

    bool operator<(const node& x) const {
      if (t == x.t) {
        if (F[0] == x.F[0]) return F[1] < x.F[1];
        return F[0] < x.F[0];
      }
      return t < x.t;
    }

    node operator+(int x) {       // 加上 x 张新牌
      if (Check()) return Hu();  // 胡了直接返回
      node res;
      res.F[0].Upd(F[0], x), res.F[1].Upd(F[1], x);  // 更新
      res.t = t;
      if (x > 1) res.F[1].Upd(F[0], x - 2), ++res.t;
      if (res.Check()) res = Hu();  // 判断是否胡了
      return res;
    }
  } A[2100];

  map<node, int> mp;

  inline int Get(node x) {
    if (mp.count(x)) return mp[x];
    mp[x] = ++tot;
    A[tot] = x;
    return tot;
  }

  inline void Expand(int x) {
    // 构建状态转移图:
    // 对于当前状态 A[x]，尝试加入 0~4 张同种牌，得到 5 个新状态
    for (int i = 0; i <= 4; i++) A[x].state[i] = Get(A[x] + i);
  }

  inline node Hu() {
    node x;
    x.t = -1;
    return x;
  }

  inline node Emp() {
    node x;
    x.F[0][0][0] = 0;
    return x;
  }

public:
  int tot, f[N + 5][M + 5][2100];

  inline void Build() {
```

```
    A[1] = Emp(), A[2] = Hu();  // 状态1是初始合法状态，2 是胡牌状态
    mp[A[1]] = 1, mp[A[2]] = 2;
    tot = 2;
    Expand(1);
    for (int i = 3; i <= tot; i++)
      Expand(i);  // 枚举所有可达状态，构建状态图（因为 2
                  // 是胡牌状态所以不需要拓展）
  }

  void DP() {
    f[0][0][1] = 1;
    for (int i = 1; i <= n; i++)
      for (int j = m; j >= 0; j--)
        for (int k = 1; k <= tot; k++)
          if (f[i - 1][j][k])
            for (int t = 0; t <= 4 - a[i]; t++)
              // 枚举这种牌能加多少个（不能超过4，总共已有a[i]个），做组合转移
              (f[i][j + t][A[k].state[a[i] + t]] +=
               1LL * f[i - 1][j][k] * C(4 - a[i], t) % mod) %= mod;
  }
} Hfu;

inline long long qpow(long long x, int y) {
  long long res = 1;
  while (y) {
    if (y & 1) res = res * x % mod;
    x = x * x % mod;
    y >>= 1;
  }
  return res;
}

void init(int n) {
  fact[0] = 1;
  for (int i = 1; i <= n; i++) fact[i] = 1LL * fact[i - 1] * i % mod;
  inv[n] = qpow(fact[n], mod - 2);
  for (int i = n - 1; i >= 0; i--) inv[i] = 1LL * inv[i + 1] * (i + 1)
% mod;
}

int main() {
  ios::sync_with_stdio(false);
  cin.tie(nullptr);

  int ans = 0;
  Hfu.Build();  // 状态图构建
  cin >> n;
  for (int i = 1; i <= 13; i++) {
    int x, y;
    cin >> x;
    cin >> y;
    ++a[x];
```

```
  }
  m = (n << 2) - 13;
  init(m);
  Hfu.DP();

  for (int i = 1; i <= m; i++) {
    (ans += 1LL * Hfu.f[n][i][1] * fact[i] % mod * fact[m - i] % mod)
%= mod;
    for (int j = 3; j <= Hfu.tot; j++)  // 因为 2 节点是胡牌节点所以不统计
      (ans += 1LL * Hfu.f[n][i][j] * fact[i] % mod * fact[m - i] % mod)
%= mod;
    // f[i][j][k]: 表示处理完前 i 种牌，共摸了 j 张牌，走到了胡牌自动机上的 k
    // 号节点的方案数。这里对 f[n][i][j] 求和，
    // 每个方案乘以 i!(用于排列这 i 张) × (m - i)!
    // (剩下的牌排列)，求和后取模
  }

  cout << (1LL * ans * inv[m] + 1) % mod;
  // 最终答案乘上 inv[m] 是除以 m!，即去掉总排列数，最后 +1 是 dp 的定义是摸完 i
  // 张还没有胡
  return 0;
}
```

习题

- CF979E Kuro and Topological Parity[3]
- [TJOI2018] 游园会[4]
- [NOI2022] 移除石子[5]

外部参考资料

[1] Hero meet devil

[2] [ZJOI2019] 麻将

[3] CF979E Kuro and Topological Parity

[4] [TJOI2018] 游园会

[5] [NOI2022] 移除石子

7.15 DP 优化

7.15.1 DP 优化简介

引入

本页面列举了一些常见的动态规划（dynamic programming，DP）优化方法．所谓 DP 优化，是指许多动态规划问题虽然容易列出朴素的状态转移方程，但直接计算往往效率低下，因此需要借助一些技巧来降低时间复杂度．

这些方法彼此联系紧密，常常包含相似的思想，且往往需要结合多种技巧共同使用．因此，本文仅作粗略分类，侧重介绍一些最基本的思路．

利用常见技巧优化

许多动态规划问题的转移，可以通过常见的算法和数据结构进行优化．

这类技巧有两种常见的情形．第一种情形中，DP 问题有着如下状态转移方程：

$$f(i) = F(a_i, \{f(j) : j < i\}).$$

其中，当前状态 $f(i)$ 的计算依赖于当前的输入 a_i 和之前整个序列的状态 $\{f(j) : j < i\}$．因此，可以维护一个数据结构，将每次 $f(i)$ 的计算都看作是一次查询操作；而得到当前状态 $f(i)$ 后，可以再执行一次修改操作，更新该数据结构，用于后续的转移．

第二种情形中，DP 问题有着如下状态转移方程：

$$f(i, \cdot) = F(a_i, f(i-1, \cdot)).$$

其中，每个 $f(i, \cdot)$ 都是一个数组或其他较复杂的对象．因此，虽然 $f(i, \cdot)$ 只依赖于之前的单个状态，但是单次转移的复杂度很高，需要通过数据结构等技巧进行优化．

前缀和优化 DP

相关页面：前缀和→5.8@p617

如果当前状态的计算依赖于之前状态的子段和，那么，可以通过维护前缀和来加速计算．其中，一类涉及高维前缀和的问题也称为 SOS DP→5.8@p617．

习题：

- Luogu P2513 [HAOI2009] 逆序对数列[1]
- AtCoder Educational DP Contest M - Candies[2]

单调队列/单调栈优化 DP

主页面：单调队列/单调栈优化→7.15.2@p814

如果当前状态依赖于之前状态的区间最值等信息时，可以通过维护单调队列、单调栈来加速计算．

线段树/树状数组优化 DP

相关页面：线段树→10.13.1@p2098、树状数组→10.12@p2078

如果每次状态转移时，查询的都是某段区间上的和、最值等信息，或是单次修改操作涉及区间更新时，可以通过维护线段树、树状数组来加速计算．

习题：

- AtCoder Educational DP Contest Q - Flowers[3]
- AtCoder Educational DP Contest W - Intervals[4]
- Codeforces 115 E. Linear Kingdom Races[5]

CDQ 分治优化 DP

主页面：CDQ 分治优化 DP→13.4.2@p3176

类似前文，将整个 DP 过程看作是一系列查询和修改操作．对于有些问题，顺次计算复杂度过高，可以将整个查询和修改过程离线，利用 CDQ 分治加速计算．

CDQ 分治优化 DP 也常见于下列各类问题中：

- 基于 CDQ 分治的斜率优化 DP→7.15.3@p818
- 分治优化斜率单调性 DP→7.15.4@p821

倍增优化 DP

相关页面：倍增→5.10@p645

有些问题中，需要将状态 $f(i)$ 设为从初始状态开始，进行了 2^i 次转移时得到的结果．它利用了倍增的思想转化了原问题，因此常常称为倍增优化 DP．

有些时候，也会将状态转移方程类似

$$f(i, j) = f(i-1, f(i-1, j))$$

的 DP 问题称为倍增（优化）DP.

习题：

- Luogu P1081 [NOIP 2012 提高组] 开车旅行[6]
- Luogu P1613 跑路[7]
- Luogu P4739 [CERC2017] Donut Drone[8]

利用问题结构优化

许多动态规划问题具备如凸性、单调性等结构性质，合理利用这些性质可以快速求解.

斜率优化 DP

主页面：斜率优化→7.15.3@p818

类似于上一节介绍的优化方法，利用问题的凸性，可以通过维护凸包加速单次转移.

四边形不等式优化 DP

主页面：四边形不等式优化→7.15.4@p821

涉及的函数满足四边形不等式的 DP 问题，往往满足某种决策单调性. 利用这一特性，有很多专门的方法可以降低计算复杂度. 常见的问题类型包括一维决策单调性问题、区间分拆问题、区间合并问题等.

Slope Trick 优化 DP

主页面：Slope Trick→7.15.5@p836

有些问题中，状态函数的差分（即斜率）在状态转移过程中更容易维护. 这种优化也往往需要问题具有凸性.

WQS 二分/凸优化 DP

主页面：WQS 二分→7.15.6@p852

对于带个数限制的最优化 DP 问题，如果不考虑个数限制时，问题更容易解决，且最优价值是关于该限制的凸函数，那么，可以通过 WQS 二分的方法简化计算.

利用数学方法优化

许多动态规划问题的转移，可以通过数学工具加速.

矩阵快速幂优化 DP

相关页面：快速幂→9.7@p1131

如果 DP 问题的状态转移方程可以写作一个自治的形式

$$f(i) = F(f(i-1)),$$

也就是说，当前状态 $f(i)$ 只取决于前一个状态 $f(i-1)$，而不依赖于其他输入，那么，可以直接通过快速幂加速计算

$$f(n) = F^n(f(0))$$

获得最终答案. 由于单次操作 F 经常可以写作矩阵的形式，所以这一优化方法常称为矩阵快速幂优化 DP. 实际上，任何满足结合律的变换（即任何 幺半群→9.16.1@p1805 内的元素）都可以应用该方法加速.

习题：

- Luogu P1397 [NOI2013] 矩阵游戏[9]
- Luogu P3176 [HAOI2015] 数字串拆分[10]
- Codeforces 576 D. Flights for Regular Customers[11]
- Luogu P6772 [NOI2020] 美食家[12]

FFT 优化 DP

相关页面：FFT→9.12.3@p1527

如果 DP 问题的状态转移方程是一个卷积的形式，那么，可以考虑用 FFT 加速转移．当然，根据具体的题目，也可能会用到其他多项式技术．

习题：

- Codeforces 553 E. Kyoya and Train[13]
- Codeforces 1784 D. Wooden Spoon[14]

Lagrange 插值优化 DP

相关页面：Lagrange 插值→9.19.1@p1930

有些 DP 问题的状态函数 $f(i,j)$ 是关于 j 的 k 次多项式函数．此时，可以直接暴力计算它在 $k+1$ 个点处的取值，并通过 Lagrange 插值求出 $f(i,\cdot)$ 的表达式，进而优化转移甚至直接得到答案．

习题：

- Luogu P5223 Function[15]
- Luogu P4463 [集训队互测 2012] calc[16]
- Luogu P5469 [NOI2019] 机器人[17]

通过简化状态优化

除了优化转移外，还可以通过简化状态降低计算复杂度．

DP 套 DP 与 DFA 最小化

主页面：DP 套 DP→7.14@p801、DFA 最小化→13.8@p3284

一些 DP 问题的状态函数可以写成 $f(i,x)$ 的形式，但是 x 自身的转移较为复杂，甚至可能依赖于另一个 DP 问题．对于这类问题，可以首先为状态 x 的转移建立自动机，并通过 DFA 最小化的方法减少状态数量，再进行外层 DP．

状态设计优化 DP

主页面：状态设计优化→7.15.7@p887

有些特殊的问题可以通过巧妙的状态设计大幅减少状态数量．

拓展阅读

- DP 优化方法大杂烩 by Alex Wei[18]

外部参考资料

[1] Luogu P2513 [HAOI2009] 逆序对数列

[2] AtCoder Educational DP Contest M - Candies

[3] AtCoder Educational DP Contest Q - Flowers

[4] AtCoder Educational DP Contest W - Intervals

[5] Codeforces 115 E. Linear Kingdom Races

[6] Luogu P1081 [NOIP 2012 提高组] 开车旅行

[7] Luogu P1613 跑路

[8] Luogu P4739 [CERC2017] Donut Drone

[9] Luogu P1397 [NOI2013] 矩阵游戏

[10] Luogu P3176 [HAOI2015] 数字串拆分

[11] Codeforces 576 D. Flights for Regular Customers

[12] Luogu P6772 [NOI2020] 美食家

[13] Codeforces 553 E. Kyoya and Train

[14] Codeforces 1784 D. Wooden Spoon

[15] Luogu P5223 Function

[16] Luogu P4463 [集训队互测 2012] calc  [17] Luogu P5469 [NOI2019] 机器人  [18] DP 优化方法大杂烩 by Alex Wei 

7.15.2 单调队列/单调栈优化

Authors: liujiaxi123456, Marcythm, hsfzLZH1, Ir1d, greyqz, Anguei, billchenchina, Chrogeek, ChungZH

引入

前置知识：单调队列→10.10@p2068、单调栈→10.9@p2067.

单调队列主要用于维护两端指针单调不减的区间最值，而单调栈则主要用于维护前/后第一个大于/小于当前值的数.

注意

- 求最小值要维护 **单调递增/不减** 的单调队列/单调栈，反之亦然.
- 维护单调递增/递减比较时用 **小于等于/大于等于**，维护单调不减/不增比较时用 **小于/大于**.

单调队列优化具体步骤

- 加入所需元素：向单调队列重复加入元素直到当前元素达到所求区间的右边界，这样就能保证所需元素都在单调队列中.
- 弹出越界队首：单调队列本质上是维护的是所有已插入元素的最值，但我们想要的往往是一个区间最值. 于是我们弹出在左边界外的元素，以保证单调队列中的元素都在所求区间中.
- 获取最值：直接取队首作为答案即可.

单调栈优化具体步骤

- 弹出非法栈顶：通过比较当前元素与栈顶的大小，弹出不满足单调栈性质的栈顶. 以单调递增的栈（即栈顶最大，维护最小值）为例，将所有大于等于当前元素的栈内元素全部弹出.
- 加入当前元素：将当前元素入栈即可.

单调队列优化多重背包

问题描述

你有 n 个物品，每个物品重量为 w_i，价值为 v_i，数量为 k_i. 你有一个承重上限为 W 的背包，现在要求你在不超过重量上限的情况下选取价值和尽可能大的物品放入背包. 求最大价值.

不了解背包 DP 的请先阅读 背包 DP→7.4.1@p724. 设 $f_{i,j}$ 表示前 i 个物品装入承重为 j 的背包的最大价值，朴素的转移方程为

$$f_{i,j}=\max_{k=0}^{k_i}(f_{i-1,j-k\times w_i}+v_i\times k)$$

时间复杂度 $O(W\sum k_i)$.

考虑优化 f_i 的转移. 为方便表述，设 $g_{x,y}=f_{i,x\times w_i+y}, g'_{x,y}=f_{i-1,x\times w_i+y}$，其中 $0\le y<w_i$，则转移方程可以表示为：

$$g_{x,y}=\max_{k=0}^{k_i}(g'_{x-k,y}+v_i\times k)$$

设 $G_{x,y}=g'_{x,y}-v_i\times x$. 则方程可以表示为：

$$g_{x,y}=\max_{k=0}^{k_i}(G_{x-k,y})+v_i\times x$$

这样就转化为一个经典的单调队列优化形式了．$G_{x,y}$ 可以 $O(1)$ 计算，因此对于固定的 y，我们可以在 $O\left(\left\lfloor\frac{W}{w_i}\right\rfloor\right)$ 的时间内计算出 $g_{x,y}$．因此求出所有 $g_{x,y}$ 的复杂度为 $O\left(\left\lfloor\frac{W}{w_i}\right\rfloor\right)\times O(w_i)=O(W)$．这样转移的总复杂度就降为 $O(nW)$．

在实现的时候，我们需要先枚举 y，这样才能保证枚举 x 的时候利用单调队列进行优化，而单调队列中存储的是 $x-k$，并不存储 k，这样使用的时候需要通过 `f[last][q.front() * w[i] + y] - q.front() * v[i]` 获取对应的 $G_{x-k,y}$，不难发现 $x-k\in[x-k_i,x]$，因此在枚举 x 的时候，我们需要删除队列中不在这个范围内的元素．

参考代码

```cpp
#include <array>
#include <deque>
#include <iostream>

constexpr int MAXV = 4e4 + 10;
constexpr int MAXN = 1e2 + 10;

using namespace std;

int n, W, last = 0, now = 1;
array<int, MAXN> v, w, k;
array<array<int, MAXV>, 2> f;
deque<int> q;

int main() {
  ios::sync_with_stdio(false);
  cin >> n >> W;
  for (int i = 1; i <= n; i++) {
    cin >> v[i] >> w[i] >> k[i];
  }
  for (int i = 1; i <= n; i++) {
    for (int y = 0; y < w[i]; y++) {
      // 清空队列
      deque<int>().swap(q);
      for (int x = 0; x * w[i] + y <= W; x++) {
        // 弹出不在范围的元素
        while (!q.empty() && q.front() < x - k[i]) {
          q.pop_front();
        }
        // 保证队列单调
        while (!q.empty() && f[last][q.back() * w[i] + y] - q.back() *
v[i] <
                                 f[last][x * w[i] + y] - x * v[i]) {
          q.pop_back();
        }
        q.push_back(x);
        f[now][x * w[i] + y] =
            f[last][q.front() * w[i] + y] - q.front() * v[i] + x * v[i];
      }
    }
    swap(last, now);
  }
```

```
  cout << f[last][W] << endl;
  return 0;
}

```

习题

例题 CF372C Watching Fireworks is Fun[1]

题目大意：城镇中有 n 个位置，有 m 个烟花要放．第 i 个烟花放出的时间记为 t_i，放出的位置记为 a_i．如果烟花放出的时候，你处在位置 x，那么你将收获 $b_i - |\ a_i - x\ |$ 点快乐值．

初始你可在任意位置，你每个单位时间可以移动不大于 d 个单位距离．现在你需要最大化你能获得的快乐值．

设 $f_{i,j}$ 表示在放第 i 个烟花时，你的位置在 j 所能获得的最大快乐值．

写出状态转移方程：$f_{i,j} = \max\{f_{i-1,k} + b_i - |\ a_i - j\ |\}$，其中 $j - (t_i - t_{i-1}) \times d \le k \le j + (t_i - t_{i-1}) \times d$．

尝试变形：

由于 max 里出现了一个确定的常量 b_i，我们可以将它提到外面去．

$f_{i,j} = \max\{f_{i-1,k} + b_i - |\ a_i - j\ |\} = \max\{f_{i-1,k} - |\ a_i - j\ |\} + b_i$

如果确定了 i 和 j 的值，那么 $|\ a_i - j\ |$ 的值也是确定的，也可以将这一部分提到外面去．

最后，式子变为：

$$f_{i,j} = \max\{f_{i-1,k} - |\ a_i - j\ |\} + b_i = \max\{f_{i-1,k}\} - |\ a_i - j\ | + b_i$$

接下来考虑单调队列优化．由于最终式子中的 max 只和上一状态中连续的一段的最大值有关，所以我们在计算一个新的 i 的状态值时候只需将原来的 f_{i-1} 构造成一个单调队列，并维护单调队列，使得其能在均摊 $O(1)$ 的时间复杂度内计算出 $\max\{f_{i-1,k}\}$ 的值，从而根据公式计算出 $f_{i,j}$ 的值．

总的时间复杂度为 $O(nm)$．

参考代码

```
#include <algorithm>
#include <cstring>
#include <iostream>
using namespace std;
using ll = long long;

constexpr int MAXN = 150000 + 10;
constexpr int MAXM = 300 + 10;

ll f[2][MAXN];
ll a[MAXM], b[MAXM], t[MAXM];
int n, m, d;

int que[MAXN];

```

```cpp
int fl = 1;

void init() {
  memset(f, 207, sizeof(f));
  memset(que, 0, sizeof(que));
  for (int i = 1; i <= n; i++) f[0][i] = 0;
  fl = 1;
}

void dp() {
  init();
  for (int i = 1; i <= m; i++) {
    int l = 1, r = 0, k = 1;
    for (int j = 1; j <= n;
         j++) {  // 在这里使用了单调队列的优化，推式子详见上面
      for (; k <= min(1ll * n, j + d * (t[i] - t[i - 1])); k++) {
        while (l <= r && f[fl ^ 1][que[r]] <= f[fl ^ 1][k]) r--;
        que[++r] = k;
      }

      while (l <= r && que[l] < max(1ll, j - d * (t[i] - t[i - 1]))) l++;
      f[fl][j] = f[fl ^ 1][que[l]] - abs(a[i] - j) + b[i];
    }

    fl ^= 1;
  }
}

int main() {
  cin >> n >> m >> d;
  for (int i = 1; i <= m; i++) cin >> a[i] >> b[i] >> t[i];

  // then dp
  dp();
  ll ans = -1e18;
  for (int i = 1; i <= n; i++) ans = max(ans, f[fl ^ 1][i]);
  cout << ans << endl;
  return 0;
}
```

- 「Luogu P1886」滑动窗口[2]
- 「NOI2005」瑰丽华尔兹[3]
- 「SCOI2010」股票交易[4]

外部参考资料

[1] CF372C Watching Fireworks is Fun 

[2] 「Luogu P1886」滑动窗口 

[3] 「NOI2005」瑰丽华尔兹 

[4] 「SCOI2010」股票交易 

7.15.3 斜率优化

Authors: Marcythm, hsfzLZH1, abc1763613206, greyqz, Ir1d, billchenchina, Chrogeek, Enter-tainer, StudyingFather, MrFoodinChina, luoguyuntianming, sshwy, wood3

例题引入

「HNOI2008」玩具装箱[1]

有 n 个玩具排成一排，第 i 个玩具价值为 c_i．要求将这 n 个玩具分成若干段．对于一段 $[l,r]$，它的代价为 $(r-l+\sum_{i=l}^{r}c_i-L)^2$．其中 L 是一个常量，求分段的最小代价．

$1\le n\le 5\times 10^4, 1\le L,c_i\le 10^7$．

朴素的 DP 做法

令 f_i 表示前 i 个物品，分若干段的最小代价．

状态转移方程：$f_i=\min_{j<i}\{f_j+(i-(j+1)+pre_i-pre_j-L)^2\}=\min_{j<i}\{f_j+(pre_i-pre_j+i-j-1-L)^2\}$．

其中 pre_i 表示前 i 个数的和，即 $\sum_{j=1}^{i}c_j$．

该做法的时间复杂度为 $O(n^2)$，无法解决本题．

优化

考虑简化上面的状态转移方程式：令 $s_i=pre_i+i, L'=L+1$，则 $f_i=\min_{j<i}\{f_j+(s_i-s_j-L')^2\}$．

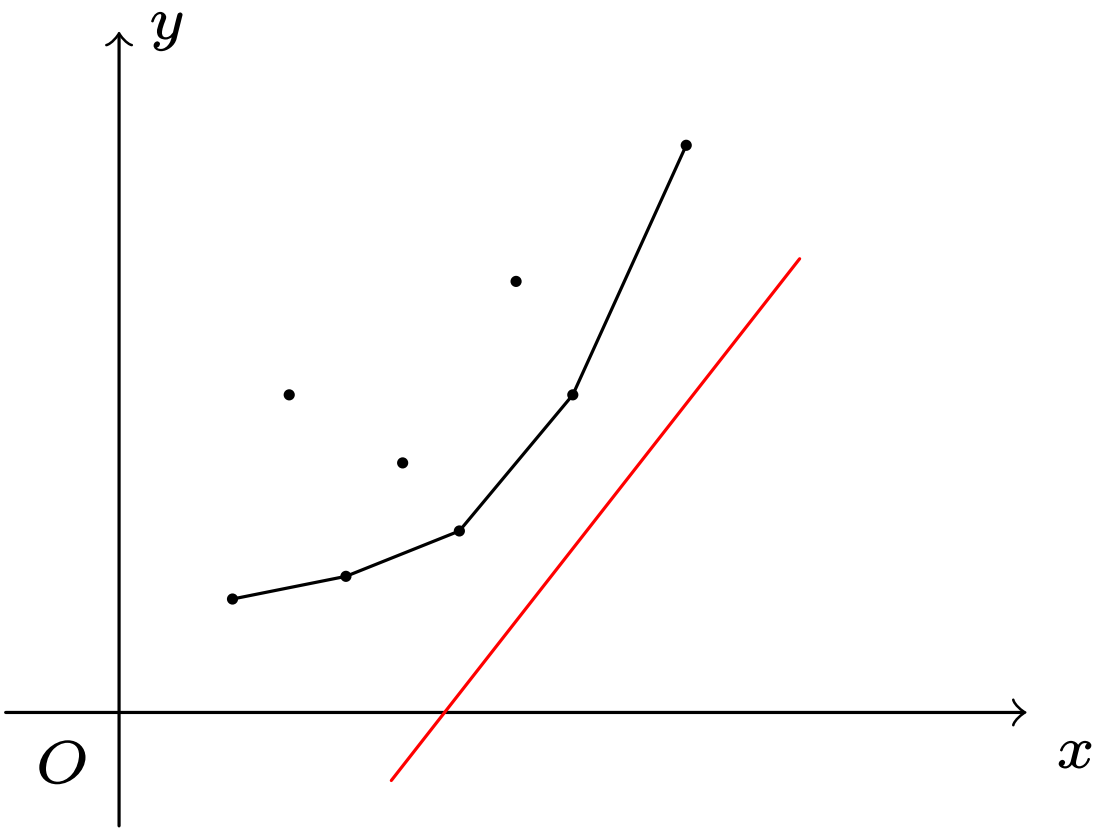
将与 j 无关的移到外面，我们得到

$$f_i-(s_i-L')^2=\min_{j<i}\{f_j+s_j^2+2s_j(L'-s_i)\}$$

考虑一次函数的斜截式 $y=kx+b$，将其移项得到 $b=y-kx$．我们将与 j 有关的信息表示为 y 的形式，把同时与 i,j 有关的信息表示为 kx，把要最小化的信息（与 i 有关的信息）表示为 b，也就是截距．具体地，设

$$\begin{aligned}x_j&=s_j\\y_j&=f_j+s_j^2\\k_i&=-2(L'-s_i)\\b_i&=f_i-(s_i-L')^2\end{aligned}$$

则转移方程就写作 $b_i=\min_{j<i}\{y_j-k_ix_j\}$．我们把 (x_j,y_j) 看作二维平面上的点，则 k_i 表示直线斜率，b_i 表示一条过 (x_j,y_j) 的斜率为 k_i 的直线的截距．问题转化为了，选择合适的 j（$1\le j<i$），最小化直线的截距．



如图，我们将这个斜率为 k_i 的直线从下往上平移，直到有一个点 (x_p, y_p) 在这条直线上，则有 $b_i = y_p - k_i x_p$，这时 b_i 取到最小值．算完 f_i，我们就把 (x_i, y_i) 这个点加入点集中，以做为新的 DP 决策．那么，我们该如何维护点集?

容易发现，可能让 b_i 取到最小值的点一定在下凸壳上．因此在寻找 p 的时候我们不需要枚举所有 $i-1$ 个点，只需要考虑凸包上的点．而在本题中 k_i 随 i 的增加而递增，因此我们可以单调队列维护凸包．

具体地，设 $K(a, b)$ 表示过 (x_a, y_a) 和 (x_b, y_b) 的直线的斜率．考虑队列 $q_l, q_{l+1}, ..., q_r$，维护的是下凸壳上的点．也就是说，对于 $l < i < r$，始终有 $K(q_{i-1}, q_i) < K(q_i, q_{i+1})$ 成立．

我们维护一个指针 e 来计算 b_i 最小值．我们需要找到一个 $K(q_{e-1}, q_e) \le k_i < K(q_e, q_{e+1})$ 的 e（特别地，当 $e = l$ 或者 $e = r$ 时要特别判断），这时就有 $p = q_e$，即 q_e 是 i 的最优决策点．由于 k_i 是单调递增的，因此 e 的移动次数是均摊 $O(1)$ 的．

在插入一个点 (x_i, y_i) 时，我们要判断是否 $K(q_{r-1}, q_r) < K(q_r, i)$，如果不等式不成立就将 q_r 弹出，直到等式满足．然后将 i 插入到 q 队尾．

这样我们就将 DP 的复杂度优化到了 $O(n)$．

概括一下上述斜率优化模板题的算法:

1. 将初始状态入队．
2. 每次使用一条和 i 相关的直线 $f(i)$ 去切维护的凸包，找到最优决策，更新 dp_i．
3. 加入状态 dp_i．如果一个状态（即凸包上的一个点）在 dp_i 加入后不再是凸包上的点，需要在 dp_i 加入前将其剔除．

接下来我们介绍斜率优化的进阶应用，将斜率优化与二分/分治/数据结构等结合，来维护性质不那么好（缺少一些单调性性质）的 DP 方程．

二分/CDQ/平衡树优化 DP

当我们在 i 这个点寻找最优决策时，会使用一个和 i 相关的直线 $f(i)$ 去切我们维护的凸包．切到的点即为最优决策．

在上述例题中，直线的斜率随 i 单调变化，但是对于有些问题，斜率并不是单调的．这时我们需要维护凸包上的每一个节点，然后每次用当前的直线去切这个凸包．这个过程可以使用二分解决，因为凸包上相邻两个点的斜率是有单调性的．

玩具装箱 改

有 n 个玩具排成一排，第 i 个玩具价值为 c_i．要求将这 n 个玩具分成若干段．对于一段 $[l,r]$，它的代价为 $(r-l+\sum_{i=l}^{r} c_i - L)^2$．其中 L 是一个常量，求分段的最小代价．

$1 \le n \le 5\times 10^4, 1 \le L \le 10^7, -10^7 \le c_i \le 10^7$．

本题与「玩具装箱」问题唯一的区别是，玩具的价值可以为负．延续之前的思路，令 f_i 表示前 i 个物品，分若干段的最小代价．

状态转移方程：$f_i = \min_{j<i}\{f_j + (pre_i - pre_j + i - j - 1 - L)^2\}$．

其中 $pre_i = \sum_{j=1}^{i} c_j$．

将方程做相同的变换

$$f_i - (s_i - L')^2 = \min_{j<i}\{f_j + s_j^2 + 2s_j(L' - s_i)\}$$

然而这时有两个条件不成立了：

1. 直线的斜率不再单调；
2. 每次加入的决策点的横坐标不再单调．

仍然考虑凸壳的维护．

在寻找最优决策点，也就是用直线切凸壳的时候，我们将单调队列找队首改为：凸壳上二分．我们二分出斜率最接近直线斜率的那条凸壳边，就可以找到最优决策．

在加入决策点，也就是凸壳上加一个点的时候，我们有两种方法维护．

第一种方法是直接用平衡树维护凸壳．那么寻找决策点的二分操作就转化为在平衡树上二分，插入决策点就转化为在平衡树上插入一个结点，并删除若干个被踢出凸壳的点．此方法思路简洁但实现繁琐．

下面介绍一种基于 CDQ 分治→13.4.2@p3176 的做法．

设 CDQ(l,r) 代表计算 $f_i, i \in [l,r]$．考虑 CDQ$(1,n)$：

- 我们先调用 CDQ$(1,mid)$ 算出 $f_i, i \in [1,mid]$．然后我们对 $[1,mid]$ 这个区间内的决策点建凸壳，然后使用这个凸壳去更新 $f_i, i \in [mid+1,n]$．这时我们决策点集是固定的，不像之前那样边计算 DP 值边加入决策点，那么我们就可以把 $i \in [mid+1,n]$ 的 f_i 先按照直线的斜率 k_i 排序，然后就可以使用单调队列来计算 DP 值了．当然，也可以在静态凸壳上二分计算 DP 值．
- 对于 $[mid+1,n]$ 中的每个点，如果它的最优决策的位置是在 $[1,mid]$ 这个区间，在这一步操作中他就会被更新成最优答案．当执行完这一步操作时，我们发现 $[1,mid]$ 中的所有点已经发挥了全部的作用，凸壳中他们存不存在已经不影响之后的答案更新．因此我们可以直接舍弃这个区间的决策点，并使用 CDQ$(mid+1,n)$ 解决右区间剩下的问题．

时间复杂度 $O(n\log^2 n)$．

对比「玩具装箱」和「玩具装箱 改」，可以总结出以下两点：

- 二分/CDQ/平衡树等能够优化 DP 方程的计算，于一定程度上降低复杂度，但不能改变这个方程本身．
- DP 方程的性质会取决于数据的特征，但 DP 方程本身取决于题目中的数学模型．

小结

斜率优化 DP 需要灵活运用，其宗旨是将最优化问题转化为二维平面上与凸包有关的截距最值问题. 遇到性质不太好的方程，有时需要辅以数据结构来加以解决，届时还请就题而论.

习题

- 「SDOI2016」征途[2]
- 「ZJOI2007」仓库建设[3]
- 「APIO2010」特别行动队[4]
- 「JSOI2011」柠檬[5]
- 「Codeforces 311B」Cats Transport[6]
- 「NOI2007」货币兑换[7]
- 「NOI2019」回家路线[8]
- 「NOI2016」国王饮水记[9]
- 「NOI2014」购票[10]

外部参考资料

[1] 「HNOI2008」玩具装箱 
[2] 「SDOI2016」征途 
[3] 「ZJOI2007」仓库建设 
[4] 「APIO2010」特别行动队 
[5] 「JSOI2011」柠檬 
[6] 「Codeforces 311B」Cats Transport 
[7] 「NOI2007」货币兑换 
[8] 「NOI2019」回家路线 
[9] 「NOI2016」国王饮水记 
[10] 「NOI2014」购票 

7.15.4 四边形不等式优化

> **Authors:** Marcythm, zyf0726, hsfzLZH1, MingqiHuang, Ir1d, greyqz, billchenchina, Chrogeek, StudyingFather, NFLSCode, c-forrest

四边形不等式优化利用的是状态转移方程中的决策单调性，也常称为 **决策单调性优化** DP.

基础知识

考虑最简单的情形，我们要解决如下一系列最优化问题：

$$f(i) = \min_{1 \le j \le i} w(j, i) \qquad (1 \le i \le n) \tag{1}$$

这里假定成本函数 $w(j, i)$ 可以在 $O(1)$ 时间内计算.

约定

动态规划的状态转移方程经常可以写作一系列最优化问题的形式. 以（1）式为例，这些问题含有参数 i，问题的目标函数和可行域都可以依赖于 i. 每一个问题都是在给定参数 i 时，选取某个可行解 j 来最小化目标函数的取值. 为表述方便，下文将参数为 i 的最优化问题简称为「问题 i」，该最优化问题的可行解 j 称为「决策 j」，目标函数在最优解处取得的值则称为「状态 $f(i)$」. 同时，记问题 i 对应的最小最优决策点为 $\mathrm{opt}(i)$.

在一般的情形下，这些问题总时间复杂度为 $O(n^2)$. 这是由于对于问题 i，我们需要考虑所有可能的决策 j. 而在满足决策单调性时，可以有效缩小决策空间，优化总复杂度.

- **决策单调性**：对于任意 $i_1 < i_2$，必然成立 $\mathrm{opt}(i_1) \le \mathrm{opt}(i_2)$.

附注

对于问题 i，最优决策集合未必是一个区间．决策单调性实际可以定义在最优决策集合上．对于集合 A 和 B，可以定义 $A \le B$ 当且仅当对于任意 $a \in A$ 和 $b \in B$，成立 $\min\{a,b\} \in A$ 和 $\max\{a,b\} \in B$．这蕴含最小（最大）最优决策点的单调性，即此处采取的定义．本文关于最小最优决策点叙述的结论，同样适用于最大最优决策点．但是，存在情形，某更大问题的最小最优决策严格小于另一更小问题的最大最优决策，亦即可能对某些 $i_1 < i_2$ 成立 $\operatorname{optmax}(i_1) > \operatorname{optmin}(i_2)$，所以在书写代码时，应保证总是求得最小或最大的最优决策点．

另一方面，拥有相同最小最优决策的问题构成一个区间．这一区间，作为最小最优决策的函数，应严格递增．亦即，给定 $j_1 = \operatorname{opt}(i_1)$，$j_2 = \operatorname{opt}(i_2)$，如果 $j_1 < j_2$，那么必然有 $i_1 < i_2$．换言之，如果决策 $j_1 < j_2$ 能够成为最小最优决策的问题区间分别是 $[l_{j_1}, r_{j_1}]$ 和 $[l_{j_2}, r_{j_2}]$，那么必然有 $r_{j_1} < l_{j_2}$．

最常见的判断决策单调性的方法是通过四边形不等式（quadrangle inequality）．在不同的语境下，这一性质也常称为 Monge 性质（用于描述矩阵 $A_{j,i}$）或次模性（submodularity，用于描述以区间为自变量的函数 $f([j,i])$）．

- **四边形不等式**：如果对于任意 $a \le b \le c \le d$ 均成立
  $w(a,c) + w(b,d) \le w(a,d) + w(b,c)$，
  则称函数 w 满足四边形不等式（简记为「交叉小于包含」）．若等号永远成立，则称函数 w 满足 **四边形恒等式**．

如果没有特别说明，以下都会保证 $a \le b \le c \le d$．四边形不等式给出了一个决策单调性的充分不必要条件．

定理 1

若 w 满足四边形不等式，则问题 (1) 满足决策单调性．

证明

要证明这一点，可采用反证法．假设对某些 $c < d$，成立 $a = \operatorname{opt}(d) < \operatorname{opt}(c) = b$．此时有 $a < b \le c < d$．根据最优化条件，$w(a,d) \le w(b,d)$ 且 $w(b,c) < w(a,c)$，于是，$w(a,d) - w(b,d) \le 0 < w(a,c) - w(b,c)$，这与四边形不等式矛盾．

四边形不等式可以理解为在合理的定义域内，w 的二阶混合差分 $\Delta_i \Delta_j w(j,i)$ 非正．

利用决策单调性，有很多常见算法都可以将算法复杂度优化到 $O(n \log n)$．这些算法的适用范围、实现难度、运行效率各不相同，需要根据实际场景选择合适的算法．这主要取决于 $w(j,i)$ 的性质．不加说明时，本文默认 $w(j,i)$ 可以 **随机访问**，即 $w(j,i)$ 可以在 $O(1)$ 时间内查询或计算．但是，并非所有问题中，$w(j,i)$ 都这样容易计算．因此，除了基本情形外，本文还讨论了 $w(j,i)$ 只具有如下性质时，利用决策单调性优化 DP 的方法：

- **移动访问**：$w(j,i)$ 可以从 $w(j \pm 1, i)$ 或 $w(j, i \pm 1)$ 以 $O(1)$ 时间转移得到．（类似 莫队算法 →13.4.4.2@p3209 中的情形）
- **动态计算**：$w(j,i)$ 的计算依赖于 $\{f(j') : j' < j\}$．这意味着 f 和 w 只能顺次计算．下文介绍了不限制区间个数的区间分拆问题，它就属于这一情形．

这两条性质并不互斥，可能存在 $w(j,i)$ 既需要动态计算，又只支持移动访问的情形．

分治

要求解所有状态，只需要求解所有最优决策点．为了对所有 $1 \le i \le n$ 求解 $\operatorname{opt}(i)$，首先计算 $\operatorname{opt}(n/2)$，而后分别计算 $1 \le i < n/2$ 和 $n/2 < i \le n$ 上的 $\operatorname{opt}(i)$，注意此时已知前半段的 $\operatorname{opt}(i)$ 必然位于 1 和 $\operatorname{opt}(n/2)$ 之间（含端点），而后半段的 $\operatorname{opt}(i)$ 必然位于 $\operatorname{opt}(n/2)$ 和 n 之间（含端点）．对于两个子区间，也

类似处理，直至计算出每个问题的最优决策. 在分治的过程中记录搜索的上下边界，就可以保证算法复杂度控制在 $O(n \log n)$. 递归树层数为 $O(\log n)$，而每层中，单个决策点至多计算两次，所以总的计算次数是 $O(n \log n)$.

参考实现

```
val_t w(int j, int i);  // 成本函数
val_t f[N];             // 最优值
int opt[N];             // 最小最优决策

// 递归求解 [l,r] 中的问题
// 已知它们的最小最优决策点一定出现在区间 [opt_l, opt_r] 中
void calc(int l, int r, int opt_l, int opt_r) {
  int mid = (l + r) / 2;
  // 求问题 mid 的最优决策点
  for (int j = opt_l; j <= std::min(opt_r, mid); ++j) {
    if (w(j, mid) < f[mid]) {
      f[mid] = w(j, mid);
      opt[mid] = j;
    }
  }
  // 根据决策单调性得出左右两部分的决策区间，递归处理
  if (l < mid) calc(l, mid - 1, opt_l, opt[mid]);
  if (r > mid) calc(mid + 1, r, opt[mid], opt_r);
}

// 求解整个区间 [1,n] 的问题
void solve(int n) {
  // 每次调用递归函数前，都需要清空数组 f
  std::fill(f + 1, f + n + 1, inf);
  // 最开始时，只知道问题 [1,n] 的所有决策点都一定在 [1,n] 中
  calc(1, n, 1, n);
}

```

除了随机访问的基本情形外，分治算法还可以应用于 $w(j,i)$ 只支持移动访问的情形. 只需要在计算过程中维护游标 (j,i) 和相应函数值 $w(j,i)$，在需要查询新的值时，将游标暴力移动到当前位置并更新函数值即可. 这样做的时间复杂度仍然为 $O(n \log n)$. 对此更为详细的讨论，可以参考下文 简化 LARSCH 算法 →7.15@p810 一节. 但是，分治算法无法解决 $w(j,i)$ 需要动态计算的情形，因为分治算法没有办法在左半区间问题尚未解决时，就计算出区间中点处的最小最优决策 $\mathrm{opt}(n/2)$.

二分队列

注意到对于每个决策点 j，能使其成为最小最优决策点的问题 i 必然构成一个区间. 可以通过单调队列记录到目前为止每个决策点可以解决的问题的区间，这样，问题的最优解自然可以通过队列中记录的决策点计算得到.

具体地，算法需要顺次遍历决策点. 当遍历到决策点 k 时，队列中需要记录到目前为止每个可行的决策点 j 和能够解决的问题区间左右端点 l_j 和 r_j 构成的 **三元组**. 对于给定区间 $[l_j, r_j]$ 内的问题，j 应该是到目前为止考虑过的决策点（即区间 $[1,k]$ 中的决策点）中最小最优的. 每时每刻，队列中存储的决策未必是连续的，但是尚未解决的问题 $[k,n]$ 应该是队列中存储的问题区间的不交并.

为了说明队列更新过程中，决策点 j 是最小最优决策的问题 i 总构成一段连续的区间，需要适当加强前文的结论：

推论 1

设 $\mathrm{opt}_k(i)$ 是仅考虑 $[1,k]$ 中的决策时，问题 i 的最小最优决策．如果 w 满足四边形不等式，那么对于任意 $i_1 < i_2$，必然成立 $\mathrm{opt}_k(i_1) \le \mathrm{opt}_k(i_2)$．

证明

设 M 是充分大的正实数．函数 $w'(j,i) = w(j,i) + M[j > k]$ 仍然满足四边形不等式，其中，$[\cdot]$ 是 Iverson 括号．考虑以 w' 为成本函数的辅助 DP．在辅助 DP 中，对于任何问题 i，决策 $j > k$ 都不可能是最小最优的，即 $\mathrm{opt}'(i) = \mathrm{opt}'_k(i) = \mathrm{opt}_k(i)$．对辅助 DP 应用定理 1 就得到本推论．

该算法过程如下：[1]

- 初始时，队列是空的．类似于单调队列，每次考虑下一个决策 j 的时候，都需要进行出队和入队操作．
- **出队**：首先将上一个问题 $j-1$ 从队列中移除．如果队首的决策能够解决的问题的右端点恰为 $j-1$，直接弹出队首；否则，将队首决策能够解决问题的左端点更新为 j．
- **入队**：要对决策 j 进行入队时，首先比较它和队尾的决策 j'．
  - 如果对于问题 $l_{j'}$，将要入队的决策 j 比已有的决策 j' 严格更优，即 $w(j,l_{j'}) < w(j',l_{j'})$ 时，则弹出队尾的决策 j'．此操作持续到队列为空或队尾的决策 j' 比起 j 对于问题 $l_{j'}$ 更优时为止．
  - 如果队列已空，入队 (j,j,n)，即认为决策 j 是尚未解决的所有问题的最优解．
  - 如果队尾决策 j' 对于问题 $r_{j'}$ 同样不劣于将入队的决策 j，那么当 $r_{j'} < n$ 时，入队 $(j, r_{j'}+1, n)$，表示 j 是问题 $[r_{j'}+1, n]$ 的最小最优决策；否则，不需要入队 j，因为它并不比已有的决策更优．
  - 最后的情形是，队尾决策 j' 比起要入队的决策 j 对于问题 $l_{j'}$ 严格更优，而对于问题 $r_{j'}$ 严格更劣．这说明，存在问题 $i \in (l_{j'}, r_{j'}]$ 使得问题 $[l_{j'}, i-1]$ 的最小最优决策为 j' 且问题 $[i, r_{j'}]$ 的最小最优决策为 j．因而，需要通过 **二分** 找到最小的 $i \in [l_{j'}, r_{j'}]$ 使得 $w(j,i) < w(j',i)$，再将队尾的区间右端点 $r_{j'}$ 修改为 $i-1$，并入队 (j,i,n)．
- 处理完决策 j 后，就已经处理了所有到 j 为止的决策．此时，队首决策就是问题 j 的最小最优决策，可以记录相应的最优解．

参考实现

```
val_t w(int j, int i);  // 成本函数
val_t f[N];             // 最优值
int opt[N];             // 最小最优决策
int lt[N], rt[N];       // 决策 j 可以解决的问题区间 [l_j,r_j]

// 求解整个区间 [1,n] 的问题
void solve(int n) {
  std::deque<int> dq;  // 存储所有可行决策的单调队列

  // 顺次考虑所有问题和决策，下标从 1 开始
  for (int j = 1; j <= n; ++j) {
    // 出队
    if (!dq.empty() && rt[dq.front()] < j) dq.pop_front();
    if (!dq.empty()) lt[dq.front()] = j;
    // 入队
```

[1]算法描述中提到的「更劣」和「更优」都应看做是在描述先比较函数值大小再比较决策点大小的字典序．在这一字典序下，「更优」意味着要么函数值更小，要么函数值一样但是决策点更小．

```
    while (!dq.empty() && w(j, lt[dq.back()]) < w(dq.back(), lt[dq.back()]))
{
      dq.pop_back();
    }
    if (dq.empty()) {
      lt[j] = j;
      rt[j] = n;
      dq.emplace_back(j);
    } else if (w(j, rt[dq.back()]) >= w(dq.back(), rt[dq.back()])) {
      if (rt[dq.back()] < n) {
        lt[j] = rt[dq.back()] + 1;
        rt[j] = n;
        dq.emplace_back(j);
      }
    } else {
      int ll = lt[dq.back()], rr = rt[dq.back()], i = rr;
      // 二分
      while (ll <= rr) {
        int mm = (ll + rr) / 2;
        if (w(j, mm) < w(dq.back(), mm)) {
          i = mm;
          rr = mm - 1;
        } else {
          ll = mm + 1;
        }
      }
      rt[dq.back()] = i - 1;
      lt[j] = i;
      rt[j] = n;
      dq.emplace_back(j);
    }
    // 计算
    f[j] = w(dq.front(), j);
    opt[j] = dq.front();
  }
}

```

类似于单调队列，每个决策点至多入队一次，出队一次．其中，出队是 $O(1)$ 的，而入队是 $O(\log n)$ 的（可能需要二分），所以总的时间复杂度是 $O(n \log n)$．

由于二分队列算法顺次考虑所有问题和决策点，它可以应用于 $w(j,i)$ 需要动态计算的情形．这是它相较于分治算法的优势．但是，因为算法中的二分步骤依赖于对 $w(j,i)$ 的随机访问，它无法应用于 $w(j,i)$ 只支持移动访问的情形．

例题 1：「POI2011」Lightning Conductor[1]

给定一个长度为 n 的序列 $a_1, a_2, \cdots, a_n$，要求对于每一个 $1 \le i \le n$，找到最小的非负整数 f_i 满足

$$\forall j \in [1,n] : a_j \le a_i + f_i - \sqrt{|\, i-j \,|}.$$

思路

显然, 经过不等式变形, 我们可以得到待求整数 $f_i = \max_j\{a_j + \sqrt{|\ i-j\ |} - a_i\}$. 不妨先考虑 $j \le i$ 的情况（另外一种情况类似），此时我们可以得到状态转移方程:

$$f_i = -\min_{j\le i}\{-a_j - \sqrt{i-j} + a_i\}.$$

根据 $-\sqrt{x}$ 的凸性，我们很容易得出（后文将详细描述）函数 $w(j,i) = -a_j - \sqrt{i-j} + a_i$ 满足四边形不等式，因此套用上述的算法便可在 $O(n\log n)$ 的时间内解决此题了.

实现

```cpp
#include <algorithm>
#include <cmath>
#include <deque>
#include <iostream>
#define all \
  for (auto i : {&q, &l, &r}) (*i)
using namespace std;
int n, a[500009], ans[500009];
deque<int> q, l, r;

double f(int i, int j) { return a[i] + sqrtl(j - i); }

void work() {
  all.clear();
  for (int i = 0; i < n; ++i) {
    if (q.size() && r.front() < i) all.pop_front();  // 队首出队
    if (q.size()) l.front() = i;
    for (; q.size() && f(q.back(), l.back()) <= f(i, l.back());)  // 队尾出队
      all.pop_back();
    if (q.empty())  // 入队
      q.emplace_back(i), l.emplace_back(i), r.emplace_back(n);
    else if (f(q.back(), n) < f(i, n)) {
      int ll = l.back(), rr = n, mid;
      for (; ll <= rr;) {
        mid = (ll + rr) >> 1;
        if (f(q.back(), mid) < f(i, mid))
          rr = mid - 1;
        else
          ll = mid + 1;
      }
      r.back() = rr;
      q.emplace_back(i), l.emplace_back(ll), r.emplace_back(n);
    }
    ans[i] = max(ans[i], (int)(ceil(f(q.front(), i))) - a[i]);
  }
}

int main() {
  cin >> n;
  for (int i = 0; i < n; cin >> a[i++]);
  work();
```

```
  reverse(a, a + n);
  reverse(ans, ans + n);
  work();
  reverse(ans, ans + n);
  for (int i = 0; i < n; cout << ans[i++] << '\n');
}

```

简化 LARSCH 算法

前两种算法都无法处理 $w(j,i)$ 既需要动态计算，也只支持移动访问的情形. 本节介绍一种能够同时克服这两种困难的算法. 它是 Larmore 和 Schieber 在 1991 年提出的 LARSCH 算法[2]的简化版本，故称为 **简化 LARSCH 算法**. 该算法的原始版本可以在 $O(n)$ 时间内解决决策单调性 DP 问题，但实现较复杂，本文不做介绍.

仍然考虑分治求解. 求解区间 $(l,r]$ 内的问题时，假设如下信息已知:

- 区间 $[1,l]$ 中问题 i 的最小最优决策 $\mathrm{opt}(i)$ 和最优值，以及
- 仅考虑区间 $[1,l]$ 中的决策，问题 r 的最小最优决策 $\mathrm{opt}_l(r)$ 和最优值.

区间 $(l,r]$ 内的问题求解结束时，需要得到区间 $(l,r]$ 内问题的最小最优决策和最优值.

设 mid 为区间 $(l,r]$ 的中点. 求解过程如下:

1. 遍历决策 $j \in [\mathrm{opt}(l), \mathrm{opt}_l(r)]$，更新问题 mid 的最小最优决策和最优值.
2. 递归求解区间 $(l, mid]$ 中的问题.
3. 遍历决策 $j \in (l, mid]$，更新问题 r 的最小最优决策和最优值.
4. 递归求解区间 $(mid, r]$ 中的问题.

对整个区间 $[1,n]$ 执行递归前，首先需要用决策 $j=1$ 更新问题 $i \in \{1,n\}$. 该算法在递归到 $l=r$ 时就终止.

参考实现

```
val_t w(int j, int i);  // 成本函数
val_t f[N];             // 最优值
int opt[N];             // 最小最优决策

// 用决策 j 更新问题 i
void check(int j, int i) {
  if (w(j, i) < f[i]) {
    f[i] = w(j, i);
    opt[i] = j;
  }
}

// 递归求解区间 (l, r] 内的问题
void calc(int l, int r) {
  int mid = (l + r + 1) / 2;
  for (int j = opt[l]; j <= opt[r]; ++j) check(j, mid);
  if (mid < r) calc(l, mid);
  for (int j = l + 1; j <= mid; ++j) check(j, r);
```

[2]Larmore, Lawrence L., and Baruch Schieber. "On-line dynamic programming with applications to the prediction of RNA secondary structure." Journal of Algorithms 12, no. 3 (1991): 490-515.

```
  if (mid > l) calc(mid, r);
}

// 求解整个区间 [1, n] 内的问题
void solve(int n) {
  // 清空 f 数组
  std::fill(f + 1, f + n + 1, inf);
  // 初始化
  check(1, 1);
  check(1, n);
  // 递归求解区间 (1, n] 内的问题
  calc(1, n);
}

```

首先，可以说明这一算法的正确性．为此，只要检查每一递归求解的步骤（即步骤 2 和 4）前都满足上文给出的前提条件．由前一节得到的推论 1，有 $\mathrm{opt}(l) = \mathrm{opt}_l(l) \le \mathrm{opt}_l(mid) \le \mathrm{opt}_l(r)$，所以步骤 1 后，$\mathrm{opt}_l(mid)$ 是已知的，进而执行步骤 2 前递归求解区间 $(l, mid]$ 中问题的前提条件成立．由于之前 $\{\mathrm{opt}(i) : i \in [1, l]\}$ 是已知的，步骤 2 又得到了 $\{\mathrm{opt}(i) : i \in (l, mid]\}$ 的取值，所以这之后 $\{\mathrm{opt}(i) : i \in [1, mid]\}$ 都是已知的；同时，由于之前 $\mathrm{opt}_l(r)$ 是已知的，步骤 3 后，$\mathrm{opt}_{mid}(r)$ 就也是已知的．因此，执行步骤 4 前递归求解区间 $(mid, r]$ 中问题的前提条件也成立．

然后，需要说明该算法的复杂度仍然是 $O(n \log n)$ 的．递归树的层数是 $O(\log n)$ 的．对于递归树中的同一层的每个结点，分别遍历区间 $[\mathrm{opt}(l), \mathrm{opt}_l(r)]$ 和 $(l, mid]$ 中的决策．因为 $\mathrm{opt}(l) \le \mathrm{opt}_l(r) \le \mathrm{opt}(r)$，所以在同一层中，每个决策点只遍历了 $O(1)$ 次．故而，递归树的每一层中总的遍历次数是 $O(n)$ 的．假设单次访问或计算 $w(j, i)$ 的复杂度是 $O(1)$ 的，算法的时间复杂度就是 $O(n \log n)$ 的．

对于一些 $w(j, i)$ 只支持移动访问的情形，这一算法的复杂度仍然是 $O(n \log n)$ 的．此时，需要为算法流程中的步骤 1 和步骤 3 分别维护游标 (j, i) 和 $w(j, i)$ 的当前取值．每次需要访问新的值时，需要将游标 (j, i) 从上一次访问时的位置暴力更新到当前位置，并对函数值 $w(j, i)$ 进行转移．容易验证，当对递归树进行遍历时，这些暴力更新的总次数是 $O(n \log n)$ 的．所以，算法的时间复杂度仍然是 $O(n \log n)$ 的．

非随机访问的复杂度证明

只需要证明游标移动的总次数是 $O(n \log n)$ 的．设 A 和 B 分别为步骤 1 和 3 对应的游标．实际上可以保证：在求解区间 $(l, r]$ 的问题之前，游标 A 处于位置 $(\mathrm{opt}(l), l)$，游标 B 处于位置 (l, l)；而在这之后，游标 A 处于位置 $(\mathrm{opt}(r), r)$，游标 B 处于位置 (r, r)．

考虑构造如下游标移动规则．步骤 1 中，可以令游标 A 沿着路径

$$(\mathrm{opt}(l), l) \to (\mathrm{opt}(l), mid) \to (\mathrm{opt}_l(r), mid) \to (\mathrm{opt}(l), mid) \to (\mathrm{opt}(l), l)$$

移动．此时，游标 A 和 B 均处于求解区间 $(l, mid]$ 的问题之前的规定位置上．步骤 2 中，按规定，游标 A 将移动到 $(\mathrm{opt}(mid), mid)$，游标 B 将移动到 (mid, mid)．步骤 3 中，可以令游标 B 沿着路径

$$(mid, mid) \to (l, mid) \to (l, r) \to (mid, r) \to (mid, mid)$$

移动．此时，游标 A 和 B 均处于求解区间 $(mid, r]$ 的问题之前的规定位置上．步骤 4 中，按规定，游标 A 将移动到 $(\mathrm{opt}(r), r)$，游标 B 将移动到 (r, r)．两个游标均在结束求解区间 $(l, r]$ 的问题的规定位置上．因此，这一移动规则符合上述规定．而且，该移动规则足以完成步骤 1 和 3 中的所有计算．直接计算该规则中游标的移动次数可知，步骤 1 需要

$$2(\mathrm{opt}_l(r) - \mathrm{opt}(l)) + 2(mid - l)$$

次移动，步骤 3 需要

$$2(mid-l)+2(r-mid)=2(r-l)$$

次移动．将这些移动次数对递归树中的所有结点求和，利用同一层中的所有 $[l,r]$ 和 $[\mathrm{opt}(l),\mathrm{opt}_l(r)]$ 至多只在端点处重合这一性质，就可以说明总的移动次数是 $O(n\log n)$ 的．

由于上述移动规则比起实际计算时游标的移动设置了更多的途径点，所以游标的实际移动次数不会超过该规则下移动次数的估计．因此，游标的实际移动次数也是 $O(n\log n)$ 的．

由于该算法在求解区间 $(l,r]$ 的问题前，已经计算出了区间 $[1,l]$ 中的最优解 $f(i)$，所以该算法也可以应用于 $w(j,i)$ 需要动态计算的情形．

区间分拆问题

考虑将某个区间拆分成若干个子区间的问题．形式化地说，将给定区间 $[1,n]$ 拆分成 $[a_1,b_1],\cdots,[a_k,b_k]$，其中，$a_1=1$，$b_k=n$，以及 $b_i+1=a_{i+1}$ 对任意 $i<k$ 都成立．对于给定拆分，成本为 $\sum_{i=1}^{k}w(a_i,b_i)$．问题要求最小化这一成本．可以列出如下的 1D1D 状态转移方程．

$$f(i)=\min_{1\le j\le i}f(j-1)+w(j,i)\qquad(1\le i\le n)$$

这里，$f(0)=0$．注意到，只要 $w(j,i)$ 满足四边形不等式，$f(j-1)+w(j,i)$ 必然满足四边形不等式，因为第一项并不包括 j 和 i 的交叉项，在混合差分时会消去．但是由于成本函数依赖于前面的子问题，这一转移只能够顺序计算，所以无法应用前文描述的第一种分治算法，通常只适合应用二分队列算法或简化 LARSCH 算法．算法复杂度为 $O(n\log n)$．

限制区间个数的情形

上述问题可以加强为限制区间个数的情形，即问题指定将区间拆分成 m 个子区间．此时需要将拆分后的区间个数作为转移状态的一维．相应地，有 2D1D 状态转移方程如下．

$$f(k,i)=\min_{1\le j\le i}f(k-1,j-1)+w(j,i)\qquad(1\le k\le m,1\le i\le n)\tag{2}$$

这里，$f(0,0)=0$，$f(0,i)=f(k,0)=\infty$ 对任意 $1\le k\le m$ 和 $1\le i\le n$ 都成立．和上文同样的道理，这里的 $f(k-1,j-1)+w(j,i)$ 必然满足四边形不等式．此时对于第 k 层的计算，并不再依赖于该层的结果，所以对于每一层，都可以通过上一节描述的任何算法进行计算，此时算法复杂度为 $O(mn\log n)$．

对于这一问题，利用决策单调性，实际上还存在其他的优化算法．第二种优化思路依赖于如下结果．这种优化算法和下文详细描述的 Knuth 优化算法十分相似．

定理 2

若 w 满足四边形不等式，则对于问题 (2) 成立 $\mathrm{opt}(k-1,i)\le\mathrm{opt}(k,i)\le\mathrm{opt}(k,i+1)$．

证明

第二个不等式只是第 k 层的决策单调性．关键在于第一个不等式．

下证 $\mathrm{opt}(k,i)\le\mathrm{opt}(k+1,i)$．假设有如下两个区间 $[1,i]$ 的分划（逆序标号）：$[a_k,d_k],\cdots,[a_1,d_1]$ 和 $[b_{k+1},c_{k+1}],\cdots,[b_1,c_1]$．其中，每个区间的左端点都是其右端点处对应问题的最小最优决策；同样地，从右向左考虑所有可能的分划，右端点也是左端点对应问题的最小最优决策．例如，d_j 和 c_j 分别是将 $[a_j,i]$ 和 $[b_j,i]$ 分成 j 段左起第一个区间右端点的最小最优决策．根据决策单调性，如果 $a_{j-1}>b_{j-1}$，亦即 $d_j>c_j$，那么必然有 $a_j>b_j$．由此，如果所证不成立，则有 $a_1>b_1$．进而可以归纳地证明 $a_k>b_k$．这显然与所设矛盾．由此得证．

第一个不等式可以另证如下．同样考虑上面证明中的两个分划．如果所证命题不成立，则有 $a_1>b_1$，但是由于有 $a_k<b_k$，我们可以找到最小的 $j>1$ 使得 $a_j\le b_j$．进而，此时有 $a_{j-1}>b_{j-1}$，故

$d_j > c_j$. 我们找到了一组区间满足 $a_j \le b_j \le c_j < d_j$. 考虑将这两个分拆重新组合的结果. 考虑分拆 $[b_{k+1}, c_{k+1}], \cdots, [b_{j+1}, c_{j+1}], [b_j, d_j], [a_{j-1}, d_{j-1}], \cdots, [a_1, d_1]$, 共 $(k+1)$ 段, 于是由前设的最优性可推知,

$$\begin{aligned} &w(b_{k+1}, c_{k+1}) + \cdots + w(b_{j+1}, c_{j+1}) + w(b_j, c_j) + w(b_{j-1}, c_{j-1}) + \cdots + w(b_1, c_1) \\ \le\ &w(b_{k+1}, c_{k+1}) + \cdots + w(b_{j+1}, c_{j+1}) + w(b_j, d_j) + w(a_{j-1}, d_{j-1}) + \cdots + w(a_1, d_1). \end{aligned}$$

同样地, 考虑分拆 $[a_k, d_k], \cdots, [a_{j+1}, d_{j+1}], [a_j, c_j], [b_{j-1}, c_{j-1}], \cdots, [b_1, c_1]$, 共 k 段, 则有

$$\begin{aligned} &w(a_k, d_k) + \cdots + w(a_{j+1}, d_{j+1}) + w(a_j, d_j) + w(a_{j-1}, d_{j-1}) + \cdots + w(a_1, d_1) \\ <\ &w(a_k, d_k) + \cdots + w(a_{j+1}, d_{j+1}) + w(a_j, c_j) + w(b_{j-1}, c_{j-1}) + \cdots + w(b_1, c_1). \end{aligned}$$

此时, 不等号是严格的, 因为 $a_1 > b_1$, 但是按假设, a_1 是所有 k 段分拆最末一段的左端点中最小最优的. 两个不等式条件相加, 得到 $w(b_j, c_j) + w(a_j, d_j) < w(b_j, d_j) + w(a_j, c_j)$, 这有悖于四边形不等式. 故而原结论得证.

利用这一结果, 我们可以限制决策 j 的搜索范围. 算法实现时, 对 k 正向遍历, 对 i 逆向遍历, 在之前已确定的上下界范围内暴力搜索 j 就可以保证 $O(n(n+m))$ 的算法复杂度.

⚠ 注意

这里算法复杂度不是 $O(nm)$ 的. 正确的复杂度计算需要考虑 $n \times m$ 维状态矩阵. 因为对于问题 (k, i) 只需要考虑 $\mathrm{opt}(k-1, i) \le j \le \mathrm{opt}(k, i+1)$ 中的决策, 所以每条次对角线上 (即 $i-k$ 为一定值) 的问题所需遍历的决策总数为 $O(n)$ 的. 这样的对角线共计 $(n+m)$ 条, 故而总的时间复杂度为 $O(n(n+m))$.

最后一种优化方法来源于如下的观察.

定理 3

若 w 满足四边形不等式, 则问题 (2) 的最优解 $g(k) := f(k, n)$ 是关于 k 的凸函数.

证明

下证 $g(k-1) + g(k+1) \ge 2g(k)$. 为此, 考虑分成 $(k-1)$ 段和 $(k+1)$ 段的最优分划, 分别是 $[a_1, d_1], \cdots, [a_{k-1}, d_{k-1}]$ 和 $[b_1, c_1], \cdots, [b_{k+1}, c_{k+1}]$. 取最小的 $1 \le j \le k-1$ 使得 $c_{j+1} \le d_j$, 其存在性可由 $c_k < n = d_{k-1}$ 推知. 根据其最小性得知, $b_{j+1} > a_j$. 所以, $a_j < b_{j+1} \le c_{j+1} \le d_j$. 与上文类似, 交换两个现有分拆的后半段, 可以得到如下两个区间分拆:

$$\begin{aligned} &[a_1, d_1], \cdots, [a_{j-1}, d_{j-1}], [a_j, c_{j+1}], [b_{j+2}, c_{j+2}], \cdots, [b_{k+1}, c_{k+1}], \\ &[b_1, c_1], \cdots, [b_j, c_j], [b_{j+1}, d_j], [a_{j+1}, d_{j+1}], \cdots, [a_{k-1}, d_{k-1}]. \end{aligned}$$

两个所得分拆都是 k 段的, 所以由最优性条件可知

$$\begin{aligned} 2g(k) \le\ &w(a_1, d_1) + \cdots + w(a_{j-1}, d_{j-1}) + w(a_j, c_{j+1}) + w(b_{j+2}, c_{j+2}) + \cdots + w(b_{k+1}, c_{k+1}) \\ &+ w(b_1, c_1) + \cdots + w(b_j, c_j) + w(b_{j+1}, d_j) + w(a_{j+1}, d_{j+1}) + \cdots + w(a_{k-1}, d_{k-1}) \\ \le\ &w(a_1, d_1) + \cdots + w(a_{j-1}, d_{j-1}) + w(a_j, d_j) + w(a_{j+1}, d_{j+1}) + \cdots + w(a_{k-1}, d_{k-1}) \\ &+ w(b_1, c_1) + \cdots + w(b_j, c_j) + w(b_{j+1}, c_{j+1}) + w(b_{j+2}, c_{j+2}) + \cdots + w(b_{k+1}, c_{k+1}) \\ =\ &g(k-1) + g(k+1). \end{aligned}$$

这里第二个不等式正是四边形不等式. 所求凸性由此得证.

这一结论保证了可以通过 WQS 二分 (国外称 Aliens Trick) 的方法解决此问题. 具体来说, 考虑带参的成本函数 $w_c(j, i) := w(j, i) + c$, 解决不限制区间个数的问题, 求得其最优解为 $f_c(n)$. 随着实数 c 递增, 相应的最优区间的数目单调递减, 故而可以通过二分的方法找到恰使得最优区间个数等于 m 的参数 c, 则

原题最优解为 $f(m,n)=f_c(n)-cm$. 这里的实数 c 可以看作区间个数限制的 Lagrange 乘子. 该算法的实现有很多细节, 可以参考 WQS 二分→7.15.6@p852 页面. 这一算法的时间复杂度为 $O(n\log n\log C)$, 这里 C 为参数 c 的取值范围的大小.

对于限制区间个数的区间分拆问题的三种算法, 在不同的数据范围时表现各有优劣, 需要结合具体的题目选择合适的算法.

例题 3：P4767 [IOI2000] 邮局 加强版[2] P6246 [IOI2000] 邮局 加强版 加强版[3]

高速公路旁边有一些村庄. 高速公路表示为整数轴, 每个村庄的位置用单个整数坐标标识. 没有两个在同样地方的村庄. 两个位置之间的距离是其整数坐标差的绝对值.

邮局将建在一些, 但不一定是所有的村庄中. 为了建立邮局, 应选择他们建造的位置, 使每个村庄与其最近的邮局之间的距离总和最小.

你要编写一个程序, 已知村庄的位置和邮局的数量, 计算每个村庄和最近的邮局之间所有距离的最小可能的总和.

思路

每个村庄有其最近的邮局, 那么每个邮局也有其管辖的村庄, 易知这是一个区间.

考虑把这 n 个村庄分成 m 个区间, 再在每个区间中决出一个邮局.

根据数学知识, 对于区间 $[i,j]$, 邮局应该建在第 $\left\lfloor\frac{i+j}{2}\right\rfloor$ 个村庄处. 使用前缀和容易算出 $w(i,j)$.

问题转化为限制区间个数的区间分拆问题. 可以证明, w 函数满足四边形不等式. 直接应用上述优化方法即可.

实现 1，前文第二种优化，复杂度 $O(n(n+m))$

```cpp
#include <iostream>
constexpr int N = 3009;
constexpr int M = 309;
using namespace std;
int n, m, a[N], s[N], g[M][N], p[M][N];

int f(int i, int j) {
  int k = (i + j) >> 1;
  return a[k] * (k - i + 1) - (s[k] - s[i - 1]) + (s[j] - s[k]) -
         a[k] * (j - k);
}

int main() {
  cin >> n >> m;
  for (int i = 1; i <= n; cin >> a[i], s[i] = s[i - 1] + a[i], ++i);
  for (int i = 1; i <= n; ++i) g[0][i] = 1 << 30, p[m + 1][i] = i;
  for (int i = 1; i <= m; ++i) p[i][0] = 1;
  for (int i = 1; i <= n; ++i)
    for (int j = m; j; --j) {
      g[j][i] = 1 << 30;
      for (int k = p[j][i - 1]; k <= p[j + 1][i]; ++k) {
        int tmp = g[j - 1][k - 1] + f(k, i);
        if (tmp < g[j][i]) g[j][i] = tmp, p[j][i] = k;
```

```
      }
    }
  cout << g[m][n];
}

```

实现 2，WQS 二分，复杂度 $O(n \log n \log C)$

```
#include <deque>
#include <iostream>
#define all \
  for (auto i : {&q, &l, &r}) (*i)
using namespace std;
long long n, m, a[500009], s[500009], u, v, w, sum[500009], cnt[500009];
deque<int> q, l, r;

long long f(int i, int j) {
  int k = (i + j) >> 1;
  return sum[i - 1] + v + a[k] * (k - i + 1) - (s[k] - s[i - 1]) +
         (s[j] - s[k]) - a[k] * (j - k);
}

void work() {
  all.clear();
  for (int i = 1; i <= n; ++i) {
    if (q.size() && r.front() < i) all.pop_front();  // 队首出队
    if (q.size()) l.front() = i;
    for (; q.size() && f(q.back(), l.back()) >= f(i, l.back());)  // 队尾出队
      all.pop_back();
    if (q.empty())  // 入队
      q.emplace_back(i), l.emplace_back(i), r.emplace_back((int)n);
    else if (f(q.back(), n) >= f(i, n)) {
      int ll = l.back(), rr = n, mid;
      for (; ll <= rr;) {
        mid = (ll + rr) >> 1;
        if (f(q.back(), mid) >= f(i, mid))
          rr = mid - 1;
        else
          ll = mid + 1;
      }
      r.back() = rr;
      q.emplace_back(i), l.emplace_back(ll), r.emplace_back((int)n);
    }
    sum[i] = f(q.front(), i);
    cnt[i] = cnt[q.front() - 1] + 1;
  }
}

int main() {
  cin >> n >> m;
  for (int i = 1; i <= n; cin >> a[i], s[i] = s[i - 1] + a[i], ++i);
```

```
  for (w = 2e12; u <= w;) {  // wqs二分
    v = (u + w) >> 1;
    work();
    if (cnt[n] < m)
      w = v - 1;
    else
      u = v + 1;
  }
  v = w;
  work();
  cout << sum[n] - m * v;
}

```

区间合并问题

另一类可以通过四边形不等式优化的动态规划问题是区间合并问题，即要将 n 个长度为一的区间 $[i,i]$ 两两合并起来，直到得到区间 $[1,n]$．每次合并 $[j,k]$ 和 $[k+1,i]$ 时都需要支付成本 $w(j,i)$．问题要求找到成本最低的合并方式．对于此类问题，有如下 2D1D 状态转移方程：

$$f(j,i)=\min_{j\le k<i} f(j,k)+f(k+1,i)+w(j,i)\qquad (1\le j<i\le n) \tag{3}$$

其中，初始成本 $f(i,i)=0$．暴力算法的总复杂度为 $O(n^3)$，而当存在决策单调性时，可以优化至 $O(n^2)$ 的算法复杂度．这一算法最早由 Knuth 在解决最优二叉搜索树问题时提出，并由姚储枫进一步研究总结，在国外称为 Knuth's optimization 或 Knuth-Yao speedup．

除了四边形不等式以外，区间合并问题的决策单调性还要求成本函数满足区间包含单调性．

- **区间包含单调性**：如果对于任意 $a\le b\le c\le d$ 均成立

  $w(b,c)\le w(a,d)$,

  则称函数 w 对于区间包含关系具有单调性．

这实质是成本函数的一阶条件，即 $w(j,i)$ 关于 j 递减，关于 i 递增．

引理 1

若 w 满足区间包含单调性和四边形不等式，则状态 $f(j,i)$ 满足四边形不等式．

证明

不妨设 $a\le b\le c\le d$．下证 $f(a,d)+f(b,c)\ge f(a,c)+f(b,d)$．考虑依 $d-a$ 归纳．当 $a=b$ 或 $c=d$ 时，所求即一等式．对于一般的情形，根据 $d'=\mathrm{opt}(a,d)$ 的位置分类讨论．

第一种情况，$c\le d'$ 或 $d'<b$，即 $[b,c]$ 包含于 $[a,d']$ 或 $[d'+1,d]$ 之中．

不妨假设 $c\le d'$，另一种情形同理．此时有

$$\begin{aligned} f(a,d)+f(b,c) &= f(a,d')+f(d'+1,d)+w(a,d)+f(b,c)\\ &\ge f(a,c)+f(b,d')+f(d'+1,d)+w(a,d)\\ &\ge f(a,c)+f(b,d')+f(d'+1,d)+w(b,d)\\ &\ge f(a,c)+f(b,d). \end{aligned}$$

这里，第一个不等式来自于归纳假设 $f(a,c)+f(b,d')\le f(a,d')+f(b,c)$，第二个不等式来自于区间包含单调性 $w(b,d)\le w(a,d)$，第三个不等式来自于最优性条件 $f(b,d)\le f(b,d')+f(d'+1,d)+w(b,d)$．

第二种情况，$b \le d' < c$，即 d' 位于 $[b,c]$ 之中．此时，考虑 $c' = \mathrm{opt}(b,c)$ 的位置．

不妨假设 $c' \le d'$，即 $[b,c']$ 包含于 $[a,d']$ 之中，另一种情形同理．此时有

$$\begin{aligned} f(a,d) + f(b,c) &= f(a,d') + f(d'+1,d) + w(a,d) + f(b,c') + f(c'+1,c) + w(b,c) \\ &\ge f(a,c') + f(c'+1,c) + w(b,c) + f(b,d') + f(d'+1,d) + w(a,d) \\ &\ge f(a,c') + f(c'+1,c) + w(a,c) + f(b,d') + f(d'+1,d) + w(b,d) \\ &\ge f(a,c) + f(b,d). \end{aligned}$$

这里，第一个不等式来自于归纳假设 $f(a,c') + f(b,d') \le f(a,d') + f(b,c')$，第二个不等式来自于四边形不等式 $w(a,c) + w(b,d) \le w(a,d) + w(b,c)$，第三个不等式来自于 $f(a,c)$ 和 $f(b,d)$ 的最优性条件．

定理 4

若 w 满足区间包含单调性和四边形不等式，则问题 (3) 中最小最优决策 $\mathrm{opt}(j,i)$ 满足

$$\mathrm{opt}(j,i-1) \le \mathrm{opt}(j,i) \le \mathrm{opt}(j+1,i). \qquad (j+1<i)$$

证明

引理 1 已经证得 $f(j,i)$ 满足四边形不等式，所以目标函数 $f(j,k) + f(k+1,i) + w(j,i)$ 对于给定 j 作为 (k,i) 的函数满足四边形不等式，所以由定理 1 有，$\mathrm{opt}(j,i-1) \le \mathrm{opt}(j,i)$．注意，不同时含有 (k,i) 的项并不影响四边形不等式成立．类似地，它对于给定 i 作为 (j,k) 的函数也满足四边形不等式，所以 $\mathrm{opt}(j,i) \le \mathrm{opt}(j+1,i)$．即得所证．

利用这一结论，同样可以限制决策点 k 的搜索范围．在这里，正序遍历区间长度 $i-j+1$，再遍历具有同样长度的所有区间 $[j,i]$，暴力搜索 $\mathrm{opt}(j,i-1)$ 和 $\mathrm{opt}(j+1,i)$ 之间的所有 k 求得最优解 $f(j,i)$ 并记录最小最优决策 $\mathrm{opt}(j,i)$．对于同样长度的所有区间，此算法中决策空间总长度是 $O(n)$ 的，而可能的区间长度的数目同样是 $O(n)$ 的，故而总的算法复杂度为 $O(n^2)$ 的．

参考实现

```
val_t w(int j, int i);  // 成本函数
val_t f[N][N];          // 最优值
int opt[N][N];          // 最小最优决策

// 求解整个区间 [1,n] 对应的问题
void solve(int n) {
  // 初始化
  for (int i = 1; i <= n; ++i) {
    f[i][i] = 0;
    opt[i][i] = i;
  }
  // 枚举区间长度
  for (int len = 2; len <= n; ++len) {
    // 枚举长度为 len 的所有区间
    for (int j = 1, i = len; i <= n; ++j, ++i) {
      f[j][i] = inf;
      for (int k = opt[j][i - 1]; k <= opt[j + 1][i]; ++k)
        if (f[j][i] > f[j][k] + f[k + 1][i] + w(j, i)) {
          f[j][i] = f[j][k] + f[k + 1][i] + w(j, i);  // 更新状态值
```

```
20            opt[j][i] = k;  // 更新（最小）最优决策点
21          }
22        }
23      }
24  }
25
```

满足四边形不等式的函数类

为了更方便地证明一个函数满足四边形不等式，我们有以下几条性质：

性质 1：若函数 $w_1(j,i)$ 和 $w_2(j,i)$ 均满足四边形不等式（或区间包含单调性），则对于任意 $c_1, c_2 \geq 0$，函数 $c_1 w_1 + c_2 w_2$ 也满足四边形不等式（或区间包含单调性）.

性质 2：若存在函数 $f(x)$ 和 $g(x)$ 使得 $w(j,i) = f(j) - g(i)$，则函数 w 满足四边形恒等式. 当函数 f 和 g 单调增加时，函数 w 还满足区间包含单调性.

性质 3：设 $h(x)$ 是一个单调增加的凸函数，若函数 $w(j,i)$ 满足四边形不等式并且对区间包含关系具有单调性，则复合函数 $h(w(j,i))$ 也满足四边形不等式和区间包含单调性.

性质 4：设 $h(x)$ 是一个凸函数，若函数 $w(j,i)$ 满足四边形恒等式并且对区间包含关系具有单调性，则复合函数 $h(w(j,i))$ 也满足四边形不等式.

首先需要澄清一点，凸函数（Convex Function）的定义在国内教材中有分歧，此处的凸函数指的是下凸函数，即（可微时）一阶导数单调增加的函数.

证明

前两条性质根据定义很容易证明，下面证明第三条性质，性质 4 的证明过程类似. 由于 $h(x)$ 单调，$h(w(j,i))$ 自然保持对区间包含的单调性. 关键在于四边形不等式的证明.

为此，下面考虑 $a \leq j \leq b \leq c \leq i \leq d$ 上的二阶混合差分.

$$\begin{aligned}\Delta_i\Delta_j h(w(j,i)) = {} & h(w(b,d)) - h\big(w(a,c) + \Delta_j w(j,c) + \Delta_i w(a,i)\big) \\ & + h\big(w(a,c) + \Delta_j w(j,c) + \Delta_i w(a,i)\big) - h\big(w(a,c) + \Delta_j w(j,c)\big) \\ & - h(w(a,c) + \Delta_i w(a,i)) + h(w(a,c)).\end{aligned}$$

这里，根据区间单调性，$\Delta_i w(a,i) := w(a,d) - w(a,c) \geq 0$ 和 $\Delta_j w(j,c) := w(b,c) - w(a,c) \leq 0$. 由于 $h(x)$ 具有凸性，对于 $t_1, t_2 \geq 0$ 成立 $h(x + t_1 - t_2) - h(x + t_1) \leq h(x - t_2) - h(x)$，所以后两行之和必然非正. 同时，由于四边形不等式，$w(b,d) \leq w(a,c) + \Delta_j w(j,c) + \Delta_i w(a,i) = w(b,c) + w(a,d) - w(a,c)$，故而，第一行的差在 $h(x)$ 单调增加的情况下必然也非正. 所以，总的二阶混合差分非正. 此即四边形不等式.

这一证明实际是如下导数证明的离散版本.

$$\frac{\partial^2}{\partial x \partial y} h(w(x,y)) = h''(w(x,y)) \frac{\partial}{\partial x} w(x,y) \frac{\partial}{\partial y} w(x,y) + h'(w(x,y)) \frac{\partial^2}{\partial x \partial y} w(x,y) \leq 0.$$

这在 $h' \geq 0$，$h'' \geq 0$，$w_x \leq 0$，$w_y \geq 0$ 以及 $w_{xy} \leq 0$ 的条件下显然成立. 其中，区间包含单调性给出了 w 的一阶条件，而四边形不等式给出了其二阶条件.

习题

- Codeforces - Ciel and Gondolas[4](Be careful with I/O!)
- SPOJ - LARMY[5]
- Codechef - CHEFAOR[6]

- Hackerrank - Guardians of the Lunatics[7]
- ACM ICPC World Finals 2017 - Money[8]

参考资料与注释

- Quora Answer by Michael Levin[9]
- Video Tutorial by "Sothe" the Algorithm Wolf[10]
- Divide and Conquer DP[11]
- Knuth's Optimization[12]
- Quadrangle Inequality Properties[13]
- 王钦石《浅析一类二分方法》[14]
- 簡易版 LARSCH Algorithm by noshi91[15]
- 四边形不等式和决策单调性 by b6e0_ - 洛谷专栏[16]
- 在线决策单调性的丐版 LARSCH 算法 by Register_int - 洛谷专栏[17]

[1] 「POI2011」 Lightning Conductor
[2] P4767 [IOI2000] 邮局 加强版
[3] P6246 [IOI2000] 邮局 加强版 加强版
[4] Codeforces - Ciel and Gondolas
[5] SPOJ - LARMY
[6] Codechef - CHEFAOR
[7] Hackerrank - Guardians of the Lunatics
[8] ACM ICPC World Finals 2017 - Money
[9] Quora Answer by Michael Levin
[10] Video Tutorial by "Sothe" the Algorithm Wolf
[11] Divide and Conquer DP
[12] Knuth's Optimization
[13] Quadrangle Inequality Properties
[14] 王钦石《浅析一类二分方法》
[15] 簡易版 LARSCH Algorithm by noshi91
[16] 四边形不等式和决策单调性 by b6e0_ - 洛谷专栏
[17] 在线决策单调性的丐版 LARSCH 算法 by Register_int - 洛谷专栏

7.15.5 Slope Trick 优化

引入

对于一类二维 DP 问题，如果它的价值函数 $f(i,x)$ 对于每个固定的 i 都是 x 的凸函数，那么将函数 $f(i,\cdot)$ 整体视为 i 处的状态，并维护它的差分（或斜率）

$$\Delta f(i,x) = f(i,x+1) - f(i,x)$$

而非函数本身，往往能够起到优化转移的效果．这种优化 DP 的思想，就称为 Slope Trick.

> **「斜率」**
>
> 因为大多数题目中涉及的函数都只在整点处取值，所以称它为差分和斜率没有本质区别，本文按照 Slope Trick 这个名词统一称呼它为斜率.

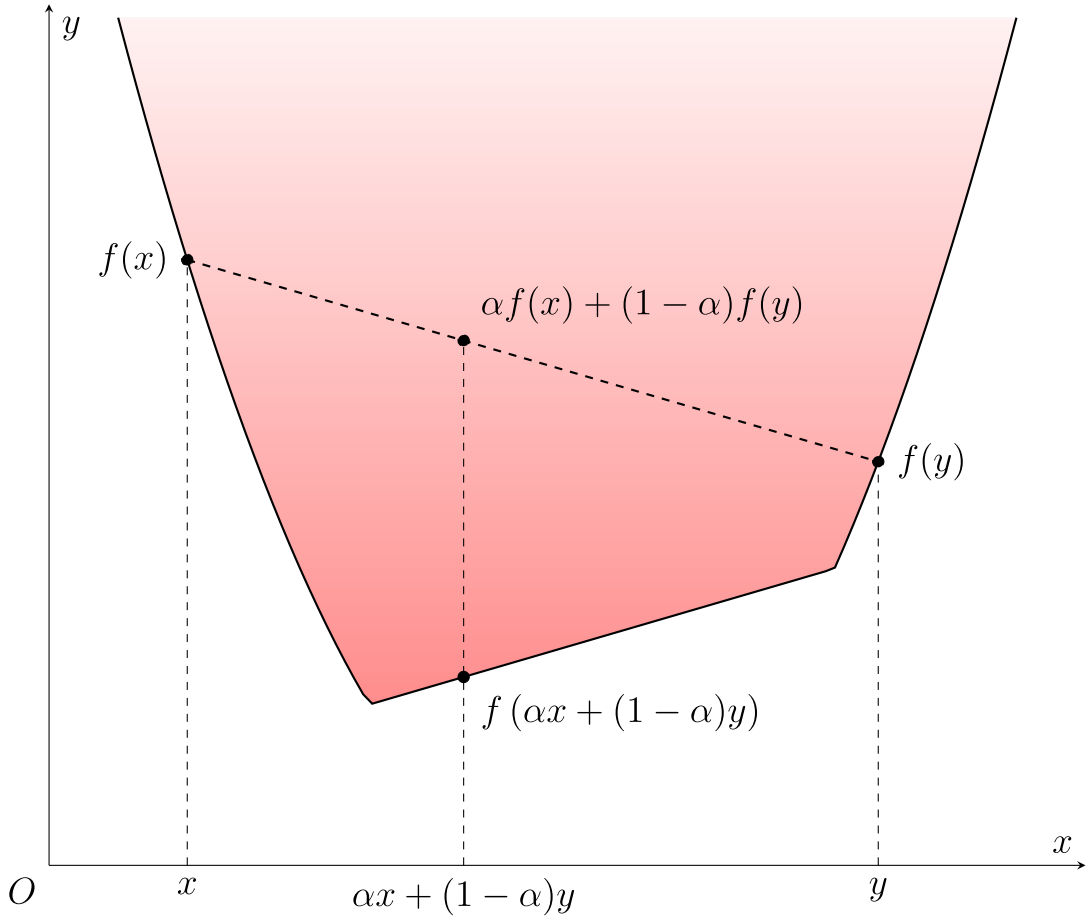
具体题目中，斜率的维护方式可能各不相同．如果斜率的取值范围较窄，维护斜率变化的点（即拐点）更为方便；而如果函数定义域较窄，维护斜率序列本身可能更为方便．更复杂的情形，可能需要同时维护每段斜率的大小和该段的长度．无论具体维护方式是什么，这类问题的本质都是利用状态转移中斜率序列变化较少这一点简化转移．因此，它们都可以称作 Slope Trick.

凸函数

在讨论具体的题目之前，有必要首先了解一下凸函数的基本性质，以及在对凸函数进行各种变换时，它的斜率会如何变化.

实轴上的凸函数

凸函数较为一般的定义是在 $\mathbf{R}$ 上给出的.



自 $\mathbf{R}$ 上的凸函数

如果函数 $f : \mathbf{R} \to \mathbf{R} \cup \{\pm\infty\}$ 对于所有 $x, y \in \mathbf{R}$ 和 $\alpha \in (0, 1)$ 都满足

$$f(\alpha x + (1-\alpha)y) \le \alpha f(x) + (1-\alpha) f(y),$$

就称函数 f 为 **凸函数**（convex function），其中 $\pm\infty$ 的运算法则规定为 $\pm\infty$ 乘以任何正实数或是加上任何实数都等于其自身，且对于任何实数 $x \in \mathbf{R}$ 都有 $-\infty < x < +\infty$.

当然，如果不等号换作 $\ge$，就相应地称它为凹函数[3]. 因为对于凹函数 f，总有 $-f$ 为凸函数，所以本节只考虑凸函数.

本文只考虑正常凸函数

为了避免讨论 $\infty - \infty$ 的取值和额外的复杂分析，本文在讨论凸函数相关概念时，总是默认函数不会取到 $-\infty$，且不总是 $+\infty$. 这样的凸函数称为 **正常凸函数**（proper convex function）. 这对于理解算法竞赛涉及的内容已经足够.

当然，函数 f 往往并不会对所有实数都有定义. 如果函数 f 的定义域仅是 $\mathbf{R}$ 的子集，那么可以将它拓展为 $\mathbf{R}$ 上的函数：

$$\tilde{f}(x) = \begin{cases} f(x), x \in \operatorname{dom} f, \\ +\infty, x \notin \operatorname{dom} f. \end{cases}$$

[3]不同教材对于凸函数的称呼可能不同.

此时，称 f 是凸函数，当且仅当相应的 $\tilde{f}$ 满足上述凸函数的定义．因此，如果没有特别指出，本文提到的凸函数的定义域均是实数集 $\mathbf{R}$．显然，凸函数 f 只能在一个区间（即 $\mathbf{R}$ 的凸子集）上取得有限值．

简单例子

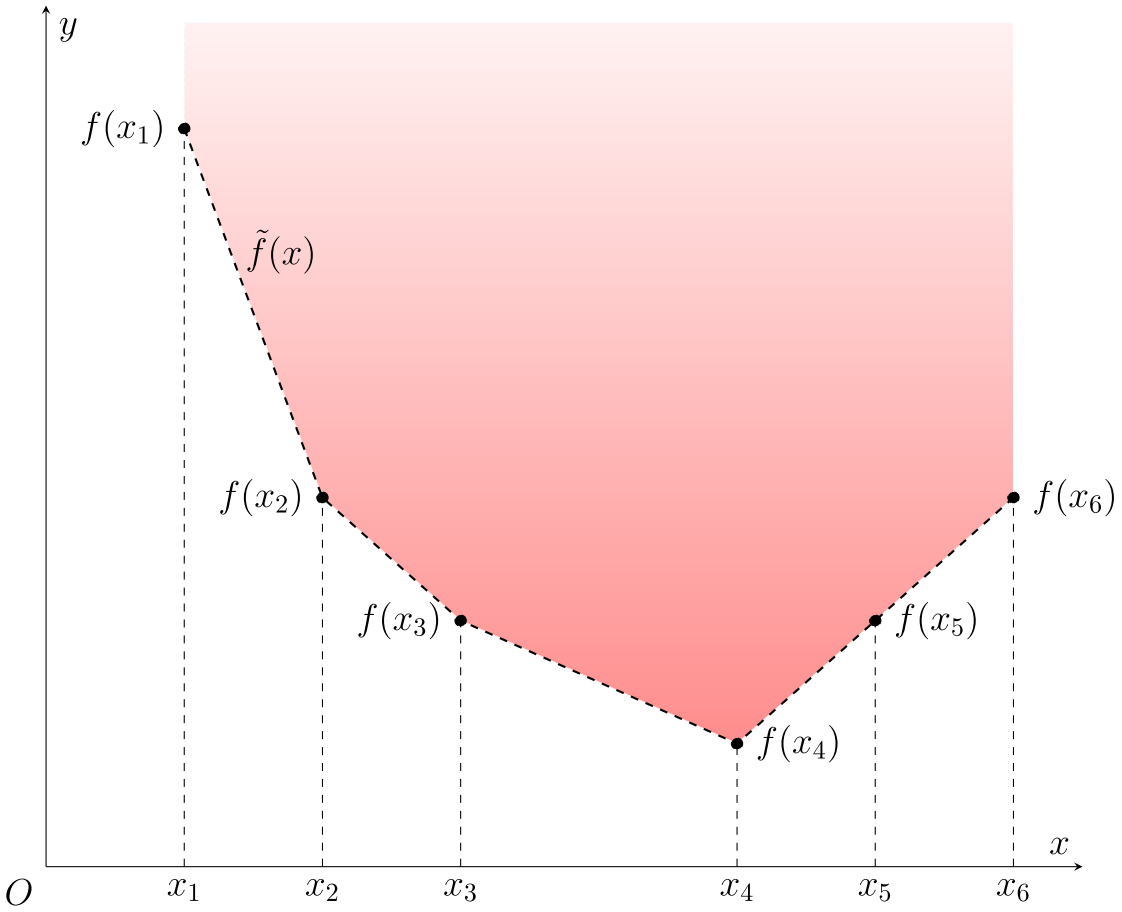
常见的凸函数的例子包括：

1. 常数函数：$f(x)=c$，其中 $c\in\mathbf{R}$；
2. 一次函数：$f(x)=kx+b$，其中 $k,b\in\mathbf{R}$ 且 $k\neq 0$；
3. 绝对值函数：$f(x)=|\,x-a\,|$，其中 $a\in\mathbf{R}$；
4. 任何凸函数限制在某个区间上的结果，例如 $0_{[a,b]}(x)$（在凸分析的语境下也称作 $[a,b]$ 的指示函数）．

当然，可以通过下文提到的保持凸性的变换组合出更为复杂的凸函数．

离散点集上的凸函数

算法竞赛中，很多函数仅在部分整数值处有定义．它们在一般情况下并不是（上文定义的）凸函数，因为它们的定义域不再是凸集．为了处理这种情形，需要单独定义离散点集上的函数的凸性．简单来说，需要首先对函数做线性插值，将其定义域拓展到区间，再判断它的凸性．



离散点集上的凸函数

设 $S\subset\mathbf{R}$ 为离散点集，即对任意闭区间 $[a,b]$，$S\cap[a,b]$ 都是有限集．对于函数 $f:S\to\mathbf{R}\cup\{\pm\infty\}$，可以定义函数 $\tilde{f}:\mathbf{R}\to\mathbf{R}\cup\{\pm\infty\}$ 使得：

- 当 $x\in S$ 时，$\tilde{f}(x)=f(x)$，

- 当 $x \in (\inf S, \sup S) \setminus S$ 时，设 $s_- = \max\{s \in S : s \le x\}$，$s_+ = \min\{s \in S : s \ge x\}$，则 $\tilde{f}(x) = \dfrac{s_+ - x}{s_+ - s_-} f(s_-) + \dfrac{x - s_-}{s_+ - s_-} f(s_+)$，
- 当 $x \notin [\inf S, \sup S]$ 时，$\tilde{f}(x) = +\infty$.

那么，如果 $\tilde{f}(x)$ 是 $\mathbf{R}$ 上的凸函数，就称 $f(x)$ 是 S 上的 **凸函数**.

因为 $\mathbf{R}$ 上的凸函数处理起来更为方便，所以本文在提及凸函数时，若非特别说明，指的都是 $\mathbf{R}$ 上的凸函数. 如果本文中某个函数仅给出了部分整数处的取值，那么它在其他实数处的取值应由定义中的 $\tilde{f}$ 确定，也就相当于直接讨论对应的分段线性函数 $\tilde{f}$.

整数集 $\mathbf{Z}$ 上的凸函数有一个更为直观的等价定义：

$\mathbf{Z}$ 上的凸函数的等价定义

函数 $f : \mathbf{Z} \to \mathbf{R} \cup \{\pm\infty\}$ 是凸的，当且仅当

$$f(x) - f(x-1) \le f(x+1) - f(x)$$

对于所有 $x \in \mathbf{Z}$ 都成立.

证明

这个命题是凸函数的斜率刻画的简单推论.

如果 f 是 $\mathbf{Z}$ 上的凸函数，那么根据斜率弱增，有

$$\Delta f(x-1, x) \le \Delta f(x-1, x+1) \le \Delta f(x, x+1).$$

这就是上述条件.

反过来，如果上述条件成立，那么对于任何 $x_1 < x_2$，都有

$$\Delta f(x_1, x_2) = \frac{1}{x_2 - x_1} \sum_{i=x_1}^{x_2-1} (f(i+1) - f(i)).$$

这相当于对所有满足 $x_1 \le i < x_2$ 的差分的算术平均值. 如果 x_2 增加一，就相当于插入一项更大的差分；如果 x_1 增加一，就相当于移除一项最小的差分. 这两个操作都会使得平均值弱增. 这就说明斜率 $\Delta f(x_1, x_2)$ 弱增，即 f 是 $\mathbf{Z}$ 上的凸函数.

也就是说，只要斜率（差分）单调不减，这个序列就可以看作是 $\mathbf{Z}$ 上的凸函数.

凸函数的两种刻画

其实，用斜率刻画凸函数的方式也可以推广到一般情况.

凸函数的斜率刻画

设 S 为 $\mathbf{R}$ 或它的离散子集，则函数 $f : S \to \mathbf{R} \cup \{\pm\infty\}$ 为凸函数，当且仅当斜率

$$\Delta f(x_1, x_2) = \frac{f(x_2) - f(x_1)}{x_2 - x_1}$$

对于任何 $x_1, x_2 \in S$ 且 $x_1 < x_2$ 都是 x_1 和 x_2 的弱增函数.

证明

对于 $\mathbf{R}$ 上的函数 $f(x)$ 以及 $x_1 < x_2$，对于 $\alpha \in (0, 1)$，令 $x_3 = \alpha x_1 + (1-\alpha) x_2$，那么

$$\Delta f(x_1, x_3) \le \Delta f(x_1, x_2) \le \Delta f(x_3, x_2)$$

就等价于

$$\frac{f(x_3)-f(x_1)}{1-\alpha} \le f(x_2)-f(x_1) \le \frac{f(x_2)-f(x_3)}{\alpha}.$$

这两侧的不等式都等价于 $f(x_3) \le \alpha f(x_1) + (1-\alpha) f(x_2)$，即函数 $f(x)$ 的凸性.

对于 $\mathbf{R}$ 的离散子集 S 上的函数 $f(x)$，斜率弱增这一条件的必要性可以由 $\tilde{f}(x)$ 的凸性推导出来. 现在要证明它的充分性，为此只要证明 $\Delta\tilde{f}(x_1,x_2)$ 也是弱增的. 设 $S=\{s_i\}$ 且 s_i 关于 i 严格递增，并设 $s_{i_1} \le x_1 \le s_{i_1+1}$ 且 $s_{i_2} \le x_2 \le s_{i_2+1}$，自然有 $i_1 \le i_2$. 令 $\Delta_i = \Delta f(s_i, s_{i+1})$，那么，可以证明 $\Delta_{i_1} \le \Delta\tilde{f}(x_1,x_2) \le \Delta_{i_2}$.

这分两种情形. 如果 $i_1 = i_2$，那么 $\Delta_{i_1} = \Delta\tilde{f}(x_1,x_2) = \Delta_{i_2}$，该不等式显然成立. 否则，有

$$\Delta\tilde{f}(x_1,x_2) = \frac{1}{x_2-x_1}\left((s_{i_1+1}-x_1)\Delta_{i_1} + (x_2-s_{i_2})\Delta_{i_2} + \sum_{j=i_1+1}^{i_2-1}(s_{j+1}-s_j)\Delta_j\right).$$

根据 S 上的斜率递增可知，Δ_i 关于 i 递增，所以，$\Delta_{i_1} \le \Delta\tilde{f}(x_1,x_2) \le \Delta_{i_2}$.

利用这个结论，对于 $x_1 < x_2$ 和 $\alpha \in (0,1)$，令 $x_3 = \alpha x_1 + (1-\alpha)x_2$，并取 i_3 使得 $s_{i_3} \le x_3 \le s_{i_3+1}$ 成立，则有

$$\Delta\tilde{f}(x_1,x_3) \le \Delta_{i_3} \le \Delta\tilde{f}(x_3,x_2).$$

代入 x_3 的表达式，就得到 $\tilde{f}(x)$ 的凸性.

斜率单调不减，可以看作是凸函数的等价定义. 正因为凸函数的斜率具有单调性，在维护斜率时，通常需要选择 堆（优先队列）→10.7.1@p2025 或 平衡树→10.15.1@p2213 等数据结构.

本文还会用到凸函数的另一种等价刻画. 对于函数 $f: \mathbf{R} \to \mathbf{R} \cup \{\pm\infty\}$，可以考察平面内函数图像上方的区域，即

$$\operatorname{epi} f = \{(x,y) \in \mathbf{R}^2 : y \ge f(x)\}.$$

这个区域也称为函数 f 的 **上境图**（epigraph）. 函数的凸性，等价于它的上境图的凸性：

凸函数的上境图刻画

函数 $f: \mathbf{R} \to \mathbf{R} \cup \{\pm\infty\}$ 是凸函数，当且仅当 $\operatorname{epi} f$ 是 $\mathbf{R}^2$ 内的凸集.

证明

如果 f 是凸函数，那么对于 $(x_1,y_1),(x_2,y_2) \in \operatorname{epi} f$ 和任意 $\alpha \in (0,1)$，有

$$\alpha y_1 + (1-\alpha)y_2 \ge \alpha f(x_1) + (1-\alpha) f(x_2) \ge f(\alpha x_1 + (1-\alpha)x_2).$$

所以，$\alpha(x_1,y_1) + (1-\alpha)(x_2,y_2) \in \operatorname{epi} f$.

反过来，如果 $\operatorname{epi} f$ 是凸集，那么对于任意 $x_1 < x_2$ 以及 $\alpha \in (0,1)$，有

$$\alpha(x_1, f(x_1)) + (1-\alpha)(x_2, f(x_2)) \in \operatorname{epi} f.$$

这就等价于 $\alpha f(x_1) + (1-\alpha) f(x_2) \ge f(\alpha x_1 + (1-\alpha)x_2)$，即 f 的凸性.

稍后会看到，利用上境图，可以将凸函数的卷积下确界与凸集的 Minkowski 和联系起来.

凸函数的变换

紧接着，本文介绍一些 Slope Trick 中经常遇见的保持凸性的变换.

非负线性组合

对于凸函数 f 和 g 以及非负实数 $\alpha, \beta \ge 0$，函数 $\alpha f + \beta g$ 也是凸函数. 而且，

$$\Delta(\alpha f + \beta g) = \alpha \Delta f + \beta \Delta g.$$

因此，如果维护了凸函数 f 和 g 的斜率，要得到它们的非负线性组合 $\alpha f + \beta g$ 的斜率，只需要逐段计算即可.

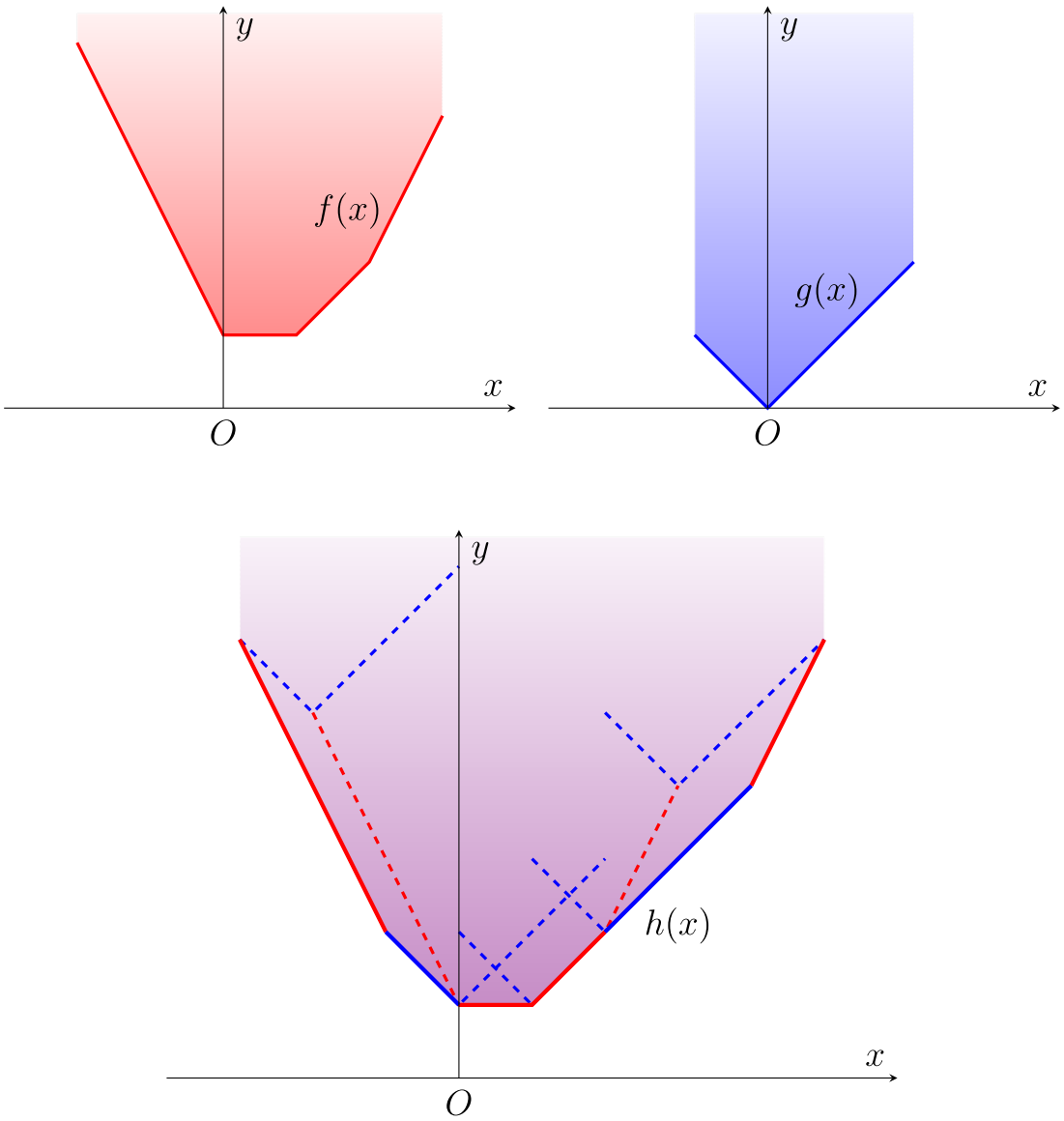
在维护斜率的问题中，往往其中一个函数的形式比较简单，此时可以通过懒标记的方式降低修改复杂度. 在维护拐点的问题中，要计算 $f + g$ 的斜率拐点，只需要将 f 和 g 的斜率拐点合并即可.

卷积下确界（Minkowski 和）

凸函数的另一种常见操作是卷积下确界. 对于函数 f 和 g，函数

$$h(x) = \inf_{y \in \mathbf{R}} f(y) + g(x - y)$$

称为 f 和 g 的 **卷积下确界**[4]（infimal convolution）. 如果 f 和 g 都是凸函数，它们的卷积下确界也是凸函数.



[4] 也常称为 min 卷积、inf 卷积或者 $(\min, +)$ 卷积.

对图示的解释

如图所示，要求 f 和 g 的卷积下确界 h，可以将 f 的图像（第三个图的红色虚线）上的每一个点都视作原点，在相应的坐标系内画出 g 的图像（第三个图中的蓝色虚线）. 当坐标系原点沿着 f 的图像移动时，g 的图像（上境图）移动的轨迹轮廓（即下凸壳），就是 h 的图像. 可以看出，h 的每一个斜率段，都要么是 f 的斜率段，要么是 g 的斜率段：只是重新按照斜率大小排序了. 这个过程中，f 和 g 的角色可以互换，即让 f 的图像沿着 g 的图像移动，得到的结果是一致的.

几何直观上，$\operatorname{epi} h$ 就是 $\operatorname{epi} f$ 和 $\operatorname{epi} g$ 的 Minkowski 和→12.7@p3113. 如果 f 和 g 都是分段线性函数，那么 h 同样是分段线性函数，且它的斜率段可以看作是 f 和 g 的斜率段合并（再排序）的结果.

证明

设 f, g 都是凸函数，h 是它们的卷积下确界. 设 $x_1 < x_2$，且 $\alpha \in (0,1)$. 根据卷积下确界的定义，对任意 $\varepsilon > 0$，存在 $y_i, z_i \in \mathbf{R}$ 使得 $y_i + z_i = x_i$ 且

$$h(x_i) + \varepsilon > f(y_i) + g(z_i).$$

故而，结合 f, g 的凸性及 h 的定义，有

$$\begin{aligned}\alpha h(x_1) + (1-\alpha)h(x_2) + \varepsilon &> \alpha f(y_1) + (1-\alpha) f(y_2) + \alpha g(z_1) + (1-\alpha) g(z_2) \\ &\geq f(\alpha y_1 + (1-\alpha) y_2) + g(\alpha z_1 + (1-\alpha) z_2) \\ &\geq h(\alpha x_1 + (1-\alpha) x_2).\end{aligned}$$

因为 $\varepsilon > 0$ 是任意选取的，所以

$$\alpha h(x_1) + (1-\alpha) h(x_2) \geq h(\alpha x_1 + (1-\alpha) x_2).$$

这就得到 h 的凸性.

然后，对于几何直观，严格地说，只能证明如下结论：

$$\operatorname{epi} f + \operatorname{epi} g \subseteq \operatorname{epi} h \subseteq \operatorname{cl}(\operatorname{epi} f + \operatorname{epi} g).$$

其中，cl 表示闭包.

对于任何 $(x, y) \in \operatorname{epi} f + \operatorname{epi} g$，都存在 $(x_1, y_1) \in \operatorname{epi} f$ 且 $(x_2, y_2) \in \operatorname{epi} g$，使得 $x = x_1 + x_2$ 且

$$y = y_1 + y_2 \geq f(x_1) + g(x_2) \geq h(x_1 + x_2) = h(x).$$

故而，$(x, y) \in \operatorname{epi} h$. 这说明 $\operatorname{epi} f + \operatorname{epi} g \subseteq \operatorname{epi} h$.

反过来，对于任何 $(x, y) \in \operatorname{epi} h$，有 $y \geq h(x)$. 根据 h 的定义，对任何 $\varepsilon > 0$，都存在 $x_1 + x_2 = x$ 使得

$$y + \varepsilon > f(x_1) + g(x_2).$$

令 $y_1 = f(x_1)$ 且 $y_2 = g(x_2)$，就有 $y + \varepsilon > y_1 + y_2$. 这说明，对于任何 $\varepsilon > 0$，都有 $(x_1, y_1) + (x_2, y_2) \in \operatorname{epi} f + \operatorname{epi} g$ 位于点 (x, y) 与点 $(x, y + \varepsilon)$ 的连线上. 取 $\varepsilon \to 0$, 就有 $\operatorname{epi} h \subseteq \operatorname{cl}(\operatorname{epi} f + \operatorname{epi} g)$.

所以，只要 $\operatorname{epi} f + \operatorname{epi} g$ 是闭集，就有 $\operatorname{epi} f + \operatorname{epi} g = \operatorname{epi} h$. 一个使其满足的条件是，$f$ 和 g 都是正常凸函数且 下半连续[1]. 对于算法竞赛的应用来说，这已经足够了，比如分段线性函数总是满足这些条件的.

在实际问题中，如果 f 和 g 其中一个的斜率段数较少，可以直接将较少的斜率段插入到较多的斜率段中；否则，可能需要利用 启发式合并→11.6.7@p2675 或 可并堆→10.7.1@p2025 等方法，降低合并的整体复杂度，或者根据具体问题寻找相应的处理方式.

最值操作

两个凸函数的最大值仍然是凸函数，但是，两个凸函数的最小值未必仍然是凸函数.

很多常见的最小值操作可以转化为卷积下确界：

例子

- $f(x)=\min_{y\in[x+a,x+b]}g(y)$ 仍然是凸函数，因为它可以看作是卷积下确界：
  $f(x)=\min\limits_{y\in\mathbf{R}}g(y)+0_{[-b,-a]}(x-y)$.
- $f(x)=\min\{g(x-a_i)+b_i\}$ 是 $\mathbf{Z}$ 上的凸函数，只要 $g(x)$ 是 $\mathbf{Z}$ 上的凸函数，且在有限集合 $\{a_i\}\subset\mathbf{Z}$ 上定义的函数 $h:a_i\mapsto b_i$ 也是该离散集合上的凸函数．这是因为延拓之后的函数 $\tilde{f}(x)$ 可以看作是卷积下确界：
  $\tilde{f}(x)=\min\limits_{y\in\mathbf{R}}\tilde{h}(y)+\tilde{g}(x-y)$.
  因此，延拓之前的函数 $f(x)$ 也是凸函数.

但并不是所有的最小值操作都保持凸性.

反例

设 $g(x)$ 是凸函数，函数 $f(x)=\min\{g(x-1)+kx,g(x)\}$ 并不一定是凸函数.

在一些特殊的问题中，尽管动态规划的转移方程可以写作两个凸函数的最小值的形式，且难以转化为卷积下确界的形式，但是价值函数依然能够保持凸性．在实际处理时，通常需要结合打表和猜测找到这类问题的合理的斜率转移方式.

了解了凸函数及其常见变换后，就可以通过具体的问题理解 Slope Trick 优化 DP 的方法．本文的例题大致分为维护拐点和维护斜率两组，用于理解这两种维护方式的常见操作和实施细节．但是，正如前文所强调的那样，维护方式并不是 Slope Trick 的本质，应当根据具体的问题需要选取合适的斜率段维护方式.

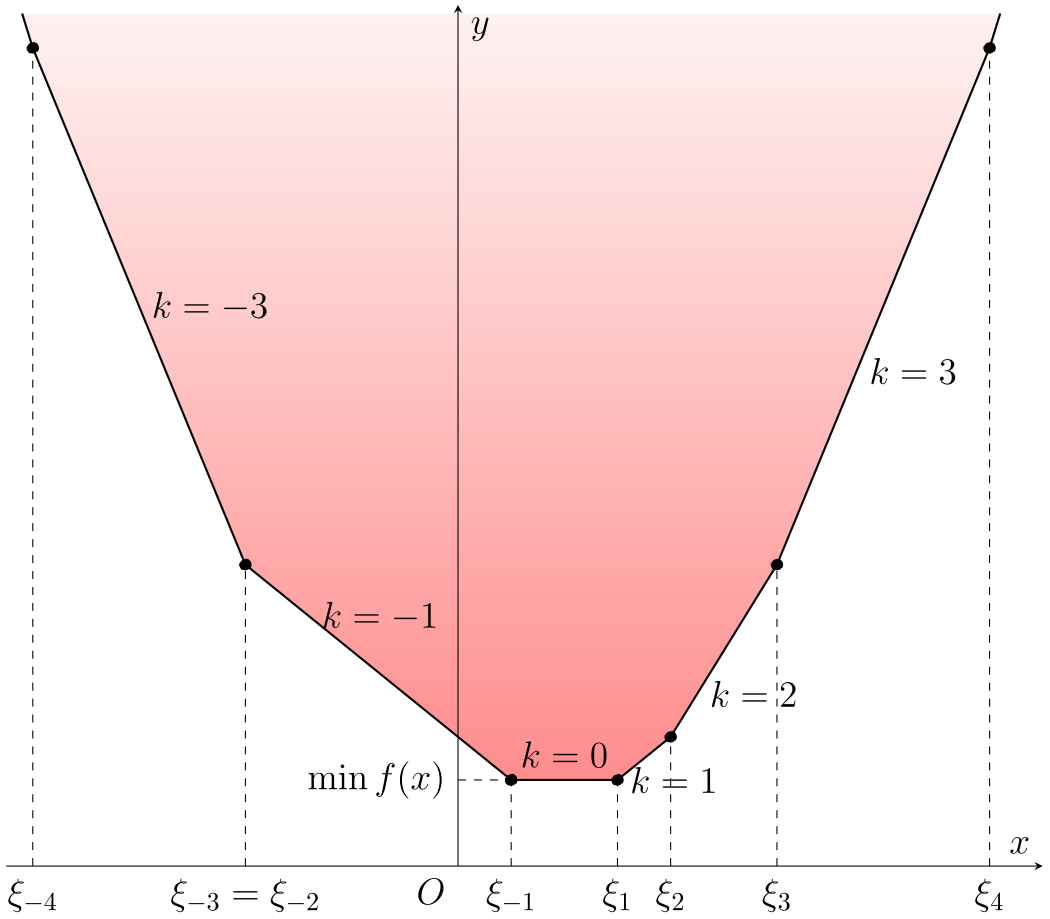
维护拐点

这类问题通常出现在需要最小化若干个绝对值的和式的问题中. 因为这类问题中，价值函数的斜率的绝对值并不大，因此维护斜率变化的拐点更为方便.

维护拐点是指维护分段线性函数中，斜率发生变化的点．相当于对于每个斜率为 k_i 的斜率段 $[l_i,r_i]$，只维护其端点信息，而斜率本身不需要额外维护；因此，这类问题斜率每次发生变化时，都应当只变化一个固定的量．比如，如果维护了拐点集 $\xi_{-s}\le\cdots\le\xi_{-1}\le\xi_1\le\cdots\le\xi_t$，就相当于说：区间 $[\xi_{-1},\xi_1]$ 内斜率为 0；向左每经过一个拐点，斜率减少一；向右每经过一个拐点，斜率增加一；故而，区间 $[\xi_2,\xi_3]$ 内，斜率就是 2，区间 $[\xi_{-3},\xi_{-2}]$ 内，斜率就是 -2，诸如此类．用形式语言表示，函数可以利用斜率拐点写作

$$f(x)=f(\xi_1)+\sum_{i=-s}^{-1}\max\{\xi_i-x,0\}+\sum_{i=1}^{t}\max\{x-\xi_i,0\}.$$

它的最小值就是 $f(\xi_{-1})=f(\xi_1)$，且可以在区间 $[\xi_{-1},\xi_1]$ 内任意位置取到.



例题：最小成本递增序列

[BalticOI 2004] Sequence 数字序列[2]

给定长度为 n 的序列 $\{a_i\}$，求严格递增序列 $\{b_i\}$ 使得 $\sum_i |a_i - b_i|$ 最小，输出最小值和任意一种最优方案 $\{b_i\}$.

解答

首先，$\{b_i\}$ 严格递增，等价于 $\{b'_i\} = \{b_i - i\}$ 弱增．因此，可以对 $\{a'_i\} = \{a_i - i\}$ 求出差值最小的弱增序列 $\{b'_i\} = \{b_i - i\}$ 再恢复成序列 $\{b_i\}$ 即可.

考虑朴素 DP 解法．设 $f_i(x)$ 是已经选取了序列 $\{b'_i\}$ 中前 i 个数字，且第 i 个数字不超过 x 时，已经选取的数字与 $\{a'_i\}$ 的前 i 个数字的最小差值：

$$f_i(x) = \min \sum_{j=1}^{i} |a'_j - b'_j| \text{ s.t. } b'_1 \le b'_2 \le \cdots \le b'_i \le x.$$

容易得到状态转移方程为

$$f_i(x) = \min_{y \le x} f_{i-1}(y) + |a'_i - y|.$$

初始状态为 $f_0(x) \equiv 0$，最后要求的就是 $\min_x f_n(x)$．利用前文提到的凸函数的变换，从 $f_{i-1}(x)$ 到 $f_i(x)$，需要经过两步变换：

1. 首先，加上 $|a'_i - x|$，这相当于对区间 $(-\infty, a'_i]$ 内的所有斜率段都增加 -1，对区间 $[a'_i, +\infty)$ 内的所有斜率段都增加 1；

2. 对得到的函数取最小值，将 $g(x)=f_{i-1}(x)+|\,a'_i-x\,|$ 变为 $f_i(x)=\min_{y\le x} g(y)$. 根据前文分析，这相当于对 $g(x)$ 和 $0_{[0,+\infty)}$ 做卷积下确界. 因为后者的斜率段只有一段，斜率为 0 且向右延伸至无限长，将其插入 $g(x)$ 的斜率段中，相当于删除其中所有正斜率段.

明晰了这些操作后，已经可以直接用平衡树维护所有斜率段了，但代码较复杂. 注意到问题中斜率每次变化至多 1，故而所有斜率的绝对值都不超过 n. 不直接维护斜率段，转而直接维护斜率拐点更为方便.

设 $f_{i-1}(x)$ 的拐点集为 $\xi_{-k}\le\cdots\le\xi_{-1}\le\xi_1\le\cdots\le\xi_\ell$. 那么，上面的两步操作分别对应：

1. 增加一个负斜率段的拐点 a'_i 和一个正斜率段的拐点 a'_i；
2. 弹出所有正斜率段的拐点 $\xi_1,\cdots,\xi_\ell$.

实际维护时，因为每次操作结束后都没有正斜率段的拐点，即斜率拐点具有形式 $\xi_{-k}\le\cdots\le\xi_{-1}$，而且操作总发生在正负斜率段交界处，所以直接维护一个最大堆存储所有拐点即可. 两步操作分别对应：

1. 插入两次 a'_i；
2. 弹出堆顶.

当然，每次结束后都需要维护当前函数的最小值. 因为操作结束后，没有正斜率段，函数最小值就是它在最大堆堆顶处的取值. 设每次操作之前堆顶为 ξ_{-1}，最小值为 $f_{i-1}(\xi_{-1})$. 因为弹出的堆顶是正斜率段的最小拐点，函数的最小值就等于该处函数的取值，所以直接计算弹出前堆顶处函数的取值即可，亦即

$$f_{i-1}(\max\{a'_i,\xi_{-1}\})+|\max\{a'_i,\xi_{-1}\}-a'_i|=f_{i-1}(\xi_{-1})+\max\{0,\xi_{-1}-a'_i\}.$$

其中，第一项相等是因为 $f_{i-1}(x)$ 没有正斜率段. 因此，每次只需要在最小值上不断累加 $\max\{0,\xi_{-1}-a'_i\}$ 即可.

本题还要求输出一种最优方案. 因为最后操作结束时，最优解就是堆顶，所以 b'_n 的取值可以直接确定. 如果已经知道了第 i 个最优解 b'_i，要求解 $f_{i-1}(x)$ 满足 $x\le b'_i$ 的最优解，只需要注意到因为 $f_{i-1}(x)$ 是凸的，所以越接近它的全局最小值点，解就越优，故而只要记录 $f_{i-1}(x)$ 的全局最小值点，并将它与 b'_i 取最小值，就可以得到最优的 b'_{i-1}.

时间复杂度为 $O(n\log n)$.

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

int main() {
  int n;
  std::cin >> n;
  std::vector<int> a(n), b(n);
  for (int& x : a) std::cin >> x;
  for (int i = 0; i < n; ++i) a[i] -= i;
  long long res = 0;
  std::priority_queue<int> max_heap;
  for (int i = 0; i < n; ++i) {
    max_heap.emplace(a[i]);
    max_heap.emplace(a[i]);
    res += max_heap.top() - a[i];
    max_heap.pop();
    b[i] = max_heap.top();
  }
  std::cout << res << '\n';
```

```
  for (int i = n - 2; i >= 0; --i) b[i] = std::min(b[i], b[i + 1]);
  for (int i = 0; i < n; ++i)
    std::cout << (b[i] + i) << (i == n - 1 ? '\n' : ' ');
  return 0;
}

```

模板题：

- Codeforces 713 C. Sonya and Problem Without a Legend[3]
- Luogu P2893 [USACO08FEB] Making the Grade G[4]
- Luogu P4331 [BalticOI 2004] Sequence 数字序列[5]
- Luogu P4597 序列 sequence[6]
- AtCoder 第 2 回 ドワンゴからの挑戦状 予選 E - 花火[7]

例题：转移带限制的情形

[NOISG 2018 Finals] Safety[8]

给定长度为 n 的序列 $\{a_i\}$，求序列 $\{b_i\}$ 使其满足 $|b_i - b_{i-1}| \le h$ 对所有 $1 < i \le n$ 都成立，并使得 $\sum_i |a_i - b_i|$ 最小，输出最小值.

解答

内容大致与上一个题目相仿，只是序列 $\{b_i\}$ 的限制发生了变化. 同样地，设 $f_i(x)$ 为第 i 个数字取 x 时，前 i 个数字的差值的最小值：

$$f_i(x) = \min \sum_{j=1}^{i} |a_j - b_j| \text{ s.t. } |b_{j-1} - b_j| \le h, \forall 1 < j \le i,\ b_i = x.$$

由此，有状态转移方程为

$$f_i(x) = |a_i - x| + \min_{|y-x| \le h} f_{i-1}(y).$$

起始条件为 $f_0(x) \equiv 0$. 最后要求的仍然是 $\min_x f_n(x)$.

状态转移拆解为对凸函数的操作，分两步：

1. 首先对 $f_{i-1}(x)$ 取最值，变为 $\min_{|y-x| \le h} f_{i-1}(y)$，这相当于 $f_{i-1}(x)$ 与 $0_{[-h,h]}(x)$ 的卷积下确界；
2. 再将得到的函数与 $|a_i - x|$ 相加.

同样因为斜率每次只变化一，可以考虑维护拐点. 这样，这两步操作就可以描述为：

1. 将所有负斜率段向左移动 h，将所有正斜率段向右移动 h；
2. 插入两次 a_i.

显然，对于本题，将正负斜率段分别维护较为方便. 因为操作主要集中在零斜率段附近，因此考虑使用 对顶堆→10.7.2@p2026，即分别用最大堆和最小堆维护负斜率段和正斜率段的拐点. 拐点的整体平移操作用懒标记完成. 第二步操作需要分别对两个堆插入一个 a_i；而且，插入完成后，最大堆的堆顶未必仍然小于等于最小堆的堆顶. 此时，交换两堆顶，直到堆顶的大小关系得到满足即可.

最后，考虑操作过程中如何更新最小值. 因为第一步平移操作并不会改变最小值，所以只要考虑交换堆顶的操作即可. 设 $\xi_{-1} > \xi_1$，将堆顶 ξ_{-1} 与 ξ_1 交换时，函数由

$$\max\{0, \xi_{-1} - x\} + \max\{0, x - \xi_1\}$$

变为

$$\max\{0, \xi_1 - x\} + \max\{0, x - \xi_{-1}\}.$$

过程中，函数形状不变，只是向下平移了 $|\xi_{-1} - \xi_1|$. 因此，要使得交换堆顶前后函数保持不变，只需要将最小值累加 $|\xi_{-1} - \xi_1|$ 即可.

算法的时间复杂度仍为 $O(n \log n)$，因为每次添加元素后，交换堆顶的操作至多执行一次.

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

int main() {
  int n;
  long long h;
  std::cin >> n >> h;
  std::priority_queue<long long> max_heap;
  std::priority_queue<long long, std::vector<long long>, std::greater<>>
      min_heap;
  long long lt = 0, rt = 0;
  long long res = 0;
  for (; n; --n) {
    long long x;
    std::cin >> x;
    lt += h;
    rt += h;
    max_heap.emplace(x + lt);
    min_heap.emplace(x - rt);
    auto l = max_heap.top() - lt;
    auto r = min_heap.top() + rt;
    while (l > r) {
      max_heap.pop();
      min_heap.pop();
      res += l - r;
      max_heap.emplace(r + lt);
      min_heap.emplace(l - rt);
      l = max_heap.top() - lt;
      r = min_heap.top() + rt;
    }
  }
  std::cout << res << std::endl;
  return 0;
}

```

模板题：

- Luogu P4272 [CTSC2009] 序列变换[9]
- Luogu P11598 [NOISG 2018 Finals] Safety[10]
- AtCoder Beginner Contest 217 H - Snuketoon[11]
- AtCoder Regular Contest 070 E - NarrowRectangles[12]
- AtCoder Regular Contest 123 D - Inc, Dec - Decomposition[13]

维护斜率

还有一些问题，维护斜率更为方便. 这类问题通常也可以使用 反悔贪心→5.6@p563 或模拟费用流的思想解决. 费用流模型中，最小费用往往是流量的凸函数，这就为使用 Slope Trick 提供了基础.

例题：股票交易问题

Codeforces 865 D. Buy Low Sell High[14]

给定 n 天股票价格序列 $\{p_i\}$（均为正数），初始持股为 0，每天可买入一股、卖出一股或不交易，求 n 天后最大利润.

解答

首先考虑朴素 DP 解法. 设 $f_i(x)$ 为第 i 天结束时持有股票数量为 $x \geq 0$ 的最大利润，则

$$f_i(x) = \max\{f_{i-1}(x-1) - p_i, f_{i-1}(x), f_{i-1}(x+1) + p_i\}.$$

初始状态为 $f_0(0) = 0$，且对所有 $x \neq 0$，有 $f_0(x) = -\infty$. 问题的答案就是 $f_n(0)$.

从 $f_{i-1}(x)$ 到 $f_i(x)$ 需要经过两步变换：

1. 将 $f_{i-1}(x)$ 与函数

$$h_i(x) = \begin{cases} p_i, & x = -1, \\ 0, & x = 0, \\ -p_i, & x = 1 \end{cases}$$

对应的分段线性函数 $\tilde{h}_i(x)$（显然是凹函数）做卷积上确界；

2. 因为这样会导致函数在区间 $[-1, 0)$ 内具有有限值，这与 $x \geq 0$ 的要求矛盾，故而需要截取函数在 $[0, +\infty)$ 内的部分.

将它们转化为斜率段的变化，就是如下两步：

1. 插入长度为 2、斜率为 $-p_i$ 的斜率段；
2. 删除斜率有限的斜率段中，斜率最大且长度为 1 的一段.

因为斜率段的长度总是自然数, 所以不妨维护若干个长度为一的斜率段, 从而只需要记录每段的斜率即可. 因为只需要插入和访问最大值操作，所以只需要一个最大堆. 操作分两步：

1. 插入两次 $-p_i$；
2. 弹出堆顶.

还需要维护 $f_i(0)$ 的值. 因为第一步操作得到的函数在 $x = -1$ 处的取值就是 $f_{i-1}(0) + p_i$，所以它在 $x = 0$ 处的取值就是该值加上马上要弹出的堆顶——它就是函数在区间 $[-1, 0]$ 上的斜率. 因为截断不改变函数在 $x = 0$ 处的取值，所以这就是 $f_i(0)$.

对比该算法实现与上文 最小成本递增序列→7.15@p810 的代码可知, 该算法等价于求将股票价格变为弱递减序列的最小成本.

时间复杂度为 $O(n \log n)$.

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

int main() {
  int n;
  std::cin >> n;
  std::vector<int> a(n);
  for (int& x : a) std::cin >> x;
  long long res = 0;
  std::priority_queue<int> max_heap;
  for (int x : a) {
    max_heap.emplace(-x);
    max_heap.emplace(-x);
```

```
    res += x + max_heap.top();
    max_heap.pop();
  }
  std::cout << res << '\n';
  return 0;
}

```

模板题：

- Codeforces 865 D. Buy Low Sell High[15]

例题：搬运土石问题

[USACO16OPEN] Landscaping P[16]

给定长度为 n 的序列 $\{a_i\}$ 和 $\{b_i\}$，分别表示第 i 个花园已经有的泥土数量和需要的泥土数量（不能多也不能少）. 购买一单位泥土放入任意花园价格为 X，从任意花园运走一单位泥土价格为 Y，从花园 i 向花园 j 运送一单位泥土价格为 $Z \mid i-j \mid$. 求满足所有花园需求的最小成本. （$a_i, b_i \leq 10$）

解答

考虑朴素 DP 解法. 设 $f_i(x)$ 为满足前 i 个花园需求且净剩余 x 单位泥土运到后面的花园时的最小代价. 如果 $x<0$，就相当于净亏空 $\mid x \mid$ 单位泥土需要从后面的花园运送过来. 那么，可以写成状态转移方程为

$$f_i(x)=\min_{y\in\mathbf{R}} f_{i-1}(y)+\mid y \mid Z+h((x-y)+(b_i-a_i)).$$

其中，函数 $h(\delta)$ 表示当前花园的泥土净购买量为 δ 时的成本，即

$$h(\delta)=\max\{0,\delta\}X+\max\{0,-\delta\}Y=\max\{\delta X,-\delta Y\}.$$

它显然是凸函数. 该状态转移方程的含义为

- 之前 $i-1$ 个花园净剩余泥土数量为 y 时，最小成本为 $f_{i-1}(y)$；
- 将净剩余（亏空）的泥土数量在 i 与 $i-1$ 之间运送的成本为 $\mid y \mid Z$；
- 通过买卖，将第 i 个花园的泥土数量从 a_i 调整为 b_i，并将净剩余泥土数量从 y 调整到 x，最小成本为 $h((x-y)+(b_i-a_i))$.

初始状态为 $f_0(0)=0$，且对所有 $x\neq 0$，有 $f_0(x)=+\infty$. 问题的答案就是 $f_n(0)$.

将函数 $f_{i-1}(x)$ 变换为 $f_i(x)$ 可以分为三步：

1. 首先，加上 $\mid x \mid Z$，得到 $f_{i-1}(x)+\mid x \mid Z$；
2. 然后，与 $h(x)$ 做卷积下确界，得到 $\min_{y\in\mathbf{R}} f_{i-1}(y)+\mid y \mid Z+h(x-y)$；
3. 最后，将函数向左平移 (b_i-a_i) 个单位.

转化为对斜率段的操作，同样分为三步：

1. 将原点左侧斜率段全体加上 $-Z$，将原点右侧斜率段全体加上 Z；
2. 将所有小于 $-Y$ 的斜率段全部替换为 $-Y$，将所有大于 X 的斜率段全部替换为 X；
3. 将所有斜率段向左平移 (b_i-a_i) 个单位.

原题中 a_i 和 b_i 很小，因此只需要维护若干个长度为 1 的斜率段即可. 虽然斜率段有无穷多个，但是有上界 X 和下界 $-Y$，且严格位于两者之间的斜率段数目并不多. 因为不涉及插入操作，所以可以用两个栈维护原点两侧的斜率段，区间加和区间最值操作全部打懒标记完成. 上述三步操作分别对应：

1. 对左右两个栈分别打懒标记，左侧加 $-Z$，右侧加 Z；

2. 每次栈内弹出元素时，都对 $-Y$ 取最大值，对 X 取最小值．如果左栈为空，则弹出 $-Y$．如果右栈为空，则弹出 X；
3. 将左栈顶部的 $(b_i - a_i)$ 个元素弹出，插入右栈；当然，$b_i - a_i < 0$ 时，就反过来．

在交换栈顶时，更新答案，向左移动就减去当前斜率，向右移动就加上当前斜率．

算法复杂度为 $O(n \max\{a_i, b_i\})$.

```cpp
#include <iostream>
#include <stack>

int main() {
  int n;
  long long x, y, z;
  std::cin >> n >> x >> y >> z;
  std::stack<long long> neg, pos;
  long long lt = 0, rt = 0;
  long long res = 0;
  for (; n; --n) {
    int a, b;
    std::cin >> a >> b;
    lt -= z;
    rt += z;
    for (; b < a; ++b) {
      auto l = -y;
      if (!neg.empty()) {
        l = std::max(l, neg.top() + lt);
        neg.pop();
      }
      pos.emplace(l - rt);
      res -= l;
    }
    for (; b > a; --b) {
      auto r = x;
      if (!pos.empty()) {
        r = std::min(r, pos.top() + rt);
        pos.pop();
      }
      neg.emplace(r - lt);
      res += r;
    }
  }
  std::cout << res << std::endl;
  return 0;
}

```

模板题：

- Luogu P2748 [USACO16OPEN] Landscaping P[17]
- Kyoto University PC 2016 H - WAAAAAAAAAAAAALL[18]
- JAG Practice Contest 2017 J - Farm Village[19]

习题

本文的最后，提供一些各类算法竞赛中出现过的且可以使用 Slope Trick 解决的问题，以供练习．

- Luogu P3642 [APIO2016] 烟火表演[20]
- Luogu P9962 [THUPC 2024 初赛] 一棵树[21]
- Luogu P11317 [RMI 2021] 路径/Paths[22]
- AtCoder Beginner Contest 383 G - Bar Cover[23]
- Codeforces 280 D. k-Maximum Subsequence Sum[24]
- Codeforces 280 E. Sequence Transformation[25]
- Codeforces 802 O. April Fools' Problem (hard)[26]
- Codeforces 1209 H. Moving Walkways[27]
- Codeforces 1229 F. Mateusz and Escape Room[28]
- Codeforces 1534 G. A New Beginning[29]
- Codeforces 1787 H. Codeforces Scoreboard[30]
- 2019 Summer Petrozavodsk Camp H. Honorable Mention[31]
- 2018 ACM-ICPC World Finals C. Conquer The World[32]
- 300iq Contest 3 F. Farm of Monsters[33]

参考文献与注释

- [Tutorial] Slope Trick - zscoder[34]
- Slope trick explained - Kuroni[35]
- Slope Trick - USACO Guide[36]
- [Tutorial] Intuition on Slope Trick - maomao90[37]

[1] 下半连续

[2] [BalticOI 2004] Sequence 数字序列

[3] Codeforces 713 C. Sonya and Problem Without a Legend

[4] Luogu P2893 [USACO08FEB] Making the Grade G

[5] Luogu P4331 [BalticOI 2004] Sequence 数字序列

[6] Luogu P4597 序列 sequence

[7] AtCoder 第 2 回 ドワンゴからの挑戦状 予選 E - 花火

[8] [NOISG 2018 Finals] Safety

[9] Luogu P4272 [CTSC2009] 序列变换

[10] Luogu P11598 [NOISG 2018 Finals] Safety

[11] AtCoder Beginner Contest 217 H - Snuketoon

[12] AtCoder Regular Contest 070 E - NarrowRectangles

[13] AtCoder Regular Contest 123 D - Inc, Dec - Decomposition

[14] Codeforces 865 D. Buy Low Sell High

[15] Codeforces 865 D. Buy Low Sell High

[16] [USACO16OPEN] Landscaping P

[17] Luogu P2748 [USACO16OPEN] Landscaping P

[18] Kyoto University PC 2016 H - WAAAAAAAAAAAAALL

[19] JAG Practice Contest 2017 J - Farm Village

[20] Luogu P3642 [APIO2016] 烟火表演

[21] Luogu P9962 [THUPC 2024 初赛] 一棵树

[22] Luogu P11317 [RMI 2021] 路径/Paths

[23] AtCoder Beginner Contest 383 G - Bar Cover

[24] Codeforces 280 D. k-Maximum Subsequence Sum

[25] Codeforces 280 E. Sequence Transformation

[26] Codeforces 802 O. April Fools' Problem (hard)

[27] Codeforces 1209 H. Moving Walkways

[28] Codeforces 1229 F. Mateusz and Escape Room

[29] Codeforces 1534 G. A New Beginning

[30] Codeforces 1787 H. Codeforces Scoreboard

[31] 2019 Summer Petrozavodsk Camp H. Honorable Mention

[32] 2018 ACM-ICPC World Finals C. Conquer The World

[33] 300iq Contest 3 F. Farm of Monsters

[34] [Tutorial] Slope Trick - zscoder

[35] Slope trick explained - Kuroni

[36] Slope Trick - USACO Guide

[37] [Tutorial] Intuition on Slope Trick - maomao90 

7.15.6 WQS 二分

引入

本文介绍利用 WQS 二分优化动态规划问题的方法．在不同的文章中，它也常称作带权二分、凸优化 DP、凸完全单调性 DP、Lagrange 乘子法等，在国外也称作 Aliens Trick．它最早由王钦石在《浅析一类二分方法》一文中总结．

WQS 二分通常用于解决这样一类优化问题：它们带有数量限制，直接求解代价较高；但一旦去除这一限制，问题本身就变得容易得多．

比如，假设要解决的问题是，要从 n 个物品中选取 m 个，并最优化某个较复杂的目标函数．如果设从前 i 个物品中选取 j 个，目标函数的最优值为 $f(i,j)$，那么原问题的答案就是 $f(n,m)$．这类问题中，状态转移方程通常是二维的．直接实现该状态转移方程，时间复杂度是 $O(nm)$ 的，难以接受．

进一步假设，没有数量限制的最优化问题容易解决．但是，选取到的最优数量未必满足原问题的数量限制．假设选取的物品过多．那么，就可以考虑在选取物品时，为每个选取到的物品都附加一个固定大小的惩罚 k（即「带权二分」中的「权」），仍然解没有数量限制的最优化问题．根据 k 的取值不同，选取到的最优数量也会有所不同；而且，随着 k 的变化，选取到的最优数量也是单调变化的．所以，可以通过二分，找到 k 使得选取到的最优数量恰为 m．假设此时目标函数的最优值为 $f_k(n)$，那么，只要消除额外附加的惩罚造成的价值损失，就能得到原问题的答案 $f(n,m)=f_k(n)-km$．假设单次求解附加惩罚的问题的复杂度是 $O(T(n))$ 的，那么，算法的整体复杂度也就降低到了 $O(T(n)\log L)$，其中，$O(\log L)$ 是二分 k 需要的次数．

这就是 WQS 二分的基本想法．但是，这一想法能够行得通，前提是 $f(n,m)$ 关于 m 是凸的．否则，可能不存在使得最优数量恰为 m 的附加惩罚 k．这也是这种 DP 优化方法常常称为「凸优化 DP」或「凸完全单调性 DP」的原因．

传统方法

设非空集合 X 为（有限的）决策空间，$f:X\to\mathbf{R}$ 为目标函数，且另有函数 $g:X\to\mathbf{R}^d$ 用于施加限制．需要求解的问题，可以看作是计算如下最优化问题的价值函数 $v(y)$ 在某处的取值：

$$v(y)=\min_{x\in X} f(x)$$

$$\text{subject to } g(x)=y.$$

比如，对于前文提到的限制数量的问题，X 可以理解为所有物品集合的子集族，$x\in X$ 是单个子集，$f(x)$ 是单个子集的价值函数，$g(x)$ 是子集 x 中的元素个数．当然，$g(x)$ 并非只能是数量限制，后文提供了更为广泛的限制条件的例子．

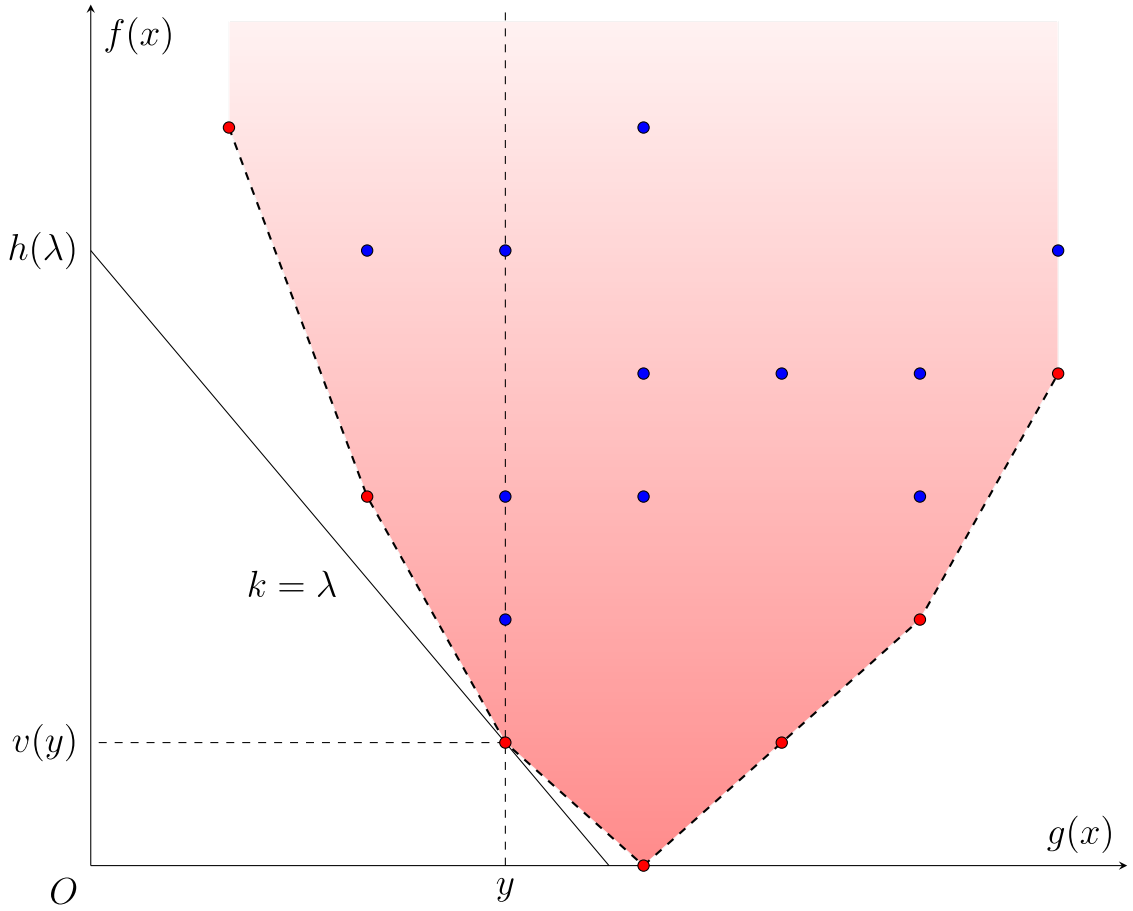
> **约定**
>
> 为了行文方便，本文仅讨论最小化目标函数的问题．最大化目标函数的问题与之相仿，只是需要将本文中的（下）凸函数相应地替换成凹函数（或称上凸函数）．或者，可以通过添加负号，将最大化目标函数的问题，转化为最小化它的相反数的问题．

几何直观

因为算法竞赛中遇到的大多数问题都是组合优化问题，决策空间 X 通常没有良好的结构，所以，可以转而考察集合

$$\mathcal{D}=\{(g(x),f(x))\in\mathbf{R}^d\times\mathbf{R}:x\in X\}.$$

传统方法能够解决的主要是 $d = 1$ 的情形，即只有一个限制的情形．下图提供了此时点集 $\mathcal{D}$ 的一种可能的图示．



图中的红点和蓝点是 X 中所有可能的选择投影在平面 $(g(x), f(x))$ 上得到的集合 $\mathcal{D}$．于是，原问题所要求的就是横坐标为 y 的那些点中，纵坐标的最小值 $v(y)$．当 y 变动时，所有这样的点 $(y, v(y))$ 就构成了图中的红点的集合．

为了求得点 $(y, v(y))$ 的纵坐标，可以考虑用斜率为 $\lambda \in \mathbf{R}$ 的直线去切集合 $\mathcal{D}$．如图所示，当直线的斜率选取得恰当时，经过点 $(y, v(y))$ 的那条直线，是所有经过集合 $\mathcal{D}$ 中的点且斜率为 λ 的直线中，截距 $f(x) - \lambda g(x)$ 最小的．将这一最小值记为

$$h(\lambda) = \min_{x \in X} f(x) - \lambda g(x).$$

那么，因为 $(y, v(y))$ 同样位于该直线上，就可以得到原问题的解

$$v(y) = h(\lambda) + \lambda y.$$

假设对于所有合理范围的 λ，上述函数 $h(\lambda)$ 都是容易求解的．这在算法竞赛中常常是成立的，因为它去掉了原问题中的限制条件．那么，现在面临的最为重要的两个问题，就是

1. 是否存在这样的直线斜率 λ，使得它的截距最小值恰好取得在点 $(y, v(y))$ 处，以及
2. 如果存在，如何找到这样的斜率 λ．

第一个问题相对容易解决．因为当直线斜率 λ 发生变化时，所有这些直线切出的集合（即它们对应的上半平面的交）必然是一个凸集．因此，这些直线能够经过某个点，当且仅当这个点在该凸集的下凸壳上．这等价于说，函数 $v(y)$ 是 凸函数→7.15.5@p836．

第二个问题则更为精细．因为所求点的横坐标已经知道是 y，所以，一个自然的思路是，计算 $h(\lambda)$ 时，顺便求出限制函数 $g(x)$ 在当前最优解 x_λ 处的取值．比如，在前文提到的例子中，求解带惩罚的问题时，

可以记录带惩罚的目标函数取得最优解时，选取的物品数量．然后，将 $g(x_\lambda)$ 与所期望的 y 进行比较，并相应调整下次计算时的 λ 的取值．这就是最为传统的 WQS 二分的方法．

总结一下，传统 WQS 二分的基本流程如下：

1. 初始时，选取一个 λ 的合理的区间；
2. 在当前的区间中选择一个 λ；
3. 求解带惩罚的问题 $h(\lambda)=\min_{x\in X} f(x)-\lambda g(x)$，并记录它的最优解 x_λ 处 $g(x)$ 的取值 $g(x_\lambda)$；
4. 如果 $g(x_\lambda)=y$，就得到原问题的最优价值 $v(y)=h(\lambda)+\lambda y$，直接结束算法；
5. 否则，根据 $g(x_\lambda)$ 与 y 的大小关系，调整 λ 的区间，并回到步骤 2.

这一基本流程已经足以解决一些问题，但并不完善．接下来，本文将讨论对这一基本流程的改进．

共线情形的处理

在应用基本流程时，首先遇到的问题就是共线情形无法正确处理．

如果点集 $\mathcal{D}$ 的下凸壳上有三个及以上的红点共线，那么在上述基本流程中，可能无法正确地判断 $g(x_\lambda)$ 与 y 的大小关系．比如，设共线的三个红点的横坐标分别为 y_1,y_2,y_3，且它们共线的直线的斜率为 λ^*. 那么, 要正确求解 $v(y_2)$, 就必须保证算法终止时, 最后计算的问题是 $h(\lambda^*)$, 因为 λ^* 是唯一一个最小化截距时能够经过点 $(y_2,v(y_2))$ 的直线的斜率. 但是, 在求解 $h(\lambda^*)$ 的过程中, 记录的 $g(x_{\lambda^*})$ 可能是 y_1,y_2,y_3 中的任意一个．如果记录到的 $g(x_{\lambda^*})$ 不等于 y_2，那么算法将错误地继续运行，并向着背离 y_2 的方向调整 λ 的区间，最终将得到错误的结果．

为了解决共线的情形，一种处理方法是在记录最优解 x_λ 对应的 $g(x_\lambda)$ 时，总是使之尽可能大（或尽可能小）．同时，将二分中的终止条件从寻找恰好满足 $g(x_\lambda)=y$ 的 λ 改为寻找满足 $g(x_\lambda)\geq y$（或 $g(x_\lambda)\leq y$）的最小（或最大）的 λ．在上一段的例子中，这相当于计算问题 $h(\lambda^*)$ 时，输出的 $g(x_{\lambda^*})$ 是 y_3．这就保证了算法终止时，最后计算的问题是 $h(\lambda^*)$．实现这一方法时，需要注意最后输出的不是 $h(\lambda)+\lambda g(x_\lambda)$ 而是 $h(\lambda)+\lambda y$，因为记录的 $g(x_\lambda)$ 未必等于实际的限制 y.

另一种处理方法是实数二分. 如果问题涉及的数字都是整数, 显然, WQS 二分中的斜率也是整数. 在二分中引入实数，是为了保证错误地排除正确选项 λ^* 时，可以通过小数部分调整回来，最终逼近正确答案 λ^*．例如，在上面的例子中，如果计算问题 $h(\lambda^*)$ 时，记录的 $g(x_{\lambda^*})$ 是 y_1，小于所希望的 y_2，那么，算法就会转而考虑区间 $(\lambda^*,\lambda_r]$，其中，λ_r 是 λ 所在区间的右端点．对于整数的情形，这一区间实际应该写作 $[\lambda^*+1,\lambda_r]$，这就排除了在后续算法中接近正确答案 λ^* 的可能．但是，实数二分时，考虑的区间仍然是 $(\lambda^*,\lambda_r]$，而且，对于该区间中的 λ，求解 $h(\lambda)$ 时记录的 $g(x_\lambda)$ 总是不小于 y_3，从而严格大于 y_2 的. 因此，随着算法继续进行，会不断地舍去右半区间，从而，最终得到的 λ 的范围可以保证在 λ^* 附近．当然, 因为已经知道所求的斜率是一个整数, 实数二分终止时的精度不必太高, 只要能保证二分的区间中只包含一个整数即可，这一整数就是要寻找的 λ^*.

正确地处理共线情形后，WQS 二分足以解决绝大多数算法竞赛会遇到的 WQS 二分的问题．但是，这一方法仍然存在一些不足之处：它无法处理 $g(x_\lambda)$ 难以记录的情形，也无法处理高维 WQS 二分中多个点共面的情形．本文将进一步考察最优化问题 $v(y)$ 的性质，并提出更为一般的处理方法．

对偶方法

本节介绍一种 WQS 二分的实现方法，它只要求对于所有 $\lambda\in\mathbf{R}^d$，可以高效地计算

$$h(\lambda)=\min_{x\in X} f(x)-\lambda\cdot g(x)$$

[5]实际问题中，y 可能只能取到 $\mathbf{R}^d$ 中的有限多个格点．此处实际需要的条件是，原问题的解 $v(y)$ 可以延拓为 $\mathbf{R}^d$ 上的凸函数 $\tilde{v}:\mathbf{R}^d\to\mathbf{R}\cup\{\pm\infty\}$，也就是说 $v(y)$ 是 **可凸延拓的**（convex-extensible）．为行文方便，正文中仍然用 $v(y)$ 表示延拓后的函数．几何直观上，这相当于说点集 $\{(y,v(y))\}$ 全部都位于它们的凸包的下凸壳上．对于一维的情形，这一条件利用代数语言 很容易刻画→7.15.5@p836；但是，对于高维的情形，这稍微有些复杂，这份讲义[1] 中提供了一些简单的充分条件.

的取值，且原问题的最优价值 $v(y)$ 是关于 $y \in \mathbf{R}^d$ 的凸函数[5]．用一句话概括，本节将证明原问题的价值函数 $v(y)$ 就等于它的对偶问题的最优价值

$$v^\star(y) = \sup_{\lambda \in \mathbf{R}^d} h(\lambda) + \lambda \cdot y,$$

而对偶问题的目标函数是关于 $\lambda \in \mathbf{R}^d$ 的凹函数，从而是单峰函数，可以通过 三分法→5.9@p629 或 黄金分割法→5.9@p629 高效地求解，复杂度仍然是 $O(T(n)\log^d L)$ 的．这就完全地解决了传统 WQS 二分方法中记录 $g(x_\lambda)$ 的值可能会出现的问题，同时，允许将 WQS 二分的思想应用至高维的情形．

另外，本节还说明，$g(x_\lambda)$ 的范围可以通过 $h(\lambda)$ 求得，而无需在求解 $h(\lambda)$ 时额外记录．例如，对于 $d = 1$ 且问题只涉及整数的情形，可以证明 $g(x_\lambda)$ 的取值范围恰为

$$[h(\lambda - 1) - h(\lambda), h(\lambda) - h(\lambda + 1)].$$

这实际上也对于不得不采取前文所述二分流程的题目，提供了又一种解决共线问题的方法．

接下来，本节将用凸分析的理论证明这些结论成立．至于这些方法的具体应用，可以参考 例题→7.15@p810 一节．

Lagrange 对偶

考虑用 Lagrange 乘子法[2] 解决该问题．引入 Lagrange 乘子 $\lambda \in \mathbf{R}^d$，那么，Lagrangian 可以写作

$$L(x, \lambda, y) = f(x) - \lambda \cdot g(x) + \lambda \cdot y.$$

因为只要 $g(x) - y$ 有一个分量非零，就可以让相应的 λ 的分量趋于（正或负）无穷，所以有

$$\sup_{\lambda \in \mathbf{R}^d} L(x, \lambda, y) = \begin{cases} f(x), g(x) = y, \\ +\infty, \text{otherwise.} \end{cases}$$

这说明，原问题可以写作

$$v(y) = \min_{x \in X} \sup_{\lambda \in \mathbf{R}^d} L(x, \lambda, y).$$

交换两次最值操作，就得到它的 对偶问题[3]：

$$\begin{aligned} v^\star(y) &= \sup_{\lambda \in \mathbf{R}^d} \min_{x \in X} L(x, \lambda, y) \\ &= \sup_{\lambda \in \mathbf{R}^d} h(\lambda) + \lambda \cdot y. \end{aligned}$$

马上要说明的是，在 $v(y)$ 是关于 y 的凸函数的条件下，强对偶（strong duality）成立，即 $v^\star(y) = v(y)$.

凸共轭

为了说明强对偶成立，需要引入凸共轭的概念．

自 凸共轭

对于函数 $f : \mathbf{R}^d \to \mathbf{R} \cup \{\pm\infty\}$，它的 **凸共轭**（convex conjugate），或称 **Legendre–Fenchel 变换**（Legendre–Fenchel transformation），是指函数

$$f^*(x^*) = \sup_{x \in \mathbf{R}^d} x^* \cdot x - f(x).$$

从变量 x^* 的角度看，$f^*(x^*)$ 是一系列线性函数的上确界，所以，必然是 $\mathbf{R}^d$ 上的凸函数．

超平面的「斜率向量」和「截距」

本文所讨论的向量空间 $\mathbf{R}^{d+1}$ 中的超平面的方程都具有形式

$$y = k \cdot x + b.$$

也就是说，本文不会涉及平行于 y 轴的超平面．为表述方便，本文并不严谨地将 k 称为该超平面的「斜率向量」，b 称为该超平面的「截距」．将这一超平面的方程写成更标准的形式，就是

$$k \cdot x - y = -b.$$

它的一个法向量是 $(k,-1)$. 因此，所谓的斜率向量其实是将超平面的法向量归一化使得它的最后一个分量等于 -1 时，所得到的法向量的前 d 个分量.

几何直观上，函数 $f(x)$ 的凸共轭描述的是，对于所有斜率向量为 x^* 且与函数 $f(x)$ 的上境图

$$\operatorname{epi} f=\{(x,y)\in\mathbf{R}^d\times\mathbf{R}:f(x)\le y\}$$

相交的超平面，截距 $f(x)-x^*\cdot x$ 的最小值就是 $-f^*(x^*)$. 换句话说，函数 $f(x)$ 总在超平面 $y=x^*\cdot x-f^*(x^*)$ 上方，且与该平面切于点 $(x_0,f(x_0))$；当然，可能存在其余的切点. 这样的超平面，称为 $f(x)$ 在 x_0 处的 **支撑超平面**（supporting hyperplane）. 函数 $f(x)$ 的一个支撑超平面的截距由它的斜率向量唯一确定，凸共轭就提供了这个从斜率向量到截距的映射.

在集合 X 上最小化 $f(x)-\lambda\cdot g(x)$ 就等价于在集合 $\{(y,v(y))\}$ 上最小化 $v(y)-\lambda\cdot y$:

$$\begin{aligned}\min_x f(x)-\lambda\cdot g(x)&=\min_{y\in g(X)}\left(\min_{x\in X:g(x)=y}f(x)-\lambda\cdot g(x)\right)\\&=\min_{y\in g(X)}\left(\min_{x\in X:g(x)=y}f(x)\right)-\lambda\cdot y\\&=\min_{y\in g(X)}v(y)-\lambda\cdot y.\end{aligned}$$

因此，有

$$h(\lambda)=\min_{y\in g(X)}v(y)-\lambda\cdot y=-v^*(\lambda).$$

这说明 $h(\lambda)$ 是关于 $\lambda\in\mathbf{R}^d$ 的凹函数. 进而，有

$$v^\star(y)=\sup_{\lambda\in\mathbf{R}^d}\lambda\cdot y-v^*(\lambda)=v^{**}(y).$$

也就是说，对偶问题的价值函数 $v^\star(y)$ 是原问题的价值函数 $v(y)$ 的双重凸共轭，也称为 **双共轭**（biconjugate）.

所以，问题转化为：什么样的函数 $v(y)$ 满足它的双共轭就等于它自身？这一问题的答案由如下定理给出：

定理（Fenchel–Moreau）

对于函数 $f:\mathbf{R}^d\to\mathbf{R}\cup\{\pm\infty\}$，它的双共轭等于它自身，即 $f^{**}=f$，当且仅当以下三个条件之一满足：

1. $f(x)$ 是正常凸函数且 下半连续[4]，
2. $f(x)\equiv+\infty$，或
3. $f(x)\equiv-\infty$.

证明

一个函数是正常的（proper），当且仅当它从不取得 $-\infty$ 的值，且不永远取得 $+\infty$ 的值.

对于非正常函数的情形，可以验证 $f(x)\equiv+\infty$ 和 $f(x)\equiv-\infty$ 互为共轭. 除此之外，只要 $f(x)$ 在任何一点处取到 $-\infty$，必然有 $f^*(x^*)\equiv+\infty$. 所以，满足 $f^{**}=f$ 的非正常函数只有这两种情形. 下面的讨论仅限于正常函数. 对于正常函数，下半连续且凸的条件等价于它的上境图是闭凸集.

这一条件的必要性是容易的. 因为 $f=f^{**}$ 是 f^* 的凸共轭，作为一系列线性函数的上确界，它的上境图必然是一系列闭凸集的交集，所以必然是闭凸集. 这就说明，满足 $f^{**}=f$ 的正常函数必然是下半连续且凸的.

反过来，这些条件也是充分的. 和其他强对偶定理的证明一样，证明可以分为两步.

第一步，说明弱对偶成立，即 $f(x) \geq f^{**}(x)$. 由凸共轭的定义可知，对于所有 $x, x^* \in \mathbf{R}^d$，都有

$$f^*(x^*) \geq x^* \cdot x - f(x).$$

这就说明，对于所有 $x, x^* \in \mathbf{R}^d$，同样有

$$f(x) \geq x^* \cdot x - f^*(x^*).$$

对不等式右侧中的 x^* 取上确界，就有 $f(x) \geq f^{**}(x)$.

第二步，利用 超平面分离定理[5] 说明 $f(x) \leq f^{**}(x)$. 假设不然，存在 $x_0 \in \mathbf{R}^d$ 使得 $f(x_0) > f^{**}(x_0)$ 成立. 因为 $f(x)$ 的上境图 $\operatorname{epi}(f)$ 是闭凸集，而且单点集 $\{(x_0, f^{**}(x_0))\}$ 是紧凸集，所以，根据超平面分离定理，存在 $(\lambda, t) \in \mathbf{R}^d \times \mathbf{R}$ 和 $\alpha \in \mathbf{R}$ 使得对于所有 $x \in \operatorname{dom} f := \{x \in \mathbf{R}^d : f(x) < +\infty\}$ 和所有 $y \geq f(x)$ 都有

$$\lambda \cdot x - ty < \alpha < \lambda \cdot x_0 - tf^{**}(x_0)$$

成立. 因为 y 可以选得任意大，所以必然有 $t \geq 0$. 这又可以分为两种情形.

首先，讨论 $t > 0$ 的情形. 此时，将不等式的各部分都同除以 t，并设 $\lambda' = t^{-1}\lambda$ 和 $\alpha' = t^{-1}\alpha$，就得到

$$\lambda' \cdot x - y < \alpha' < \lambda' \cdot x_0 - f^{**}(x_0).$$

对所有 $x \in \operatorname{dom} f$，令 $y = f(x)$，就都成立

$$\alpha' > \lambda' \cdot x - f(x).$$

故而，对不等号右侧的 x 取上确界，有

$$\alpha' \geq \sup_{x \in \mathbf{R}^d} \lambda' \cdot x - f(x) = f^*(\lambda').$$

进而，有

$$f^{**}(x_0) < \lambda' \cdot x_0 - f^*(\lambda') \leq \sup_{x^* \in \mathbf{R}^d} x^* \cdot x_0 - f^*(x^*) = f^{**}(x_0).$$

这一矛盾说明 $t > 0$ 的情形并不成立.

最后，讨论 $t = 0$ 的情形. 事实上，将要说明的是，可以通过微扰，将它转化为 $t > 0$ 的情形. 任取 $\lambda_0 \in \operatorname{dom} f^*$，根据凸共轭的定义可知，对于任何 $x \in \operatorname{dom} f$ 和 $y \geq f(x)$ 都有

$$\lambda_0 \cdot x - y \leq f^*(\lambda_0).$$

因此，对于任意 $\varepsilon > 0$，都有

$$(\lambda + \varepsilon\lambda_0) \cdot x - \varepsilon y < \alpha + \varepsilon f^*(\lambda_0).$$

同时，因为 $\alpha < \lambda \cdot x_0$，所以，对于充分小的 $\varepsilon > 0$，又有

$$\alpha + \varepsilon f^*(\lambda_0) < (\lambda + \varepsilon\lambda_0) \cdot x_0 - \varepsilon f^{**}(x_0).$$

因此，如果取 $\lambda' = \lambda + \varepsilon\lambda_0$，$t' = \varepsilon$ 和 $\alpha' = \alpha + \varepsilon f^*(\lambda_0)$，那么，就有

$$\lambda' \cdot x - t'y < \alpha' < \lambda' \cdot x_0 - t' f^{**}(x_0).$$

这就又回到了前一种情形，仍然会导致矛盾.

这一矛盾说明，并不存在满足 $f(x_0) > f^{**}(x_0)$ 的点 $x_0 \in \mathbf{R}^d$. 故而，总有 $f(x_0) \leq f^{**}(x_0)$.

结合这两步证明的结果，就得到 $f^{**}(x) = f(x)$ 成立.

因此，强对偶成立，当且仅当 $v(y)$ 是关于 $y \in \mathbf{R}^d$ 的凸函数[6].

[6]定理中提供的条件看似比凸函数更强一些，但是，对于算法竞赛能够遇到的情形，特别是 X 为有限集合时，仅强调凸函数就已经足够. 由离散集合上的正常凸函数 v 延拓而来的函数 $\tilde{v}$ 必然是下半连续的凸函数，因为有限多个点的凸包必然是闭凸集，而所谓下半连续的凸函数，就等价于它的上境图是闭凸集. 至于正常凸函数中的「正常」一词，只要 $v(y)$ 在至少一个点处取得有限值且是凸函数，就可以保证.

次梯度

上一节说明了，带惩罚的问题的价值函数 $h(\lambda)$ 是原问题的价值函数 $v(y)$ 的凸共轭的相反数．因为凸共轭的定义实际上是一个含参数的最优化问题，所以它也成立类似 包络定理[6] 的结论．但是，因为凸函数并非处处可微的，所以需要首先将导数的定义推广到凸函数的情形．这就引出了次梯度的概念．

自 次梯度

对于凸函数 $f:\mathbf{R}^d\to\mathbf{R}\cup\{\pm\infty\}$ 和 $x_0\in\operatorname{dom} f$，如果向量 $x^*\in\mathbf{R}^d$ 满足对于任何 $x\in\mathbf{R}^d$，都有

$$f(x)\geq f(x_0)+x^*\cdot(x-x_0),$$

那么，就称 x^* 是 $f(x)$ 在 x_0 处的一个 **次梯度**（subgradient）．函数 $f(x)$ 在 x_0 处的全体次梯度的集合称为它在该处的 **次微分**（subdifferential），记作 $\partial f(x_0)$．

几何直观上，凸函数 $f(x)$ 在 x_0 处的次微分，就是它在该处的所有支撑超平面的斜率向量的集合．对于一维的情形，次微分

$$\partial f(x_0)=[\partial_- f(x_0),\partial_+ f(x_0)],$$

其中，$\partial_- f(x_0)$ 和 $\partial_+ f(x_0)$ 分别是函数 $f(x)$ 在 x_0 处的左右导数．进一步地，对于整数集上的凸函数 $f:\mathbf{Z}\to\mathbf{R}\cup\{\pm\infty\}$ 延拓而来的 $\tilde f(x)$，它在整数点 $x=k$ 处的左右导数就是左右两侧的一阶差分：

$$\partial\tilde f(k)=[f(k)-f(k-1),f(k+1)-f(k)].$$

显然，凸函数 $f(x)$ 在点 x_0 处可微，当且仅当它在该处的次微分 $\partial f(x_0)$ 是单点集．

因为凸共轭提供了从支撑超平面的斜率向量到它的截距的映射，所以，利用凸共轭，可以判断一个斜率向量 x^* 是否是凸函数 $f(x)$ 在给定点 x 处的一个次梯度．

定理（凸共轭与次梯度）

对于正常凸函数 $f:\mathbf{R}^d\to\mathbf{R}\cup\{\pm\infty\}$ 和任意 $x,x^*\in\mathbf{R}^d$，都有

$$x^*\in\partial f(x)\iff x^*\cdot x=f(x)+f^*(x^*).$$

进而，如果 f 还是下半连续的，那么这两个条件都等价于 $x\in\partial f^*(x^*)$．

证明

按照次梯度的定义，$x^*\in\partial f(x)$，当且仅当

$$f(x')\geq f(x)+x^*\cdot(x'-x),\ \forall x'\in\mathbf{R}^d.$$

这等价于

$$x^*\cdot x-f(x)\geq x^*\cdot x'-f(x'),\ \forall x'\in\mathbf{R}^d.$$

这又等价于

$$x^*\cdot x-f(x)\geq\sup_{x'\in\mathbf{R}^d}x^*\cdot x'-f(x')=f^*(x^*).$$

但是，依据凸共轭的定义，总是有

$$x^*\cdot x-f(x)\leq f^*(x^*).$$

因此，前一式中的大于等于号实际上等价于等号，也就等价于下式

$$x^*\cdot x=f(x)+f^*(x^*).$$

这就完成了第一部分的证明．

对于 f 是下半连续的正常凸函数的情形，依 Fenchel–Moreau 定理，有 $f^{**}=f$．因此，这两个条件等价于

$$x^*\cdot x=f^*(x^*)+f^{**}(x).$$

再次应用第一部分的结论，它们也就等价于 $x \in \partial f^*(x^*)$.

这一结论说明，如果 $f^{**} = f$，那么凸共轭 f^* 在 x^* 处的次微分 $\partial f^*(x^*)$，恰好就是斜率向量为 x^* 的支撑超平面与上境图 $\operatorname{epi} f$ 的交点的 x 分量的集合.

推论

对于下半连续的正常凸函数 $f : \mathbf{R}^d \to \mathbf{R} \cup \{\pm\infty\}$ 和任意 $x, x^* \in \mathbf{R}^d$，都有

$$\partial f(x) = \arg\max_{y^* \in \mathbf{R}^d} x \cdot y^* - f^*(y^*),$$

$$\partial f^*(x^*) = \arg\max_{y \in \mathbf{R}^d} x^* \cdot y - f(y).$$

证明

下面，证明第二个等式. 第一个等式的证明与之类似.

按照凸共轭的定义，有

$$f^*(x^*) = \sup_{y \in \mathbf{R}^d} x^* \cdot y - f(y),$$

所以，

$$x \in \arg\max_{y \in \mathbf{R}^d} x^* \cdot y - f(y)$$

当且仅当 $f^*(x^*) = x^* \cdot x - f(x)$，而这一等式成立，又当且仅当 $x \in \partial f^*(x^*)$. 这就证明了两个集合是相等的.

应用到本文的场景中，这一结论说明，求解问题

$$h(\lambda) = \min_{x \in X} f(x) - \lambda \cdot g(x) = \min_{y \in g(X)} v(y) - \lambda \cdot y$$

时，限制函数 $g(x)$ 在最优决策集合上的取值恰为 $\partial(-h(\lambda))$. 对于 $d = 1$ 且问题只涉及整数的情形，这一集合就是区间

$$[h(\lambda - 1) - h(\lambda), h(\lambda) - h(\lambda + 1)].$$

对于连续的整数 λ，这些区间首尾相接，所以，如果用于二分，只需要计算一侧的端点即可.

凸性证明

应用 WQS 二分的前提条件是价值函数的凸性. 在算法竞赛中，可以通过打表、感性理解等方式猜测凸性成立. 但是，严格地证明凸性成立，往往并不容易. 本节结合如下经典题目，介绍算法竞赛中常见的证明凸性的方法.

种树问题

有 n 个树坑，要种 m 棵树. 树不能栽种于相邻的两个坑. 给定长度为 n 的序列 $\{a_i\}$，表示在每个坑种树的收益，收益可正可负. 求种完这 m 棵树最大可能的收益和.

简言之，就是在长度为 n 的链上，求解大小为 m 的最大权独立集的问题.

这些方法粗略地可以分为四类：

- 归约为凸优化问题（包括 线性规划→9.15.1@p1774 等）的价值函数对参数的凸性，这包括建立 费用流→11.23.4@p2935 模型等方法；
- 利用状态转移方程也可以归纳地证明凸性成立，过程中可能会用到一些 保持凸性的变换→7.15.5@p836；
- 对于区间分拆类型的问题，可以验证每段区间的成本函数满足 四边形不等式→7.15.4@p821；

- 最后，对于特殊的问题，也可以通过交换论证直接说明凸性成立.

这些证明方法本身往往都同该问题的某种解法联系在一起.

归约为含参凸优化

考虑如下形式的含参凸优化问题：

$$v(y) = \inf_{x \in \mathcal{D}(y)} f(x, y).$$

其中，目标函数 $f : \mathbf{R}^m \times \mathbf{R}^d \to \mathbf{R} \cup \{\pm\infty\}$ 对于每个 $y \in \mathbf{R}^d$ 都是关于 $x \in \mathbf{R}^m$ 的凸函数，而可行域 $\mathcal{D} : \mathbf{R}^d \to \mathcal{P}(\mathbf{R}^m)$ 是 $\mathbf{R}^d$ 上的集合值函数，且对于每个 $y \in \mathbf{R}^d$，集合 $\mathcal{D}(y)$ 都是凸集. 这些条件保证了对于任意参数 $y \in \mathbf{R}^d$，这都是一个凸优化问题.

定理

假设上述含参凸优化问题满足如下条件：

1. 目标函数 $f(x, y)$ 是关于 (x, y) 的凸函数；
2. 可行域映射 $y \mapsto \mathcal{D}(y)$ 的图像 $\{(x, y) : x \in \mathcal{D}(y)\}$ 是凸集.

如果对于任意 $y \in \mathbf{R}^d$，都有 $v(y) > -\infty$，那么，价值函数 $v(y)$ 是关于 y 的正常凸函数.

证明

对于任意 $y_1, y_2 \in \mathbf{R}^d$ 和 $\alpha \in (0, 1)$，需要证明

$$v(\alpha y_1 + (1-\alpha)y_2) \le \alpha v(y_1) + (1-\alpha)v(y_2).$$

如果 $v(y_1) = +\infty$ 或 $v(y_2) = +\infty$，那么不等式的右侧就是 $+\infty$，不等式必然成立. 否则，$v(y_1)$ 和 $v(y_2)$ 都是有限值. 对于任意 $\varepsilon > 0$ 和 $i = 1, 2$，都存在 $x_i \in \mathcal{D}(y_i)$ 使得 $f(x_i, y_i) < v(y_i) + \varepsilon$ 成立. 利用映射 $\mathcal{D}$ 的图像的凸性可知

$$\alpha x_1 + (1-\alpha)x_2 \in \mathcal{D}(\alpha y_1 + (1-\alpha)y_2).$$

也就是说，$\alpha x_1 + (1-\alpha)x_2$ 是参数为 $\alpha y_1 + (1-\alpha)y_2$ 的最优化问题的一个可行解. 利用最优化条件和目标函数的凸性可知

$$\begin{aligned} v(\alpha y_1 + (1-\alpha)y_2) &\le f(\alpha x_1 + (1-\alpha)x_2, \alpha y_1 + (1-\alpha)y_2) \\ &\le \alpha f(x_1, y_1) + (1-\alpha) f(x_2, y_2) \\ &< \alpha v(y_1) + (1-\alpha)v(y_2) + \varepsilon. \end{aligned}$$

因为 ε 的选取是任意的，令 $\varepsilon \to 0$，就有

$$v(\alpha y_1 + (1-\alpha)y_2) \le \alpha v(y_1) + (1-\alpha)v(y_2).$$

因此，价值函数 $v(y)$ 的凸性成立.

算法竞赛中，最为常见的凸优化问题就是线性规划问题.

推论

设 $c \in \mathbf{R}^n$，$A_1 \in \mathbf{R}^{d_1 \times n}$，$A_2 \in \mathbf{R}^{d_2 \times n}$，$y_1 \in \mathbf{R}^{d_1}$，$y_2 \in \mathbf{R}^{d_2}$. 考虑如下含参线性规划问题：

$$v(y_1, y_2) = \min_{x \in \mathbf{R}^n} c \cdot x \text{ subject to } A_1 x \le y_1, A_2 x = y_2, x \ge 0.$$

那么，价值函数 $v(y_1, y_2)$ 是关于 (y_1, y_2) 的凸函数.

无论是不等式约束，还是等式约束，线性规划的价值函数都是约束条件参数的凸函数.

很多图论问题都可以写成线性规划问题的形式：

- 网络流问题：最大流、最小割、最小费用流；
- 无负环的最短路问题；
- 二分图的最大（权）匹配、最小点覆盖等问题；

- 一般图的最大（权）匹配问题；
- 最小生成树问题[7].

因此，这些问题的价值函数都是这些问题的参数的凸（凹）函数.

⚠ 整数约束

利用图论模型为实际问题建模时，通常有隐含的整数限制，例如一条边只能选或不选、流量只能是整数等. 因此，它们只能转化为整数线性规划（integer linear programming, ILP）问题而非线性规划（LP）问题. 因为 ILP 问题并非凸优化问题，所以它的价值函数未必是该问题的参数的凸函数. 将一个 ILP 问题中的整数约束松弛掉后就得到一个 LP 问题，但后者未必存在满足整数约束的最优解. 因此，松弛整数约束后得到的 LP 的最优价值有可能严格优于相应的 ILP 问题，两者未必等价.

上文列举的那些图论问题，都可以写成一个 LP 问题而不需要施加整数约束；但对于其他的一些问题，例如一般图的最大独立集问题等，整数约束则是必要的. 另外，即使一个图论问题可以写成 LP 的形式，在该问题引入额外的线性约束条件后，仍然可能会破坏相应的 ILP 问题和 LP 问题的等价性，从而这个带约束的图论问题不再能够写成线性规划的形式.

例如，在费用流的语境下，有如下常见结论：

推论

最小费用流模型→11.23.4@p2935 中，最小费用 $v(m)$ 是流量 m 的凸函数.

证明

设有向图 $G=(V,E)$，边 (i,j) 的容量为 c_{ij}，单位流量的费用为 w_{ij}，源点和汇点分别为 s 和 t. 记决策变量为 $\{f_{ij}\}$，其中，f_{ij} 为边 $(i,j)\in E$ 的流量. 那么，最小费用流可以写成如下线性规划问题：

$$v(m)=\min_{\{f_{ij}\}}\sum_{(i,j)\in E}w_{ij}f_{ij}$$

$$\text{subject to}\sum_{(j,i)\in E}f_{ji}-\sum_{(i,j)\in E}f_{ij}=\begin{cases}-m, & i=s,\\ m, & i=t,\\ 0, & \text{otherwise},\end{cases}\qquad \forall i\in V,$$

$$0\le f_{ij}\le c_{ij},\ \forall (i,j)\in E.$$

因此，最小费用 $v(m)$ 是参数 m 的凸函数.

算法竞赛中很多问题都可以归约为网络流等图论问题，从而都可以通过类似的方式建立价值函数的凸性.

利用这一方法，可以得到种树问题的第一个凸性证明：

凸性证明一

种树问题的最大收益实际上可以通过如下最大费用最大流模型得出：

- 从源点 s 出发，向结点 r，连接一条容量为 m、费用为 0 的边；
- 从结点 r 出发，向每个奇数结点 $i=1,3,\cdots,2\lceil n/2\rceil-1$ 连接一条容量为 1、费用为 0 的边；
- 从每个偶数结点 $i=0,2,\cdots,2\lfloor n/2\rfloor$ 出发，向汇点 t 连接一条容量为 1、费用为 0 的边；
- 对于每个 $i=1,\cdots,n$，从 $i-1$ 和 i 中的奇数结点出发，向偶数结点连接一条容量为 1、费用为 a_i 的边.

[7]最小生成树问题有两种常见的写成线性规划问题的方法[7]：子回路消除模型（subtour-elimination formulation）和基于割集的模型（cut-based formulation）. 只有前一种建模方式能够保证得到的线性规划问题和原问题是等价的.

最终答案就是求得的最大费用. 将这一图论模型转换为相应的线性规划问题(具体见上文推论的证明), 那么, 总流量 m 将出现在表示边 (s,r) 的流量限制的不等式中. 由推论可知, 最大费用 $v(m)$ 是流量 m 的凹函数.

利用该费用流模型, 可以通过模拟费用流或 反悔贪心→5.6@p563 的方法在 $O(n\log n)$ 的复杂度内解决该问题.

利用状态转移方程

尽管状态转移方程无法提供有效的计算方式, 但是, 它常常可以用于证明状态函数 $f(i,j)$ 对于参数 j 具有凸性. 具体地说, 就是将 $f(i,\cdot)$ 这一函数视为 i 处的状态, 就可以将关于 $f(i,j)$ 的状态转移方程看作是关于 $f(i,\cdot)$ 的递推关系, 从而可以归纳地证明每个 $f(i,\cdot)$ 都是凸函数. 这类证明凸性的方法在 Slope Trick 优化 DP→7.15.5@p836 的场景中更为常见, 该页面也讨论了常见的保持凸性的变换.

这一方法同样可以用于证明种树问题的凸性:

凸性证明二

设 $f(i,j)$ 为前 i 个坑种 j 棵树时的最大收益. 考察如下状态转移方程:

$$f(i,j)=\max\{f(i-1,j),f(i-2,j-1)+a_i\}.$$

将这一状态转移方程看作是函数 $f(i,\cdot)$ 的递推关系式. 因为最值符号内涉及两个不同的函数, 这并不能表达为卷积上确界的形式. 但是, 仍然可以通过归纳的方法证明函数 $f(i,\cdot)$ 是凹函数.

实际上, 需要归纳地证明如下两点:

- $f(i,j)-f(i-2,j-1)$ 关于 j 递减;
- $f(i,j)-f(i-1,j)$ 关于 j 递增.

归纳起点是平凡的. 假设它们对于所有 $i-1$ 及之前的自然数都成立, 现在证明它对 i 也成立. 直接验证即可.

首先, 由归纳假设, 有

$$f(i-1,j)-f(i-2,j-1)=(f(i-1,j)-f(i-3,j-1))-(f(i-2,j-1)-f(i-3,j-1))$$

关于 j 递减. 所以, 有

$$f(i,j)-f(i-2,j-1)=\max\{f(i-1,j)-f(i-2,j-1),a_i\}$$

关于 j 递减, 且

$$f(i,j)-f(i-1,j)=\max\{0,a_i-(f(i-1,j)-f(i-2,j-1))\}$$

关于 j 递增. 这就完成了归纳.

进而, 有

$$j)-f(i,j-1)=(f(i,j)-f(i-2,j-1))-(f(i,j-1)-f(i-1,j-1))-(f(i-1,j-1)-f(i-2,j-$$

关于 j 递减. 这就说明 $f(i,j)$ 是关于 j 的凹函数, 从而, 价值函数 $v(m)=f(n,m)$ 是关于 m 的凹函数.

这个证明的一个副产品是, 对于任意 i, 都存在 p_i 使得

$$f(i,j)=\begin{cases}f(i-1,j), & j\le p_i,\\ f(i-2,j-1)+a_i, & j>p_i.\end{cases}$$

这说明, 可以通过平衡树直接维护序列 $f(i,\cdot)$, 复杂度是 $O(n\log^2 n)$ 的. 但是, 好处是可以处理任意种树间隔的一般情形, 且一次性地获得了所有 $v(m)$ 的值.

四边形不等式

算法竞赛中，另一类常见的具有凸性的问题是 区间分拆问题→7.15.4@p821. 该页面证明了，如果单个区间的成本函数满足四边形不等式，那么限制区间个数的区间分拆问题的最小成本是区间个数的凸函数. 该页面同样提供了一些判断某个函数 $w(l,r)$ 是否满足四边形不等式的方法. 最为直接的方法就是计算它的二阶混合差分：

$$\begin{aligned}\Delta_l\Delta_r w(l,r) &= \Delta_l(w(l,r+1)-w(l,r))\\ &= w(l+1,r+1)-w(l+1,r)-w(l,r+1)+w(l,r).\end{aligned}$$

函数 $w(l,r)$ 满足四边形不等式，当且仅当 $\Delta_l\Delta_r w(l,r)$ 非正. 直观上，满足四边形不等式的函数通常意味着，区间向两侧扩大——即左端点向左移动和右端点向右移动——具有某种协同效应.

种树问题同样可以看作是一个区间分拆问题，可以通过验证四边形不等式进行证明.

凸性证明三

在种树的收益序列前添加一个 a_0，它可以是任何值. 然后，种树问题就等价于将序列 $\{a_0,a_1,\cdots,a_n\}$ 分成 m 段，且每段的收益函数为

$$w(l,r)=\max_{i\in[l+1,r]} a_i$$

的区间分拆问题. 也就是说，每一段的收益是除去第一棵树外，其余树的收益的最大值——这就保证了间隔种树.

因为这是最大化问题，需要验证「交叉大于包含」，即对于任意 $a<b<c<d$，都成立

$$w(a,c)+w(b,d)\geq w(a,d)+w(b,c).$$

代入收益函数的表达式，并设

$$A=\max_{i\in[a+1,b]} a_i,\ B=\max_{i\in[b+1,c]} a_i,\ C=\max_{i\in[c+1,d]} a_i,$$

则要证明的不等式可以写作

$$\max\{A,B\}+\max\{B,C\}\geq\max\{A,B,C\}+B.$$

注意到不等号左侧的两项 $\max\{A,B\}$ 和 $\max\{B,C\}$ 中较大的那个就等于 $\max\{A,B,C\}$，而它们中较小的那个总是不小于 B，因此该不等式成立.

将种树问题转化为区间分拆问题后，只需要用 ST 表等方式预处理区间最值，可以单次 $O(1)$ 地计算单个区间的成本，就可以套用区间分拆问题的算法在 $O(n\log n\log L)$ 或 $O(n(n+m))$ 的时间复杂度内解决该问题. 该方法同样可以处理任意种树间隔的问题.

交换论证

组合优化问题中，证明价值函数的凸性常常会用到交换论证（exchange argument）. 具体地说，就是从参数为 $m-1$ 和 $m+1$ 的问题的最优解出发，通过交换部分元素，构造出参数为 m 且价值不超过 $(v(m-1)+v(m+1))/2$ 的可行解，从而利用 $v(m)$ 的最优性来证明凸性的论证方法. 相较于凸优化的情形，组合优化问题中并不存在自然地构造两个解的「中间形态」的方法，因此，交换论证的应用通常具有一定的技巧性.

⚠ 「边际成本递增」并不一定导致凸性

组合优化问题中，目标函数常常具有一些「边际成本递增」的性质，但是这并不必然导致凸性. 一个典型的例子是 [IOI 2005] Riv 河流[8]，该问题的链上版本是满足四边形不等式因而具有凸性的，但是树上版本存在凸性不成立的例子.

用于刻画「边际成本递增」的一个常见性质是函数的超模性（supermodularity）. 对于有限集合 X 的子集族 $\mathcal{P}X$ 上的函数 $f:\mathcal{P}X\to\mathbf{R}$，如果它满足以下两条等价性质之一：

1. （交叉小于包含）对于任何子集 $A, B \subseteq X$，都有 $f(A) + f(B) \leq f(A \cup B) + f(A \cap B)$ 成立；
2. （边际成本递增）对于任何子集 $A \subseteq B \subseteq X$ 以及 $x \in X \setminus B$，都有 $f(A \cup \{x\}) - f(A) \leq f(B \cup \{x\}) - f(B)$ 成立；

就称函数 f 是 **超模的**（supermodular）. 但是，超模函数作为目标函数的最优化问题中，价值函数

$$v(m) = \min_{A \subseteq X} f(A) \text{ subject to } | A | = m$$

未必 是 m 的凸函数. 究其原因，就是从子集大小分别为 $m-1$ 和 $m+1$ 的最优解，一般来说是无法构造出子集大小为 m 且满足前述价值函数大小关系的可行解的.

交换论证提供了种树问题凸性的又一种证明方式.

凸性证明四

利用交换论证. 设种 $m-1$ 棵树和种 $m+1$ 棵树的最优方案分别由 $\{x_i^{(m-1)}\} \in \{0,1\}^n$ 和 $\{x_i^{(m+1)}\} \in \{0,1\}^n$ 给出，其中，取值为 1 表示该坑位种了一棵树，取值为 0 表示该坑位没有种树. 定义序列 $\{z_i\} \in \{0, \pm 1\}^n$ 满足

$$z_i = x_i^{(m+1)} - x_i^{(m-1)},\ i = 1, \cdots, n.$$

这个序列标记了两个种树方案的差异. 这一序列中取值为 0 的位置表示该坑位要么在两个方案中都种了一棵树，要么在两个方案中都没有种树；而取值为 -1 和 $+1$ 的位置则分别表示只在方案 $x^{(m-1)}$ 或只在方案 $x^{(m+1)}$ 中，该坑位种了一棵树. 因为任何方案中都不能在相邻的坑位种树，所以有如下观察：

- 连续的非零子段中，z_i 的取值必然在 ± 1 之间相互交错；
- 极大的连续非零子段左右两侧的 0，必然表示在两个方案中都没有种树.

因此，如果某个极大的连续非零子段中，z_i 的和恰好为 $+1$，也就是说，在该段坑位中，方案 $x^{(m+1)}$ 比方案 $x^{(m-1)}$ 多种了一棵树，那么就可以在该段内交换两个方案的种树位置. 这样，就得到了两个各种 m 棵树的可行方案. 因为没有改变两个方案中总的种树的位置和数量，只是将它们重新分配，所以总的收益不变，仍然是 $v(m-1) + v(m+1)$. 但是，这两个种 m 棵树的方案未必是最优的，因此，它们各自的收益不会超过 $v(m)$. 这就证明了

$$v(m-1) + v(m+1) \leq 2v(m),$$

也就是说，$v(m)$ 是关于 m 的凹函数.

现在，只剩下一个问题，就是和恰好为 $+1$ 的极大连续非零子段是否存在. 因为是若干个交错的 ± 1 相加，一个连续非零子段的和只能是 0 或 ± 1. 又因为所有这些极大连续非零子段的和等于 2，所以，一定存在至少两个和恰好为 $+1$ 的极大连续非零子段. 这就完成了证明.

例题

本节介绍几个不同场景下应用 WQS 二分方法的例题.

模板题目

Luogu P1484 种树[9]

有 n 个树坑，**至多** 种 m 棵树. 树不能栽种于相邻的两个坑. 给定长度为 n 的序列 $\{a_i\}$，表示在每个坑种树的收益，收益可正可负. 求最大可能的收益和.

解答

与前文讨论的种树问题稍有不同，本题要求至多种 m 棵树，而非恰好种 m 棵树．仍然用 $v(m)$ 表示前文讨论的种树问题的价值函数，本题的答案实际上是 $\tilde{v}(m)=\max_{k\le m} v(k)$．因为 $v(m)$ 是凹函数，也就是一个单峰函数，本题的答案相当于只保留 $v(m)$ 上升至峰顶的部分，然后函数会一直留在峰顶；这相当于仅仅保留切线斜率非负的部分．因此，本题与前文讨论的题目的唯一差别，就是 WQS 二分时，初始的斜率范围为 $[0,\max_i a_i]$ 而非 $[\min_i a_i,\max_i a_i]$．

应用 WQS 二分的方法移除数量限制后，问题转化为计算链上最大权独立集，只是将原本的收益 $\{a_i\}$ 替换为了 $\{a_i+k\}$．这是经典的动态规划题目．可以设 $f(i,j)$ 为第 i 个坑位选择种树（$j=1$）或不种树（$j=0$）时，前 i 个树坑的子问题的最大收益．由此，可以写出状态转移方程为

$$f(i,0)=\max\{f(i-1,0),f(i-1,1)\},$$

$$f(i,1)=f(i-1,0)+a_i+k.$$

初始条件为 $f(0,0)=0$ 和 $f(0,1)=-\infty$，最终答案为 $\max\{f(n,0),f(n,1)\}$．单次计算复杂度为 $O(n)$，整体时间复杂度为 $O(n\log L)$，其中，$L=\max_i|a_i|$．

参考实现如下：

- 传统方法

```cpp
#include <algorithm>
#include <cstring>
#include <iostream>
#include <tuple>
#include <vector>

int main() {
  int n, m;
  std::cin >> n >> m;
  std::vector<int> a(n + 1);
  for (int i = 1; i <= n; ++i) std::cin >> a[i];
  // Calculate h(k) = max_x f(x) - k * g(x).
  // Meanwhile, obtain the maximum value g(x) of the optimizer x.
  auto calc = [&](int k) -> std::pair<long long, int> {
    long long dp[2] = {0, -0x3f3f3f3f3f3f3f3f};
    int opt[2] = {0, 0};
    for (int i = 1; i <= n; ++i) {
      long long tmp_dp[2];
      int tmp_opt[2];
      if (dp[0] > dp[1]) {
        tmp_dp[0] = dp[0];
        tmp_opt[0] = opt[0];
      } else if (dp[1] > dp[0]) {
        tmp_dp[0] = dp[1];
        tmp_opt[0] = opt[1];
      } else {
        tmp_dp[0] = dp[0];
        tmp_opt[0] = std::max(opt[0], opt[1]);
      }
      tmp_dp[1] = dp[0] + a[i] - k;
      tmp_opt[1] = opt[0] + 1;
      std::memcpy(dp, tmp_dp, sizeof(dp));
      std::memcpy(opt, tmp_opt, sizeof(opt));
```

```
    }
    long long val;
    int opt_m;
    if (dp[0] > dp[1]) {
      val = dp[0];
      opt_m = opt[0];
    } else if (dp[1] > dp[0]) {
      val = dp[1];
      opt_m = opt[1];
    } else {
      val = dp[0];
      opt_m = std::max(opt[0], opt[1]);
    }
    return {val, opt_m};
  };
  // WQS binary search.
  long long val, tar_val;
  int opt_m, tar_k;
  std::tie(val, opt_m) = calc(0);
  if (opt_m <= m) {
    // Have already reached the peak.
    tar_k = 0;
    tar_val = val;
  } else {
    // Find the maximum k such that g(x) >= m.
    int ll = 0, rr = 1000000;
    while (ll <= rr) {
      int mm = (ll + rr) / 2;
      std::tie(val, opt_m) = calc(mm);
      if (opt_m >= m) {
        tar_k = mm;
        tar_val = val;
        ll = mm + 1;
      } else {
        rr = mm - 1;
      }
    }
  }
  long long res = tar_val + (long long)tar_k * m;
  std::cout << res << std::endl;
  return 0;
}

```

- 对偶方法

```
#include <algorithm>
#include <iostream>
#include <tuple>
#include <type_traits>
#include <vector>

// Golden section search on integer domain (unimodal function)
```

```cpp
template <typename T, typename F>
typename std::enable_if<
    std::is_integral<T>::value,
    std::pair<T, decltype(std::declval<F>()(std::declval<T>()))>>::type
golden_section_search(T ll, T rr, F func) {
  constexpr long double phi = 0.61803398874989484820458683436L;
  T ml = ll + static_cast<T>((rr - ll) * (1 - phi));
  T mr = ll + static_cast<T>((rr - ll) * phi);
  auto fl = func(ml), fr = func(mr);
  while (ml < mr) {
    if (fl > fr) {
      rr = mr;
      mr = ml;
      fr = fl;
      ml = ll + static_cast<T>((rr - ll) * (1 - phi));
      fl = func(ml);
    } else {
      ll = ml;
      ml = mr;
      fl = fr;
      mr = ll + static_cast<T>((rr - ll) * phi);
      fr = func(mr);
    }
  }
  T best_x = ll;
  auto best_val = func(ll);
  for (T i = ll + 1; i <= rr; ++i) {
    auto val = func(i);
    if (val > best_val) {
      best_val = val;
      best_x = i;
    }
  }
  return {best_x, best_val};
}

int main() {
  int n, m;
  std::cin >> n >> m;
  std::vector<int> a(n + 1);
  for (int i = 1; i <= n; ++i) std::cin >> a[i];
  // Calculate h(k) = max_x f(x) + k * g(x).
  auto calc = [&](int k) -> long long {
    long long dp[2] = {0, -0x3f3f3f3f3f3f3f3f};
    for (int i = 1; i <= n; ++i) {
      std::tie(dp[0], dp[1]) =
          std::make_pair(std::max(dp[0], dp[1]), dp[0] + a[i] + k);
    }
    return std::max(dp[0], dp[1]);
  };
  // Solve the dual problem to find v(m).
  // Implemented as a minimization problem by adding negative signs.
```

```
60   // Only consider tangent lines of negative slopes to ignore the part
61   //     of the curve after the peak.
62   auto res = -golden_section_search(-1000000, 0, [&](int k) -> long
long {
63                 return -calc(k) + (long long)k * m;
64               }).second;
65   std::cout << res << std::endl;
66   return 0;
67 }
68
```

Luogu P2619 [国家集训队] Tree I[10]

给定一张带权无向连通图，每条边是黑色或白色．求恰有 m 条白边的生成树的最小权．

解答

首先，通过交换论证可以证明 $v(m)$ 是凸函数．不妨假设所有边的边权各不相同：那些存在两边边权相同的情形，可以通过微扰使之变为边权各不相同的情形；然后只要令微扰的幅度趋近于零，就可以证明函数的凸性在极限情形——也就是存在两边边权相同的情形——仍然成立．证明的关键在于如下引理：[8]

引理

设 S 和 T 是无向连通图 $G=(V,E)$ 的两个生成树．对于任意 $e\in S\setminus T$，都存在至少一条边 $f\in T\setminus S$，使得 $S-e+f$ 和 $T-f+e$ 都是图 G 的生成树．

证明

设 $e=(u,v)$，且 P 是树 T 中连接 u 和 v 的唯一一条路径．因为 $P+e$ 是图 $T+e$ 中唯一的环路，所以删掉 P 中的任何一条边 f 都可以使得 $T-f+e$ 是一棵生成树．与此同时，图 $S-e$ 是有两个连通分量的森林，它们的顶点集分别记作 V_1 和 V_2，所以，只要选择边 $f\in P$ 使得 f 连通了 V_1 和 V_2，就能保证 $S-e+f$ 是一棵生成树．这样的边 f 总是存在的，因为 u 和 v 分别属于 V_1 和 V_2，而 P 连接了 u 和 v．而且，$f\notin S$，因为图 $S-e$ 中，V_1 和 V_2 并不是连通的．这就完成了证明．

设 T_{m-1} 和 T_{m+1} 是白边数量分别为 $m-1$ 和 $m+1$ 的最小生成树．设 e 是 $T_{m+1}\setminus T_{m-1}$ 的一条白边，对它应用上述引理可知，存在边 $f\in T_{m-1}\setminus T_{m+1}$，使得 $T'=T_{m+1}-e+f$ 和 $T''=T_{m-1}+e-f$ 都是生成树．因为只是交换了一对边，所以，树 T' 和树 T'' 的边权和仍然是 $v(m-1)+v(m+1)$．进而，分两种情形讨论：

- 如果 f 是一条黑边，那么，T' 和 T'' 中的白边数量都是 m．它们各自的边权和都不会小于 $v(m)$．这就证明了 $2v(m)\le v(m-1)+v(m+1)$，故而 $v(m)$ 关于 m 是凸的．
- 如果 f 是一条白边，那么，T' 和 T'' 的白边数量分别是 $m+1$ 和 $m-1$，所以它们的边权和分别不小于 $v(m+1)$ 和 $v(m-1)$．但是，上面已经说明，它们的边权和加在一起就等于 $v(m-$

[8] 这一引理对于一般的 拟阵→9.22@p1965 也是成立的．它称为 **对称基交换性质**（symmetric base-exchange property），相关资料可以参考 Wikipedia 页面[11]．因此，本题关于凸性的结论可以推广到一般的拟阵上．

1) + v(m + 1). 这说明，T' 的边权和就等于 $v(m+1)$. 将 T' 与 T_{m+1} 比较可知，e 和 f 的边权必然相等. 这与假设矛盾，所以该情形并不成立.

这就证明了 $v(m)$ 是 m 的凸函数.

建立了函数 $v(m)$ 的凸性后，就可以用 WQS 二分解决该问题. 移除数量限制并将每条白边的权重都减去 k，并求解最小生成树问题. 为此，可以应用 Kruskal 算法→11.10.1@p2764. 利用并查集维护连通性，算法的复杂度就是 $O(E \log E + E\alpha(V))$，其中，E 和 V 分别为边数和顶点数，$\alpha(\cdot)$ 为反 Ackerman 函数. 复杂度的主要部分 $O(E \log E)$ 是给边排序的复杂度，在本题中可以进一步优化. 虽然在 WQS 二分的过程中需要多次计算最小生成树，但是每次只有白边的边权会整体加减一个数. 所以，可以在预处理时给白边、黑边分别排序，然后每次计算最小生成树时，只需要将调整完权重后的白边和黑边归并到一起就可以了. 这样，整体复杂度就降低到了 $O(E \log E + E\alpha(V) \log L)$，其中，$L$ 为边权的取值范围的长度.

参考实现如下：

- 传统方法

```cpp
#include <algorithm>
#include <array>
#include <iostream>
#include <numeric>
#include <tuple>
#include <vector>

class DisjointSet {
  std::vector<int> fa, sz;

  int find(int x) { return fa[x] == x ? x : fa[x] = find(fa[x]); }

 public:
  DisjointSet(int n) : fa(n), sz(n, 1) { std::iota(fa.begin(), fa.end(), 0); }

  bool unite(int x, int y) {
    x = find(x);
    y = find(y);
    if (x == y) return false;
    if (sz[x] < sz[y]) std::swap(x, y);
    fa[y] = x;
    sz[x] += sz[y];
    return true;
  }
};

int main() {
  int V, E, m;
  std::cin >> V >> E >> m;
  std::array<std::vector<std::array<int, 3>>, 2> edges;
  edges[0].reserve(E);  // white edges.
  edges[1].reserve(E);  // black edges.
  for (int i = 0; i < E; ++i) {
    int u, v, c, w;
    std::cin >> u >> v >> w >> c;
```

```
    edges[c].push_back({u, v, w});
  }
  // Sort edges.
  std::sort(edges[0].begin(), edges[0].end(),
            [&](const auto& lhs, const auto& rhs) -> bool {
              return lhs[2] < rhs[2];
            });
  std::sort(edges[1].begin(), edges[1].end(),
            [&](const auto& lhs, const auto& rhs) -> bool {
              return lhs[2] < rhs[2];
            });
  // Calculate h(k) = min_x f(x) - k * g(x) by Kruskal algorithm.
  // Use white edges first, whenever possible.
  auto calc = [&](int k) -> std::pair<int, int> {
    int res = 0, cnt = 0;
    DisjointSet djs(V);
    int i[2] = {};
    while (i[0] < edges[0].size() && i[1] < edges[1].size()) {
      int c = edges[0][i[0]][2] - k > edges[1][i[1]][2];
      if (djs.unite(edges[c][i[c]][0], edges[c][i[c]][1])) {
        res += edges[c][i[c]][2] - (c ? 0 : k);
        cnt += c ? 0 : 1;
      }
      ++i[c];
    }
    while (i[0] < edges[0].size()) {
      if (djs.unite(edges[0][i[0]][0], edges[0][i[0]][1])) {
        res += edges[0][i[0]][2] - k;
        ++cnt;
      }
      ++i[0];
    }
    while (i[1] < edges[1].size()) {
      if (djs.unite(edges[1][i[1]][0], edges[1][i[1]][1])) {
        res += edges[1][i[1]][2];
      }
      ++i[1];
    }
    return {res, cnt};
  };
  // WQS binary search.
  // Find the minimum k such that g(x) >= m.
  int val, opt_m, tar_val, tar_k;
  int ll = -100, rr = 100;
  while (ll <= rr) {
    int mm = ll + (rr - ll) / 2;
    std::tie(val, opt_m) = calc(mm);
    if (opt_m >= m) {
      tar_val = val;
      tar_k = mm;
      rr = mm - 1;
    } else {
```

```
      ll = mm + 1;
    }
  }
  int res = tar_val + tar_k * m;
  std::cout << res << std::endl;
  return 0;
}

```

- 对偶方法

```
#include <algorithm>
#include <array>
#include <iostream>
#include <numeric>
#include <tuple>
#include <type_traits>
#include <vector>

// Golden section search on integer domain (unimodal function)
template <typename T, typename F>
typename std::enable_if<
    std::is_integral<T>::value,
    std::pair<T, decltype(std::declval<F>()(std::declval<T>()))>>::type
golden_section_search(T ll, T rr, F func) {
  constexpr long double phi = 0.61803398874989484820458683436L;
  T ml = ll + static_cast<T>((rr - ll) * (1 - phi));
  T mr = ll + static_cast<T>((rr - ll) * phi);
  auto fl = func(ml), fr = func(mr);
  while (ml < mr) {
    if (fl > fr) {
      rr = mr;
      mr = ml;
      fr = fl;
      ml = ll + static_cast<T>((rr - ll) * (1 - phi));
      fl = func(ml);
    } else {
      ll = ml;
      ml = mr;
      fl = fr;
      mr = ll + static_cast<T>((rr - ll) * phi);
      fr = func(mr);
    }
  }
  T best_x = ll;
  auto best_val = func(ll);
  for (T i = ll + 1; i <= rr; ++i) {
    auto val = func(i);
    if (val > best_val) {
      best_val = val;
      best_x = i;
    }
  }
```

```
  return {best_x, best_val};
}

class DisjointSet {
  std::vector<int> fa, sz;

  int find(int x) { return fa[x] == x ? x : fa[x] = find(fa[x]); }

 public:
  DisjointSet(int n) : fa(n), sz(n, 1) { std::iota(fa.begin(), fa.end(), 0); }

  bool unite(int x, int y) {
    x = find(x);
    y = find(y);
    if (x == y) return false;
    if (sz[x] < sz[y]) std::swap(x, y);
    fa[y] = x;
    sz[x] += sz[y];
    return true;
  }
};

int main() {
  int V, E, m;
  std::cin >> V >> E >> m;
  std::array<std::vector<std::array<int, 3>>, 2> edges;
  edges[0].reserve(E);  // white edges.
  edges[1].reserve(E);  // black edges.
  for (int i = 0; i < E; ++i) {
    int u, v, c, w;
    std::cin >> u >> v >> w >> c;
    edges[c].push_back({u, v, w});
  }
  // Sort edges.
  std::sort(edges[0].begin(), edges[0].end(),
            [&](const auto& lhs, const auto& rhs) -> bool {
              return lhs[2] < rhs[2];
            });
  std::sort(edges[1].begin(), edges[1].end(),
            [&](const auto& lhs, const auto& rhs) -> bool {
              return lhs[2] < rhs[2];
            });
  // Calculate h(k) = min_x f(x) - k * g(x) by Kruskal algorithm.
  auto calc = [&](int k) -> int {
    int res = 0;
    DisjointSet djs(V);
    int i[2] = {};
    while (i[0] < edges[0].size() && i[1] < edges[1].size()) {
      int c = edges[0][i[0]][2] - k > edges[1][i[1]][2];
      if (djs.unite(edges[c][i[c]][0], edges[c][i[c]][1])) {
        res += edges[c][i[c]][2] - (c ? 0 : k);
```

```
      }
      ++i[c];
    }
    while (i[0] < edges[0].size()) {
      if (djs.unite(edges[0][i[0]][0], edges[0][i[0]][1])) {
        res += edges[0][i[0]][2] - k;
      }
      ++i[0];
    }
    while (i[1] < edges[1].size()) {
      if (djs.unite(edges[1][i[1]][0], edges[1][i[1]][1])) {
        res += edges[1][i[1]][2];
      }
      ++i[1];
    }
    return res;
  };
  // Solve the dual problem to find v(m).
  auto res = golden_section_search(-100, 100, [&](int k) -> int {
               return calc(k) + k * m;
             }).second;
  std::cout << res << std::endl;
  return 0;
}

```

区间分拆问题

Luogu P6246 [IOI 2000] 邮局 加强版 加强版[12]

给定长度为 n 且递增的正整数序列 $\{a_i\}$ 表示一条高速公路旁的 n 个村庄的位置，需要修建 m 个邮局．邮局位置的选择，需要最小化所有村庄与其最近邮局的距离之和．求这个最小值．

解答

这是典型的 区间分拆问题→7.15.4@p821．二分队列的实现细节请参考该页面．

每个邮局服务离它最近的村庄，那么，这些村庄必然是高速公路旁连续的若干个村庄．所以，修建 m 个邮局，就相当于将所有村庄划分为连续的 m 段，并为每一段村庄修建一个成本最低的邮局．众所周知，邮局应当修建在村庄位置的中位数的位置．由此，可以写出区间 $[l,r]$ 的成本函数为

$$w(l,r)=\sum_{i=l}^{r}\left|a_i-a_{\lfloor (l+r)/2\rfloor}\right|.$$

它满足四边形不等式，因为它的二阶混合差分非正：

$$\Delta_l\Delta_r w(l,r)=\Delta_l(a_{r+1}-a_{\lfloor (l+r+1)/2\rfloor})=a_{\lfloor (l+r+1)/2\rfloor}-a_{\lfloor (l+r+2)/2\rfloor}\le 0.$$

这说明，可以通过二分队列结合 WQS 二分的方法在 $O(n\log n\log L)$ 的复杂度内求解该问题．

参考实现如下：

- 传统方法

```
#include <cmath>
```

```cpp
#include <deque>
#include <iostream>
#include <tuple>
#include <type_traits>
#include <vector>

// Monotone decision DP.
// This solves f(i) = min f(j-1) + w(j,i) s.t. 1 <= j <= i.
// Also records the minimal optimal decision j for each f(i).
template <typename W>
std::pair<std::vector<decltype(std::declval<W>()(0,
0))>, std::vector<int>>
monotone_decision_opt_dp(int n, W ww) {
  using ValueT = decltype(std::declval<W>()(0, 0));
  std::vector<ValueT> f(n + 1);
  std::vector<int> opt(n + 1), lt(n + 1), rt(n + 1);
  std::deque<int> dq;
  auto w = [&](int j, int i) -> ValueT { return ww(j, i) + f[j - 1]; };
  for (int j = 1; j <= n; ++j) {
    if (!dq.empty() && rt[dq.front()] < j) dq.pop_front();
    if (!dq.empty()) lt[dq.front()] = j;
    while (!dq.empty() && w(j, lt[dq.back()]) < w(dq.back(),
lt[dq.back()])) {
      dq.pop_back();
    }
    if (dq.empty()) {
      lt[j] = j;
      rt[j] = n;
      dq.emplace_back(j);
    } else if (w(j, rt[dq.back()]) >= w(dq.back(), rt[dq.back()])) {
      if (rt[dq.back()] < n) {
        lt[j] = rt[dq.back()] + 1;
        rt[j] = n;
        dq.emplace_back(j);
      }
    } else {
      int ll = lt[dq.back()], rr = rt[dq.back()], i = rr;
      while (ll <= rr) {
        int mm = (ll + rr) / 2;
        if (w(j, mm) < w(dq.back(), mm)) {
          i = mm;
          rr = mm - 1;
        } else {
          ll = mm + 1;
        }
      }
      rt[dq.back()] = i - 1;
      lt[j] = i;
      rt[j] = n;
      dq.emplace_back(j);
    }
    f[j] = w(dq.front(), j);
```

```
    opt[j] = dq.front();
  }
  return {f, opt};
}

int main() {
  int n, m;
  std::cin >> n >> m;
  std::vector<long long> a(n + 1), ps(n + 1);
  for (int i = 1; i <= n; ++i) {
    std::cin >> a[i];
    ps[i] = ps[i - 1] + a[i];
  }
  // Cost function for interval [l,r].
  auto w = [&](int j, int i) -> long long {
    int mm = j + (i - j) / 2;
    return ps[i] + ps[j - 1] - 2 * ps[mm] + (2 * mm - j - i + 1)
* a[mm];
  };
  // Calculate h(k) = min_x f(x) - k * g(x).
  // Also record the minimum of optimal number of segments.
  auto calc = [&](long long k) -> std::pair<long long, int> {
    auto res = monotone_decision_opt_dp(
        n, [&](int j, int i) -> long long { return w(j, i) - k; });
    auto val = res.first[n];
    int cnt = 0;
    for (int i = n; i; i = res.second[i] - 1) {
      ++cnt;
    }
    return {val, cnt};
  };
  // WQS binary search.
  // Find the largest k such that g(x) <= m.
  long long val, tar_val;
  int opt_m, tar_k;
  long long ll = -(1LL << 32), rr = 0;
  while (ll <= rr) {
    long long mm = ll + (rr - ll) / 2;
    std::tie(val, opt_m) = calc(mm);
    if (opt_m <= m) {
      tar_val = val;
      tar_k = mm;
      ll = mm + 1;
    } else {
      rr = mm - 1;
    }
  }
  long long res = tar_val + (long long)tar_k * m;
  std::cout << res << std::endl;
  return 0;
}

```

- 对偶方法

```cpp
#include <cmath>
#include <deque>
#include <iostream>
#include <type_traits>
#include <utility>
#include <vector>

// Monotone decision DP.
// This solves f(i) = min f(j-1) + w(j,i) s.t. 1 <= j <= i.
// Also records the minimal optimal decision j for each f(i).
template <typename W>
std::pair<std::vector<decltype(std::declval<W>()(0,
0))>, std::vector<int>>
monotone_decision_opt_dp(int n, W ww) {
  using ValueT = decltype(std::declval<W>()(0, 0));
  std::vector<ValueT> f(n + 1);
  std::vector<int> opt(n + 1), lt(n + 1), rt(n + 1);
  std::deque<int> dq;
  auto w = [&](int j, int i) -> ValueT { return ww(j, i) + f[j - 1]; };
  for (int j = 1; j <= n; ++j) {
    if (!dq.empty() && rt[dq.front()] < j) dq.pop_front();
    if (!dq.empty()) lt[dq.front()] = j;
    while (!dq.empty() && w(j, lt[dq.back()]) < w(dq.back(),
lt[dq.back()])) {
      dq.pop_back();
    }
    if (dq.empty()) {
      lt[j] = j;
      rt[j] = n;
      dq.emplace_back(j);
    } else if (w(j, rt[dq.back()]) >= w(dq.back(), rt[dq.back()])) {
      if (rt[dq.back()] < n) {
        lt[j] = rt[dq.back()] + 1;
        rt[j] = n;
        dq.emplace_back(j);
      }
    } else {
      int ll = lt[dq.back()], rr = rt[dq.back()], i = rr;
      while (ll <= rr) {
        int mm = (ll + rr) / 2;
        if (w(j, mm) < w(dq.back(), mm)) {
          i = mm;
          rr = mm - 1;
        } else {
          ll = mm + 1;
        }
      }
      rt[dq.back()] = i - 1;
      lt[j] = i;
      rt[j] = n;
      dq.emplace_back(j);
```

```
    }
    f[j] = w(dq.front(), j);
    opt[j] = dq.front();
  }
  return {f, opt};
}

// Golden section search on integer domain (unimodal function)
template <typename T, typename F>
typename std::enable_if<
    std::is_integral<T>::value,
    std::pair<T, decltype(std::declval<F>()(std::declval<T>()))>>::type
golden_section_search(T ll, T rr, F func) {
  constexpr long double phi = 0.6180339887498948482045868343L;
  T ml = ll + static_cast<T>((rr - ll) * (1 - phi));
  T mr = ll + static_cast<T>((rr - ll) * phi);
  auto fl = func(ml), fr = func(mr);
  while (ml < mr) {
    if (fl > fr) {
      rr = mr;
      mr = ml;
      fr = fl;
      ml = ll + static_cast<T>((rr - ll) * (1 - phi));
      fl = func(ml);
    } else {
      ll = ml;
      ml = mr;
      fl = fr;
      mr = ll + static_cast<T>((rr - ll) * phi);
      fr = func(mr);
    }
  }
  T best_x = ll;
  auto best_val = func(ll);
  for (T i = ll + 1; i <= rr; ++i) {
    auto val = func(i);
    if (val > best_val) {
      best_val = val;
      best_x = i;
    }
  }
  return {best_x, best_val};
}

int main() {
  int n, m;
  std::cin >> n >> m;
  std::vector<long long> a(n + 1), ps(n + 1);
  for (int i = 1; i <= n; ++i) {
    std::cin >> a[i];
    ps[i] = ps[i - 1] + a[i];
  }
```

```
  // Cost function for interval [l,r].
  auto w = [&](int j, int i) -> long long {
    int mm = j + (i - j) / 2;
    return ps[i] + ps[j - 1] - 2 * ps[mm] + (2 * mm - j - i + 1) * a[mm];
  };
  // Calculate h(k) = min_x f(x) - k * g(x).
  auto solve = [&](long long k) -> long long {
    return monotone_decision_opt_dp(
               n, [&](int j, int i) -> long long { return w(j, i) - k; })
               .first[n] +
           k * m;
  };
  // Solve the dual problem to find v(m).
  auto res = golden_section_search(-(1LL << 32), 0LL, solve).second;
  std::cout << res << std::endl;
  return 0;
}
```

二维的限制条件

Codeforces 739 E. Gosha is hunting[13]

有 n 只神奇宝贝，序列 $\{p_i\}$ 和 $\{q_i\}$ 分别表示用宝贝球和超级球抓到第 i 只神奇宝贝的概率．可以向一个神奇宝贝扔一个宝贝球，或者扔一个超级球，或者两种球各扔一个，或者什么球都不扔．现有 m_1 个宝贝球和 m_2 个超级球，需要合理分配，并同时扔出．求抓到的神奇宝贝的期望数量的最大值．单次抓捕成功与否，与其它抓捕的结果无关．

更一般地，可以抽象为如下问题：

给定长度为 n 的三个正实数序列 $\{A_i\},\{B_i\},\{C_i\}$，而且，对所有 $i=1,\cdots,n$ 都有 $C_i \le A_i + B_i$ 成立．求最优的下标集合 X 和 Y 满足 $|X| = m_1$ 和 $|Y| = m_2$ 且最大化

$$\sum_{i\in X\setminus Y} A_i + \sum_{i\in Y\setminus X} B_i + \sum_{i\in X\cap Y} C_i.$$

解答

原问题可以看作是这个更一般的问题在

$$A_i = p_i,\ B_i = q_i,\ C_i = p_i + q_i - p_i q_i$$

时的特殊情形．因此，只需要讨论更一般的问题的解决方案就可以了．

用 $v(m_1,m_2)$ 表示该问题的价值函数，需要证明它是关于 (m_1,m_2) 的凹函数．考虑如下费用流模型：

- 从源点 s 出发，分别向结点 x 和 y 连一条边，容量分别为 m_1 和 m_2，费用均为 0；
- 对于所有 $i=1,\cdots,n$，分别从结点 x 和 y 向结点 i 连一条边，容量均为 1，费用分别为 A_i 和 B_i；
- 对于所有 $i=1,\cdots,n$，从结点 i 出发向汇点 t 连两条边，容量均为 1，费用分别为 0 和 $C_i - A_i - B_i$．

问题的答案就是该费用流模型的最大费用最大流. 条件 $C_i - A_i - B_i \le 0$ 保证了当流经结点 i 的流量为 1 时, 会优先选择费用为 0 的那条边流出. 将这个费用流模型写成线性规划问题, 那么, m_1 和 m_2 就会分别出现在表示边 (s, x) 和边 (s, y) 的流量限制的不等式中. 因此, $v(m_1, m_2)$ 确实是 (m_1, m_2) 的凹函数.

为了应用 WQS 二分, 需要考虑移除数量限制后的最优化问题. 设 k_1 和 k_2 分别为将一个下标放入集合 X 和 Y 时获得的额外奖励. 没有数量限制后, 关于每个下标的决策都是独立的, 因此, 有

$$h(k_1, k_2) = \sum_{i=1}^{n} \max\{0, A_i + k_1, B_i + k_2, C_i + k_1 + k_2\}.$$

原问题的答案就由

$$v(m_1, m_2) = \min_{k_1, k_2} h(k_1, k_2) - k_1 m_1 - k_2 m_2$$

给出. 总的时间复杂度为 $O(n \log^2 L)$, 其中, $O(\log L)$ 为对单个维度二分的次数.

抓捕神奇宝贝问题的参考实现如下:

```cpp
#include <algorithm>
#include <iomanip>
#include <iostream>
#include <tuple>
#include <type_traits>
#include <vector>

// Golden section search on floating-point domain (unimodal function)
template <typename T, typename F>
typename std::enable_if<
    std::is_floating_point<T>::value,
    std::pair<T, decltype(std::declval<F>()(std::declval<T>()))>>::type
golden_section_search(T ll, T rr, F func, T eps) {
  constexpr long double phi = 0.61803398874989484820458683436L;
  T ml = ll + (rr - ll) * (1 - phi);
  T mr = ll + (rr - ll) * phi;
  auto fl = func(ml), fr = func(mr);
  while ((rr - ll) > eps) {
    if (fl > fr) {
      rr = mr;
      mr = ml;
      fr = fl;
      ml = ll + (rr - ll) * (1 - phi);
      fl = func(ml);
    } else {
      ll = ml;
      ml = mr;
      fl = fr;
      mr = ll + (rr - ll) * phi;
      fr = func(mr);
    }
  }
  T mid = (ll + rr) / 2;
  return {mid, func(mid)};
}

int main() {
```

```cpp
  int n, m1, m2;
  std::cin >> n >> m1 >> m2;
  std::vector<long double> p(n + 1), q(n + 1);
  for (int i = 1; i <= n; ++i) std::cin >> p[i];
  for (int i = 1; i <= n; ++i) std::cin >> q[i];
  // Calculate h(k1,k2).
  auto solve = [&](long double k1, long double k2) -> long double {
    long double res = 0;
    for (int i = 1; i <= n; ++i) {
      res += std::max(
          {0.0l, p[i] + k1, q[i] + k2, p[i] + q[i] - p[i] * q[i] + k1
+ k2});
    }
    return res;
  };
  // Solve the dual problem to find v(m1,m2).
  // Implemented as a minimization problem by adding negative signs.
  auto res = -golden_section_search(
                  -1.0l, 0.0l,
                  [&](long double k1) -> long double {
                    return golden_section_search(
                                -1.0l, 0.0l,
                                [&](long double k2) -> long double {
                                  return -solve(k1, k2) + k2 * m2;
                                },
                                1e-8l)
                                .second +
                           k1 * m1;
                  },
                  1e-8l)
                  .second;
  std::cout << std::fixed << std::setprecision(10) << res << std::endl;
  return 0;
}

```

更广泛的限制条件

Codeforces 1661 F. Teleporters[14]

有 n 条线段，它们的长度由序列 $\{a_i\}$ 给出．可以将它们任意切割为若干条整数长度的线段，目标是最小化所有线段长度平方的总和．求至少需要切割多少次，才能使这个平方和降到不超过 V．

解答

设将长度为 a 的线段切割 m 次能得到的最小平方和为 $f(a,m)$．由于均值不等式，当两数之和一定时，两数之差越小，两数的平方和也越小．所以，切割之后得到的线段长度越均匀，总的长度平方和也就越小．但是由于整数约束的存在，最均匀的情形就是得到了 $a \bmod (m+1)$ 条长度为 $\lceil a/(m+1) \rceil$ 的线段和 $m+1-(a \bmod (m+1))$ 条长度为 $\lfloor a/(m+1) \rfloor$ 的线段．因此，有如下表达式：

$$f(a,m)=(a \bmod (m+1))\left\lceil\frac{a}{m+1}\right\rceil^2+(m+1-(a \bmod (m+1)))\left\lfloor\frac{a}{m+1}\right\rfloor^2$$
$$=(a \bmod (m+1))\left(\left\lfloor\frac{a}{m+1}\right\rfloor+1\right)^2+(m+1-(a \bmod (m+1)))\left\lfloor\frac{a}{m+1}\right\rfloor^2.$$

第二步的等号成立，是因为 $\lceil a/(m+1)\rceil \neq \lfloor a/(m+1)\rfloor+1$ 当且仅当 $a \bmod (m+1)=0$.

可以证明，函数 $f(a,m)$ 是关于 m 的凸函数. 为此，需要将它延拓到 $m\in\mathbf{R}_+$ 的情形. 当 $\lfloor a/(m+1)\rfloor=q$ 时，有

$$f(a,m)=(a-(m+1)q)(q+1)^2+((m+1)(q+1)-a)q^2$$
$$=a(2q+1)-q(q+1)(m+1).$$

这是斜率为 $-q(q+1)$ 的直线. 因此，$f(a,m)$ 是分段线性函数，且斜率随着 m 的增加而增加. 这就说明了 $f(a,m)$ 是凸函数，它限制在整点上当然也是凸函数[9].

利用 $f(\cdot,\cdot)$，可以将所有线段总共切割 m 次得到的最小平方和写作如下最优化问题的价值函数：

$$v(m)=\min_{\{m_i\}}\sum_i f(a_i,m_i)\ \text{subject to}\sum_i m_i=m,\ m_i\in\mathbf{N}.$$

这是若干个凸函数的 卷积下确界→7.15.5@p836，所以也是凸函数. 如果题目要求的是 $v(m)$，那么，可以使用与之前的例题一致的方法求解，时间复杂度为 $O(n\log^2 L)$；但是，本题求的是满足 $v(m)\le V$ 的最小的 m. 利用 WQS 二分计算 $v(m)$ 再对 m 二分的方法是行不通的，它的复杂度达到了 $O(n\log^3 L)$. 就本题而言，有如下两种处理方法.

方法一：仍然二分斜率 k，但是二分的依据是对 $v(m)$ 上下界的估计.

传统的 WQS 二分的方法中，对于给定的斜率 k，可以计算出相应的最优的 m 的取值范围. 因为这些 $(m,v(m))$ 共线，所以这相当于确定了 $v(m)$ 的取值范围. 因此，可以直接二分斜率 k. 得到斜率 k 之后，可以利用直线方程

$$v(m)=h(k)+km$$

计算出最小的 m. 整体复杂度为 $O(n\log^2 L)$.

为了确定 $v(m)$ 的取值范围，需要确定 m 的取值范围. 一种做法是，在计算 $h(k)$ 时记录相应的最大最优解，利用它可以计算相应的 $v(m)$ 的下界；另一种做法是，利用 $h(k)-h(k-1)$ 得到相应的 m 的上界，进而得到相应的 $v(m)$ 的下界. 参考实现中，采用的是第二种做法，它不依赖于问题的具体结构，无需特别处理.

方法二：改写最优化问题，使得对偶问题的价值函数就是本问题的解.

本问题可以直接看作是如下最优化问题：

$$m(V)=\min_{\{m_i\}}\sum_i m_i\ \text{subject to}\sum_i f(a_i,m_i)\le V.$$

本文的分析仍然适用于这一问题. 故而，可以利用它的对偶问题求解所要求的 $m(V)$：

$$m(V)=\max_{\lambda}\sum_i \min_{m_i}(m_i-\lambda f(a_i,m_i))+\lambda V.$$

整体算法复杂度仍然是 $O(n\log^2 L)$ 的.

参考代码如下：

- 方法一　　代码仅做示意，为通过原题数据范围，需要 128 位整数，并调整二分初始区间为 $[0,10^{60}]$.

[9] 当然，$f(a,m)$ 和它限制在整点上得到的函数的凸包并不相同，因为 $f(a,m)$ 可能存在非整数位置处的极点. 这说明，并不能将定义域为实数的 $f(a,m)$ 直接用于本题的最优化问题中.

```cpp
#include <algorithm>
#include <iostream>
#include <tuple>
#include <type_traits>
#include <vector>

// Golden section search on integer domain (unimodal function)
template <typename T, typename F>
typename std::enable_if<
    std::is_integral<T>::value,
    std::pair<T, decltype(std::declval<F>()(std::declval<T>()))>>::type
golden_section_search(T ll, T rr, F func) {
  constexpr long double phi = 0.61803398874989484820458683 4L;
  T ml = ll + static_cast<T>((rr - ll) * (1 - phi));
  T mr = ll + static_cast<T>((rr - ll) * phi);
  auto fl = func(ml), fr = func(mr);
  while (ml < mr) {
    if (fl > fr) {
      rr = mr;
      mr = ml;
      fr = fl;
      ml = ll + static_cast<T>((rr - ll) * (1 - phi));
      fl = func(ml);
    } else {
      ll = ml;
      ml = mr;
      fl = fr;
      mr = ll + static_cast<T>((rr - ll) * phi);
      fr = func(mr);
    }
  }
  T best_x = ll;
  auto best_val = func(ll);
  for (T i = ll + 1; i <= rr; ++i) {
    auto val = func(i);
    if (val > best_val) {
      best_val = val;
      best_x = i;
    }
  }
  return {best_x, best_val};
}

int main() {
  int n;
  std::cin >> n;
  std::vector<int> a(n + 1);
  for (int i = 1; i <= n; ++i) std::cin >> a[i];
  for (int i = n; i >= 1; --i) a[i] -= a[i - 1];
  long long v;
  std::cin >> v;
  // Cost of adding M more teleporters to a segment of length LEN.
```

```
  auto f = [&](int len, int m) -> long long {
    long long rem = len % (m + 1);
    int q = len / (m + 1);
    return (m + 1 - rem) * q * q + rem * (q + 1) * (q + 1);
  };
  // Calculate h(k) = min_x f(x) - k * g(x).
  auto calc = [&](long long k) -> long long {
    long long res = 0;
    for (int i = 1; i <= n; ++i) {
      res += -golden_section_search(0, a[i], [&](int m) -> long long {
                return -f(a[i], m) + m * k;
              }).second;
    }
    return res;
  };
  // Find the smallest k such that h(k) + k * m <= v.
  long long ll = -(1LL << 30), rr = 0, ti = 0;
  while (ll <= rr) {
    auto mm = ll + (rr - ll) / 2;
    auto fi = calc(mm);
    auto ub = fi - calc(mm + 1);
    if (fi + ub * mm <= v) {
      ti = mm;
      rr = mm - 1;
    } else {
      ll = mm + 1;
    }
  }
  std::cout << (int)((calc(ti) - v - 1 - ti) / (-ti)) << std::endl;
  return 0;
}

```

- 方法二　　代码仅做示意，由于浮点数精度问题无法通过原题数据范围.

```
#include <algorithm>
#include <cmath>
#include <iostream>
#include <tuple>
#include <type_traits>
#include <vector>

// Golden section search on integer domain (unimodal function)
template <typename T, typename F>
typename std::enable_if<
    std::is_integral<T>::value,
    std::pair<T, decltype(std::declval<F>()(std::declval<T>()))>>::type
golden_section_search(T ll, T rr, F func) {
  constexpr long double phi = 0.6180339887498948482045868343L;
  T ml = ll + static_cast<T>((rr - ll) * (1 - phi));
  T mr = ll + static_cast<T>((rr - ll) * phi);
  auto fl = func(ml), fr = func(mr);
  while (ml < mr) {
```

```
    if (fl > fr) {
      rr = mr;
      mr = ml;
      fr = fl;
      ml = ll + static_cast<T>((rr - ll) * (1 - phi));
      fl = func(ml);
    } else {
      ll = ml;
      ml = mr;
      fl = fr;
      mr = ll + static_cast<T>((rr - ll) * phi);
      fr = func(mr);
    }
  }
  T best_x = ll;
  auto best_val = func(ll);
  for (T i = ll + 1; i <= rr; ++i) {
    auto val = func(i);
    if (val > best_val) {
      best_val = val;
      best_x = i;
    }
  }
  return {best_x, best_val};
}

// Golden section search on floating-point domain (unimodal function)
template <typename T, typename F>
typename std::enable_if<
    std::is_floating_point<T>::value,
    std::pair<T, decltype(std::declval<F>()(std::declval<T>()))>>::type
golden_section_search(T ll, T rr, F func, T eps) {
  constexpr long double phi = 0.61803398874989484820458683 4L;
  T ml = ll + (rr - ll) * (1 - phi);
  T mr = ll + (rr - ll) * phi;
  auto fl = func(ml), fr = func(mr);
  while ((rr - ll) > eps) {
    if (fl > fr) {
      rr = mr;
      mr = ml;
      fr = fl;
      ml = ll + (rr - ll) * (1 - phi);
      fl = func(ml);
    } else {
      ll = ml;
      ml = mr;
      fl = fr;
      mr = ll + (rr - ll) * phi;
      fr = func(mr);
    }
  }
  T mid = (ll + rr) / 2;
```

```
  return {mid, func(mid)};
}

int main() {
  int n;
  std::cin >> n;
  std::vector<int> a(n + 1);
  for (int i = 1; i <= n; ++i) std::cin >> a[i];
  for (int i = n; i >= 1; --i) a[i] -= a[i - 1];
  long long v;
  std::cin >> v;
  // Cost of adding M more teleporters to a segment of length LEN.
  auto f = [&](int len, int m) -> long long {
    long long rem = len % (m + 1);
    int q = len / (m + 1);
    return (m + 1 - rem) * q * q + rem * (q + 1) * (q + 1);
  };
  // Calculate h(k) = min_x f(x) - k * g(x).
  auto calc = [&](long double k) -> long double {
    long double res = 0;
    for (int i = 1; i <= n; ++i) {
      res += -golden_section_search(0, a[i], [&](int m) -> long double
{
                return -m + k * f(a[i], m);
              }).second;
    }
    return res;
  };
  // Solve the dual problem.
  auto res =
      golden_section_search(
          -1.0l, 0.0l,
          [&](long double k) -> long double { return calc(k) + k *
v; }, 1e-12l)
          .second;
  std::cout << (int)ceill(res) << std::endl;
  return 0;
}

```

习题

最后，列举一些可以通过 WQS 二分解决的题目，以供练习：

- Luogu P1484 种树[15]
- Luogu P1792 [国家集训队] 种树[16]
- Luogu P2619 [国家集训队] Tree I[17]
- Luogu P3620 [APIO/CTSC2007] 数据备份[18]
- Luogu P4072 [SDOI2016] 征途[19]
- Luogu P4383 [八省联考 2018] 林克卡特树[20]
- Luogu P4983 忘情[21]
- Luogu P5308 [COCI 2018/2019 #4] Akvizna[22]

- Luogu P5633 最小度限制生成树[23]
- Luogu P5896 [IOI 2016] aliens[24]
- Luogu P6246 [IOI 2000] 邮局 加强版 加强版[25]
- AtCoder Beginner Contest 218 H - Red and Blue Lamps[26]
- AtCoder Beginner Contest 305 Ex - Shojin[27]
- AtCoder Regular Contest 164 E - Segment-Tree Optimization[28]
- Codeforces 125 E. MST Company[29]
- Codeforces 321 E. Ciel and Gondolas[30]
- Codeforces 739 E. Gosha is hunting[31]
- Codeforces 802 O. April Fools' Problem (hard)[32]
- Codeforces 958 E2. Guard Duty (medium)[33]
- Codeforces 1279 F. New Year and Handle Change[34]
- Codeforces 1661 F. Teleporters[35]
- Codeforces 1799 F. Halve or Subtract[36]
- 2019 Summer Petrozavodsk Camp H. Honorable Mention[37]

参考资料与注释

- 王钦石《浅析一类二分方法》[38]
- Theoretical grounds of lambda optimization by adamant - Codeforces blog[39]
- 严谨的 WQS 二分方法 by YeahPotato - 洛谷博客[40]
- 【学习笔记】WQS 二分详解及常见理解误区解释 by ikrvxt - CSDN 博客[41]
- Convex conjugate - Wikipedia[42]
- Fenchel–Moreau theorem - Wikipedia[43]
- Subderivative - Wikipedia[44]
- Boyd, Stephen P., and Lieven Vandenberghe. Convex optimization. Cambridge university press, 2004.[45]
- Papadimitriou, Christos H., and Kenneth Steiglitz. Combinatorial optimization: algorithms and complexity. Courier Corporation, 1998.
- Conforti, Michele, Gérard Cornuéjols, and Giacomo Zambelli. Integer programming. Springer International Publishing, 2014.
- Schrijver, Alexander. Combinatorial optimization: polyhedra and efficiency. Vol. 24, no. 2. Berlin: Springer, 2003.

[1] 这份讲义 

[2] Lagrange 乘子法 

[3] 对偶问题 

[4] 下半连续 

[5] 超平面分离定理 

[6] 包络定理 

[7] 方法 

[8] [IOI 2005] Riv 河流 

[9] Luogu P1484 种树 

[10] Luogu P2619 [国家集训队] Tree I 

[11] Wikipedia 页面 

[12] Luogu P6246 [IOI 2000] 邮局 加强版 加强版 

[13] Codeforces 739 E. Gosha is hunting 

[14] Codeforces 1661 F. Teleporters 

[15] Luogu P1484 种树 

[16] Luogu P1792 [国家集训队] 种树 

[17] Luogu P2619 [国家集训队] Tree I 

[18] Luogu P3620 [APIO/CTSC2007] 数据备份 

[19] Luogu P4072 [SDOI2016] 征途 

[20] Luogu P4383 [八省联考 2018] 林克卡特树 

[21] Luogu P4983 忘情 

[22] Luogu P5308 [COCI 2018/2019 #4] Akvizna 
[23] Luogu P5633 最小度限制生成树 
[24] Luogu P5896 [IOI 2016] aliens 
[25] Luogu P6246 [IOI 2000] 邮局 加强版 加强版 
[26] AtCoder Beginner Contest 218 H - Red and Blue Lamps 
[27] AtCoder Beginner Contest 305 Ex - Shojin 
[28] AtCoder Regular Contest 164 E - Segment-Tree Optimization 
[29] Codeforces 125 E. MST Company 
[30] Codeforces 321 E. Ciel and Gondolas 
[31] Codeforces 739 E. Gosha is hunting 
[32] Codeforces 802 O. April Fools' Problem (hard) 
[33] Codeforces 958 E2. Guard Duty (medium) 
[34] Codeforces 1279 F. New Year and Handle Change 
[35] Codeforces 1661 F. Teleporters 
[36] Codeforces 1799 F. Halve or Subtract 
[37] 2019 Summer Petrozavodsk Camp H. Honorable Mention 
[38] 王钦石《浅析一类二分方法》 
[39] Theoretical grounds of lambda optimization by adamant - Codeforces blog 
[40] 严谨的 WQS 二分方法 by YeahPotato - 洛谷博客 
[41] 【学习笔记】WQS 二分详解及常见理解误区解释 by ikrvxt - CSDN 博客 
[42] Convex conjugate - Wikipedia 
[43] Fenchel–Moreau theorem - Wikipedia 
[44] Subderivative - Wikipedia 
[45] Boyd, Stephen P., and Lieven Vandenberghe. Convex optimization. Cambridge university press, 2004. 

7.15.7 状态设计优化

Authors: Marcythm, partychicken, Xeonacid, hhc0001

概述

优化 dp 时，不止可以从转移过程入手，加速转移．有时，也可以从状态定义入手，通过改变设计状态的方式实现复杂度上的优化．

令人比较头疼的是，这类优化大多不具有通用性，即不能很套路地应用于多个题目中．因此，下文将从具体例题出发，力求提供思路上的启发，希望可以对读者有一定帮助．

例 1

题面

给定两个长度分别为 n, m 且仅由小写字母构成的字符串 A, B，求 A, B 的最长公共子序列．($n \leq 10^6, m \leq 10^3$)

朴素的解法

您一眼秒了它，这不是板子吗?

定义状态 $f_{i,j}$ 为 A 的前 i 位与 B 的前 j 位最长公共子序列，则有

$$f_{i,j} = \begin{cases} \max(f_{i-1,j}, f_{i,j-1}), A_i \neq B_j \\ f_{i-1,j-1} + 1 \qquad\quad , A_i = B_j \end{cases}$$

上述做法的时间复杂度 $O(nm)$，无法通过本题．

更优的解法

我们仔细一想，发现了一个性质：最终答案不会超过 m．

我们又仔细一想，发现 LCS 满足贪心的性质．

更改状态定义 $f_{i,j}$ 为与 B 前 i 位的最长公共子序列长度为 j 的 A 的最短前缀长度（即将朴素做法的答案与第一维状态对调）

可以通过预处理 A 的每一位的下一个 $a, b, \cdots, z$ 的出现位置进行 $O(1)$ 的顺推转移.

复杂度 $O(m^2 + 26n)$，可以通过本题.

例 2

题面

给定一个 n 个点的无权有向图，判断该图是否存在哈密顿回路. $(2 \leq n \leq 20)$

朴素的解法

看到数据范围，我们考虑状压.

设 $f_{s,i}$ 表示从点 1 出发，仅经过点集 s 中的点能否到达点 i. 记 g 为原图的邻接矩阵. 则有

$$f_{s,i} = \bigvee_{j \in s, j \neq i} f_{s \setminus \{i\}, j} \wedge g_{j,i} (i \in s)$$

时间复杂度 $O(n^2 \times 2^n)$，写得好看或许能过，但是并不优美.

更优的解法

上面的状态设计中，每个 dp 值只代表一个 `bool` 值，这让我们觉得有些浪费.

我们可以考虑对于每个状态 s 将 $f_{s,1}, f_{s,2}, \cdots, f_{s,n}$ 压成一个 `int`，发现我们可以将邻接矩阵同样压缩后进行 $O(1)$ 转移.

时间复杂度 $O(n^2/w \times 2^n)$, 可以通过这道题，其中 w 为 `int` 的位数.

例 3

题面

常规的背包问题. n 为物品数量, m 为背包容量, v_i, w_i 为第 i 个物品的体积、价值, $1 \leq n \leq 10^3$, $1 \leq m, v_i \leq 10^{18}$，$1 \leq \sum w_i \leq 10^3$.

朴素的解法

这是一个模板背包题.

定义状态 $f_{i,j}$ 为选了前 i 个物品，目前背包里塞了 j 的容量的最大价值和.

易得 $f_{i,j} = \max(f_{i-1,j}, f_{i-1,j-v_i} + w_i)$.

$v_i \leq 10^{18}$，无法通过此题.

更优的解法

交换答案和状态的第二维，设 $f_{i,j}$ 为选了前 i 个物品，目前背包里的物品 **的价值为** j 的最小体积和.

依然，易得 $f_{i,j} = \min(f_{i-1,j}, f_{i-1,j-w_i} + v_i)$.

注意状态的第二维改变之后转移也要一起改变.

时间复杂度 $O(n \sum w_i)$，可以通过此题.

第八章

字符串

8.1 字符串部分简介

字符串，就是由字符连接而成的序列.

常见的字符串问题包括字符串匹配问题、子串相关问题、前缀/后缀相关问题、回文串相关问题、子序列相关问题等.

8.2 字符串基础

Authors: lr1d, ouuan, qinggniq, i-Yirannn, minghu6

定义

字符集

一个 **字符集** Σ 是一个建立了 全序→9.20@p1951 关系的集合，也就是说，Σ 中的任意两个不同的元素 α 和 β 都可以比较大小，要么 $\alpha < \beta$，要么 $\beta < \alpha$. 字符集 Σ 中的元素称为字符.

字符串

一个 **字符串** S 是将 n $(n \geq 0)$ 个字符顺次排列形成的序列，n 称为 S 的长度，表示为 $|S|$. 特别地，$n = 0$ 时 S 不含任何字符，称为 **空串**，记作 ε.

如果字符串下标从 1 开始计算，S 的第 i 个字符表示为 $S[i]$;

如果字符串下标从 0 开始计算，S 的第 i 个字符表示为 $S[i-1]$.

子串

字符串 S 的 **子串** $S[i..j], i \leq j$，表示 S 串中从 i 到 j 这一段，也就是顺次排列 $S[i], S[i+1], ..., S[j]$ 形成的字符串.

有时也会用 $S[i..j]$，$i > j$ 来表示空串 ε.

子序列

字符串 S 的 **子序列** 是从 S 中将若干元素提取出来并不改变相对位置形成的序列，即 $S[p_1], S[p_2], ..., S[p_k]$，$1 \leq p_1 < p_2 < \cdots < p_k \leq |S|$，$k \geq 0$.

后缀

后缀 是指从某个位置 i 开始到整个串末尾结束的一个特殊子串. 字符串 S 的从 i 开头的后缀表示为 $Suffix(S, i)$，也就是 $Suffix(S, i) = S[i..|S|-1]$.

真后缀 指除了 S 本身的 S 的后缀.

举例来说，字符串 `abcabcd` 的所有后缀为 `{ε, d, cd, bcd, abcd, cabcd, bcabcd, abcabcd}`，而它的真后缀为 `{ε, d, cd, bcd, abcd, cabcd, bcabcd}`.

前缀

前缀 是指从串首开始到某个位置 i 结束的一个特殊子串. 字符串 S 的以 i 结尾的前缀表示为 $Prefix(S, i)$，也就是 $Prefix(S, i) = S[0..i]$.

真前缀 指除了 S 本身的 S 的前缀.

举例来说，字符串 `abcabcd` 的所有前缀为 `{ε, a, ab, abc, abca, abcab, abcabc, abcabcd}`，而它的真前缀为 `{ε, a, ab, abc, abca, abcab, abcabc}`.

字典序

以第 i 个字符作为第 i 关键字进行大小比较，空字符小于字符集内任何字符（即：$a < aa$）.

回文串

回文串 是正着写和倒着写相同的字符串，即满足 $\forall 1 \leq i \leq |\ s\ |, s[i] = s[|\ s\ | + 1 - i]$ 的 s.

汉明距离

汉明距离 是两个等长字符串之间的距离，它表示两个长度相同的字符串对应位字符不同的数量.

我们可以简单的认为对两个串进行异或运算，结果为 1 的数量就是两个串的汉明距离.

字符串的存储

- 使用 `char` 数组存储，用空字符 `\0` 表示字符串的结尾（C 风格字符串）.
- 使用 C++ 标准库提供的 `string` 类→4.3.5@p402.
- 字符串常量可以用字符串字面量（用双引号括起来的字符串）表示.

8.3 标准库

Authors: Frankaiyou, henrytbtrue, zymooll

C 标准库

C 标准库操作字符数组 `char[]/const char*`.

参见：fprintf[1]、fscanf[2]、空终止字节字符串[3]

- `printf("%s", s)`：用 `%s` 来输出一个字符串（字符数组）.
- `scanf("%s", &s)`：用 `%s` 来读入一个字符串（字符数组）.
- `sscanf(const char *__source, const char *__format, ...)`：从字符串 `__source` 里读取变量，比如 `sscanf(str,"%d",&a)`.
- `sprintf(char *__stream, const char *__format, ...)`：将 `__format` 字符串里的内容输出到 `__stream` 中，比如 `sprintf(str,"%d",i)`.
- `strlen(const char *str)`：返回从 `str[0]` 开始直到 `'\0'` 的字符数. 注意，未开启 O2 优化时，该操作写在循环条件中复杂度是 $\Theta(N)$ 的.
- `strcmp(const char *str1, const char *str2)`：按照字典序比较 `str1 str2` 若 `str1` 字典序小返回负值，两者一样返回 `0`，`str1` 字典序更大则返回正值. 请注意，不要简单的认为返回值只有 `0`、`1`、`-1` 三种，在不同平台下的返回值都遵循正负，但并非都是 `0`、`1`、`-1`.
- `strcpy(char *str, const char *src)`: 把 `src` 中的字符复制到 `str` 中，`str src` 均为字符数组头指针，返回值为 `str` 包含空终止符号 `'\0'`.
- `strncpy(char *str, const char *src, int cnt)`：复制至多 `cnt` 个字符到 `str` 中，若 `src` 终止而数量未达 `cnt` 则写入空字符到 `str` 直至写入总共 `cnt` 个字符.
- `strcat(char *str1, const char *str2)`: 将 `str2` 接到 `str1` 的结尾，用 `*str2` 替换 `str1` 末尾的 `'\0'` 返回 `str1`.
- `strstr(char *str1, const char *str2)`：若 `str2` 是 `str1` 的子串，则返回 `str2` 在 `str1` 的首次出现的地址；如果 `str2` 不是 `str1` 的子串，则返回 `NULL`.
- `strchr(const char *str, int c)`：找到在字符串 `str` 中第一次出现字符 `c` 的位置，并返回这个位置的地址. 如果未找到该字符则返回 `NULL`.
- `strrchr(const char *str, int c)`：找到在字符串 `str` 中最后一次出现字符 `c` 的位置，并返回这个位置的地址. 如果未找到该字符则返回 `NULL`.

C++ 标准库

C++ 标准库操作字符串对象 `std::string`→4.3.5@p402，同时也提供对字符数组的兼容.

参见：std::basic_string[4]、std::basic_string_view[5]

- 重载了赋值运算符 `+`，当 `+` 两边是 `string/char/char[]/const char*` 类型时，可以将这两个变量连接，返回连接后的字符串（`string`）.
- 赋值运算符 `=` 右侧可以是 `const string/string/const char*/char*`.
- 访问运算符 `[cur]` 返回 `cur` 位置的引用.
- 访问函数 `data()/c_str()` 返回一个 `const char*` 指针，内容与该 `string` 相同.
- 容量函数 `size()` 返回字符串字符个数.
- `find(ch, start = 0)` 查找并返回从 `start` 开始的字符 `ch` 的位置；`rfind(ch)` 从末尾开始，查找并返回第一个找到的字符 `ch` 的位置（皆从 `0` 开始）（如果查找不到，返回 `-1`）.
- `substr(start, len)` 可以从字符串的 `start`（从 `0` 开始）截取一个长度为 `len` 的字符串（缺省 `len` 时代码截取到字符串末尾）.
- `append(s)` 将 `s` 添加到字符串末尾.
- `append(s, pos, n)` 将字符串 `s` 中，从 `pos` 开始的 `n` 个字符连接到当前字符串结尾.
- `replace(pos, n, s)` 删除从 `pos` 开始的 `n` 个字符，然后在 `pos` 处插入串 `s`.
- `erase(pos, n)` 删除从 `pos` 开始的 `n` 个字符.
- `insert(pos, s)` 在 `pos` 位置插入字符串 `s`.
- `std::string` 重载了比较逻辑运算符，复杂度是 $\Theta(N)$ 的.

外部参考资料

[1] fprintf [2] fscanf [3] 空终止字节字符串

[4] std::basic_string [5] std::basic_string_view

8.4 字符串匹配

本页面将简述字符串匹配问题以及它的解法.

字符串匹配问题

定义

又称模式匹配 (pattern matching). 该问题可以概括为「给定字符串 S 和 T，在主串 S 中寻找子串 T」. 字符 T 称为模式串 (pattern).

类型

- 单串匹配：给定一个模式串和一个待匹配串，找出前者在后者中的所有位置.
- 多串匹配：给定多个模式串和一个待匹配串，找出这些模式串在后者中的所有位置.
  - 出现多个待匹配串时，将它们直接连起来便可作为一个待匹配串处理.
  - 可以直接当做单串匹配，但是效率不够高.
- 其他类型：例如匹配一个串的任意后缀，匹配多个串的任意后缀……

暴力做法

简称 BF (Brute Force) 算法. 该算法的基本思想是从主串 S 的第一个字符开始和模式串 T 的第一个字符进行比较，若相等，则继续比较二者的后续字符；否则，模式串 T 回退到第一个字符，重新和主串 S 的第二个字符进行比较. 如此往复，直到 S 或 T 中所有字符比较完毕.

实现

- C++

```cpp
/*
 * s: 待匹配的主串
 * t: 模式串
 * n: 主串的长度
 * m: 模式串的长度
 */
std::vector<int> match(char *s, char *t, int n, int m) {
  std::vector<int> ans;
  int i, j;
  for (i = 0; i < n - m + 1; i++) {
    for (j = 0; j < m; j++) {
      if (s[i + j] != t[j]) break;
    }
    if (j == m) ans.push_back(i);
  }
  return ans;
}
```

- Python

```python
def match(s, t, n, m):
    if m < 1:
        return []

    ans = []
    for i in range(0, n - m + 1):
        for j in range(0, m):
            if s[i + j] != t[j]:
                break
        else:
            ans.append(i)
    return ans
```

时间复杂度

设 n 为主串的长度，m 为模式串的长度．默认 $m \ll n$.

BF 算法匹配成功时，在最好情况下，只有一趟匹配成功，此趟比较次数为 m，而其余每趟不成功的匹配都发生在模式串的第一个字符，还需要 $n-m$ 次比较，总比较次数为 n，故时间复杂度为 $O(n)$；在最坏情况下，匹配成功的趟数为 $n-m+1$，每趟比较次数为 m，总比较次数为 $m(n-m+1)$，故时间复杂度为 $O(mn)$.

BF 算法匹配失败时，在最好情况下，每趟不成功的匹配都发生在模式串的第一个字符，BF 算法要执行 $n-m+1$ 次比较，时间复杂度为 $O(n)$；在最坏情况下，每趟不成功的匹配都发生在模式串的最后一个字符，BF 算法要执行 $m(n-m+1)$ 次比较，时间复杂度为 $O(mn)$.

如果模式串有至少两个不同的字符，则 BF 算法的平均时间复杂度为 $O(n)$．但是在 OI 题目中，给出的字符串一般都不是纯随机的.

Hash 的方法

参见：字符串哈希→8.5@p894

KMP 算法

参见：前缀函数与 KMP 算法→8.7@p917

8.5 字符串哈希

定义

我们定义一个把字符串映射到整数的函数 f，这个 f 称为是 Hash 函数.

我们希望这个函数 f 可以方便地帮我们判断两个字符串是否相等.

Hash 的思想

Hash 的核心思想在于，将输入映射到一个值域较小、可以方便比较的范围.

> ⚠ **Warning**
>
> 这里的「值域较小」在不同情况下意义不同.
>
> 在 哈希表→10.5@p1995 中，值域需要小到能够接受线性的空间与时间复杂度.
>
> 在字符串哈希中，值域需要小到能够快速比较（10^9、10^{18} 都是可以快速比较的）.
>
> 同时，为了降低哈希冲突率，值域也不能太小.

性质

具体来说，哈希函数最重要的性质可以概括为下面两条：

1. 在 Hash 函数值不一样的时候，两个字符串一定不一样；
2. 在 Hash 函数值一样的时候，两个字符串不一定一样（但有大概率一样，且我们当然希望它们总是一样的）.

   我们将 Hash 函数值一样但原字符串不一样的现象称为哈希碰撞.

解释

我们需要关注的是什么?

时间复杂度和 Hash 的准确率.

通常我们采用的是多项式 Hash 的方法，对于一个长度为 l 的字符串 s 来说，我们可以这样定义多项式 Hash 函数：$f(s)=\sum_{i=1}^{l} s[i]\times b^{l-i} \pmod M$. 例如，对于字符串 xyz，其哈希函数值为 xb^2+yb+z.

特别要说明的是，也有很多人使用的是另一种 Hash 函数的定义，即 $f(s)=\sum_{i=1}^{l} s[i]\times b^{i-1} \pmod M$，这种定义下，同样的字符串 xyz 的哈希值就变为了 $x+yb+zb^2$ 了.

显然，上面这两种哈希函数的定义函数都是可行的，但二者在之后会讲到的计算子串哈希值时所用的计算式是不同的，因此千万注意 **不要弄混了这两种不同的 Hash 方式**.

由于前者的 Hash 定义计算更简便、使用人数更多、且可以类比为一个 b 进制数来帮助理解，所以本文下面所将要讨论的都是使用 $f(s)=\sum_{i=1}^{l} s[i]\times b^{l-i} \pmod M$ 来定义的 Hash 函数.

还有，有时为了方便和扩大模数，我们在 C++ 中我们会使用 `unsigned long long` 来定义 Hash 函数的结果. 由于 C++ 的特性，我们相当于把模数 M 定为 2^{64}，也是一个不错的选择.

准确率会在后面讨论.

Hash 的错误率分析

Hash 冲突

Hash 冲突是指两个不同的字符串映射到相同的 Hash 值.

我们设 Hash 的取值空间（所有可能出现的字符串的数量）为 d，计算次数（要计算的字符串数量）为 n.

则 Hash 冲突的概率为：

$$p(n,d)=1-\frac{d!}{d^n(d-n)!}\approx 1-\exp(-\frac{n(n-1)}{2d})$$

证明

当 Hash 中每个值生成概率相同时，Hash 不冲突的概率为：

$$\overline{p}(n,d)=1\cdot\left(1-\frac{1}{d}\right)\cdot\left(1-\frac{2}{d}\right)\cdots\left(1-\frac{n-1}{d}\right)$$

化简得到：

$$\begin{aligned}\overline{p}(n,d)&=\frac{d}{d}\cdot\frac{d-1}{d}\cdot\frac{d-2}{d}\cdots\frac{d-n+1}{d}\\&=\frac{d\cdot(d-1)\cdot(d-2)\cdots(d-n+1)}{d^n}\\&=\frac{d!}{d^n(d-n)!}\end{aligned}$$

则 Hash 冲突的概率为：

$$p(n,d)=1-\frac{d!}{d^n(d-n)!}$$

这个公式还是太复杂了，我们进一步化简.

根据泰勒公式：

$$\exp(x)=\sum_{k=0}^{\infty}\frac{x^k}{k!}=1+x+\frac{x^2}{2}+\frac{x^3}{6}+\frac{x^4}{24}+\cdots$$

当 x 为一个极小值时，$\exp(x)$ 趋近于 $1+x$.

将它带入 Hash 不冲突的原始公式：

$$\overline{p}(n,d)\approx 1\cdot\exp(-\frac{1}{d})\cdot\exp(-\frac{2}{d})\cdots\exp(-\frac{n-1}{d})$$

化简：

$$\begin{aligned}\overline{p}(n,d)&\approx\exp(-\frac{1}{d}-\frac{2}{d}-\cdots-\frac{n-1}{d})\\&=\exp(-\frac{n(n-1)}{2d})\end{aligned}$$

则 Hash 冲突的概率为：

$$p(n,d)\approx 1-\exp(-\frac{n(n-1)}{2d})$$

卡大模数 Hash

注意到这个公式：

$$p(n,d)\approx 1-\exp(-\frac{n(n-1)}{2d})$$

为了卡掉 Hash，我们要满足以下条件：

1. d 要大于模数.
2. $1-p(d,n)$ 要尽量小.

举个例子：

若字符集为 **大小写字母和数字**，模数为 10^9+7 时：

$\log_{62} 10^9+7 \approx 6$

$p(10^6, 62^6) \approx 0.9$

所以对于这个范围，我们随机生成 10^6 个长度为 6 的字符串，它们 Hash 值相同的概率高达 90%.

卡自然溢出 Hash

这种 Hash 由于模数太大，用上面的方法卡不了，所以我们需要另一种方法.

首先，这种 Hash 是形如 $f(s)=\sum_{i=1}^{l} s[i] \times b^{l-i}$，我们根据 b 来分类讨论.

b 为偶数

此时 $f(s)=s_1 \cdot b^l + s_2 \cdot b^{l-1} + \cdots + s_l \cdot b \pmod M$，其中 M 为 2^{64}.

容易发现若 $l \geq 64$，$s_i \cdot b^l \equiv 0 \pmod M$.

所以我们只要构造形如：

`aaa...a`

`baa...a`

且长度大于 64 的字符串就能冲突.

b 为奇数

定义 $!s_i$ 为把 s_i 中所有字符反转.

例：

$s_i = abaab$

$!s_i = babba$

即把 `a` 变成 `b`，把 `b` 变成 `a`.

再定义 $hash_i$ 为 s_i 的 Hash 值，$!hash_i$ 为 $!s_i$ 的 Hash 值.

不断构造 $s_i = s_{i-1} + !s_{i-1}$.

s_{12} 和 $!s_{12}$ 就是我们要的两个字符串.

推导

首先，有：

$$hash_i = hash_{i-1} \cdot base^{2^{i-2}} + !hash_{i-1}$$

$$!hash_i = !hash_{i-1} \cdot base^{2^{i-2}} + hash_{i-1}$$

尝试相减：

$$\begin{aligned} & hash_i - !hash_i \\ =\ & hash_{i-1} \cdot base^{2^{i-2}} + !hash_{i-1} - (!hash_{i-1} \cdot base^{2^{i-2}} + hash_{i-1}) \\ =\ & (hash_{i-1} - !hash_{i-1}) \cdot (base^{2^{i-2}} - 1) \end{aligned}$$

发现出现了 2^i，但是原式太复杂，尝试换元：

设：

$$f_i = hash_i - !hash_i$$

$$g_i = base^{2^{i-2}} - 1$$

根据原式得：

$$\begin{aligned} f_i &= f_{i-1} \cdot g_i \\ &= f_1 \cdot g_1 \cdot g_2 \cdots g_{i-1} \end{aligned}$$

因为 $base^{2^{i-2}}$ 一定是奇数，所以 g_i 一定是偶数.

所以:

$$2^{i-1} \mid f_i$$

但这样太大了，$i-1 \geq 64$ 才能卡掉，继续化简:

$$g_i = base^{2^{i-2}} - 1 = (base^{2^{i-3}} - 1) \cdot (base^{2^{i-3}} + 1)$$

即 g_i 为 $g_{i-1} \cdot c$ $(c \equiv 0 \pmod 2)$ 的形式.

所以 $2 \mid s_1$，$4 \mid s_2$，……，即

$$\begin{aligned} 2^i &\mid g_i \\ 2^1 \cdot 2^2 \cdot 2^3 \cdots 2^{i-1} &\mid f_i \\ 2^{i(i-1)/2} &\mid f_i \end{aligned}$$

即当 $i = 12$ 时就可以使 $2^{64} \mid hash_i - !hash_i$ 达到要求.

例题

例题：BZOJ 3097 Hash Killer I[1]

给定一个用 **自然溢出** 实现的 Hash，要求构造一个字符串来卡掉它.

例题：BZOJ 3097 Hash Killer II[2]

给定一个用 **大模数** 实现的 Hash，要求构造一个字符串来卡掉它.

例题：洛谷 U461211 字符串 Hash（数据加强）[3]

给定 n 个字符串，判断不同的字符串有多少个.

Hash 的改进

多值 Hash

看了上面这么多的卡法，当然也有解决办法.

多值 Hash，就是有多个 Hash 函数，每个 Hash 函数的模数不一样，这样就能解决 Hash 冲突的问题.

判断时只要有其中一个的 Hash 值不同，就认为两个字符串不同，若 Hash 值都相同，则认为两个字符串相同.

一般来说，双值 Hash 就够用了.

多次询问子串哈希

单次计算一个字符串的哈希值复杂度是 $O(n)$，其中 n 为串长，与暴力匹配没有区别，如果需要多次询问一个字符串的子串的哈希值，每次重新计算效率非常低下.

一般采取的方法是对整个字符串先预处理出每个前缀的哈希值，将哈希值看成一个 b 进制的数对 M 取模的结果，这样的话每次就能快速求出子串的哈希了:

令 $f_i(s)$ 表示 $f(s[1..i])$，即原串长度为 i 的前缀的哈希值，那么按照定义有 $f_i(s) = s[1] \cdot b^{i-1} + s[2] \cdot b^{i-2} + ... + s[i-1] \cdot b + s[i]$

现在，我们想要用类似前缀和的方式快速求出 $f(s[l..r])$，按照定义有字符串 $s[l..r]$ 的哈希值为 $f(s[l..r]) = s[l] \cdot b^{r-l} + s[l+1] \cdot b^{r-l-1} + ... + s[r-1] \cdot b + s[r]$

对比观察上述两个式子，我们发现 $f(s[l..r]) = f_r(s) - f_{l-1}(s) \times b^{r-l+1}$ 成立（可以手动代入验证一下），因此我们用这个式子就可以快速得到子串的哈希值．其中 b^{r-l+1} 可以 $O(n)$ 的预处理出来然后 $O(1)$ 的回答每次询问（当然也可以快速幂 $O(\log n)$ 的回答每次询问）．

实现

模数 Hash：

注：效率较低，实际使用中不推荐．

- C++

```cpp
using std::string;

constexpr int M = 1e9 + 7;
constexpr int B = 233;

using ll = long long;

int get_hash(const string& s) {
  int res = 0;
  for (int i = 0; i < s.size(); ++i) {
    res = ((ll)res * B + s[i]) % M;
  }
  return res;
}

bool cmp(const string& s, const string& t) {
  return get_hash(s) == get_hash(t);
}
```

- Python

```python
M = int(1e9 + 7)
B = 233


def get_hash(s):
    res = 0
    for char in s:
        res = (res * B + ord(char)) % M
    return res


def cmp(s, t):
    return get_hash(s) == get_hash(t)
```

双值 Hash：

- C++

```cpp
using ull = unsigned long long;
ull base = 131;
ull mod1 = 212370440130137957, mod2 = 1e9 + 7;

```

```cpp
ull get_hash1(std::string s) {
  int len = s.size();
  ull ans = 0;
  for (int i = 0; i < len; i++) ans = (ans * base + (ull)s[i]) % mod1;
  return ans;
}

ull get_hash2(std::string s) {
  int len = s.size();
  ull ans = 0;
  for (int i = 0; i < len; i++) ans = (ans * base + (ull)s[i]) % mod2;
  return ans;
}

bool cmp(const std::string s, const std::string t) {
  bool f1 = get_hash1(s) != get_hash1(t);
  bool f2 = get_hash2(s) != get_hash2(t);
  return f1 || f2;
}
```

- Python

```python
def get_hash1(s: str) -> int:
    base = 131
    mod1 = 212370440130137957
    ans = 0
    for char in s:
        ans = (ans * base + ord(char)) % mod1
    return ans


def get_hash2(s: str) -> int:
    base = 131
    mod2 = 1000000007
    ans = 0
    for char in s:
        ans = (ans * base + ord(char)) % mod2
    return ans


def cmp(s: str, t: str) -> bool:
    f1 = get_hash1(s) != get_hash1(t)
    f2 = get_hash2(s) != get_hash2(t)
    return f1 or f2
```

Hash 的应用

字符串匹配

求出模式串的哈希值后，求出文本串每个长度为模式串长度的子串的哈希值，分别与模式串的哈希值比较即可.

允许 k 次失配的字符串匹配

问题：给定长为 n 的源串 s，以及长度为 m 的模式串 p，要求查找源串中有多少子串与模式串匹配. s' 与 s 匹配，当且仅当 s' 与 s 长度相同，且最多有 k 个位置字符不同. 其中 $1 \le n, m \le 10^6$，$0 \le k \le 5$.

这道题无法使用 KMP 解决，但是可以通过哈希 + 二分来解决.

枚举所有可能匹配的子串，假设现在枚举的子串为 s'，通过哈希 + 二分可以快速找到 s' 与 p 第一个不同的位置. 之后将 s' 与 p 在这个失配位置及之前的部分删除掉，继续查找下一个失配位置. 这样的过程最多发生 k 次.

总的时间复杂度为 $O(m + kn \log_2 m)$.

最长回文子串

二分答案，判断是否可行时枚举回文中心（对称轴），哈希判断两侧是否相等. 需要分别预处理正着和倒着的哈希值. 时间复杂度 $O(n \log n)$.

这个问题可以使用 manacher 算法→8.16@p1049 在 $O(n)$ 的时间内解决.

通过哈希同样可以 $O(n)$ 解决这个问题，具体方法就是记 R_i 表示以 i 作为结尾的最长回文的长度，那么答案就是 $\max_{i=1}^{n} R_i$. 考虑到 $R_i \le R_{i-1} + 2$，因此我们只需要暴力从 $R_{i-1} + 2$ 开始递减，直到找到第一个回文即可. 记变量 z 表示当前枚举的 R_i，初始时为 0，则 z 在每次 i 增大的时候都会增大 2，之后每次暴力循环都会减少 1，故暴力循环最多发生 $2n$ 次，总的时间复杂度为 $O(n)$.

最长公共子字符串

问题：给定 m 个总长不超过 n 的非空字符串，查找所有字符串的最长公共子字符串，如果有多个，任意输出其中一个. 其中 $1 \le m, n \le 10^6$.

很显然如果存在长度为 k 的最长公共子字符串，那么 $k-1$ 的公共子字符串也必定存在. 因此我们可以二分最长公共子字符串的长度. 假设现在的长度为 k，`check(k)` 的逻辑为我们将所有字符串的长度为 k 的子串分别进行哈希，将哈希值放入 n 个哈希表中存储. 之后求交集即可.

时间复杂度为 $O(m + n \log n)$.

确定字符串中不同子字符串的数量

问题：给定长为 n 的字符串，仅由小写英文字母组成，查找该字符串中不同子串的数量.

为了解决这个问题，我们遍历了所有长度为 $l = 1, \cdots, n$ 的子串. 对于每个长度为 l，我们将其 Hash 值乘以相同的 b 的幂次方，并存入一个数组中. 数组中不同元素的数量等于字符串中长度不同的子串的数量，并此数字将添加到最终答案中.

为了方便起见，我们将使用 $h[i]$ 作为 Hash 的前缀字符，并定义 $h[0] = 0$.

参考代码

```cpp
int count_unique_substrings(string const& s) {
  int n = s.size();

  constexpr static int b = 31;
  constexpr static int m = 1e9 + 9;
  vector<long long> b_pow(n);
  b_pow[0] = 1;
  for (int i = 1; i < n; i++) b_pow[i] = (b_pow[i - 1] * b) % m;

  vector<long long> h(n + 1, 0);
  for (int i = 0; i < n; i++)
    h[i + 1] = (h[i] + (s[i] - 'a' + 1) * b_pow[i]) % m;

  int cnt = 0;
```

```
  for (int l = 1; l <= n; l++) {
    set<long long> hs;
    for (int i = 0; i <= n - l; i++) {
      long long cur_h = (h[i + l] + m - h[i]) % m;
      cur_h = (cur_h * b_pow[n - i - 1]) % m;
      hs.insert(cur_h);
    }
    cnt += hs.size();
  }
  return cnt;
}
```

例题

CF1200E Compress Words[4]

给你若干个字符串，答案串初始为空．第 i 步将第 i 个字符串加到答案串的后面，但是尽量地去掉重复部分（即去掉一个最长的、是原答案串的后缀、也是第 i 个串的前缀的字符串），求最后得到的字符串．

字符串个数不超过 10^5，总长不超过 10^6．

题解

每次需要求最长的、是原答案串的后缀、也是第 i 个串的前缀的字符串．枚举这个串的长度，哈希比较即可．

当然，这道题也可以使用 KMP 算法→8.7@p917 解决．

参考代码

```
#include <cassert>
#include <cstring>
#include <iostream>
#include <vector>
using namespace std;

constexpr int L = 1e6 + 5;
constexpr int HASH_CNT = 2;

int hashBase[HASH_CNT] = {29, 31};
int hashMod[HASH_CNT] = {int(1e9 + 9), 998244353};

struct StringWithHash {
  char s[L];
  int ls;
  int hsh[HASH_CNT][L];
  int pwMod[HASH_CNT][L];

  void init() {  // 初始化
    ls = 0;
```

```
    for (int i = 0; i < HASH_CNT; ++i) {
      hsh[i][0] = 0;
      pwMod[i][0] = 1;
    }
  }

  StringWithHash() { init(); }

  void extend(char c) {
    s[++ls] = c;                              // 记录字符数和每一个字符
    for (int i = 0; i < HASH_CNT; ++i) {  // 双哈希的预处理
      pwMod[i][ls] =
          1ll * pwMod[i][ls - 1] * hashBase[i] % hashMod[i];  // 得到b^ls
      hsh[i][ls] = (1ll * hsh[i][ls - 1] * hashBase[i] + c) %
hashMod[i];
    }
  }

  vector<int> getHash(int l, int r) {  // 得到哈希值
    vector<int> res(HASH_CNT, 0);
    for (int i = 0; i < HASH_CNT; ++i) {
      int t =
          (hsh[i][r] - 1ll * hsh[i][l - 1] * pwMod[i][r - l + 1])
% hashMod[i];
      t = (t + hashMod[i]) % hashMod[i];
      res[i] = t;
    }
    return res;
  }
};

bool equal(const vector<int> &h1, const vector<int> &h2) {
  assert(h1.size() == h2.size());
  for (unsigned i = 0; i < h1.size(); i++)
    if (h1[i] != h2[i]) return false;
  return true;
}

int n;
StringWithHash s, t;
char str[L];

void work() {
  int len = strlen(str);  // 取字符串长度
  t.init();
  for (int j = 0; j < len; ++j) t.extend(str[j]);
  int d = 0;
  for (int j = min(len, s.ls); j >= 1; --j) {
    if (equal(t.getHash(1, j), s.getHash(s.ls - j + 1, s.ls))) {  // 比较
哈希值
      d = j;
```

```
      break;
    }
  }
  for (int j = d; j < len; ++j) s.extend(str[j]);  // 更新答案数组
}

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  cin >> n;
  for (int i = 1; i <= n; ++i) {
    cin >> str;
    work();
  }
  cout << s.s + 1 << '\n';
  return 0;
}

```

本页面部分内容译自博文 строковый хеш[5] 与其英文翻译版 String Hashing[6]．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] 例题：BZOJ 3097 Hash Killer I 

[2] 例题：BZOJ 3097 Hash Killer II 

[3] 例题：洛谷 U461211 字符串 Hash（数据加强） 

[4] CF1200E Compress Words 

[5] строковый хеш 

[6] String Hashing 

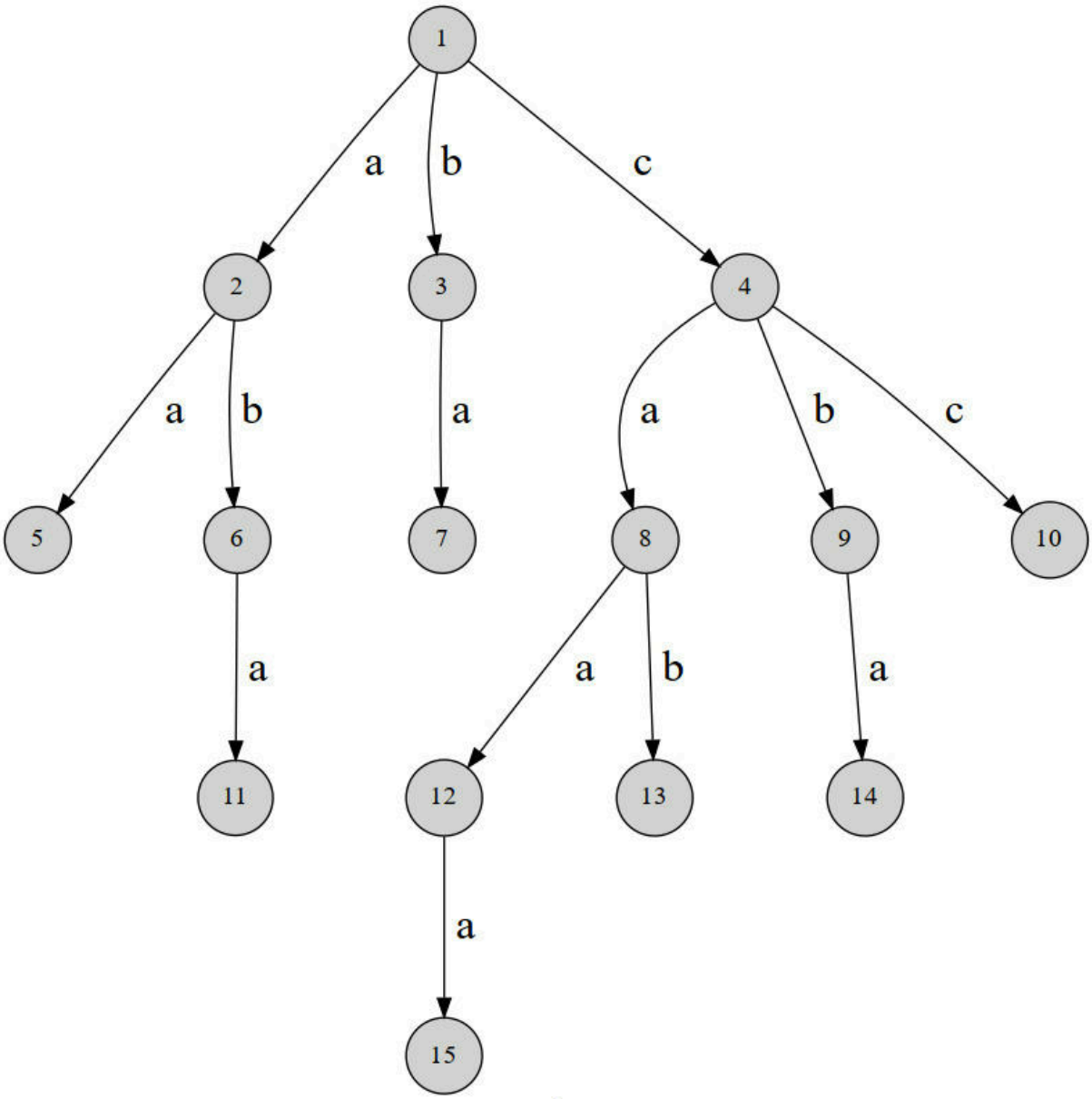
8.6 字典树 (Trie)

定义

字典树，英文名 trie．顾名思义，就是一个像字典一样的树．

引入

先放一张图：



Trie

可以发现，这棵字典树用边来代表字母，而从根结点到树上某一结点的路径就代表了一个字符串．举个例子，$1 \to 4 \to 8 \to 12$ 表示的就是字符串 `caa`.

trie 的结构非常好懂，我们用 $\delta(u, c)$ 表示结点 u 的 c 字符指向的下一个结点，或者说是结点 u 代表的字符串后面添加一个字符 c 形成的字符串的结点．（c 的取值范围和字符集大小有关，不一定是 $0 \sim 26$.）

有时需要标记插入进 trie 的是哪些字符串，每次插入完成时在这个字符串所代表的节点处打上标记即可．

实现

放一个结构体封装的模板：

- C++

```
struct trie {
  int nex[100000][26], cnt;
  bool exist[100000];  // 该结点结尾的字符串是否存在
```

```
4
5   void insert(char *s, int l) {  // 插入字符串
6     int p = 0;
7     for (int i = 0; i < l; i++) {
8       int c = s[i] - 'a';
9       if (!nex[p][c]) nex[p][c] = ++cnt;  // 如果没有，就添加结点
10      p = nex[p][c];
11    }
12    exist[p] = true;
13  }
14
15  bool find(char *s, int l) {  // 查找字符串
16    int p = 0;
17    for (int i = 0; i < l; i++) {
18      int c = s[i] - 'a';
19      if (!nex[p][c]) return 0;
20      p = nex[p][c];
21    }
22    return exist[p];
23  }
24 };
```

- Python

```
1 class trie:
2     def __init__(self):
3         self.nex = [[0 for i in range(26)] for j in range(100000)]
4         self.cnt = 0
5         self.exist = [False] * 100000  # 该结点结尾的字符串是否存在
6
7     def insert(self, s):  # 插入字符串
8         p = 0
9         for i in s:
10            c = ord(i) - ord("a")
11            if not self.nex[p][c]:
12                self.cnt += 1
13                self.nex[p][c] = self.cnt  # 如果没有，就添加结点
14            p = self.nex[p][c]
15        self.exist[p] = True
16
17    def find(self, s):  # 查找字符串
18        p = 0
19        for i in s:
20            c = ord(i) - ord("a")
21            if not self.nex[p][c]:
22                return False
23            p = self.nex[p][c]
24        return self.exist[p]
```

- Java

```
1 public class Trie {
2     int[][] tree = new int[10000][26];
3     int cnt = 0;
4     boolean[] end = new boolean[10000];
```

```java

    public void insert(String word) {
        int p = 0;
        char[] chars = word.toCharArray();
        for (int i = 0; i < chars.length; i++) {
            int c = chars[i] - 'a';
            if (tree[p][c] == 0) {
                tree[p][c] = ++cnt;
            }
            p = tree[p][c];
        }
        end[p] = true;
    }

    public boolean find(String word) {
        int p = 0;
        char[] chars = word.toCharArray();
        for (int i = 0; i < chars.length; i++) {
            int c = chars[i] - 'a';
            if (tree[p][c] == 0) {
                return false;
            }
            p = tree[p][c];
        }
        return end[p];
    }
}
```

应用

检索字符串

字典树最基础的应用——查找一个字符串是否在「字典」中出现过.

于是他错误的点名开始了[1]

给你 n 个名字串，然后进行 m 次点名，每次你需要回答「名字不存在」、「第一次点到这个名字」、「已经点过这个名字」之一.

$1 \le n \le 10^4$，$1 \le m \le 10^5$，所有字符串长度不超过 50.

题解

对所有名字建 trie，再在 trie 中查询字符串是否存在、是否已经点过名，第一次点名时标记为点过名.

参考代码

```cpp
#include <cstdio>
using namespace std;
constexpr int N = 500010;

char s[N];
```

```cpp
int n, m, ch[N][26], tag[N], tot = 1;

int main() {
  scanf("%d", &n);

  for (int i = 1; i <= n; ++i) {
    scanf("%s", s + 1);
    int u = 1;
    for (int j = 1; s[j]; ++j) {
      int c = s[j] - 'a';
      // 如果这个节点的子节点中没有这个字符，添加上并将该字符的节点号记录为++tot
      if (!ch[u][c]) ch[u][c] = ++tot;
      u = ch[u][c];  // 往更深一层搜索
    }
    tag[u] = 1;  // 最后一个字符为节点 u 的名字未被访问到记录为 1
  }

  scanf("%d", &m);

  while (m--) {
    scanf("%s", s + 1);
    int u = 1;
    for (int j = 1; s[j]; ++j) {
      int c = s[j] - 'a';
      u = ch[u][c];
      if (!u) break;  // 不存在对应字符的出边说明名字不存在
    }
    if (tag[u] == 1) {
      tag[u] = 2;  // 最后一个字符为节点 u 的名字已经被访问
      puts("OK");
    } else if (tag[u] == 2)  // 已经被访问，重复访问
      puts("REPEAT");
    else
      puts("WRONG");
  }

  return 0;
}
```

AC 自动机

trie 是 AC 自动机→8.10@p951 的一部分.

维护异或极值

将数的二进制表示看做一个字符串，就可以建出字符集为 $\{0,1\}$ 的 trie 树.

BZOJ1954 最长异或路径[2]

给你一棵带边权的树，求 (u,v) 使得 u 到 v 的路径上的边权异或和最大，输出这个最大值. 这里的异或和指的是所有边权的异或.

点数不超过 10^5，边权在 $[0, 2^{31})$ 内.

题解

随便指定一个根 $root$，用 $T(u, v)$ 表示 u 和 v 之间的路径的边权异或和，那么 $T(u, v) = T(root, u) \oplus T(root, v)$，因为 LCA→11.6.5@p2646 以上的部分异或两次抵消了.

那么，如果将所有 $T(root, u)$ 插入到一棵 trie 中，就可以对每个 $T(root, u)$ 快速求出和它异或和最大的 $T(root, v)$:

从 trie 的根开始，如果能向和 $T(root, u)$ 的当前位不同的子树走，就向那边走，否则没有选择.

贪心的正确性：如果这么走，这一位为 1；如果不这么走，这一位就会为 0. 而高位是需要优先尽量大的.

参考代码

```cpp
#include <algorithm>
#include <iostream>
using namespace std;

constexpr int N = 100010;

int head[N], nxt[N << 1], to[N << 1], weight[N << 1], cnt;
int n, dis[N], ch[N << 5][2], tot = 1, ans;

void insert(int x) {
  for (int i = 30, u = 1; i >= 0; --i) {
    int c = ((x >> i) & 1);  // 二进制一位一位向下取
    if (!ch[u][c]) ch[u][c] = ++tot;
    u = ch[u][c];
  }
}

void get(int x) {
  int res = 0;
  for (int i = 30, u = 1; i >= 0; --i) {
    int c = ((x >> i) & 1);
    if (ch[u][c ^ 1]) {  // 如果能向和当前位不同的子树走，就向那边走
      u = ch[u][c ^ 1];
      res |= (1 << i);
    } else
      u = ch[u][c];
  }
  ans = max(ans, res);  // 更新答案
}

void add(int u, int v, int w) {  // 建边
  nxt[++cnt] = head[u];
  head[u] = cnt;
  to[cnt] = v;
```

```cpp
  weight[cnt] = w;
}

void dfs(int u, int fa) {
  insert(dis[u]);
  get(dis[u]);
  for (int i = head[u]; i; i = nxt[i]) {  // 遍历子节点
    int v = to[i];
    if (v == fa) continue;
    dis[v] = dis[u] ^ weight[i];
    dfs(v, u);
  }
}

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  cin >> n;

  for (int i = 1; i < n; ++i) {
    int u, v, w;
    cin >> u >> v >> w;
    add(u, v, w);  // 双向边
    add(v, u, w);
  }

  dfs(1, 0);

  cout << ans;
  return 0;
}

```

维护异或和

01-trie 是指字符集为 $\{0,1\}$ 的 trie. 01-trie 可以用来维护一些数字的异或和，支持修改（删除 + 重新插入），和全局加一（即：让其所维护所有数值递增 1，本质上是一种特殊的修改操作）.

如果要维护异或和，需要按值从低位到高位建立 trie.

一个约定：文中说当前节点 **往上** 指当前节点到根这条路径，当前节点 **往下** 指当前结点的子树.

插入 & 删除

如果要维护异或和，我们 **只需要** 知道某一位上 `0` 和 `1` 个数的 **奇偶性** 即可，也就是对于数字 `1` 来说，当且仅当这一位上数字 `1` 的个数为奇数时，这一位上的数字才是 `1`，请时刻记住这段文字：如果只是维护异或和，我们只需要知道某一位上 `1` 的数量即可，而不需要知道 trie 到底维护了哪些数字.

对于每一个节点，我们需要记录以下三个量：

- `ch[o][0/1]` 指节点 `o` 的两个儿子，`ch[o][0]` 指下一位是 `0`，同理 `ch[o][1]` 指下一位是 `1`.
- `w[o]` 指节点 `o` 到其父亲节点这条边上数值的数量（权值）. 每插入一个数字 `x`，`x` 二进制拆分后在 trie 上 路径的权值都会 `+1`.
- `xorv[o]` 指以 `o` 为根的子树维护的异或和.

具体维护结点的代码如下所示.

```
void maintain(int o) {
  w[o] = xorv[o] = 0;
  if (ch[o][0]) {
    w[o] += w[ch[o][0]];
    xorv[o] ^= xorv[ch[o][0]] << 1;
  }
  if (ch[o][1]) {
    w[o] += w[ch[o][1]];
    xorv[o] ^= (xorv[ch[o][1]] << 1) | (w[ch[o][1]] & 1);
  }
  // w[o] = w[o] & 1;
  // 只需知道奇偶性即可，不需要具体的值．当然这句话删掉也可以，因为上文就只利用了他的奇偶性．
}
```

插入和删除的代码非常相似.

需要注意的地方就是:

- 这里的 `MAXH` 指 trie 的深度，也就是强制让每一个叶子节点到根的距离为 `MAXH`．对于一些比较小的值，可能有时候不需要建立这么深（例如：如果插入数字 `4`，分解成二进制后为 `100`，从根开始插入 `001` 这三位即可），但是我们强制插入 `MAXH` 位．这样做的目的是为了便于全局 `+1` 时处理进位．例如：如果原数字是 `3`（`11`），递增之后变成 `4`（`100`），如果当初插入 `3` 时只插入了 `2` 位，那这里的进位就没了．
- 插入和删除，只需要修改叶子节点的 `w[]` 即可，在回溯的过程中一路维护即可.

实现

```
namespace trie {
constexpr int MAXH = 21;
int ch[_ * (MAXH + 1)][2], w[_ * (MAXH + 1)], xorv[_ * (MAXH + 1)];
int tot = 0;

int mknode() {
  ++tot;
  ch[tot][1] = ch[tot][0] = w[tot] = xorv[tot] = 0;
  return tot;
}

void maintain(int o) {
  w[o] = xorv[o] = 0;
  if (ch[o][0]) {
    w[o] += w[ch[o][0]];
    xorv[o] ^= xorv[ch[o][0]] << 1;
  }
  if (ch[o][1]) {
    w[o] += w[ch[o][1]];
    xorv[o] ^= (xorv[ch[o][1]] << 1) | (w[ch[o][1]] & 1);
  }
  w[o] = w[o] & 1;
}

void insert(int &o, int x, int dp) {
  if (!o) o = mknode();
```

```
  if (dp > MAXH) return (void)(w[o]++);
  insert(ch[o][x & 1], x >> 1, dp + 1);
  maintain(o);
}

void erase(int o, int x, int dp) {
  if (dp > 20) return (void)(w[o]--);
  erase(ch[o][x & 1], x >> 1, dp + 1);
  maintain(o);
}
}  // namespace trie
```

全局加一

所谓全局加一就是指，让这棵 trie 中所有的数值 `+1`.

形式化的讲，设 trie 中维护的数值有 $V_1, V_2, V_3 ... V_n$，全局加一后 其中维护的值应该变成 $V_1 + 1, V_2 + 1, V_3 + 1 ... V_n + 1$

```
void addall(int o) {
  swap(ch[o][0], ch[o][1]);
  if (ch[o][0]) addall(ch[o][0]);
  maintain(o);
}
```

过程

我们思考一下二进制意义下 `+1` 是如何操作的.

我们只需要从低位到高位开始找第一个出现的 `0`，把它变成 `1`，然后这个位置后面的 `1` 都变成 `0` 即可.

下面给出几个例子感受一下：（括号内的数字表示其对应的十进制数字）

```
1000(8)  + 1 = 1001(9)  ;
10011(19) + 1 = 10100(20) ;
11111(31) + 1 = 100000(32);
10101(21) + 1 = 10110(22) ;
1000000000111111(16447) + 1 = 1000000001000000(16448);
```

对应 trie 的操作，其实就是交换其左右儿子，顺着 **交换后** 的 `0` 边往下递归操作即可.

回顾一下 `w[o]` 的定义：`w[o]` 指节点 `o` 到其父亲节点这条边上数值的数量（权值）.

有没有感觉这个定义有点怪呢？如果在父亲结点存储到两个儿子的这条边的边权也许会更接近于习惯. 但是在这里，在交换左右儿子的时候，在儿子结点存储到父亲这条边的距离，显然更加方便.

01-trie 合并

指的是将上述的两个 01-trie 进行合并，同时合并维护的信息.

可能关于合并 trie 的文章比较少，其实合并 trie 和合并线段树的思路非常相似，可以搜索「合并线段树」来学习如何合并 trie.

其实合并 trie 非常简单，就是考虑一下我们有一个 `int merge(int a, int b)` 函数，这个函数传入两个 trie 树位于同一相对位置的结点编号，然后合并完成后返回合并完成的结点编号.

过程

考虑怎么实现?

分三种情况：

- 如果 a 没有这个位置上的结点，新合并的结点就是 b
- 如果 b 没有这个位置上的结点，新合并的结点就是 a
- 如果 a,b 都存在，那就把 b 的信息合并到 a 上，新合并的结点就是 a，然后递归操作处理 a 的左右儿子.

  提示：如果需要的合并是将 a，b 合并到一棵新树上，这里可以新建结点，然后合并到这个新结点上，这里的代码实现仅仅是将 b 的信息合并到 a 上.

实现

```
int merge(int a, int b) {
  if (!a) return b;  // 如果 a 没有这个位置上的结点，返回 b
  if (!b) return a;  // 如果 b 没有这个位置上的结点，返回 a
  /*
    如果 `a`, `b` 都存在，
    那就把 `b` 的信息合并到 `a` 上.
  */
  w[a] = w[a] + w[b];
  xorv[a] ^= xorv[b];
  /* 不要使用 maintain(),
    maintain() 是合并a的两个儿子的信息
    而这里需要 a b 两个节点进行信息合并
   */
  ch[a][0] = merge(ch[a][0], ch[b][0]);
  ch[a][1] = merge(ch[a][1], ch[b][1]);
  return a;
}
```

其实 trie 都可以合并，换句话说，trie 合并不仅仅限于 01-trie.

【luogu-P6018】【Ynoi2010】Fusion tree[3]

给你一棵 n 个结点的树，每个结点有权值. m 次操作. 需要支持以下操作.

- 将树上与一个节点 x 距离为 1 的节点上的权值 $+1$. 这里树上两点间的距离定义为从一点出发到另外一点的最短路径上边的条数.
- 在一个节点 x 上的权值 $-v$.
- 询问树上与一个节点 x 距离为 1 的所有节点上的权值的异或和. 对于 100% 的数据，满足 $1 \leq n \leq 5 \times 10^5$，$1 \leq m \leq 5 \times 10^5$，$0 \leq a_i \leq 10^5$，$1 \leq x \leq n$，$opt \in \{1, 2, 3\}$. 保证任意时刻每个节点的权值非负.

题解

每个结点建立一棵 trie 维护其儿子的权值，trie 应该支持全局加一. 可以使用在每一个结点上设置懒标记来标记儿子的权值的增加量.

参考代码

```
#include <iostream>
using namespace std;
constexpr int _ = 5e5 + 10;

```

```cpp
namespace trie {
constexpr int _n = _ * 25;
int rt[_];
int ch[_n][2];
int w[_n];  //`w[o]` 指节点 `o` 到其父亲节点这条边上数值的数量（权值）。
int xorv[_n];
int tot = 0;

void maintain(int o) {  // 维护w数组和xorv（权值的异或）数组
  w[o] = xorv[o] = 0;
  if (ch[o][0]) {
    w[o] += w[ch[o][0]];
    xorv[o] ^= xorv[ch[o][0]] << 1;
  }
  if (ch[o][1]) {
    w[o] += w[ch[o][1]];
    xorv[o] ^= (xorv[ch[o][1]] << 1) | (w[ch[o][1]] & 1);
  }
}

int mknode() {  // 创造一个新的节点
  ++tot;
  ch[tot][0] = ch[tot][1] = 0;
  w[tot] = 0;
  return tot;
}

void insert(int &o, int x, int dp) {  // x是权重，dp是深度
  if (!o) o = mknode();
  if (dp > 20) return (void)(w[o]++);
  insert(ch[o][x & 1], x >> 1, dp + 1);
  maintain(o);
}

void erase(int o, int x, int dp) {
  if (dp > 20) return (void)(w[o]--);
  erase(ch[o][x & 1], x >> 1, dp + 1);
  maintain(o);
}

void addall(int o) {  // 对所有节点+1即将所有节点的ch[o][1]和ch[o][0]交换
  swap(ch[o][1], ch[o][0]);
  if (ch[o][0]) addall(ch[o][0]);
  maintain(o);
}
}  // namespace trie

int head[_];

struct edges {
  int node;
```

```
  int nxt;
} edge[_ << 1];

int tot = 0;

void add(int u, int v) {
  edge[++tot].nxt = head[u];
  head[u] = tot;
  edge[tot].node = v;
}

int n, m;
int rt;
int lztar[_];
int fa[_];

void dfs0(int o, int f) {  // 得到fa数组
  fa[o] = f;
  for (int i = head[o]; i; i = edge[i].nxt) {  // 遍历子节点
    int node = edge[i].node;
    if (node == f) continue;
    dfs0(node, o);
  }
}

int V[_];

// 权值函数
int get(int x) { return (fa[x] == -1 ? 0 : lztar[fa[x]]) + V[x]; }

int main() {
  cin >> n >> m;
  for (int i = 1; i < n; i++) {
    int u, v;
    cin >> u >> v;
    add(u, v);  // 双向建边
    add(rt = v, u);
  }
  dfs0(rt, -1);  // rt是随机的一个点
  for (int i = 1; i <= n; i++) {
    cin >> V[i];
    if (fa[i] != -1) trie::insert(trie::rt[fa[i]], V[i], 0);
  }
  while (m--) {
    int opt, x;
    cin >> opt >> x;
    if (opt == 1) {
      lztar[x]++;
      if (x != rt) {
        if (fa[fa[x]] != -1) trie::erase(trie::rt[fa[fa[x]]],
get(fa[x]), 0);
```

```
106         V[fa[x]]++;
107         if (fa[fa[x]] != -1)
108           trie::insert(trie::rt[fa[fa[x]]], get(fa[x]), 0);  // 重新插入
109       }
110       trie::addall(trie::rt[x]);  // 对所有节点+1
111     } else if (opt == 2) {
112       int v;
113       cin >> v;
114       if (x != rt) trie::erase(trie::rt[fa[x]], get(x), 0);
115       V[x] -= v;
116       if (x != rt) trie::insert(trie::rt[fa[x]], get(x), 0);  // 重新插入
117     } else {
118       int res = 0;
119       res = trie::xorv[trie::rt[x]];
120       res ^= get(fa[x]);
121       cout << res << '\n';
122     }
123   }
124   return 0;
125 }
126
```

【luogu-P6623】【省选联考 2020 A 卷】树[4]

给定一棵 n 个结点的有根树 T，结点从 1 开始编号，根结点为 1 号结点，每个结点有一个正整数权值 v_i. 设 x 号结点的子树内（包含 x 自身）的所有结点编号为 $c_1, c_2, \ldots, c_k$，定义 x 的价值为：

$val(x) = (v_{c_1} + d(c_1, x)) \oplus (v_{c_2} + d(c_2, x)) \oplus \cdots \oplus (v_{c_k} + d(c_k, x))$ 其中 $d(x, y)$.

表示树上 x 号结点与 y 号结点间唯一简单路径所包含的边数，$d(x, x) = 0$. $\oplus$ 表示异或运算. 请你求出 $\sum_{i=1}^{n} val(i)$ 的结果.

题解

考虑每个结点对其所有祖先的贡献. 每个结点建立 trie，初始先只存这个结点的权值，然后从底向上合并每个儿子结点上的 trie，然后再全局加一，完成后统计答案.

参考代码

```
1  constexpr int _ = 526010;
2  int n;
3  int V[_];
4  int debug = 0;
5
6  namespace trie {
7  constexpr int MAXH = 21;
8  int ch[_ * (MAXH + 1)][2], w[_ * (MAXH + 1)], xorv[_ * (MAXH + 1)];
9  int tot = 0;
10
```

```
int mknode() {
  ++tot;
  ch[tot][1] = ch[tot][0] = w[tot] = xorv[tot] = 0;
  return tot;
}

void maintain(int o) {
  w[o] = xorv[o] = 0;
  if (ch[o][0]) {
    w[o] += w[ch[o][0]];
    xorv[o] ^= xorv[ch[o][0]] << 1;
  }
  if (ch[o][1]) {
    w[o] += w[ch[o][1]];
    xorv[o] ^= (xorv[ch[o][1]] << 1) | (w[ch[o][1]] & 1);
  }
  w[o] = w[o] & 1;
}

void insert(int &o, int x, int dp) {
  if (!o) o = mknode();
  if (dp > MAXH) return (void)(w[o]++);
  insert(ch[o][x & 1], x >> 1, dp + 1);
  maintain(o);
}

int merge(int a, int b) {
  if (!a) return b;
  if (!b) return a;
  w[a] = w[a] + w[b];
  xorv[a] ^= xorv[b];
  ch[a][0] = merge(ch[a][0], ch[b][0]);
  ch[a][1] = merge(ch[a][1], ch[b][1]);
  return a;
}

void addall(int o) {
  swap(ch[o][0], ch[o][1]);
  if (ch[o][0]) addall(ch[o][0]);
  maintain(o);
}
}  // namespace trie

int rt[_];
long long Ans = 0;
vector<int> E[_];

void dfs0(int o) {
  for (int i = 0; i < E[o].size(); i++) {
    int node = E[o][i];
    dfs0(node);
```

```
62      rt[o] = trie::merge(rt[o], rt[node]);
63    }
64    trie::addall(rt[o]);
65    trie::insert(rt[o], V[o], 0);
66    Ans += trie::xorv[rt[o]];
67  }
68
69  int main() {
70    n = read();
71    for (int i = 1; i <= n; i++) V[i] = read();
72    for (int i = 2; i <= n; i++) E[read()].push_back(i);
73    dfs0(1);
74    printf("%lld", Ans);
75    return 0;
76  }
```

可持久化字典树

参见 可持久化字典树→10.17.5@p2437.

外部参考资料

[1] 于是他错误的点名开始了 

[2] BZOJ1954 最长异或路径 

[3] 【luogu-P6018】【Ynoi2010】Fusion tree 

[4] 【luogu-P6623】【省选联考 2020 A 卷】树 

8.7 前缀函数与 KMP 算法

Authors: Ir1d, LeoJacob, Xeonacid, greyqz, StudyingFather, Marcythm, minghu6, Backl1ght

字符串前缀和后缀定义

关于字符串前缀、真前缀，后缀、真后缀的定义详见 字符串基础→8.2@p890

前缀函数

定义

给定一个长度为 n 的字符串 s，其 **前缀函数** 被定义为一个长度为 n 的数组 π. 其中 $\pi[i]$ 的定义是：

1. 如果子串 $s[0...i]$ 有一对相等的真前缀与真后缀：$s[0...k-1]$ 和 $s[i-(k-1)...i]$，那么 $\pi[i]$ 就是这个相等的真前缀（或者真后缀，因为它们相等）的长度，也就是 $\pi[i]=k$；
2. 如果不止有一对相等的，那么 $\pi[i]$ 就是其中最长的那一对的长度；
3. 如果没有相等的，那么 $\pi[i]=0$.

简单来说 $\pi[i]$ 就是，子串 $s[0...i]$ 最长的相等的真前缀与真后缀的长度.

用数学语言描述如下：

$$\pi[i]=\max_{k=0...i}\{k:s[0...k-1]=s[i-(k-1)...i]\}$$

特别地，规定 $\pi[0]=0$.

过程

举例来说，对于字符串 `abcabcd`，

$\pi[0]=0$，因为 `a` 没有真前缀和真后缀，根据规定为 0

$\pi[1]=0$，因为 `ab` 无相等的真前缀和真后缀

$\pi[2]=0$，因为 `abc` 无相等的真前缀和真后缀

$\pi[3]=1$，因为 `abca` 只有一对相等的真前缀和真后缀：`a`，长度为 1

$\pi[4]=2$，因为 `abcab` 相等的真前缀和真后缀只有 `ab`，长度为 2

$\pi[5]=3$，因为 `abcabc` 相等的真前缀和真后缀只有 `abc`，长度为 3

$\pi[6]=0$，因为 `abcabcd` 无相等的真前缀和真后缀

同理可以计算字符串 `aabaaab` 的前缀函数为 $[0,1,0,1,2,2,3]$.

计算前缀函数的朴素算法

过程

一个直接按照定义计算前缀函数的算法流程：

- 在一个循环中以 $i=1\to n-1$ 的顺序计算前缀函数 $\pi[i]$ 的值（$\pi[0]$ 被赋值为 0）.
- 为了计算当前的前缀函数值 $\pi[i]$，我们令变量 j 从最大的真前缀长度 i 开始尝试.
- 如果当前长度下真前缀和真后缀相等，则此时长度为 $\pi[i]$，否则令 j 自减 1，继续匹配，直到 $j=0$.
- 如果 $j=0$ 并且仍没有任何一次匹配，则置 $\pi[i]=0$ 并移至下一个下标 $i+1$.

实现

具体实现如下：

- C++

```cpp
// 注:
// string substr (size_t pos = 0, size_t len = npos) const;
vector<int> prefix_function(string s) {
  int n = (int)s.length();
  vector<int> pi(n);
  for (int i = 1; i < n; i++)
    for (int j = i; j >= 0; j--)
      if (s.substr(0, j) == s.substr(i - j + 1, j)) {
        pi[i] = j;
        break;
      }
  return pi;
}
```

- Python

```python
def prefix_function(s):
    n = len(s)
    pi = [0] * n
    for i in range(1, n):
        for j in range(i, -1, -1):
            if s[0:j] == s[i - j + 1 : i + 1]:
                pi[i] = j
                break
    return pi
```

- Java

```java
static int[] prefix_function(String s) {
    int n = s.length();
    int[] pi = new int[n];
    for (int i = 1; i < n; i++) {
        for (int j = i; j >= 0; j--) {
            if (s.substring(0, j).equals(s.substring(i - j + 1, i + 1))) {
                pi[i] = j;
                break;
            }
        }
    }
    return pi;
}
```

显见该算法的时间复杂度为 $O(n^3)$，具有很大的改进空间.

计算前缀函数的高效算法

第一个优化

第一个重要的观察是 **相邻的前缀函数值至多增加 1**.

参照下图所示，只需如此考虑：当取一个尽可能大的 $\pi[i+1]$ 时，必然要求新增的 $s[i+1]$ 也与之对应的字符匹配，即 $s[i+1]=s[\pi[i]]$, 此时 $\pi[i+1]=\pi[i]+1$.

$$\underbrace{\overbrace{s_0\ s_1\ s_2}^{\pi[i]=3}\ s_3}_{\pi[i+1]=4}\ \cdots\ \underbrace{\overbrace{s_{i-2}\ s_{i-1}\ s_i}^{\pi[i]=3}\ s_{i+1}}_{\pi[i+1]=4}$$

所以当移动到下一个位置时，前缀函数的值要么增加一，要么维持不变，要么减少.

实现

此时的改进的算法为:

- C++

```cpp
vector<int> prefix_function(string s) {
  int n = (int)s.length();
  vector<int> pi(n);
  for (int i = 1; i < n; i++)
    for (int j = pi[i - 1] + 1; j >= 0; j--)  // improved: j=i => j=pi[i-1]+1
      if (s.substr(0, j) == s.substr(i - j + 1, j)) {
        pi[i] = j;
        break;
      }
  return pi;
}
```

- Python

```python
def prefix_function(s):
    n = len(s)
    pi = [0] * n
```

```
    for i in range(1, n):
        for j in range(pi[i - 1] + 1, -1, -1):
            if s[0:j] == s[i - j + 1 : i + 1]:
                pi[i] = j
                break
    return pi
```

- Java

```
static int[] prefix_function(String s) {
    int n = s.length();
    int[] pi = new int[n];
    for (int i = 1; i < n; i++) {
        for (int j = pi[i - 1] + 1; j >= 0; j--) {
            if (s.substring(0, j).equals(s.substring(i - j + 1, i + 1))) {
                pi[i] = j;
                break;
            }
        }
    }
    return pi;
}
```

在这个初步改进的算法中，在计算每个 $\pi[i]$ 时，最好的情况是第一次字符串比较就完成了匹配，也就是说基础的字符串比较次数是 $n-1$ 次.

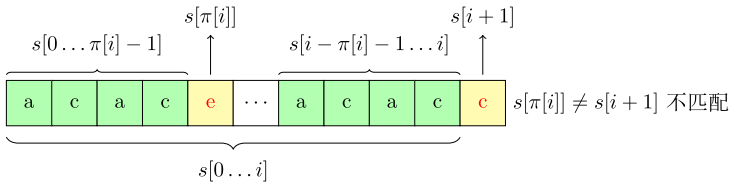
而由于存在 `j = pi[i-1]+1`（`pi[0]=0`）对于最大字符串比较次数的限制，可以看出每次只有在最好情况才会为字符串比较次数的上限积累 1，而每次超过一次的字符串比较消耗的是之后次数的增长空间.

由此我们可以得出字符串比较次数最多的一种情况：至少 1 次字符串比较次数的消耗和最多 $n-2$ 次比较次数的积累，此时字符串比较次数为 $n-1+n-2=2n-3$.

可见经过此次优化，计算前缀函数只需要进行 $O(n)$ 次字符串比较，总复杂度降为了 $O(n^2)$.

第二个优化

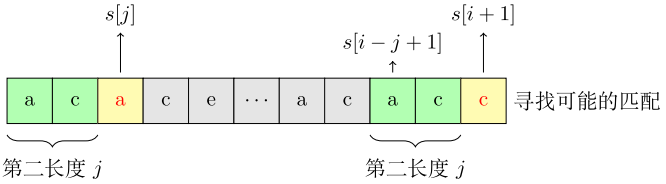
在第一个优化中，我们讨论了计算 $\pi[i+1]$ 时的最好情况：$s[i+1]=s[\pi[i]]$，此时 $\pi[i+1]=\pi[i]+1$. 现在让我们沿着这个思路走得更远一点：讨论当 $s[i+1]\neq s[\pi[i]]$ 时如何跳转.



如上图所示，失配时，我们希望找到对于子串 $s[0...i]$，仅次于 $\pi[i]$ 的第二长度 j，使得在位置 i 的前缀性质仍得以保持，也即 $s[0...j-1]=s[i-j+1...i]$：

$$\overbrace{\underbrace{s_0\ s_1}_{j}\ s_2\ s_3}^{\pi[i]}\ \dots\ \overbrace{s_{i-3}\ s_{i-2}\ \underbrace{s_{i-1}\ s_i}_{j}}^{\pi[i]}\ s_{i+1}$$

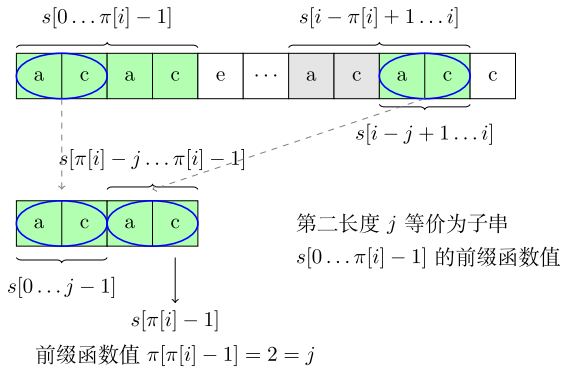
如果我们找到了这样的长度 j，那么仅需要再次比较 $s[i+1]$ 和 $s[j]$. 如果它们相等，那么就有 $\pi[i+1]=j+1$. 否则，我们需要找到子串 $s[0...i]$ 仅次于 j 的第二长度 $j^{(2)}$，使得前缀性质得以保持，如此反复，直到 $j=0$. 如果 $s[i+1]\neq s[0]$，则 $\pi[i+1]=0$. 第二次比较的示意图如下所示



观察上图可以发现，因为 $s[0...\pi[i]-1]=s[i-\pi[i]+1...i]$，所以对于 $s[0...i]$ 的第二长度 j，有这样的性质：

$$s[0...j-1]=s[i-j+1...i]=s[\pi[i]-j...\pi[i]-1]$$

该公式的示意图如下所示：



也就是说 j 等价于子串 $s[\pi[i]-1]$ 的前缀函数值，对应于上图下半部分，即 $j=\pi[\pi[i]-1]$. 同理，次于 j 的第二长度等价于 $s[j-1]$ 的前缀函数值，$j^{(2)}=\pi[j-1]$.

显然我们可以得到一个关于 j 的状态转移方程：$j^{(n)}=\pi[j^{(n-1)}-1]$，$(j^{(n-1)}>0)$

最终算法

所以最终我们可以构建一个不需要进行任何字符串比较，并且只进行 $O(n)$ 次操作的算法.

而且该算法的实现出人意料的短且直观：

实现

- C++

```cpp
vector<int> prefix_function(string s) {
  int n = (int)s.length();
  vector<int> pi(n);
  for (int i = 1; i < n; i++) {
    int j = pi[i - 1];
    while (j > 0 && s[i] != s[j]) j = pi[j - 1];
    if (s[i] == s[j]) j++;
    pi[i] = j;
  }
  return pi;
```

```
}
```

- Python

```python
def prefix_function(s):
    n = len(s)
    pi = [0] * n
    for i in range(1, n):
        j = pi[i - 1]
        while j > 0 and s[i] != s[j]:
            j = pi[j - 1]
        if s[i] == s[j]:
            j += 1
        pi[i] = j
    return pi
```

- Java

```java
static int[] prefix_function(String s) {
    int n = s.length();
    int[] pi = new int[n];
    for (int i = 1; i < n; i++) {
        int j = pi[i - 1];
        while (j > 0 && s.charAt(i) != s.charAt(j)) {
            j = pi[j - 1];
        }
        if (s.charAt(i) == s.charAt(j)) {
            j++;
        }
        pi[i] = j;
    }
    return pi;
}
```

这是一个 **在线** 算法，即其当数据到达时处理它——举例来说，你可以一个字符一个字符的读取字符串，立即处理它们以计算出每个字符的前缀函数值. 该算法仍然需要存储字符串本身以及先前计算过的前缀函数值，但如果我们已经预先知道该字符串前缀函数的最大可能取值 M，那么我们仅需要存储该字符串的前 $M+1$ 个字符以及对应的前缀函数值.

应用

在字符串中查找子串：Knuth–Morris–Pratt 算法

该算法由 Knuth、Pratt 和 Morris 在 1977 年共同发布[1]. 该任务是前缀函数的一个典型应用.

过程

给定一个文本 t 和一个字符串 s，我们尝试找到并展示 s 在 t 中的所有出现（occurrence）.

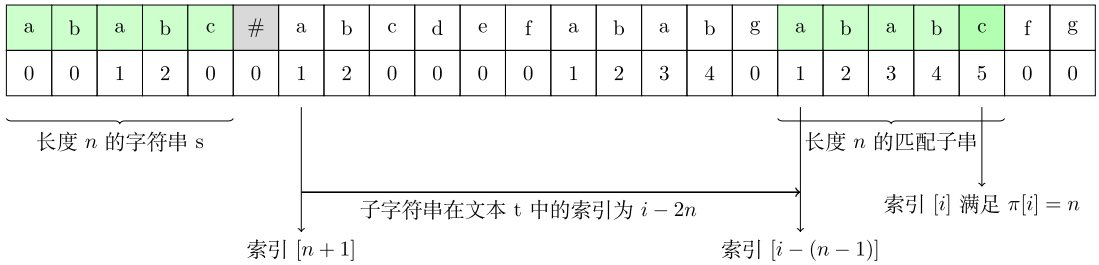
为了简便起见，我们用 n 表示字符串 s 的长度，用 m 表示文本 t 的长度.

我们构造一个字符串 $s+\#+t$，其中 $\#$ 为一个既不出现在 s 中也不出现在 t 中的分隔符. 接下来计算该字符串的前缀函数. 现在考虑该前缀函数除去最开始 $n+1$ 个值（即属于字符串 s 和分隔符的函数值）后其余函数值的意义. 根据定义，$\pi[i]$ 为右端点在 i 且同时为一个前缀的最长真子串的长度，具体到我们

[1] Knuth, Donald E., James H. Morris, Jr, and Vaughan R. Pratt. "Fast pattern matching in strings." SIAM journal on computing 6.2 (1977): 323-350.doi: 10.1137/0206024[1]

的这种情况下，其值为与 s 的前缀相同且右端点位于 i 的最长子串的长度．由于分隔符的存在，该长度不可能超过 n．而如果等式 $\pi[i]=n$ 成立，则意味着 s 完整出现在该位置（即其右端点位于位置 i）．注意该位置的下标是对字符串 $s+\#+t$ 而言的．

因此如果在某一位置 i 有 $\pi[i]=n$ 成立，则字符串 s 在字符串 t 的 $i-(n-1)-(n+1)=i-2n$ 处出现．下图所示为索引的示意图．



正如在前缀函数的计算中已经提到的那样，如果我们知道前缀函数的值永远不超过一特定值，那么我们不需要存储整个字符串以及整个前缀函数，而只需要二者开头的一部分．在我们这种情况下这意味着只需要存储字符串 $s+\#$ 以及相应的前缀函数值即可．我们可以一次读入字符串 t 的一个字符并计算当前位置的前缀函数值．

因此 Knuth–Morris–Pratt 算法（简称 KMP 算法）用 $O(n+m)$ 的时间以及 $O(n)$ 的内存解决了该问题．

实现

- C++

```cpp
vector<int> find_occurrences(string text, string pattern) {
  string cur = pattern + '#' + text;
  int sz1 = text.size(), sz2 = pattern.size();
  vector<int> v;
  vector<int> lps = prefix_function(cur);
  for (int i = sz2 + 1; i <= sz1 + sz2; i++) {
    if (lps[i] == sz2) v.push_back(i - 2 * sz2);
  }
  return v;
}
```

- Python

```python
def find_occurrences(t, s):
    cur = s + "#" + t
    sz1, sz2 = len(t), len(s)
    ret = []
    lps = prefix_function(cur)
    for i in range(sz2 + 1, sz1 + sz2 + 1):
        if lps[i] == sz2:
            ret.append(i - 2 * sz2)
    return ret
```

- Java

```java
static List<Integer> find_occurrences(String text, String pattern) {
    String cur = pattern + '#' + text;
    int sz1 = text.length(), sz2 = pattern.length();
```

```
    List<Integer> v = new ArrayList<>();
    int[] lps = prefix_function(cur);
    for (int i = sz2 + 1; i <= sz1 + sz2; i++) {
        if (lps[i] == sz2) {
            v.add(i - 2 * sz2);
        }
    }
    return v;
}
```

字符串的周期

对字符串 s 和 $0 < p \leq |\ s\ |$，若 $s[i] = s[i+p]$ 对所有 $i \in [0, |\ s\ | - p - 1]$ 成立，则称 p 是 s 的周期.

对字符串 s 和 $0 \leq r < |\ s\ |$，若 s 长度为 r 的前缀和长度为 r 的后缀相等，就称 s 长度为 r 的前缀是 s 的 border.

由 s 有长度为 r 的 border 可以推导出 $|\ s\ | - r$ 是 s 的周期.

根据前缀函数的定义，可以得到 s 所有的 border 长度，即 $\pi[n-1], \pi[\pi[n-1]-1], \ldots$. [2]

所以根据前缀函数可以在 $O(n)$ 的时间内计算出 s 所有的周期. 其中，由于 $\pi[n-1]$ 是 s 最长 border 的长度，所以 $n - \pi[n-1]$ 是 s 的最小周期.

统计每个前缀的出现次数

在该节我们将同时讨论两个问题. 给定一个长度为 n 的字符串 s，在问题的第一个变种中我们希望统计每个前缀 $s[0...i]$ 在同一个字符串的出现次数，在问题的第二个变种中我们希望统计每个前缀 $s[0...i]$ 在另一个给定字符串 t 中的出现次数.

首先让我们来解决第一个问题. 考虑位置 i 的前缀函数值 $\pi[i]$. 根据定义，其意味着字符串 s 一个长度为 $\pi[i]$ 的前缀在位置 i 出现并以 i 为右端点，同时不存在一个更长的前缀满足前述定义. 与此同时，更短的前缀可能以该位置为右端点. 容易看出，我们遇到了在计算前缀函数时已经回答过的问题：给定一个长度为 j 的前缀，同时其也是一个右端点位于 i 的后缀，下一个更小的前缀长度 $k < j$ 是多少？该长度的前缀需同时也是一个右端点为 i 的后缀. 因此以位置 i 为右端点，有长度为 $\pi[i]$ 的前缀，有长度为 $\pi[\pi[i]-1]$ 的前缀，有长度为 $\pi[\pi[\pi[i]-1]-1]$ 的前缀，等等，直到长度变为 0. 故而我们可以通过下述方式计算答案.

实现

- C++

```
vector<int> ans(n + 1);
for (int i = 0; i < n; i++) ans[pi[i]]++;
for (int i = n - 1; i > 0; i--) ans[pi[i - 1]] += ans[i];
for (int i = 0; i <= n; i++) ans[i]++;
```

- Python

```
ans = [0] * (n + 1)
for i in range(0, n):
    ans[pi[i]] += 1
for i in range(n - 1, 0, -1):
```

[2] 金策 - 字符串算法选讲[2]

```
    ans[pi[i - 1]] += ans[i]
for i in range(0, n + 1):
    ans[i] += 1
```

解释

在上述代码中我们首先统计每个前缀函数值在数组 π 中出现了多少次，然后再计算最后答案：如果我们知道长度为 i 的前缀出现了恰好 $\text{ans}[i]$ 次，那么该值必须被叠加至其最长的既是后缀也是前缀的子串的出现次数中．在最后，为了统计原始的前缀，我们对每个结果加 1．

现在考虑第二个问题．我们应用来自 Knuth–Morris–Pratt 的技巧：构造一个字符串 $s + \# + t$ 并计算其前缀函数．与第一个问题唯一的不同之处在于，我们只关心与字符串 t 相关的前缀函数值，即 $i \geq n + 1$ 的 $\pi[i]$．有了这些值之后，我们可以同样应用在第一个问题中的算法来解决该问题．

一个字符串中本质不同子串的数目

给定一个长度为 n 的字符串 s，我们希望计算其本质不同子串的数目．

我们将迭代的解决该问题．换句话说，在知道了当前的本质不同子串的数目的情况下，我们要找出一种在 s 末尾添加一个字符后重新计算该数目的方法．

令 k 为当前 s 的本质不同子串数量．我们添加一个新的字符 c 至 s．显然，会有一些新的子串以字符 c 结尾．我们希望对这些以该字符结尾且我们之前未曾遇到的子串计数．

构造字符串 $t = s + c$ 并将其反转得到字符串 $t^{\sim}$．现在我们的任务变为计算有多少 $t^{\sim}$ 的前缀未在 $t^{\sim}$ 的其余任何地方出现．如果我们计算了 $t^{\sim}$ 的前缀函数最大值 $\pi_{\max}$，那么最长的出现在 s 中的前缀其长度为 $\pi_{\max}$．自然的，所有更短的前缀也出现了．

因此，当添加了一个新字符后新出现的子串数目为 $|\ s\ | + 1 - \pi_{\max}$．

所以对于每个添加的字符，我们可以在 $O(n)$ 的时间内计算新子串的数目，故最终复杂度为 $O(n^2)$．

值得注意的是，我们也可以重新计算在头部添加一个字符，或者从尾或者头移除一个字符时的本质不同子串数目．

字符串压缩

给定一个长度为 n 的字符串 s，我们希望找到其最短的「压缩」表示，也即我们希望寻找一个最短的字符串 t，使得 s 可以被 t 的一份或多份拷贝的拼接表示．

显然，我们只需要找到 t 的长度即可．知道了该长度，该问题的答案即为长度为该值的 s 的前缀．

让我们计算 s 的前缀函数．通过使用该函数的最后一个值 $\pi[n-1]$，我们定义值 $k = n - \pi[n-1]$．我们将证明，如果 k 整除 n，那么 k 就是答案，否则不存在一个有效的压缩，故答案为 n．

假定 n 可被 k 整除．那么字符串可被划分为长度为 k 的若干块．根据前缀函数的定义，该字符串长度为 $n - k$ 的前缀等于其后缀．但是这意味着最后一个块同倒数第二个块相等，并且倒数第二个块同倒数第三个块相等，等等．作为其结果，所有块都是相等的，因此我们可以将字符串 s 压缩至长度 k．

证明

诚然，我们仍需证明该值为最优解．实际上，如果有一个比 k 更小的压缩表示，那么前缀函数的最后一个值 $\pi[n-1]$ 必定比 $n - k$ 要大．因此 k 就是答案．

[3] 在俄文版及英文版中该部分证明均疑似有误．本文章中的该部分证明由作者自行添加．

现在假设 n 不可以被 k 整除，我们将通过反证法证明这意味着答案为 n^3. 假设其最小压缩表示 r 的长度为 p（p 整除 n），字符串 s 被划分为 $n/p \geq 2$ 块. 那么前缀函数的最后一个值 $\pi[n-1]$ 必定大于 $n-p$（如果等于则 n 可被 k 整除），也即其所表示的后缀将部分的覆盖第一个块. 现在考虑字符串的第二个块. 该块有两种解释：第一种为 $r_0 r_1 \ldots r_{p-1}$，另一种为 $r_{p-k} r_{p-k+1} \ldots r_{p-1} r_0 r_1 \ldots r_{p-k-1}$. 由于两种解释对应同一个字符串，因此可得到 p 个方程组成的方程组，该方程组可简写为 $r_{(i+k) \bmod p} = r_{i \bmod p}$，其中 $\cdot \bmod p$ 表示模 p 意义下的最小非负剩余.

$$\overbrace{r_0\ r_1\ r_2\ r_3\ r_4\ r_5}^{p}\ \overbrace{r_0\ r_1\ r_2\ r_3\ r_4 r_5}^{p}$$

$$r_0\ r_1\ r_2\ r_3\ \overbrace{r_0\ r_1\ r_2\ r_3\ r_4\ r_5}^{p}\ \underbrace{r_0\ r_1}_{\pi[11]=8}$$

根据扩展欧几里得算法我们可以得到一组 x 和 y 使得 $xk + yp = \gcd(k, p)$. 通过与等式 $pk - kp = 0$ 适当叠加我们可以得到一组 $x' > 0$ 和 $y' < 0$ 使得 $x'k + y'p = \gcd(k, p)$. 这意味着通过不断应用前述方程组中的方程我们可以得到新的方程组 $r_{(i+\gcd(k,p)) \bmod p} = r_{i \bmod p}$.

由于 $\gcd(k, p)$ 整除 p，这意味着 $\gcd(k, p)$ 是 r 的一个周期. 又因为 $\pi[n-1] > n-p$，故有 $n - \pi[n-1] = k < p$，所以 $\gcd(k, p)$ 是一个比 p 更小的 r 的周期. 因此字符串 s 有一个长度为 $\gcd(k, p) < p$ 的压缩表示，同 p 的最小性矛盾.

综上所述，不存在一个长度小于 k 的压缩表示，因此答案为 k.

根据前缀函数构建一个自动机

让我们重新回到通过一个分隔符将两个字符串拼接的新字符串. 对于字符串 s 和 t 我们计算 $s + \# + t$ 的前缀函数. 显然，因为 $\#$ 是一个分隔符，前缀函数值永远不会超过 $|\ s\ |$. 因此我们只需要存储字符串 $s + \#$ 和其对应的前缀函数值，之后就可以动态计算对于之后所有字符的前缀函数值：

$$\underbrace{s_0\ s_1\ \ldots\ s_{n-1}\ \#}_{\text{need to store}}\ \underbrace{t_0\ t_1\ \ldots\ t_{m-1}}_{\text{do not need to store}}$$

实际上在这种情况下，知道 t 的下一个字符 c 以及之前位置的前缀函数值便足以计算下一个位置的前缀函数值，而不需要用到任何其它 t 的字符和对应的前缀函数值.

换句话说，我们可以构造一个 **自动机**（一个有限状态机）：其状态为当前的前缀函数值，而从一个状态到另一个状态的转移则由下一个字符确定.

因此，即使没有字符串 t，我们同样可以应用构造转移表的算法构造一个转移表 $(\text{old}\ \pi, c) \to \text{new_}\pi$：

实现

```
void compute_automaton(string s, vector<vector<int>>& aut) {
  s += '#';
  int n = s.size();
  vector<int> pi = prefix_function(s);
  aut.assign(n, vector<int>(26));
  for (int i = 0; i < n; i++) {
    for (int c = 0; c < 26; c++) {
      int j = i;
      while (j > 0 && 'a' + c != s[j]) j = pi[j - 1];
      if ('a' + c == s[j]) j++;
      aut[i][c] = j;
    }
  }
```

```
14 }
```

然而在这种形式下，对于小写字母表，算法的时间复杂度为 $O(|\Sigma| n^2)$. 注意到我们可以应用动态规划来利用表中已计算过的部分. 只要我们从值 j 变化到 $\pi[j-1]$，那么我们实际上在说转移 (j, c) 所到达的状态同转移 $(\pi[j-1], c)$ 一样，但该答案我们之前已经精确计算过了.

实现

```
1  void compute_automaton(string s, vector<vector<int>>& aut) {
2    s += '#';
3    int n = s.size();
4    vector<int> pi = prefix_function(s);
5    aut.assign(n, vector<int>(26));
6    for (int i = 0; i < n; i++) {
7      for (int c = 0; c < 26; c++) {
8        if (i > 0 && 'a' + c != s[i])
9          aut[i][c] = aut[pi[i - 1]][c];
10       else
11         aut[i][c] = i + ('a' + c == s[i]);
12     }
13   }
14 }
```

最终我们可在 $O(|\Sigma| n)$ 的时间复杂度内构造该自动机.

该自动机在什么时候有用呢？首先，记得大部分时候我们为了一个目的使用字符串 $s + \# + t$ 的前缀函数：寻找字符串 s 在字符串 t 中的所有出现.

因此使用该自动机的最直接的好处是 **加速计算字符串 $s + \# + t$ 的前缀函数**.

通过构建 $s + \#$ 的自动机，我们不再需要存储字符串 s 以及其对应的前缀函数值. 所有转移已经在表中计算过了.

但除此以外，还有第二个不那么直接的应用. 我们可以在字符串 t 是 **某些通过一些规则构造的巨型字符串** 时，使用该自动机加速计算. Gray 字符串，或者一个由一些短的输入串的递归组合所构造的字符串都是这种例子.

出于完整性考虑，我们来解决这样一个问题：给定一个数 $k \le 10^5$，以及一个长度 $\le 10^5$ 的字符串 s，我们需要计算 s 在第 k 个 Gray 字符串中的出现次数. 回想起 Gray 字符串以下述方式定义：

$$g_1 = \texttt{a}$$
$$g_2 = \texttt{aba}$$
$$g_3 = \texttt{abacaba}$$
$$g_4 = \texttt{abacabadabacaba}$$

由于其天文数字般的长度，在这种情况下即使构造字符串 t 都是不可能的：第 k 个 Gray 字符串有 $2^k - 1$ 个字符. 然而我们可以在仅仅知道开头若干前缀函数值的情况下，有效计算该字符串末尾的前缀函数值.

除了自动机之外，我们同时需要计算值 $G[i][j]$：在从状态 j 开始处理 g_i 后的自动机的状态，以及值 $K[i][j]$：当从状态 j 开始处理 g_i 后，s 在 g_i 中的出现次数. 实际上 $K[i][j]$ 为在执行操作时前缀函数取值为 $|s|$ 的次数. 易得问题的答案为 $K[k][0]$.

我们该如何计算这些值呢？首先根据定义，初始条件为 $G[0][j] = j$ 以及 $K[0][j] = 0$. 之后所有值可以通过先前的值以及使用自动机计算得到. 为了对某个 i 计算相应值，回想起字符串 g_i 由 g_{i-1}，字母表中第 i 个字符，以及 g_{i-1} 三者拼接而成. 因此自动机会途径下列状态：

$$\text{mid} = \text{aut}[G[i-1][j]][i]$$

$$G[i][j] = G[i-1][\text{mid}]$$

$K[i][j]$ 的值同样可被简单计算.

$$K[i][j] = K[i-1][j] + [\text{mid} == |\ s\ |] + K[i-1][\text{mid}]$$

其中 $[\cdot]$ 当其中表达式取值为真时值为 1，否则为 0. 综上，我们已经可以解决关于 Gray 字符串的问题，以及一大类与之类似的问题. 举例来说，应用同样的方法可以解决下列问题：给定一个字符串 s 以及一些模式 t_i，其中每个模式以下列方式给出：该模式由普通字符组成，当中可能以 t_k^{cnt} 的形式递归插入先前的字符串，也即在该位置我们必须插入字符串 t_k cnt 次. 以下是这些模式的一个例子：

$$t_1 = \texttt{abdeca}$$

$$t_2 = \texttt{abc} + t_1^{30} + \texttt{abd}$$

$$t_3 = t_2^{50} + t_1^{100}$$

$$t_4 = t_2^{10} + t_3^{100}$$

递归代入会使字符串长度爆炸式增长，他们的长度甚至可以达到 100^{100} 的数量级. 而我们必须找到字符串 s 在每个字符串中的出现次数.

该问题同样可通过构造前缀函数的自动机解决. 同之前一样，我们利用先前计算过的结果对每个模式计算其转移然后相应统计答案即可.

练习题目

- UVa 455 "Periodic Strings"[3]
- UVa 11022 "String Factoring"[4]
- UVa 11452 "Dancing the Cheeky-Cheeky"[5]
- UVa 12604 - Caesar Cipher[6]
- UVa 12467 - Secret Word[7]
- UVa 11019 - Matrix Matcher[8]
- SPOJ - Pattern Find[9]
- Codeforces - Anthem of Berland[10]
- Codeforces - MUH and Cube Walls[11]

参考资料与注释

本页面主要译自博文 Префикс-функция. Алгоритм Кнута-Морриса-Пратта[12] 与其英文翻译版 Prefix function. Knuth–Morris–Pratt algorithm[13]. 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.

[1] doi: 10.1137/0206024

[2] 金策 - 字符串算法选讲

[3] UVa 455 "Periodic Strings"

[4] UVa 11022 "String Factoring"

[5] UVa 11452 "Dancing the Cheeky-Cheeky"

[6] UVa 12604 - Caesar Cipher

[7] UVa 12467 - Secret Word

[8] UVa 11019 - Matrix Matcher

[9] SPOJ - Pattern Find

[10] Codeforces - Anthem of Berland

[11] Codeforces - MUH and Cube Walls

[12] Префикс-функция. Алгоритм Кнута-Морриса-Пратта

[13] Prefix function. Knuth–Morris–Pratt algorithm 

8.8 Boyer–Moore 算法

Authors: minghu6

前置知识：前缀函数与 KMP 算法→8.7@p917.

KMP 算法将前缀匹配的信息用到了极致，

而 BM 算法背后的基本思想是通过后缀匹配获得比前缀匹配更多的信息来实现更快的字符跳转.

引入

想象一下，如果我们的模式字符串 pat，被放在文本字符串 $string$ 的左手起头部，使它们的第一个字符对齐.

pat : EXAMPLE

$string$: HERE IS A SIMPLE EXAMPLE ...

⇑

在这里做定义，往后不赘述：

pat 的长度为 $patlen$，特别地对于从 0 开始的串来说，规定 $patlastpos = patlen - 1$ 为 pat 串最后一个字符的位置；

$string$ 的长度 $stringlen$，$stringlastpos = stringlen - 1$.

假如我们知道了 $string$ 的第 $patlen$ 个字符 $char$（与 pat 的最后一个字符对齐）考虑我们能得到什么信息：

观察 1

如果我们知道 $char$ 这个字符不在 pat 中，我们就不用考虑 pat 从 $string$ 的第 1 个、第 2 个.....第 $patlen$ 个字符起出现的情况，而可以直接将 pat 向下滑动 $patlen$ 个字符.

观察 2

更一般地，**如果出现在 pat 最末尾（也就是最右边）的那一个 $char$ 字符的位置是离末尾端差了 $delta_1$ 个字符**，

那么就可以不用匹配，直接将 pat 向后滑动 $delta_1$ 个字符：如果滑动距离少于 $delta_1$，那么仅就 $char$ 这个字符就无法被匹配，当然模式字符串 pat 也就不会被匹配.

因此除非 $char$ 字符可以和 pat 末尾的那个字符匹配，否则 $string$ 要跳过 $delta_1$ 个字符（相当于 pat 向后滑动了 $delta_1$ 个字符）. 并且我们可以得到一个计算 $delta_1$ 的函数 $delta_1(char)$：

```
int delta1(char char)
    if char 不在 pat 中 || char 是 pat 上最后一个字符
        return patlen
    else
        return patlastpos − i   // i 为出现在 pat 最末尾的那一个 char 出现的位置，即 pat[i]=char
```

需要注意，显然这个表只需计算到 $patlastpos - 1$ 的位置.

现在假设 $char$ 和 pat 最后一个字符匹配到了，那我们就看看 $char$ 前一个字符和 pat 的倒数第二个字符是否匹配：

如果是，就继续回退直到整个模式串 pat 完成匹配（这时我们就在 $string$ 上成功得到了一个 pat 的匹配）；

或者，我们也可能会在匹配完 pat 的倒数第 m 个字符后，在倒数第 $m+1$ 个字符上失配，这时我们就希望把 pat 向后滑动到下一个可能会实现匹配的位置，当然我们希望滑动得越远越好.

观察 3(a)

在 **观察 2** 中提到, 当匹配完 pat 的倒数 m 个字符后, 如果在倒数第 $m+1$ 个字符失配, 为了使得 $string$ 中的失配字符与 pat 上对应字符对齐,

需要把 pat 向后滑动 k 个字符，也就是说我们应该把注意力看向之后的 $k+m$ 个字符（也就是看向 pat 滑动 k 之后，末段与 $string$ 对齐的那个字符）.

而 $k = delta_1 - m$,

所以我们的注意力应该沿着 $string$ 向后跳 $delta_1 - m + m = delta_1$ 个字符.

然而，我们有机会跳过更多的字符，请继续看下去.

观察 3(b)

如果我们知道 $string$ 接下来的 m 个字符和 pat 的最后 m 个字符匹配，假设这个子串为 $subpat$,

我们还知道在 $string$ 失配字符 $char$ 后面是与 $subpat$ 相匹配的子串，而假如 pat 对应失配字符前面存在 $subpat$，我们可以将 pat 向下滑动一段距离,

使得失配字符 $char$ 在 pat 上对应的字符前面出现的 $subpat$（合理重现，plausible reoccurrence，以下也简称 pr）与 $string$ 的 $subpat$ 对齐. 如果 pat 上有多个 $subpat$，按照从右到左的后缀匹配顺序，取第一个（rightmost plausible reoccurrence，以下也简称 rpr）.

假设此时 pat 向下滑动的 k 个字符（也即 pat 末尾端的 $subpat$ 与其最右边的合理重现的距离），这样我们的注意力应该沿着 $string$ 向后滑动 $k+m$ 个字符，这段距离我们称之为 $delta_2(j)$:

假定 $rpr(j)$ 为 $subpat = pat[j+1...patlastpos]$ 在 $pat[j]$ 上失配时的最右边合理重现的位置，$rpr(j) < j$（这里只给出简单定义，在下文的算法设计章节里会有更精确的讨论），那么显然 $k = j - rpr(j), m = patlastpos - j$.

所以有:

int $delta2$(**int** j) // j 为失配字符在 pat 上对应字符的位置
return $patlastpos - rpr(j)$

于是我们在失配时，可以把 $string$ 上的注意力往后跳过 $\max(delta_1, delta_2)$ 个字符

过程

箭头指向失配字符 $char$:

pat : AT-THAT
$string$: ... WHICH-FINALLY-HALTS.– AT-THAT-POINT ...
⇑

F 没有出现 pat 中，根据 **观察 1**，pat 直接向下移动 $patlen$ 个字符，也就是 7 个字符:

pat : AT-THAT
$string$: ... WHICH-FINALLY-HALTS.– AT-THAT-POINT ...
⇑

根据 **观察 2**，我们需要将 pat 向下移动 4 个字符使得短横线字符对齐:

pat : AT-THAT
$string$: ... WHICH-FINALLY-HALTS.– AT-THAT-POINT ...
⇑

现在 *char*:T 匹配了，把 $string$ 上的指针左移一步继续匹配:

```
pat :                          AT-THAT
string : ... WHICH-FINALLY-HALTS.– AT-THAT-POINT ...
                                  ⇑
```

根据 **观察 3(a)**，L 失配，因为 L 不在 pat 中，所以 pat 向下移动 $k = delta_1 - m = 7 - 1 = 6$ 个字符，而 $string$ 上指针向下移动 $delta_1 = 7$ 个字符：

```
pat :                              AT-THAT
string : ... WHICH-FINALLY-HALTS.– AT-THAT-POINT ...
                                      ⇑
```

这时 $char$ 又一次匹配到了 pat 的最后一个字符 T，$string$ 上的指针向左匹配，匹配到了 A，继续向左匹配，发现在字符 - 失配：

```
pat :                              AT-THAT
string : ... WHICH-FINALLY-HALTS.– AT-THAT-POINT ...
                                      ⇑
```

显然直观上看，此时根据 **观察 3(b)**，将 pat 向下移动 $k = 5$ 个字符，使得后缀 AT 对齐，这种滑动可以获得 $string$ 指针最大的滑动距离，此时 $delta_2 = k + patlastpos - j = 5 + 6 - 4 = 7$，即 $string$ 上指针向下滑动 7 个字符.

而从形式化逻辑看，此时，$delta_1 = 7 - 1 - 2 = 4, delta_2 = 7, \max(delta_1, delta_2) = 7$，这样从形式逻辑上支持了进行 **观察 3(b)** 的跳转：

```
pat :                                  AT-THAT
string : ... WHICH-FINALLY-HALTS.– AT-THAT-POINT ...
                                          ⇑
```

现在我们发现了 pat 上每一个字符都和 $string$ 上对应的字符相等，我们在 $string$ 上找到了一个 pat 的匹配. 而只花费了 14 次对 $string$ 的引用，其中 7 次是完成一个成功的匹配所必需的比较次数（$patlen = 7$），另外 7 次让我们跳过了 22 个字符.

算法设计

最初的匹配算法

解释

现在看这样一个利用 $delta_1$ 和 $delta_2$ 进行字符串匹配的算法：

$i \leftarrow patlastpos.$
$j \leftarrow patlastpos.$
loop
    if $j < 0$
        return $i + 1$

    if $string[i] = pat[j]$
        $j \leftarrow j - 1$
        $i \leftarrow i - 1$
        continue

    $i \leftarrow i + max(delta_1(string[i]), delta_2(j))$

    if $i > stringlastpos$
        return $false$
    $j \leftarrow patlastpos$

如果上面的算法 **return** $false$，表明 pat 不在 $string$ 中；如果返回一个数字，表示 pat 在 $string$ 左起第一次出现的位置.

然后让我们更精细地描述下计算 $delta_2$，所依靠的 $rpr(j)$ 函数.

根据前文定义，$rpr(j)$ 表示在 $pat(j)$ 失配时，子串 $subpat = pat[j+1...patlastpos]$ 在 $pat[j]$ 最右边合理重现的位置.

也就是说需要找到一个最好的 k, 使得 $pat[k...k+patlastpos-j-1] = pat[j+1...patlastpos]$，另外要考虑两种特殊情况：

1. 当 $k < 0$ 时，相当于在 pat 前面补充了一段虚拟的前缀，实际上也符合 $delta_2$ 跳转的原理.
2. 当 $k > 0$ 时，如果 $pat[k-1] = pat[j]$，则这个 $pat[k...k+patlastpos-j-1]$ 不能作为 $subpat$ 的合理重现. 原因是 $pat[j]$ 本身是失配字符，所以 pat 向下滑动 k 个字符后，在后缀匹配过程中仍然会在 $pat[k-1]$ 处失配.

还要注意两个限制条件：

1. $k < j$. 因为当 $k = j$ 时，有 $pat[k] = pat[j]$，在 $pat[j]$ 上失配的字符也会在 $pat[k]$ 上失配.
2. 考虑到 $delta_2(patlastpos) = 0$，所以规定 $rpr(patlastpos) = patlastpos$.

过程

由于理解 $rpr(j)$ 是实现 BoyerMoore 算法的核心，所以我们使用如下两个例子进行详细说明：

j :	0 1 2 3 4 5 6 7 8
pat :	A B C X X X A B C
$rpr(j)$:	5 4 3 2 1 0 2 1 8
sgn :	• ▸ – • ▸ – • ▸ +

对于 $rpr(0)$，$subpat$ 为 BCXXXABC，在 $pat[0]$ 之前的最右边合理重现只能是 [(BCXXX)ABC]XXXABC，也就是最右边合理重现位置为 -5，即 $rpr(j) = -5$；

对于 $rpr(1)$，$subpat$ 为 CXXXABC，在 $pat[1]$ 之前的最右边的合理重现是 [(CXXX)ABC]XXXABC，所以 $rpr(j) = -4$；

对于 $rpr(2)$, $subpat$ 为 XXXABC, 在 $pat[2]$ 之前的最右边的合理重现是 [(XXX)ABC]XXXABC, 所以 $rpr(j) = -3$；

对于 $rpr(3)$，$subpat$ 为 XXABC，在 $pat[3]$ 之前的最右边的合理重现是 [(XX)ABC]XXXABC，所以 $rpr(j) = -2$；

对于 $rpr(4)$，$subpat$ 为 XABC，在 $pat[4]$ 之前的最右边的合理重现是 [(X)ABC]XXXABC，所以 $rpr(j) = -1$；

对于 $rpr(5)$，$subpat$ 为 ABC，在 $pat[5]$ 之前的最右边的合理重现是 [ABC]XXXABC，所以 $rpr(j) = 0$；

对于 $rpr(6)$，$subpat$ 为 BC，又因为 $string[0] = string[6]$，即 $string[0]$ 等于失配字符 $string[6]$，所以 $string[0...2]$ 并不是符合条件的 $subpat$ 的合理重现，所以在最右边的合理重现是 [(BC)]ABCXXXABC，所以 $rpr(j) = -2$；

对于 $rpr(7)$，$subpat$ 为 C，同理又因为 $string[7] = string[1]$，所以 $string[1...2]$ 并不是符合条件的 $subpat$ 的合理重现，在最右边的合理重现是 [(C)]ABCXXXABC，所以 $rpr(j) = -1$；

对于 $rpr(8)$，根据 $delta_2$ 定义，$rpr(patlastpos) = patlastpos$，得到 $rpr(8) = 8$.

现在再看一下另一个例子：

j :	0 1 2 3 4 5 6 7 8
pat :	A B Y X C D E Y X
$rpr(j)$:	8 7 6 5 4 3 2 1 8
sgn :	• ▸ – • ▸ – 1. • +

对于 $rpr(0)$，$subpat$ 为 BYXCDEYX，在 $pat[0]$ 之前的最右边合理重现只能是 [(BYXCDEYX)]ABYXCDEYX，也就是最右边合理重现位置为 -8，即 $rpr(j) = -8$；

对于 $rpr(1)$，$subpat$ 为 `YXCDEYX`，在 $pat[1]$ 之前的最右边合理重现只能是 [(`YXCDEYX`)]`ABYXCDEYX`，$rpr(j) = -7$；

对于 $rpr(2)$，$subpat$ 为 `XCDEYX`，在 $pat[2]$ 之前的最右边合理重现只能是 [(`XCDEYX`)]`ABYXCDEYX`，$rpr(j) = -6$；

对于 $rpr(3)$，$subpat$ 为 `CDEYX`，在 $pat[3]$ 之前的最右边合理重现只能是 [(`CDEYX`)]`ABYXCDEYX`，$rpr(j) = -5$；

对于 $rpr(4)$，$subpat$ 为 `DEYX`，在 $pat[4]$ 之前的最右边合理重现只能是 [(`DEYX`)]`ABYXCDEYX`，$rpr(j) = -4$；

对于 $rpr(5)$，$subpat$ 为 `EYX`，在 $pat[5]$ 之前的最右边合理重现只能是 [(`EYX`)]`ABYXCDEYX`，$rpr(j) = -3$；

对于 $rpr(6)$，$subpat$ 为 `YX`，因为 $string[2...3] = string[7...8]$ 并且有 $string[6] \neq string[1]$，所以在 $pat[6]$ 之前的最右边的合理重现是 `AB`[`YX`]`CDEYX`，$rpr(j) = 2$；

对于 $rpr(7)$，$subpat$ 为 `X`，虽然 $string[3] = string[8]$ 但是因为 $string[2] = string[7]$，所以在 $pat[7]$ 之前的最右边的合理重现是 [`X`]`ABYXCDEYX`，$rpr(j) = -1$

对于 $rpr(8)$，根据 $delta_2$ 定义，$rpr(patlastpos) = patlastpos$，得到 $rpr(8) = 8$.

对匹配算法的一个改进

最后，实践过程中考虑到搜索过程中估计有 80% 的时间用在了 **观察 1** 的跳转上，也就是 $string[i]$ 和 $pat[patlastpos]$ 不匹配，然后跳跃整个 $patlen$ 进行下一次匹配的过程.

于是，可以为此进行特别的优化：

我们定义一个 $delta0$：

```
int delta0(char char)
    if char = pat[patlastpos]
        return large  // large 为一个整数，需要满足 large>stringlastpos+patlen
    return delta1(char)
```

用 $delta0$ 代替 $delta_1$，得到改进后的匹配算法：

```
i ← patlastpos
loop
    if i > stringlastpos
        return false

    while i < stringlen
        i ← i + delta0(string(i))  // 除非 string[i]和 pat 末尾字符匹配，否则至多向下滑动 patlen
    if i ⩽ large                   //此时表示 string 上没有一个字符和 pat 末尾字符匹配
        return false

    i ← i − large
    j ← patlastpos.
    while j ⩾ 0 and string[i] = pat[j]
        j ← j − 1
        i ← i − 1

    if j < 0
        return i + 1
    i ← i + max(delta1(string[i]), delta2(j))
```

其中 $large$ 起到多重作用，一是类似后面介绍的 Horspool 算法进行快速的坏字符跳转，二是辅助检测字符串搜索是否完成.

经过改进，比起原算法，在做 **观察 1** 跳转时不必每次进行 $delta_2$ 的多余计算，使得在通常字符集下搜索字符串的性能有了明显的提升.

delta2 构建细节

引入

在 1977 年 10 月的 *Communications of the ACM* 上， Boyer、Moor 的论文[1]中只描述了 $delta_2$ 静态表，

构造 $delta_2$ 的具体实现的讨论出现在 1977 年 6 月 Knuth、Morris、Pratt 在 *SIAM Journal on Computing* 上正式联合发表的 KMP 算法的论文[2].

朴素算法

在介绍 Knuth 的 $delta_2$ 构建算法之前，根据定义，我们会有一个适用于小规模问题的朴素算法:

1. 对于 `[0, patlen)` 区间的每一个位置 `i`，根据 `subpat` 的长度确定其重现位置的区间，也就是 `[-subpatlen, i]`;
2. 可能的重现位置按照从右到左进行逐字符比较, 寻找符合 $delta_2$ 要求的最右边 $subpat$ 的重现位置;
3. 最后别忘了令 $delta_2(lastpos) = 0$.

实现

```rust
use std::cmp::PartialEq;

pub fn build_delta_2_table_naive(p: &[impl PartialEq]) -> Vec<usize> {
    let patlen = p.len();
    let lastpos = patlen - 1;
    let mut delta_2 = vec![];

    for i in 0..patlen {
        let subpatlen = (lastpos - i) as isize;

        if subpatlen == 0 {
            delta_2.push(0);
            break;
        }

        for j in (-subpatlen..(i + 1) as isize).rev() {
            // subpat 匹配
            if (j..j + subpatlen)
            .zip(i + 1..patlen)
            .all(|(rpr_index, subpat_index)| {
                if rpr_index < 0 {
                    return true;
                }

                if p[rpr_index as usize] == p[subpat_index] {
                    return true;
                }

                false
            })
```

[1]1977 年 Boyer–Moore 算法论文[1]
[2]1977 年 KMP 算法论文[2]

```
            && (j <= 0 || p[(j - 1) as usize] != p[i])
            {
                delta_2.push((lastpos as isize - j) as usize);
                break;
            }
        }
    }

    delta_2
}
```

特别地，对 Rust 语言特性进行必要地解释，下不赘述：

- `usize` 和 `isize` 是和内存指针同字节数的无符号整数和有符号整数，在 32 位机上相当于 `u32` 和 `i32`，64 位机上相当于 `u64` 和 `i64`.
- 索引数组、向量、分片时使用 `usize` 类型的数字（因为在做内存上的随机访问并且下标不能为负值），所以如果需要处理负值要用 `isize`，而进行索引时又要用 `usize`，这就看到使用 `as` 关键字进行二者之间的显式转换.
- `impl PartialEq` 只是用作泛型，可以同时支持 `Unicode` 编码的 `char` 和二进制的 `u8`.

显然，该暴力算法的时间复杂度为 $O(n^3)$.

高效算法

下面我们要介绍的是时间复杂度为 $O(n)$，但是需要额外 $O(n)$ 空间复杂度的高效算法.

虽然 1977 年 Knuth 提出了这个构建方法，然而他的原始版本的构建算法存在一个缺陷，实际上对于某些 pat 产生不出符合定义的 $delta_2$.

Rytter 在 1980 年 *SIAM Journal on Computing* 上发表的文章[3]对此提出了修正，以下是 $delta_2$ 的构建算法：

首先考虑到 $delta_2$ 的定义比较复杂，我们按照 $subpat$ 的重现位置进行分类，每一类进行单独处理，这是高效实现的关键思路.

按照重现位置由远到近，也就是偏移量由大到小，分成如下几类：

1. 整个 $subpat$ 重现位置完全在 pat 左边的，比如 [(`EYX`)]`ABYXCDEYX`，此时 $delta_2(j) = patlastpos \times 2 - j$;
2. $subpat$ 的重现有一部分在 pat 左边，有一部分是 pat 头部，比如 [(`XX`)`ABC`]`XXXABC`，此时 $patlastpos < delta_2(j) < patlastpos \times 2 - j$；我们把 $subpat$ 完全在 pat 头部的边际情况也归类在这里（当然根据实现也可以归类在下边），比如 [`ABC`]`XXXABC`，此时 $patlastpos = delta_2(j)$;
3. $subpat$ 的重现完全在 pat 中，比如 `AB`[`YX`]`CDEYX`，此时 $delta_2(j) < patlastpos$.

现在来讨论如何高效地计算这三种情况：

第一种情况

这是最简单的情况，只需一次遍历并且可以顺便将 $delta_2$ 初始化.

第二种情况

我们观察什么时候会出现 $subpat$ 的重现一部分在 pat 左边，一部分是 pat 的头部的情况呢？应该是 $subpat$ 的某个后缀和 pat 的某个前缀相等，

比如之前的例子：

[3] 1980 年 Rytter 纠正 Knuth 的论文[3]

j : 0 1 2 3 4 5 6 7 8

pat : A B C X X X A B C

$delta_2(3)$ 的重现 [(XX)ABC]XXXABC，$subpat$ XXABC 的后缀与 pat 前缀中，有相等的，是 ABC.

实际上，对第二种和第三种情况的计算的关键都需要前缀函数的计算和和应用.

那么只要 j 取值使得 $subpat$ 包含这个相等的后缀，那么就可以得到第二种情况的 $subpat$ 的重现，对于例子，我们只需要使得 $j \leqslant 5$,

而当 $j = 5$ 时，就是 $subpat$ 完全在 pat 头部的边际情况.

可以计算此时的 $delta_2(j)$:

设此时这对相等的前后缀长度为 $prefixlen$，可知 $subpatlen = patlastpos - j$，那么在 pat 左边的部分长度是 $subpatlen - prefixlen$,

而 $rpr(j) = -(subpatlen - prefixlen)$, 所以得到 $delta_2(j) = patlastpos - rpr(j) = patlastpos \times 2 - j - prefixlen$.

其后面可能会有多对相等的前缀和后缀，比如:

j : 0 1 2 3 4 5 6 7 8 9

pat : A B A A B A A B A A

在 $j \leq 2$ 处有 ABAABAA，$2 < j \leq 5$ 处有 ABAA，在 $5 < j \leq 8$ 处有 A

Knuth 算法的缺陷是只考虑了最长的那一对的情况, 但实际上我们要考虑所有 $subpat$ 后缀与 pat 前缀相等的情况，等同于计算 pat 所有真后缀和真前缀相等的情况，并按照长度从大到小，j 分区间计算不同的 $delta_2(j)$.

利用前缀函数和逆向运用计算前缀函数的状态转移方程: $j^{(n)} = \pi[j^{(n-1)} - 1]$, 以得到 pat 所有相等的真前缀和真后缀长度. 从 $\pi[patlastpos]$ 开始作为最长一对的长度, 然后通过逆向运行状态转移方程, 得到下一个次长相等真前缀和真后缀的长度.

如此就完成了第二种情况的 $delta_2$ 的计算.

第三种情况

$subpat$ 的重现恰好就在 pat 中（不包括 pat 的头部），也就是按照从右到左的顺序，在 $pat[0...patlastpos - 1]$ 中寻找 $subpat$.

如果用 BM 算法解决，我们就得到了一个 BM 的递归实现的第三种情况，结束条件是 $patlen \leqslant 2$.

而且根据 $delta_2$ 的定义，找到的 $subpat$ 的重现的下一个（也就是左边一个）字符和作为 pat 后缀的 $subpat$ 的下一个字符不能一样.

这就很好地启发了我们, 可以使用类似于计算前缀函数的过程计算第三种情况, 只不过是左右反过来的前缀函数:

- 两个指针分别指向子串的左端点和子串最长公共前后缀的「前缀」位置，从右向左移动，在发现指向的两个字符相等时继续移动，此时相当于「前缀」变大;
- 当两个字符不相等时，之前相等的部分就满足了 $delta_2$ 对重现的要求，并且回退指向「前缀」位置的指针直到构成新的字符相等或者出界.

同前缀函数一样，需要一个辅助数组，用于回退，可以使用之前计算第二种情况所生成的前缀数组的空间.

实现

上述实现

```rust
use std::cmp::PartialEq;
use std::cmp::min;

pub fn build_delta_2_table_improved_minghu6(p: &[impl PartialEq]) ->
Vec<usize> {
    let patlen = p.len();
    let lastpos = patlen - 1;
    let mut delta_2 = Vec::with_capacity(patlen);

    // 第一种情况
    // delta_2[j] = lastpos * 2 - j
    for i in 0..patlen {
        delta_2.push(lastpos * 2 - i);
    }

    // 第二种情况
    // lastpos <= delata2[j] = lastpos * 2 - j
    let pi = compute_pi(p);  // 计算前缀函数
    let mut i = lastpos;
    let mut last_i = lastpos; // 只是为了初始化
    while pi[i] > 0 {
        let start;
        let end;

        if i == lastpos {
            start = 0;
        } else {
            start = patlen - pi[last_i];
        }

        end = patlen - pi[i];

        for j in start..end {
            delta_2[j] = lastpos * 2 - j - pi[i];
        }

        last_i = i;
        i = pi[i] - 1;
    }

    // 第三种情况
    // delata2[j] < lastpos
    let mut j = lastpos;
    let mut t = patlen;
    let mut f = pi;
    loop {
        f[j] = t;
        while t < patlen && p[j] != p[t] {
            // 使用min函数保证后面可能的回退不会覆盖前面的数据
            delta_2[t] = min(delta_2[t], lastpos - 1 - j);
            t = f[t];
        }
```

```
52
53          t -= 1;
54          if j == 0 {
55              break;
56          }
57          j -= 1;
58      }
59
60      // 没有实际意义，只是为了完整定义
61      delta_2[lastpos] = 0;
62
63      delta_2
64  }
```

Galil 规则对多次匹配时最坏情况的改善

关于后缀匹配算法的多次匹配问题

之前的搜索算法只涉及到在 $string$ 中寻找第一次 pat 匹配的情况，而对与在 $string$ 中寻找全部 pat 的匹配的情况有很多不同的算法思路，这个问题的核心关注点是：如何利用之前匹配成功的字符的信息，将最坏情况下的时间复杂度降为线性.

在原始的成功匹配后，简单的 $string$ 的指针向后滑动 $patlen$ 距离后重新开始后缀匹配，这会导致最坏情况下回到 $O(mn)$ 的时间复杂度（按照惯例，m 为 $patlen$，n 为 $stringlen$，下同）.

比如一个极端的例子：pat：`AAA`，$string$：`AAAAA`....

对此 Knuth 提出来的一个方法是用一个「数量有限」的状态的集合来记录 $patlen$ 长度的字符，这种算法保证 $string$ 上每一个字符最多比较一次，但代价是这个「数量有限」的状态可能规模并不小，对于一个字符彼此不相等的 pat，需要 $\frac{1}{2}m^2 + m$ 个状态.

下面介绍的思路简单且不需要额外预处理开销的 Galil 算法[4].

Galil 规则

假定一个 pat，它是某个子串 U 重复 n 次构成的字符串 $UUUU...$ 的前缀，那么我们称 U 为 pat 的一个周期.

比如，pat : `ABCABCAB`，是 `ABC` 的重复 `ABCABCABC` 的前缀，所以 `ABC` 的长度 3 就是这个 pat 的周期长度，也即 pat 满足 $pat[i] = pat[i + 3]$.

pat 至少拥有一个长度为它自身的周期，我们规定最短的周期为 k，$k \leq patlen$.

在搜索过程中，假如我们的 pat 成功地完成了一次匹配，那么依照周期的特点，实际上只需将 $string$ 向后滑动 k 个字符，比较这 k 个字符是否对应相等就可以直接判断是否存在 pat 的又一个匹配.

为计算这个最短周期的长度, 我们假设已知 pat 的相等的一对前缀 - 后缀, 设它们的长度为 $prefixlen$, 那么有 $pat[i] = pat[i + (patlen - prefixlen)]$. 从而得到长度为 $patlen - prefixlen$ 的周期,

当我们知道 pat 最长的那一对相等的前缀 - 后缀，我们就得到了 pat 最短的周期.

而最长相等的前后缀长度，$\pi[patlastpos]$，已经在我们在计算 $delta_2$ 的过程中，所以实际不需要额外的预处理时间和空间，就能将后缀匹配算法最坏情况的时间复杂度改善成线性.

[4] 1979 年介绍 Galil 算法的论文[4]

结合上述优化的 BM 的搜索算法最终实现

```
#[cfg(target_pointer_width = "64")]
const LARGE: usize = 10_000_000_000_000_000_000_000;

#[cfg(not(target_pointer_width = "64"))]
const LARGE: usize = 2_000_000_000;

pub struct BMPattern<'a> {
    pat_bytes: &'a [u8],
    delta_1: [usize; 256],
    delta_2: Vec<usize>,
    k: usize  // pat的最短周期长度
}

impl<'a> BMPattern<'a> {
    // ...

    pub fn find_all(&self, string: &str) -> Vec<usize> {
        let mut result = vec![];
        let string_bytes = string.as_bytes();
        let stringlen = string_bytes.len();
        let patlen = self.pat_bytes.len();
        let pat_last_pos = patlen - 1;
        let mut string_index = pat_last_pos;
        let mut pat_index;
        let l0 =  patlen - self.k;
        let mut l = 0;

        while string_index < stringlen {
            let old_string_index = string_index;

            while string_index < stringlen {
                string_index += self.delta0(string_bytes[string_index]);
            }
            if string_index < LARGE {
                break;
            }

            string_index -= LARGE;

            // 如果string_index发生移动，意味着自从上次成功匹配后发生了至少一次的失败匹配.
            // 此时需要将Galil规则的二次匹配的偏移量归零.
            if old_string_index < string_index {
                l = 0;
            }

            pat_index = pat_last_pos;

            while pat_index > l && string_bytes[string_index] == self.pat_bytes[pat_index] {
                string_index -= 1;
```

```
                pat_index -= 1;
            }

            if pat_index == l && string_bytes[string_index] ==
self.pat_bytes[pat_index] {
                result.push(string_index - l);

                string_index += pat_last_pos - l + self.k;
                l = l0;
            } else {
                l = 0;
                string_index += max(
                    self.delta_1[string_bytes[string_index] as usize],
                    self.delta_2[pat_index],
                );
            }
        }

        result
    }
}
```

最坏情况在实践中性能影响

从实践的角度上说，理论上的最坏情况并不容易影响性能表现，哪怕是很小的只有 4 的字符集的随机文本测试下这种最坏情况的影响也小到难以观察.

也因此如果没有很好地设计，使用 Galil 法则会拖累一点平均的性能表现，但对于一些极端特殊的 pat 和 $string$ 比如例子中的：pat：`AAA`，$string$：`AAAAA`...，Galil 规则的应用确实会使得性能表现提高数倍.

改进算法

Simplified Boyer–Moore 算法

BM 算法最复杂的地方就在于 $delta_2$ 表（也就是好后缀表）的构建，而实践中发现，在一般的字符集上的匹配性能主要依靠 $delta_1$ 表（也就是坏字符表），于是出现了仅仅使用 $delta_1$ 表的简化版 BM 算法，通常性能和原版差距很小.

Boyer–Moore–Horspol 算法

Horspol 算法同样是基于坏字符的规则，在与 pat 尾部对齐的字符上应用 $delta_1$. 效果类似于对原版匹配算法的改进，通常性能优于原版本.

实现

```
pub struct HorspoolPattern<'a> {
    pat_bytes: &'a [u8],
    bm_bc: [usize; 256],
}

impl<'a> HorspoolPattern<'a> {
    // ...
    pub fn find_all(&self, string: &str) -> Vec<usize> {
```

```
        let mut result = vec![];
        let string_bytes = string.as_bytes();
        let stringlen = string_bytes.len();
        let pat_last_pos = self.pat_bytes.len() - 1;
        let mut string_index = pat_last_pos;

        while string_index < stringlen {
            if &string_bytes[string_index-pat_last_pos..string_index+1] ==
self.pat_bytes {
                result.push(string_index-pat_last_pos);
            }

            string_index += self.bm_bc[string_bytes[string_index] as usize];
        }

        result
    }
}
```

Boyer–Moore–Sunday 算法

Sunday 算法同样是利用坏字符规则，只不过相比 Horspool 它更进一步，直接关注 pat 尾部对齐的那个字符的下一个字符.

实现它只需要稍微修改 $delta_1$ 表，相当于在 $patlen + 1$ 长度的 pat 上进行构建.

Sunday 算法通常用作一般情况下实现最简单而且平均表现最好之一的实用算法, 通常性能比 Horspool 和 BM 要好一点.

实现

```
pub struct SundayPattern<'a> {
    pat_bytes: &'a [u8],
    sunday_bc: [usize; 256],
}

impl<'a> SundayPattern<'a> {
    // ...
    fn build_sunday_bc(p: &'a [u8]) -> [usize; 256] {
        let mut sunday_bc_table = [p.len() + 1; 256];

        for i in 0..p.len() {
            sunday_bc_table[p[i] as usize] = p.len() - i;
        }

        sunday_bc_table
    }

    pub fn find_all(&self, string: &str) -> Vec<usize> {
        let mut result = vec![];
        let string_bytes = string.as_bytes();
        let pat_last_pos = self.pat_bytes.len() - 1;
```

```
        let stringlen = string_bytes.len();
        let mut string_index = pat_last_pos;

        while string_index < stringlen {
            if &string_bytes[string_index - pat_last_pos..string_index+1] ==
self.pat_bytes {
                result.push(string_index - pat_last_pos);
            }

            if string_index + 1 == stringlen {
                break;
            }

            string_index += self.sunday_bc[string_bytes[string_index + 1]
as usize];
        }

        result
    }
}
```

BMHBNFS 算法

该算法结合了 Horspool 和 Sunday，是 CPython 实现 `stringlib` 模块时用到的 `find` 的算法[5]，以下简称 B5S.

B5S 基本思路是：

1. 按照后缀匹配的思路，首先比较 $patlastpos$ 位置对应的字符是否相等，如果相等就比较 $0...patlastpos-1$ 对应位置的字符是否相等，如果仍然相等，那么就发现一个匹配；
2. 如果任何一个阶段发生不匹配，就进入跳转阶段；
3. 在跳转阶段，首先观察 $patlastpos$ 位置的下一个字符是否在 pat 中，如果不在，直接向右滑动 $patlen+1$，这是 Sunday 算法的最大利用；

   如果这个字符在 pat 中，对 $patlastpos$ 处的字符利用 $delta_1$ 进行 Horspool 跳转.

而根据时间节省还是空间节省为第一目标，算法会有差别巨大的不同实现.

时间节省版本

实现

```
pub struct B5STimePattern<'a> {
    pat_bytes: &'a [u8],
    alphabet: [bool;256],
    bm_bc: [usize;256],
    k: usize
}

impl<'a> B5STimePattern<'a> {
    pub fn new(pat: &'a str) -> Self {
```

[5] B5S 算法的介绍[5]

```
        assert_ne!(pat.len(), 0);

        let pat_bytes = pat.as_bytes();
        let (alphabet, bm_bc, k) = B5STimePattern::build(pat_bytes);

        B5STimePattern { pat_bytes, alphabet, bm_bc, k }
    }

    fn build(p: &'a [u8]) -> ([bool;256], [usize;256], usize)  {
        let mut alphabet = [false;256];
        let mut bm_bc = [p.len(); 256];
        let lastpos = p.len() - 1;

        for i in 0..lastpos {
            alphabet[p[i] as usize] = true;
            bm_bc[p[i] as usize] = lastpos - i;
        }

        alphabet[p[lastpos] as usize] = true;

        (alphabet, bm_bc, compute_k(p))
    }

    pub fn find_all(&self, string: &str) -> Vec<usize> {
        let mut result = vec![];
        let string_bytes = string.as_bytes();
        let pat_last_pos = self.pat_bytes.len() - 1;
        let patlen = self.pat_bytes.len();
        let stringlen = string_bytes.len();
        let mut string_index = pat_last_pos;
        let mut offset = pat_last_pos;
        let offset0 = self.k - 1;

        while string_index < stringlen {
            if string_bytes[string_index] == self.pat_bytes[pat_last_pos] {
                if &string_bytes[string_index-offset..string_index] ==
&self.pat_bytes[pat_last_pos-offset..pat_last_pos] {
                    result.push(string_index-pat_last_pos);

                    offset = offset0;

                    // Galil rule
                    string_index += self.k;
                    continue;
                }
            }

            if string_index + 1 == stringlen {
                break;
            }

            offset = pat_last_pos;
```

```
61
62              if !self.alphabet[string_bytes[string_index+1] as usize] {
63                  string_index += patlen + 1;  // sunday
64              } else {
65                  string_index += self.bm_bc[string_bytes[string_index] as
usize];  // horspool
66              }
67          }
68
69          result
70      }
71  }
```

该版本的 B5S 性能表现非常理想，在目前介绍的后缀匹配系列算法中是通常情况下是最快的.

空间节省版本

同样在 CPython `stringlib` 中实现，使用了两个整数近似取代了字符表和 $delta_1$ 的作用，极大地节省了空间：

1. 用一个简单的 Bloom 过滤器取代字符表（alphabet）

实现

```
1  pub struct BytesBloomFilter {
2      mask: u64,
3  }
4
5  impl BytesBloomFilter {
6      pub fn new() -> Self {
7          SimpleBloomFilter {
8              mask: 0,
9          }
10     }
11
12     fn insert(&mut self, byte: &u8) {
13         (self.mask) |= 1u64 << (byte & 63);
14     }
15
16     fn contains(&self, char: &u8) -> bool {
17         (self.mask & (1u64 << (byte & 63))) != 0
18     }
19 }
```

Bloom 过滤器设设计通过牺牲准确率（实际还有运行时间）来极大地节省存储空间的 `Set` 类型的数据结构，它的特点是会将集合中不存在的项误判为存在（False Positives，简称 FP），但不会把集合中存在的项判断为不存在（False Negatives，简称 FN），因此使用它可能会因为 FP 而没有得到最大的字符跳转，但不会因为 FN 而跳过本应匹配的字符.

理论上分析，上述「Bloom 过滤器」的实现在 *pat* 长度在 50 个 Bytes 时，FP 概率约为 0.5，而 *pat* 长度在 10 个 Bytes 时，FP 概率约为 0.15.

虽然这不是一个标准的 Bloom 过滤器，首先它实际上没有使用一个真正的哈希函数，实际上它只是一个字符映射，将 0-255 的字节映射为它的前六位构成的数.

但考虑到我们在内存上的进行字符搜索, 这种简化就非常重要, 即使用目前已知最快的非加密哈希算法 xxHash[6]，计算所需要的时间仍比它高一个数量级.

另外当 pat 在 30 字节以下时，为了达到最佳的 FP 概率，需要不止一个哈希函数. 但这么做意义不大，因为用装有两个 `u128` 数字的数组就已经可以构建字符表的全字符集.

2. 使用 $delta_1(pat[patlastpos])$ 代替整个 $delta_1$

观察 $delta_1$，最常使用处就是后缀匹配时第一个字符就不匹配是最常见的不匹配的情况，于是令 `skip = delta1(pat[patlastpos])`,

在第一阶段不匹配时，直接向下滑动 `skip` 个字符；但当第二阶段不配时，因为缺乏整个 $delta_1$ 的信息，只能向下滑动一个字符.

实现

```rust
pub struct B5SSpacePattern<'a> {
    pat_bytes: &'a [u8],
    alphabet: BytesBloomFilter,
    skip: usize,
}

impl<'a> B5SSpacePattern<'a> {
    pub fn new(pat: &'a str) -> Self {
        assert_ne!(pat.len(), 0);

        let pat_bytes = pat.as_bytes();
        let (alphabet, skip) = B5SSpacePattern::build(pat_bytes);

        B5SSpacePattern { pat_bytes, alphabet, skip}
    }

    fn build(p: &'a [u8]) -> (BytesBloomFilter, usize)  {
        let mut alphabet = BytesBloomFilter::new();
        let lastpos = p.len() - 1;
        let mut skip = p.len();

        for i in 0..p.len()-1 {
            alphabet.insert(&p[i]);

            if p[i] == p[lastpos] {
                skip = lastpos - i;
            }
        }

        alphabet.insert(&p[lastpos]);

        (alphabet, skip)
    }

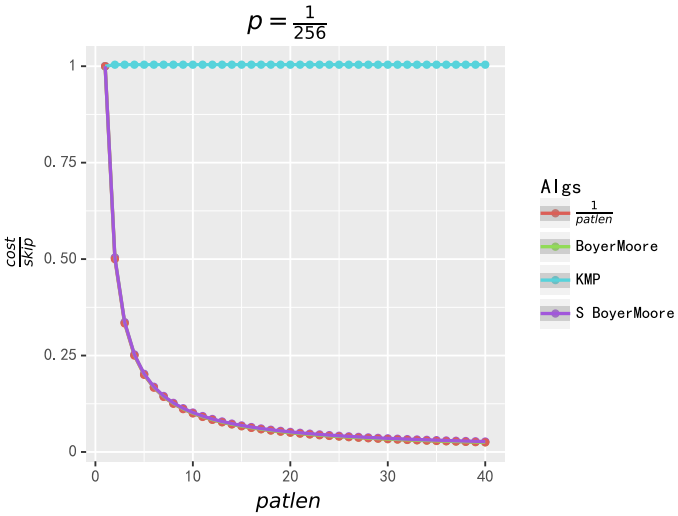
    pub fn find_all(&self, string: &'a str) -> Vec<usize> {
        let mut result = vec![];
        let string_bytes = string.as_bytes();
        let pat_last_pos = self.pat_bytes.len() - 1;
        let patlen = self.pat_bytes.len();
```

```
40          let stringlen = string_bytes.len();
41          let mut string_index = pat_last_pos;
42
43          while string_index < stringlen {
44              if string_bytes[string_index] ==
self.pat_bytes[pat_last_pos] {
45                  if &string_bytes[string_index-
pat_last_pos..string_index] == &self.pat_bytes[..patlen-1] {
46                      result.push(string_index-pat_last_pos);
47                  }
48
49                  if string_index + 1 == stringlen {
50                      break;
51                  }
52
53                  if !
self.alphabet.contains(&string_bytes[string_index+1]) {
54                      string_index += patlen + 1;  // sunday
55                  } else {
56                      string_index += self.skip;  // horspool
57                  }
58              } else {
59                  if string_index + 1 == stringlen {
60                      break;
61                  }
62
63                  if !
self.alphabet.contains(&string_bytes[string_index+1]) {
64                      string_index += patlen + 1;  // sunday
65                  } else {
66                      string_index += 1;
67                  }
68              }
69
70          }
71
72          result
73      }
74  }
```

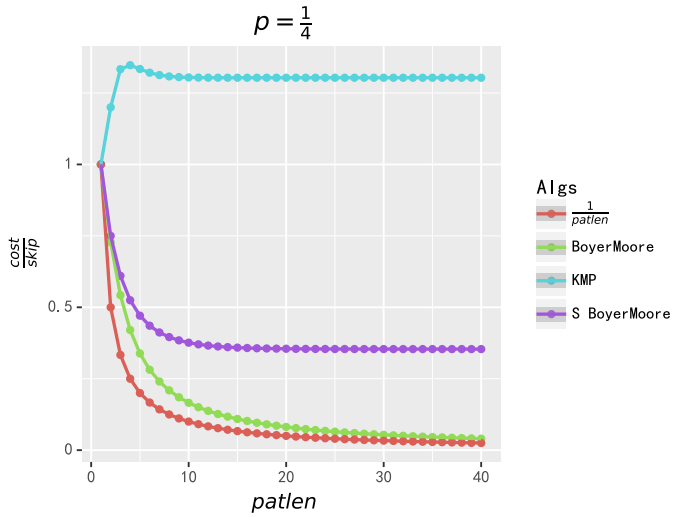
这个版本的算法相较于前面的后缀匹配算法不够快，但差距不大，性能仍然优于 KMP，得益于它至多两个 `u64` 的整数的优秀空间复杂度.

理论分析

以下是一般字符集下各算法的表现，纵坐标类似于执行开销（cost 指匹配成功 m 个字符后失配时的代价，skip 指发生失配时向下滑动 k 个字符的概率），越小性能越好．横坐标为模式字符串 pat 的长度：



在较小字符集（DNA {A, C, T, G} 碱基对序列）中的表现：



综上，在较大的字符集，比如日常搜索的过程中，BoyerMoore 系列算法的优越表现，其中主要依赖 $delta_1$ 表实现字符跳转；

另一方面，在较小的字符集里，$delta_1$ 的作用下降，而 $delta_2$ 的作用得到了体现.

如果有一定富裕空间的情况下，完整的空间复杂度为 $O(m)$ 的 BoyerMoore 算法更加通用，综合表现最优.

参考资料与注释

[1] 1977 年 Boyer–Moore 算法论文

[2] 1977 年 KMP 算法论文

[3] 1980 年 Rytter 纠正 Knuth 的论文

[4] 1979 年介绍 Galil 算法的论文

[5] B5S 算法的介绍

[6] xxHash

8.9 Z 函数（扩展 KMP）

Authors: LeoJacob, Marcythm, minghu6

约定：字符串下标以 0 为起点.

定义

对于一个长度为 n 的字符串 s，定义函数 $z[i]$ 表示 s 和 $s[i,n-1]$（即以 $s[i]$ 开头的后缀）的最长公共前缀（LCP）的长度，则 z 被称为 s 的 **Z 函数**. 特别地，$z[0]=0$.

国外一般将计算该数组的算法称为 **Z Algorithm**，而国内则称其为 **扩展 KMP**（exKMP）.

这篇文章介绍在 $O(n)$ 时间复杂度内计算 Z 函数的算法以及其各种应用.

解释

下面若干样例展示了对于不同字符串的 Z 函数:

- $z(\texttt{aaaaa})=[0,4,3,2,1]$
- $z(\texttt{aaabaab})=[0,2,1,0,2,1,0]$
- $z(\texttt{abacaba})=[0,0,1,0,3,0,1]$

朴素算法

Z 函数的朴素算法复杂度为 $O(n^2)$:

实现

- C++

```cpp
vector<int> z_function_trivial(string s) {
  int n = (int)s.length();
  vector<int> z(n);
  for (int i = 1; i < n; ++i)
    while (i + z[i] < n && s[z[i]] == s[i + z[i]]) ++z[i];
  return z;
}
```

- Python

```python
def z_function_trivial(s):
    n = len(s)
    z = [0] * n
    for i in range(1, n):
        while i + z[i] < n and s[z[i]] == s[i + z[i]]:
            z[i] += 1
    return z
```

线性算法

如同大多数字符串主题所介绍的算法，其关键在于，运用自动机的思想寻找限制条件下的状态转移函数，使得可以借助之前的状态来加速计算新的状态.

在该算法中，我们从 1 到 $n-1$ 顺次计算 $z[i]$ 的值（$z[0]=0$）. 在计算 $z[i]$ 的过程中，我们会利用已经计算好的 $z[0],\ldots,z[i-1]$.

对于 i，我们称区间 $[i,i+z[i]-1]$ 是 i 的 **匹配段**，也可以叫 Z-box.

算法的过程中我们维护右端点最靠右的匹配段. 为了方便，记作 $[l,r]$. 根据定义，$s[l,r]$ 是 s 的前缀. 在计算 $z[i]$ 时我们保证 $l\le i$. 初始时 $l=r=0$.

在计算 $z[i]$ 的过程中:

- 如果 $i\le r$，那么根据 $[l,r]$ 的定义有 $s[i,r]=s[i-l,r-l]$，因此 $z[i]\ge\min(z[i-l],r-i+1)$. 这时:
  - 若 $z[i-l]<r-i+1$，则 $z[i]=z[i-l]$.

    - 否则 $z[i-l] \geq r-i+1$，这时我们令 $z[i]=r-i+1$，然后暴力枚举下一个字符扩展 $z[i]$ 直到不能扩展为止.
- 如果 $i>r$，那么我们直接按照朴素算法，从 $s[i]$ 开始比较，暴力求出 $z[i]$.
- 在求出 $z[i]$ 后，如果 $i+z[i]-1>r$，我们就需要更新 $[l,r]$，即令 $l=i, r=i+z[i]-1$.

可以访问 这个网站[1] 来看 Z 函数的模拟过程.

实现

- C++

```cpp
vector<int> z_function(string s) {
  int n = (int)s.length();
  vector<int> z(n);
  for (int i = 1, l = 0, r = 0; i < n; ++i) {
    if (i <= r && z[i - l] < r - i + 1) {
      z[i] = z[i - l];
    } else {
      z[i] = max(0, r - i + 1);
      while (i + z[i] < n && s[z[i]] == s[i + z[i]]) ++z[i];
    }
    if (i + z[i] - 1 > r) l = i, r = i + z[i] - 1;
  }
  return z;
}
```

- Python

```python
def z_function(s):
    n = len(s)
    z = [0] * n
    l, r = 0, 0
    for i in range(1, n):
        if i <= r and z[i - l] < r - i + 1:
            z[i] = z[i - l]
        else:
            z[i] = max(0, r - i + 1)
            while i + z[i] < n and s[z[i]] == s[i + z[i]]:
                z[i] += 1
        if i + z[i] - 1 > r:
            l = i
            r = i + z[i] - 1
    return z
```

复杂度分析

对于内层 `while` 循环，每次执行都会使得 r 向后移至少 1 位，而 $r<n-1$，所以总共只会执行 n 次.

对于外层循环，只有一遍线性遍历.

总复杂度为 $O(n)$.

应用

我们现在来考虑在若干具体情况下 Z 函数的应用.

这些应用在很大程度上同 前缀函数→8.7@p917 的应用类似.

匹配所有子串

为了避免混淆，我们将 t 称作 **文本**，将 p 称作 **模式**. 所给出的问题是：寻找在文本 t 中模式 p 的所有出现（occurrence）.

为了解决该问题，我们构造一个新的字符串 $s = p + \diamond + t$，也即我们将 p 和 t 连接在一起，但是在中间放置了一个分割字符 $\diamond$（我们将如此选取 $\diamond$ 使得其必定不出现在 p 和 t 中）.

首先计算 s 的 Z 函数. 接下来，对于在区间 $[0, \mid t \mid -1]$ 中的任意 i，我们考虑以 $t[i]$ 为开头的后缀在 s 中的 Z 函数值 $k = z[i + \mid p \mid + 1]$. 如果 $k = \mid p \mid$，那么我们知道有一个 p 的出现位于 t 的第 i 个位置，否则没有 p 的出现位于 t 的第 i 个位置.

其时间复杂度（同时也是其空间复杂度）为 $O(\mid t \mid + \mid p \mid)$.

本质不同子串数

给定一个长度为 n 的字符串 s，计算 s 的本质不同子串的数目.

考虑计算增量，即在知道当前 s 的本质不同子串数的情况下，计算出在 s 末尾添加一个字符后的本质不同子串数.

令 k 为当前 s 的本质不同子串数. 我们添加一个新的字符 c 至 s 的末尾. 显然，会出现一些以 c 结尾的新的子串（以 c 结尾且之前未出现过的子串）.

设串 t 是 $s + c$ 的反串（反串指将原字符串的字符倒序排列形成的字符串）. 我们的任务是计算有多少 t 的前缀未在 t 的其他地方出现. 考虑计算 t 的 Z 函数并找到其最大值 $z_{\max}$. 则 t 的长度小于等于 $z_{\max}$ 的前缀的反串在 s 中是已经出现过的以 c 结尾的子串.

所以，将字符 c 添加至 s 后新出现的子串数目为 $\mid t \mid - z_{\max}$.

算法时间复杂度为 $O(n^2)$.

值得注意的是，我们可以用同样的方法在 $O(n)$ 时间内，重新计算在端点处添加一个字符或者删除一个字符（从尾或者头）后的本质不同子串数目.

字符串整周期

给定一个长度为 n 的字符串 s，找到其最短的整周期，即寻找一个最短的字符串 t，使得 s 可以被若干个 t 拼接而成的字符串表示.

考虑计算 s 的 Z 函数，则其整周期的长度为最小的 n 的因数 i，满足 $i + z[i] = n$.

该事实的证明同应用 前缀函数→8.7@p917 的证明一样.

练习题目

- luogu P5410【模板】扩展 KMP/exKMP（Z 函数）[2]
- luogu P7114【NOIP2020】字符串匹配[3]
- CF126B Password[4]
- UVa # 455 Periodic Strings[5]
- UVa # 11022 String Factoring[6]
- UVa 11475 - Extend to Palindrome[7]
- Codechef - Chef and Strings[8]
- Codeforces - Prefixes and Suffixes[9]
- Leetcode 2223 - Sum of Scores of Built Strings[10]

本页面主要译自博文 Z-функция строки и её вычисление[11] 与其英文翻译版 Z-function and its calculation[12]. 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] 这个网站 

[2] luogu P5410【模板】扩展 KMP/exKMP（Z 函数） 

[3] luogu P7114【NOIP2020】字符串匹配 

[4] CF126B Password 

[5] UVa # 455 Periodic Strings 

[6] UVa # 11022 String Factoring 

[7] UVa 11475 - Extend to Palindrome 

[8] Codechef - Chef and Strings 

[9] Codeforces - Prefixes and Suffixes 

[10] Leetcode 2223 - Sum of Scores of Built Strings 

[11] Z-функция строки и её вычисление 

[12] Z-function and its calculation 

8.10 AC 自动机

> **Authors:** Ir1d, Tiphereth-A, sshwy, ksyx, Marcythm, orzAtalod, Xeonacid, Enter-tainer, GavinZhengOI, Henry-ZHR, iamtwz, 383494, abc1763613206, aofall, Chrogeek, CoelacanthusHex, Dafenghh, DanJoshua, Gesrua, kenlig, lyccrius, Menci, opsiff, ouuan, partychicken, Persdre, Ruakker, shuzhouliu, StudyingFather, szdytom, XuYueming520, ZXyaang, alphagocc, c-forrest, Early0v0, GoodCoder666, HeRaNO, liangbob2023, qq2964, r-value, rickyxrc, Rickyxrc, shawlleyw, Unnamed2964, zica87, ZnPdCo, sun2snow

概述

AC（Aho–Corasick）自动机是 **以 Trie 的结构为基础**，结合 **KMP 的思想** 建立的自动机，用于解决多模式匹配等任务.

AC 自动机本质上是 Trie 上的自动机.

在阅读本文之前，请先阅读 KMP→8.7@p917 和 Trie→8.6@p903.

解释

简单来说，建立一个 AC 自动机有两个步骤：

1. 基础的 Trie 结构：将所有的模式串构成一棵 Trie；
2. KMP 的思想：对 Trie 树上所有的结点构造失配指针.

建立完毕后，就可以利用它进行多模式匹配.

字典树构建

AC 自动机在初始时会将若干个模式串插入到一个 Trie 里，然后在 Trie 上建立 AC 自动机. 这个 Trie 就是普通的 Trie，按照 Trie 原本的建树方法建树即可.

需要注意的是，Trie 中的结点表示的是某个模式串的前缀. 我们在后文也将其称作状态. 一个结点表示一个状态，Trie 的边就是状态的转移.

形式化地说，对于若干个模式串 $s_1, s_2, \cdots, s_n$，将它们构建一棵字典树后的所有状态的集合记作 Q.

失配指针

AC 自动机利用一个 fail 指针来辅助多模式串的匹配.

状态 u 的 fail 指针指向另一个状态 v，其中 $v \in Q$，且 v 是 u 的最长后缀（即在若干个后缀状态中取最长的一个作为 fail 指针）.

fail 指针与 KMP→8.7@p917 中的 next 指针相比：

1. 共同点：两者同样是在失配的时候用于跳转的指针.

2. 不同点：next 指针求的是最长 Border（即最长的相同前后缀），而 fail 指针指向所有模式串的前缀中匹配当前状态的最长后缀.

因为 KMP 只对一个模式串做匹配，而 AC 自动机要对多个模式串做匹配. 有可能 fail 指针指向的结点对应着另一个模式串，两者前缀不同.

总结下来，AC 自动机的失配指针指向当前状态的最长后缀状态.

注意：AC 自动机在做匹配时，同一位上可匹配多个模式串.

构建指针

下面介绍构建 fail 指针的 **基础思想**:

构建 fail 指针，可以参考 KMP 中构造 next 指针的思想.

考虑字典树中当前的结点 u，u 的父结点是 p，p 通过字符 c 的边指向 u，即 $\text{trie}(p,c)=u$. 假设深度小于 u 的所有结点的 fail 指针都已求得.

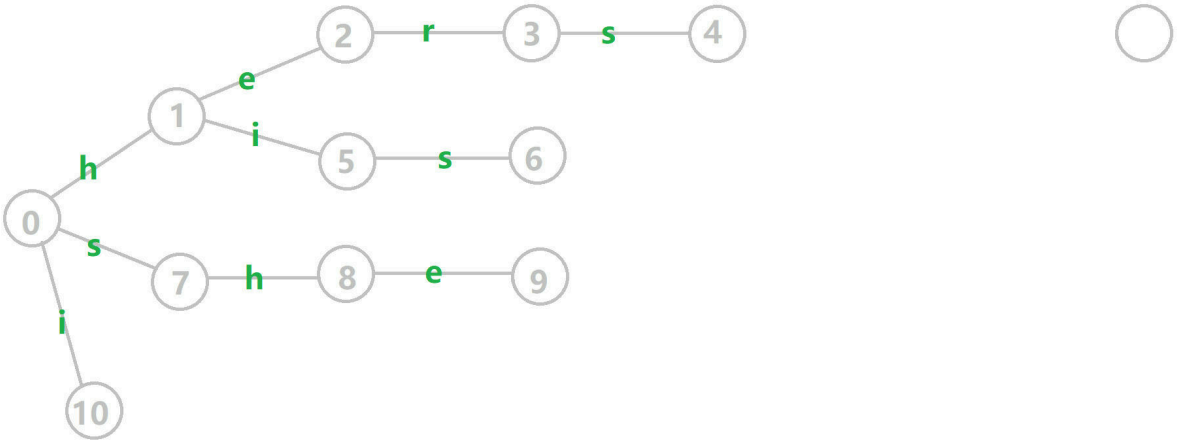
1. 如果 $\text{trie}(\text{fail}(p),c)$ 存在：则让 u 的 fail 指针指向 $\text{trie}(\text{fail}(p),c)$. 相当于在 p 和 $\text{fail}(p)$ 后面加一个字符 c，分别对应 u 和 $\text{fail}(u)$;
2. 如果 $\text{trie}(\text{fail}(p),c)$ 不存在：那么我们继续找到 $\text{trie}(\text{fail}(\text{fail}(p)),c)$. 重复判断过程，一直跳 fail 指针直到根结点;
3. 如果依然不存在，就让 fail 指针指向根结点.

如此即完成了 $\text{fail}(u)$ 的构建.

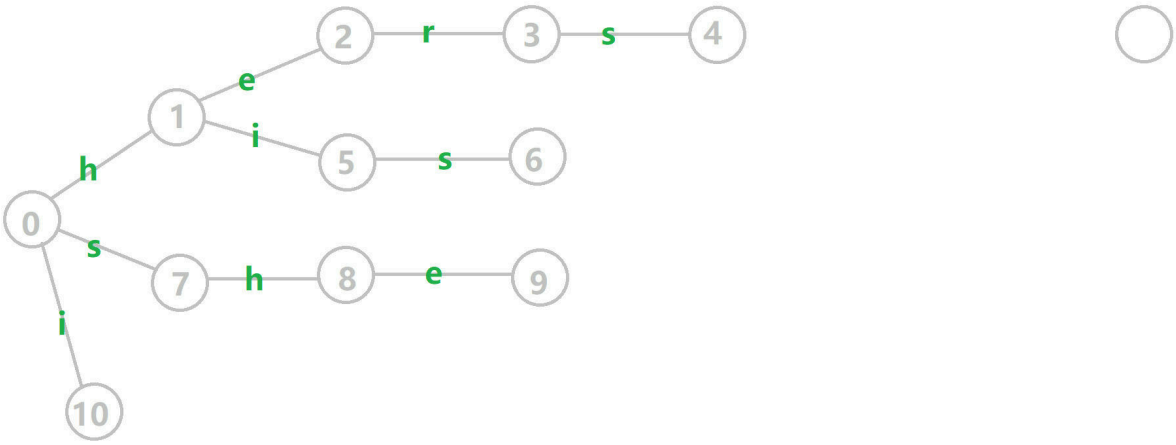
例子

下面将使用若干张 GIF 动图来演示对字符串 `i`、`he`、`his`、`she`、`hers` 组成的字典树构建 fail 指针的过程:

1. 黄色结点：当前的结点 u.
2. 绿色结点：表示已经 BFS 遍历完毕的结点.
3. 橙色的边：fail 指针.
4. 红色的边：当前求出的 fail 指针.

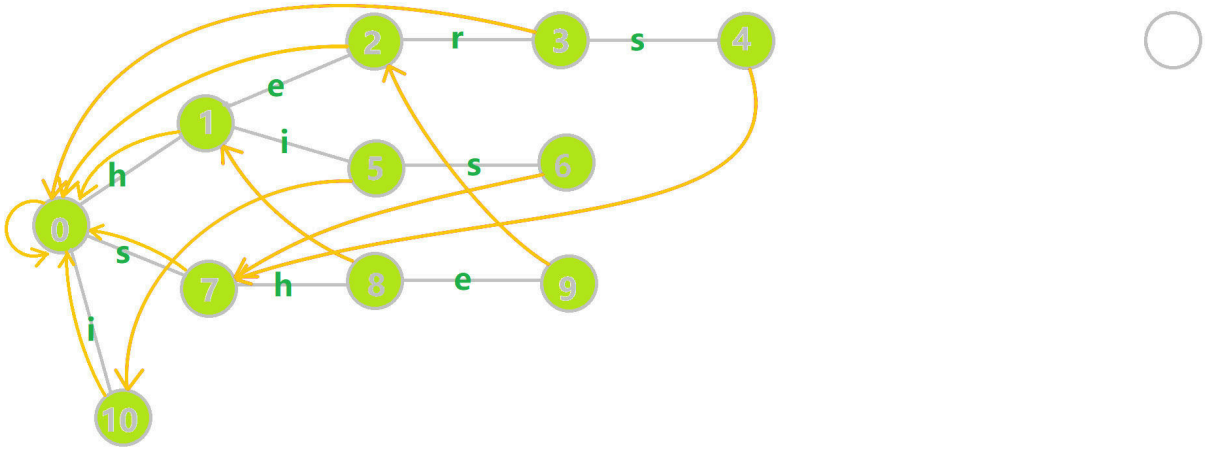


我们重点分析结点 6 的 fail 指针构建:



找到 6 的父结点 5，$\text{fail}(5)=10$. 然而结点 10 没有字母 s 连出的边；继续跳到 10 的 fail 指针，$\text{fail}(10)=0$. 发现 0 结点有字母 s 连出的边，指向 7 结点；所以 $\text{fail}(6)=7$.

下图展示了构建完毕的状态：



字典树与字典图

关注构建函数 `build`，该函数的目标有两个，一个是构建 fail 指针，一个是构建自动机. 相关变量定义如下：

1. `tr[u].son[c]`：有两种理解方式. 我们可以简单理解为字典树上的一条边，即 $\text{trie}(u,c)$；也可以理解为从状态（结点）u 后加一个字符 c 到达的状态（结点），即一个状态转移函数 $\text{trans}(u,c)$. 为了方便，下文中我们将用第二种理解方式.
2. 队列 `q`：用于 BFS 遍历字典树.
3. `tr[u].fail`：结点 u 的 fail 指针.

实现

- C++

```cpp
void build() {
  queue<int> q;
```

```
  for (int i = 0; i < 26; i++)
    if (tr[0].son[i]) q.push(tr[0].son[i]);
  while (!q.empty()) {
    int u = q.front();
    q.pop();
    for (int i = 0; i < 26; i++) {
      if (tr[u].son[i]) {
        tr[tr[u].son[i]].fail = tr[tr[u].fail].son[i];
        q.push(tr[u].son[i]);
      } else
        tr[u].son[i] = tr[tr[u].fail].son[i];
    }
  }
}
```

- Python

```
def build():
    for i in range(0, 26):
        if tr[0][i] != 0:
            q.append(tr[0][i])
    while q:
        u = q.pop(0)
        for i in range(0, 26):
            if tr[u][i] != 0:
                fail[tr[u][i]] = tr[fail[u]][i]
                q.append(tr[u][i])
            else:
                tr[u][i] = tr[fail[u]][i]
```

解释

`build` 函数将结点按 BFS 顺序入队，依次求 fail 指针．这里的字典树根结点为 0，我们将根结点的子结点一一入队．若将根结点入队，则在第一次 BFS 的时候，会将根结点儿子的 fail 指针标记为本身．因此我们将根结点的儿子一一入队，而不是将根结点入队．

然后开始 BFS：每次取出队首的结点 u（$\text{fail}(u)$ 在之前的 BFS 过程中已求得），然后遍历字符集（这里是 $0 \sim 25$，对应 `a` $\sim$ `z`，即 u 的各个子结点）：

1. 如果 $\text{trans}(u, c)$ 存在，我们就将 $\text{trans}(u, c)$ 的 fail 指针赋值为 $\text{trans}(\text{fail}(u), c)$．根据之前的描述，我们应该用 `while` 循环，不停地跳 fail 指针，判断是否存在字符 c 对应的结点，然后赋值，但此处通过特殊处理简化了这些代码，将在下文说明；
2. 否则，令 $\text{trans}(u, c)$ 指向 $\text{trans}(\text{fail}(u), c)$ 的状态．

这里的处理是，通过 `else` 语句的代码修改字典树的结构，将不存在的字典树的状态链接到了失配指针的对应状态．在原字典树中，每一个结点代表一个字符串 S，是某个模式串的前缀．而在修改字典树结构后，尽管增加了许多转移关系，但结点（状态）所代表的字符串是不变的．

而 $\text{trans}(S, c)$ 相当于是在 S 后添加一个字符 c 变成另一个状态 S'．如果 S' 存在，说明存在一个模式串的前缀是 S'，否则我们让 $\text{trans}(S, c)$ 指向 $\text{trans}(\text{fail}(S), c)$．由于 $\text{fail}(S)$ 对应的字符串是 S 的后缀，因此 $\text{trans}(\text{fail}(S), c)$ 对应的字符串也是 S' 的后缀．

换言之在 Trie 上跳转的时候，我们只会从 S 跳转到 S'，相当于匹配了一个 S'；但在 AC 自动机上跳转的时候，我们会从 S 跳转到 S' 的后缀，也就是说我们匹配一个字符 c，然后舍弃 S 的部分前缀．舍弃

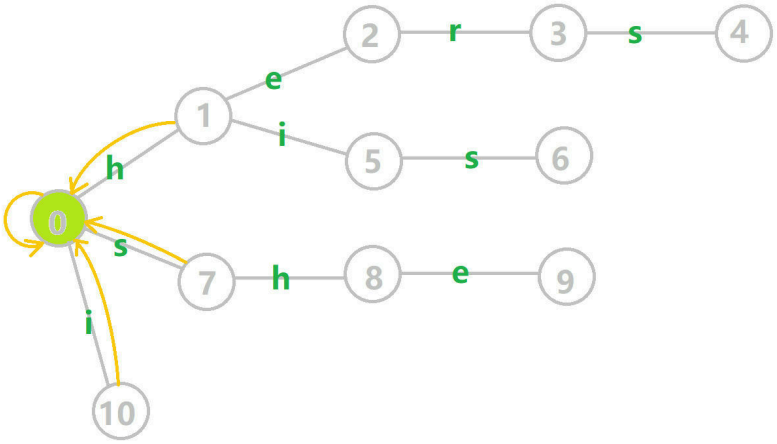
前缀显然是能匹配的．同时如果文本串能匹配 S，显然它也能匹配 S 的后缀，所以 fail 指针同样在舍弃前缀．所谓的 fail 指针其实就是 S 的一个后缀集合．

Trie 的结点的孩子数组 `son` 还有另一种比较简单的理解方式：如果在位置 u 失配，我们会跳转到 $\text{fail}(u)$ 的位置．注意这会导致我们可能沿着 fail 数组跳转多次才能来到下一个能匹配的位置．所以我们可以用 `son` 直接记录下一个能匹配的位置，这样保证了程序的时间复杂度．

此处对字典树结构的修改，可以使得匹配转移更加完善．同时它将 fail 指针跳转的路径做了压缩，使得本来需要跳很多次 fail 指针变成跳一次．

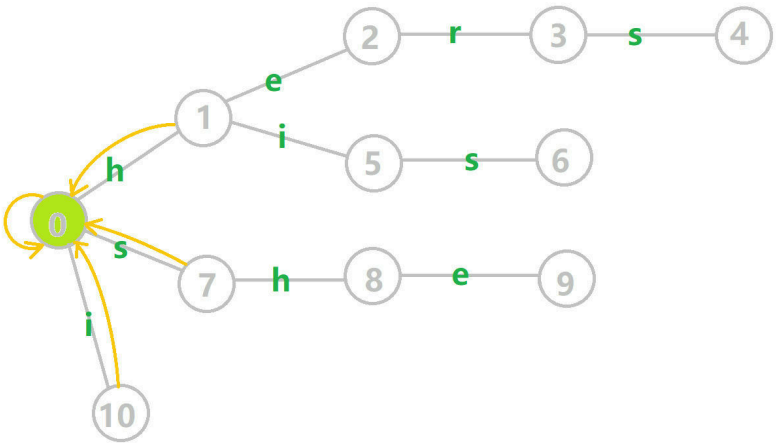
过程

这里依然用若干张 GIF 动图展示构建过程：



1. 蓝色结点：BFS 遍历到的结点 u．
2. 蓝色的边：当前结点下，AC 自动机修改字典树结构连出的边．
3. 黑色的边：AC 自动机修改字典树结构连出的边．
4. 红色的边：当前结点求出的 fail 指针．
5. 黄色的边：fail 指针．
6. 灰色的边：字典树的边．

可以发现，众多交错的黑色边将字典树变成了 **字典图**．图中省略了连向根结点的黑边（否则会更乱）．我们重点分析一下结点 5 遍历时的情况．我们求 $\text{trans}(5, \texttt{s}) = 6$ 的 fail 指针：



本来的策略是找 fail 指针，于是我们跳到 $\text{fail}(5)=10$ 发现没有 s 连出的字典树的边，于是跳到 $\text{fail}(10)=0$，发现有 $\text{trie}(0,\texttt{s})=7$，于是 $\text{fail}(6)=7$；但是有了黑边、蓝边，我们跳到 $\text{fail}(5)=10$ 之后直接走 $\text{trans}(10,\texttt{s})=7$ 就走到 7 号结点了.

这就是 `build` 完成的两件事：构建 fail 指针和建立字典图. 这个字典图也会在查询的时候起到关键作用.

多模式匹配

接下来分析匹配函数 `query`：

实现

- C++

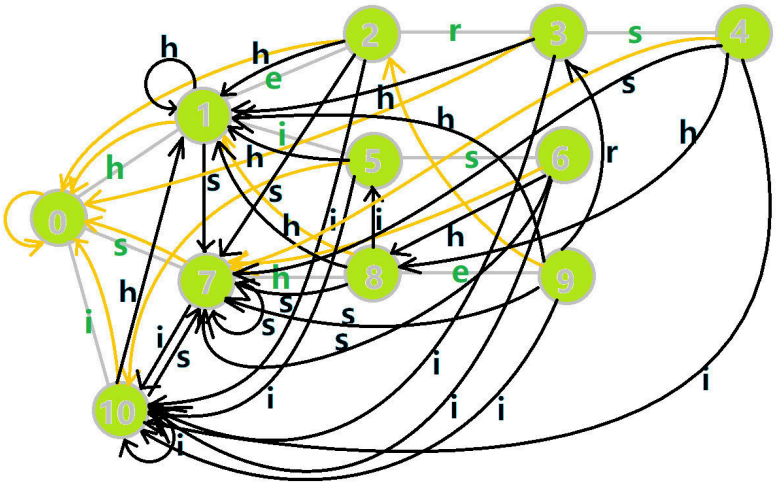
```cpp
int query(const char t[]) {
  int u = 0, res = 0;
  for (int i = 1; t[i]; i++) {
    u = tr[u].son[t[i] - 'a'];
    for (int j = u; j && tr[j].cnt != -1; j = tr[j].fail) {
      res += tr[j].cnt, tr[j].cnt = -1;
    }
  }
  return res;
}
```

- Python

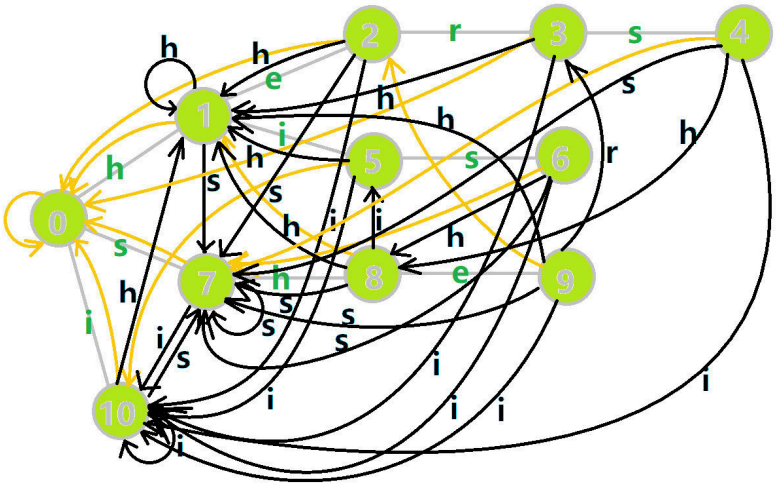
```python
def query(t: str) -> int:
    u, res = 0, 0
    for c in t:
        u = tr[u][c - ord("a")]
        j = u
        while j and e[j] != -1:
            res += e[j]
            e[j] = -1
            j = fail[j]
    return res
```

解释

这里 u 作为字典树上当前匹配到的结点，`res` 即返回的答案. 循环遍历匹配串，u 在字典树上跟踪当前字符. 利用 fail 指针找出所有匹配的模式串，并累加到答案中. 然后将匹配到的串的出现次数清零，这样就不会重复统计同一个串. 在上文中我们分析过，字典树的结构其实就是一个 trans 函数，而构建好这个函数后，在匹配字符串的过程中，我们会舍弃部分前缀达到最低限度的匹配. fail 指针则指向了更多的匹配状态. 最后上一份图. 对于刚才的自动机：



我们从根结点开始尝试匹配 `ushersheishis`，那么 p 的变化将是：



1. 红色结点：p 结点.
2. 粉色箭头：p 在自动机上的跳转.
3. 蓝色的边：成功匹配的模式串.
4. 蓝色结点：示跳 fail 指针时的结点（状态）.

效率优化

题目请参考洛谷 P5357【模板】AC 自动机[1].

因为我们的 AC 自动机中，每次匹配，会一直向 fail 边跳来找到所有的匹配，但是这样的效率较低，在某些题目中会超时.

那么需要如何优化呢？首先需要了解到 fail 指针的一个性质：一个 AC 自动机中，如果只保留 fail 边，那么剩余的图一定是一棵树.

这是显然的，因为 fail 不会成环，且深度一定比现在低，所以得证.

这样 AC 自动机的匹配就可以转化为在 fail 树上的链求和问题，只需要优化一下该部分就可以了.

这里提供两种思路.

拓扑排序优化

观察到时间主要浪费在每次都要跳 fail. 如果我们可以预先记录，最后一并求和，那么效率就会优化.

于是我们按照 fail 树，做一次内向树上的拓扑排序，就能一次性求出所有模式串的出现次数.

`build` 函数在原先的基础上，增加了入度统计一部分，为拓扑排序做准备.

构建

```
void build() {
  queue<int> q;
  for (int i = 0; i < 26; i++)
    if (tr[0].son[i]) q.push(tr[0].son[i]);
  while (!q.empty()) {
    int u = q.front();
    q.pop();
    for (int i = 0; i < 26; i++) {
      if (tr[u].son[i]) {
        tr[tr[u].son[i]].fail = tr[tr[u].fail].son[i];
        tr[tr[tr[u].fail].son[i]].du++;  // 入度计数
        q.push(tr[u].son[i]);
      } else
        tr[u].son[i] = tr[tr[u].fail].son[i];
    }
  }
}
```

然后我们在查询的时候就可以只为找到结点的 `ans` 打上标记，在最后再用拓扑排序求出答案.

查询

```
void query(const char t[]) {
  int u = 0;
  for (int i = 1; t[i]; i++) {
    u = tr[u].son[t[i] - 'a'];
    tr[u].ans++;
  }
}

void topu() {
  queue<int> q;
  for (int i = 0; i <= tot; i++)
    if (tr[i].du == 0) q.push(i);
  while (!q.empty()) {
    int u = q.front();
    q.pop();
    ans[tr[u].idx] = tr[u].ans;
    int v = tr[u].fail;
    tr[v].ans += tr[u].ans;
    if (!--tr[v].du) q.push(v);
  }
}
```

最后是主函数:

主函数

```cpp
int main() {
  // do_something();
  AC::build();
  scanf("%s", s + 1);
  AC::query(s);
  AC::topu();
  for (int i = 1; i <= n; i++) printf("%d\n", AC::ans[idx[i]]);
  // do_another_thing();
}
```

模板题 Luogu P5357「模板」AC 自动机[2] 拓扑排序优化参考代码

```cpp
#include <cstdio>
#include <cstring>
#include <queue>
using namespace std;

constexpr int N = 2e5 + 6;
constexpr int LEN = 2e6 + 6;
constexpr int SIZE = 2e5 + 6;

int n;

namespace AC {
struct Node {
  int son[26];  // 子结点
  int ans;      // 匹配计数
  int fail;     // fail 指针
  int du;       // 入度
  int idx;

  void init() {  // 结点初始化
    memset(son, 0, sizeof(son));
    ans = fail = idx = 0;
  }
} tr[SIZE];

int tot;  // 结点总数
int ans[N], pidx;

void init() {
  tot = pidx = 0;
  tr[0].init();
}

void insert(char s[], int &idx) {
  int u = 0;
  for (int i = 1; s[i]; i++) {
    int &son = tr[u].son[s[i] - 'a'];  // 下一个子结点的引用
    if (!son) son = ++tot, tr[son].init();  // 如果没有则插入新结点，并初始化
    u = son;                                // 从下一个结点继续
```

```
  }
  // 由于有可能出现相同的模式串，需要将相同的映射到同一个编号
  if (!tr[u].idx) tr[u].idx = ++pidx;  // 第一次出现，新增编号
  idx = tr[u].idx;  // 这个模式串的编号对应这个结点的编号
}

void build() {
  queue<int> q;
  for (int i = 0; i < 26; i++)
    if (tr[0].son[i]) q.push(tr[0].son[i]);
  while (!q.empty()) {
    int u = q.front();
    q.pop();
    for (int i = 0; i < 26; i++) {
      if (tr[u].son[i]) {                                  // 存在对应子结点
        tr[tr[u].son[i]].fail = tr[tr[u].fail].son[i];  // 只用跳一次 fail 指针
        tr[tr[tr[u].fail].son[i]].du++;                    // 入度计数
        q.push(tr[u].son[i]);                              // 并加入队列
      } else
        tr[u].son[i] =
            tr[tr[u].fail]
                .son[i];  // 将不存在的字典树的状态链接到了失配指针的对应状态
    }
  }
}

void query(char t[]) {
  int u = 0;
  for (int i = 1; t[i]; i++) {
    u = tr[u].son[t[i] - 'a'];  // 转移
    tr[u].ans++;
  }
}

void topu() {
  queue<int> q;
  for (int i = 0; i <= tot; i++)
    if (tr[i].du == 0) q.push(i);
  while (!q.empty()) {
    int u = q.front();
    q.pop();
    ans[tr[u].idx] = tr[u].ans;
    int v = tr[u].fail;
    tr[v].ans += tr[u].ans;
    if (!--tr[v].du) q.push(v);
  }
}
}  // namespace AC

char s[LEN];
int idx[N];

```

```
int main() {
  AC::init();
  scanf("%d", &n);
  for (int i = 1; i <= n; i++) {
    scanf("%s", s + 1);
    AC::insert(s, idx[i]);
    AC::ans[i] = 0;
  }
  AC::build();
  scanf("%s", s + 1);
  AC::query(s);
  AC::topu();
  for (int i = 1; i <= n; i++) {
    printf("%d\n", AC::ans[idx[i]]);
  }
  return 0;
}

```

DFS 优化

和拓扑排序的思路接近，不过我们使用 DFS 来代替拓扑排序．其实这两种方法本质上是相同的，都是将 fail 树的子树求和．

完整代码请见总结模板 3．

AC 自动机上 DP

这部分将以 P2292 [HNOI2004] L 语言[3] 为例题讲解．

不难想到一个朴素的思路：建立 AC 自动机，在 AC 自动机上对于所有 fail 指针的子串转移，最后取最大值得到答案．

主要代码如下．若不熟悉代码中的类型定义，可以先看末尾的完整代码：

查询部分主要代码

```
int query(const char t[]) {
  int u = 0, len = strlen(t + 1);
  for (int i = 1; i <= len; i++) dp[i] = 0;
  for (int i = 1; i <= len; i++) {
    u = tr[u].son[t[i] - 'a'];
    for (int j = u; j; j = tr[j].fail) {
      if (tr[j].idx && (dp[i - tr[j].depth] || i - tr[j].depth == 0)) {
        dp[i] = dp[i - tr[j].depth] + tr[j].depth;
      }
    }
  }
  int ans = 0;
  for (int i = 1; i <= len; i++) ans = std::max(ans, dp[i]);
  return ans;
}
```

但是这样的思路复杂度不是线性（因为要跳每个结点的 fail），会在第二个子任务中超时，所以我们需要进行优化.

我们再看看题目的特殊性质，我们发现所有单词的长度只有 20，所以可以想到状态压缩优化.

我们发现，目前的时间瓶颈主要在跳 fail 这一步，如果我们可以将这一步优化到 $O(1)$，就可以保证整个问题在严格线性的时间内被解出.

我们可以将前 20 位字母中，可能的子串长度存下来，并压缩到状态中，存在每个子结点中.

那么我们在 `build` 的时候就可以这么写：

构建 fail 指针

```
void build() {
  queue<int> q;
  for (int i = 0; i < 26; i++)
    if (tr[0].son[i]) {
      q.push(tr[0].son[i]);
      tr[tr[0].son[i]].depth = 1;
    }
  while (!q.empty()) {
    int u = q.front();
    q.pop();
    int v = tr[u].fail;
    // 对状态的更新在这里
    tr[u].stat = tr[v].stat;
    if (tr[u].idx) tr[u].stat |= 1 << tr[u].depth;
    for (int i = 0; i < 26; i++) {
      if (tr[u].son[i]) {
        tr[tr[u].son[i]].fail = tr[tr[u].fail].son[i];
        tr[tr[u].son[i]].depth = tr[u].depth + 1;  // 记录深度
        q.push(tr[u].son[i]);
      } else
        tr[u].son[i] = tr[tr[u].fail].son[i];
    }
  }
}
```

然后查询时就可以去掉跳 fail 的循环，将代码简化如下：

查询

```
int query(const char t[]) {
  int u = 0, mx = 0;
  unsigned st = 1;
  for (int i = 1; t[i]; i++) {
    u = tr[u].son[t[i] - 'a'];
    st <<= 1;  // 往下跳了一位每一位的长度都+1
    if (tr[u].stat & st) st |= 1, mx = i;
  }
  return mx;
}
```

我们的 `tr[u].stat` 维护的是从结点 u 开始，整条 fail 链上的长度集（因为长度集小于 32 所以不影响），而 `st` 则维护的是查询字符串走到现在，前 32 位（因为状态压缩自然溢出）的长度集.

& 运算后结果不为 0，则代表两个长度集的交集非空，我们此时就找到了一个匹配.

P2292 [HNOI2004] L 语言[4] 完整代码

```cpp
#include <cstdio>
#include <cstring>
#include <queue>
using namespace std;

constexpr int N = 20 + 6, M = 50 + 6;
constexpr int LEN = 2e6 + 6;
constexpr int SIZE = 450 + 6;

int n, m;

namespace AC {
struct Node {
  int son[26];
  int fail;
  int idx;
  int depth;
  unsigned stat;

  void init() {
    memset(son, 0, sizeof(son));
    fail = idx = depth = 0;
  }
} tr[SIZE];

int tot;

void init() {
  tot = 0;
  tr[0].init();
}

void insert(char s[], int idx) {
  int u = 0;
  for (int i = 1; s[i]; i++) {
    int &son = tr[u].son[s[i] - 'a'];
    if (!son) son = ++tot, tr[son].init();
    u = son;
  }
  tr[u].idx = idx;
}

void build() {
  queue<int> q;
  for (int i = 0; i < 26; i++)
    if (tr[0].son[i]) {
      q.push(tr[0].son[i]);
```

```
48        tr[tr[0].son[i]].depth = 1;
49      }
50    while (!q.empty()) {
51      int u = q.front();
52      q.pop();
53      int v = tr[u].fail;
54      // 对状态的更新在这里
55      tr[u].stat = tr[v].stat;
56      if (tr[u].idx) tr[u].stat |= 1 << tr[u].depth;
57      for (int i = 0; i < 26; i++) {
58        if (tr[u].son[i]) {
59          tr[tr[u].son[i]].fail = tr[tr[u].fail].son[i];
60          tr[tr[u].son[i]].depth = tr[u].depth + 1;  // 记录深度
61          q.push(tr[u].son[i]);
62        } else
63          tr[u].son[i] = tr[tr[u].fail].son[i];
64      }
65    }
66 }
67
68 int query(char t[]) {
69    int u = 0, mx = 0;
70    unsigned st = 1;
71    for (int i = 1; t[i]; i++) {
72      u = tr[u].son[t[i] - 'a'];
73      st <<= 1;
74      if (tr[u].stat & st) st |= 1, mx = i;
75    }
76    return mx;
77 }
78 }  // namespace AC
79
80 char s[LEN];
81
82 int main() {
83    AC::init();
84    scanf("%d%d", &n, &m);
85    for (int i = 1; i <= n; i++) {
86      scanf("%s", s + 1);
87      AC::insert(s, i);
88    }
89    AC::build();
90    for (int i = 1; i <= m; i++) {
91      scanf("%s", s + 1);
92      printf("%d\n", AC::query(s));
93    }
94    return 0;
95 }
96
```

总结

时间复杂度：定义 $|s_i|$ 是模板串的长度，$|S|$ 是文本串的长度，$|\Sigma|$ 是字符集的大小（常数，一般为 26）．如果连了 trie 图，时间复杂度就是 $O(\sum|s_i|+n|\Sigma|+|S|)$，其中 n 是 AC 自动机中结点的数目，并且最大可以达到 $O(\sum|s_i|)$．如果不连 trie 图，并且在构建 fail 指针的时候避免遍历到空儿子，时间复杂度就是 $O(\sum|s_i|+|S|)$．

模板题 Luogu P3808 AC 自动机（简单版）[5] 参考代码

```cpp
#include <cstdio>
#include <cstring>
#include <queue>
using namespace std;

constexpr int N = 1e6 + 6;
constexpr int LEN = 1e6 + 6;
constexpr int SIZE = 1e6 + 6;

int n;

namespace AC {
struct Node {
  int son[26];  // 子结点
  int cnt;      // 尾为该结点的串的个数
  int fail;     // fail 指针

  void init() {  // 结点初始化
    memset(son, 0, sizeof(son));
    cnt = fail = 0;
  }
} tr[SIZE];

int tot;  // 结点总数

void init() {
  tot = 0;
  tr[0].init();
}

void insert(char s[]) {
  int u = 0;
  for (int i = 1; s[i]; i++) {
    int &son = tr[u].son[s[i] - 'a'];  // 下一个子结点的引用
    if (!son) son = ++tot, tr[son].init();  // 如果没有则插入新结点，并初始化
    u = son;                                // 从下一个结点继续
  }
  tr[u].cnt++;
}

void build() {
  queue<int> q;
  for (int i = 0; i < 26; i++)
    if (tr[0].son[i]) q.push(tr[0].son[i]);
```

```
  while (!q.empty()) {
    int u = q.front();
    q.pop();
    for (int i = 0; i < 26; i++) {
      if (tr[u].son[i]) {                                  // 存在对应子结点
        tr[tr[u].son[i]].fail = tr[tr[u].fail].son[i];  // 只用跳一次 fail 指针
        q.push(tr[u].son[i]);                              // 并加入队列
      } else
        tr[u].son[i] =
            tr[tr[u].fail]
                .son[i];  // 将不存在的字典树的状态链接到了失配指针的对应状态
    }
  }
}

int query(char t[]) {
  int u = 0, res = 0;
  for (int i = 1; t[i]; i++) {
    u = tr[u].son[t[i] - 'a'];  // 转移
    for (int j = u; j && tr[j].cnt != -1; j = tr[j].fail) {
      res += tr[j].cnt, tr[j].cnt = -1;
    }
  }
  return res;
}
}  // namespace AC

char s[LEN];

int main() {
  AC::init();
  scanf("%d", &n);
  for (int i = 1; i <= n; i++) {
    scanf("%s", s + 1);
    AC::insert(s);
  }
  AC::build();
  scanf("%s", s + 1);
  printf("%d", AC::query(s));
  return 0;
}

```

模板题 Luogu P3796 AC 自动机（简单版 II）[6] 参考代码

```
#include <cstdio>
#include <cstring>
#include <queue>
using namespace std;

constexpr int N = 150 + 6;
```

```
constexpr int LEN = 1e6 + 6;
constexpr int SIZE = N * 70 + 6;

int n;

namespace AC {
struct Node {
  int son[26];
  int fail;
  int idx;

  void init() {
    memset(son, 0, sizeof(son));
    idx = fail = 0;
  }
} tr[SIZE];

int tot;

void init() {
  tot = 0;
  tr[0].init();
}

void insert(char s[], int idx) {  // 将第 idx 个字符串 s 插入
  int u = 0;
  for (int i = 1; s[i]; i++) {
    int &son = tr[u].son[s[i] - 'a'];
    if (!son) son = ++tot, tr[son].init();
    u = son;
  }
  tr[u].idx = idx;
}

void build() {
  queue<int> q;
  for (int i = 0; i < 26; i++)
    if (tr[0].son[i]) q.push(tr[0].son[i]);
  while (!q.empty()) {
    int u = q.front();
    q.pop();
    for (int i = 0; i < 26; i++) {
      if (tr[u].son[i]) {
        tr[tr[u].son[i]].fail = tr[tr[u].fail].son[i];
        q.push(tr[u].son[i]);
      } else
        tr[u].son[i] = tr[tr[u].fail].son[i];
    }
  }
}

int query(char t[], int cnt[]) {
```

```
  int u = 0, res = 0;
  for (int i = 1; t[i]; i++) {
    u = tr[u].son[t[i] - 'a'];
    for (int j = u; j; j = tr[j].fail)
      ++cnt[tr[j].idx];  // 统计每个字符串出现的次数
  }
  for (int i = 0; i <= tot; ++i)
    if (tr[i].idx) res = max(res, cnt[tr[i].idx]);
  return res;
}
}  // namespace AC

char s[N][75], t[LEN];
int cnt[N];  // 每一个字符串出现的次数

int main() {
  while (scanf("%d", &n) != EOF && n != 0) {
    AC::init();
    for (int i = 1; i <= n; i++) {
      scanf("%s", s[i] + 1);
      AC::insert(s[i], i);
      cnt[i] = 0;
    }
    AC::build();
    scanf("%s", t + 1);
    int x = AC::query(t, cnt);
    printf("%d\n", x);
    for (int i = 1; i <= n; i++)
      if (cnt[i] == x) printf("%s\n", s[i] + 1);
  }
  return 0;
}

```

模板题 Luogu P5357「模板」AC 自动机[7] DFS 优化参考代码

```
#include <cstdio>
#include <cstring>
#include <queue>
#include <vector>
using namespace std;

constexpr int N = 2e5 + 6;
constexpr int LEN = 2e6 + 6;
constexpr int SIZE = 2e5 + 6;

int n;

namespace AC {
struct Node {
  int son[26];
```

```
16   int ans;
17   int fail;
18   int idx;
19
20   void init() {
21     memset(son, 0, sizeof(son));
22     ans = idx = 0;
23   }
24 } tr[SIZE];
25
26 int tot;
27 int ans[N], pidx;
28
29 vector<int> g[SIZE];  // fail 树
30
31 void init() {
32   tot = pidx = 0;
33   tr[0].init();
34 }
35
36 void insert(char s[], int &idx) {
37   int u = 0;
38   for (int i = 1; s[i]; i++) {
39     int &son = tr[u].son[s[i] - 'a'];
40     if (!son) son = ++tot, tr[son].init();
41     u = son;
42   }
43   // 由于有可能出现相同的模式串，需要将相同的映射到同一个编号
44   if (!tr[u].idx) tr[u].idx = ++pidx;  // 第一次出现，新增编号
45   idx = tr[u].idx;  // 这个模式串的编号对应这个结点的编号
46 }
47
48 void build() {
49   queue<int> q;
50   for (int i = 0; i < 26; i++)
51     if (tr[0].son[i]) {
52       q.push(tr[0].son[i]);
53       g[0].push_back(tr[0].son[i]);  // 不要忘记这里的 fail
54     }
55   while (!q.empty()) {
56     int u = q.front();
57     q.pop();
58     for (int i = 0; i < 26; i++) {
59       if (tr[u].son[i]) {
60         tr[tr[u].son[i]].fail = tr[tr[u].fail].son[i];
61         g[tr[tr[u].fail].son[i]].push_back(tr[u].son[i]);  // 记录 fail 树
62         q.push(tr[u].son[i]);
63       } else
64         tr[u].son[i] = tr[tr[u].fail].son[i];
65     }
66   }
67 }
```

```
68
69 void query(char t[]) {
70   int u = 0;
71   for (int i = 1; t[i]; i++) {
72     u = tr[u].son[t[i] - 'a'];
73     tr[u].ans++;
74   }
75 }
76
77 void dfs(int u) {
78   for (int v : g[u]) {
79     dfs(v);
80     tr[u].ans += tr[v].ans;
81   }
82   ans[tr[u].idx] = tr[u].ans;
83 }
84 }  // namespace AC
85
86 char s[LEN];
87 int idx[N];
88
89 int main() {
90   AC::init();
91   scanf("%d", &n);
92   for (int i = 1; i <= n; i++) {
93     scanf("%s", s + 1);
94     AC::insert(s, idx[i]);
95     AC::ans[i] = 0;
96   }
97   AC::build();
98   scanf("%s", s + 1);
99   AC::query(s);
100  AC::dfs(0);
101  for (int i = 1; i <= n; i++) {
102    printf("%d\n", AC::ans[idx[i]]);
103  }
104  return 0;
105 }
106
```

外部参考资料

[1] P5357【模板】AC 自动机

[2] Luogu P5357「模板」AC 自动机

[3] P2292 [HNOI2004] L 语言

[4] P2292 [HNOI2004] L 语言

[5] Luogu P3808 AC 自动机（简单版）

[6] Luogu P3796 AC 自动机（简单版 II）

[7] Luogu P5357「模板」AC 自动机

8.11 后缀数组 (SA)

8.11.1 后缀数组简介

一些约定

字符串相关的定义请参考 字符串基础→8.2@p890.

字符串下标从 1 开始.

字符串 s 的长度为 n.

" 后缀 i" 代指以第 i 个字符开头的后缀，存储时用 i 代表字符串 s 的后缀 $s[i...n]$.

后缀数组是什么?

后缀数组（Suffix Array）主要关系到两个数组：sa 和 rk.

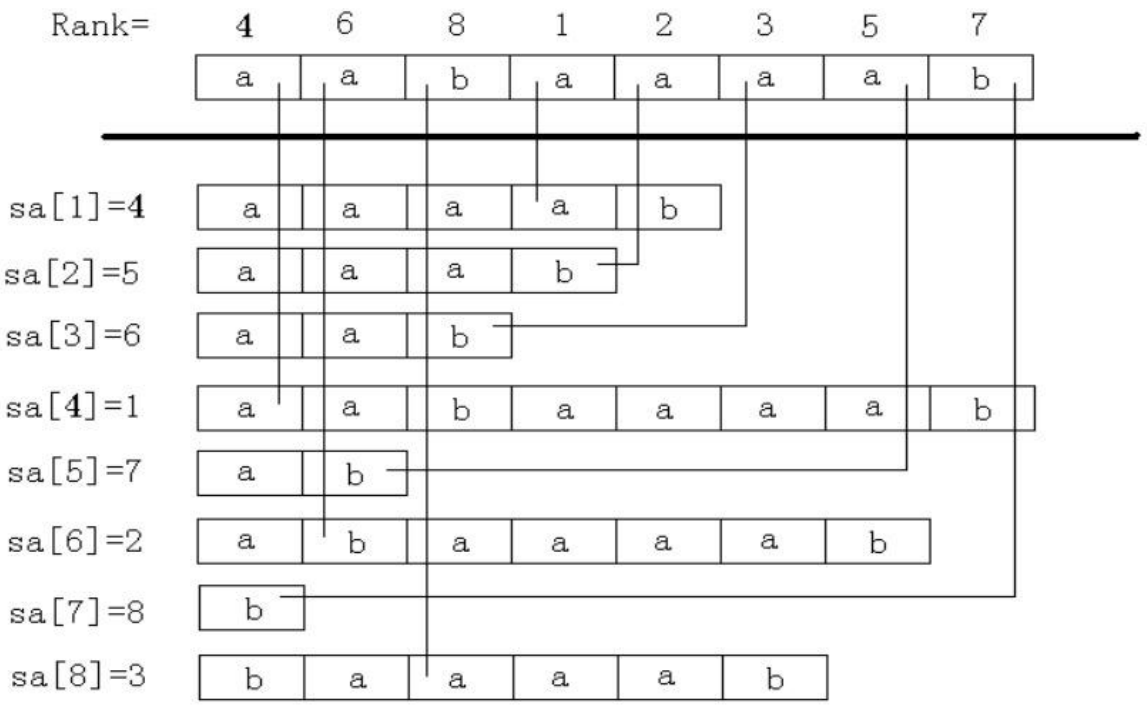
其中, $sa[i]$ 表示将所有后缀排序后第 i 小的后缀的编号, 也是所说的后缀数组, 后文也称编号数组 sa;

$rk[i]$ 表示后缀 i 的排名，是重要的辅助数组，后文也称排名数组 rk.

这两个数组满足性质：$sa[rk[i]] = rk[sa[i]] = i$.

解释

后缀数组示例:



[1]

后缀数组怎么求?

O(n^2logn) 做法

相信这个做法大家还是能自己想到的：将盛有全部后缀字符串的数组进行 `sort` 排序，由于排序进行 $O(n \log n)$ 次字符串比较，每次字符串比较要 $O(n)$ 次字符比较，所以这个排序是 $O(n^2 \log n)$ 的时间复杂度.

O(nlog^2n) 做法

这个做法要用到倍增的思想.

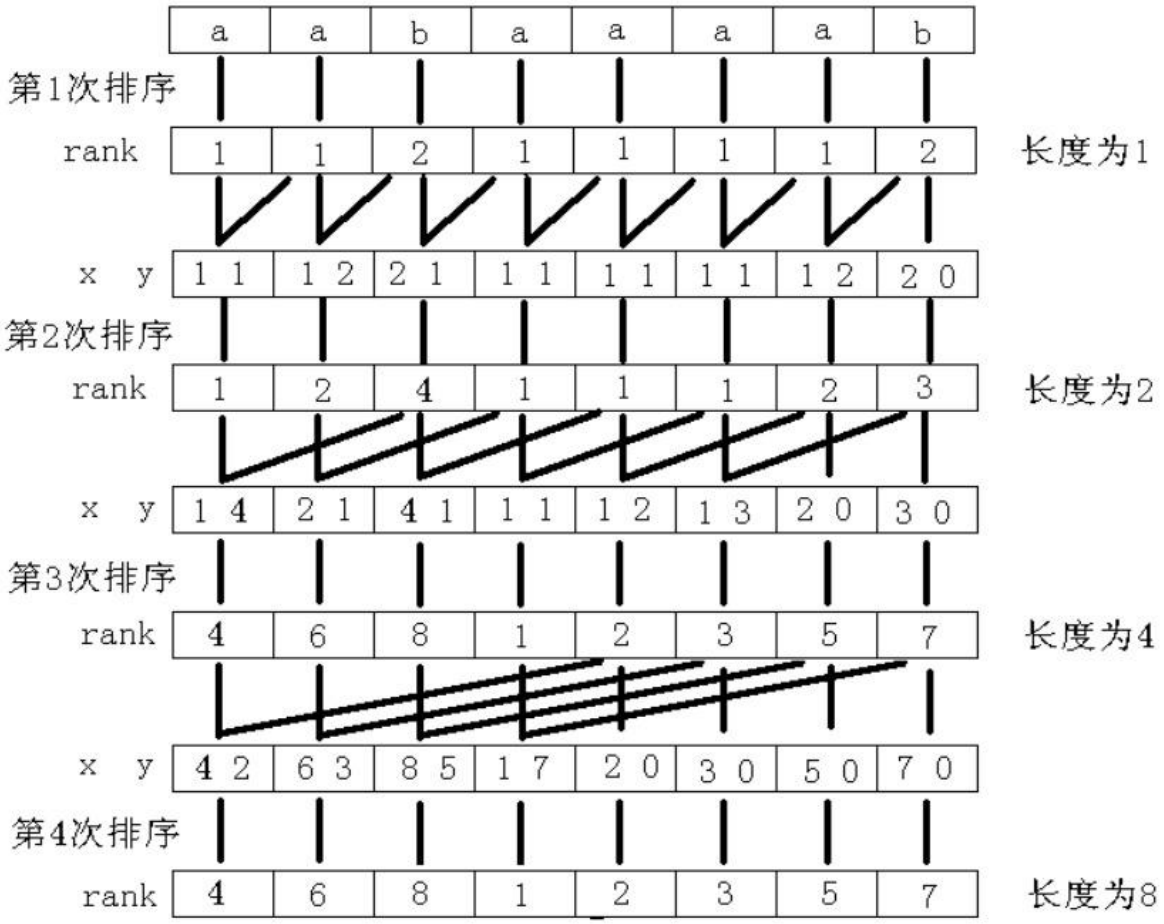
首先对字符串 s 的所有长度为 1 的子串，即每个字符进行排序，得到排序后的编号数组 sa_1 和排名数组 rk_1.

倍增过程：

1. 用两个长度为 1 的子串的排名，即 $rk_1[i]$ 和 $rk_1[i+1]$，作为排序的第一第二关键字，就可以对字符串 s 的每个长度为 2 的子串：$\{s[i...\min(i+1,n)] \mid i \in [1,n]\}$ 进行排序，得到 sa_2 和 rk_2；
2. 之后用两个长度为 2 的子串的排名，即 $rk_2[i]$ 和 $rk_2[i+2]$，作为排序的第一第二关键字，就可以对字符串 s 的每个长度为 4 的子串：$\{s[i...\min(i+3,n)] \mid i \in [1,n]\}$ 进行排序，得到 sa_4 和 rk_4；
3. 以此倍增，用长度为 $w/2$ 的子串的排名，即 $rk_{w/2}[i]$ 和 $rk_{w/2}[i+w/2]$，作为排序的第一第二关键字，就可以对字符串 s 的每个长度为 w 的子串 $s[i...\min(i+w-1,n)]$ 进行排序，得到 sa_w 和 rk_w. 其中，类似字母序排序规则，当 $i+w>n$ 时，$rk_w[i+w]$ 视为无穷小；
4. $rk_w[i]$ 即是子串 $s[i...i+w-1]$ 的排名，这样当 $w \geqslant n$ 时，得到的编号数组 sa_w，也就是我们需要的后缀数组.

过程

倍增排序示意图：



[2]

显然倍增的过程是 $O(\log n)$，而每次倍增用 `sort` 对子串进行排序是 $O(n \log n)$，而每次子串的比较花费 2 次字符比较；

除此之外，每次倍增在 `sort` 排序完后，还有额外的 $O(n)$ 时间复杂度的，更新 rk 的操作，但是相对于 $O(n \log n)$ 被忽略不计；

所以这个算法的时间复杂度就是 $O(n \log^2 n)$.

实现

```cpp
#include <algorithm>
#include <cstdio>
#include <cstring>
#include <iostream>

using namespace std;

constexpr int N = 1000010;

char s[N];
int n, w, sa[N], rk[N << 1], oldrk[N << 1];

// 为了防止访问 rk[i+w] 导致数组越界，开两倍数组.
// 当然也可以在访问前判断是否越界，但直接开两倍数组方便一些.

int main() {
  int i, p;

  scanf("%s", s + 1);
  n = strlen(s + 1);
  for (i = 1; i <= n; ++i) sa[i] = i, rk[i] = s[i];

  for (w = 1; w < n; w <<= 1) {
    sort(sa + 1, sa + n + 1, [](int x, int y) {
      return rk[x] == rk[y] ? rk[x + w] < rk[y + w] : rk[x] < rk[y];
    });  // 这里用到了 lambda
    memcpy(oldrk, rk, sizeof(rk));
    // 由于计算 rk 的时候原来的 rk 会被覆盖，要先复制一份
    // 若两个子串相同，它们对应的 rk 也需要相同，所以要去重
    for (p = 0, i = 1; i <= n; ++i) {
      if (oldrk[sa[i]] == oldrk[sa[i - 1]] &&
          oldrk[sa[i] + w] == oldrk[sa[i - 1] + w]) {
        rk[sa[i]] = p;
      } else {
        rk[sa[i]] = ++p;
      }
    }
  }

  for (i = 1; i <= n; ++i) printf("%d ", sa[i]);

  return 0;
}
```

O(nlogn) 做法

在刚刚的 $O(n \log^2 n)$ 做法中，单次排序是 $O(n \log n)$ 的，如果能 $O(n)$ 排序，就能 $O(n \log n)$ 计算后缀数组了.

前置知识：计数排序→5.7.5@p574，基数排序→5.7.6@p578.

由于计算后缀数组的过程中排序的关键字是排名，值域为 $O(n)$，并且是一个双关键字的排序，可以使用基数排序优化至 $O(n)$.

实现

```cpp
#include <algorithm>
#include <cstdio>
#include <cstring>
#include <iostream>

using namespace std;

constexpr int N = 1000010;

char s[N];
int n, sa[N], rk[N << 1], oldrk[N << 1], id[N], cnt[N];

int main() {
  int i, m, p, w;

  scanf("%s", s + 1);
  n = strlen(s + 1);
  m = 127;
  for (i = 1; i <= n; ++i) ++cnt[rk[i] = s[i]];
  for (i = 1; i <= m; ++i) cnt[i] += cnt[i - 1];
  for (i = n; i >= 1; --i) sa[cnt[rk[i]]--] = i;
  memcpy(oldrk + 1, rk + 1, n * sizeof(int));
  for (p = 0, i = 1; i <= n; ++i) {
    if (oldrk[sa[i]] == oldrk[sa[i - 1]]) {
      rk[sa[i]] = p;
    } else {
      rk[sa[i]] = ++p;
    }
  }

  for (w = 1; w < n; w <<= 1, m = n) {
    // 对第二关键字：id[i] + w进行计数排序
    memset(cnt, 0, sizeof(cnt));
    memcpy(id + 1, sa + 1,
           n * sizeof(int));  // id保存一份儿sa的拷贝，实质上就相当于oldsa
    for (i = 1; i <= n; ++i) ++cnt[rk[id[i] + w]];
    for (i = 1; i <= m; ++i) cnt[i] += cnt[i - 1];
    for (i = n; i >= 1; --i) sa[cnt[rk[id[i] + w]]--] = id[i];

    // 对第一关键字：id[i]进行计数排序
    memset(cnt, 0, sizeof(cnt));
    memcpy(id + 1, sa + 1, n * sizeof(int));
    for (i = 1; i <= n; ++i) ++cnt[rk[id[i]]];
    for (i = 1; i <= m; ++i) cnt[i] += cnt[i - 1];
    for (i = n; i >= 1; --i) sa[cnt[rk[id[i]]]--] = id[i];

    memcpy(oldrk + 1, rk + 1, n * sizeof(int));
```

```
48      for (p = 0, i = 1; i <= n; ++i) {
49        if (oldrk[sa[i]] == oldrk[sa[i - 1]] &&
50            oldrk[sa[i] + w] == oldrk[sa[i - 1] + w]) {
51          rk[sa[i]] = p;
52        } else {
53          rk[sa[i]] = ++p;
54        }
55      }
56    }
57
58    for (i = 1; i <= n; ++i) printf("%d ", sa[i]);
59
60    return 0;
61  }
```

一些常数优化

如果你把上面那份代码交到 LOJ #111: 后缀排序[3] 上：

测试点 #1	Accepted	得分: 100	用时: 20 ms	内存: 11900 KiB
测试点 #2	Accepted	得分: 100	用时: 27 ms	内存: 12004 KiB
测试点 #3	Accepted	得分: 100	用时: 27 ms	内存: 11900 KiB
测试点 #4	Accepted	得分: 100	用时: 37 ms	内存: 12112 KiB
测试点 #5	Accepted	得分: 100	用时: 36 ms	内存: 12112 KiB
测试点 #6	Accepted	得分: 100	用时: 103 ms	内存: 13200 KiB
测试点 #7	Accepted	得分: 100	用时: 111 ms	内存: 13248 KiB
测试点 #8	Time Limit Exceeded	得分: 0	用时: 4018 ms	内存: 24576 KiB
测试点 #9	Time Limit Exceeded	得分: 0	用时: 4014 ms	内存: 24576 KiB
测试点 #10	Time Limit Exceeded	得分: 0	用时: 4018 ms	内存: 24576 KiB

这是因为，上面那份代码的常数的确很大.

第二关键字无需计数排序

思考一下第二关键字排序的实质，其实就是把超出字符串范围（即 $sa[i]+w>n$）的 $sa[i]$ 放到 sa 数组头部，然后把剩下的依原顺序放入：

```
int cur = 0;
for (int i = n - w + 1; i <= n; i++) id[++cur] = i;
for (int i = 1; i <= n; i++)
  if (sa[i] > w) id[++cur] = sa[i] - w;
```

优化计数排序的值域

每次对 rk 进行更新之后，我们都计算了一个 p，这个 p 即是 rk 的值域，将值域改成它即可.

若排名都不相同可直接生成后缀数组

考虑新的 rk 数组，若其值域为 $[1,n]$ 那么每个排名都不同，此时无需再排序.

实现

```cpp
#include <algorithm>
#include <cstdio>
#include <cstring>
#include <iostream>

using namespace std;

constexpr int N = 1000010;

char s[N];
int n;
int m, p, rk[N * 2], oldrk[N], sa[N * 2], id[N], cnt[N];

int main() {
  scanf("%s", s + 1);
  n = strlen(s + 1);
  m = 128;

  for (int i = 1; i <= n; i++) cnt[rk[i] = s[i]]++;
  for (int i = 1; i <= m; i++) cnt[i] += cnt[i - 1];
  for (int i = n; i >= 1; i--) sa[cnt[rk[i]]--] = i;

  for (int w = 1;; w <<= 1, m = p) {  // m = p 即为值域优化
    int cur = 0;
    for (int i = n - w + 1; i <= n; i++) id[++cur] = i;
    for (int i = 1; i <= n; i++)
      if (sa[i] > w) id[++cur] = sa[i] - w;

    memset(cnt, 0, sizeof(cnt));
    for (int i = 1; i <= n; i++) cnt[rk[i]]++;
    for (int i = 1; i <= m; i++) cnt[i] += cnt[i - 1];
    for (int i = n; i >= 1; i--) sa[cnt[rk[id[i]]]--] = id[i];

    p = 0;
    memcpy(oldrk, rk, sizeof(oldrk));
    for (int i = 1; i <= n; i++) {
      if (oldrk[sa[i]] == oldrk[sa[i - 1]] &&
          oldrk[sa[i] + w] == oldrk[sa[i - 1] + w])
        rk[sa[i]] = p;
      else
        rk[sa[i]] = ++p;
    }

    if (p == n) break;  // p = n 时无需再排序
  }

  for (int i = 1; i <= n; i++) printf("%d ", sa[i]);

  return 0;
}
```

O(n) 做法

在一般的题目中，常数较小的倍增求后缀数组是完全够用的，求后缀数组以外的部分也经常有 $O(n \log n)$ 的复杂度，倍增求解后缀数组不会成为瓶颈.

但如果遇到特殊题目、时限较紧的题目，或者是你想追求更短的用时，就需要学习 $O(n)$ 求后缀数组的方法.

SA-IS

可以参考 诱导排序与 SA-IS 算法[4]，另外它的 评论页面[5] 也有参考价值.

DC3

可以参考[2009] 后缀数组——处理字符串的有力工具 by. 罗穗骞[6].

后缀数组的应用

寻找最小的循环移动位置

将字符串 S 复制一份变成 SS 就转化成了后缀排序问题.

例题：「JSOI2007」字符加密[7].

在字符串中找子串

任务是在线地在主串 T 中寻找模式串 S. 在线的意思是，我们已经预先知道主串 T，但是当且仅当询问时才知道模式串 S. 我们可以先构造出 T 的后缀数组，然后查找子串 S. 若子串 S 在 T 中出现，它必定是 T 的一些后缀的前缀. 因为我们已经将所有后缀排序了，我们可以通过在 p 数组中二分 S 来实现. 比较子串 S 和当前后缀的时间复杂度为 $O(|S|)$，因此找子串的时间复杂度为 $O(|S|\log|T|)$. 注意，如果该子串在 T 中出现了多次，每次出现都是在 p 数组中相邻的. 因此出现次数可以通过再次二分找到，输出每次出现的位置也很轻松.

从字符串首尾取字符最小化字典序

例题：「USACO07DEC」Best Cow Line[8].

题意：给你一个字符串，每次从首或尾取一个字符组成字符串，问所有能够组成的字符串中字典序最小的一个.

题解

暴力做法就是每次最坏 $O(n)$ 地判断当前应该取首还是尾（即比较取首得到的字符串与取尾得到的反串的大小），只需优化这一判断过程即可.

由于需要在原串后缀与反串后缀构成的集合内比较大小，可以将反串拼接在原串后，并在中间加上一个没出现过的字符（如 #，代码中可以直接使用空字符），求后缀数组，即可 $O(1)$ 完成这一判断.

参考代码

```cpp
#include <cctype>
#include <cstring>
#include <iostream>
using namespace std;

constexpr int N = 1000010;

char s[N];
int n, sa[N], id[N], oldrk[N * 2], rk[N * 2], px[N], cnt[N];
```

```cpp

bool cmp(int x, int y, int w) {
  return oldrk[x] == oldrk[y] && oldrk[x + w] == oldrk[y + w];
}

int main() {
  int i, w, m = 200, p, l = 1, r, tot = 0;

  cin >> n;
  r = n;

  for (i = 1; i <= n; ++i)
    while (cin >> s[i], !isalpha(s[i]));
  for (i = 1; i <= n; ++i)
    rk[i] = rk[2 * n + 2 - i] = s[i];  // 拼接正反两个字符串，中间空出一个字符

  n = 2 * n + 1;
  // 求后缀数组
  for (i = 1; i <= n; ++i) ++cnt[rk[i]];
  for (i = 1; i <= m; ++i) cnt[i] += cnt[i - 1];
  for (i = n; i >= 1; --i) sa[cnt[rk[i]]--] = i;

  for (w = 1; w < n; w *= 2, m = p) {  // m=p 就是优化计数排序值域
    for (p = 0, i = n; i > n - w; --i) id[++p] = i;
    for (i = 1; i <= n; ++i)
      if (sa[i] > w) id[++p] = sa[i] - w;
    memset(cnt, 0, sizeof(cnt));
    for (i = 1; i <= n; ++i) ++cnt[px[i] = rk[id[i]]];
    for (i = 1; i <= m; ++i) cnt[i] += cnt[i - 1];
    for (i = n; i >= 1; --i) sa[cnt[px[i]]--] = id[i];
    memcpy(oldrk, rk, sizeof(rk));
    for (p = 0, i = 1; i <= n; ++i)
      rk[sa[i]] = cmp(sa[i], sa[i - 1], w) ? p : ++p;
  }
  // 利用后缀数组O(1)进行判断
  while (l <= r) {
    cout << (rk[l] < rk[n + 1 - r] ? s[l++] : s[r--]);
    if ((++tot) % 80 == 0) cout << '\n';  // 回车
  }

  return 0;
}

```

height 数组

LCP（最长公共前缀）

两个字符串 S 和 T 的 LCP 就是最大的 $x(x \leq \min(|\ S\ |, |\ T\ |))$ 使得 $S_i = T_i\ (\forall\ 1 \leq i \leq x)$.

下文中以 $lcp(i, j)$ 表示后缀 i 和后缀 j 的最长公共前缀（的长度）.

height 数组的定义

$height[i] = lcp(sa[i], sa[i-1])$，即第 i 名的后缀与它前一名的后缀的最长公共前缀.

$height[1]$ 可以视作 0.

O(n) 求 height 数组需要的一个引理

$height[rk[i]] \geq height[rk[i-1]] - 1$

证明

当 $height[rk[i-1]] \leq 1$ 时，上式显然成立（右边小于等于 0）.

当 $height[rk[i-1]] > 1$ 时：

根据 $height$ 定义，有 $lcp(sa[rk[i-1]], sa[rk[i-1]-1]) = height[rk[i-1]] > 1$.

既然后缀 $i-1$ 和后缀 $sa[rk[i-1]-1]$ 有长度为 $height[rk[i-1]]$ 的最长公共前缀，

那么不妨用 aA 来表示这个最长公共前缀.（其中 a 是一个字符，A 是长度为 $height[rk[i-1]]-1$ 的字符串，非空）

那么后缀 $i-1$ 可以表示为 aAD，后缀 $sa[rk[i-1]-1]$ 可以表示为 aAB.（$B < D$，B 可能为空串，D 非空）

进一步地，后缀 i 可以表示为 AD，存在后缀（$sa[rk[i-1]-1]+1$）AB.

因为后缀 $sa[rk[i]-1]$ 在大小关系的排名上仅比后缀 $sa[rk[i]]$ 也就是后缀 i，小一位，而 $AB < AD$.

所以 $AB \leqslant$ 后缀 $sa[rk[i]-1] < AD$，显然后缀 i 和后缀 $sa[rk[i]-1]$ 有公共前缀 A.

于是就可以得出 $lcp(i, sa[rk[i]-1])$ 至少是 $height[rk[i-1]]-1$，也即 $height[rk[i]] \geq height[rk[i-1]]-1$.

O(n) 求 height 数组的代码实现

利用上面这个引理暴力求即可：

```
for (i = 1, k = 0; i <= n; ++i) {
  if (rk[i] == 0) continue;
  if (k) --k;
  while (s[i + k] == s[sa[rk[i] - 1] + k]) ++k;
  height[rk[i]] = k;
}
```

k 不会超过 n，最多减 n 次，所以最多加 $2n$ 次，总复杂度就是 $O(n)$.

height 数组的应用

两子串最长公共前缀

$lcp(sa[i], sa[j]) = \min\{height[i+1..j]\}$

感性理解：如果 $height$ 一直大于某个数，前这么多位就一直没变过；反之，由于后缀已经排好序了，不可能变了之后变回来.

严格证明可以参考[2004] 后缀数组 by. 许智磊[9].

有了这个定理，求两子串最长公共前缀就转化为了 RMQ 问题→14.1@p3342.

比较一个字符串的两个子串的大小关系

假设需要比较的是 $A = S[a..b]$ 和 $B = S[c..d]$ 的大小关系.

若 $lcp(a, c) \geq \min(|A|, |B|)$，$A < B \iff |A| < |B|$.

否则，$A < B \iff rk[a] < rk[c]$.

不同子串的数目

子串就是后缀的前缀，所以可以枚举每个后缀，计算前缀总数，再减掉重复.

「前缀总数」其实就是子串个数，为 $n(n+1)/2$.

如果按后缀排序的顺序枚举后缀，每次新增的子串就是除了与上一个后缀的 LCP 剩下的前缀. 这些前缀一定是新增的，否则会破坏 $lcp(sa[i],sa[j])=\min\{height[i+1..j]\}$ 的性质. 只有这些前缀是新增的，因为 LCP 部分在枚举上一个前缀时计算过了.

所以答案为:

$\frac{n(n+1)}{2}-\sum_{i=2}^{n}height[i]$

出现至少 k 次的子串的最大长度

例题: 「USACO06DEC」Milk Patterns[10].

题解

出现至少 k 次意味着后缀排序后有至少连续 k 个后缀以这个子串作为公共前缀.

所以，求出每相邻 $k-1$ 个 $height$ 的最小值，再求这些最小值的最大值就是答案.

可以使用单调队列 $O(n)$ 解决，但使用其它方式也足以 AC.

参考代码

```cpp
#include <cstring>
#include <iostream>
#include <set>

using namespace std;

constexpr int N = 40010;

int n, k, a[N], sa[N], rk[N], oldrk[N], id[N], px[N], cnt[1000010], ht[N], ans;
multiset<int> t;  // multiset 是最好写的实现方式

bool cmp(int x, int y, int w) {
  return oldrk[x] == oldrk[y] && oldrk[x + w] == oldrk[y + w];
}

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  int i, j, w, p, m = 1000000;

  cin >> n >> k;
  --k;

  for (i = 1; i <= n; ++i) cin >> a[i];  // 求后缀数组
  for (i = 1; i <= n; ++i) ++cnt[rk[i] = a[i]];
  for (i = 1; i <= m; ++i) cnt[i] += cnt[i - 1];
  for (i = n; i >= 1; --i) sa[cnt[rk[i]]--] = i;

  for (w = 1; w < n; w <<= 1, m = p) {
```

```
    for (p = 0, i = n; i > n - w; --i) id[++p] = i;
    for (i = 1; i <= n; ++i)
      if (sa[i] > w) id[++p] = sa[i] - w;
    memset(cnt, 0, sizeof(cnt));
    for (i = 1; i <= n; ++i) ++cnt[px[i] = rk[id[i]]];
    for (i = 1; i <= m; ++i) cnt[i] += cnt[i - 1];
    for (i = n; i >= 1; --i) sa[cnt[px[i]]--] = id[i];
    memcpy(oldrk, rk, sizeof(rk));
    for (p = 0, i = 1; i <= n; ++i)
      rk[sa[i]] = cmp(sa[i], sa[i - 1], w) ? p : ++p;
  }

  for (i = 1, j = 0; i <= n; ++i) {  // 求 height
    if (j) --j;
    while (a[i + j] == a[sa[rk[i] - 1] + j]) ++j;
    ht[rk[i]] = j;
  }

  for (i = 1; i <= n; ++i) {  // 求所有最小值的最大值
    t.insert(ht[i]);
    if (i > k) t.erase(t.find(ht[i - k]));
    ans = max(ans, *t.begin());
  }

  cout << ans;

  return 0;
}
```

是否有某字符串在文本串中至少不重叠地出现了两次

可以二分目标串的长度 $|s|$，将 h 数组划分成若干个连续 LCP 大于等于 $|s|$ 的段，利用 RMQ 对每个段求其中出现的数中最大和最小的下标，若这两个下标的距离满足条件，则一定有长度为 $|s|$ 的字符串不重叠地出现了两次.

连续的若干个相同子串

我们可以枚举连续串的长度 $|s|$，按照 $|s|$ 对整个串进行分块，对相邻两块的块首进行 LCP 与 LCS 查询，具体可见[2009] 后缀数组——处理字符串的有力工具[11].

例题：「NOI2016」优秀的拆分[12].

结合并查集

某些题目求解时要求你将后缀数组划分成若干个连续 LCP 长度大于等于某一值的段，亦即将 h 数组划分成若干个连续最小值大于等于某一值的段并统计每一段的答案. 如果有多次询问，我们可以将询问离线. 观察到当给定值单调递减的时候，满足条件的区间个数总是越来越少，而新区间都是两个或多个原区间相连所得，且新区间中不包含在原区间内的部分的 h 值都为减少到的这个值. 我们只需要维护一个并查集，每次合并相邻的两个区间，并维护统计信息即可.

经典题目：「NOI2015」品酒大会[13]

结合线段树

某些题目让你求满足条件的前若干个数，而这些数又在后缀排序中的一个区间内. 这时我们可以用归并排序的性质来合并两个结点的信息，利用线段树维护和查询区间答案.

结合单调栈

例题：「AHOI2013」差异[14]

题解

被加数的前两项很好处理，为 $n(n-1)(n+1)/2$（每个后缀都出现了 $n-1$ 次，后缀总长是 $n(n+1)/2$），关键是最后一项，即后缀的两两 LCP.

我们知道 $lcp(i,j)=k$ 等价于 $\min\{height[i+1..j]\}=k$. 所以，可以把 $lcp(i,j)$ 记作 $\min\{x \mid i+1 \le x \le j, height[x]=lcp(i,j)\}$ 对答案的贡献.

考虑每个位置对答案的贡献是哪些后缀的 LCP，其实就是从它开始向左若干个连续的 $height$ 大于它的后缀中选一个，再从向右若干个连续的 $height$ 不小于它的后缀中选一个. 这个东西可以用 单调栈→10.9@p2067 计算.

单调栈部分类似于 Luogu P2659 美丽的序列[15] 以及 悬线法→13.7@p3280.

参考代码

```cpp
#include <cstring>
#include <iostream>
#include <string>
using namespace std;

constexpr int N = 500010;

string s;
int n, sa[N], rk[N << 1], oldrk[N << 1], id[N], px[N], cnt[N], ht[N],
sta[N],
    top, l[N];
long long ans;

bool cmp(int x, int y, int w) {
  return oldrk[x] == oldrk[y] && oldrk[x + w] == oldrk[y + w];
}

int main() {
  int i, k, w, p, m = 300;

  cin >> s;
  n = s.size();
  s = " " + s;
  ans = 1ll * n * (n - 1) * (n + 1) / 2;
  // 求后缀数组
  for (i = 1; i <= n; ++i) ++cnt[rk[i] = s[i]];
  for (i = 1; i <= m; ++i) cnt[i] += cnt[i - 1];
  for (i = n; i >= 1; --i) sa[cnt[rk[i]]--] = i;

  for (w = 1; w < n; w <<= 1, m = p) {
```

```
    for (p = 0, i = n; i > n - w; --i) id[++p] = i;
    for (i = 1; i <= n; ++i)
      if (sa[i] > w) id[++p] = sa[i] - w;
    memset(cnt, 0, sizeof(cnt));
    for (i = 1; i <= n; ++i) ++cnt[px[i] = rk[id[i]]];
    for (i = 1; i <= m; ++i) cnt[i] += cnt[i - 1];
    for (i = n; i >= 1; --i) sa[cnt[px[i]]--] = id[i];
    memcpy(oldrk, rk, sizeof(rk));
    for (p = 0, i = 1; i <= n; ++i)
      rk[sa[i]] = cmp(sa[i], sa[i - 1], w) ? p : ++p;
  }
  // 求 height
  for (i = 1, k = 0; i <= n; ++i) {
    if (k) --k;
    while (s[i + k] == s[sa[rk[i] - 1] + k]) ++k;
    ht[rk[i]] = k;
  }
  // 维护单调栈
  for (i = 1; i <= n; ++i) {
    while (ht[sta[top]] > ht[i]) --top;  // top类似于一个指针
    l[i] = i - sta[top];
    sta[++top] = i;
  }
  // 最后利用单调栈算 ans
  sta[++top] = n + 1;
  ht[n + 1] = -1;
  for (i = n; i >= 1; --i) {
    while (ht[sta[top]] >= ht[i]) --top;
    ans -= 2ll * ht[i] * l[i] * (sta[top] - i);
    sta[++top] = i;
  }

  cout << ans;

  return 0;
}

```

类似的题目：「HAOI2016」找相同字符[16].

习题

- UVa 760 - DNA Sequencing[17]
- UVa 1223 - Editor[18]
- Codechef - Tandem[19]
- Codechef - Substrings and Repetitions[20]
- Codechef - Entangled Strings[21]
- Codeforces - Martian Strings[22]
- Codeforces - Little Elephant and Strings[23]
- SPOJ - Ada and Terramorphing[24]
- SPOJ - Ada and Substring[25]
- UVa - 1227 - The longest constant gene[26]
- SPOJ - Longest Common Substring[27]

- UVa 11512 - GATTACA[28]
- QOJ 11240 - Suffixes and Palindromes[29]
- GYM - Por Costel and the Censorship Committee[30]
- UVa 1254 - Top 10[31]
- UVa 12191 - File Recover[32]
- UVa 12206 - Stammering Aliens[33]
- Codechef - Jarvis and LCP[34]
- 洛谷 P8617 - 重复模式[35]
- UVa 11107 - Life Forms[36]
- UVa 12974 - Exquisite Strings[37]
- UVa 10526 - Intellectual Property[38]
- UVa 12338 - Anti-Rhyme Pairs[39]
- DevSkills Reconstructing Blue Print of Life[40]
- UVa 12191 - File Recover[41]
- SPOJ - Suffix Array[42]
- Gym 102470J - Stammering Aliens[43]
- SPOJ - LCS2[44]
- Codeforces - Fake News (hard)[45]
- SPOJ - Longest Commong Substring[46]
- SPOJ - Lexicographical Substring Search[47]
- Codeforces - Forbidden Indices[48]
- Codeforces - Tricky and Clever Password[49]
- Gym 101470B - Circle of digits[50]

参考资料

本页面中（4070a9b[51] 引入的部分）主要译自博文 Суффиксный массив[52] 与其英文翻译版 Suffix Array[53]. 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.

论文：

1. [2004] 后缀数组 by. 许智磊[54]
2. [2009] 后缀数组——处理字符串的有力工具 by. 罗穗骞[55]

外部参考资料

[1] ./images/sa1.png

[2] ./images/sa2.png

[3] LOJ #111: 后缀排序

[4] 诱导排序与 SA-IS 算法

[5] 评论页面

[6] [2009] 后缀数组——处理字符串的有力工具 by. 罗穗骞

[7] 「JSOI2007」字符加密

[8]「USACO07DEC」Best Cow Line

[9] [2004] 后缀数组 by. 许智磊

[10] 「USACO06DEC」Milk Patterns

[11] [2009] 后缀数组——处理字符串的有力工具

[12] 「NOI2016」优秀的拆分

[13] 「NOI2015」品酒大会

[14] 「AHOI2013」差异

[15] Luogu P2659 美丽的序列

[16]「HAOI2016」找相同字符

[17] UVa 760 - DNA Sequencing

[18] UVa 1223 - Editor

[19] Codechef - Tandem

[20] Codechef - Substrings and Repetitions

[21] Codechef - Entangled Strings

[22] Codeforces - Martian Strings 

[23] Codeforces - Little Elephant and Strings 

[24] SPOJ - Ada and Terramorphing 

[25] SPOJ - Ada and Substring 

[26] UVa - 1227 - The longest constant gene 

[27] SPOJ - Longest Common Substring 

[28] UVa 11512 - GATTACA 

[29] QOJ 11240 - Suffixes and Palindromes 

[30] GYM - Por Costel and the Censorship Committee 

[31] UVa 1254 - Top 10 

[32] UVa 12191 - File Recover 

[33] UVa 12206 - Stammering Aliens 

[34] Codechef - Jarvis and LCP 

[35] 洛谷 P8617 - 重复模式 

[36] UVa 11107 - Life Forms 

[37] UVa 12974 - Exquisite Strings 

[38] UVa 10526 - Intellectual Property 

[39] UVa 12338 - Anti-Rhyme Pairs 

[40] DevSkills Reconstructing Blue Print of Life 

[41] UVa 12191 - File Recover 

[42] SPOJ - Suffix Array 

[43] Gym 102470J - Stammering Aliens 

[44] SPOJ - LCS2 

[45] Codeforces - Fake News (hard) 

[46] SPOJ - Longest Commong Substring 

[47] SPOJ - Lexicographical Substring Search 

[48] Codeforces - Forbidden Indices 

[49] Codeforces - Tricky and Clever Password 

[50] Gym 101470B - Circle of digits 

[51] 4070a9b 

[52] Суффиксный массив 

[53] Suffix Array 

[54] [2004] 后缀数组 by. 许智磊 

[55] [2009] 后缀数组——处理字符串的有力工具 by. 罗穗骞 

8.11.2 最优原地后缀排序算法

本章介绍线性时间复杂度的后缀排序的就地算法[1]（Optimal In-Place Suffix Sorting）.

> ⚠ **Warning**
>
> 本章 **只建议** 在 **非常非常熟悉** SA-IS[2][3]的前提下阅读.

全局设定

目标字符串 `Pat`，后缀数组 `SA`，串的序号从 0 开始，结尾字符是警戒哨，不妨设为 0.

在整形字母表上的后缀排序

事实上这一部分可以看成是原地版本的 SA-IS 算法.

因为是原文中细节相对最清楚，实现也较为简单的算法，也是了解后续算法的基础，是本文介绍的重点.

---

[1] Li, Zhize; Li, Jian; Huo, Hongwei (2016).*Optimal In-Place Suffix Sorting*. Proceedings of the 25th International Symposium on String Processing and Information Retrieval (SPIRE). Lecture Notes in Computer Science. 11147. Springer. pp. 268–284. arXiv:1610.08305. doi:10.1007/978-3-030-00479-8_22. ISBN:978-3-030-00478-1.

[2] Ge Nong, Sen Zhang, and Wai Hong Chan. Linear suffix array construction by almost pure induced-sorting. In Data Compression Conference (DCC), pages 193–202. IEEE, 2009.

[3] 推荐阅读 博文[1] 和它的 issue 列表[2]

原地化的原理是用重命名的 Pat 代替 S、L 桶，用额外 $O(n)$ 的操作代替类型桶.

重命名目标串 Pat

简单来说，我们会在不改变后缀大小的相对顺序的前提下，重命名 Pat，用重命名后的 Pat 来取代原来 S、L 桶，来指明桶头或者桶尾.

重命名的方法是将 Pat 中的 S 型字符替换为所在桶的桶尾索引，L 型字符替换为所在桶的桶头索引.

如下图所示:

```
Index :   0123456789101112
  Pat :   2113311331210
 Type :   LSSLLSSLLSLLS
Bucket :  (0)(111111)(22)(3333)
```

重命名后的 Pat‘（之后直接将重命名后的 Pat‘ 称做 Pat）:

```
Index :   0123456789101112
 Pat‘ :   7669966996710
```

由于桶内的字符，L 型字符后缀小，作为桶头；而 S 型字符后缀大，作为桶尾，因此保持了后缀大小的相对顺序.

描述一下重命名的具体步骤:

1. 和 SA-IS 一样，对 Pat 中每个字符计数，计算其前缀和（计数排序），来构建 S/L 桶，只不过这里用 SA 盛放这个前缀和;
2. 从尾到头，扫描 Pat 的每个字符，这样只需记录上一个字符的类型，就可以动态地判断每个字符的类型，然后依据前缀和将其重命名.

对 LMS 字符排序

这里重点是使用了一个内部计数器的技巧.

初始化

初始的时候将 SA 每一项设为 E（EMPTY）.

从尾到头扫描 Pat，如果发现是 LMS 字符，Pat[i]，那么就设置 SA[Pat[i]] 的标记:

如果 SA[Pat[i]] 是 E，就将其设为 U（UNIQUE）;

如果 SA[Pat[i]] 是 U，就将其设为 M（MULTIPLE）;

其他情况，不做处理.

结果如下图所示:

```
Index :   0123456789101112
  Pat :   7669966996710
  LMS :   ****
   SA :   (U)(E)(EEEEM)(EE)(EEEE)
```

把 LMS 字符的索引放入 SA

从尾到头扫描 Pat，对于 LMS 字符 Pat[i]，根据 SA[Pat[i]] 的符号进行分类讨论:

U：直接让 SA[Pat[i]]=i

M：意味着桶中有至少两个 LMS 字符.

1. 如果桶中有至少三个 LMS 字符： 就把桶中倒数第二个位置作为临时计数器，标志桶中已填充的 LMS 字符数（桶中倒数第一位就是标志 M） 将新的 LMS 字符从倒数第三个位置开始插入，让临时计数器自增 1. 如果发现桶已经满了，就把桶中从桶头到倒数第三个的所有元素向右平移 2 个位置，然后把新元素插入到桶中第二个位置（桶中第一个位置填为 E）

2. 如果桶中有且只有 2 个 LMS 字符，显然不需要计数器，直接从右到左顺序插入即可.

正常的值:

```
根据我们之前的讨论，此时不管桶中有两个还是两个以上的 LMS 字符，这都意味着 $\texttt{i}$ 是桶中最后一个待插入的 LMS 字符的位置，

只需要从桶头开始向左扫描，找到第一个标记为 E 的位置，将其设为 $\texttt{i}$.
```

最后要从尾到头扫描一遍 SA，清除可能残余的特殊符号 M（桶中未被填满，所以 M 和计数器未被覆盖）.

方法是将桶中 LMS 字符如上述步骤一样向右平移 2 位，将左边空出来的位置填为 E.

如下图所示:

```
Index :  0123456789101112
  Pat :  7669966996710
   SA :  (12)(E)(EEEEM)(EE)(EEEE)
   SA :  (12)(E)(EE91M)(EE)(EEEE)
   SA :  (12)(E)(E592M)(EE)(EEEE)
   SA :  (12)(E)(1593M)(EE)(EEEE)
   SA :  (12)(E)(EE159)(EE)(EEEE)
```

这个阶段，由于每个桶只需要被移动和扫描一次，所以时间复杂度是 $O(n)$.

诱导排序 LMS 子串

诱导排序 LMS 前缀

将 LMS 前缀进行诱导排序，同 SA-IS 一样，这部分同后面对后缀的诱导排序完全一样（使用同一个函数），因此这里直接跳过.

这里直接给出排序结果:

```
Index :  0123456789101112
   SA :  (12)(11)(15926)(100)(4837)
```

将已排序的 LMS 子串放到 SA 尾部

```
Index :  0123456789101112
   SA :  EEEEEEEEE12159
```

构建规模缩减的子目标串 Pat1

从左到右扫描 SA 尾部的 LMS 子串, 确定其大小关系「重命名」, 将 SA[i] 重命名的值存储在 $\mathtt{SA}\left[\left\lfloor\frac{\mathtt{SA}[i]}{2}\right\rfloor\right]$.

因为 LMS 字符并不相邻，所以不会有冲突，这样做是将重命名后的值按照所代表的子串在 Pat 中的原顺序放置:

```
Index :  0123456789101112
   SA :  1E1E2E0EE12159
```

然后扫描 SA，收集这些重命名的值到 SA 头部:

```
Index :  0123456789101112
   SA :  1120EEEEE12159
```

通过递归解决 Pat1，完成对 LMS 后缀的排序

同 SA-IS 一样，递归解决 SA 头部的规模缩减的 Pat1 的后缀排序，结果存到 SA 尾部:

```
Index :  0123456789101112
   SA :  1120EEEEE3012
```

将 SA 尾部的 SA1 挪到 SA 头部，重新从尾到头扫描 Pat，将其中 LMS 字符按照在 Pat 中的顺序放到 SA 尾部：

```
Index :  0123456789101112
   SA :  3012EEEEE15912
```

依照 SA 尾部的「对照表」，将 SA1 头部的 SA 还原为 Pat 中对应的 LMS 后缀的索引位置：

```
Index :  0123456789101112
   SA :  12159EEEEE15912
```

将 SA 头部的排好序的 LMS 后缀按顺序放入到对应的桶中（从尾部开始放）：

```
Index :  0123456789101112
   SA :  (12)(E)(EE159)(EE)(EEEE)
```

对 Pat1 中所有的后缀进行诱导排序

这一部分就是利用前面用过的内部计数器技巧，进行原地版的诱导排序.

假如我们已经有排好序的 LMS 后缀（在桶尾），来诱导 L 型后缀[4]：

```
Index :  0123456789101112
  Pat :  7669966996710
   SA :  (12)(E)(EE159)(EE)(EEEE)
```

如同排序 LMS 字符一样，先对 L 型字符用特殊符号计数：

```
Index :  0123456789101112
  Pat :  7669966996710
   SA :  (12)(U)(EE159)(ME)(MEEE)
```

从左到右扫描 SA，同对 LMS 字符排序一样，复杂一点的是判断 suf[SA[i]-1] 的类型，需要分类讨论（详情参考代码）：

```
Index :  0123456789101112
   SA :  (12)(11)(EE159)(ME)(MEEE)
   SA :  (12)(11)(EE159)(10E)(MEEE)
   SA :  (12)(11)(EE159)(100)(MEEE)
   SA :  (12)(11)(EE159)(100)(M14E)
   SA :  (12)(11)(EE159)(100)(M248)
   SA :  (12)(11)(EE159)(100)(483E)
   SA :  (12)(11)(EE159)(100)(4837)
```

区别于 SA-IS 的是，对一个类型字符诱导排序后，需要清理 LMS 字符以免对后面的原地诱导排序：

```
Index :  0123456789101112
   SA :  (12)(11)(EEEEE)(100)(4837)
```

至于从 L 后缀诱导 S 后缀与从 LMS 后缀诱导 L 后缀完全对称，这里就不做多余介绍.

到这儿为止，诱导排序就完成了.

实现

时间性能上和 SA-IS 没有显著差别，空间占用变为不到原来的 $\frac{1}{3}$（代码量多 1 倍），算是不愧为原文 Optimal In-Place Suffix Sorting[5] 的标题.

---

[4] 如果是 LML 后缀，就先诱导 S 型后缀，唯一区别是计算 LML 后缀时需要将警戒哨也算进去.

[5] Li, Zhize; Li, Jian; Huo, Hongwei (2016). *Optimal In-Place Suffix Sorting*. Proceedings of the 25th International Symposium on String Processing and Information Retrieval (SPIRE). Lecture Notes in Computer Science. 11147. Springer. pp. 268–284. arXiv:1610.08305. doi:10.1007/978-3-030-00479-8_22. ISBN:978-3-030-00478-1.

参考代码

```rust
use std::cmp::max;
use std::cmp::Ordering;
use std::slice::from_raw_parts_mut;


const LTYPE: bool = false;
const STYPE: bool = true;
const MAX_SA_VALUE: usize = usize::MAX / 2;
const EMPTY: usize = MAX_SA_VALUE + 1;
const UNIQUE: usize = MAX_SA_VALUE + 2;
const MULTI: usize = MAX_SA_VALUE + 3;  // >= 258


fn lms_str_cmp<E: Ord>(l1: &[E], l2: &[E]) -> Ordering {
    for (x, y) in l1.iter().zip(l2.iter()) {
        let cmp_res = x.cmp(&y);

        if cmp_res != Ordering::Equal { return cmp_res; }
    }

    Ordering::Equal
}

#[inline]
fn pat_char_type(cur: usize, prev: usize, last_scanned_type: bool) -> bool {
    if cur < prev || cur == prev && last_scanned_type == STYPE { STYPE }
    else { LTYPE }
}


fn rename_pat(pat: &mut [usize], sa: &mut [usize]) {
    let patlastpos = pat.len() - 1;
    // 全部刷成bucket head
    //sa.fill(0);
    for i in 0..sa.len() { sa[i] = 0 }

    for i in 0..pat.len() { sa[pat[i]] += 1 }
    for i in 1..sa.len() { sa[i] += sa[i - 1] }

    for i in 0..pat.len() - 1 {
        pat[i] = sa[pat[i]] - 1;
    };
    // 将L-suffix刷成bucket head
    //sa.fill(0);
    for i in 0..sa.len() { sa[i] = 0 }

    for i in 0..pat.len() { sa[pat[i]] += 1 }
    let mut last_scanned_type = STYPE;
    pat[patlastpos] = 0;
    for i in (0..pat.len() - 1).rev() {
        if pat_char_type(pat[i], pat[i + 1], last_scanned_type) == STYPE {
```

```
52              last_scanned_type = STYPE;
53          } else {
54              pat[i] -= sa[pat[i]] - 1;
55              last_scanned_type = LTYPE;
56          }
57      }
58
59  }
60
61
62  fn sort_lms_char(pat: &mut [usize], sa: &mut [usize]) -> usize {
63      //sa.fill(EMPTY);
64      for i in 0..sa.len() { sa[i] = EMPTY }
65
66      let mut last_scanned_type = STYPE;
67      for i in (0..pat.len() - 1).rev() {
68          if pat_char_type(pat[i], pat[i + 1], last_scanned_type) == STYPE {
69              last_scanned_type = STYPE;
70          } else {
71              if last_scanned_type == STYPE {  // pat[i + 1] is LMS type
72                  sa[pat[i + 1]] += 1;
73              }
74
75              last_scanned_type = LTYPE;
76          }
77      }
78
79      let mut lms_cnt = 0;
80      last_scanned_type = STYPE;
81      for i in (0..pat.len() - 1).rev() {
82          if pat_char_type(pat[i], pat[i + 1], last_scanned_type) == STYPE {
83              last_scanned_type = STYPE;
84          } else {
85              let e_i = i + 1;
86              let e = pat[e_i];
87
88              if last_scanned_type == STYPE {  // pat[i + 1] is LMS type
89                  lms_cnt += 1;
90                  if sa[e] == UNIQUE {
91                      sa[e] = e_i;
92                  } else if sa[e] >= MULTI && sa[e - 1] == EMPTY {
93                      if sa[e - 2] == EMPTY {
94                          sa[e - 2] = e_i;
95                          sa[e - 1] = 1;  // set counter
96                      } else {  // MUL = 2
97                          sa[e] = e_i;
98                          sa[e - 1] = EMPTY;
99                      }
100                 } else if sa[e] >= MULTI && sa[e - 1] != EMPTY {
101                     let c = sa[e - 1];  // get counter
102
103                     if sa[e - 2 - c] == EMPTY {
```

```
                        sa[e - 2 - c] = e_i;
                        sa[e - 1] += 1;  // update counter
                    } else {
                        for j in (1..c + 1).rev() {
                            sa[e - c + j] = sa[e - 2 - c + j]
                        }
                        sa[e - c] = e_i;
                        sa[e - c - 1] = EMPTY;
                    }
                } else if sa[e] < EMPTY {
                    for j in (0..e).rev() {
                        if sa[j] == EMPTY {
                            sa[j] = e_i;
                            break;
                        }
                    }
                }
            }

            last_scanned_type = LTYPE;
        }
    }

    for i in (0..pat.len()).rev() {
        if sa[i] >= MULTI {
            let c = sa[i - 1];
            for j in (1..c + 1).rev() {  // 逆序防止前面的覆盖后面的
                sa[i - c + j] = sa[i - 2 - c + j];
            }
            sa[i - c - 1] = EMPTY;
            sa[i - c] = EMPTY;
        }
    }

    lms_cnt
}


fn sort_lms_substr(pat: &mut [usize], sa: &mut [usize]) {
    // step 1
    induced_sort(pat, sa);

    // step 2
    let pat_last_pos = pat.len() - 1;
    let mut lms_cnt = 0;
    let mut i = pat_last_pos;
    let mut bucket_tail_ptr = pat_last_pos + 1;  // for renamed bucket ver
    let mut bucket = EMPTY;  // 可以省略，但是为了书写代码方便
    let mut num = 0;  // S type number of bucket
    while i > 0 {
        if pat[sa[i]] != bucket {  // reach new bucket
            num = 0;
```

```
156
157             let mut l = 0;
158             while pat[sa[i - l]] == pat[sa[i]] {  // 扫描桶来计算桶中S字符数量，根据定义 当l=i时循环必然终止
159                 let pat_i = sa[i - l];              // l < i, 即 i - l > 0, 0 <= pat_i < patlen - 1
160                 if pat[pat_i] < pat[pat_i + 1] {
161                     let mut k = pat_i;
162                     while k > 0 && pat[k - 1] == pat[pat_i] { k -= 1 }
163                     num += pat_i - k + 1;
164                 } else {
165                     break;   // bucket不含S字符，结束扫描
166                 }
167
168                 l += 1;
169             }
170
171             bucket_tail_ptr = i;
172             bucket = pat[sa[bucket_tail_ptr]];
173         }
174
175         if num > 0
176         && i > bucket_tail_ptr - num
177         && sa[i] > 0
178         && pat[sa[i]] < pat[sa[i] - 1]  {
179             sa[pat_last_pos - lms_cnt] = sa[i];
180             lms_cnt += 1;
181         }
182
183         i -= 1;
184     }
185
186     sa[pat_last_pos - lms_cnt ] = sa[i];  // i = 0
187     lms_cnt += 1;
188     //sa[0..pat_last_pos - lms_cnt + 1].fill(EMPTY);
189     for i in 0..pat_last_pos - lms_cnt + 1 { sa[i] = EMPTY }
190 }
191
192
193 fn construct_pat1(pat: &mut [usize], sa: &mut [usize], lms_cnt: usize) -> bool {
194     let patlen = pat.len();
195
196     let mut prev_lms_str_len = 1;
197     let mut rank = 0;
198     sa[(patlen - 1) / 2] = rank;
199     let mut has_duplicated_char = false;
200     for i in patlen - lms_cnt + 1..patlen {  // 从警戒哨字符的下一个字符开始
201         let mut j = sa[i];
202         while pat[j] <= pat[j + 1] { j += 1 } // 寻找suf(sa[i])右边第一个L字符，因为排除了警戒哨这个LMS后缀，所以必然不会越界
203         let mut k = j;
```

```
            while k + 1 < patlen && pat[k] >= pat[k + 1] { k += 1 }  // 找到suf(sa[i])右边第一个LMS字符
            let cur_lms_str_len = k + 1 - sa[i];
            let cmp_res = lms_str_cmp(&pat[sa[i]..sa[i] + cur_lms_str_len], &pat[sa[i - 1]..sa[i - 1] + prev_lms_str_len]);

            if  cmp_res != Ordering::Equal {
                rank += 1
            }

            if rank == sa[sa[i - 1] / 2] {
                has_duplicated_char = true;
            }
            let rank_index = sa[i] / 2;
            sa[rank_index] = rank;  // 整除

            prev_lms_str_len = cur_lms_str_len;
        }

        // move to head of sa
        let mut j = 0;
        for i in 0..patlen - lms_cnt {
            if sa[i] != EMPTY {
                sa[j] = sa[i];
                if i > j {
                    sa[i] = EMPTY;
                }
                j += 1;
            }
        }
        //sa[lms_cnt..patlen].fill(EMPTY);
        for i in lms_cnt..patlen { sa[i] = EMPTY }

        has_duplicated_char
    }

    fn sort_lms_suf(pat: &mut [usize], sa: &mut [usize], lms_cnt: usize, has_duplicated_char: bool) {
        // solve T1 recursively
        let patlen = pat.len();
        let salen = sa.len();
        unsafe {
            let sa_ptr = sa.as_mut_ptr();
            let mut pat1 = from_raw_parts_mut(sa_ptr, lms_cnt);
            let mut sa1 = from_raw_parts_mut(sa_ptr.offset((patlen - lms_cnt) as isize), salen - (patlen - lms_cnt));

            if has_duplicated_char {
                _compute_suffix_array_16_1(&mut pat1, &mut sa1);
            } else {
                for i in 0..lms_cnt { sa1[pat1[i]] = i }
            }
```

```
    }

    // move SA1 to SA[0...n1-1]
    for i in 0..lms_cnt {
        sa[i] = sa[patlen- lms_cnt + i];
    }

    // put all LMS-suffixes in SA tail
    let mut last_scanned_type = STYPE;
    let mut j = 0;
    for i in (0..pat.len() - 1).rev() {
        if pat[i] < pat[i + 1] || pat[i] == pat[i + 1] && last_scanned_type
== STYPE {
            last_scanned_type = STYPE;
        } else {
            if last_scanned_type == STYPE {
                sa[patlen - 1 - j] = i + 1;
                j += 1;
            }

            last_scanned_type = LTYPE;
        }
    }

    // backward map the LMS-suffixes rank
    for i in 0..lms_cnt {
        let relative_rank = sa[i];
        sa[i] = sa[patlen - lms_cnt + relative_rank];
        sa[patlen - lms_cnt + relative_rank] = EMPTY;
    }

    let mut tail = EMPTY;
    let mut rfp = EMPTY;
    for i in (1..lms_cnt).rev() { // sa[0] 保持原位
        if pat[sa[i]] != tail {
            tail = pat[sa[i]];
            rfp = tail;
        }

        sa[rfp] = sa[i];
        if rfp != i { sa[i] = EMPTY }
        rfp -= 1;
    }
}

// PASS!
fn induced_sort(pat: &mut [usize], sa: &mut [usize]) {
    let patlen = pat.len();

    // place L-suff in SA
    // init
    let mut last_scanned_type = STYPE;
```

```
    for i in (0..patlen - 1).rev() {
        if pat_char_type(pat[i], pat[i + 1], last_scanned_type) == LTYPE {
            sa[pat[i]] += 1;  // >= EMPTY
            last_scanned_type = LTYPE;
        } else {
            last_scanned_type = STYPE;
        }
    }
    //place
    let mut i = 0;
    while i < patlen {
        if sa[i] < EMPTY && sa[i] > 0 {
            let j = sa[i] - 1;
            let mut is_ltype = false;
            if pat[j] > pat[j + 1] {
                is_ltype = true;
            } else if pat[j] == pat[j + 1] {  // 判断sa[i]是否是L后缀的编号
                let next_i = sa[pat[sa[i]]];
                if next_i >= MULTI {
                    is_ltype = true;
                } else if next_i < EMPTY && pat[sa[i]] + 1 < patlen {
                    if sa[pat[sa[i]] + 1] == EMPTY {
                        is_ltype = true;
                    } else if sa[pat[sa[i]] + 1] < EMPTY {
                        if pat[sa[pat[sa[i]] + 1]] == pat[sa[i]] {
                            is_ltype = true;
                        }
                    }
                }
            }

            if is_ltype {
                if sa[pat[j]] == UNIQUE {
                    sa[pat[j]] = j;
                } else if sa[pat[j]] >= MULTI && sa[pat[j] + 1] == EMPTY {
                    if sa[pat[j]] - EMPTY > 2 {
                        sa[pat[j] + 2] = j;
                        sa[pat[j] + 1] = 1;  // set counter
                    } else {
                        sa[pat[j]] = j;
                    }
                } else if sa[pat[j]] >= MULTI && sa[pat[j] + 1] != EMPTY {
                    let e = pat[j];
                    let c = sa[e + 1];
                    let lfp = e + c + 2;
                    if  c + 2 < sa[pat[j]] - EMPTY {  // 没到bucket尾部
                        sa[lfp] = j;
                        sa[e + 1] += 1;  // update counter
                    } else {
                        for k in 1..c + 1 {
                            sa[e + k - 1] = sa[e + k + 1];
                        }
```

```
                        sa[e + c] = j;
                        sa[e + c + 1] = EMPTY;
                        if i >= e + 2 && i <= e + c + 1 {
                            i -= 2;
                        }
                    }
                } else if sa[pat[j]] < EMPTY {
                    for k in pat[j]..patlen {
                        if sa[k] == EMPTY {
                            sa[k] = j;
                            break;
                        }
                    }
                }
            }
        } else if sa[i] >= MULTI {
            i += 1;
        }

        i += 1;
    }

    // remove LMS-suff form SA, 一个桶里可能有多个LMS后缀
    last_scanned_type = STYPE;
    for i in (0..pat.len() - 1).rev() {
        if pat_char_type(pat[i], pat[i + 1], last_scanned_type) == STYPE {
            last_scanned_type = STYPE;
        } else {
            if last_scanned_type == STYPE {  // pat[i + 1] is LMS type
                if sa[pat[i + 1]] <= EMPTY {
                    sa[pat[i + 1]] = UNIQUE;
                } else {
                    sa[pat[i + 1]] += 1;
                }
            }

            last_scanned_type = LTYPE;
        }
    }
    i = patlen - 1;
    while i > 0 {
        if sa[i] > EMPTY {
            let c = sa[i] - EMPTY;
            for k in 0..c {
                sa[i - k] = EMPTY;
            }
            i -= c - 1;
        }

        i -= 1;
    }
    sa[0] = pat.len() - 1;
```

```
407
408     // place S-suff in SA
409     // init
410     let mut last_scanned_type = STYPE;
411     for i in (0..patlen - 1).rev() {
412         if pat_char_type(pat[i], pat[i + 1], last_scanned_type) == STYPE {
413             if sa[pat[i]] >= EMPTY {
414                 sa[pat[i]] += 1;
415             } else {
416                 sa[pat[i]] = UNIQUE;
417             }
418             last_scanned_type = STYPE;
419         } else {
420             last_scanned_type = LTYPE;
421         }
422     }
423     i = patlen - 1;
424     while i > 0 {
425         if sa[i] < EMPTY && sa[i] > 0 {
426             let j = sa[i] - 1;
427             let mut is_stype = false;
428             if pat[j] < pat[j + 1] {
429                 is_stype = true;
430             } else if pat[j] == pat[j + 1] {  // 判断sa[i]是否是S后缀的编号
431                 let next_i = sa[pat[sa[i]]];
432                 if next_i >= MULTI {
433                     is_stype = true;
434                 } else if next_i < EMPTY && pat[sa[i]] - 1 > 0 {
435                     if sa[pat[sa[i]] - 1] == EMPTY {
436                         is_stype = true;
437                     } else if sa[pat[sa[i]] - 1] < EMPTY {
438                         if pat[sa[pat[sa[i]] - 1]] == pat[sa[i]] {
439                             is_stype = true;
440                         }
441                     }
442                 }
443             }
444
445             if is_stype {
446                 if sa[pat[j]] == UNIQUE {
447                     sa[pat[j]] = j;
448                 } else if sa[pat[j]] >= MULTI && sa[pat[j] - 1] == EMPTY {
449                     if sa[pat[j]] - EMPTY > 2 {
450                         sa[pat[j] - 2] = j;
451                         sa[pat[j] - 1] = 1;  // set counter
452                     } else {
453                         sa[pat[j]] = j;
454                     }
455                 } else if sa[pat[j]] >= MULTI && sa[pat[j] - 1] != EMPTY {
456                     let e = pat[j];
457                     let c = sa[e - 1];
458                     let num = sa[pat[j]] - EMPTY;
```

```
                    if c + 2 < num {  // 没到bucket头部
                        let rfp = e - c - 2;
                        sa[rfp] = j;
                        sa[e - 1] += 1;
                    } else {
                        for k in 1..c + 1 {
                            sa[e - k + 1] = sa[e - k - 1];
                        }
                        sa[e - c] = j;
                        sa[e - c - 1] = EMPTY;
                        if i >= e - num + 1 && i <= e - 2 {
                            i += 2;
                        }
                    }
                } else if sa[pat[j]] < EMPTY {
                    for k in (0..pat[j]).rev() {
                        if sa[k] == EMPTY {
                            sa[k] = j;
                            break;
                        }
                    }
                }
            }
        } else if sa[i] >= MULTI {
            i -= 1;
        }
        i -= 1;
    }
}

fn _compute_suffix_array_16_1(pat: &mut [usize], sa: &mut [usize]) {
    rename_pat(pat, sa);
    let lms_cnt = sort_lms_char(pat, sa);
    sort_lms_substr(pat, sa);
    let has_duplicated_char = construct_pat1(pat, sa, lms_cnt);
    sort_lms_suf(pat, sa, lms_cnt, has_duplicated_char);
    induced_sort(pat, sa);
}

pub fn suffix_array_16(pat: &[u8]) -> Vec<usize> {
    let mut pat = pat.into_iter().map(|x| *x as
usize).collect::<Vec<usize>>();
    pat.push(0);
    let mut sa = vec![0; max(pat.len(), 256) * 1];
    _compute_suffix_array_16_1(&mut pat[..], &mut sa[..]);

    sa
}

fn input() -> String {
    use std::io;

```

```
510     let mut input = String::new();
511     io::stdin().read_line(&mut input).unwrap();
512     String::from(input.trim())
513 }
514
515
516 fn main() {
517     let pat = input();
518
519     let sa_16 = suffix_array_16(pat.as_bytes());
520
521     for i in 1..pat.len() + 1 { print!("{} ", sa_16[i] + 1) }
522 }
```

在只读的整形字母表上的后缀排序

使用复杂方法解决复杂问题，通过分治，解决空间紧张的问题.

算法实现的难点在于在 SA 上构建 BitMaps[6]，来替代本来由重命名后的 T 所指示的指示桶尾/桶头的位置.

这里的 BitMaps 指得是使用比特向量（bit vector）表示的有序字典（multiset），是一种紧凑型结构（compact data structure）.

有兴趣了解的暂时只能阅读原文以及本文引用的 BitMaps 的有关论文自行了解.

在只读的一般字母表上的后缀排序

前置知识是归并排序和堆排序.

由于笔者对于其中确定字符类型的方法的时间复杂度有疑问，这里也不再介绍，建议阅读原文自行了解.

注解

[1] 博文  [2] issue 列表 

8.12 后缀自动机 (SAM)

Authors: GoodCoder666, abc1763613206, ksyx

一些记号

- Σ：字符集. 字符集大小记为 $|\Sigma|$.
- s：字符串. 字符串长 $|s| = n$，标号自 0 开始.
- t_0：初始状态.
- $\text{endpos}(t)$：字符串 s 中子串 t 的结束位置的集合.
- $\text{link}(v)$：状态 v 的后缀链接.
- $\text{len}(v)$：状态 v 对应的最长子串的长度.
- $\text{longest}(v)$：状态 v 对应的最长子串.
- $\text{minlen}(v)$：状态 v 对应的最短子串的长度.

[6] Gonzalo Navarro and Eliana Providel. Fast, small, simple rank/select on bitmaps. In Proc. 11th International Symposium on Experimental Algorithms (SEA), pages 295–306, 2012.

- shortest(v)：状态 v 对应的最短子串.

后缀自动机概述

后缀自动机（suffix automaton, SAM）是一个能解决许多字符串相关问题的有力的数据结构.

举个例子，以下的字符串问题都可以在线性时间内通过 SAM 解决：

- 在另一个字符串中搜索一个字符串的所有出现位置；
- 计算给定的字符串中有多少个不同的子串.

直观上，字符串的 SAM 可以理解为给定字符串的 **所有子串** 的压缩形式. 值得注意的事实是，SAM 将所有的这些信息以高度压缩的形式储存. 对于一个长度为 n 的字符串，它的空间复杂度仅为 $O(n)$. 而且，构造 SAM 的时间复杂度也仅为 $O(n)$. 准确地说，当 $n \geq 3$ 时，一个 SAM 最多有 $2n-1$ 个结点和 $3n-4$ 条转移边.

定义

字符串 s 的 SAM 是一个接受 s 的所有后缀的最小 DFA→13.8@p3284.

换句话说：

- SAM 是一张有向无环图. 结点被称作 **状态**，边被称作状态间的 **转移**.
- 图存在一个源点 t_0，称作 **初始状态**，其它各结点均可从 t_0 出发到达.
- 每个 **转移** 都标有某个字符. 从一个结点出发的所有转移均 **不同**.
- 存在一个或多个 **终止状态**. 如果我们从初始状态 t_0 出发，最终转移到了一个终止状态，则路径上的所有转移的标号连接起来一定是字符串 s 的一个后缀. 反过来，s 的每个后缀均可用一条从 t_0 到某个终止状态的路径构成.
- 在所有满足上述条件的自动机中，SAM 的结点数是最少的.

SAM 的关键恰在于这个最小性. 实际上，直接对字符串 s 的所有后缀建立 字典树→8.6@p903（即后缀 Trie）同样可以得到一个接受 s 的所有后缀的 DFA. 但是，最差情况下，这样得到的自动机有 $\Theta(n^2)$ 个结点，复杂度难以接受. 从下面的例子可以看出，对所有后缀建立字典树得到的 DFA 中很多结点是重复的，因而可以合并. SAM 正是将结点的合并做到了极致，故而将得到的 DFA 的规模控制在 $O(n)$. 从这个意义上，SAM 是字符串的全体后缀的「压缩」的字典树.

子串和路径

SAM 最简单、也最重要的性质是，它包含关于字符串 s 的所有子串的信息. 任意从初始状态 t_0 开始的路径，如果我们将路径上的所有转移的标号写下来，都会形成 s 的一个 **子串**. 反之，每个 s 的子串都对应从 t_0 开始的某条路径.

为了简化表达，我们称子串 **对应** 这条（从 t_0 出发且它上面所有转移的标号构成这个子串的）路径. 反过来，我们说任意一条路径都 **对应** 它的标号构成的字符串.

到达某个状态的路径可能不止一条，因此我们说一个状态对应一些字符串的集合，这个集合中的字符串分别对应着这些路径.

简单例子

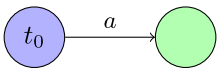
我们将会在这里展示一些简单的字符串的后缀自动机.

我们用蓝色表示初始状态，用绿色表示终止状态.

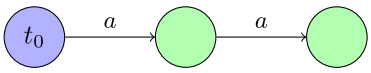
对于字符串 $s = \varepsilon$：



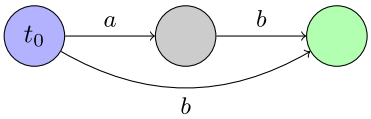
对于字符串 $s = \texttt{a}$：



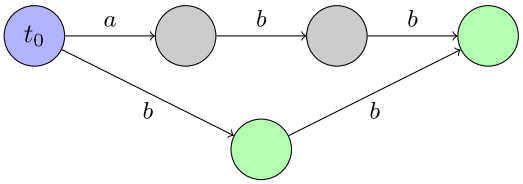
对于字符串 $s = \texttt{aa}$：



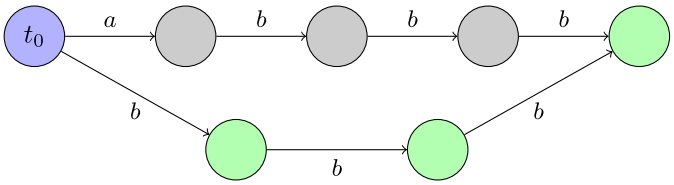
对于字符串 $s = \texttt{ab}$：



对于字符串 $s = \texttt{abb}$：



对于字符串 $s = \texttt{abbb}$：



在最后这个例子中可以看到，如果直接建立它的所有后缀的字典树，路径 bbb 和路径 abbb 应当导向不同的结点，但是这两个结点都是终止状态，且无论再添加任何字符，都不会得到更长的匹配串，这说明两个结点在自动机的转移上表现出同样的性质，因而可以合并成同一个结点．这样就得到了如图所示的 SAM．下面的讨论会将合并结点这一想法拓展到所有的情形，并说明，只要合理地合并结点，最后得到的 SAM 只有 $O(n)$ 个结点和转移．

线性复杂度的构造算法

在我们描述线性时间内构造 SAM 的算法之前，我们需要引入两个对理解构造过程非常重要的概念，并对其性质进行简单证明．其中，结束位置 endpos 定义了 SAM 中的结点（亦即指出了结点可以合并的充要条件），而后缀链接 link 不过是 AC 自动机中的 失配指针→8.10@p951 在 SAM 中的自然对应．

结束位置 endpos

考虑字符串 s 的任意非空子串 t，我们记 $\text{endpos}(t)$ 为在字符串 s 中 t 的所有结束位置的集合（假设对字符串中字符的编号从零开始）. 例如，对于字符串 `abcbc`，我们有 $\text{endpos}(\texttt{bc}) = \{2, 4\}$.

两个子串 t_1 与 t_2 的结束位置可能完全相同：$\text{endpos}(t_1) = \text{endpos}(t_2)$. 这定义了字符串 s 的子串之间的等价关系. 字符串 s 的所有非空子串可以根据它们的结束位置集合 endpos 分为若干 **等价类**.

一个事实是，每个这样的等价类都对应 SAM 的一个状态[1]. 也就是说，只要两个子串的结束位置相同，它们在 SAM 中的路径就对应着同一个状态. 换句话说，SAM 中的每个非初始状态都对应一个或多个 endpos 相同的非空子串. 总之，SAM 中的状态就是所有非空子串的等价类，再加上初始状态.

暂且接受这个事实，我们将基于它介绍构造 SAM 的算法. 我们还将说明，SAM 需要满足的所有性质，除了最小性以外都满足了；而最小性可以由 Myhill–Nerode 定理→13.8@p3284 得出.

由 endpos 的值我们可以得到一些重要结论，它们解释了同一个状态对应的不同的子串之间的关系.

引理 1

字符串 s 的两个非空子串 u 和 w（假设 $|u| \le |w|$）的 endpos 相同，当且仅当字符串 u 在 s 中每次出现时，都是以 w 的后缀的形式存在的.

证明

引理显然成立. 如果 u 和 w 的 endpos 相同，则 u 是 w 的一个后缀，且在 s 中只以 w 的后缀的形式出现. 反过来，根据定义，如果 u 为 w 的一个后缀，且只以 w 的后缀的形式在 s 中出现，那么两个子串的 endpos 相同.

引理 2

考虑两个非空子串 u 和 w（假设 $|u| \le |w|$）. 那么，要么 $\text{endpos}(u) \cap \text{endpos}(w) = \emptyset$，要么 $\text{endpos}(w) \subseteq \text{endpos}(u)$，取决于 u 是否为 w 的一个后缀：

$$\begin{cases} \text{endpos}(w) \subseteq \text{endpos}(u), & \text{if } u \text{ is a suffix of } w, \\ \text{endpos}(w) \cap \text{endpos}(u) = \emptyset, & \text{otherwise.} \end{cases}$$

证明

如果集合 $\text{endpos}(u)$ 与 $\text{endpos}(w)$ 有至少一个公共元素，那么由于字符串 u 与 w 在相同位置结束，u 是 w 的一个后缀. 所以在每次 w 出现的位置，子串 u 也会出现. 所以 $\text{endpos}(w) \subseteq \text{endpos}(u)$.

引理 3

考虑一个 endpos 相同的子串等价类，将类中的所有子串按长度递减的顺序排序. 那么，每个子串都比它前一个子串恰好短 1，与此同时每个子串也是它前一个子串的后缀. 换句话说，对于同一等价类的任意两子串，较短者为较长者的后缀，且该等价类中的子串长度是连续的，取遍某个区间内的所有整数值.

[1] 需要将每个状态都取作一个 endpos 等价类的原因，其实就是下一段提到的 Myhill–Nerode 定理. 简单来说，如果两个字符串 t 和 u 的 endpos 集合不同，那么它们不能对应于 SAM 的同一个状态：同一个状态到达终止状态的路径总是一样的，这意味着在 t 和 u 末尾添加字符到达 s 的结尾的方式也是一样的，而这正说明 t 和 u 在字符串 s 中的结束位置一样. 反过来，只要两个字符串 t 和 u 的 endpos 集合相同，就可以将它们对应到 SAM 的同一个状态. 这样做可行，就是 Myhill–Nerode 定理的证明的内容，在此不多讨论. 但是，此处的讨论至少可以相信，将 endpos 集合相同的字符串放到同一个状态，这样得到的 SAM 一定是最小的，因为进一步合并节点是不可能的.

证明

如果等价类中只包含一个子串，引理显然成立．现在我们来讨论子串元素个数大于 1 的等价类．

由引理 1，endpos 相同的两个不同字符串中，必定一长一短，且较短者总是较长者的真后缀．也就是说，等价类中没有等长的字符串．

记 w 为等价类中最长的字符串，u 为等价类中最短的字符串．由引理 1，字符串 u 是字符串 w 的真后缀．现在考虑长度在区间 $[|u|, |w|]$ 中的 w 的任意后缀．容易看出，这个后缀也在同一等价类中，因为这个后缀只能在字符串 s 中以 w 的一个后缀的形式存在（这是因为较短的后缀 u 在 s 中只以 w 的后缀的形式存在）．因此，由引理 1，这个后缀和字符串 w 的 endpos 相同．

一句话概括，同一个状态对应的子串的长度各不相同，而且是连续的若干自然数，其中较短的总是较长的子串的后缀．

后缀链接 `link`

考虑 SAM 中某个状态 $v \neq t_0$．我们已经知道，状态 v 对应于具有相同 endpos 的子串等价类．我们如果定义 w 为这些字符串中最长的一个，则所有其它的字符串都是 w 的后缀．

我们还知道字符串 w 的前几个后缀（按长度降序考虑）全部属于这个等价类，且其它后缀（至少有一个——空后缀）在别的等价类中．我们记 t 为其它后缀中最长的，然后将 v 的后缀链接连到 t 所属等价类对应的状态上．

换句话说，v 的 **后缀链接** $\text{link}(v)$ 连接到的状态，对应于 w 的后缀中与它的 endpos 集合不同且最长的那个，也是 w 的后缀中在 s 中的出现次数比 w 更多且最长的那个．

为方便讨论，我们规定初始状态 t_0 对应的等价类，只包含一个空字符串，而且 $\text{endpos}(t_0) = \{-1, 0, ..., |s| - 1\}$．

引理 4

所有后缀链接构成一棵根节点为 t_0 的树．

证明

考虑任意状态 $v \neq t_0$，后缀链接 $\text{link}(v)$ 连接到的状态对应于严格更短的字符串（后缀链接的定义、引理 3）．因此，沿后缀链接移动，我们总是能到达对应空串的初始状态 t_0．

引理 5

以 endpos 集合为结点、集合的包含关系作为边，这样构造的树（即每个子节点的 endpos 集合都包含在父节点的 endpos 集合中）与通过后缀链接 link 构造的树相同．

证明

由引理 2，任意一个 SAM 的 endpos 集合形成了一棵树（因为两个集合要么完全没有交集要么其中一个是另一个的子集）．

我们现在考虑任意状态 $v \neq t_0$ 及后缀链接 $\text{link}(v)$，由后缀链接和引理 2，我们可以得到

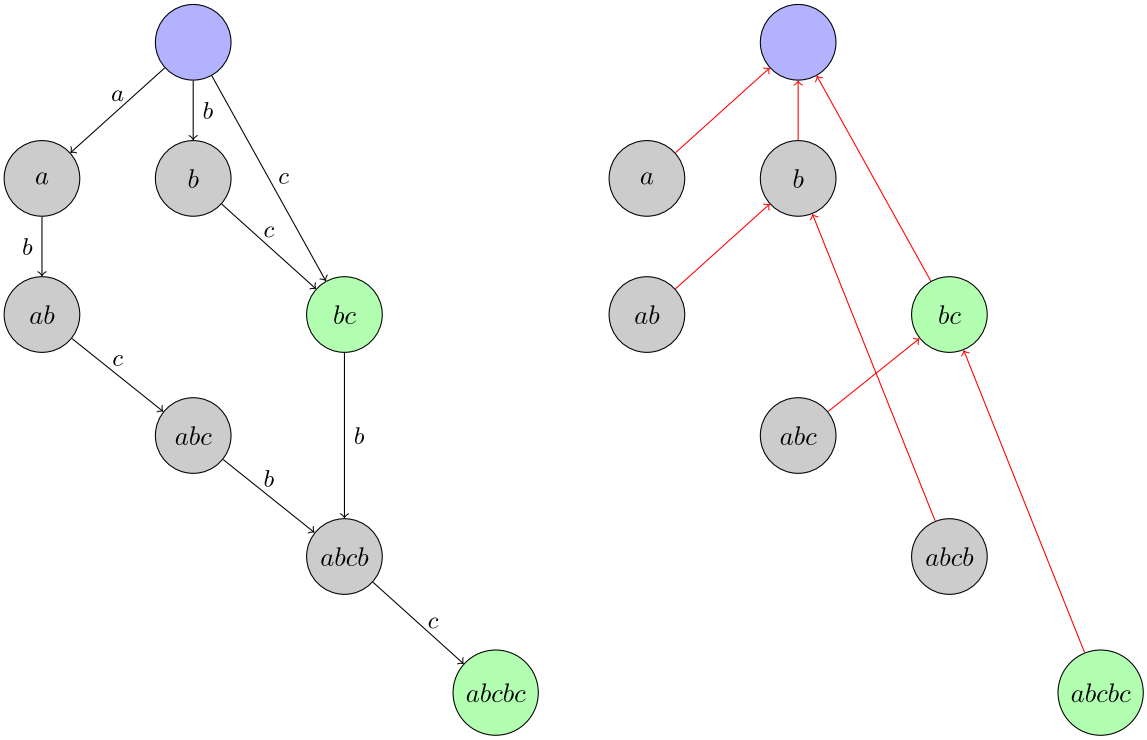
$$\text{endpos}(v) \subsetneq \text{endpos}(\text{link}(v)).$$

注意这里应该是 $\subsetneq$ 而不是 $\subset$，因为若 $\text{endpos}(v) = \text{endpos}(\text{link}(v))$，那么 v 和 $\text{link}(v)$ 应该被合并为一个结点．

而且，endpos(link(v)) 恰为真包含 endpos(v) 的最小的 endpos 集合.

结合前面的引理有：后缀链接构成的树本质上是 endpos 集合构成的一棵树.

以下是对字符串 abcbc 构造 SAM 时产生的后缀链接树的一个 **例子**，结点被标记为对应等价类中最长的子串.



结合图示，如果能够形成一些对于后缀自动机的认识，将对下文理解其构造算法和应用都有所帮助.

对图示的解释

- SAM 上存在一条最长的路径，其标号恰好为字符串 abcbc 本身. 该路径从初始状态开始，经过的每个状态都对应着字符串 abcbc 的前缀（ε, a, ab, abc, abcb, abcbc）. 这些状态在后续 应用→8@p889 中至关重要.
- 后缀链接树可以看做是将这些「前缀状态」沿着后缀链接移动到根节点（即初始状态）的路径「压缩」得到.
  - 沿着每条路径，结点对应的字符串集合构成了相应的前缀的所有后缀的分划. 例如，标记为 abcbc 的状态沿着后缀链接移动到根节点的路径为 abcbc $\to$ bc $\to \varepsilon$. 其中，结点 abcbc 实际对应着字符串集合 {abcbc, bcbc, cbc}，结点 bc 实际对应着字符串集合 {bc, c}，结点 ε 就对应空字符串.
  - 不同的路径可能共用同一个结点，这就是为什么会有「压缩」. 例如，路径 abc $\to$ bc $\to \varepsilon$ 和路径 abcbc $\to$ bc $\to \varepsilon$ 共用了结点 bc. 这是因为 endpos(bc) $= \{2, 4\}$，而结束在位置 2 的字符串 bc 前紧接着字符 a 而结束在位置 4 的字符串 bc 前紧接着字符 c，因此，当在前方添加字符（即逆着后缀链接移动）时，结束位置集合（即状态）会分裂.
  - 后缀链接树只需要将这些后缀路径合理地「压缩」在一起即可，而不需要考虑别的结点. 这是因为，所有子串都是某个前缀的后缀，故而必然出现在某个这样的路径中. 后文的

构造算法本质上就是逐个添加字符，并为每个新增加的前缀，构造这样一条后缀路径，并使其合理地「压缩」到之前已有的路径中（即不重复构造已经存在的状态和转移）.
    - 终止状态恰为字符串 abcbc 本身所在的后缀路径上的所有结点.
- 到达同一个状态的转移必然具有相同的标号，而且这些转移的起点一定是位于后缀链接树上的某条（连续的）路径. 比如，转移到状态 abcb 的状态就有两个：abc 和 bc. 它们位于后缀链接树上的路径 abc → bc 上. 注意，它们分别对应于字符串集合 {abc} 和 {bc, c}，这些字符串在后面添加字符 b，就得到状态 abcb 对应的字符串集合 {abcb, bcb, cb}.
    - 添加字符后，不同状态可能转移到同一个状态，是因为在末尾添加字符后，出现位置只会减少，原本不同的结束位置集合可能变得相同.
- 后缀链接树上，每个结点的 endpos 集合都是其子节点的 endpos 集合的并集，至多再增加一个位置. 而且，这个新位置存在，当且仅当该结点恰好对应着结束在该位置的原字符串的前缀. 因为图示中，后缀链接树的非根非叶的结点都不对应着字符串 abcbc 的前缀，所以不存在这种情形.

后缀自动机中存储着字符串全部子串的信息. 这件事可以通过两个角度理解：

- SAM 本身可以看作是字符串全体后缀的字典树的压缩版本. 因此，它存储了字符串的全体后缀的所有前缀的信息，这就相当于存储了字符串全体子串的信息.
- SAM 的后缀链接树可以看做是字符串全体前缀的后缀路径的压缩版本. 因此，它存储了字符串的全体前缀的所有后缀的信息，这也相当于存储了字符串全体子串的信息.

这两种思考的角度在处理不同问题时都是有用的.

小结

在进一步讨论算法本身前，我们总结一下之前的内容，并引入一些辅助记号.

- s 的子串可以根据它们的结束位置集合 endpos 划分为多个等价类；
- SAM 由初始状态 t_0 和与每一个（非空子串的）endpos 等价类对应的每个状态组成；
- 每一个状态 v 都匹配一个或多个子串. 我们记 $\text{longest}(v)$ 为其中最长的一个字符串，记 $\text{len}(v)$ 为它的长度. 类似地，记 $\text{shortest}(v)$ 为最短的子串，它的长度为 $\text{minlen}(v)$. 那么对应这个状态的所有字符串都是字符串 $\text{longest}(v)$ 的不同的后缀，且所有字符串的长度恰好取遍区间 $[\text{minlen}(v), \text{len}(v)]$ 中的每一个整数.
- 对于任意状态 $v \neq t_0$，定义后缀链接为连接到对应字符串 $\text{longest}(v)$ 的长度为 $\text{minlen}(v) - 1$ 的后缀的一条边. 所有后缀链接可以形成一棵以 t_0 为根的树. 这棵树也表示 endpos 集合间的包含关系.
- 对于任意状态 $v \neq t_0$，可用后缀链接 $\text{link}(v)$ 表达 $\text{minlen}(v)$：
  $\text{minlen}(v) = \text{len}(\text{link}(v)) + 1$.
- 如果我们从任意状态 v_0 开始顺着后缀链接遍历，总会到达初始状态 t_0. 这种情况下我们可以得到一个互不相交的区间 $[\text{minlen}(v_i), \text{len}(v_i)]$ 的序列，且它们的并集形成了连续的区间 $[0, \text{len}(v_0)]$.

算法

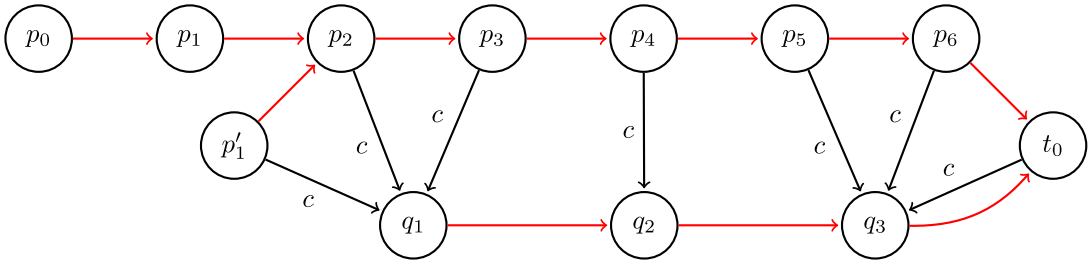
现在我们可以讨论构造 SAM 的算法了. 这个算法是 **在线** 算法，我们可以逐个加入字符串中的每个字符，并且在每一步中对应地维护 SAM.

在讨论详细的实现之前，首先通过图示初步感受一下增加新字符 c 时，SAM 可能发生的变化.

简单理解增量构造过程

在字符串 s 的 SAM 的基础上，可以构造字符串 $s + c$ 的 SAM. 根据前文对图示的解释，只需要构造出新增加的前缀（即 $s + c$）的后缀路径，并压缩到现有的路径上即可. 而且，根据前文的描述，新的后缀路径上的结点必然都可以通过原字符串 s 的后缀路径上的结点经由字符 c 转移而来.

我们首先考虑在添加新字符 c 之前，原来的字符串 s 的后缀路径可能具有什么形式，而且会怎样经由字符 c 转移．最一般的情形，如下图示：



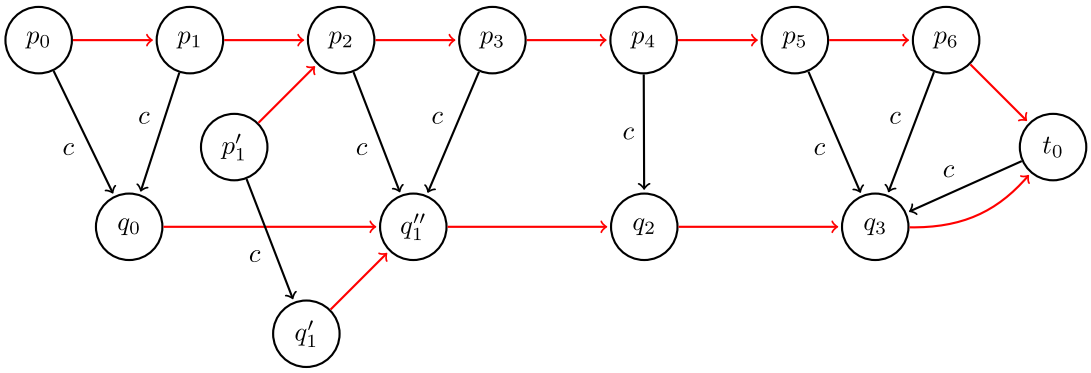
图中，原字符串 s 的后缀路径为 $p_0 \to p_1 \to \cdots \to p_6 \to t_0$，后缀链接由红色箭头表示．其中部分结点（即 $p_2 \sim p_6$）已经存在经由字符 c 的转移；因为将连续的后缀串添加同一个字符会同样得到连续的后缀，所以这些转移的终点组成另一串后缀路径 $q_1 \to q_2 \to q_3 \to t_0$．此时，有两点观察：

- 原字符串 s 的后缀路径上，没有经由 c 的转移的结点一定是最初的几个结点．只要从某个结点（图中的 p_2）开始，存在经由 c 的转移，后续经过的结点也一定存在经由 c 的转移．

  解释：设 $s_2 = \text{longest}(p_2)$，那么后续经过的所有结点都对应 s_2 的后缀，因而如果 $s_2 + c$ 也出现在 s 中，那么 s_2 的后缀再加 c 的结果也一定出现在 s 中，因此这些结点都有经由 c 的转移．
- 虽然经由字符 c 能够到达结点 q_i 的结点一定是后缀链接树上的连续段，但是这个连续段未必全体都位于自 p_0 到根的后缀路径上．特别地，只有第一个结点 q_1 对应的连续段中起始的若干个结点 **可能** 不在这个后缀路径上．例如图中的结点 q_1 就对应结点 $p_{(1)'} \to p_2 \to p_3$，其中，$p_{(1)'}$ 不在 p_0 的后缀路径上．

  解释：设 $s_2 = \text{longest}(p_2)$，则 $s_2 + c$ 对应着 q_1，但是图中显然有 $s_2 + c \neq \text{longest}(q_1)$，因为后者是 $\text{longest}(p'_1) + c$．这就说明，q_1 对应的部分字符串不能由 s_2 及其后缀转移来．反过来，q_2 中的字符串必然是 $s_2 + c$ 的后缀，因此删去末尾的 c 后必然是 s_2 的后缀．也就是说，经由 c 转移到 q_2 的结点必然在 p_2 起始的后缀路径上．这也是为什么只有起始的 q_1 对应的连续段中的部分结点可能不在 p_0 的后缀路径上．

对于这个图示，如果要在原字符串 s 的末尾添加一个字符 c，并构造出相应的后缀路径，会发生什么变化呢？答案是如下图所示：



因为结点 q_0 由原字符串 s 对应结点 p_0 经由字符 c 转移而来，它就对应新字符串 $s + c$．所以，它的后缀路径 $q_0 \to q_{(1)''} \to q_2 \to q_3 \to t_0$ 就是新增的后缀路径．如果原来的后缀路径上的结点 p_i 本就有经由 c 的转移，那么新的后缀路径也必然会经过这些转移到达的结点，因此可以直接复用旧有的结点．新的后缀路径上有且只有一个完全新建的节点 q_0，用于接受原来的后缀路径上起始的那些没有经由 c 的转移的结点的转移．

除了这些显然的事实外，还可以注意到，原来的结点 q_1 也经过了一次复制，或者说是分裂成了两个结点 ${q'}_1 \to q_{(1)''}$．这是因为新增的后缀路径只是与现有路径部分重合：原来的结点 q_1 对应的字符串中，只有较短的那些（即结点 p_2 和 p_3 能够转移到的那些）才会出现在新增的后缀路径上，而较长的那些（即结点 $p_{(1)'}$ 能够转移到的那些）并不会出现在新增的后缀路径上，因此新增的后缀路径只能经过结点 q_1 的一部分，后者只能分裂成两个结点用于表示这种情形．同样的道理，前面已经解释过，q_1 之后的结点 q_2 和 q_3 都无法由不在 p_0 的后缀路径上的结点转移，因此这些结点对应的所有字符串都会出现在新增的后缀路径上，也就不需要分裂了．

从SAM中状态代表的含义看，每个状态都是一个endpos集合．设延长字符串时，新增的结束位置为 i，那么新增的结点 q_0 就是结束位置集合 $\{i\}$，而分裂的结点 $q_{(1)'}$ 和 $q_{(1)''}$ 分别对应集合 $\text{endpos}(q_1)$ 和 $\text{endpos}(q_1) \cup \{i\}$，之后的结点 q_2 和 q_3 其实都在原有的结束位置集合上新增了 i．也就是说，虽然 q_2 和 q_3 及其相关的转移没有发生变化，但是它们对应的 endpos 集合的确扩大了．

以上说明的是最复杂、最一般的情形（即下文的 **情况三**）．实际操作时，可能并不存在结点 ${p'}_1$，因而也就不需要分裂（即下文的 **情况二**）．要判断这种情形，只需要判断 $\text{longest}(q_1) = \text{longest}(p_2) + c$ 即可，亦即 $\text{len}(q_1) = \text{len}(p_2) + 1$．也有可能 p_0 的后缀路径上的所有结点都没有经由 c 的转移，此时，只要新建 q_0 就好了（即下文的 **情况一**）．

掌握了新增后缀路径的思想后，现在讨论增量构造的具体步骤．

过程

为了保证线性的空间复杂度，我们将只保存 len 和 link 的值和每个状态的转移列表，我们不会标记终止状态（但是我们稍后会展示在构造 SAM 后如何分配这些标记）．

一开始 SAM 只包含一个状态 t_0，编号为 0（其它状态的编号为 $1, 2, \ldots$）．为了方便，对于状态 t_0 我们指定 $\text{len}(t_0) = 0$，$\text{link}(t_0) = -1$（-1 表示虚拟状态）．

现在，只需要实现给当前字符串添加一个字符 c 的过程．算法流程如下：

SAM 增量构造过程

- 令 $last$ 为添加字符 c 之前，整个字符串对应的状态（一开始我们设 $last = 0$，算法的最后一步更新 $last$）．
- 创建一个新的状态 cur，并将 $\text{len}(cur)$ 赋值为 $\text{len}(last) + 1$，在这时 $\text{link}(cur)$ 的值还未知．
- 现在我们进行如下流程：从状态 $last$ 开始，如果当前状态还没有标号为字符 c 的转移，我们就添加一个经字符 c 到状态 cur 的转移，并将当前状态沿后缀链接移动．如果过程中遇到某个状态已经存在到字符 c 的转移，我们就停下来，并将这个状态标记为 p．
- **情况一**：如果没有找到这样的状态 p，我们就到达了虚拟状态 -1，我们将 $\text{link}(cur)$ 赋值为 0，然后转到最后一步．
- 假设现在我们找到了一个状态 p，它可以通过字符 c 转移．我们将转移到的状态标记为 q．此时，要么 $\text{len}(p) + 1 = \text{len}(q)$，要么 $\text{len}(p) + 1 < \text{len}(q)$．
- **情况二**：如果 $\text{len}(p) + 1 = \text{len}(q)$，我们只要将 $\text{link}(cur)$ 赋值为 q，然后转到最后一步．
- **情况三**：否则就会有些复杂，需要 **复制** 状态 q：我们创建一个新的状态 $clone$，复制 q 的除了 len 的值以外的所有信息（后缀链接和转移）．我们将 $\text{len}(clone)$ 赋值为 $\text{len}(p) + 1$．

  复制之后，我们将后缀链接从 cur 指向 $clone$，也从 q 指向 $clone$．

  最终我们需要沿着后缀链接从状态 p 往回走，只要经过的状态存在指向状态 q 的转移，就将该转移重新连接到状态 $clone$．
- 处理完以上三种情况后，我们都需要将 $last$ 的值更新为状态 cur．

如果我们还想知道哪些状态是 **终止状态** 而哪些不是，我们可以在为字符串 s 构造完完整的 SAM 后找到所有的终止状态. 为此，我们从对应整个字符串的状态（存储在变量 $last$ 中），遍历它的后缀链接，直到到达初始状态. 我们将所有遍历到的状态都标记为终止状态. 容易理解这样做我们会准确地标记字符串 s 的所有后缀，这些状态都是终止状态.

因为我们只为 s 的每个字符创建一个或两个新状态，所以 SAM 只包含 **线性个** 状态. 而 SAM 只有线性规模的转移个数，以及算法总体的线性运行时间，都还没有说清楚，将在后文说明.

解释

我们详细解释算法每一步的细节，并说明它的 **正确性**.

对算法的详细解释

- 若一个转移 (p, q) 满足 $\text{len}(p) + 1 = \text{len}(q)$，则我们称这个转移是 **连续的**. 否则，即当 $\text{len}(p) + 1 < \text{len}(q)$ 时，这个转移被称为 **不连续的**.

  从算法描述中可以看出，连续的和不连续的转移，在算法中的处理也并不相同. 连续的转移是固定的，我们不会再改变了. 与此相反，当向字符串中插入一个新的字符时，不连续的转移可能会改变（转移边的端点可能会改变）.
- 为了避免引起歧义，我们记向 SAM 中插入当前字符 c 之前的字符串为 s.
- 算法从创建一个新状态 cur 开始，对应于整个字符串 $s + c$. 我们创建一个新的节点的原因很清楚. 与此同时我们也创建了一个新的等价类.
- 在创建一个新的状态之后，我们会从对应整个字符串 s 的状态 $last$ 沿着后缀链接进行移动. 对于经过的每一个状态，我们尝试添加一个通过字符 c 到新状态 cur 的转移.

  然而我们只能添加与原有转移不冲突的转移. 因此我们只要找到已存在的 c 的转移，我们就必须停止.
- 最简单的情况是我们到达了虚拟状态 -1，这意味着我们为所有 s 的后缀添加了 c 的转移. 这也意味着，字符 c 从未在字符串 s 中出现过. 因此 cur 的后缀链接为状态 0.
- 否则（即情况二和情况三），我们找到了现有的转移 (p, q). 这意味着我们尝试向自动机内添加一个 **已经存在的** 字符串 $x + c$（其中 $x = \text{longest}(p)$ 为 s 的一个后缀，且字符串 $x + c$ 已经作为 s 的一个子串出现过了）. 因为我们假设字符串 s 的自动机的构造是正确的，我们不应该在这里添加一个新的转移.

  然而，难点在于，从状态 cur 出发的后缀链接应该连接到哪个状态呢? 我们要把后缀链接连到一个状态上，且对应的最长的字符串恰好是 $x + c$，即这个状态的 len 应该是 $\text{len}(p) + 1$. 然而这样的状态有可能并不存在，即 $\text{len}(q) > \text{len}(p) + 1$. 这种情况下，我们必须通过拆开状态 q 来创建一个这样的状态.
- 当然，如果转移 (p, q) 是连续的，那么 $\text{len}(q) = \text{len}(p) + 1$. 在这种情况下一切都很简单. 我们只需要将 cur 的后缀链接指向状态 q.
- 否则转移是不连续的，即 $\text{len}(q) > \text{len}(p) + 1$，这意味着状态 q 不只对应于长度为 $\text{len}(p) + 1$ 的后缀 $x + c$，还对应于 s 的更长的子串. 除了将状态 q 拆成两个子状态以外我们别无他法，所以第一个子状态的长度就是 $\text{len}(p) + 1$ 了.

  我们如何拆开一个状态呢? 我们 **复制** 状态 q，产生一个状态 $clone$，我们将 $\text{len}(clone)$ 赋值为 $\text{len}(p) + 1$. 由于我们不想改变经过 q 的路径，我们将 q 的所有转移复制到 $clone$. 我们也将从 $clone$ 出发的后缀链接设置为 q 的后缀链接的目标，并设置 q 的后缀链接为 $clone$.

  在拆开状态后，我们将从 cur 出发的后缀链接设置为 $clone$.

  最后一步我们将一些原本指向 q 的转移重新连接到 $clone$. 我们需要修改哪些转移呢? 只需重新连接对应于字符串 $x + c$（其中 $x = \text{longest}(p)$）的所有后缀的转移就够了. 也就是说，我们

> 需要继续沿着后缀链接移动，从结点 p 直到虚拟状态 -1，或者当前状态经 c 的转移不再指向状态 q.

线性时间复杂度

我们假设字符集大小为 **常数**，即每次对一个字符搜索转移、添加转移、查找下一个转移这些操作的时间复杂度都为 $O(1)$ 的. 如果将每个结点的转移分别存储为一个长度为 $|\Sigma|$ 的数组（用于快速查询给定标号的转移）和一个动态列表（用于快速遍历所有可用转移），以空间换时间，那么算法的时间复杂度[2]为 $O(n)$，空间复杂度为 $O(n|\Sigma|)$.

证明

如果我们考虑算法的各个部分，算法中有三处时间复杂度不明显是线性的：

- 第一处是遍历所有状态 $last$ 的后缀链接，添加字符 c 的转移.
- 第二处是当状态 q 被复制到一个新的状态 $clone$ 时复制转移的过程.
- 第三处是修改指向 q 的转移，将它们重新连接到 $clone$ 的过程.

我们使用 SAM 的大小（状态数和转移数）为 **线性** 的事实（对状态数是线性的证明就是算法本身，对转移数为线性的证明将在稍后实现算法后给出）.

因此上述 **第一处和第二处** 的总复杂度显然为线性的，因为单次操作均摊只为自动机添加了一个新转移.

还需为 **第三处** 估计总复杂度，我们将最初指向 q 的转移重新连接到 $clone$. 我们记 $v = \text{longest}(p)$，这是字符串 s 的一个后缀. 每迭代一次，v 的长度都减小，因而 v 作为 s 的后缀的起始位置必然在后移. 因此，循环中 p 沿后缀链接移动的次数，不超过 v 作为 s 的后缀的起始位置向后移动的距离. 因为 p 至少要向后移动一次，才能终止循环，而且 p 至少是 $last$ 沿后缀链接移动一次的结果，因此循环终止时，v 作为 s 的后缀的起始位置并不比字符串 $\text{longest}(\text{link}(\text{link}(last)))$ 更靠前. 而且，循环终止时，字符串 v 作为 s 的后缀的起始位置将恰好是 $v + c$ 作为 $s + c$ 的后缀的起始位置，而作为 $s + c$ 的后缀，字符串 $v + c$ 恰好是字符串 $\text{longest}(\text{link}(\text{link}(cur)))$. 因为 cur 是更新后的 $last$ 的值，所以循环中移动的次数不会超过更新前后 $\text{longest}(\text{link}(\text{link}(last)))$ 作为当前字符串后缀的起始位置向后移动的距离，再加一（即为了终止循环必须移动的次数）.

因为作为当前字符串后缀的字符串 $\text{longest}(\text{link}(\text{link}(last)))$ 的位置在整个 SAM 构造过程中单调递增[3]，它的总移动距离必然不超过 n. 这就说明，需要修改指向 q 的转移的循环中，迭代次数不超过 $2n$. 这正是我们需要证明的.

当然，如果字符集大小不是常数，又不愿付出 $O(n|\Sigma|)$ 的空间，SAM 的时间复杂度就不是线性的. 从一个结点出发的转移需要存储在支持快速查询和插入的平衡树中. 因此如果我们记 Σ 为字符集，$|\Sigma|$ 为字符集大小，则算法的渐近时间复杂度为 $O(n \log|\Sigma|)$，空间复杂度为 $O(n)$.

---

[2] 如果不额外使用列表记录当前状态的可用转移，只用数组存储所有可能的转移（无论是否存在）并在复制节点时直接复制，那么时间复杂度也是 $O(n|\Sigma|)$ 的.

[3] 此处正文没有解释的是，在情况一和情况二中，$\text{longest}(\text{link}(\text{link}(last)))$ 的位置是否也是单调（弱）递增的. 情况一容易验证，因为更新后 $\text{link}(last) = t_0$，而 $\text{link}(\text{link}(last))$ 是虚拟状态 -1，不妨将它对应的字符串视为位于当前字符串末尾的空串，其位置取到最大值. 情况二，转移是连续的，说明 $\text{longest}(q) = \text{longest}(p) + c$. 然而，向子串的末尾添加新的字符只会使得该子串更难以出现在字符串中，也就是说，当字符串 $\text{longest}(p)$ 的长度为 $\text{len}(\text{link}(p))$ 的后缀的结束位置集合严格包含 $\text{endpos}(p)$ 时，字符串 $\text{longest}(q)$ 的长度为 $\text{len}(\text{link}(p)) + 1$ 的后缀的结束位置集合可能仍然与 $\text{endpos}(q)$ 相同. 故而，$\text{len}(\text{link}(q)) \le \text{len}(\text{link}(p)) + 1$，亦即 $\text{longest}(\text{link}(p))$ 作为 s 的后缀的起始位置必然不大于 $\text{longest}(\text{link}(q))$ 作为 $s + c$ 的后缀的起始位置. 而当首次找到状态 p 使得存在经由 c 的转移时，必定移动了至少一次，这说明 $\text{longest}(\text{link}(p))$ 作为 s 的后缀的起始位置不小于 $\text{longest}(\text{link}(\text{link}(last)))$ 作为 s 的后缀的起始位置. 最后，$\text{longest}(\text{link}(q)) = \text{longest}(\text{link}(\text{link}(cur)))$. 这就说明，在情况二中，$\text{longest}(\text{link}(\text{link}(last)))$ 的位置也是单调（弱）递增的.

实现

首先，我们实现一种存储一个转移的全部信息的数据结构．如果需要的话，你可以在这里加入一个终止标记，也可以是一些其它信息．我们将用一个 `map` 存储转移的列表，允许我们在总计 $O(n)$ 的空间复杂度和 $O(n \log|\Sigma|)$ 的时间复杂度内处理整个字符串．当然，在字符集大小 $|\Sigma| = K$ 为较小的常数（比如 26）时，将 `next` 声明为 `int[K]` 更方便．

```
struct state {
  int len, link;
  std::map<char, int> next;
};
```

SAM 本身将会存储在一个 `state` 结构体数组中．我们记录当前自动机的大小 `sz` 和变量 `last`，当前整个字符串对应的状态．

```
constexpr int MAXLEN = 100000;
state st[MAXLEN * 2];
int sz, last;
```

我们定义一个函数来初始化 SAM（创建一个只有初始状态的 SAM）．

```
void sam_init() {
  for (int i = 0; i < sz; i++) st[i].next.clear();
  st[0].len = 0;
  st[0].link = -1;
  sz = 1;
  last = 0;
}
```

最终我们给出主函数的实现：给当前字符串末尾增加一个字符，对应地在之前的基础上建造自动机．

实现

```
void sam_extend(char c) {
  int cur = sz++;
  st[cur].len = st[last].len + 1;
  int p = last;
  while (p != -1 && !st[p].next.count(c)) {
    st[p].next[c] = cur;
    p = st[p].link;
  }
  if (p == -1) {
    st[cur].link = 0;
  } else {
    int q = st[p].next[c];
    if (st[p].len + 1 == st[q].len) {
      st[cur].link = q;
    } else {
      int clone = sz++;
      st[clone].len = st[p].len + 1;
      st[clone].next = st[q].next;
      st[clone].link = st[q].link;
      while (p != -1 && st[p].next[c] == q) {
        st[p].next[c] = clone;
        p = st[p].link;
      }
```

```
24        st[q].link = st[cur].link = clone;
25      }
26    }
27    last = cur;
28  }
```

正如之前提到的一样，如果你用内存换时间（空间复杂度为 $O(n|\Sigma|)$，其中 $|\Sigma|$ 为字符集大小），你可以在 $O(n)$ 的时间[4]内构造字符集大小任意的 SAM．但是这样你需要为每一个状态储存一个大小为 $|\Sigma|$ 的数组（用于快速根据字符找到相应的转移）以及一个包含所有可用转移的列表（用于快速遍历所有可用的转移）．

更多性质

状态数

对于一个长度为 n 的字符串 s，它的 SAM 中的状态数 **不会超过** $2n-1$（假设 $n \geq 2$）．

证明

算法本身即可证明该结论．一开始，自动机含有一个状态，第一次和第二次迭代中只会创建一个节点，剩余的 $n-2$ 步中每步会创建至多 2 个状态．

然而我们也能在 **不借助这个算法** 的情况下 **证明** 这个估计值．若 s 由同一个字符构成，则状态数为 $n+1$，结论显然；下面设 s 中至少有两种字符．我们回忆一下状态数等于不同的 endpos 集合个数．这些 endpos 集合形成了一棵树（父节点的 endpos 集合包含子节点的 endpos 集合）．考虑将这棵树稍微变形一下：只要它有一个只有一个子节点的内部节点（这意味着该子节点的集合至少遗漏了它的父节点的集合中的一个位置），我们就创建一个含有这些遗漏位置的集合作为它的子节点．最后我们可以获得一棵每一个内部结点的子节点数都大于一的树，且叶子节点的个数不超过 n．这样的树里有不超过 $2n-1$ 个节点，因此，原来的不同的 endpos 集合个数也不超过 $2n-1$．

字符串 `abbb` $\cdots$ `bbb` 的状态数达到了该上界：从第三次迭代开始的每次迭代，算法都会拆开一个状态，最终产生恰好 $2n-1$ 个状态．

转移数

对于一个长度为 n 的字符串 s，它的 SAM 中的转移数 **不会超过** $3n-4$（假设 $n \geq 3$）．

证明

我们首先估计连续的转移的数量．考虑自动机中由从状态 t_0 开始到达所有状态的最长路径组成的生成树．生成树只包含连续的边，因此数量少于状态数，即边数不会超过 $2n-2$．

现在我们来估计不连续的转移的数量．令当前不连续转移为 (p,q)，其字符为 c．我们取它的对应字符串 $u+c+w$，其中字符串 u 对应于初始状态到 p 的最长路径，w 对应于从 q 到任意终止状态的最长路径．一方面，每个不连续的转移所对应的形如 $u+c+w$ 的字符串是不同的（因为 u 仅由连续的转移组成，沿 $u+c+w$ 经过的第一个不连续的转移恰为 (p,q)）．另一方面，由终止状态的定义，每个形如 $u+c+w$ 的字符串都是整个字符串 s 的后缀．因为 s 只有 n 个非空后缀，且形如 $u+c+w$ 的字符串都不等于 s（因为 s 对应的路径只包含连续的转移），所以不连续的转移的总数不会超过 $n-1$．

[4]如果不额外使用列表记录当前状态的可用转移，只用数组存储所有可能的转移（无论是否存在）并在复制节点时直接复制，那么时间复杂度也是 $O(n|\Sigma|)$ 的．

将以上两个估计值相加，我们可以得到上界 $3n-3$．然而，最大的状态数只能在类似于 abbb ⋯ bbb 的情况中产生，而此时转移数量显然少于 $3n-3$．

因此我们可以获得更为紧确的 SAM 的转移数的上界：$3n-4$．字符串 abbb ⋯ bbbc 就达到了这个上界．

后缀链接树

尽管构造 SAM 是为了得到它的状态和转移的信息，但是构造过程中记录的后缀链接 link 和该状态对应的最长子串长度 len 在应用中常常比 SAM 的转移更为重要，甚至可以抛开转移单独使用．

在构建 SAM 的过程中，需要更新 $last$ 状态的值．它对应的是每次添加字符前（后）的字符串，也就是整个字符串 s 的所有前缀．将第 i 个前缀对应的状态记为 v_i，这样就得到 $v_0, v_1, \cdots, v_{n-1}$ 共计 n 个状态．另外，规定初始状态 t_0 为 v_{-1}，对应着空前缀．这些状态姑且称为「前缀节点」．

引理 4 中提及，所有状态和所有后缀链接构成根为 t_0 的根向树，这个树也称为 **后缀链接树**（国内 OI 选手也常称它为 **parent 树**）．它记录了字符串全体前缀的所有后缀的信息，亦即全体子串的信息．

后缀链接树有如下性质：

- 祖先节点对应的字符串总是子孙节点对应的字符串的后缀．
- 每个节点处的 endpos 集合就是它的子树内的所有「前缀节点」v_i 的下标 i 的集合．
- 后缀链接树的祖先节点的 endpos 集合总是严格包含子孙节点的 endpos 集合．
- 每个节点处的 len 的值就是它的子树内的所有「前缀节点」v_i 对应前缀的最长公共后缀的长度．
- 除根节点 t_0 外，每个节点对应的不同子串的数目，就是它的 len 值，减去它的父节点的 len 值，即 $\text{len}(v) - \text{len}(\text{link}(v))$．

这些性质有很多应用．比如，第 i 个前缀和第 j 个前缀的最长公共后缀对应的字符串就是 v_i 和 v_j 的 LCA 对应的最长字符串．

最后，对字符串 s 建立的后缀链接树与对它的翻转 s_R 建立的 后缀树→8.15@p1038 有相同的结构．这一点常常用于离线构造后缀树．

应用

下面我们来看一些可以用 SAM 解决的问题．简单起见，假设字符集的大小 $|\Sigma|$ 为常数．这允许我们认为增加一个字符和遍历的复杂度为常数．

检查字符串是否出现

问题

给一个文本串 T 和多个模式串 P，我们要检查字符串 P 是否作为 T 的一个子串出现．

解法

我们在 $O(|T|)$ 的时间内对文本串 T 构造后缀自动机．为了检查模式串 P 是否在 T 中出现，我们沿转移（边）从 t_0 开始根据 P 的字符进行转移．如果在某个点无法转移下去，则模式串 P 不是 T 的一个子串．如果我们能够这样处理完整个字符串 P，那么模式串在 T 中出现过．

对于每个字符串 P，算法的时间复杂度为 $O(|P|)$．此外，这个算法还找到了模式串 P 在文本串中出现的最大前缀长度．

不同子串个数

问题

给一个字符串 S，计算不同子串的个数.

解法一

对字符串 S 构造后缀自动机.

每个 S 的子串都相当于自动机中的一条路径. 因此不同子串的个数等于自动机中以 t_0 为起点的不同非空路径的条数.

考虑到 SAM 为有向无环图，不同路径的条数可以通过动态规划计算. 即令 d_v 为从状态 v 开始的路径数量（包括长度为零的路径），则我们有如下递推方程：

$$d_v = 1 + \sum_{w:(v,w,c)\in \text{ SAM}} d_w$$

即，d_v 等于 1 加上 v 的所有转移的终点的 d 值之和，其中 $(v,w,c) \in \text{SAM}$ 表示后缀自动机中存在自 v 经 c 至 w 的转移.

所以不同子串的个数为 $d_{t_0} - 1$（因为要去掉空子串）.

总时间复杂度为：$O(|S|)$.

解法二

另一种方法是在构造完后缀自动机后, 利用得到的后缀链接树的信息. 每个节点对应的子串数量是 $\text{len}(v) - \text{len}(\text{link}(v))$，对除 t_0 外的所有节点求和即可.

总时间复杂度仍然为：$O(|S|)$.

例题：【模板】后缀自动机[1]，SDOI2016 生成魔咒[2]

所有不同子串的总长度

问题

给定一个字符串 S，计算所有不同子串的总长度.

解法一

本题做法与上一题类似，只是现在我们需要考虑分两部分进行动态规划：不同子串的数量 d_v 和它们的总长度 ans_v.

我们已经在上一题中介绍了如何计算 d_v. ans_v 的值可以通过以下递推式计算：

$$ans_v = \sum_{w:(v,w,c)\in \text{ SAM}} (d_w + ans_w)$$

我们取每个邻接结点 w 的答案，并加上 d_w（因为从状态 v 出发的子串都增加了一个字符）.

算法的时间复杂度仍然是 $O(|S|)$.

解法二

同样可以利用后缀链接树的信息. 每个节点对应的最长子串的所有后缀长度之和是

$$\frac{\text{len}(v) \times (\text{len}(v)+1)}{2},$$

减去其 link 节点的对应值就是该节点的净贡献，对除 t_0 外的所有节点求和即可.

总时间复杂度仍然为：$O(|S|)$.

字典序第 k 小子串

问题

给定一个字符串 S. 多组询问，每组询问给定一个数 K_i，查询 S 的所有本质不同的子串中字典序第 K_i 小的子串.

解法

解决这个问题的思路可以从解决前两个问题的思路发展而来. 字典序第 K_i 小的子串对应于 SAM 中字典序第 K_i 小的非空路径，因此在计算每个状态的路径数后，我们可以很容易地从 SAM 的根开始找到第 K_i 小的路径.

预处理的时间复杂度为 $O(|S|)$，单次查询的复杂度为 $O(|ans| \cdot |\Sigma|)$（其中 ans 是查询的答案，$|\Sigma|$ 为字符集的大小）.

另注

虽然该题是后缀自动机的经典题，但实际上这题由于涉及字典序，用后缀数组做最方便.

例题：SPOJ - SUBLEX[3]， TJOI2015 弦论[4]

最小循环移位

问题

给定一个字符串 S. 找出字典序最小的循环移位.

解法

容易发现字符串 $S+S$ 包含字符串 S 的所有循环移位作为子串.

所以问题简化为在 $S+S$ 对应的后缀自动机上寻找最小的长度为 $|S|$ 的路径，这可以通过平凡的方法做到：我们从初始状态开始，贪心地访问最小的字符即可.

总的时间复杂度为 $O(|S|)$.

出现次数

问题

对于一个给定的文本串 T，有多组询问，每组询问给一个模式串 P，回答模式串 P 在字符串 T 中作为子串出现了多少次.

解法一

利用后缀链接树的信息，进行 dfs 即可预处理每个节点的 endpos 集合的大小.

所有「前缀节点」的初始集合大小为 1，非「前缀节点」的初始集合大小为 0. 然后，沿着后缀链接自下向上回溯时, 每个父节点的集合大小都加上它的所有子节点的集合大小（不要遗漏父节点本身的初始值）. 这样得到的每个节点处的值，就是该节点的 endpos 集合的大小. 不同子节点的集合大小可以直接相加的理由是，同一个 v_i 只会出现在一个子树内，故而相加不会重复.

查询时，在自动机上查找模式串 P 对应的节点，如果存在，则答案就是该节点的 endpos 集合的大小；如果不存在，则答案为 0.

预处理时间复杂度为 $O(|\ T\ |)$. 单次查询的时间复杂度为 $O(|\ P\ |)$.

解法二

对文本串 T 构造后缀自动机.

接下来做预处理: 对于自动机中的每个状态 v, 预处理 cnt_v, 使之等于 $\text{endpos}(v)$ 集合的大小. 事实上, 对应同一状态 v 的所有子串在文本串 T 中的出现次数相同, 这相当于集合 endpos 中的位置数.

然而我们不能明确地构造集合 endpos，因此我们只考虑它们的大小 cnt.

为了计算这些值，我们进行以下操作. 对于每个状态，如果它不是通过复制创建的（且它不是初始状态 t_0），我们将它的 cnt 初始化为 1. 然后我们按它们的长度 len 降序遍历所有状态，并将当前的 cnt_v 的值加到后缀链接指向的状态上，即:

$$cnt_{\text{link}(v)} \mathrel{+}= cnt_v$$

这样做每个状态的答案都是正确的.

为什么这是正确的? 不是通过复制获得的状态（除 t_0 外）恰好有 $|T|$ 个，并且它们中的前 i 个在我们插入前 i 个字符时产生, 各自对应一个结束位置. 因此我们初始将这些状态的 cnt 的值赋为 1, 其它状态的 cnt 值赋为 0.

接下来我们对每一个 v 执行以下操作: $cnt_{\text{link}(v)} \mathrel{+}= cnt_v$. 其背后的含义是，如果状态 v 对应的字符串出现了 cnt_v 次，那么它的所有后缀也在完全相同的地方结束，即也出现了 cnt_v 次.

为什么我们在这个过程中不会重复计数（即把某些位置数了两次）呢? 因为我们只将一个状态的位置添加到 **一个** 其它的状态上，所以一个状态不可能以两种不同的方式将其位置重复地指向另一个状态.

因此，我们可以在 $O(|T|)$ 的时间内计算出所有状态的 cnt 的值.

最后回答询问只需要查找值 cnt_t, 其中 t 为模式串对应的状态, 如果该模式串不存在答案就为 0. 单次查询的时间复杂度为 $O(|P|)$.

第一次出现的位置

问题

给定一个文本串 T，多组查询. 每次查询字符串 P 在字符串 T 中第一次出现的位置（P 的开头位置）.

解法一

利用后缀链接树的信息，进行 dfs 即可预处理每个节点的 endpos 集合中的最小值.

所有「前缀节点」v_i 的初始值为 i，非「前缀节点」的初始值为 ∞. 然后，沿着后缀链接自下向上回溯时，每个父节点的值都与它的所有子节点的值比较，取最小值（不要遗漏父节点本身的初始值）. 这样得到的每个节点处的值，就是该节点的 endpos 集合中的最小值.

查询时，在自动机上查找模式串 P 对应的节点，如果存在，则答案就是该节点的值，减去 $|P|-1$；如果不存在，则答案不存在.

预处理时间复杂度为 $O(|T|)$. 单次查询的时间复杂度为 $O(|P|)$.

解法二

我们构造一个后缀自动机. 我们对 SAM 中的所有状态预处理位置 firstpos. 即, 对每个状态 v 我们想要找到第一次出现这个状态的末端的位置 $\text{firstpos}[v]$. 换句话说，我们希望先找到每个集合 endpos 中的最小的元素（显然我们不能显式地维护所有 endpos 集合）.

为了维护 firstpos 这些位置，我们对函数 `sam_extend()` 进行扩展. 当我们创建新状态 cur 时，我们令：

$$\text{firstpos}(cur) = \text{len}(cur) - 1.$$

当我们将结点 q 复制到 $clone$ 时，我们令：

$$\text{firstpos}(clone) = \text{firstpos}(q).$$

（因为值的唯一的其它选项 $\text{firstpos}(cur)$ 显然太大了）.

那么查询的答案就是 $\text{firstpos}(t) - |P| + 1$，其中 t 为对应字符串 P 的状态. 单次查询只需要 $O(|P|)$ 的时间.

所有出现的位置

问题

问题同上，这一次需要查询文本串 T 中模式串 P 出现的所有位置.

解法一

找到模式串 P 对应的节点后，利用后缀链接树的信息，遍历子树，一旦发现「前缀节点」就输出.

单次查询复杂度为 $O(|P|) + O(answer(P))$，其中，$answer(P)$ 为本次询问的答案数目. 仿照状态数为线性的证明→8@p889 可以说明，后缀链接树的子树大小不会超过该节点的 endpos 集合的大小的二倍，因此遍历子树的复杂度是 $O(answer(P))$ 的.

解法二

我们还是对文本串 T 构造后缀自动机. 与上一个问题相似，我们为所有状态计算位置 firstpos.

如果 t 为对应于模式串 P 的状态，显然 $\text{firstpos}(t)$ 为答案之一. 我们已经找到了自动机中对应于 P 的状态. 还需要找到其它哪些位置？正是那些对应于以 P 为后缀的字符串的状态. 换句话说，我们要找到所有可以通过后缀链接到达状态 t 的状态.

因此为了解决这个问题，我们需要为每一个状态保存一个指向它的后缀链接列表. 查询的答案就包含了对于每个我们能从状态 t 只使用反向的后缀链接进行 DFS 或 BFS 找到的所有状态的 firstpos 值.

预处理的复杂度为 $O(|\ T\ |)$，单次查询的复杂度为 $O(|\ P\ | + answer(P))$.

我们不会重复访问一个状态（因为每个状态仅有一条后缀链接，所以不存在两条不同的路径指向同一状态）.

我们只需要考虑两个可能有相同 firstpos 值的不同状态. 这种情形只在一个状态是由另一个状态复制而来时发生. 然而，这并不会对复杂度分析造成影响. 仿照 状态数为线性的证明→8@p889，所有这种后缀为 P 的状态数目不会超过 2 $answer(P)$.

此外，我们可以通过不考虑复制而来的节点的 firstpos 值来去除重复的位置. 事实上，t 的子树中每个出现位置恰好对应一个非复制而来的状态. 因此，如果我们给每个状态记录标记 `is_clone` 来代表这个状态是不是被复制出来的，我们就可以简单地忽略掉被复制的状态，只输出其它所有状态的 firstpos 的值.

以下是大致的实现：

```
struct state {
  bool is_clone;
  int first_pos;
  std::vector<int> inv_link;
  // some other variables
};

// 在构造 SAM 后
for (int v = 1; v < sz; v++) st[st[v].link].inv_link.push_back(v);

// 输出所有出现位置
void output_all_occurrences(int v, int P_length) {
  if (!st[v].is_clone) cout << st[v].first_pos - P_length + 1 << endl;
  for (int u : st[v].inv_link) output_all_occurrences(u, P_length);
}
```

最短的没有出现的字符串

问题

给定一个字符串 S 和一个特定的字符集，我们要找一个长度最短的没有在 S 中出现过的字符串.

解法

我们在字符串 S 的后缀自动机上做动态规划.

假定我们已经处理完了子串的一部分，当前在状态 v，想找到为了遇到不存在的转移需要添加的最小字符数量，将节点 v 处的这个数量记作 d_v.

计算 d_v 非常简单. 如果字符集中至少有一个字符在 v 处没有转移，则 $d_v = 1$. 否则添加一个字符是不够的，我们需要求出所有转移中的最小值：

$$d_v = 1 + \min_{w:(v,w,c)\in \text{SAM}} d_w$$

问题的答案就是 d_{t_0}，字符串可以通过计算过的数组 d 逆推回去.

两个字符串的最长公共子串

问题

给定两个字符串 S 和 T，求出最长公共子串，公共子串定义为在 S 和 T 中都作为子串出现过的字符串 X.

解法

我们对字符串 S 构造后缀自动机.

我们现在处理字符串 T，对于每一个前缀，都在 S 中寻找这个前缀的最长后缀. 换句话说，对于每个字符串 T 中的位置，我们想要找到这个位置结束的 S 和 T 的最长公共子串的长度.

为了达到这一目的，我们使用两个变量，**当前状态** v 和 **当前长度** l. 这两个变量描述当前匹配的部分：它的长度和它对应的状态.

一开始 $v = t_0$ 且 $l = 0$，即，匹配为空串.

现在我们来描述如何添加一个字符 T_i 并为其重新计算答案：

- 如果存在一个从 v 到字符 T_i 的转移，我们只需要转移并让 l 自增一.
- 如果不存在这样的转移，我们需要缩短当前匹配的部分，这意味着我们需要按照后缀链接进行转移：

  $v = \text{link}(v)$

  与此同时，需要缩短当前长度. 显然我们需要将 l 赋值为 $\text{len}(v)$，因为经过这个后缀链接后我们到达的状态所对应的最长字符串是一个子串.
- 如果仍然没有使用这一字符的转移，我们继续重复经过后缀链接并减小 l，直到我们找到一个转移，或到达初始状态 t_0 且它也没有这一转移（这意味着字符 T_i 根本没有在 S 中出现过，此时 $v = l = 0$）.

显然问题的答案就是所有 l 的最大值.

这一部分的时间复杂度为 $O(|T|)$，因为每次移动我们要么可以使 l 增加一，要么可以在后缀链接间移动几次，每次都减小 l 的值.

代码实现：

```
string lcs(const string &S, const string &T) {
  sam_init();
  for (int i = 0; i < S.size(); i++) sam_extend(S[i]);

  int v = 0, l = 0, best = 0, bestpos = 0;
  for (int i = 0; i < T.size(); i++) {
    while (v && !st[v].next.count(T[i])) {
      v = st[v].link;
      l = st[v].len;
    }
    if (st[v].next.count(T[i])) {
      v = st[v].next[T[i]];
      l++;
    }
    if (l > best) {
      best = l;
      bestpos = i;
    }
```

```
19   }
20   if (best == 0) return "";
21   return T.substr(bestpos - best + 1, best);
22 }
```

例题：SPOJ Longest Common Substring[5]

多个字符串间的最长公共子串

问题

给定 k 个字符串 S_i．我们需要找到它们的最长公共子串，即作为子串出现在每个字符串中的字符串 X．

解法一

我们将所有的字符串连接成一个较长的字符串 T，以特殊字符 D_i 分开每个字符串（一个字符对应一个字符串）：

$$T = S_1 + D_1 + S_2 + D_2 + \cdots + S_k + D_k.$$

然后对字符串 T 构造后缀自动机．

现在我们需要在自动机中找到存在于所有字符串 S_i 中的一个字符串，为此可以利用添加的特殊字符．如果 S_j 包含了一个子串 X，则从子串 X 对应的节点 t 出发，必然存在一条到达 D_j 但是不经过任何其它特殊字符 $D_1, \cdots, D_{j-1}, D_{j+1}, \cdots, D_k$ 的路径．对于公共子串 X，应当对每个特殊字符 D_j 都存在这样的路径．

因此我们需要计算可达性，即对于自动机中的每个状态和每个字符 D_i，是否存在这样的一条路径．这可以容易地通过 DFS 或 BFS 及动态规划计算．这之后，问题的答案就是所有能够达到所有特殊字符的状态 v 对应的最长子串 $\text{longest}(v)$ 中最长的那个．

解法二

不妨设 **最短** 的字符串为 S_1，对它构造 SAM．利用解决两个字符串最长公共子串的算法，计算剩余的每个字符串与 S_1 的最长公共子串长度．在匹配过程中，每添加一个要匹配的字符串 S_j 中的字符，就相应地在 SAM 上移动，因此，可以直接记录 **在匹配过程中** SAM 每个状态能够匹配上的 S_j 的最长子串的长度．

因为匹配过程中，每次匹配到 SAM 的一个状态时，必然同时匹配到了它在后缀链接树上的所有祖先节点，但是祖先节点的匹配长度的信息并没有更新．所以，在完成对字符串 S_j 的匹配后，需要自下而上地沿着后缀链接更新，将子节点匹配到的最长子串的信息更新到父节点．此时，需要注意父节点记录的最长匹配长度不能超过它自身的 len 值．这样，就得到了 S_1 的 SAM 上每个状态 **实际能够匹配到** 的 S_j 的最长子串长度．

最后，只需要对每个 $S_2, \cdots, S_k$ 都匹配一遍，再对 SAM 上每个状态记录的实际匹配到的长度取最小值，就得到 SAM 上每个状态实际能够匹配到的 $S_2, \cdots, S_k$ 的最长公共子串的长度．然后，遍历 SAM 所有状态，取最大值就是这 k 个串的最长公共子串长度．

算法时间复杂度是 $O(\sum_i |S_i|)$ 的．字符串 S_1 的 SAM 虽然遍历了 k 遍，但是因为 $|S_1|$ 是最小的，所以 $k \mid S_1| \le \sum_i |S_i|$，复杂度的主要项依然是匹配过程遍历所有字符串．

例题：SPOJ Longest Common Substring II[6]

习题

- 【模板】后缀自动机[7]
- SDOI2016 生成魔咒[8]
- SPOJ - SUBLEX[9]
- TJOI2015 弦论[10]
- SPOJ Longest Common Substring[11]
- SPOJ Longest Common Substring II[12]
- Codeforces 1037H Security[13]
- Codeforces 666E Forensic Examination[14]
- HDU4416 Good Article Good sentence[15]
- HDU4436 str2int[16]
- HDU6583 Typewriter[17]
- Codeforces 235C Cyclical Quest[18]
- CTSC2012 熟悉的文章[19]
- NOI2018 你的名字[20]

相关资料

我们先给出与 SAM 有关的最初的一些文献：

- A. Blumer, J. Blumer, A. Ehrenfeucht, D. Haussler, R. McConnell. Linear Size Finite Automata for the Set of All Subwords of a Word. An Outline of Results. [1983]
- A. Blumer, J. Blumer, A. Ehrenfeucht, D. Haussler. The Smallest Automaton Recognizing the Subwords of a Text. [1984]
- Maxime Crochemore. Optimal Factor Transducers. [1985]
- Maxime Crochemore. Transducers and Repetitions. [1986]
- A. Nerode. Linear automaton transformations. [1958]

另外，在更新的一些资源以及很多关于字符串算法的书中，都能找到这个主题：

- Maxime Crochemore, Wojciech Rytter. Jewels of Stringology. [2002]
- Bill Smyth. Computing Patterns in Strings. [2003]
- Bill Smyth. Методы и алгоритмы вычислений на строках（上一本书的俄译本）. [2006]

另外，还有一些资料：

- 《后缀自动机》，陈立杰.
- 《后缀自动机在字典树上的拓展》，刘研绎.
- 《后缀自动机及其应用》，张天扬.
- https://www.cnblogs.com/zinthos/p/3899679.html[21]
- https://codeforces.com/blog/entry/20861[22]
- https://zhuanlan.zhihu.com/p/25948077[23]

本页面主要译自博文 Суффиксный автомат[24] 与其英文翻译版 Suffix Automaton[25]. 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.

[1] 【模板】后缀自动机 

[2] SDOI2016 生成魔咒 

[3] SPOJ - SUBLEX 

[4] TJOI2015 弦论 

[5] SPOJ Longest Common Substring 

[6] SPOJ Longest Common Substring II 

[7] 【模板】后缀自动机 

[8] SDOI2016 生成魔咒 

[9] SPOJ - SUBLEX 

[10] TJOI2015 弦论 
[11] SPOJ Longest Common Substring 
[12] SPOJ Longest Common Substring II 
[13] Codeforces 1037H Security 
[14] Codeforces 666E Forensic Examination 
[15] HDU4416 Good Article Good sentence 
[16] HDU4436 str2int 
[17] HDU6583 Typewriter 
[18] Codeforces 235C Cyclical Quest 
[19] CTSC2012 熟悉的文章 
[20] NOI2018 你的名字 
[21] https://www.cnblogs.com/zinthos/p/3899679.html 
[22] https://codeforces.com/blog/entry/20861 
[23] https://zhuanlan.zhihu.com/p/25948077 
[24] Суффиксный автомат 
[25] Suffix Automaton 

8.13 后缀平衡树

定义

后缀之间的大小由字典序定义，后缀平衡树就是一个维护这些后缀顺序的平衡树，即字符串 T 的后缀平衡树是 T 所有后缀的有序集合．后缀平衡树上的一个节点相当于原字符串的一个后缀．

特别地，后缀平衡树的中序遍历即为后缀数组．

构造过程

对长度为 n 的字符串 T 建立其后缀平衡树，考虑逆序将其后缀加入后缀平衡树．

记后缀平衡树维护的集合为 X，当前添加的后缀为 S，则添加下一个后缀就是向 X 中加入 $\mathtt{c}S$（亦可理解为后缀平衡树维护的字符串为 S，下一步往 S 前加入一个字符 $\mathtt{c}$）．这一操作其实就是向平衡树中插入节点．

这里使用期望树高为 $O(\log n)$ 的平衡树，例如替罪羊树或 Treap 等．

做法 1

插入时，暴力比较两个后缀之间的大小关系，从而判断之后是往哪一个子树添加．这样子，单次插入至多比较 $O(\log n)$ 次，单次比较的时间复杂度至多为 $O(n)$，一共 $O(n \log n)$．

一共会插入 n 次，所以该做法的时间复杂度存在上界 $O(n^2 \log n)$．

做法 2

注意到 $\mathtt{c}S$ 与 S 的区别仅在于 $\mathtt{c}$，且 S 已经属于 X 了，可以利用这一点来优化插入操作．

假设当前要比较 $\mathtt{c}S$ 与 A 两个字符串的大小，且 $A, S \in X$．每次比较时，首先比较两串的首字符．若首字符不等，则两串的大小关系就已经确定了；若首字符相等，那么就只需要判断去除首字符后两字符串的大小关系．而两串去除首字符后都已经属于 X 了，这时候可以借助平衡树 $O(\log n)$ 求排名的操作来完成后续的比较．这样，单次插入的操作至多 $O(\log^2 n)$．

一共会插入 n 次，所以该做法的时间复杂度存在上界 $O(n \log^2 n)$．

做法 3

根据做法 2，如果能够 $O(1)$ 判断平衡树中两个节点之间的大小关系，那么就可以在 $O(n \log n)$ 的时间内完成后缀平衡树的构造．

记 val_i 表示节点 i 的值．如果在建平衡树时，每个节点多维护一个标记 tag_i，使得若 $tag_i > tag_j \iff val_i > val_j$，那么就可以根据 tag_i 的大小 $O(1)$ 判断平衡树中两个节点的大小．

不妨令平衡树中每个节点对应一个实数区间，令根节点对应 $(0,1)$．对于节点 i，记其对应的实数区间为 (l,r)，则 $tag_i = \frac{l+r}{2}$，其左子树对应实数区间 (l, tag_i)，其右子树对应实数区间 (tag_i, r)．易证 tag_i 满足上述要求．

由于使用了期望树高为 $O(\log n)$ 的平衡树，所以精度是有一定保证的．实际实现时也可以用一个较大的区间来做，例如让根对应 $(0, 10^{18})$．

做法 4

其实可以先构建出后缀数组，然后再根据后缀数组构建后缀平衡树．这样做的复杂度瓶颈在于后缀数组的构建复杂度或者所用平衡树一次性插入 n 个元素的复杂度．

删除操作

假设当前添加的后缀为 cS，上一个添加的后缀为 S．后缀平衡树还支持删除后缀 cS 的操作（亦可理解为后缀平衡树维护的字符串为 cS，将开头的 c 删除）．

类似于插入操作，借助平衡树的删除节点操作可以完成删除 cS 的操作．

后缀平衡树的优点

- 后缀平衡树的思路比较清晰，相比后缀自动机等后缀结构更好理解，会写平衡树就能写．
- 后缀平衡树的复杂度不依赖于字符集的大小
- 后缀平衡树支持在字符串开头删除一个字符
- 如果使用支持可持久化的平衡树，那么后缀平衡树也能可持久化

例题

P3809【模板】后缀排序[1]

后缀数组的模板题，建出后缀平衡树之后，通过中序遍历得到后缀数组．

SGT 版本的参考代码

```cpp
#include <algorithm>
#include <iostream>
#include <string>
using namespace std;

constexpr int N = 1e6 + 5;
constexpr double INF = 1e18;

int n, m, sa[N];
string t;

// SuffixBST(SGT Ver)

// 顺序加入，查询时将询问串翻转
// 以i开始的后缀，对应节点的编号为i
constexpr double alpha = 0.75;
int root;
int sz[N], L[N], R[N];
double tag[N];
```

```
int buffer_size, buffer[N];

bool cmp(int x, int y) {
  if (t[x] != t[y]) return t[x] < t[y];
  return tag[x + 1] < tag[y + 1];
}

void init() { root = 0; }

void new_node(int& rt, int p, double lv, double rv) {
  rt = p;
  sz[rt] = 1;
  tag[rt] = (lv + rv) / 2;
  L[rt] = R[rt] = 0;
}

void push_up(int x) {
  if (!x) return;
  sz[x] = sz[L[x]] + 1 + sz[R[x]];
}

bool balance(int rt) { return alpha * sz[rt] > max(sz[L[rt]], sz[R[rt]]); }

void flatten(int rt) {
  if (!rt) return;
  flatten(L[rt]);
  buffer[++buffer_size] = rt;
  flatten(R[rt]);
}

void build(int& rt, int l, int r, double lv, double rv) {
  if (l > r) {
    rt = 0;
    return;
  }
  int mid = (l + r) >> 1;
  double mv = (lv + rv) / 2;

  rt = buffer[mid];
  tag[rt] = mv;
  build(L[rt], l, mid - 1, lv, mv);
  build(R[rt], mid + 1, r, mv, rv);
  push_up(rt);
}

void rebuild(int& rt, double lv, double rv) {
  buffer_size = 0;
  flatten(rt);
  build(rt, 1, buffer_size, lv, rv);
}

void insert(int& rt, int p, double lv, double rv) {
```

```
  if (!rt) {
    new_node(rt, p, lv, rv);
    return;
  }

  if (cmp(p, rt))
    insert(L[rt], p, lv, tag[rt]);
  else
    insert(R[rt], p, tag[rt], rv);

  push_up(rt);
  if (!balance(rt)) rebuild(rt, lv, rv);
}

void inorder(int rt) {
  if (!rt) return;
  inorder(L[rt]);
  sa[++m] = rt;
  inorder(R[rt]);
}

void solve(int Case) {
  cin >> t;
  n = t.size();
  t = " " + t;

  init();
  for (int i = n; i >= 1; --i) {
    insert(root, i, 0, INF);
  }

  // 后缀平衡树的中序遍历即为后缀数组
  m = 0;
  inorder(root);

  for (int i = 1; i <= n; ++i) cout << sa[i] << ' ';
  cout << '\n';
}

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  solve(1);
  return 0;
}

```

P6164【模板】后缀平衡树[2]

题意

给定初始字符串 s 和 q 个操作：

1. 在当前字符串的后面插入若干个字符.

2. 在当前字符串的后面删除若干个字符.
3. 询问字符串 t 作为连续子串在当前字符串中出现了几次?

题目 **强制在线**，字符串变化长度以及初始长度 $\leq 8 \times 10^5$，$q \leq 10^5$，询问的总长度 $\leq 3 \times 10^6$.

对于操作 1 和操作 2，由于后缀平衡树维护头插和头删操作比较方便，所以想到把尾插和尾删操作搞成头插和头删. 这里如果维护 s 的反串的后缀平衡树，而非 s 的后缀平衡树，就可以完成上述转换. 平衡树的添加和删除都是 $O(\log n)$ 的，所以添加或者删除一个字符的时间复杂度为 $O(\log n)$. 记添加和删除的总字符数为 N，那么这一部分总的时间复杂度为 $O(N \log n)$.

对于操作 3，t 的出现次数等于以 t 为前缀的后缀数量，而以 t 为前缀的后缀数量等于其后继的排名减去其前驱的排名. 在 t 后面加入一个极大的字符，就可以构造出 t 的一个后继. 将 t 的最后一个字符减小 1，就可以构造出 t 的一个前驱.

现在要查询某一个串 t 在后缀平衡树中排名，由于不能保证 t 在后缀平衡树中出现过，所以每次只能暴力比较字符串大小. 单次比较的时间复杂度为 $O(|t|)$，每次查询至多比较 $O(\log n)$ 次，所以单次查询的复杂度为 $O(|t| \log n)$. 记所有询问串的长度和为 L，那么这一部分总的时间复杂度为 $O(L \log n)$.

SGT 版本的参考代码

```cpp
#include <algorithm>
#include <iostream>
#include <string>
using namespace std;

constexpr int N = 8e5 + 5;
constexpr double INF = 1e18;

void decode(string& s, int len, int mask) {
  for (int i = 0; i < len; ++i) {
    mask = (mask * 131 + i) % len;
    swap(s[i], s[mask]);
  }
}

int q, n, na;
string a;
char t[N];

// SuffixBST(SGT Ver)

// 顺序加入，查询时将询问串翻转
// 以i结束的前缀，对应节点的编号为i
// 注意：不能写懒惰删除，否则可能会破坏树的结构
constexpr double alpha = 0.75;
int root;
int sz[N], L[N], R[N];
double tag[N];
int buffer_size, buffer[N];

bool cmp(int x, int y) {
  if (t[x] != t[y]) return t[x] < t[y];
  return tag[x - 1] < tag[y - 1];
```

```
}

void init() { root = 0; }

void new_node(int& rt, int p, double lv, double rv) {
  rt = p;
  sz[rt] = 1;
  tag[rt] = (lv + rv) / 2;
  L[rt] = R[rt] = 0;
}

void push_up(int x) {
  if (!x) return;
  sz[x] = sz[L[x]] + 1 + sz[R[x]];
}

bool balance(int rt) { return alpha * sz[rt] > max(sz[L[rt]], sz[R[rt]]); }

void flatten(int rt) {
  if (!rt) return;
  flatten(L[rt]);
  buffer[++buffer_size] = rt;
  flatten(R[rt]);
}

void build(int& rt, int l, int r, double lv, double rv) {
  if (l > r) {
    rt = 0;
    return;
  }
  int mid = (l + r) >> 1;
  double mv = (lv + rv) / 2;

  rt = buffer[mid];
  tag[rt] = mv;
  build(L[rt], l, mid - 1, lv, mv);
  build(R[rt], mid + 1, r, mv, rv);
  push_up(rt);
}

void rebuild(int& rt, double lv, double rv) {
  buffer_size = 0;
  flatten(rt);
  build(rt, 1, buffer_size, lv, rv);
}

void insert(int& rt, int p, double lv, double rv) {
  if (!rt) {
    new_node(rt, p, lv, rv);
    return;
  }

```

```
  if (cmp(p, rt))
    insert(L[rt], p, lv, tag[rt]);
  else
    insert(R[rt], p, tag[rt], rv);

  push_up(rt);
  if (!balance(rt)) rebuild(rt, lv, rv);
}

void remove(int& rt, int p, double lv, double rv) {
  if (!rt) return;

  if (rt == p) {
    if (!L[rt] || !R[rt]) {
      rt = (L[rt] | R[rt]);
      rebuild(rt, lv, rv);
    } else {
      // 找到rt的前驱来替换rt
      int nrt = L[rt];
      while (R[nrt]) {
        nrt = R[nrt];
      }
      remove(L[rt], nrt, lv, tag[rt]);
      L[nrt] = L[rt];
      R[nrt] = R[rt];
      rt = nrt;
      tag[rt] = (lv + rv) / 2;
    }
  } else {
    double mv = (lv + rv) / 2;
    if (cmp(p, rt))
      remove(L[rt], p, lv, mv);
    else
      remove(R[rt], p, mv, rv);
  }

  push_up(rt);
  if (!balance(rt)) rebuild(rt, lv, rv);
}

bool cmp1(const string& s, int len, int p) {
  for (int i = 1; i <= len; ++i, --p) {
    if (s[i] < t[p]) return true;
    if (s[i] > t[p]) return false;
  }
  return false;
}

int query(int rt, const string& s, int len) {
  if (!rt) return 0;
  if (cmp1(s, len, rt))
    return query(L[rt], s, len);
```

```
138   else
139     return sz[L[rt]] + 1 + query(R[rt], s, len);
140 }
141
142 void solve() {
143   cin.tie(nullptr)->sync_with_stdio(false);
144   n = 0;
145   cin >> q;
146   init();
147
148   cin >> a;
149   na = a.size();
150   a = " " + a;
151   for (int i = 1; i <= na; ++i) {
152     t[++n] = a[i];
153     insert(root, n, 0, INF);
154   }
155
156   int mask = 0;
157   char op[10];
158   for (int i = 1; i <= q; ++i) {
159     cin >> op;
160
161     // 三种情况分别处理
162
163     if (op[0] == 'A') {  // ADD
164       cin >> a;
165       na = a.size();
166       decode(a, na, mask);
167       a = " " + a;
168
169       for (int i = 1; i <= na; ++i) {
170         t[++n] = a[i];
171         insert(root, n, 0, INF);
172       }
173     } else if (op[0] == 'D') {  // DEL
174       int x;
175       cin >> x;
176       while (x) {
177         remove(root, n, 0, INF);
178         --n;
179         --x;
180       }
181     } else if (op[0] == 'Q') {  // QUERY
182       cin >> a;
183       na = a.size();
184       decode(a, na, mask);
185       a = " " + a;
186
187       reverse(a.begin() + 1, a.begin() + 1 + na);
188
189       a.push_back('Z' + 1);
```

```
190       int ans = query(root, a, na + 1);
191
192       --a[na];
193       ans -= query(root, a, na + 1);
194
195       cout << ans << '\n';
196       mask ^= ans;
197     }
198   }
199 }
200
201 int main() {
202   solve();
203   return 0;
204 }
205
```

参考资料

- 陈立杰 -《重量平衡树和后缀平衡树在信息学奥赛中的应用》

外部参考资料

[1] P3809【模板】后缀排序 [2] P6164【模板】后缀平衡树

8.14 广义后缀自动机

前置知识

广义后缀自动机基于下面的知识点

- 字典树（Trie 树）→8.6@p903
- 后缀自动机→8.12@p999

请务必对上述两个知识点非常熟悉之后，再来阅读本文，特别是对于 **后缀自动机** 中的 **后缀链接** 能够有一定的理解

引入

起源

广义后缀自动机是由刘研绎在其 2015 国家队论文《后缀自动机在字典树上的拓展》上提出的一种结构，即将后缀自动机直接建立在字典树上.

> 大部分可以用后缀自动机处理的字符串的问题均可扩展到 Trie 树上. ——刘研绎

约定

参考 字符串约定→8.2@p890

字符串个数为 k 个，即 $S_1, S_2, S_3 ... S_k$

约定字典树和广义后缀自动机的根节点为 0 号节点

概述

后缀自动机 (suffix automaton, SAM) 是用于处理单个字符串的子串问题的强力工具.

而广义后缀自动机 (General Suffix Automaton) 则是将后缀自动机整合到字典树中来解决对于多个字符串的子串问题

常见的伪广义后缀自动机

1. 通过用特殊符号将多个串直接连接后，再建立 SAM
2. 对每个串，重复在同一个 SAM 上进行建立，每次建立前，将 `last` 指针置零

方法 1 和方法 2 的实现方式简单，而且在面对题目时通常可以达到和广义后缀自动机一样的正确性. 所以在网络上很多人会选择此类写法，例如在后缀自动机一文中最后一个应用，便使用了方法 1 （原文链接）→8.12@p999

但是无论方法 1 还是方法 2，其时间复杂度较为危险

构造广义后缀自动机

根据原论文的描述，应当在多个字符串上先建立字典树，然后在字典树的基础上建立广义后缀自动机.

字典树的使用

首先应对多个串创建一棵字典树，这不是什么难事，如果你已经掌握了前置知识的前提下，可以很快的建立完毕. 这里为了统一上下文的代码，给出一个可能的字典树代码.

实现

```cpp
constexpr int MAXN = 2000000;
constexpr int CHAR_NUM = 30;

struct Trie {
  int next[MAXN][CHAR_NUM];  // 转移
  int tot;                   // 节点总数: [0, tot)

  void init() { tot = 1; }

  int insertTrie(int cur, int c) {
    if (next[cur][c]) return next[cur][c];
    return next[cur][c] = tot++;
  }

  void insert(const string &s) {
    int root = 0;
    for (auto ch : s) root = insertTrie(root, ch - 'a');
  }
};
```

这里我们得到了一棵依赖于 `next` 数组建立的一棵字典树.

后缀自动机的建立

如果我们把这样一棵树直接认为是一个后缀自动机，则我们可以得到如下结论

- 对于节点 `i`，其 `len[i]` 和它在字典树中的深度相同
- 如果我们对字典树进行拓扑排序，我们可以得到一串根据 `len` 不递减的序列. BFS 的结果相同

而后缀自动机在建立的过程中，可以视为不断的插入 `len` 严格递增的值，且差值为 1．所以我们可以将对字典树进行拓扑排序后的结果做为一个队列，然后按照这个队列的顺序不断地插入到后缀自动机中．

由于在普通后缀自动机上，其前一个节点的 `len` 值为固定值，即为 `last` 节点的 `len`．但是在广义后缀自动机中，插入的队列是一个不严格递增的数列．所以对于每一个值，对于它的 `last` 应该是已知而且固定的，在字典树上，即为其父亲节点．

由于在字典树中，已经建立了一个近似的后缀自动机，所以只需要对整个字典树的结构进行一定的处理即可转化为广义后缀自动机．我们可以按照前面提出的队列顺序来对整个字典树上的每一个节点进行更新操作．最终我们可以得到广义后缀自动机．

对于每个点的更新操作，我们可以稍微修改一下 SAM 中的插入操作来得到．

对于整个插入的过程，需要注意的是，由于插入是按照 `len` 不递减的顺序插入，在进行 `clone` 后的数据复制过程中，不可以复制其 `len` 小于当前 `len` 的数据．

过程

根据上述的逻辑，可以将整个构建过程描述为如下操作

1. 将所有字符串插入到字典树中
2. 从字典树的根节点开始进行 BFS，记录下顺序以及每个节点的父亲节点
3. 将得到的 BFS 序列按照顺序，对每个节点在原字典树上进行构建，注意不能将 `len` 小于当前 `len` 的数据进行操作

对操作次数为线性的证明

由于仅处理 BFS 得到的序列，可以保证字典树上所有节点仅经过一次．

对于最坏情况，考虑字典树本身节点个数最多的情况，即任意两个字符串没有相同的前缀，则节点个数为 $\sum_{i=1}^{k}|S_i|$，即所有的字符串长度之和．

而在后缀自动机的更新操作的复杂度已经在 后缀自动机→8.12@p999 中证明

所以可以证明其最坏复杂度为线性

而通常伪广义后缀自动机的平均复杂度等同于广义后缀自动机的最差复杂度，面对大量的字符串时，伪广义后缀自动机的效率远不如标准的广义后缀自动机

实现

对插入函数进行少量必要的修改即可得到所需要的函数

参考代码

```
struct GSA {
  int len[MAXN];              // 节点长度
  int link[MAXN];             // 后缀链接，link
  int next[MAXN][CHAR_NUM];   // 转移
  int tot;                    // 节点总数：[0, tot)

  int insertSAM(int last, int c) {
    int cur = next[last][c];
    len[cur] = len[last] + 1;
    int p = link[last];
    while (p != -1) {
      if (!next[p][c])
        next[p][c] = cur;
```

```
14        else
15          break;
16        p = link[p];
17      }
18      if (p == -1) {
19        link[cur] = 0;
20        return cur;
21      }
22      int q = next[p][c];
23      if (len[p] + 1 == len[q]) {
24        link[cur] = q;
25        return cur;
26      }
27      int clone = tot++;
28      for (int i = 0; i < CHAR_NUM; ++i)
29        next[clone][i] = len[next[q][i]] != 0 ? next[q][i] : 0;
30      len[clone] = len[p] + 1;
31      while (p != -1 && next[p][c] == q) {
32        next[p][c] = clone;
33        p = link[p];
34      }
35      link[clone] = link[q];
36      link[cur] = clone;
37      link[q] = clone;
38      return cur;
39    }
40
41    void build() {
42      queue<pair<int, int>> q;
43      for (int i = 0; i < CHAR_NUM; ++i)
44        if (next[0][i]) q.push({i, 0});
45      while (!q.empty()) {
46        auto item = q.front();
47        q.pop();
48        auto last = insertSAM(item.second, item.first);
49        for (int i = 0; i < CHAR_NUM; ++i)
50          if (next[last][i]) q.push({i, last});
51      }
52    }
53  }
```

- 由于整个 BFS 的过程得到的顺序，其父节点始终在变化，所以并不需要保存 `last` 指针.
- 插入操作中，`int cur = next[last][c];` 与正常后缀自动机的 `int cur = tot++;` 有差异，因为我们插入的节点已经在树型结构中完成了，所以只需要直接获取即可
- 在 `clone` 后的数据拷贝中，有这样的判断 `next[clone][i] = len[next[q][i]] != 0 ? next[q][i] : 0;` 这与正常的后缀自动机的直接赋值 `next[clone][i] = next[q][i];` 有一定差异，此次是为了避免更新了 `len` 大于当前节点的值. 由于数组中 `len` 当且仅当这个值被 BFS 遍历并插入到后缀自动机后才会被赋值

性质

1. 广义后缀自动机与后缀自动机的结构一致，在后缀自动机上的性质绝大部分均可在广义后缀自动机上生效（后缀自动机的性质→8.12@p999）
2. 当广义后缀自动机建立后，通常字典树结构将会被破坏，即通常不可以用广义后缀自动机来解决字典树问题．当然也可以选择准备双倍的空间，将后缀自动机建立在另外一个空间上．

应用

所有字符中不同子串个数

可以根据后缀自动机的性质得到，以点 i 为结束节点的子串个数等于 $len[i]-len[link[i]]$

所以可以遍历所有的节点求和得到

例题：【模板】广义后缀自动机（广义 SAM）[1]

参考代码

```cpp
#include <cstring>
#include <iostream>
#include <queue>
using namespace std;
constexpr int MAXN = 2000000;  // 双倍字符串长度
constexpr int CHAR_NUM = 30;   // 字符集个数，注意修改下方的 (-'a')

struct exSAM {
  int len[MAXN];              // 节点长度
  int link[MAXN];             // 后缀链接，link
  int next[MAXN][CHAR_NUM];   // 转移
  int tot;                    // 节点总数：[0, tot)

  void init() {  // 初始化函数
    tot = 1;
    link[0] = -1;
  }

  int insertSAM(int last, int c) {  // last 为父 c 为子
    int cur = next[last][c];
    if (len[cur]) return cur;
    len[cur] = len[last] + 1;
    int p = link[last];
    while (p != -1) {
      if (!next[p][c])
        next[p][c] = cur;
      else
        break;
      p = link[p];
    }
    if (p == -1) {
      link[cur] = 0;
      return cur;
    }
    int q = next[p][c];
    if (len[p] + 1 == len[q]) {
```

```
37        link[cur] = q;
38        return cur;
39      }
40      int clone = tot++;
41      for (int i = 0; i < CHAR_NUM; ++i)
42        next[clone][i] = len[next[q][i]] != 0 ? next[q][i] : 0;
43      len[clone] = len[p] + 1;
44      while (p != -1 && next[p][c] == q) {
45        next[p][c] = clone;
46        p = link[p];
47      }
48      link[clone] = link[q];
49      link[cur] = clone;
50      link[q] = clone;
51      return cur;
52    }
53
54    int insertTrie(int cur, int c) {
55      if (next[cur][c]) return next[cur][c];  // 已有该节点 直接返回
56      return next[cur][c] = tot++;            // 无该节点 建立节点
57    }
58
59    void insert(const string &s) {
60      int root = 0;
61      for (auto ch : s) root = insertTrie(root, ch - 'a');
62    }
63
64    void insert(const char *s, int n) {
65      int root = 0;
66      for (int i = 0; i < n; ++i)
67        root =
68            insertTrie(root, s[i] - 'a');  // 一边插入一边更改所插入新节点的父节点
69    }
70
71    void build() {
72      queue<pair<int, int>> q;
73      for (int i = 0; i < CHAR_NUM; ++i)
74        if (next[0][i]) q.push({i, 0});
75      while (!q.empty()) {  // 广搜遍历
76        auto item = q.front();
77        q.pop();
78        auto last = insertSAM(item.second, item.first);
79        for (int i = 0; i < CHAR_NUM; ++i)
80          if (next[last][i]) q.push({i, last});
81      }
82    }
83  } exSam;
84
85  char s[1000100];
86
87  int main() {
88    int n;
```

```
  cin >> n;
  exSam.init();
  for (int i = 0; i < n; ++i) {
    cin >> s;
    int len = strlen(s);
    exSam.insert(s, len);
  }
  exSam.build();
  long long ans = 0;
  for (int i = 1; i < exSam.tot; ++i) {
    ans += exSam.len[i] - exSam.len[exSam.link[i]];
  }
  cout << ans << endl;
}

```

多个字符串间的最长公共子串

我们需要对每个节点建立一个长度为 k 的数组 `flag`（对于本题而言，可以仅为标记数组，若需要求出此子串的个数，则需要改成计数数组）

在字典树插入字符串时，对所有节点进行计数，保存在当前字符串所在的数组

然后按照 `len` 递减的顺序遍历，通过后缀链接将当前节点的 `flag` 与其他节点的合并

遍历所有的节点，找到一个 `len` 最大且满足对于所有的 `k`，其 `flag` 的值均为非 0 的节点，此节点的 len 即为解

例题：SPOJ Longest Common Substring II[2]

参考代码

```
#include <iostream>
#include <queue>
using namespace std;

constexpr int MAXN = 2000000;  // 双倍字符串长度
constexpr int CHAR_NUM = 30;   // 字符集个数，注意修改下方的 (-'a')
constexpr int NUM = 15;        // 字符串个数

struct exSAM {
  int len[MAXN];              // 节点长度
  int link[MAXN];             // 后缀链接，link
  int next[MAXN][CHAR_NUM];  // 转移
  int tot;                    // 节点总数：[0, tot)
  int lenSorted[MAXN];   // 按照 len 排序后的数组，仅排序 [1, tot)
                         // 部分，最终下标范围 [0, tot - 1)
  int sizeC[MAXN][NUM];  // 表示某个字符串的子串个数
  int curString;         // 字符串实际个数
  /**
   * 计数排序使用的辅助空间数组
   */
  int lc[MAXN];  // 统计个数
```

```
22
23  void init() {
24    tot = 1;
25    link[0] = -1;
26  }
27
28  int insertSAM(int last, int c) {
29    int cur = next[last][c];
30    len[cur] = len[last] + 1;
31    int p = link[last];
32    while (p != -1) {
33      if (!next[p][c])
34        next[p][c] = cur;
35      else
36        break;
37      p = link[p];
38    }
39    if (p == -1) {
40      link[cur] = 0;
41      return cur;
42    }
43    int q = next[p][c];
44    if (len[p] + 1 == len[q]) {
45      link[cur] = q;
46      return cur;
47    }
48    int clone = tot++;
49    for (int i = 0; i < CHAR_NUM; ++i)
50      next[clone][i] = len[next[q][i]] != 0 ? next[q][i] : 0;
51    len[clone] = len[p] + 1;
52    while (p != -1 && next[p][c] == q) {
53      next[p][c] = clone;
54      p = link[p];
55    }
56    link[clone] = link[q];
57    link[cur] = clone;
58    link[q] = clone;
59    return cur;
60  }
61
62  int insertTrie(int cur, int c) {
63    if (!next[cur][c]) next[cur][c] = tot++;
64    sizeC[next[cur][c]][curString]++;
65    return next[cur][c];
66  }
67
68  void insert(const string &s) {
69    int root = 0;
70    for (auto ch : s) root = insertTrie(root, ch - 'a');
71    curString++;
72  }
73
```

```
74    void insert(const char *s, int n) {
75      int root = 0;
76      for (int i = 0; i < n; ++i) root = insertTrie(root, s[i] - 'a');
77      curString++;
78    }
79
80    void build() {
81      queue<pair<int, int>> q;
82      for (int i = 0; i < CHAR_NUM; ++i)
83        if (next[0][i]) q.push({i, 0});
84      while (!q.empty()) {  // 广搜遍历
85        auto item = q.front();
86        q.pop();
87        auto last = insertSAM(item.second, item.first);
88        for (int i = 0; i < CHAR_NUM; ++i)
89          if (next[last][i]) q.push({i, last});
90      }
91    }
92
93    void sortLen() {
94      for (int i = 1; i < tot; ++i) lc[i] = 0;
95      for (int i = 1; i < tot; ++i) lc[len[i]]++;
96      for (int i = 2; i < tot; ++i) lc[i] += lc[i - 1];
97      for (int i = 1; i < tot; ++i) lenSorted[--lc[len[i]]] = i;
98    }
99
100   void getSizeLen() {
101     for (int i = tot - 2; i >= 0; --i)
102       for (int j = 0; j < curString; ++j)
103         sizeC[link[lenSorted[i]]][j] += sizeC[lenSorted[i]][j];
104   }
105 } exSam;
106
107 int main() {
108   exSam.init();  // 初始化
109   string s;
110   while (cin >> s) exSam.insert(s);
111   exSam.build();
112   exSam.sortLen();
113   exSam.getSizeLen();
114   int ans = 0;
115   for (int i = 0; i < exSam.tot; ++i) {
116     bool flag = true;
117     for (int j = 0; j < exSam.curString; ++j) {
118       if (!exSam.sizeC[i][j]) {
119         flag = false;
120         break;
121       }
122     }
123     if (flag) ans = max(ans, exSam.len[i]);
124   }
125   cout << ans << endl;
```

```
126 }
127
```

外部参考资料

[1] 【模板】广义后缀自动机（广义 SAM）

[2] SPOJ Longest Common Substring II

8.15 后缀树

后缀树是一种维护一个字符串所有后缀的数据结构.

一些记号

记构建后缀树的母串为 S，长度为 n，字符集为 Σ.

令 $S[i]$ 表示 S 中的第 i 个字符，其中 $1 \leq i \leq n$.

令 $S[l,r]$ 表示 S 中第 l 个字符至第 r 个字符组成的字符串，称为 S 的一个子串.

记 $S[i,n]$ 为 S 的以 i 开头的后缀，$S[1,i]$ 为 S 的以 i 结尾的前缀.

定义

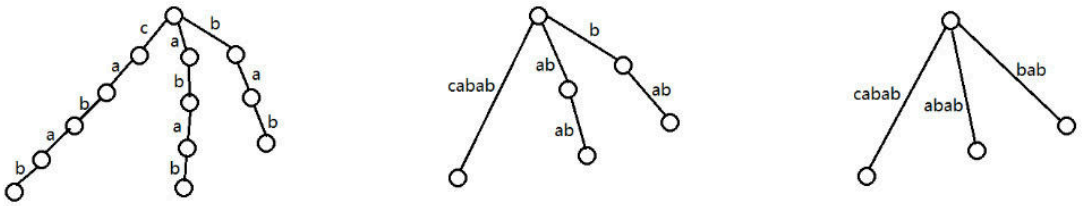
定义字符串 S 的 **后缀 trie** 为将 S 的所有后缀插入至 trie 树中得到的字典树. 在后缀 trie 中，节点 x 对应的字符串为从根节点走到 x 的路径上经过的字符拼接而成的字符串. 记后 缀 trie 中所有对应 S 的某个后缀的节点为后缀节点.

容易看出后缀 trie 的优越性质：它的非根节点恰好能接受 S 的所有本质不同非空子串. 但构建后缀 trie 的时空复杂度均为 $O(n^2)$，在很多情况下不能接受，所以我们引入后缀树的概念.

如果令后缀 trie 中所有拥有多于一个儿子的节点和后缀节点为关键点，定义只保留关键点，将非关键点形成的链压缩成一条边形成的压缩 trie 树为 **后缀树 (Suffix Tree)**. 如果仅令后缀 trie 中所有拥有多于一个儿子的节点和叶结点为关键点，定义只保留关键点形成的压缩 trie 树为 **隐式后缀树 (Implicit Suffix Tree)**. 容易看出隐式后缀树为后缀树进一步压缩后得到的结果.

在后缀树和隐式后缀树中，每条边对应一个字符串；每个非根节点 x 对应了一个字符串集合，为从根节点走到 x 的父亲节点 fa_x 经过的字符串，拼接上 fa_x 至 x 的树边对应的字符串的任意一个非空前缀，称为 str_x. 同时，在隐式后缀树中，称一个没有对应任何节点的后缀为 **隐式后缀**.

下图从左至右分别为以字符串 cabab 为母串构建的后缀 trie、后缀树和隐式后缀树.



考虑将 S 的后缀逐个插入至后缀 trie 中. 从第二次插入开始，每次最多新增一个拥有多于一个儿子的节点和一个后缀节点，所以后缀树中节点个数最多为 $2n$ 个，十分优秀.

后缀树的建立

支持前端动态添加字符的算法

反串建 SAM 建出的 parent 树就是这个串的后缀树，所以我们将反串的字符逐个加入 SAM 即可.

参考实现

```
struct SuffixAutomaton {
  int tot, lst;
  int siz[N << 1];
  int buc[N], id[N << 1];

  struct Node {
    int len, link;
    int ch[26];
  } st[N << 1];

  SuffixAutomaton() : tot(1), lst(1) {}

  void extend(int ch) {
    int cur = ++tot, p = lst;
    lst = cur;
    siz[cur] = 1, st[cur].len = st[p].len + 1;
    for (; p && !st[p].ch[ch]; p = st[p].link) st[p].ch[ch] = cur;
    if (!p)
      st[cur].link = 1;
    else {
      int q = st[p].ch[ch];
      if (st[q].len == st[p].len + 1)
        st[cur].link = q;
      else {
        int pp = ++tot;
        st[pp] = st[q];
        st[pp].len = st[p].len + 1;
        st[cur].link = st[q].link = pp;
        for (; p && st[p].ch[ch] == q; p = st[p].link) st[p].ch[ch] = pp;
      }
    }
  }
} SAM;
```

支持后端动态添加字符的算法

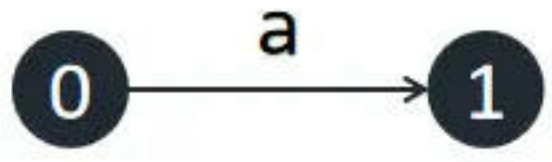
Ukkonen 算法是一种增量构造算法. 我们依次向树中插入串 S 的每一个字符，并在每一次插入之后正确地维护当前的后缀树.

朴素算法

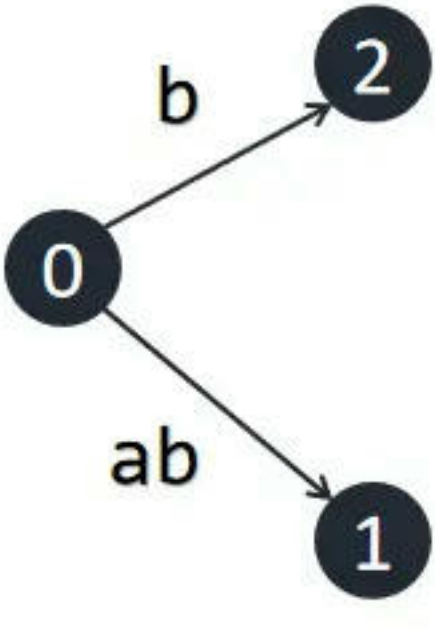
首先介绍一下一种较为暴力的构建方式，我们用字符串 abbbc 来演示一下构建的过程.

初始建立一个根节点，称为 0 号节点. 同时每条边我们维护一个区间 $[l,r]$ 表示这条边上的字符串为 $S[l,r]$. 另外，维护已经插入的字符个数 m，初始为 0.

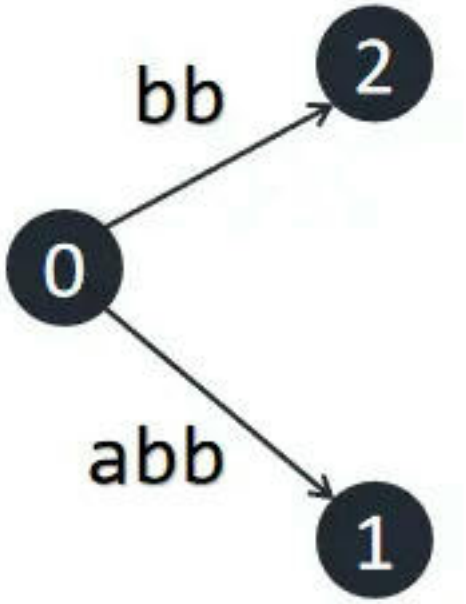
首先插入字符 a，直接从 0 号节点伸出一条边，标为 $[1,\infty]$，指向一个新建的节点. 这里的 ∞ 是一个极大值，可理解为串的结尾，这样在插入新字符时，这条边会自动的包含新的字符.



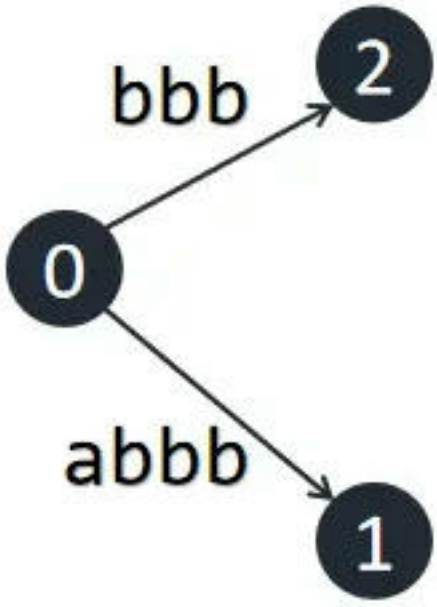
接下来我们插入字符 b，同样从 0 伸出一条边，标为 $[2,\infty]$．注意到之前延伸出的边 $[1,\infty]$ 的意义自动地发生了变化，随着串结尾的改变，其表示的串从 a 变为了 ab．这样是正确的，因为之前所有后缀都已经以一个叶节点的形式出现在树中，只需要向所有叶节点的末端插入一个当前字符即可．



接下来，我们要再次插入一个字符 b，但是 b 是之前已经插入的字符串的一个子串，因此原树已经包含 b，此时，我们什么都不做，记录一个 k 表示 $S[k,m]$ 是当前最长的隐式后缀．



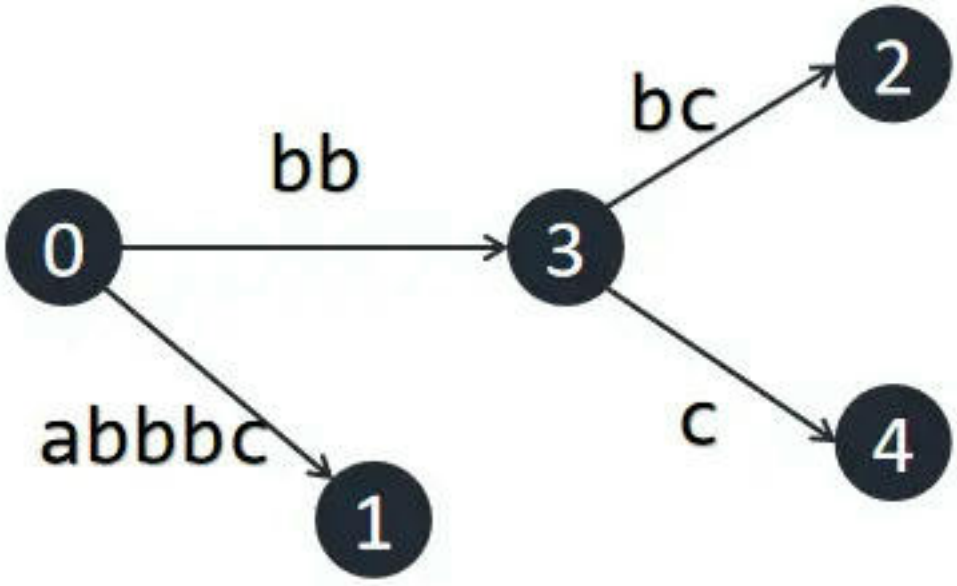
接下来我们插入另一个 b．因为前一个 b 没有插入成功，此时 $k=3$，代表要插入的后缀为 bb．我们从根开始向下寻找 bb，发现也在原树之中．同样，我们还是什么都不做．



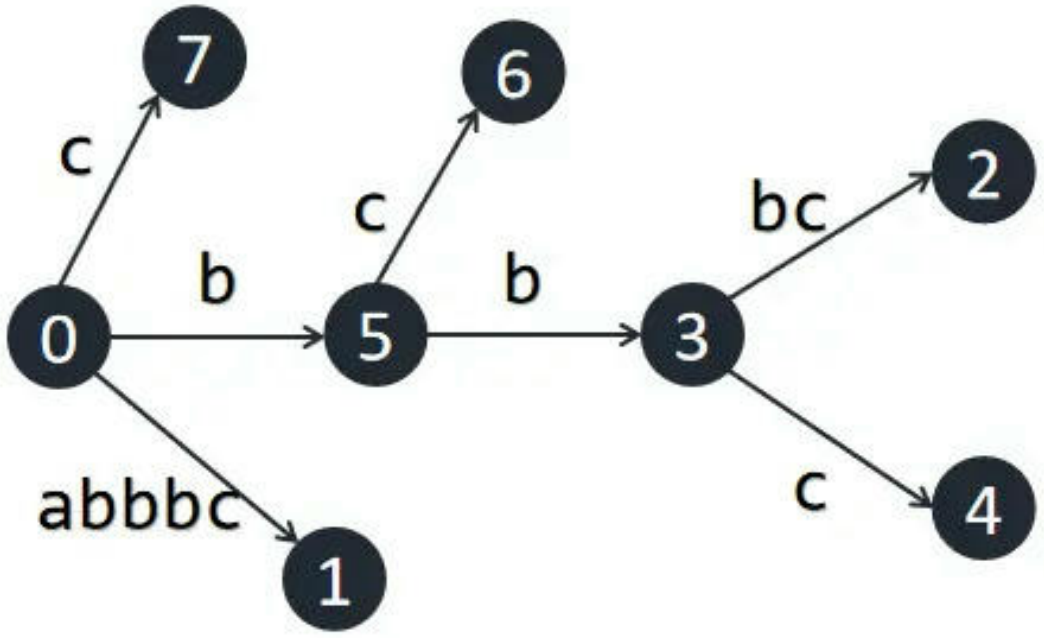
注意到我们没有管 k 之后的后缀．因为如果 $S[k,m]$ 是一个隐式后缀，那么对于 $l>k$，$S[l,m]$ 都是隐式后缀．因为由 $S[k,m]$ 为隐式后缀可知，存在字符 c 使得 $S[k,m]+c$ 为 S 的子串，所以 $S[l,m]+c$ 也为 S 的子串，由隐式后缀树的定义可知 $S[l,m]$ 也不作为叶结点出现．

接下来我们插入 c，此时 $k=3$，因此我们需要沿着根向下寻找 bbc，发现不在原树中．我们需要在 bb 处代表的节点延伸出一条为 $[5,\infty]$ 的出边．但发现这个节点其实不存在，而是包含在一条边中，因此我们

需要分裂这条边，创建一个新节点，再在创建的节点处伸展出我们要创建的出边．此时成功插入，令 $k \to k+1$，因为 $S[k,m]$ 不再是隐式后缀．



接下来，因为 k 变化了，我们重复这个过程，直到再次出现隐式后缀，或 $k > m$（在这个例子中，是后者）．



构建过程结束．

该算法每次暴力从根向下寻找并插入的复杂度最坏为 $O(n)$，所以总的复杂度为 $O(n^2)$．

后缀链接

朴素算法慢主要是因为每次 extend 都要从根找到最长隐式后缀的插入位置．所以考虑把这个位置记下来．首先，我们采用一个二元组 (now, rem) 来描述当前这个最长的被隐式包含的后缀 $S[k, m]$．沿着节点 now 的开头为 $S[m - rem + 1]$ 的出边走长度 rem 到达的位置应该唯一表示一个字符串，每次插入新的字符时，我们只需要从 now 和 rem 描述的位置查找即可．

现在，我们只需要在 $k \to k + 1$ 时更新 (now, rem)．此时如果 $now = 0$，只需要让 $rem \to rem - 1$，因为下一个要插入的后缀是刚才插入的长度 -1．否则，设 str_{now} 对应的子串为 $S[l, r]$，我们需要找到一个节点 now' 对应 $S[l + 1, r]$，令 $now \to now'$ 即可．

首先有引理：对隐式后缀树中任意非叶非根节点 x，在树中存在另一非叶节点 y，使得 str_y 是 str_x 对应的子串删去开头的字符．

证明．令 s 表示 str_x 删去开头字符形成的字符串．由隐式后缀树的定义可知，存在两个不同的字符 c_1, c_2，满足 $str_x + c1$ 与 $str_x + c_2$ 均为 S 的子串．所以，$s + c_1$ 与 $s + c_2$ 也为 S 的子串，所以 s 在后缀 trie 中也对应了一个有分叉的关键点，即在隐式后缀 trie 中存在 y 使得 $str_y = s$．证毕．

由该引理，我们定义 $\text{Link}(x) = y$，称为 x 的 **后缀链接 (Suffix Link)**．于是 $now' = \text{Link}(now)$ 一定存在．现在我们只要能求出隐式后缀树中所有非根非叶节点的 Link 即可．

Ukkonen 算法

Ukkonen 算法的整体流程如下：

为了构建隐式后缀树，我们从前往后加入 S 中的字符．假设根节点为 0，且当前已经建出 $S[1, m]$ 的隐式后缀树且维护好了后缀链接．$S[1, m]$ 的最长隐式后缀为 $S[k, m]$，在树中的位置为 (now, rem)．设 $S[m + 1] = x$，现在我们需要加入字符 x．此时，$S[1, m]$ 的每一个后缀都需要在末尾添加字符 x．由于所有显式后缀都对应树中某个叶结点，它们父边右端点为 ∞，无需维护．所以，现在我们只用考虑隐式后缀末尾添加 x 对树的形态产生的影响．首先考虑 $S[k, m]$，有两种情况：

1. (now, rem) 位置已经存在 x 的转移．此时后缀树形态不会发生变化．由于 $S[k, m + 1]$ 已经在后缀树中出现，所以对于 $l > k$，$S[l, m + 1]$ 也会在后缀树中出现，此时只需将 $rem \to rem + 1$，不需做任何修改．
2. (now, rem) 不存在 x 的转移．如果 (now, rem) 恰好为树中的节点，则此节点新增一条出边 x；否则需要对节点进行分裂，在此位置新增一个节点，并在新增节处添加出边 x．此时对于 $l > k$，我们并不知道 $S[l, m]$ 会对后缀树形态造成什么影响，所以我们还需继续考虑 $S[k + 1, m]$．考虑怎么求出 $S[k + 1, m]$ 在后缀树中的位置：如果 now 不为 0，可以利用后缀链接，令 $now = \text{Link}(now)$；否则，令 $rem \to rem - 1$．最后令 $k \to k + 1$，再次重复这个过程．

每一步都只消耗常数时间，而算法在插入全部的字符后停止，所以时间复杂度为 $O(n)$．

由于 Ukkonen 算法只能处理出 S 的隐式后缀树，而隐式后缀树在一些问题中的功能可能不如后缀树强大，所以在需要时，可以在 S 的末端添加一个从未出现过的字符，这时 S 的所有后缀可以和树的所有叶子一一对应．

参考实现

```
struct SuffixTree {
  int ch[M + 5][RNG + 1], st[M + 5], len[M + 5], link[M + 5];
  int s[N + 5];
  int now{1}, rem{0}, n{0}, tot{1};

  SuffixTree() { len[0] = inf; }

```

```
  int new_node(int s, int le) {
    ++tot;
    st[tot] = s;
    len[tot] = le;
    return tot;
  }

  void extend(int x) {
    s[++n] = x;
    ++rem;
    for (int lst{1}; rem;) {
      while (rem > len[ch[now][s[n - rem + 1]]])
        rem -= len[now = ch[now][s[n - rem + 1]]];
      int &v{ch[now][s[n - rem + 1]]}, c{s[st[v] + rem - 1]};
      if (!v || x == c) {
        lst = link[lst] = now;
        if (!v)
          v = new_node(n, inf);
        else
          break;
      } else {
        int u{new_node(st[v], rem - 1)};
        ch[u][c] = v;
        ch[u][x] = new_node(n, inf);
        st[v] += rem - 1;
        len[v] -= rem - 1;
        lst = link[lst] = v = u;
      }
      if (now == 1)
        --rem;
      else
        now = link[now];
    }
  }
} Tree;
```

作用

后缀树上每一个节点到根的路径都是 S 的一个非空子串，这在处理很多字符串问题时都很有用.

后缀树的 DFS 序就是后缀数组. 后缀树的一个子树也就对应到后缀数组上的一个区间. 后缀树上两个后缀的最长公共前缀是它们对应的叶节点的 LCA，因此，后缀数组的 height 的结论可以理解为树上若干个节点的 LCA 等于 DFS 序最小的和最大的节点的 LCA.

例题

洛谷 P3804【模板】后缀自动机（SAM）[1]

题意：

给定一个只包含小写字母的字符串 S.

请你求出 S 的所有出现次数不为 1 的子串的出现次数乘上该子串长度的最大值.

解法

建出插入一个终止符的隐式后缀树. 树上每条从根出发的路径都构成子串. 一个显示后缀的出现次数即为对应节点子树内的叶子节点个数, 隐式后缀不用考虑, 因为一个隐式后缀的出现次数等于向下走到的第一个节点对应显示后缀的出现次数, 而且一定没有该显示后缀长. 所以遍历整棵树, 求出每个节点子树内叶子个数和每个节点到根的路径长度. 如果叶子个数 > 1 则更新答案. 复杂度 $O(|S||\Sigma|)$.

参考代码

```cpp
#include <iostream>
#include <string>
using namespace std;
constexpr int N(1e6), M(2 * N), inf(1e7), RNG{26};

struct SuffixTree {
  int ch[M + 5][RNG + 1], st[M + 5], len[M + 5], link[M + 5];
  int s[N + 5];
  int now{1}, rem{0}, n{0}, tot{1};

  SuffixTree() { len[0] = inf; }

  int new_node(int s, int le) {
    ++tot;
    st[tot] = s;
    len[tot] = le;
    return tot;
  }

  void extend(int x) {
    s[++n] = x;
    ++rem;
    for (int lst{1}; rem;) {
      while (rem > len[ch[now][s[n - rem + 1]]])
        rem -= len[now = ch[now][s[n - rem + 1]]];
      int &v{ch[now][s[n - rem + 1]]}, c{s[st[v] + rem - 1]};
      if (!v || x == c) {
        lst = link[lst] = now;
        if (!v)
          v = new_node(n, inf);
        else
          break;
      } else {
        int u{new_node(st[v], rem - 1)};
        ch[u][c] = v;
        ch[u][x] = new_node(n, inf);
        st[v] += rem - 1;
        len[v] -= rem - 1;
        lst = link[lst] = v = u;
      }
      if (now == 1)
        --rem;
```

```
      else
        now = link[now];
    }
  }

  pair<long long, int> search(int u, int dep = 0) {
    if (st[u] + len[u] >= n) return {0, 1};
    dep += len[u];
    long long ans{0};
    int ys{0};
    for (int i{0}; i <= RNG; ++i)
      if (ch[u][i]) {
        auto res = search(ch[u][i], dep);
        ans = max(ans, res.first);
        ys += res.second;
      }
    if (ys > 1) ans = max(ans, 1LL * dep * ys);
    return {ans, ys};
  }
} T;

string s;

int main() {
  cin >> s;
  for (int i = 0; i < s.size(); ++i) T.extend(s[i] - 'a' + 1);
  T.extend(0);
  cout << T.search(1).first << endl;
  return 0;
}
```

CF235C Cyclical Quest[2]

题意：给定一个小写字母主串 S 和 n 个询问串，求每个询问串 x_i 的所有循环同构在主串中出现的次数总和.

解法

建立插入终止符的隐式后缀树.

枚举当前在那个循环节，记录在树上能查找到多长的前缀.

重复类似 Ukkonen 算法的过程，记录当前能匹配到的位置 (now, rem). 每次尝试插入下一个字符，如果成功则继续插入，否则跳出循环.

如果某一个次成功匹配了当前的循环节，且该循环节之前没出现过，则更新答案.

然后切换到下个循环节的时候，我们要删去当前匹配的子串开头的字符：这正好就相当于令 $now \to \mathrm{Link}(now)$. 当然，如果 $now = 1$ 则直接让 $rem \to rem - 1$ 就行了.

复杂度 $O(|S| \cdot |\Sigma| + \sum |x_i|)$

参考代码

```cpp
#include <cstring>
#include <iostream>
#include <string>
using namespace std;
constexpr int N(1e6);

struct SuffixTree {
  static constexpr int N{2 * ::N}, RNG{26}, inf = 1e7;
  int ch[N + 5][RNG + 1];
  int st[N + 5], len[N + 5], link[N + 5], s[::N + 5];
  int now{1}, rem{0}, tot{1}, n{0};
  int cnt[N + 5], vis[N + 5];

  SuffixTree() { len[0] = inf; }

  void clear() {
    memset(ch, 0, sizeof ch);
    now = tot = 1;
    rem = n = 0;
  }

  int new_node(int s, int le) {
    ++tot;
    st[tot] = s;
    len[tot] = le;
    return tot;
  }

  void extend(int x) {
    s[++n] = x;
    ++rem;
    for (int lst{1}; rem;) {
      while (rem > len[ch[now][s[n - rem + 1]]])
        rem -= len[now = ch[now][s[n - rem + 1]]];
      int &v{ch[now][s[n - rem + 1]]}, c{s[st[v] + rem - 1]};
      if (!v || x == c) {
        lst = link[lst] = now;
        if (!v)
          v = new_node(n, inf);
        else
          break;
      } else {
        int u{new_node(st[v], rem - 1)};
        ch[u][c] = v;
        ch[u][x] = new_node(n, inf);
        st[v] += rem - 1;
        len[v] -= rem - 1;
        lst = link[lst] = v = u;
      }
      if (now == 1)
        --rem;
      else
```

```
53          now = link[now];
54        }
55      }
56
57      void init(int u) {
58        if (len[u] > 1e6) return cnt[u] = 1, void();
59        for (int i{0}; i <= RNG; ++i)
60          if (ch[u][i]) init(ch[u][i]), cnt[u] += cnt[ch[u][i]];
61      }
62
63      long long test(const char *t, int m) {
64        static int time{0};
65        ++time;
66        int now{1}, rem{0}, o{0};
67        long long ans{0};
68        for (int i{1}; i <= m; ++i) {
69          while (o < i + m - 1) {
70            while (rem >= len[ch[now][t[o - rem + 1]]])
71              rem -= len[now = ch[now][t[o - rem + 1]]];
72            int v{ch[now][t[o - rem + 1]]}, c{s[st[v] + rem]};
73            if (v && c == t[o + 1]) {
74              ++o;
75              ++rem;
76            } else {
77              break;
78            }
79          }
80          if (o == i + m - 1 && vis[ch[now][t[o - rem + 1]]] != time)
81            ans += cnt[ch[now][t[o - rem + 1]]],
82                vis[ch[now][t[o - rem + 1]]] = time;
83          if (now == 1)
84            --rem;
85          else
86            now = link[now];
87        }
88        return ans;
89      }
90  } T;
91
92  string s;
93
94  int main() {
95    cin >> s;
96    for (int i = 0; i < s.size(); ++i) T.extend(s[i] - 'a' + 1);
97    T.extend(0);
98    T.init(1);
99    int pw;
100   cin >> pw;
101   while (pw--) {
102     cin >> s;
103     int n = s.size();
104     for (auto &ch : s) ch += 1 - 'a';
```

```
105     s = " " + s + s;
106     cout << T.test(s.data(), n) << "\n";
107   }
108   return 0;
109 }
```

参考文献

1. 2021 国家集训队论文《后缀树的构建》代晨昕
2. 炫酷后缀树魔术 - EternalAlexander 的博客[3]

外部参考资料

[1] 洛谷 P3804【模板】后缀自动机（SAM）

[2] CF235C Cyclical Quest

[3] 炫酷后缀树魔术 - EternalAlexander 的博客

8.16 Manacher

描述

给定一个长度为 n 的字符串 s，请找到所有对 (i,j) 使得子串 $s[i...j]$ 为一个回文串．当 $t=t_{\text{rev}}$ 时，字符串 t 是一个回文串（t_{rev} 是 t 的反转字符串）．

解释

显然在最坏情况下可能有 $O(n^2)$ 个回文串，因此似乎一眼看过去该问题并没有线性算法．

但是关于回文串的信息可用 **一种更紧凑的方式** 表达：对于每个位置 $i=0...n-1$，我们找出值 $d_1[i]$ 和 $d_2[i]$．二者分别表示以位置 i 为中心的长度为奇数和长度为偶数的回文串个数．换个角度，二者也表示了以位置 i 为中心的最长回文串的半径长度（半径长度 $d_1[i]$，$d_2[i]$ 均为从位置 i 到回文串最右端位置包含的字符个数）．

举例来说，字符串 $s=\texttt{ababab}$ 以 $s[3]=b$ 为中心有三个奇数长度的回文串，最长回文串半径为 3，也即 $d_1[3]=3$：

$$a\ \overbrace{b\ a\ \underbrace{b}_{s_3}\ a\ b}^{d_1[3]=3}\ c$$

字符串 $s=\texttt{cbaabd}$ 以 $s[3]=a$ 为中心有两个偶数长度的回文串，最长回文串半径为 2，也即 $d_2[3]=2$：

$$c\ \overbrace{b\ a\ \underbrace{a}_{s_3}\ b}^{d_2[3]=2}\ d$$

因此关键思路是，如果以某个位置 i 为中心，我们有一个长度为 l 的回文串，那么我们有以 i 为中心的长度为 $l-2$，$l-4$，等等的回文串．所以 $d_1[i]$ 和 $d_2[i]$ 两个数组已经足够表示字符串中所有子回文串的信息．

一个令人惊讶的事实是，存在一个复杂度为线性并且足够简单的算法计算上述两个「回文性质数组」$d_1[]$ 和 $d_2[]$．在这篇文章中我们将详细的描述该算法．

解法

总的来说，该问题具有多种解法：应用字符串哈希，该问题可在 $O(n \log n)$ 时间内解决，而使用后缀数组和快速 LCA 该问题可在 $O(n)$ 时间内解决.

但是这里描述的算法 **压倒性** 的简单，并且在时间和空间复杂度上具有更小的常数. 该算法由 **Glenn K. Manacher** 在 1975 年提出.

朴素算法

为了避免在之后的叙述中出现歧义，这里我们指出什么是「朴素算法」.

该算法通过下述方式工作：对每个中心位置 i，在比较一对对应字符后，只要可能，该算法便尝试将答案加 1.

该算法是比较慢的：它只能在 $O(n^2)$ 的时间内计算答案.

该朴素算法的实现如下：

实现

- C++

```cpp
vector<int> d1(n), d2(n);
for (int i = 0; i < n; i++) {
  d1[i] = 1;
  while (0 <= i - d1[i] && i + d1[i] < n && s[i - d1[i]] == s[i + d1[i]]) {
    d1[i]++;
  }

  d2[i] = 0;
  while (0 <= i - d2[i] - 1 && i + d2[i] < n &&
         s[i - d2[i] - 1] == s[i + d2[i]]) {
    d2[i]++;
  }
}
```

- Python

```python
d1 = [0] * n
d2 = [0] * n
for i in range(0, n):
    d1[i] = 1
    while 0 <= i - d1[i] and i + d1[i] < n and s[i - d1[i]] == s[i + d1[i]]:
        d1[i] += 1

    d2[i] = 0
    while 0 <= i - d2[i] - 1 and i + d2[i] < n and s[i - d2[i] - 1] == s[i + d2[i]]:
        d2[i] += 1
```

Manacher 算法

这里我们将只描述算法中寻找所有奇数长度子回文串的情况，即只计算 $d_1[]$；寻找所有偶数长度子回文串的算法（即计算数组 $d_2[]$）将只需对奇数情况下的算法进行一些小修改.

为了快速计算，我们维护已找到的最靠右的子回文串的 **边界** $[l,r]$（即具有最大 r 值的回文串，其中 l 和 r 分别为该回文串左右边界的位置）．初始时，我们置 $l=0$ 和 $r=-1$（*-1* 需区别于倒序索引位置，这里可为任意负数，仅为了循环初始时方便）．

过程

现在假设我们要对下一个 i 计算 $d_1[i]$，而之前所有 $d_1[]$ 中的值已计算完毕．我们将通过下列方式计算：

- 如果 i 位于当前子回文串之外，即 $i>r$，那么我们调用朴素算法．

  因此我们将连续地增加 $d_1[i]$，同时在每一步中检查当前的子串 $[i-d_1[i]...i+d_1[i]]$（$d_1[i]$ 表示半径长度，下同）是否为一个回文串．如果我们找到了第一处对应字符不同，又或者碰到了 s 的边界，则算法停止．在两种情况下我们均已计算完 $d_1[i]$．此后，仍需记得更新 (l,r)．
- 现在考虑 $i\le r$ 的情况．我们将尝试从已计算过的 $d_1[]$ 的值中获取一些信息．首先在子回文串 (l,r) 中反转位置 i，即我们得到 $j=l+(r-i)$．现在来考察值 $d_1[j]$．因为位置 j 同位置 i 对称，我们 **几乎总是** 可以置 $d_1[i]=d_1[j]$．该想法的图示如下（可认为以 j 为中心的回文串被「拷贝」至以 i 为中心的位置上）：

  $$\dots \overbrace{s_l \dots \underbrace{s_{j-d_1[j]+1} \dots s_j \dots s_{j+d_1[j]-1}}_{\text{palindrome}} \dots \underbrace{s_{i-d_1[j]+1} \dots s_i \dots s_{i+d_1[j]-1}}_{\text{palindrome}} \dots s_r}^{\text{palindrome}} \dots$$

  然而有一个 **棘手的情况** 需要被正确处理：当「内部」的回文串到达「外部」回文串的边界时，即 $j-d_1[j]+1\le l$（或者等价的说，$i+d_1[j]-1\ge r$）．因为在「外部」回文串范围以外的对称性没有保证，因此直接置 $d_1[i]=d_1[j]$ 将是不正确的：我们没有足够的信息来断言在位置 i 的回文串具有同样的长度．

  实际上，为了正确处理这种情况，我们应该「截断」回文串的长度，即置 $d_1[i]=r-i+1$．之后我们将运行朴素算法以尝试尽可能增加 $d_1[i]$ 的值．

  该种情况的图示如下（以 j 为中心的回文串已经被截断以落在「外部」回文串内）：

  $$\dots \overbrace{\underbrace{s_l \dots s_j \dots s_{j+(j-l)}}_{\text{palindrome}} \dots \underbrace{s_{i-(r-i)} \dots s_i \dots s_r}_{\text{palindrome}}}^{\text{palindrome}} \underbrace{\cdots\cdots\cdots\cdots}_{\text{try moving here}}$$

  该图示显示出，尽管以 j 为中心的回文串可能更长，以致于超出「外部」回文串，但在位置 i，我们只能利用其完全落在「外部」回文串内的部分．然而位置 i 的答案可能比这个值更大，因此接下来我们将运行朴素算法来尝试将其扩展至「外部」回文串之外，也即标识为 "try moving here" 的区域．

  最后，仍有必要提醒的是，我们应当记得在计算完每个 $d_1[i]$ 后更新值 (l,r)．

同时，再让我们重复一遍：计算偶数长度回文串数组 $d_2[]$ 的算法同上述计算奇数长度回文串数组 $d_1[]$ 的算法十分类似．

Manacher 算法的复杂度

因为在计算一个特定位置的答案时我们总会运行朴素算法，所以一眼看去该算法的时间复杂度为线性的事实并不显然．

然而更仔细的分析显示出该算法具有线性复杂度．此处我们需要指出，计算 Z 函数的算法→8.9@p947 和该算法较为类似，并同样具有线性时间复杂度．

实际上，注意到朴素算法的每次迭代均会使 r 增加 1，以及 r 在算法运行过程中从不减小．这两个观察告诉我们朴素算法总共会进行 $O(n)$ 次迭代．

Manacher 算法的另一部分显然也是线性的，因此总复杂度为 $O(n)$．

Manacher 算法的实现

分类讨论

为了计算 $d_1[]$，我们有以下代码：

- C++

```cpp
vector<int> d1(n);
for (int i = 0, l = 0, r = -1; i < n; i++) {
  int k = (i > r) ? 1 : min(d1[l + r - i], r - i + 1);
  while (0 <= i - k && i + k < n && s[i - k] == s[i + k]) {
    k++;
  }
  d1[i] = k--;
  if (i + k > r) {
    l = i - k;
    r = i + k;
  }
}
```

- Python

```python
d1 = [0] * n
l, r = 0, -1
for i in range(0, n):
    k = 1 if i > r else min(d1[l + r - i], r - i + 1)
    while 0 <= i - k and i + k < n and s[i - k] == s[i + k]:
        k += 1
    d1[i] = k
    k -= 1
    if i + k > r:
        l = i - k
        r = i + k
```

计算 $d_2[]$ 的代码十分类似，但是在算术表达式上有些许不同：

- C++

```cpp
vector<int> d2(n);
for (int i = 0, l = 0, r = -1; i < n; i++) {
  int k = (i > r) ? 0 : min(d2[l + r - i + 1], r - i + 1);
  while (0 <= i - k - 1 && i + k < n && s[i - k - 1] == s[i + k]) {
    k++;
  }
  d2[i] = k--;
  if (i + k > r) {
    l = i - k - 1;
    r = i + k;
  }
}
```

- Python

```python
d2 = [0] * n
l, r = 0, -1
for i in range(0, n):
    k = 0 if i > r else min(d2[l + r - i + 1], r - i + 1)
    while 0 <= i - k - 1 and i + k < n and s[i - k - 1] == s[i + k]:
```

```
        k += 1
    d2[i] = k
    k -= 1
    if i + k > r:
        l = i - k - 1
        r = i + k
```

统一处理

虽然在讲解过程及上述实现中我们将 $d_1[]$ 和 $d_2[]$ 的计算分开考虑，但实际上可以通过一个技巧将二者的计算统一为 $d_1[]$ 的计算.

给定一个长度为 n 的字符串 s，我们在其 $n+1$ 个空中插入分隔符 $\#$，从而构造一个长度为 $2n+1$ 的字符串 s'. 举例来说，对于字符串 $s=$ `abababc`，其对应的 $s'=$ `#a#b#a#b#a#b#c#`.

对于字母间的 $\#$，其实际意义为 s 中对应的「空」. 而两端的 $\#$ 则是为了实现的方便.

注意到，在对 s' 计算 $d_1[]$ 后，对于一个位置 i，$d_1[i]$ 所描述的最长的子回文串必定以 $\#$ 结尾（若以字母结尾，由于字母两侧必定各有一个 $\#$，因此可向外扩展一个得到一个更长的）. 因此，对于 s 中一个以字母为中心的极大子回文串，设其长度为 $m+1$，则其在 s' 中对应一个以相应字母为中心，长度为 $2m+3$ 的极大子回文串；而对于 s 中一个以空为中心的极大子回文串，设其长度为 m，则其在 s' 中对应一个以相应表示空的 $\#$ 为中心，长度为 $2m+1$ 的极大子回文串（上述两种情况下的 m 均为偶数，但该性质成立与否并不影响结论）. 综合以上观察及少许计算后易得，在 s' 中，$d_1[i]$ 表示在 s 中以对应位置为中心的极大子回文串的 **总长度加一**.

上述结论建立了 s' 的 $d_1[]$ 同 s 的 $d_1[]$ 和 $d_2[]$ 间的关系.

由于该统一处理本质上即求 s' 的 $d_1[]$，因此在得到 s' 后，代码同上节计算 $d_1[]$ 的一样.

练习题目

- UVa #11475 "Extend to Palindrome"[1]
- 「国家集训队」最长双回文串[2]
- CF1326D2. Labyrinth[3]

本页面主要译自博文 Нахождение всех подпалиндромов[4] 与其英文翻译版 Finding all sub-palindromes in $O(N)$[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.

外部参考资料

[1] UVa #11475 "Extend to Palindrome"

[2] 「国家集训队」最长双回文串

[3] CF1326D2. Labyrinth

[4] Нахождение всех подпалиндромов

[5] Finding all sub-palindromes in $O(N)$

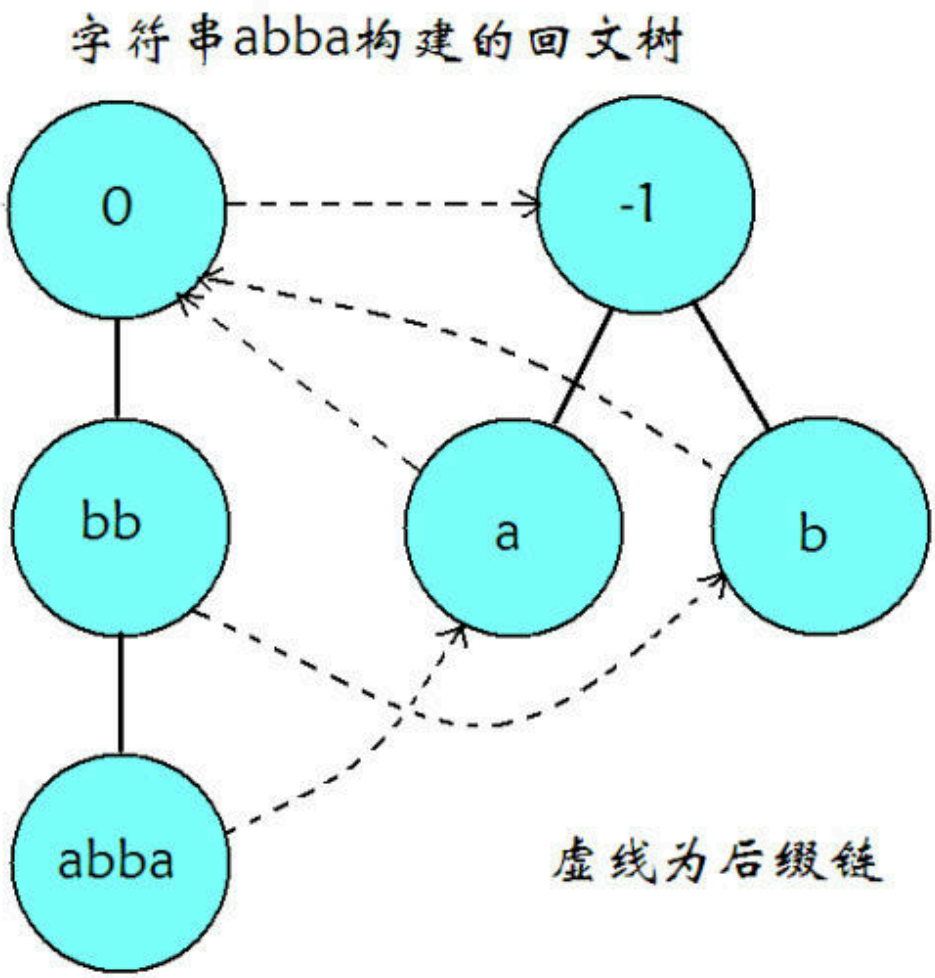
8.17 回文树

定义

回文树（EER Tree，Palindromic Tree，也被称为回文自动机）是一种可以存储一个串中所有回文子串的高效数据结构. 最初由 Mikhail Rubinchik 和 Arseny M. Shur 在 2015 年发表. 它的灵感来源于后缀树等字符串后缀数据结构，使用回文树可以简单高效地解决一系列涉及回文串的问题.

结构

回文树大概长这样



和其它自动机类似的，回文树也是由转移边和后缀链接（fail 指针）组成，每个节点都可以代表一个回文子串.

因为回文串长度分为奇数和偶数，我们可以像 manacher 那样加入一个不在字符集中的字符（如 '#'）作为分隔符来将所有回文串的长度都变为奇数，但是这样过于麻烦了．有没有更好的办法呢?

答案自然是有．更好的办法就是建两棵树，一棵树中的节点对应的回文子串长度均为奇数，另一棵树中的节点对应的回文子串长度均为偶数.

和其它的自动机一样，一个节点的 fail 指针指向的是这个节点所代表的回文串的最长回文后缀所对应的节点，但是转移边并非代表在原节点代表的回文串后加一个字符，而是表示在原节点代表的回文串前后各加一个相同的字符（不难理解，因为要保证存的是回文串）.

我们还需要在每个节点上维护此节点对应回文子串的长度 len，这个信息保证了我们可以轻松地构造出回文树.

建造

回文树有两个初始状态，分别代表长度为 $-1, 0$ 的回文串．我们可以称它们为奇根，偶根．它们不表示任何实际的字符串，仅作为初始状态存在，这与其他自动机的根节点是异曲同工的.

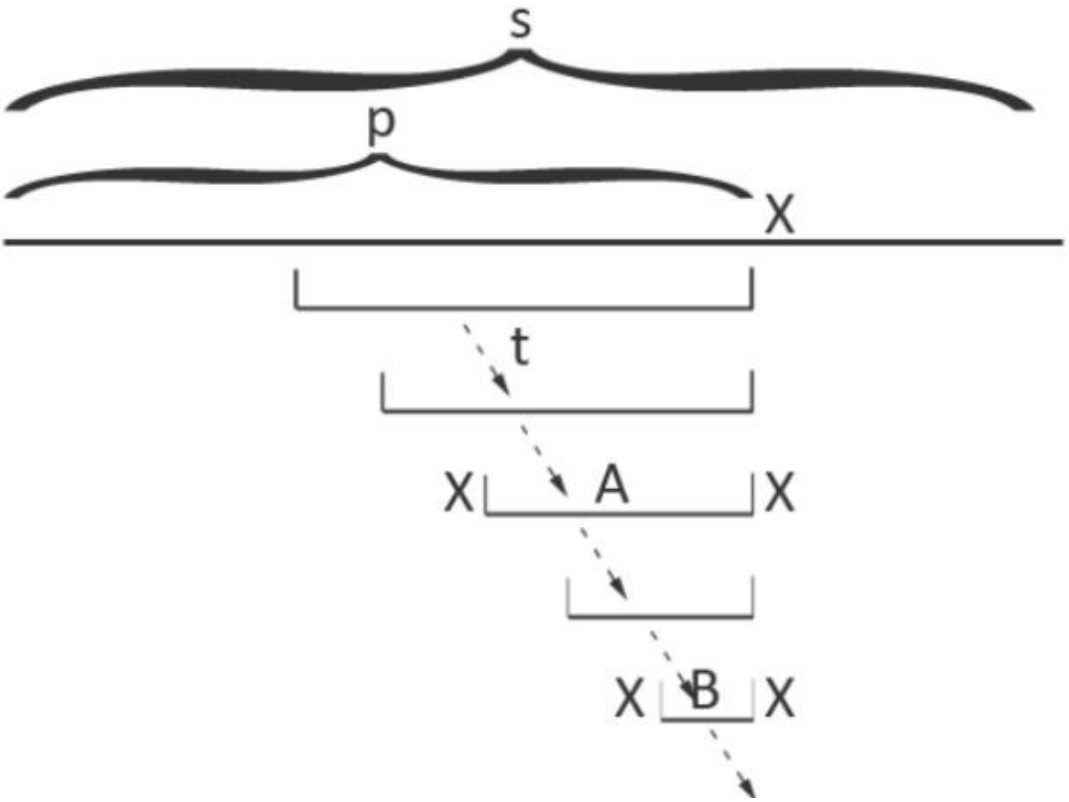
偶根的 fail 指针指向奇根，而我们并不关心奇根的 fail 指针，因为奇根不可能失配（奇根转移出的下一个状态长度为 1，即单个字符．一定是回文子串）

类似后缀自动机，我们增量构造回文树．

考虑构造完前 $p-1$ 个字符的回文树后，向自动机中添加在原串里位置为 p 的字符．

我们从以上一个字符结尾的最长回文子串对应的节点开始，不断沿着 fail 指针走，直到找到一个节点满足 $s_p = s_{p-len-1}$，即满足此节点所对应回文子串的上一个字符与待添加字符相同．

这里贴出论文中的那张图



我们通过跳 fail 指针找到 A 所对应的节点，然后两边添加 X 就到了现在的回文串了（即 XAX），很显然，这个节点就是以 p 结尾的最长回文子串对应的树上节点．（同时，这个时候长度 -1 节点优势出来了，如果没有 X 能匹配条件就是同一个位置的 $s_p = s_p$，就自然得到了代表字符 X 的节点．）此时要判断一下：没有这个节点，就需要新建．

然后我们还需要求出新建的节点的 fail 指针．具体方法与上面的过程类似，不断跳转 fail 指针，从 A 出发，即可找到 XAX 的最长回文后缀 XBX，将对应节点设为 fail 指针所指的对象即可．

显然，这个节点是不需新建的，A 的前 len_B 位和后 len_B 位相同，都是 B，前 len_B 位的两端根据回文串对应关系，都是 X，后面被钦定了是 X，于是这个节点 XBX 肯定已经被包含了．

如果 fail 没匹配到，那么将它连向长度为 0 的那个节点，显然这是可行的（因为这是所有节点的后缀）．

线性状态数证明

定理

对于一个字符串 s，它的本质不同回文子串个数最多只有 $|s|$ 个．

证明

考虑使用数学归纳法．

- 当 $|s|=1$ 时，s 只有一个字符，同时也只有一个子串，并且这个子串是回文的，因此结论成立.
- 当 $|s|>1$ 时，设 $t=sc$，其中 t 表示 s 最后增加一个字符 c 后形成的字符串，假设结论对 s 串成立. 考虑以最后一个字符 c 结尾的回文子串，假设它们的左端点由小到大排序为 $l_1,l_2,...,l_k$. 由于 $t[l_1..|t|]$ 是回文串，因此对于所有位置 $l_1\le p\le|t|$，有 $t[p..|t|]=t[l_1..l_1+|t|-p]$. 所以，对于 $1<i\le k$，$t[l_i..|t|]$ 已经在 $t[1..|t|-1]$ 中出现过. 因此，每次增加一个字符，本质不同的回文子串个数最多增加 1 个.

由数学归纳法，可知该定理成立.

因此回文树状态数是 $O(|s|)$ 的. 对于每一个状态，它实际只代表一个本质不同的回文子串，即转移到该节点的状态唯一，因此总转移数也是 $O(|s|)$ 的.

正确性证明

以上图为例，增加当前字符 X，由线性状态数的证明，我们只需要找到包含最后一个字符 X 的最长回文后缀，也就是 XAX. 继续寻找 XAX 的最长回文后缀 XBX，建立后缀链接. XBX 对应状态已经在回文树中出现，包含最后一个字符的回文后缀就是 XAX，XBX 本身及其对应状态在 fail 树上的所有祖先.

对于 s 回文树的构造，令 $n=|s|$，显然除了跳 fail 指针的其他操作都是 $O(n)$ 的.

加入字符时，在上一次的基础上，每次跳 fail 后对应节点在 fail 树的深度 -1，而连接 fail 后，仅为深度 $+1$（但 fail 为 0 时（即到 -1 才符合），深度相当于在 -1 的基础上 $+2$）.

因为只加入 n 个字符，所以只会加 n 次深度，最多也只会跳 $2n$ 次 fail.

因此，构造 s 的回文树的时间复杂度是 $O(|s|)$.

应用

本质不同回文子串个数

由线性状态数的证明，容易知道一个串的本质不同回文子串个数等于回文树的状态数（排除奇根和偶根两个状态）.

回文子串出现次数

建出回文树，使用类似后缀自动机统计出现次数的方法.

由于回文树的构造过程中，节点本身就是按照拓扑序插入，因此只需要逆序枚举所有状态，将当前状态的出现次数加到其 fail 指针对应状态的出现次数上即可.

例题：「APIO2014」回文串[1]

定义 s 的一个子串的存在值为这个子串在 s 中出现的次数乘以这个子串的长度. 对于给定的字符串 s，求所有回文子串中的最大存在值.

参考代码

```cpp
#include <algorithm>
#include <cstring>
#include <iostream>
#include <string>
using namespace std;
constexpr int MAXN = 300000 + 5;

namespace pam {
int sz, tot, last;
int cnt[MAXN], ch[MAXN][26], len[MAXN], fail[MAXN];
```

```
char s[MAXN];

int node(int l) {  // 建立一个新节点，长度为 l
  sz++;
  memset(ch[sz], 0, sizeof(ch[sz]));
  len[sz] = l;
  fail[sz] = cnt[sz] = 0;
  return sz;
}

void clear() {  // 初始化
  sz = -1;
  last = 0;
  s[tot = 0] = '$';
  node(0);
  node(-1);
  fail[0] = 1;
}

int getfail(int x) {  // 找后缀回文
  while (s[tot - len[x] - 1] != s[tot]) x = fail[x];
  return x;
}

void insert(char c) {  // 建树
  s[++tot] = c;
  int now = getfail(last);
  if (!ch[now][c - 'a']) {
    int x = node(len[now] + 2);
    fail[x] = ch[getfail(fail[now])][c - 'a'];
    ch[now][c - 'a'] = x;
  }
  last = ch[now][c - 'a'];
  cnt[last]++;
}

long long solve() {
  long long ans = 0;
  for (int i = sz; i >= 0; i--) {
    cnt[fail[i]] += cnt[i];
  }
  for (int i = 1; i <= sz; i++) {  // 更新答案
    ans = max(ans, 1ll * len[i] * cnt[i]);
  }
  return ans;
}
}  // namespace pam

string s;

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
```

```
63   pam::clear();
64   cin >> s;
65   for (int i = 0; i < s.size(); i++) {
66     pam::insert(s[i]);
67   }
68   cout << pam::solve() << '\n';
69   return 0;
70 }
71
```

最小回文划分

给定一个字符串 $s(1 \le |s| \le 10^5)$，求最小的 k，使得存在 $s_1, s_2, ..., s_k$，满足 $s_i(1 \le i \le k)$ 均为回文串，且 $s_1, s_2, ..., s_k$ 依次连接后得到的字符串等于 s.

考虑动态规划，记 $dp[i]$ 表示 s 长度为 i 的前缀的最小划分数，转移只需要枚举以第 i 个字符结尾的所有回文串

$$dp[i] = 1 + \min_{s[j+1..i] \text{ 为回文串}} dp[j]$$

由于一个字符串最多会有 $O(n^2)$ 个回文子串，因此上述算法的时间复杂度为 $O(n^2)$，无法接受，为了优化转移过程，下面给出一些引理.

记字符串 s 长度为 i 的前缀为 $pre(s, i)$，长度为 i 的后缀为 $suf(s, i)$.

周期：若 $0 < p \le |s|$，$\forall 1 \le i \le |s| - p, s[i] = s[i + p]$，就称 p 是 s 的周期.

border：若 $0 \le r < |s|$，$pre(s, r) = suf(s, r)$，就称 $pre(s, r)$ 是 s 的 border.

周期和 border 的关系：t 是 s 的 border，当且仅当 $|s| - |t|$ 是 s 的周期.

证明

若 t 是 s 的 border，那么 $pre(s, |t|) = suf(s, |t|)$，因此 $\forall 1 \le i \le |t|, s[i] = s[|s| - |t| + i]$，所以 $|s| - |t|$ 就是 s 的周期.

若 $|s| - |t|$ 为 s 周期，则 $\forall 1 \le i \le |s| - (|s| - |t|) = |t|, s[i] = s[|s| - |t| + i]$，因此 $pre(s, |t|) = suf(s, |t|)$，所以 t 是 s 的 border.

引理一

t 是回文串 s 的后缀，t 是 s 的 border 当且仅当 t 是回文串.

证明

对于 $1 \le i \le |t|$，由 s 和 t 为回文串，因此有 $s[i] = s[|s| - i + 1] = s[|s| - |t| + i]$，所以 t 是 s 的 border.

对于 $1 \le i \le |t|$，由 t 是 s 的 border，有 $s[i] = s[|s| - |t| + i]$，由 s 是回文串，有 $s[i] = s[|s| - i + 1]$，因此 $s[|s| - i + 1] = s[|s| - |t| + i]$，所以 t 是回文串.

下图中，相同颜色的位置表示字符对应相同.



引理二

t 是串 s 的 border ($|s| \le 2|t|$)，s 是回文串当且仅当 t 是回文串.

证明

若 s 是回文串，由引理 1，t 也是回文串.

若 t 是回文串，由 t 是 s 的 border，因此 $\forall 1 \le i \le |t|, s[i] = s[|s| - |t| + i] = s[|s| - i + 1]$，因为 $|s| \le 2|t|$，所以 s 也是回文串.

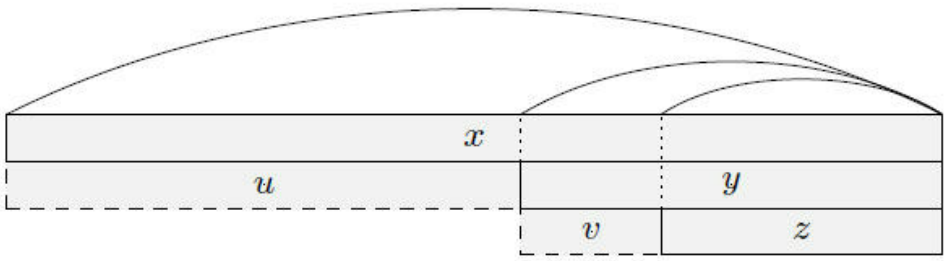
引理三

t 是回文串 s 的 border，则 $|s| - |t|$ 是 s 的周期，$|s| - |t|$ 为 s 的最小周期，当且仅当 t 是 s 的最长回文真后缀.

引理四

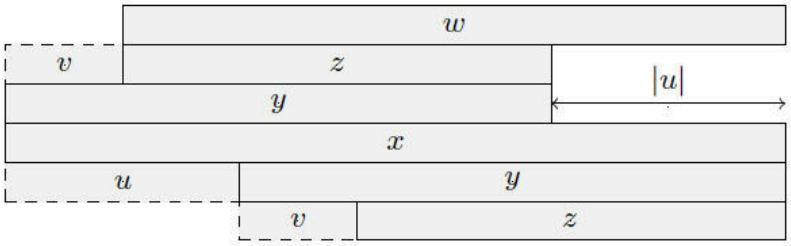
x 是一个回文串，y 是 x 的最长回文真后缀，z 是 y 的最长回文真后缀. 令 u, v 分别为满足 $x = uy, y = vz$ 的字符串，则有下面三条性质

1. $|u| \ge |v|$;
2. 如果 $|u| > |v|$，那么 $|u| > |z|$;
3. 如果 $|u| = |v|$，那么 $u = v$.



证明

1. 由引理 3 的推论，$|u| = |x| - |y|$ 是 x 的最小周期，$|v| = |y| - |z|$ 是 y 的最小周期. 考虑反证法，假设 $|u| < |v|$，因为 y 是 x 的后缀，所以 u 既是 x 的周期，也是 y 的周期，而 $|v|$ 是 y 的最小周期，矛盾. 所以 $|u| \ge |v|$.
2. 因为 y 是 x 的 border，所以 v 是 x 的前缀，设字符串 w，满足 $x = vw$ (如下图所示)，其中 z 是 w 的 border. 考虑反证法，假设 $|u| \le |z|$，那么 $|zu| \le 2|z|$，所以由引理 2，w 是回文串，由引理 1，w 是 x 的 border，又因为 $|u| > |v|$，所以 $|w| > |y|$，矛盾. 所以 $|u| > |z|$.
3. u, v 都是 x 的前缀，$|u| = |v|$，所以 $u = v$.



推论

s 的所有回文后缀按照长度排序后，可以划分成 $\log|s|$ 段等差数列.

证明

设 s 的所有回文后缀长度从小到大排序为 $l_1, l_2, ..., l_k$. 对于任意 $2 \le i \le k-1$，若 $l_i - l_{i-1} = l_{i+1} - l_i$，则 l_{i-1}, l_i, l_{i+1} 构成一个等差数列. 否则 $l_i - l_{i-1} \neq l_{i+1} - l_i$，由引理 4，有 $l_{i+1} - l_i > l_i - l_{i-1}$，且 $l_{i+1} - l_i > l_{i-1}$，$l_{i+1} > 2l_{i-1}$. 因此，若相邻两对回文后缀的长度之差发生变化，那么这个最大长度一定会相对于最小长度翻一倍. 显然，长度翻倍最多只会发生 $O(\log|s|)$ 次，也就是 s 的回文后缀长度可以划分成 $\log|s|$ 段等差数列.

该推论也可以通过使用弱周期引理，对 s 的最长回文后缀的所有 border 按照长度 x 分类，$x \in [2^0, 2^1), [2^1, 2^2), ..., [2^k, n)$，考虑这 $\log|s|$ 组内每组的最长 border 进行证明. 详细证明可以参考金策的《字符串算法选讲》和陈孙立的 2019 年 IOI 国家候选队论文《子串周期查询问题的相关算法及其应用》.

有了这个结论后，我们现在可以考虑如何优化 dp 的转移.

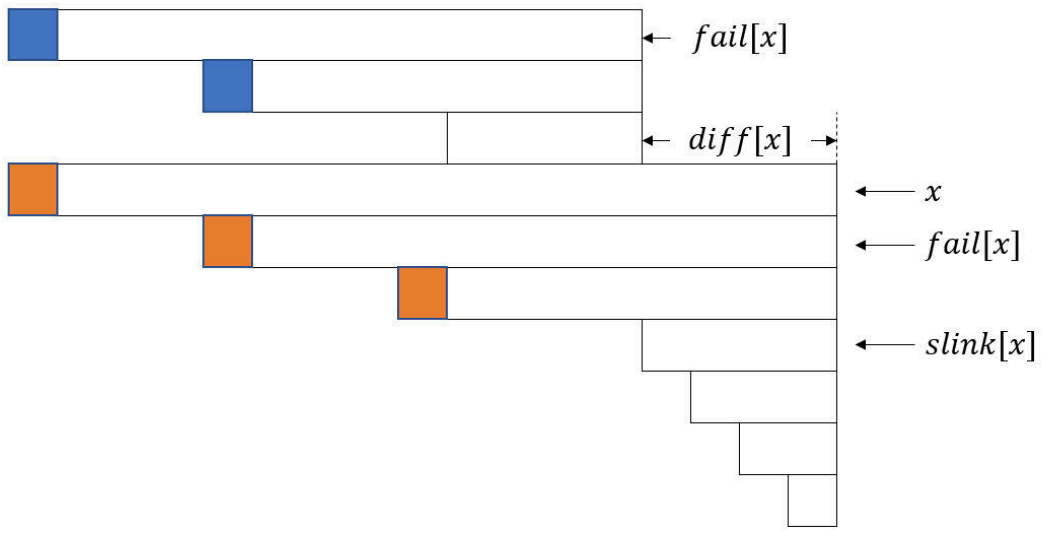
优化

回文树上的每个节点 u 需要多维护两个信息，$diff[u]$ 和 $slink[u]$. $diff[u]$ 表示节点 u 和 $fail[u]$ 所代表的回文串的长度差，即 $len[u] - len[fail[u]]$. $slink[u]$ 表示 u 一直沿着 fail 向上跳到第一个节点 v，使得 $diff[v] \neq diff[u]$，也就是 u 所在等差数列中长度最小的那个节点.

根据上面证明的结论，如果使用 $slink$ 指针向上跳的话，每向后填加一个字符，只需要向上跳 $O(\log|s|)$ 次. 因此，可以考虑将一个等差数列表示的所有回文串的 dp 值之和（在原问题中指 min），记录到最长的那一个回文串对应节点上.

$g[v]$ 表示 v 所在等差数列的 dp 值之和，且 v 是这个等差数列中长度最长的节点，则 $g[v] = \sum_{slink[x]=slink[v]} dp[i - len[x]]$，这里 i 是当前枚举到的下标.

下面我们考虑如何更新 g 数组和 dp 数组. 以下图为例，假设当前枚举到第 i 个字符，回文树上对应节点为 x. $g[x]$ 为橙色三个位置的 dp 值之和（最短的回文串 $slink[x]$ 算在下一个等差数列中）. $fail[x]$ 上一次出现位置是 $i - diff[x]$（在 $i - diff[x]$ 处结束），$g[fail[x]]$ 包含的 dp 值是蓝色位置. 因此，$g[x]$ 实际上等于 $g[fail[x]]$ 和多出来一个位置的 dp 值之和，多出来的位置是 $i - (len[slink[x]] + diff[x])$. 最后再用 $g[x]$ 去更新 $dp[i]$，这部分等差数列的贡献就计算完毕了，不断跳 $slink[x]$，重复这个过程即可. 具体实现方式可参考例题代码.



最后，上述做法的正确性依赖于：如果 x 和 $fail[x]$ 属于同一个等差数列，那么 $fail[x]$ 上一次出现位置是 $i - diff[x]$.

证明

根据引理 1，$fail[x]$ 是 x 的 border，因此其在 $i - diff[x]$ 处出现.

假设 $fail[x]$ 在 $(i - diff[x], i)$ 中的 j 位置出现. 由于 x 和 $fail[x]$ 属于同一个等差数列，因此 $2 \mid fail[x]| \geq x$. 多余的 $fail[x]$ 和 $i - diff[x]$ 处的 $fail[x]$ 有交集，记交集为 w，设串 u 满足 $uw = fail[x]$. 用类似引理 1 的方式可以证明，w 是回文串，而 x 的前缀 $s[i - len[x] + 1..j] = uwu$ 也是回文串，这与 $fail[x]$ 是 x 的最长回文前缀（后缀）矛盾.

例题：Codeforces 932G Palindrome Partition[2]

给定一个字符串 s，要求将 s 划分为 $t_1, t_2, ..., t_k$，其中 k 是偶数，且 $t_i = t_{k-i+1}$，求这样的划分方案数.

题解

构造字符串 $t = s[0]s[n-1]s[1]s[n-2]s[2]s[n-3]...s[n/2-1]s[n/2]$，问题等价于求 t 的偶回文划分方案数，把上面的转移方程改成求和形式并且只在偶数位置更新 dp 数组即可. 时间复杂度 $O(n \log n)$，空间复杂度 $O(n)$.

参考代码

```cpp
#include <cstring>
#include <iostream>
#include <string>
using namespace std;
using ll = long long;
constexpr int mod = 1e9 + 7;
constexpr int MAXN = 1000000 + 5;

int add(int x, int y) {
```

```
  x += y;
  return x >= mod ? x -= mod : x;
}

namespace pam {
int sz, tot, last;
int ch[MAXN][26], len[MAXN], fail[MAXN];
int cnt[MAXN], dep[MAXN], dif[MAXN], slink[MAXN];
char s[MAXN];

int node(int l) {  // 建立一个长度为 l 的新节点
  sz++;
  memset(ch[sz], 0, sizeof(ch[sz]));
  len[sz] = l;
  fail[sz] = 0;
  cnt[sz] = 0;
  dep[sz] = 0;
  return sz;
}

void clear() {  // 初始化
  sz = -1;
  last = 0;
  s[tot = 0] = '$';
  node(0);
  node(-1);
  fail[0] = 1;
}

int getfail(int x) {  // 找到后缀回文
  while (s[tot - len[x] - 1] != s[tot]) x = fail[x];
  return x;
}

void insert(char c) {  // 建树
  s[++tot] = c;
  int now = getfail(last);
  if (!ch[now][c - 'a']) {
    int x = node(len[now] + 2);
    fail[x] = ch[getfail(fail[now])][c - 'a'];
    dep[x] = dep[fail[x]] + 1;
    ch[now][c - 'a'] = x;
    dif[x] = len[x] - len[fail[x]];
    if (dif[x] == dif[fail[x]])
      slink[x] = slink[fail[x]];
    else
      slink[x] = fail[x];
  }
  last = ch[now][c - 'a'];
  cnt[last]++;
}
}  // namespace pam
```

```
62
63 using pam::dif;
64 using pam::fail;
65 using pam::len;
66 using pam::slink;
67 int n, dp[MAXN], g[MAXN];
68 string s;
69 char t[MAXN];
70
71 int main() {
72   cin.tie(nullptr)->sync_with_stdio(false);
73   pam::clear();
74   cin >> s;
75   n = s.size();
76   s = " " + s;
77   for (int i = 1, j = 0; i <= n; i++) t[++j] = s[i], t[++j] = s[n - i + 1];
78   dp[0] = 1;
79   for (int i = 1; i <= n; i++) {
80     pam::insert(t[i]);
81     for (int x = pam::last; x > 1; x = slink[x]) {
82       g[x] = dp[i - len[slink[x]] - dif[x]];
83       if (dif[x] == dif[fail[x]]) g[x] = add(g[x], g[fail[x]]);
84       if (i % 2 == 0) dp[i] = add(dp[i], g[x]);  // 在偶数位置更新 dp 数组
85     }
86   }
87   cout << dp[n];
88   return 0;
89 }
90
```

例题

- 最长双回文串[3]
- 拉拉队排练[4]
- 「SHOI2011」双倍回文[5]
- HDU 5421 Victor and String[6]
- CodeChef Palindromeness[7]

相关资料

- EERTREE: An Efficient Data Structure for Processing Palindromes in Strings[8]
- Palindromic tree[9]
- 2017 年 IOI 国家候选队论文集 回文树及其应用 翁文涛
- 2019 年 IOI 国家候选队论文集 子串周期查询问题的相关算法及其应用 陈孙立
- 字符串算法选讲 金策
- A bit more about palindromes[10]
- A Subquadratic Algorithm for Minimum Palindromic Factorization[11]

外部参考资料

[1] 「APIO2014」回文串 

[2] Codeforces 932G Palindrome Partition 

[3] 最长双回文串 

[4] 拉拉队排练 

[5] 「SHOI2011」双倍回文 

[6] HDU 5421 Victor and String 

[7] CodeChef Palindromeness 

[8] EERTREE: An Efficient Data Structure for Processing Palindromes in Strings 

[9] Palindromic tree 

[10] A bit more about palindromes 

[11] A Subquadratic Algorithm for Minimum Palindromic Factorization 

8.18 序列自动机

在阅读本文之前，请先阅读 自动机→13.8@p3284.

定义

序列自动机是接受且仅接受一个字符串的子序列的自动机.

本文中用 s 代指这个字符串.

状态

若 s 包含 n 个字符，那么序列自动机包含 $n+1$ 个状态.

令 t 是 s 的一个子序列，那么 $\delta(start,t)$ 是 t 在 s 中第一次出现时末端的位置.

也就是说，一个状态 i 表示前缀 $s[1..i]$ 的子序列与前缀 $s[1..i-1]$ 的子序列的差集.

序列自动机上的所有状态都是接受状态.

转移

由状态定义可以得到，$\delta(u,c)=\min\{i \mid i>u, s[i]=c\}$，也就是字符 c 下一次出现的位置.

为什么是「下一次」出现的位置呢？因为若 $i>j$，后缀 $s[i..\mid s\mid]$ 的子序列是后缀 $s[j..\mid s\mid]$ 的子序列的子集，一定是选尽量靠前的最优.

实现

从后向前扫描，过程中维护每个字符最前的出现位置：

```
1  Input. A string S
2  Output. The state transition of the sequence automaton of S
3  Method.
4  for c ∈ Σ
5      next[c] ← null
6  for i ← | S | downto 1
7      next[S[i]] ← i
8      for c ∈ Σ
9          δ(i − 1, c) ← next[c]
10 return δ
```

这样构建的复杂度是 $O(n \mid \Sigma \mid)$.

例题

「HEOI2015」最短不公共子串[1]

给你两个由小写英文字母组成的串 A 和 B（$1 \leq |A|, |B| \leq 2000$），求：

1. A 的一个最短的子串，它不是 B 的子串；
2. A 的一个最短的子串，它不是 B 的子序列；
3. A 的一个最短的子序列，它不是 B 的子串；
4. A 的一个最短的子序列，它不是 B 的子序列.

题解

题目的 1 和 3 两问需要后缀自动机，而且做法类似，在这里只讲解 2 和 4 两问.

第 2 问比较简单，枚举 A 的子串输入进 B 的序列自动机，若不接受则计入答案.

第 4 问需要 DP. 令 $f(i,j)$ 表示在 A 的序列自动机中处于状态 i，在 B 的序列自动机中处于状态 j，需要再添加多少个字符能够不是公共子序列. 状态转移方程为：

$$f(i,j) = \min_{\delta_A(i,c) \neq null} f(\delta_A(i,c), \delta_B(j,c)) + 1.$$

转移起点为 $f(i, null) = 0$.

参考代码

```cpp
#include <algorithm>
#include <cstring>
#include <iostream>
#include <string>
using namespace std;

constexpr int N = 2005;

string s, t;
int na[N][26], nb[N][26], nxt[26];
int n, m, a[N], b[N], tot = 1, p = 1, f[N][N << 1];

struct SAM {
  int par, ch[26], len;
} sam[N << 1];

void insert(int x) {
  int np = ++tot;  // 新节点
  sam[np].len = sam[p].len + 1;
  while (p && !sam[p].ch[x]) {
    sam[p].ch[x] = np;
    p = sam[p].par;
  }
  if (p == 0)
    sam[np].par = 1;
  else {
    int q = sam[p].ch[x];
    if (sam[q].len == sam[p].len + 1)
```

```
      sam[np].par = q;
    else {
      int nq = ++tot;
      sam[nq].len = sam[p].len + 1;
      memcpy(sam[nq].ch, sam[q].ch, sizeof(sam[q].ch));
      sam[nq].par = sam[q].par;
      sam[q].par = sam[np].par = nq;
      while (p && sam[p].ch[x] == q) {
        sam[p].ch[x] = nq;
        p = sam[p].par;
      }
    }
  }
  p = np;
}

int main() {
  cin >> s >> t;

  n = s.size();
  m = t.size();
  s = " " + s;
  t = " " + t;

  for (int i = 1; i <= n; ++i) a[i] = s[i] - 'a';
  for (int i = 1; i <= m; ++i) b[i] = t[i] - 'a';

  for (int i = 1; i <= m; ++i) insert(b[i]);

  // nxt[S[i]]<-i
  for (int i = 0; i < 26; ++i) nxt[i] = n + 1;
  for (int i = n; i >= 0; --i) {
    memcpy(na[i], nxt, sizeof(nxt));
    nxt[a[i]] = i;
  }

  for (int i = 0; i < 26; ++i) nxt[i] = m + 1;
  for (int i = m; i >= 0; --i) {
    memcpy(nb[i], nxt, sizeof(nxt));
    nxt[b[i]] = i;
  }

  // 四种情况计算答案
  //  1
  int ans = N;
  for (int l = 1; l <= n; ++l) {
    for (int r = l, u = 1; r <= n; ++r) {
      u = sam[u].ch[a[r]];
      if (!u) {
        ans = min(ans, r - l + 1);
        break;
      }
```

```
    }
  }

  cout << (ans == N ? -1 : ans) << '\n';

  // 2
  ans = N;

  for (int l = 1; l <= n; ++l) {
    for (int r = l, u = 0; r <= n; ++r) {
      u = nb[u][a[r]];
      if (u == m + 1) {
        ans = min(ans, r - l + 1);
        break;
      }
    }
  }

  cout << (ans == N ? -1 : ans) << '\n';

  // 3
  for (int i = n; i >= 0; --i) {
    for (int j = 1; j <= tot; ++j) {
      f[i][j] = N;
      for (int c = 0; c < 26; ++c) {
        int u = na[i][c];
        int v = sam[j].ch[c];
        if (u <= n) f[i][j] = min(f[i][j], f[u][v] + 1);
      }
    }
  }

  cout << (f[0][1] == N ? -1 : f[0][1]) << '\n';

  // 4
  memset(f, 0, sizeof(f));

  for (int i = n; i >= 0; --i) {
    for (int j = 0; j <= m; ++j) {
      f[i][j] = N;
      for (int c = 0; c < 26; ++c) {
        int u = na[i][c];
        int v = nb[j][c];
        if (u <= n) f[i][j] = min(f[i][j], f[u][v] + 1);
      }
    }
  }

  cout << (f[0][0] == N ? -1 : f[0][0]) << '\n';

  return 0;
}
```

```
133
```

外部参考资料

[1]「HEOI2015」最短不公共子串 

8.19 最小表示法

定义

最小表示法是用于解决字符串最小表示问题的方法.

字符串的最小表示

循环同构

当字符串 S 中可以选定一个位置 i 满足

$$S[i\cdots n]+S[1\cdots i-1]=T$$

则称 S 与 T 循环同构

最小表示

字符串 S 的最小表示为与 S 循环同构的所有字符串中字典序最小的字符串

simple 的暴力

我们每次比较 i 和 j 开始的循环同构，把当前比较到的位置记作 k，每次遇到不一样的字符时便把大的跳过，最后剩下的就是最优解.

实现

- C++

```cpp
int k = 0, i = 0, j = 1;
while (k < n && i < n && j < n) {
  if (sec[(i + k) % n] == sec[(j + k) % n]) {
    ++k;
  } else {
    if (sec[(i + k) % n] > sec[(j + k) % n])
      ++i;
    else
      ++j;
    k = 0;
    if (i == j) i++;
  }
}
i = min(i, j);
```

- Python

```python
k, i, j = 0, 0, 1
while k < n and i < n and j < n:
    if sec[(i + k) % n] == sec[(j + k) % n]:
        k += 1
```

```
    else:
        if sec[(i + k) % n] > sec[(j + k) % n]:
            i += 1
        else:
            j += 1
        k = 0
        if i == j:
            i += 1
i = min(i, j)
```

解释

该实现方法随机数据下表现良好，但是可以构造特殊数据卡掉.

例如：对于 aaa··· aab, 不难发现这个算法的复杂度退化为 $O(n^2)$.

我们发现，当字符串中出现多个连续重复子串时，此算法效率降低，我们考虑优化这个过程.

最小表示法

算法核心

考虑对于一对字符串 A, B, 它们在原字符串 S 中的起始位置分别为 i, j, 且它们的前 k 个字符均相同, 即

$$S[i \cdots i+k-1] = S[j \cdots j+k-1]$$

不妨先考虑 $S[i+k] > S[j+k]$ 的情况，我们发现起始位置下标 l 满足 $i \le l \le i+k$ 的字符串均不能成为答案. 因为对于任意一个字符串 S_{i+p}（表示以 $i+p$ 为起始位置的字符串，$p \in [0, k]$）一定存在字符串 S_{j+p} 比它更优.

所以我们比较时可以跳过下标 $l \in [i, i+k]$, 直接比较 S_{i+k+1}

这样，我们就完成了对于上文暴力的优化.

时间复杂度

$O(n)$

过程

1. 初始化指针 i 为 0，j 为 1；初始化匹配长度 k 为 0
2. 比较第 k 位的大小，根据比较结果跳转相应指针. 若跳转后两个指针相同，则随意选一个加一以保证比较的两个字符串不同
3. 重复上述过程，直到比较结束
4. 答案为 i, j 中较小的一个

实现

- C++

```
int k = 0, i = 0, j = 1;
while (k < n && i < n && j < n) {
  if (sec[(i + k) % n] == sec[(j + k) % n]) {
    k++;
  } else {
    sec[(i + k) % n] > sec[(j + k) % n] ? i = i + k + 1 : j = j + k + 1;
    if (i == j) i++;
    k = 0;
```

```
  }
}
i = min(i, j);
```

- Python

```
k, i, j = 0, 0, 1
while k < n and i < n and j < n:
    if sec[(i + k) % n] == sec[(j + k) % n]:
        k += 1
    else:
        if sec[(i + k) % n] > sec[(j + k) % n]:
            i = i + k + 1
        else:
            j = j + k + 1
        if i == j:
            i += 1
        k = 0
i = min(i, j)
```

8.20 Lyndon 分解

Authors: sshwy, StudyingFather, orzAtalod

定义

首先我们介绍 Lyndon 分解的概念.

Lyndon 串：对于字符串 s，如果 s 的字典序严格小于 s 的所有后缀的字典序，我们称 s 是简单串，或者 **Lyndon 串**. 举一些例子，a,b,ab,aab,abb,ababb,abcd 都是 Lyndon 串. 当且仅当 s 的字典序严格小于它的所有非平凡的（非平凡：非空且不同于自身）循环同构串时，s 才是 Lyndon 串.

Lyndon 分解：串 s 的 Lyndon 分解记为 $s = w_1 w_2 \cdots w_k$，其中所有 w_i 为简单串，并且他们的字典序按照非严格单减排序，即 $w_1 \geq w_2 \geq \cdots \geq w_k$. 可以发现，这样的分解存在且唯一.

Duval 算法

解释

Duval 可以在 $O(n)$ 的时间内求出一个串的 Lyndon 分解.

首先我们介绍另外一个概念：如果一个字符串 t 能够分解为 $t = ww\cdots\overline{w}$ 的形式，其中 w 是一个 Lyndon 串，而 $\overline{w}$ 是 w 的前缀（$\overline{w}$ 可能是空串），那么称 t 是近似简单串（pre-simple），或者近似 Lyndon 串. 一个 Lyndon 串也是近似 Lyndon 串.

Duval 算法运用了贪心的思想. 算法过程中我们把串 s 分成三个部分 $s = s_1 s_2 s_3$, 其中 s_1 是一个 Lyndon 串，它的 Lyndon 分解已经记录；s_2 是一个近似 Lyndon 串；s_3 是未处理的部分.

过程

整体描述一下，该算法每一次尝试将 s_3 的首字符添加到 s_2 的末尾. 如果 s_2 不再是近似 Lyndon 串，那么我们就可以将 s_2 截出一部分前缀（即 Lyndon 分解）接在 s_1 末尾.

我们来更详细地解释一下算法的过程. 定义一个指针 i 指向 s_2 的首字符，则 i 从 1 遍历到 n（字符串长度）. 在循环的过程中我们定义另一个指针 j 指向 s_3 的首字符，指针 k 指向 s_2 中我们当前考虑的字符（意义是 j 在 s_2 的上一个循环节中对应的字符）. 我们的目标是将 $s[j]$ 添加到 s_2 的末尾，这就需要将 $s[j]$ 与 $s[k]$ 做比较：

1. 如果 $s[j] = s[k]$，则将 $s[j]$ 添加到 s_2 末尾不会影响它的近似简单性．于是我们只需要让指针 j, k 自增（移向下一位）即可．
2. 如果 $s[j] > s[k]$，那么 $s_2 s[j]$ 就变成了一个 Lyndon 串，于是我们将指针 j 自增，而让 k 指向 s_2 的首字符，这样 s_2 就变成了一个循环次数为 1 的新 Lyndon 串了．
3. 如果 $s[j] < s[k]$，则 $s_2 s[j]$ 就不是一个近似简单串了，那么我们就要把 s_2 分解出它的一个 Lyndon 子串，这个 Lyndon 子串的长度将是 $j - k$，即它的一个循环节．然后把 s_2 变成分解完以后剩下的部分，继续循环下去（注意，这个情况下我们没有改变指针 j, k），直到循环节被截完．对于剩余部分，我们只需要将进度「回退」到剩余部分的开头即可．

实现

下面的代码返回串 s 的 Lyndon 分解方案．

- C++

```cpp
// duval_algorithm
vector<string> duval(string const& s) {
  int n = s.size(), i = 0;
  vector<string> factorization;
  while (i < n) {
    int j = i + 1, k = i;
    while (j < n && s[k] <= s[j]) {
      if (s[k] < s[j])
        k = i;
      else
        k++;
      j++;
    }
    while (i <= k) {
      factorization.push_back(s.substr(i, j - k));
      i += j - k;
    }
  }
  return factorization;
}
```

- Python

```python
# duval_algorithm
def duval(s):
    n, i = len(s), 0
    factorization = []
    while i < n:
        j, k = i + 1, i
        while j < n and s[k] <= s[j]:
            if s[k] < s[j]:
                k = i
            else:
                k += 1
            j += 1
        while i <= k:
            factorization.append(s[i : i + j - k])
            i += j - k
    return factorization
```

复杂度分析

接下来我们证明一下这个算法的复杂度.

外层的循环次数不超过 n，因为每一次 i 都会增加. 第二个内层循环也是 $O(n)$ 的，因为它只记录 Lyndon 分解的方案. 接下来我们分析一下内层循环. 很容易发现，每一次在外层循环中找到的 Lyndon 串是比我们所比较过的剩余的串要长的，因此剩余的串的长度和要小于 n，于是我们最多在内层循环 $O(n)$ 次. 事实上循环的总次数不超过 $4n-3$，时间复杂度为 $O(n)$.

最小表示法（Finding the smallest cyclic shift）

对于长度为 n 的串 s，我们可以通过上述算法寻找该串的最小表示法.

我们构建串 ss 的 Lyndon 分解，然后寻找这个分解中的一个 Lyndon 串 t，使得它的起点小于 n 且终点大于等于 n. 可以很容易地使用 Lyndon 分解的性质证明，子串 t 的首字符就是 s 的最小表示法的首字符，即我们沿着 t 的开头往后 n 个字符组成的串就是 s 的最小表示法.

于是我们在分解的过程中记录每一次的近似 Lyndon 串的开头即可.

- C++

```cpp
// smallest_cyclic_string
string min_cyclic_string(string s) {
  s += s;
  int n = s.size();
  int i = 0, ans = 0;
  while (i < n / 2) {
    ans = i;
    int j = i + 1, k = i;
    while (j < n && s[k] <= s[j]) {
      if (s[k] < s[j])
        k = i;
      else
        k++;
      j++;
    }
    while (i <= k) i += j - k;
  }
  return s.substr(ans, n / 2);
}
```

- Python

```python
# smallest_cyclic_string
def min_cyclic_string(s):
    s += s
    n = len(s)
    i, ans = 0, 0
    while i < n / 2:
        ans = i
        j, k = i + 1, i
        while j < n and s[k] <= s[j]:
            if s[k] < s[j]:
                k = i
            else:
                k += 1
            j += 1
        while i <= k:
```

```
            i += j - k
    return s[ans : ans + n / 2]
```

习题

- UVa #719 - Glass Beads[1]

本页面主要译自博文 Декомпозиция Линдона. Алгоритм Дюваля. Нахождение наименьшего циклического сдвига[2] 与其英文翻译版 Lyndon factorization[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.

外部参考资料

[1] UVa #719 - Glass Beads

[2] Декомпозиция Линдона. Алгоритм Дюваля. Нахождение наименьшего циклического сдвига

[3] Lyndon factorization

8.21 Main–Lorentz 算法

重串

定义

给定一个长度为 n 的字符串 s.

我们将一个字符串连续写两遍所产生的新字符串称为 **重串 (tandem repetition)**. 下文中，为了表述精准，我们将被重复的这个字符串称为原串. 换言之，一个重串等价于一对下标 (i, j)，其使得 $s[i...j]$ 是两个相同字符串拼接而成的.

你的目标是找出给定的字符串 s 中所有的重串. 或者，解决一个较为简单的问题：找到字符串 s 中任意重串或者最长的一个重串.

下文的算法由 Michael Main 和 Richard J. Lorentz 在 1982 年提出.

约定

下文所有的字符串下标从 0 开始.

下文中，记 $\overline{s}$ 为 s 的反串. 如 $\overline{\texttt{abc}} = \texttt{cba}$.

解释

考虑字符串 `acababaee`，这个字符串包括三个重串，分别是：

- $s[2...5] = \texttt{abab}$
- $s[3...6] = \texttt{baba}$
- $s[7...8] = \texttt{ee}$

下面是另一个例子，考虑字符串 `abaaba`，这个字符串只有两个重串：

- $s[0...5] = \texttt{abaaba}$
- $s[2...3] = \texttt{aa}$

重串的个数

一个长度为 n 的字符串可能有高达 $O(n^2)$ 个重串，一个显然的例子就是 n 个字母全部相同的字符串，这种情况下，只要其子串长度为偶数，这个子串就是重串. 多数情况下，一个周期比较小的周期字符串会有很多重串.

但这并不影响我们在 $O(n \log n)$ 的时间内计算出重串数量. 因为这个算法通过某种压缩形式来表达一个重串，使得我们可以将多个重串压缩为一个.

这里有一些关于重串数量的有趣结论：

- 如果一个重串的原串不是重串，则我们称这个重串为 **本原重串 (primitive repetition)**. 可以证明，本原重串最多有 $O(n \log n)$ 个.
- 如果我们把一个重串用 Crochemore 三元组 (i, p, r) 进行压缩，其中 i 是重串的起始位置，p 是该重串某个循环节的长度（注意不是原串长度!），r 为这个循环节重复的次数. 则某字符串的所有重串可以被 $O(n \log n)$ 个 Crochemore 三元组表示.
- Fibonacci 字符串定义如下：

$$t_0 = a,$$
$$t_1 = b,$$
$$t_i = t_{i-1} + t_{i-2},$$

可以发现 Fibonacci 字符串具有高度的周期性. 对于长度为 f_i 的 Fibonacci 字符串 t_i, 即使用 Crochemore 三元组压缩，也有 $O(f_i \log f_i)$ 个三元组. 其本原重串的数量也有 $O(f_i \log f_i)$ 个.

Main–Lorentz 算法

解释

Main–Lorentz 算法的核心思想是 **分治**.

这个算法将字符串分为左、右两部分，首先计算完全处于字符串左部（或右部）的重串数量，然后计算起始位置在左部，终止在右部的重串数量.（下文中，我们将这种重串称为 **交叉重串**）

计算交叉重串的数量是 Main–Lorentz 算法的关键点，我们将在下文详细探讨.

过程

寻找交叉重串

我们记某字符串的左部为 u，右部为 v. 则 $s = u + v$，且 u, v 的长度大约等于 s 长度的一半.

对于任意一个重串，我们考虑其中间字符. 此处我们将一个重串右半边的第一个字符称为其中间字符，换言之，若 $s[i...j]$ 为重串，则其中间字符为 $s[(i + j + 1)/2]$. 如果一个重串的中间字符在 u 中，则称这个重串 **左偏 (left)**，反之则称其 **右偏 (right)**.

接下来，我们将会探讨如何找到所有的左偏重串.

我们记一个左边重串的长度为 $2l$. 考虑该重串第一个落入 v 的字符（即 $s[|\ u\ |]$），这个字符一定与 u 中的某个字符 $u[cntr]$ 一致.

我们考虑固定 $cntr$，并找到所有符合条件的重串. 举个例子：对于字符串 $\texttt{c}\ \underset{cntr}{\texttt{a}}\ \texttt{c} \mid \texttt{a d a}$（这个 | 是用于分辨左右两部分的），固定 $cntr = 1$，则我们可以发现重串 `caca` 符合要求.

显然，我们一旦固定了 $cntr$，那我们同时也固定了 l 的取值. 我们一旦知道如何找到所有重串，我们就可以从 0 到 $|\ u\ | - 1$ 枚举 $cntr$ 的取值，然后找到所有符合条件的重串.

左偏重串的判定

即使固定 $cntr$ 后，仍然可能会有多个符合条件的重串，我们怎么找到所有符合条件的重串呢?

我们再来举一个例子，对于字符串 `abcabcac` 中的重串 $\overbrace{\texttt{a}}^{l_1}\overbrace{\underset{cntr}{\texttt{b c}}}^{l_2}\overbrace{\texttt{a}}^{l_1} \mid \overbrace{\texttt{b c}}^{l_2}$，我们记 l_1 为该重串的首字符到 $s[cntr - 1]$ 所组成的子串的长度，记 l_2 为 $s[cntr]$ 到该重串左边原串的末字符所组成的子串的长度.

于是，我们可以给出某个长度为 $2l = 2(l_1 + l_2) = 2(|\ u\ | - cntr)$ 的子串是重串的 **充分必要条件**：

记 k_1 为满足 $u[cntr-k_1 \ldots cntr-1]=u[|\ u\ |-k_1 \ldots |\ u\ |-1]$ 的最大整数，记 k_2 为满足 $u[cntr \ldots cntr+k_2-1]=v[0 \ldots k_2-1]$ 的最大整数. 则对于任意满足 $l_1 \le k_1$, $l_2 \le k_2$ 的二元组 (l_1, l_2), 我们都能恰好找到一个与之对应的重串.

总结一下，即有:

- 固定一个 $cntr$.
- 那么我们此时要找的重串长度均为 $2l=2(|\ u\ |-cntr)$. 此时可能仍有多个符合条件的重串，取决于 l_1 与 l_2 的取值.
- 计算上文提到的 k_1, k_2.
- 则所有符合条件的重串符合条件:

$$\begin{aligned} l_1+l_2=l&=|\ u\ |-cntr \\ l_1&\le k_1, \\ l_2&\le k_2. \end{aligned}$$

接下来，只需要考虑如何快速算出 k_1 与 k_2 了. 借助 Z 函数→8.9@p947，我们可以 $O(1)$ 计算它们:

- 计算 k_1: 只需计算 $\overline{u}$ 的 Z 函数即可.
- 计算 k_2: 只需计算 $v+\#+u$ 的 Z 函数即可，其中 $\#$ 是一个 u, v 中均没有的字符.

右偏重串

计算右偏重串的方法与计算左偏重串的方法几乎一致. 考虑该重串第一个落入 u 的字符（即 $s[|\ u\ |-1]$），则其一定与 v 中的某个字符一致，记这个字符在 v 中的位置为 $cntr$.

令 k_1 为满足 $v[cntr-k_1+1 \ldots cntr]=u[|\ u\ |-k_1 \ldots |\ u\ |-1]$ 的最大整数，k_2 为满足 $v[cntr+1 \ldots cntr+k_2]=v[0 \ldots k_2-1]$ 的最大整数. 则我们可以分别通过计算 $\overline{u}+\#+\overline{v}$ 和 v 的 Z 函数来得出 k_1 与 k_2.

枚举 $cntr$，用相仿的方法寻找右偏重串即可.

实现

Main–Lorentz 算法以四元组 $(cntr, l, k_1, k_2)$ 的形式给出所有重串. 如果你只需要计算重串的数量，或者只需要找到最长的一个重串，这个四元组给的信息是足够的. 由 主定理→5.2.1@p544 可得，Main–Lorentz 算法的时间复杂度为 $O(n \log n)$.

请注意, 如果你想通过这些四元组来找到所有重串的起始位置与终止位置, 则最坏时间复杂度会达到 $O(n^2)$. 我们在下面的程序中实现了这一点，将所有重串的起始位置与终止位置存于 `repetitions` 中.

```
vector<int> z_function(string const& s) {
  int n = s.size();
  vector<int> z(n);
  for (int i = 1, l = 0, r = 0; i < n; i++) {
    if (i <= r) z[i] = min(r - i + 1, z[i - l]);
    while (i + z[i] < n && s[z[i]] == s[i + z[i]]) z[i]++;
    if (i + z[i] - 1 > r) {
      l = i;
      r = i + z[i] - 1;
    }
  }
  return z;
}

int get_z(vector<int> const& z, int i) {
  if (0 <= i && i < (int)z.size())
    return z[i];
```

```
  else
    return 0;
}

vector<pair<int, int>> repetitions;

void convert_to_repetitions(int shift, bool left, int cntr, int l, int k1,
                            int k2) {
  for (int l1 = max(1, l - k2); l1 <= min(l, k1); l1++) {
    if (left && l1 == l) break;
    int l2 = l - l1;
    int pos = shift + (left ? cntr - l1 : cntr - l - l1 + 1);
    repetitions.emplace_back(pos, pos + 2 * l - 1);
  }
}

void find_repetitions(string s, int shift = 0) {
  int n = s.size();
  if (n == 1) return;

  int nu = n / 2;
  int nv = n - nu;
  string u = s.substr(0, nu);
  string v = s.substr(nu);
  string ru(u.rbegin(), u.rend());
  string rv(v.rbegin(), v.rend());

  find_repetitions(u, shift);
  find_repetitions(v, shift + nu);

  vector<int> z1 = z_function(ru);
  vector<int> z2 = z_function(v + '#' + u);
  vector<int> z3 = z_function(ru + '#' + rv);
  vector<int> z4 = z_function(v);

  for (int cntr = 0; cntr < n; cntr++) {
    int l, k1, k2;
    if (cntr < nu) {
      l = nu - cntr;
      k1 = get_z(z1, nu - cntr);
      k2 = get_z(z2, nv + 1 + cntr);
    } else {
      l = cntr - nu + 1;
      k1 = get_z(z3, nu + 1 + nv - 1 - (cntr - nu));
      k2 = get_z(z4, (cntr - nu) + 1);
    }
    if (k1 + k2 >= l) convert_to_repetitions(shift, cntr < nu, cntr, l, k1, k2);
  }
}
```

本页面主要译自博文 Поиск всех тандемных повторов в строке. Алгоритм Мейна-Лоренца[1] 与其英文翻译版 Finding repetitions[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] Поиск всех тандемных повторов в строке. Алгоритм Мейна-Лоренца 

[2] Finding repetitions 

第九章

数学

9.1 数学部分简介

本章介绍 OI 中可能会用到的数学知识．计算机科学与数学紧密相关，而在算法竞赛中尤其强调以数论、排列组合、概率期望、多项式为代表离散、具体的数学：其注重程序实现和现实问题，可以出现在几乎任何类别的题目中．

实际上，算法竞赛中涉及到的算法和数据结构以及自动机等也可以被认为属于数学范畴，但是这些内容被细分到诸如字符串等的具体章节加以应用背景以更好理解．本章主要介绍数学中一些基础概念、代数、数论、博弈论及概率论等知识．运用这些知识可以帮助优化其他算法和数据结构，例如：

- 用多项式优化卷积形式的背包，来做一些字符串题．
- 很多递推类型的题背景都是排列组合或概率期望，可以更进一步用生成函数推导和解决，以及使用基于 FFT 的分治优化算法效率．
- 利用同余和环分析图上非简单路径在模意义下可能的权值和，并用带权并查集维护．

此外，高中数学是信息学竞赛数学的基础，学好课本上的基本概念和性质能更好地帮助学习本章内容．

9.2 布尔代数

在数理逻辑中，布尔代数（boolean algebra）是代数的一个分支．初等代数中变量的值是数字，其研究的主要运算符有加法、乘法、乘方以及这三种运算的逆运算．而布尔代数中变量的值仅为 **真** 和 **假** 两种（通常记作 1 和 0），其研究的主要运算符有合取（与，$\wedge$）、析取（或，$\vee$）、否定（非，$\neg$）．就像初等代数是描述数字运算的一种形式一样，布尔代数是描述逻辑运算的一种形式．

布尔函数

自 定义

布尔函数（boolean function）指的是形如 $f : \mathbf{B}^k \to \mathbf{B}$ 的函数，其中 $\mathbf{B} = \{0, 1\}$ 为 **布尔域**（boolean domain），非负整数 k 为该布尔函数的 **元数**（arity）．$k = 1$ 的布尔函数为一元函数，以此类推．$k = 0$ 时，我们认为函数退化为 $\mathbf{B}$ 中的常量．

我们一般只研究一元和二元的布尔函数．如无特殊说明，下文的布尔函数仅限于一元和二元的情况．

除了函数的一般表达方式外，我们还可以用 **真值表**（truth table）、**逻辑门**（logic gate）、Venn 图[1] 来表示布尔函数．

自 真值表

对一个布尔函数，我们枚举其输入的所有情况，并将输入和对应的输出列成一张表，这个表就叫做真值表．

n 元布尔函数也可以用含 n 个变量的 **命题公式**（propositional formula）表示，命题公式 p 与 q **逻辑等价**（logically equivalent）当且仅当其描述的是同一个布尔函数，记作 $p \iff q$．

以下是一些常见布尔函数，我们也会把这些布尔函数统称为 **逻辑运算符**（logical connective）或 **逻辑算子**（logical operator）：

名称（数理逻辑）	其他名称	记号
恒真（truth、tautology）		$\top$
恒假（falsity、contradiction）		$\bot$
命题	自身	A
否定（negation）	非（NOT）	$\neg A$

合取（conjunction）	与（AND）	$A \wedge B$
析取（disjunction）	或（OR）	$A \vee B$
非合取（non-conjunction）	与非（NAND）、Sheffer 竖线	$A \barwedge B$、$A \uparrow B$
非析取（non-disjunction）	或非（NOR）	$A \barvee B$、$A \downarrow B$
	异或（Exclusive-OR，XOR）	$A \oplus B$
	同或（Exclusive-NOR）	$A \odot B$
实质蕴含（material implication）[1]		$A \to B$
实质非蕴含（material nonimplication）[2]		$A \nrightarrow B$
反蕴涵（converse implication）[3]		$A \leftarrow B$
非反蕴涵（converse nonimplication）[4]		$A \nleftarrow B$
双条件（biconditional）、等价（equivalence）[5][6]		$A \leftrightarrow B$
非等价（non-equivalence）[7][8]		$A \nleftrightarrow B$

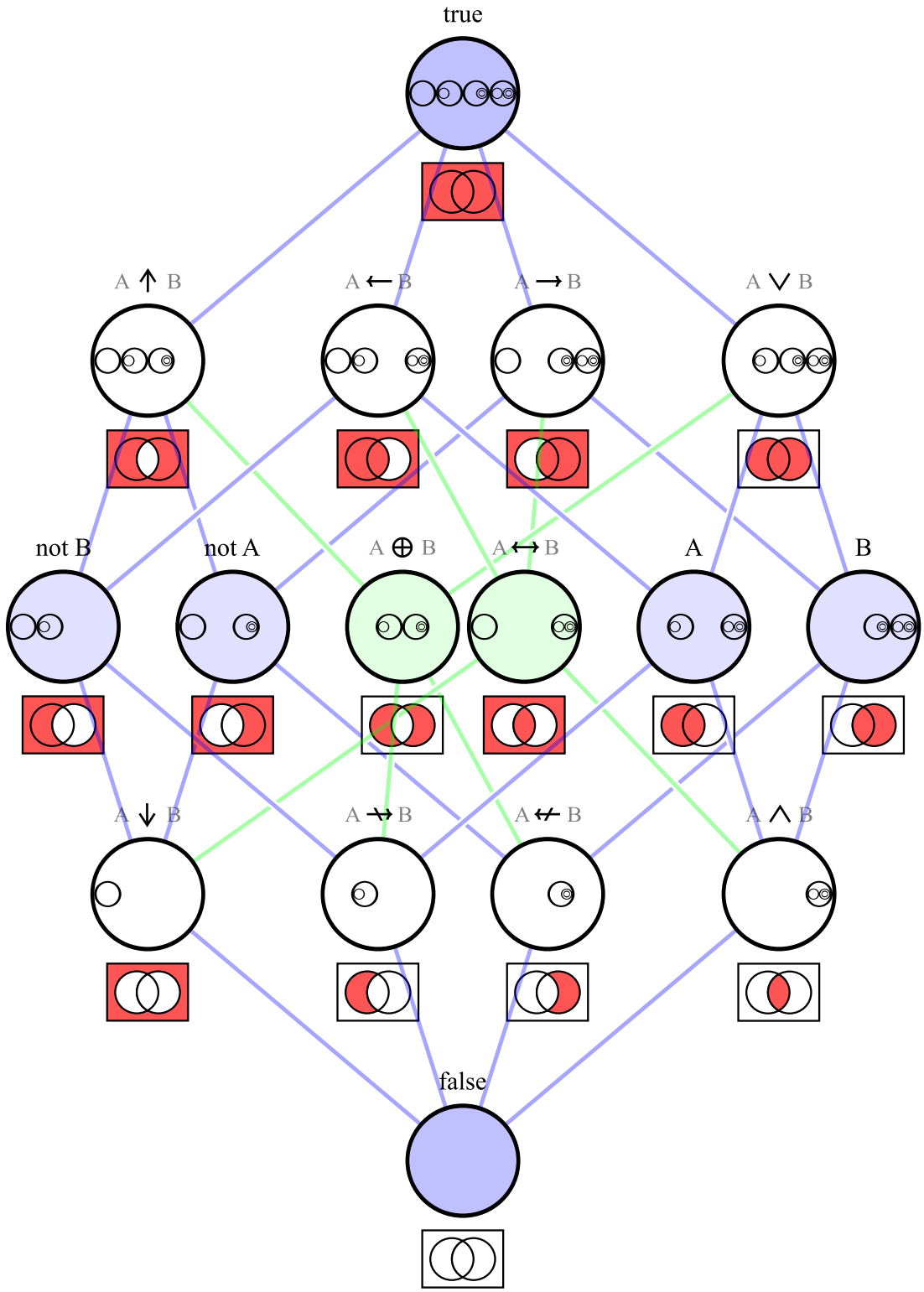
对应的真值表（From Wikipedia[2]）：

[1]用于命题推导时应使用双横长箭头，如 $A \Longrightarrow B$、$A \Longleftarrow B$、$A \Longleftrightarrow B$ 等.
[2]用于命题推导时应使用双横长箭头，如 $A \Longrightarrow B$、$A \Longleftarrow B$、$A \Longleftrightarrow B$ 等.
[3]用于命题推导时应使用双横长箭头，如 $A \Longrightarrow B$、$A \Longleftarrow B$、$A \Longleftrightarrow B$ 等.
[4]用于命题推导时应使用双横长箭头，如 $A \Longrightarrow B$、$A \Longleftarrow B$、$A \Longleftrightarrow B$ 等.
[5]用于命题推导时应使用双横长箭头，如 $A \Longrightarrow B$、$A \Longleftarrow B$、$A \Longleftrightarrow B$ 等.
[6]等价于同或.
[7]用于命题推导时应使用双横长箭头，如 $A \Longrightarrow B$、$A \Longleftarrow B$、$A \Longleftrightarrow B$ 等.
[8]等价于异或.

A	0	1	0	1
B	0	0	1	1

false	0	0	0	0
$A \wedge B \Leftrightarrow \overline{A} \not\leftarrow B \Leftrightarrow A \not\rightarrow \overline{B} \Leftrightarrow \overline{A} \downarrow \overline{B}$	0	0	0	1
$A \not\leftarrow B \Leftrightarrow \overline{A} \wedge B \Leftrightarrow A \downarrow \overline{B} \Leftrightarrow \overline{A} \not\rightarrow \overline{B}$	0	0	1	0
B	0	0	1	1
$A \not\rightarrow B \Leftrightarrow \overline{A} \downarrow B \Leftrightarrow A \wedge \overline{B} \Leftrightarrow \overline{A} \not\leftarrow \overline{B}$	0	1	0	0
A	0	1	0	1
$A \oplus B \Leftrightarrow \overline{A} \leftrightarrow B \Leftrightarrow A \leftrightarrow \overline{B} \Leftrightarrow \overline{A} \oplus \overline{B}$	0	1	1	0
$A \vee B \Leftrightarrow \overline{A} \rightarrow B \Leftrightarrow A \leftarrow \overline{B} \Leftrightarrow \overline{A} \uparrow \overline{B}$	0	1	1	1
$A \downarrow B \Leftrightarrow \overline{A} \not\rightarrow B \Leftrightarrow A \not\leftarrow \overline{B} \Leftrightarrow \overline{A} \wedge \overline{B}$	1	0	0	0
$A \leftrightarrow B \Leftrightarrow \overline{A} \oplus B \Leftrightarrow A \oplus \overline{B} \Leftrightarrow \overline{A} \leftrightarrow \overline{B}$	1	0	0	1
$\overline{A}$	1	0	1	0
$A \rightarrow B \Leftrightarrow \overline{A} \vee B \Leftrightarrow A \uparrow \overline{B} \Leftrightarrow \overline{A} \leftarrow \overline{B}$	1	0	1	1
$\overline{B}$	1	1	0	0
$A \leftarrow B \Leftrightarrow \overline{A} \uparrow B \Leftrightarrow A \vee \overline{B} \Leftrightarrow \overline{A} \rightarrow \overline{B}$	1	1	0	1
$A \uparrow B \Leftrightarrow \overline{A} \leftarrow B \Leftrightarrow A \rightarrow \overline{B} \Leftrightarrow \overline{A} \vee \overline{B}$	1	1	1	0
true	1	1	1	1

对应的 Venn 图和 Hasse 图→9.20@p1951（以集合的包含关系 $\subseteq$ 为偏序，From Wikipedia[3]）：



由于 n 元布尔函数的输入有 2^n 种，所以 n 元布尔函数有 $2 \uparrow (2 \uparrow n)$ 种，其中 $\uparrow$ 为 Knuth 箭头.

我们把逻辑算子的组合称为 **逻辑表达式**（logical expression）.

如果我们把 $\mathbf{B}$ 视作模 2 的一个 剩余类→9.11.1@p1160，此时异或等价于模 2 加法，与等价于模 2 乘法，所以有时我们也用 $\mathbf{Z}_2$ 表示布尔域.

优先级

一元逻辑算子优先级高于二元逻辑算子，即 $\neg$ 的优先级高于 $\wedge$、$\vee$、$\oplus$ 等的优先级.

二元逻辑算子之间的优先级有多种规定，有的资料认为 $\wedge$、$\vee$、$\oplus$ 的优先级比 $\to$、$\leftarrow$、$\leftrightarrow$ 更高，而有的资料持相反观点. 所以在使用时推荐多加括号来明确顺序.

C++ 中的规定参见 C++ 运算符优先级总表→4.2.4@p339.

自足算子与完备算子集

实际上，我们只用与非或者或非即可表达其余的逻辑算子，CPU 也是基于这一点构建的. 但是，由于**与、或、非、异或**这四种逻辑算子的性质更好，所以我们在研究布尔代数时一般只使用这四种函数.

如何分别用与非、或非表示其余的逻辑算子

我们有

- $\neg p = p \barwedge p = p \barvee p$,
- $p \wedge q = (p \barwedge q) \barwedge (p \barwedge q) = (p \barvee p) \barvee (q \barvee q)$,
- $p \vee q = (p \barwedge p) \barwedge (q \barwedge q) = (p \barvee q) \barvee (p \barvee q)$,
- $p \to q = p \barwedge (q \barwedge q) = ((p \barvee p) \barvee q) \barvee ((p \barvee p) \barvee q)$.

另外

- $p = \neg\neg p$,
- $p \not\leftrightarrow q = p \oplus q = (p \vee q) \wedge \neg(p \wedge q)$,
- $p \leftrightarrow q = p \odot q = \neg(p \oplus q)$,
- $p \not\to q = \neg(p \to q)$,
- $p \leftarrow q = q \to p$,
- $p \not\leftarrow q = \neg(p \leftarrow q)$.

我们能不能用指定的若干逻辑算子描述所有的逻辑算子？这便引出了完备算子集的定义.

定义

对一个给定的逻辑算子集，如果能只用这个集合里的函数描述所有的逻辑算子，则称该集合为**完备算子集**（functionally complete operator set）. 特别地，如果只用一个逻辑算子即可描述所有的逻辑算子，则称该算子为**自足算子**（sole sufficient operator）或 **Sheffer 函数**（Sheffer function）.

如果在一个完备算子集中删去任意一个元素，其都不能描述所有的逻辑算子，则称该集合为**极小完备算子集**（minimal functionally complete operator set）.

可以证明逻辑算子中只有 $\barwedge$、$\barvee$ 是自足算子.

以下为常见的极小完备算子集[9]:

- $\{\barwedge\}$, $\{\barvee\}$,
- $\{\wedge, \neg\}$, $\{\vee, \neg\}$, $\{\leftarrow, \neg\}$, $\{\to, \neg\}$, $\{\not\leftarrow, \neg\}$, $\{\not\to, \neg\}$,
- $\{\leftarrow, \bot\}$, $\{\to, \bot\}$, $\{\not\leftarrow, \top\}$, $\{\not\to, \top\}$,
- $\{\leftarrow, \not\leftarrow\}$, $\{\to, \not\leftarrow\}$, $\{\leftarrow, \not\to\}$, $\{\to, \not\to\}$,
- $\{\leftarrow, \not\leftrightarrow\}$, $\{\to, \not\leftrightarrow\}$, $\{\not\leftarrow, \leftrightarrow\}$, $\{\not\to, \leftrightarrow\}$,
- $\{\vee, \leftrightarrow, \bot\}$, $\{\vee, \leftrightarrow, \not\leftrightarrow\}$, $\{\vee, \not\leftrightarrow, \top\}$,
- $\{\wedge, \leftrightarrow, \bot\}$, $\{\wedge, \leftrightarrow, \not\leftrightarrow\}$, $\{\wedge, \not\leftrightarrow, \top\}$.

[9] Vaughan, H. E. (1942). Complete sets of logical functions. *Transactions of the American Mathematical Society 51*: 117–32.

性质

首先是代数结构的相关性质：

- 与、或均关于 $\mathbf{B}$ 构成 交换幺半群→9.16.1@p1805. 即与运算和或运算均具有交换律、结合律和幺元（$x \wedge 1 = x \vee 0 = x$）.
- 异或、同或均关于 $\mathbf{B}$ 构成 群→9.16.1@p1805. 即异或运算和同或运算均具有交换律、结合律、幺元（$x \oplus 0 = x \odot 1 = x$）和逆元（$x \oplus x = 0$，$x \odot x = 1$）.
- 与非、或非均不具有结合律，所以不构成半群.

对于 $\wedge$、$\vee$，我们有

- 分配律：
  - $a \wedge (b \diamond c) = (a \wedge b) \diamond (a \wedge c)$，其中 $\diamond$ 可以为 $\wedge$、$\vee$、$\oplus$，
  - $a \vee (b \diamond c) = (a \vee b) \diamond (a \vee c)$，其中 $\diamond$ 可以为 $\wedge$、$\vee$、$\odot$.
- **幂等**（idempotence）律：$x \wedge x = x$、$x \vee x = x$.
- 单调性：$a \to b \iff (a \wedge c) \to (b \wedge c)$、$a \to b \iff (a \vee c) \to (b \vee c)$.
- **吸收**（absorption）律：$x \wedge (x \vee y) = x \vee (x \wedge y) = x$.
- 与「$\to$」的关系：
  - $a \vee b \iff (\neg a \to b) \wedge (\neg b \to a)$，
  - $a \wedge b \iff \neg((a \to \neg b) \vee (b \to \neg a))$.

自 布尔函数的单调性

对一个布尔函数 $f(x_1, ..., x_n)$ 和 $\mathbf{B}^n$ 中的两个元素 $(a_1, ..., a_n), (b_1, ..., b_n)$，若当 $a_i \le b_i$，$\forall i = 1, ..., n$ 时恒有 $f(a_1, ..., a_n) \le f(b_1, ..., b_n)$，则称该布尔函数是单调的.

我们还有如下性质：

- **排中律**（law of excluded middle）：$p \vee \neg p$ 恒真.
- $\neg p \iff p \to \bot$.
- 双重否定/$\neg$ 的 **对合**（involution）律：$\neg\neg x = x$.
- $\oplus$、$\odot$ 的对合律：$x \oplus y \oplus y = x$、$x \odot y \odot y = x$.
- De Morgan 律：$\neg(p \wedge q) = \neg p \vee \neg q$、$\neg(p \vee q) = \neg p \wedge \neg q$.

逻辑表达式的标准化

根据上述性质，我们可以对逻辑表达式进行一定的等价变换，使其符合特定的范式，这一点可用于自动定理证明中. 常见的标准化范式有 **合取范式**（conjunctive normal form，CNF）、**析取范式**（disjunctive normal form，DNF）和 **代数范式**（algebraic normal form，ANF）.

自 合取范式与析取范式

我们做如下递归式的定义：

1. **文字**（literal）：对变量 x，x 和 $\neg x$ 是文字.
2. 子式：
   - 文字是子式，
   - 若 A 是文字、B 是子式，则 $A \vee B$ 是子式.
3. 合取范式：
   - 若 A 是子式，则 (A) 是合取范式，
   - 若 A 是子式、B 是合取范式，则 $(A) \wedge B$ 是合取范式.

类似地，交换上面定义中的 $\wedge$ 与 $\vee$ 即可得到析取范式的定义.

例如以下逻辑表达式均为析取范式：

- $(A \wedge \neg B) \vee (C \wedge D \wedge \neg E)$,
- $(A \wedge B) \vee (C)$,
- $(A \wedge B)$,
- (A).

以下逻辑表达式均为合取范式：

- $(\neg A \vee \neg B \vee C) \wedge (\vee D \vee \neg E)$,
- $(A \vee B) \wedge (C)$,
- $(A \vee B)$,
- (A).

以下逻辑表达式既不为合取范式也不为析取范式：

- $\neg(A \wedge B)$,
- $A \wedge (B \vee (C \wedge D))$.

我们可以通过如下的步骤将任意一个只含有 $\neg$、$\wedge$、$\vee$ 运算的逻辑表达式变形为 DNF：

$$\begin{aligned} \neg\neg x &\mapsto x, \\ \neg(x \vee y) &\mapsto \neg x \wedge \neg y, \\ \neg(x \wedge y) &\mapsto \neg x \vee \neg y, \\ x \wedge (y \vee z) &\mapsto (x \wedge y) \vee (x \wedge z), \\ (x \vee y) \wedge z &\mapsto (x \wedge z) \vee (y \wedge z). \end{aligned}$$

要得到表达式 X 的 CNF，只需得到 $\neg X$ 的 DNF 后取反并应用 De Morgan 律即可.

自 代数范式

首先，我们用如下递归式的定义来定义子式：

- 变量 x 是子式,
- 若 A 是子式，x 是变量，则 $x \wedge A$ 是子式.

则满足如下三种形式之一的逻辑表达式为代数范式：

1. 1、0,
2. 若干不等价子式的异或，如 $a \oplus b \oplus (a \wedge b) \oplus (a \wedge b \wedge c)$,
3. 若干不等价子式与唯一的 1 的异或，如 $1 \oplus a \oplus b \oplus (a \wedge b) \oplus (a \wedge b \wedge c)$.

注意到代数范式和 $\mathbf{Z}_2$ 上的多项式一一对应，所以代数范式也被称为 **Zhegalkin 多项式**（Zhegalkin polynomial）.

我们可以通过如下的步骤将任意一个只含有 $\neg$、$\wedge$、$\vee$、$\oplus$ 运算的逻辑表达式变形为 ANF：

1. $\oplus$：直接展开，如 $(1 \oplus x) \oplus (1 \oplus x \oplus y) = 1 \oplus x \oplus 1 \oplus x \oplus y = y$,
2. $\wedge$：用分配律展开，如 $x \wedge (1 \oplus x \oplus y) = (x \wedge 1) \oplus (x \wedge x) \oplus (x \wedge y) = x \oplus (x \wedge y)$,
3. $\neg$：将 $\neg x$ 用 $1 \oplus x$ 代替，如 $\neg(1 \oplus x \oplus y) = 1 \oplus 1 \oplus x \oplus y = x \oplus y$,
4. $\vee$：将 $x \vee y$ 用 $1 \oplus ((1 \oplus x) \wedge (1 \oplus y))$ 或 $x \oplus y \oplus (x \wedge y)$ 代替，如 $(1 \oplus x) \vee (1 \oplus x \oplus y) = 1 \oplus ((1 \oplus 1 \oplus x) \wedge (1 \oplus 1 \oplus x \oplus y)) = 1 \oplus x \oplus (x \wedge y)$.

参考资料与注释

1. Boolean algebra - Wikipedia[4]
2. Boolean function - Wikipedia[5]
3. Logical connective - Wikipedia[6]
4. Disjunctive normal form - Wikipedia[7]
5. Zhegalkin polynomial - Wikipedia[8]

[1] Venn 图 [2] Wikipedia [3] Wikipedia

[4] Boolean algebra - Wikipedia [5] Boolean function - Wikipedia [6] Logical connective - Wikipedia

[7] Disjunctive normal form - Wikipedia [8] Zhegalkin polynomial - Wikipedia

9.3 数字系统

9.3.1 数字系统简介

> ⚠ **Warning**
>
> 此处讨论的不是 number system[1]，数学中的数字系统（number system）指的是一类集合，如整数集 $\mathbf{Z}$、复数集 $\mathbf{C}$ 等.

数字系统（numeral system），又称 **记数系统**，指的是用以表示数字的书写系统，如 印度–阿拉伯数字系统[2]、罗马数字[3]、苏州码子[4] 等. 数字系统是我们给数做编码的工具.

一般来说，一个数字系统下的数字都是一串符号，同时有一套规则将这串符号和对应的数一一对应起来，例如罗马数字 XLII、二进制数 $101010_{(2)}$ 和十进制数 42 均能对应到相同的数.

算法竞赛中，我们一般只关注不同数字系统间的转化关系，详见 进位制→9.3.2@p1087.

参考资料与注释

- Numeral system - Wikipedia[5]

外部参考资料

[1] number system [2] 印度–阿拉伯数字系统 [3] 罗马数字

[4] 苏州码子 [5] Numeral system - Wikipedia

9.3.2 进位制

> **Authors:** c-forrest, Enter-tainer, hhc0001, Ir1d, KingMario, ksyx, Lutra-Fs, MegaOwIer, niujiaxing, StudyingFather, Tiphereth-A, TOMWT-qwq, ZnPdCo

进位制，又称 **进位系统**（carry system）、**进制系统**、**位置记法**（positional notation）、**位值记数法**（place-value notation）、**位置数值系统**（positional numeral system），是一种能用有限种符号表示所有自然数的数字系统. 一种进位制可以使用的符号数目称为 **基数**（radix）或 **底数**（base），基数为 n 的进位制称为 n 进制（$n>1$），例如我们最常用的十进制，通常只使用 `0, 1, 2, 3, 4, 5, 6, 7, 8, 9` 这十个符号来记数. 进位指的是「当一个数字的某一位达到基数时，将其置为 0 并使高一位的数加一」的操作.

一般地，我们常将一个 n 进制的数记作 $(a_k\cdots a_1a_0)_n$、$(a_k\cdots a_1a_0)_{(n)}$、$a_k\cdots a_1a_{(0)_{(n)}}$、$a_k\cdots a_1a_{(0)_n}$ 等. 若基数隐含在上下文中我们也可以省略下标. 注意此处的 $a_k\cdots a_1a_0$ 并不是 $k+1$ 个数的乘积，而是一串序列.

对于 k 进制数 $a_n\cdots a_1a_0$，其表示的值为 $a_nk^n+\cdots+a_1k^1+a_0k^0=\sum_{i=0}^{n}a_ik^i$；对一个数 m，设其 k 进制下的表示为 $a_n\cdots a_1a_0$，则有：

$$\begin{aligned} a_0&=m-q_0k, & q_0&=f(m/k),\\ a_1&=q_0-q_1k, & q_1&=f(q_0/k),\\ &\vdots & &\vdots\\ a_n&=q_{n-1}-q_nk, & q_n&=f(q_{n-1}/k)=0, \end{aligned}$$

其中 $f(x)=\lfloor x\rfloor$.

数 n 在 k 进制下表示的长度为 $\lceil \log_k(n+1) \rceil$.

我们一般通过添加小数点「.」来表示小数[1]、添加负号「−」来表示负数、在小数末尾的一段上添加上划线表示无限循环小数等. 为了方便阅读，我们可以每隔若干数位添加间隔符号（如空格、,、`'` 等），如 12 345 表示 12345.

在计算机中，比较常用的进位制有二进制、八进制和十六进制.

不同进位制间的转换

十进制转其他进制

这里以二进制为例来演示，其他进制的原理与其类似.

整数部分，把十进制数不断执行除 2 操作，直至商数为 0，之后从下到上，取所有余数的数字，即为二进制的整数部分数字. 小数部分，则用其乘 2，取其整数部分的结果，再用计算后的小数部分依此重复计算，算到小数部分全为 0 为止，之后从上到下，取所有计算后整数部分的数字，即为二进制的小数部分数字.

例子

将 35.25 转化为二进制数.

整数部分：

$$
\begin{aligned}
35/2 &= 17 \ldots 1, \\
17/2 &= 8 \ \ldots 1, \\
8/2 &= 4 \ \ldots 0, \\
4/2 &= 2 \ \ldots 0, \\
2/2 &= 1 \ \ldots 0, \\
1/2 &= 0 \ \ldots 1.
\end{aligned}
$$

小数部分：

$$
\begin{aligned}
0.25 \times 2 &= 0.5 \ldots 0, \\
0.5 \times 2 &= 1 \ \ \ldots 1.
\end{aligned}
$$

即 $35.25 = (100011.01)_2$.

实现

```cpp
#include <algorithm>
#include <string>

// only non-negative
// 2 <= base && base <= 36
std::string from_dec(int x, int base) {
  if (x == 0) return "0";
  std::string res;
  while (x) {
    int r = x % base;
    x /= base;

    // 写法 1
```

[1] 有的地区使用「,」表示小数点.

```
    res.push_back(r < 10 ? '0' + r : 'A' + r - 10);
    // 写法 2
    // const static std::string digits =
"0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ";
    // res.push_back(digits[r]);
  }
  std::reverse(res.begin(), res.end());
  return res;
}

```

其他进制转十进制

还是以二进制为例. 二进制数转换为十进制数, 只需将每个位的值, 乘以 2^i 次即可, 其中 i 为当前位的位数, 个位的位数为 0.

例子

将 $(11010.01)_2$ 转换为十进制数.

$$\begin{aligned}(11010.01)_2 = & \quad 1\times 2^4+1\times 2^3+0\times 2^2+1\times 2^1+0\times 2^0 \\ & +0\times 2^{-1}+1\times 2^{-2} \\ = & \ 26.25.\end{aligned}$$

即 $(11010.01)_2=(26.25)_{10}$.

实现

```
#include <cctype>
#include <string>

// only non-negative
// 2 <= base && base <= 36
int to_dec(std::string const& s, int base) {
  int res = 0;
  for (char c : s) {
    res *= base;
    if (isdigit(c))
      res += c - '0';
    else if (isupper(c))
      res += c - 'A' + 10;
    else if (islower(c))
      res += c - 'a' + 10;
  }
  return res;
}

```

二进制/八进制/十六进制间的相互转换

一个八进制位可以用 3 个二进制位来表示（因为 $2^3=8$），一个十六进制位可以用 4 个二进制位来表示（$2^4=16$），反之同理.

补数法

另请参阅：反码与补码→9.4@p1099

补数法（method of complements）是用正数表示负数，使得可以用和正数加法相同的算法/电路/机械结构来计算减法的方法．补数法广泛用于计算器和计算机的设计中，用以简化结构．

对 b 进制下 n 位数 a，其 **数基补数**（radix complement，称为 b 的补数）是 b^n-a，其 **缩减数基补数**（diminished radix complement，称为 $b-1$ 的补数，简称 **减补数**）是 b^n-1-a．在二进制下，数基补数叫做 2 的补数（two's complement），又叫做 **补码**；缩减数基补数叫做 1 的补数（ones' complement），又叫做 **反码**；在十进制下，数基补数又叫做 10 的补数（ten's complement），缩减数基补数又叫做 9 的补数（nine's complement）；其他进制以此类推．

对 b 进制下的 n 位数 x,y，当计算 $x-y$ 时，我们有如下方法（若结果超过 n 位，则舍弃高位）：

1. 考虑 x 的减补数 $x'=b^n-1-x$，计算 $x'+y=b^n-1-x+y$，则该结果的减补数即为答案．
2. 考虑 y 的减补数 $y'=b^n-1-y$，计算 $x+y'=b^n-1+x-y$，直接加一即为答案．
3. 考虑 x 的数基补数 $x'=b^n-x$，计算 $x'+y=b^n-x+y$，则该结果的数基补数即为答案．
4. 考虑 y 的数基补数 $y'=b^n-y$，计算 $x+y'=b^n+x-y$，此即为答案．

另外，对 k 进制下的数，我们设 $d=k-1$，则 $\cdots dd=:\overline{d}=\sum_{i=0}^{\infty}dk^i=-1$，所以对 n 位数 x，设其数基补数在 k 进制下的表示为 $a_{n-1}\cdots a_1a_0$，则 $\overline{d}a_{n-1}\cdots a_1a_0$ 即为 $\sum_{i=0}^{n-1}a_ik^i+\sum_{i=n}^{\infty}dk^i=k^n-x+(-k^n)=-x$．这种「无限位的数」的思想可以推广为 p **进数**[1]（p-adic number）．

此外，我们有一个关于补数和无限循环小数的有趣定理：

Midy 定理

设 a 是正整数，p 是正素数，a/p 在 b 进制下的表示为 $0.\overline{a_1a_2\cdots a_l}$，其中 l 为（最短）循环节长度．若 l 是偶数[2]，设 $l=2k$，则 $a_1a_2\cdots a_k$ 是 $a_{k+1}a_{k+2}\cdots a_{2k}$ 的减补数，即：

- $a_i+a_{i+k}=b$，
- $a_1a_2\cdots a_k+a_{k+1}a_{k+2}\cdots a_{2k}=b^k-1$．

进一步，若 l 有非平凡因子 k，设 $l=nk$，则 $\sum_{i=0}^{n-1}a_{ik+1}a_{ik+2}\cdots a_{(i+1)k}$ 是 b^k-1 的倍数．

例子

对于 $1/19=0.\overline{052\ 631\ 578\ 947\ 368\ 421}=0.\overline{032\ 745}_{(8)}$，我们有：

- $052\ 631\ 578+947\ 368\ 421=999\ 999\ 999$，
- $052+631+578+947+368+421=3\times 999$，
- $032_{(8)}+745_{(8)}=777_{(8)}$，
- $03_{(8)}+27_{(8)}+45_{(8)}=77_{(8)}$．

证明

对定理中的 a,b,p,l,n,k，不难发现 $1\le a<p$，$b>1$ 且 $(a,p)=(b,p)=1$．

对整数 $0\le i<l$，令 $f(i)=b^i\cdot a/p-\lfloor b^i\cdot a/p\rfloor$，我们有

$$0<f(i)=0.\overline{a_{i+1}a_{i+2}\cdots a_{nk}a_1a_2\cdots a_i}<1\Longrightarrow 0<pf(i)<p.$$

注意到 $pf(i)\in\mathbf{N}_+$ 且 $pf(i)\equiv ab^i\pmod p$，因此 $pf(i)=ab^i \bmod p$．

令 $S_n=\sum_{i=0}^{n-1}f(ik)=\sum_{i=0}^{n-1}0.\overline{a_{ik+1}a_{ik+2}\cdots a_{nk}a_1a_2\cdots a_{ik}}$，我们可以在各个小数间「交换」若干位（例如 $0.\overline{142857}+0.\overline{285714}+0.\overline{571428}=0.\overline{141414}+0.\overline{282828}+0.\overline{575757}=0.\overline{14}+0.\overline{28}+0.\overline{57}=14/99+28/99+57/99=1$），则

[2] 当 $a=1$ 时，在十进制下满足该条件的素数序列是 A028416[2]．

$$\begin{aligned} S_n &= \sum_{i=0}^{n-1} 0.\overline{a_{ik+1}a_{ik+2}\cdots a_{(i+1)k}} \\ &= \sum_{i=0}^{n-1} a_{ik+1}a_{ik+2}\cdots a_{(i+1)k}/\left(b^k-1\right), \end{aligned}$$

进而

$$pS_n = p\sum_{i=0}^{n-1} a_{ik+1}a_{ik+2}\cdots a_{(i+1)k}/\left(b^k-1\right) = \sum_{i=0}^{n-1}\left(ab^{ik} \bmod p\right),$$

因此

$$\sum_{i=0}^{n-1} a_{ik+1}a_{ik+2}\cdots a_{(i+1)k} = \left(b^k-1\right)\frac{\sum_{i=0}^{n-1}\left(ab^{ik} \bmod p\right)}{p}.$$

若 $p \mid \left(b^k-1\right)$，注意到

$$\left(b^k-1\right)a/p = a_1a_2\cdots a_k.\overline{a_{k+1}a_{k+2}\cdots a_{nk}a_1a_2\cdots a_k} - 0.\overline{a_1a_2\cdots a_{nk}},$$

则有 $a_{k+1}a_{k+2}\cdots a_{nk}a_1a_2\cdots a_k = a_1a_2\cdots a_{nk}$，进而 $a_1a_2\cdots a_k = a_{k+1}a_{k+2}\cdots a_{2k} = \ldots = a_{(n-1)k+1}a_{(n-1)k+2}\cdots a_{nk}$，即 $0.\overline{a_1a_2\cdots a_l} = 0.\overline{a_1a_2\cdots a_k}$，这与 l 的定义矛盾，因此 $p \nmid \left(b^k-1\right)$.

故存在正整数 $c = \dfrac{\sum_{i=0}^{n-1}\left(ab^{ik} \bmod p\right)}{p}$，使得

$$\sum_{i=0}^{n-1} a_{ik+1}a_{ik+2}\cdots a_{(i+1)k} = c\left(b^k-1\right).$$

推论

对上述的 b, n, k, p，有

$$\sum_{i=0}^{n-1} b^{ik} \equiv 0 \pmod{p}.$$

广义进制系统

标准的进制系统中，基数 b 总是一个固定的正数，每个数位在 b 种不同的符号中选取，用以表示一个非负数（不考虑小数点和负号）. 实际上仍有许多记数系统和进制系统有类似的特征，但不完全符合进制系统的规定，我们把这样的记数系统称为 **广义进制系统** 或 **非标准进制系统**（Non-standard positional numeral systems）. 下面介绍几种常见的广义进制系统.

双射记数系统

标准的进制系统并不能和其表示的数字建立双射，如 1、01、001 表示的数是相同的[3]，而 **双射记数系统**（bijective numeral system）可以和其表示的数字建立双射.

双射 k 进制（$k \geq 1$）使用数集 $\{1, 2, \ldots, k\}$ 来唯一表示一个数，规则如下：

1. 用空串表示 0；
2. 用非空串 $a_n\cdots a_1a_0$ 表示数 $a_nk^n + \cdots + a_1k^1 + a_0k^0 = \sum_{i=0}^{n} a_ik^i$.

对一个正数 m，设其双射 k 进制下的表示为 $a_n\cdots a_1a_0$，则有：

$$\begin{aligned} a_0 &= m - q_0k, & q_0 &= f(m/k), \\ a_1 &= q_0 - q_1k, & q_1 &= f(q_0/k), \\ &\vdots & &\vdots \\ a_n &= q_{n-1} - q_nk, & q_n &= f(q_{n-1}/k) = 0, \end{aligned}$$

[3]我们把最高的非 0 位之前的 0 称为 **前导零**[3]（leading zero）. 类似地，我们可以定义 **后导零**[4]（trailing zero）.

其中 $f(x)=\lceil x\rceil-1$.

例如 Microsoft Excel 的列标签采用的就是双射 26 进制.

在双射记数系统下，我们有 一进制[5]，一进制下的非空串只由 1 构成，串的长度即为其表示的数.

类似 补数法→9.3@p1087 下的叙述，在 $k>1$ 的双射 k 进制下，我们设 $d=k-1$，则 $\cdots dd=:\overline{d}=\sum_{i=0}^{\infty}dk^i=-1$，进而 $\overline{d}k=0$，所以设 x 在双射 k 进制下的表示为 $a_{n-1}\cdots a_1a_0$，则 $\overline{d}ka_{n-1}\cdots a_1a_0$ 即为 $-x$.

下面是双射 k 进制数的一些性质：

- 长度为 $l\geq 0$ 的数共有 k^l 种.
- $k\geq 2$ 时，数 n 在双射 k 进制下表示的长度为 $\lfloor\log_k(n+1)(k-1)\rfloor$.
- $k\geq 2$ 时，若一个数 n 在 k 进制下的表示中不含 0，则其在 k 进制下的表示和在双射 k 进制下的表示相同.

双射 k 进制转十进制和 k 进制转十进制的代码相同，下面是十进制转双射 k 进制的参考实现

实现

```cpp
#include <algorithm>
#include <string>

// only non-negative
// 1 <= base && base < 36
std::string from_dec_bi(int x, int base) {
  std::string res;
  while (x > 0) {
    int q = (x + base - 1) / base - 1, r = x - q * base;
    x = q;

    // 写法 1
    res.push_back(r < 10 ? '0' + r : 'A' + r - 10);
    // 写法 2
    // const static std::string digits = "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ";
    // res.push_back(digits[r]);
  }
  std::reverse(res.begin(), res.end());
  return res;
}

```

有符号位数进制

有的进位制系统允许数位中出现负数，例如 平衡三进制→9.3.3@p1095.

Gray 码

主条目：格雷码→9.3.4@p1097

Gray 码又叫 **循环二进制码** 或 **反射二进制码**（reflected binary code，RBC），是一种特殊的二进制数字系统，常用于数据校验中.

非正基数进制

我们知道对于 k 进制数 $a_n\cdots a_1a_0$，其表示的值为 $\sum_{i=0}^{n}a_ik^i$，我们稍加修改即可定义 $-k$ 进制数 $a_n\cdots a_1a_{(0)_{(-k)}}$ 表示 $\sum_{i=0}^{n}a_i(-k)^i$，其中 $a_n,\ldots,a_1,a_0\in\{0,1,\ldots,k-1\}$. 例如 $12345_{(-10)}=8265_{(10)}$. 这种进制系统叫做 **负底数进制**[6]（negative-base system）.

类似地，我们也可以定义 **复底数进制**[7]（complex-base system），如 2i **进制**[8]（quater-imaginary base、quater-imaginary numeral system），我们还可以定义 **非整数进位制**[9]（non-integer base of numeration）用于表示实数的 β **展开**（β-expansion）等.

混合基数进制

标准的进制系统中，每一个数位对应的基数都是固定的，而混合基数进制允许每一个数位对应不同的基数. 混合基数进制系统最常见的应用就是计时：小时采用 24 进制，分钟和秒采用 60 进制.

$a_n \cdots a_1 a_0$ 在 b 进制下表示的数为 $\sum_{i=0}^{n} a_i b^i$，而在混合基数进制下，其表示的数为 $\sum_{i=0}^{n} a_i \prod_{j=0}^{i-1} b_j$，其中 b_j 为 a_j 对应的基数.

在算法竞赛中，最常见的混合基数进制系统是 **阶乘进制**[10]（factorial number system），其中的数可以记作 $a_n \cdots a_1 a_{(0)_!}$，其表示的数为 $\sum_{i=0}^{n} a_i i!$[4]. 阶乘进制在算法竞赛中的应用可参见 Lehmer 码/Cantor 展开→9.8@p1138.

实现（十进制转阶乘进制）

```cpp
#include <algorithm>
#include <string>

// only non-negative
std::string from_dec_factorial(int x) {
  if (x == 0) return "0";
  std::string res;
  int base = 1;
  while (x) {
    int r = x % base;
    x /= base++;

    // 写法 1
    res.push_back(r < 10 ? '0' + r : 'A' + r - 10);
    // 写法 2
    // const static std::string digits =
"0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ";
    // res.push_back(digits[r]);
  }
  std::reverse(res.begin(), res.end());
  return res;
}

```

实现（阶乘进制转十进制）

```cpp
#include <cctype>
#include <string>

// only non-negative
int to_dec_factorial(std::string const& s) {
  int res = 0, base = s.size();

```

[4] a_i 对应的基数是 $i+1$，$0 \le a_i \le i$. 注意到 $(n+1)! - n! = n \cdot n!$，所以数在阶乘进制下的表示在去除前导零的情况下是唯一的.

```
  for (char c : s) {
    res *= base--;
    if (isdigit(c))
      res += c - '0';
    else if (isupper(c))
      res += c - 'A' + 10;
    else if (islower(c))
      res += c - 'a' + 10;
  }
  return res;
}

```

C++ 中的实现

对于非负数，C++ 中用 `<前缀><数位><后缀>` 表示一个整数字面量，其中 `<数位>` 和 `<后缀>` 均可以为空. `<后缀>` 用于表示字面量的类型，如 `u` 或 `U` 表示该字面量为 `unsigned`、`l` 或 `L` 表示该字面量为 `long` 等. 对于 `<前缀>`:

- 当 `<前缀>` 为 `0x` 或 `0X` 时表示十六进制字面量，此时 `<数位>` 中的字符只能在 `0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, A, b, B, c, C, d, D, e, E, f, F` 中选取. 例如 `0x1234ABCD` 为 $1234\mathrm{ABCD}_{(16)} = 305\ 441\ 741$;
- 当 `<前缀>` 为 `0` 时表示八进制字面量，此时 `<数位>` 中的字符只能在 `0, 1, 2, 3, 4, 5, 6, 7` 中选取. 例如 `01234567` 为 $1234567_{(8)} = 342391$; [5]
- 当 `<前缀>` 为 `1`、`2`、`3`、`4`、`5`、`6`、`7`、`8` 或 `9` 时表示十进制字面量，此时 `<数位>` 中的字符只能在 `0, 1, 2, 3, 4, 5, 6, 7, 8, 9` 中选取;
- C++14 起，当 `<前缀>` 为 `0b` 或 `0B` 时表示二进制字面量，此时 `<数位>` 中的字符只能在 `0, 1` 中选取. 例如 `0b11001010` 为 $11001010_{(2)} = 202$.

参考资料与注释

- Positional notation - Wikipedia[11]
- Method of complements - Wikipedia[12]
- Non-standard positional numeral systems - Wikipedia[13]
- Bijective numeration - Wikipedia[14]
- Midy's theorem - Wikipedia[15]
- N3472 - Binary Literals in the C++ Core Language[16]

[1] *p* 进数 [2] A028416 [3] 前导零

[4] 后导零 [5] 一进制 [6] 负底数进制

[7] 复底数进制 [8] 2i 进制 [9] 非整数进位制

[10] 阶乘进制 [11] Positional notation - Wikipedia [12] Method of complements - Wikipedia

[13] Non-standard positional numeral systems - Wikipedia [14] Bijective numeration - Wikipedia [15] Midy's theorem - Wikipedia

[5] `0` 是八进制字面量.

[16] N3472 - Binary Literals in the C++ Core Language 

9.3.3 平衡三进制

Authors: Enter-tainer, Falicitas, HeRaNO, iamtwz, ImpleLee, Tiphereth-A, Xeonacid, Yanjun-Zhao

定义

平衡三进制，也称为对称三进制．这是一个广义进位系统．

正规的三进制的数字都是由 `0`,`1`,`2` 构成的，而平衡三进制的数字是由 `-1`,`0`,`1` 构成的．它的基数也是 `3`（因为有三个可能的值）．由于将 `-1` 写成数字不方便，我们将使用字母 `Z` 来代替 `-1`．

解释

这里有几个例子：

十进制	平衡三进制	十进制	平衡三进制
`0`	`0`	`5`	`1ZZ`
`1`	`1`	`6`	`1Z0`
`2`	`1Z`	`7`	`1Z1`
`3`	`10`	`8`	`10Z`
`4`	`11`	`9`	`100`

该数字系统的负数表示起来很容易：只需要将正数的数字倒转即可（`Z` 变成 `1`,`1` 变成 `Z`）．

十进制	平衡三进制
`-1`	`Z`
`-2`	`Z1`
`-3`	`Z0`
`-4`	`ZZ`
`-5`	`Z11`

很容易就可以看到，负数最高位是 `Z`，正数最高位是 `1`．

过程

在平衡三进制的转转换法中，需要先写出一个给定的数 `x` 在标准三进制中的表示．当 `x` 是用标准三进制表示时，其数字的每一位都是 `0`、`1` 或 `2`．从最低的数字开始迭代，我们可以先跳过任何的 `0` 和 `1`，但是如果遇到 `2` 就应该先将其变成 `Z`，下一位数字再加上 `1`．而遇到数字 `3` 则应该转换为 `0` 下一位数字再加上 `1`．

应用一

把 `64` 转换成平衡三进制．

首先，我们用标准三进制数来重写这个数：

$$64_{10} = 02101_3$$

让我们从对整个数影响最小的数字（最低位）进行处理：

- `101` 被跳过（因为在平衡三进制中允许 `0` 和 `1`）；
- `2` 变成了 `Z`，它左边的数字加 `1`，得到 `1Z101`；
- `1` 被跳过，得到 `1Z101`．

最终的结果是 `1Z101`．

我们再把它转换回十进制：

$$\texttt{1Z101} = 81 \times 1 + 27 \times (-1) + 9 \times 1 + 3 \times 0 + 1 \times 1 = 64_{10}$$

应用二

把 `237` 转换成平衡三进制.

首先，我们用标准三进制数来重写这个数：

$$237_{10} = 22210_3$$

- `0` 和 `1` 被跳过（因为在平衡三进制中允许 `0` 和 `1`）；
- `2` 变成 `Z`，左边的数字加 `1`，得到 `23Z10`；
- `3` 变成 `0`，左边的数字加 `1`，得到 `30Z10`；
- `3` 变成 `0`，左边的数字（默认是 `0`）加 `1`，得到 `100Z10`；
- `1` 被跳过，得到 `100Z10`.

最终的结果是 `100Z10`.

我们再把它转换回十进制：

$$\texttt{100Z10} = 243 \cdot 1 + 81 \cdot 0 + 27 \cdot 0 + 9 \cdot (-1) + 3 \cdot 1 + 1 \cdot 0 = 237_{10}$$

性质

对于一个平衡三进制数 X_3 来说，其可以按照每一位 x_i 乘上对应的权值 3^i 来唯一得到一个十进制数 Y_{10}.

那对于一个十进制数 Y_{10}，是否 **唯一对应一个平衡三进制数** 呢?

答案是肯定的，这种性质被叫做平衡三进制的唯一性.

证明

我们利用 **反证法** 来求证：

假设一个十进制数 Y_{10}，存在两个 **不同的平衡三进制数** A_3, B_3 转化成十进制时等于 Y_{10}，即证 $A_3 = B_3$. 分情况讨论：

1. 当 $Y_{10} = 0$，显然 $A_3 = B_3 = 0_3$，与假设矛盾.
2. 当 $Y_{10} > 0$：
   - 将 A_3，B_3 的数位按低位到高位编号，记 a_i 为 A_3 的第 i 位，b_i 为 B 的第 i 位. 在 A_3, B_3 中，必存在 i 使得 $a_i \neq b_i$. 可以发现第 $i-1, i-2, ..., 0$ 位均与证明无关. 因此，将 A_3, B_3 按位右移 i 位，得到 $A_{(3)'}, B_{(3)'}$，原问题等价于证明 $A_{(3)'} = B_{(3)'}$.
   - 对于 $A_{(3)'}, B_{(3)'}$ 第 0 位，$a_0 \neq b_0$. 假设 $b_0 > a_0$（$a_0 > b_0$ 时结果相同），易知 $b_0 - a_0 \in \{1, 2\}$. $A_{(3)'}$ 的位 $i = 1, 2, 3, ...$ 对于 $A_{(3)'}$ 的值的贡献为 $S_1 = a_1 \times 3^1 + a_2 \times 3^2 + ...$，$B_{(3)'}$ 的位 $i = 1, 2, 3, ...$ 对于 $B_{(3)'}$ 的值的贡献为 $S_2 = b_1 \times 3^1 + b_2 \times 3^2 + ...$. 由于 $A_{(3)'} = B_{(3)'}$，得 $S_1 - S_2 = b_0 - a_0$. S_1, S_2 有公因子 3，而 $b_0 - a_0$ 不能被 3 整除，与假设矛盾，因此 $A_{(3)'} \neq B_{(3)'}$
3. 当 $Y_{10} < 0$，证法与 $Y_{10} > 0$ 相同.

故对于任意十进制 Y_{10}，均有唯一对应的平衡三进制 X_3.

练习题

Topcoder SRM 604 PowerOfThree[1]

本页面部分内容译自博文 Троичная сбалансированная система счисления[2] 与其英文翻译版 Balanced Ternary[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.

外部参考资料

[1] Topcoder SRM 604 PowerOfThree  [2] Троичная сбалансированная система счисления  [3] Balanced Ternary 

9.3.4 格雷码

> **Authors:** CCXXXI, danni, Early0v0, Enter-tainer, frank-xjh, Henry-ZHR, infiWang, Ir1d, mingyEx, ouuan, sshwy, StudyingFather, Tiphereth-A, Yanjun-Zhao

格雷码是一个二进制数字系统，其中两个相邻数的二进制位只有一位不同．举个例子，3 位二进制数的格雷码序列为

$$000, 001, 011, 010, 110, 111, 101, 100$$

注意序列的下标我们以 0 为起点，也就是说 $G(0) = 000, G(4) = 110$.

格雷码由贝尔实验室的 Frank Gray 于 1940 年代提出，并于 1953 年获得专利.

构造格雷码（变换）

格雷码的构造方法很多．我们首先介绍手动构造方法，然后会给出构造的代码以及正确性证明.

手动构造

k 位的格雷码可以通过以下方法构造．我们从全 0 格雷码开始，按照下面策略：

1. 翻转最低位得到下一个格雷码，（例如 $000 \to 001$）；
2. 把最右边的 1 的左边的位翻转得到下一个格雷码，（例如 $001 \to 011$）；

交替按照上述策略生成 2^{k-1} 次，可得到 k 位的格雷码序列.

镜像构造

k 位的格雷码可以从 $k-1$ 位的格雷码以上下镜射后加上新位的方式快速得到，如下图：

$$\begin{array}{cccccc} k=1 & & k=2 & & k=3 \\ 0 & 0 & 00 & 00 & 000 \\ 1 & 1 & 01 & 01 & 001 \\ & 1 & 11 & 11 & 011 \\ & \to 0 \to & 10 & \to 10 \to & 010 \\ & & & 10 & 110 \\ & & & 11 & 111 \\ & & & 01 & 101 \\ & & & 00 & 100 \end{array}$$

计算方法

我们观察一下 n 的二进制和 $G(n)$．可以发现，如果 $G(n)$ 的二进制第 i 位为 1，仅当 n 的二进制第 i 位为 1，第 $i+1$ 位为 0 或者第 i 位为 0，第 $i+1$ 位为 1．于是我们可以当成一个异或的运算，即

$$G(n) = n \oplus \left\lfloor \frac{n}{2} \right\rfloor$$

```
1 int g(int n) { return n ^ (n >> 1); }
```

正确性证明

接下来我们证明一下，按照上述公式生成的格雷码序列，相邻两个格雷码的二进制位有且仅有一位不同.

我们考虑 n 和 $n+1$ 的区别．把 n 加 1，相当于把 n 的二进制下末位的连续的 1 全部变成取反，然后把最低位的 0 变成 1．我们这样表示 n 和 $n+1$ 的二进制位：

$$(n)_2 = \cdots 0\underbrace{11\cdots11}_{k个}$$
$$(n+1)_2 = \cdots 1\underbrace{00\cdots00}_{k个}$$

于是我们在计算 $g(n)$ 和 $g(n+1)$ 的时候，后 k 位都会变成 $\underbrace{100\cdots00}_{k个}$ 的形式，而第 $k+1$ 位是不同的，因为 n 和 $n+1$ 除了后 $k+1$ 位，其他位都是相同的．因此第 $k+1$ 位要么同时异或 1，要么同时异或 0．两种情况，第 $k+1$ 位都是不同的．而除了后 $k+1$ 位以外的二进制位也是做相同的异或运算，结果是相同的．

证毕．

通过格雷码构造原数（逆变换）

接下来我们考虑格雷码的逆变换，即给你一个格雷码 g，要求你找到原数 n．我们考虑从二进制最高位遍历到最低位（最低位下标为 1，即个位；最高位下标为 k）．则 n 的二进制第 i 位与 g 的二进制第 i 位 g_i 的关系如下：

$$\begin{aligned}
n_k &= g_k \\
n_{k-1} &= g_{k-1} \oplus n_k &&= g_k \oplus g_{k-1} \\
n_{k-2} &= g_{k-2} \oplus n_{k-1} &&= g_k \oplus g_{k-1} \oplus g_{k-2} \\
n_{k-3} &= g_{k-3} \oplus n_{k-2} &&= g_k \oplus g_{k-1} \oplus g_{k-2} \oplus g_{k-3} \\
&\vdots \\
n_{k-i} &= \bigoplus_{j=0}^{i} g_{k-j}
\end{aligned}$$

```
int rev_g(int g) {
  int n = 0;
  for (; g; g >>= 1) n ^= g;
  return n;
}
```

实际应用

格雷码有一些十分有用的应用，有些应用让人意想不到：

- k 位二进制数的格雷码序列可以当作 k 维空间中的一个超立方体（二维里的正方形，一维里的单位向量）顶点的哈密尔顿回路，其中格雷码的每一位代表一个维度的坐标．
- 格雷码被用于最小化数字模拟转换器（比如传感器）的信号传输中出现的错误，因为它每次只改变一个位．
- 格雷码可以用来解决汉诺塔的问题．

  设盘的数量为 n．我们从 n 位全 0 的格雷码 $G(0)$ 开始，依次移向下一个格雷码（$G(i)$ 移向 $G(i+1)$）．当前格雷码的二进制第 i 位表示从小到大第 i 个盘子．

  由于每一次只有一个二进制位会改变，因此当第 i 位改变时，我们移动第 i 个盘子．在移动盘子的过程中，除了最小的盘子，其他任意一个盘子在移动的时候，只能有一个放置选择．在移动第一个盘子的时候，我们总是有两个放置选择．于是我们的策略如下：

  如果 n 是一个奇数，那么盘子的移动路径为 $f \to t \to r \to f \to t \to r \to \cdots$，其中 f 是最开始的柱子，t 是最终我们把所有盘子放到的柱子，r 是中间的柱子．

  如果 n 是偶数：$f \to r \to t \to f \to r \to t \to \cdots$
- 格雷码也在遗传算法理论中得到应用．

习题

- CSP S2 2019 D1T1[1] Difficulty: easy
- SGU #249 Matrix[2] Difficulty: medium

> 本页面部分内容译自博文 Код Грея[3] 与其英文翻译版 Gray code[4]. 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] CSP S2 2019 D1T1

[2] SGU #249 Matrix

[3] Код Грея

[4] Gray code

9.4 位操作

位操作指的是对整数二进制表示的一元和二元操作，分为 **位运算** 和 **移位** 两类. 位操作是 CPU 中最基础的一类运算，其速度往往是相当快的.

整数与位序列

另请参阅：整数类型→4.2.3@p328、补数法→9.3.2@p1087

我们将只由 `0` 或 `1` 构成的长度固定的序列称为位序列. 最左边的位称为最高位，最右边的位称为最低位.

计算机中用位序列表示一定范围内的整数. 长度为 N 的位序列只有 2^N 种，所以只能和 2^N 个整数建立一一对应关系. 这种一一对应关系可以分为两类：**有符号** 和 **无符号**. 有符号指的是对应的整数有负数，无符号指的是对应的整数全部为非负数.

- 对于无符号的对应关系，我们可以直接将整数的二进制表示作为位序列，长度不足就在高位补 `0`.

  在无符号的对应关系下，长度为 N 的位序列可以表示 $[0, 2^N-1]$ 内的整数.
- 对于有符号的对应关系，我们有两种表示规则：**反码**（ones' complement）和 **补码**（two's complement）.

  对于非负整数来说，其表示规则和无符号的规则一致；对于负整数来说，我们将其相反数对应的位序列 **按位取反**（即将 `0` 变为 `1`，将 `1` 变为 `0`）后的结果称为反码，将反码按无符号的对应关系转为整数，然后加一，最后按无符号的对应关系转为位序列，超出原位序列长度的部分舍弃，得到的新序列称为补码.

  在反码的对应关系下，长度为 N 的位序列可以表示 $[-2^{N-1}+1, 2^{N-1}-1]$ 内的整数.

  在补码的对应关系下，长度为 N 的位序列可以表示 $[-2^{N-1}, 2^{N-1}-1]$ 内的整数.

以 3 位的位序列为例：

位序列	无符号整数	有符号整数（反码）	有符号整数（补码）
`000`	0	0	0
`001`	1	1	1
`010`	2	2	2
`011`	3	3	3
`100`	4	-3	-4
`101`	5	-2	-3
`110`	6	-1	-2
`111`	7	-0	-1

可以看到反码的最大问题是会出现 -0 这个实际上不存在的「负数」，所以一般情况下我们只用补码. 由于表示有符号整数时，其正负号仅由位序列的最高位决定，所以我们将这一位称为 **符号位**.

将位序列转为整数也是容易做到的：对非负数来说不需要特别操作，对反码来说取反即可得到对应的相反数，对补码来说取反加一即可得到对应的相反数.

位运算

位运算指的是对位序列逐位应用某些 布尔函数→9.2@p1080 的运算. 形式化地说，对布尔函数 $f:\mathbf{B}^k\to\mathbf{B}$，位运算即为形如

$$F:(\mathbf{B}^m)^k\to\mathbf{B}^m$$

$$((p_{1,1},\dots,p_{m,1}),\dots,(p_{1,k},\dots,p_{m,k}))\mapsto(f(p_{1,1},\dots,p_{1,k}),\dots,f(p_{m,1},\dots,p_{m,k}))$$

的函数，其中 m 为位序列的长度. 同样的，我们一般只研究一元和二元的位运算. 如无特殊说明，下文的位运算仅限于一元和二元的情况.

一般来说，我们把 **按位取反**、**按位与**、**按位或**、**按位异或** 视作基本的位运算，其余的位运算均可以通过这些运算组合得到.

位运算	数学符号表示	对应的布尔函数	C++ 运算符	解释
按位取反	NOT	$\neg$	`~`	0 变为 1，1 变为 0
按位与	AND	$\wedge$	`&`	只有两个对应位都为 1 时才为 1
按位或	OR	$\vee$	`|`	只要两个对应位中有一个 1 时就为 1
按位异或	$\oplus$、XOR	$\oplus$	`^`	只有两个对应位不同时才为 1

> ⚠ **Warning**
>
> 注意区分位运算与布尔函数.

例如：

- $\mathrm{NOT}\,01010111=10101000$,
- $01010011\,\mathrm{AND}\,00110010=00010010$,
- $01010011\,\mathrm{OR}\,00110010=01110011$,
- $01010011\,\mathrm{XOR}\,00110010=01100001$.

由于上述四种位运算在运算时，各个位的运算独立，所以这四种位运算能直接继承其对应布尔函数的性质.

为方便起见，在位序列长度已知时，我们也可以直接对整数做位运算，例如：

$$\mathrm{NOT}\,5=-6,$$

$$\mathrm{NOT}(-5)=4,$$

$$5\,\mathrm{AND}\,6=4,$$

$$5\,\mathrm{OR}\,6=7,$$

$$5\,\mathrm{XOR}\,6=3.$$

假设 $x,y\geq 0$，我们也可以将位运算用求和的方式表示：

$$\begin{aligned}
\operatorname{NOT} x &= \sum_{n=0}^{\lfloor \log_2 x \rfloor} 2^n \left(\left(\left\lfloor \frac{x}{2^n} \right\rfloor \bmod 2 + 1 \right) \bmod 2 \right) \\
&= \sum_{n=0}^{\lfloor \log_2 x \rfloor} \left(2^{\lfloor \log_2 x \rfloor + 1} - 1 - x \right) \\
x \operatorname{AND} y &= \sum_{n=0}^{\lfloor \log_2 \max\{x,y\} \rfloor} 2^n \left(\left\lfloor \frac{x}{2^n} \right\rfloor \bmod 2 \right) \left(\left\lfloor \frac{y}{2^n} \right\rfloor \bmod 2 \right) \\
x \operatorname{OR} y &= \sum_{n=0}^{\lfloor \log_2 \max\{x,y\} \rfloor} 2^n \left(\left(\left\lfloor \frac{x}{2^n} \right\rfloor \bmod 2 \right) + \left(\left\lfloor \frac{y}{2^n} \right\rfloor \bmod 2 \right) - \left(\left\lfloor \frac{x}{2^n} \right\rfloor \bmod 2 \right) \left(\left\lfloor \frac{y}{2^n} \right\rfloor \bmod 2 \right) \right) \\
x \operatorname{XOR} y &= \sum_{n=0}^{\lfloor \log_2 \max\{x,y\} \rfloor} 2^n \left(\left(\left(\left\lfloor \frac{x}{2^n} \right\rfloor \bmod 2 \right) + \left(\left\lfloor \frac{y}{2^n} \right\rfloor \bmod 2 \right) \right) \bmod 2 \right) \\
&= \sum_{n=0}^{\lfloor \log_2 \max\{x,y\} \rfloor} 2^n \left(\left(\left\lfloor \frac{x}{2^n} \right\rfloor + \left\lfloor \frac{y}{2^n} \right\rfloor \right) \bmod 2 \right)
\end{aligned}$$

在不引起歧义的情况下，下文中省略「按位」.

移位

另请参阅：C++ 位操作符→4.2.4@p339.

移位为一类将位序列「按位向左或向右移动」的二元运算，第一个参数为位序列，第二个参数一般为非负整数. 向左移动称为 **左移**，向右移动称为 **右移**. 根据对移动后的空位填充方式，可将移位操作分为 **算术移位**、**逻辑移位**、**循环移位**. 其中

- 逻辑移位用 0 填充空位，
- 算术右移用符号位填充空位，算术左移和逻辑左移相同，
- 循环移位用溢出位填充空位.

例如对 8 位的位序列 `10 01 01 10`：

操作	结果
算术左移 2 位	`01 01 10 00`
算术右移 2 位	`11 10 01 01`
逻辑左移 2 位	`01 01 10 00`
逻辑右移 2 位	`00 10 01 01`
循环左移 2 位	`01 01 10 10`
循环右移 2 位	`10 10 01 01`

在 C++ 中，我们用 `a << b` 表示左移，`a >> b` 表示右移，具体采用何种移位规则参见 C++ 位操作符→4.2.4@p339.

我们可以用如下代码实现循环移位：

实现

```cpp
// From <https://stackoverflow.com/a/776523/224132>
#include <climits>
#include <cstdint>

uint32_t rotl32(uint32_t value, unsigned int count) {
  const unsigned int mask = CHAR_BIT * sizeof(value) - 1;
```

```
  count &= mask;
  return (value << count) | (value >> (-count & mask));
}

uint32_t rotr32(uint32_t value, unsigned int count) {
  const unsigned int mask = CHAR_BIT * sizeof(value) - 1;
  count &= mask;
  return (value >> count) | (value << (-count & mask));
}

```

位操作的应用

位操作一般有三种作用：

1. 高效地进行某些运算，代替其它低效的方式．参见 编译优化 # 强度削减→4.4.10@p456．
2. 表示集合→9.5@p1107（常用于 状压 DP→7.8@p744）．
3. 题目本来就要求进行位操作．

需要注意的是，用位操作代替其它运算方式在很多时候并不能带来太大的优化，反而会使代码变得复杂，使用时需要斟酌．

有关 2 的幂的应用

由于位操作针对的是二进制表示，因此可以推广出许多与 2 的整数次幂有关的应用．

将一个数乘（除）2 的非负整数次幂：

- C++

```cpp
int mulPowerOfTwo(int n, int m) {  // 计算 n*(2^m)
  return n << m;
}

int divPowerOfTwo(int n, int m) {  // 计算 n/(2^m)
  return n >> m;
}

```

- Python

```python
def mulPowerOfTwo(n, m):  # 计算 n*(2^m)
    return n << m


def divPowerOfTwo(n, m):  # 计算 n/(2^m)
    return n >> m

```

> ⚠ **Warning**
>
> 我们平常写的除法是向 0 取整，而这里的右移是向下取整（注意这里的区别），即当数大于等于 0 时两种方法等价，当数小于 0 时会有区别，如：`-1 / 2` 的值为 0，而 `-1 >> 1` 的值为 −1．

取绝对值

在某些机器上，效率比 `n > 0 ? n : -n` 高.

- C++

```cpp
int Abs(int n) {
  return (n ^ (n >> 31)) - (n >> 31);
  /* n>>31 取得 n 的符号，若 n 为正数，n>>31 等于 0，若 n 为负数，n>>31 等于 -1
    若 n 为正数 n^0=n，数不变，若 n 为负数有 n^(-1)
    需要计算 n 和 -1 的补码，然后进行异或运算，
    结果 n 变号并且为 n 的绝对值减 1，再减去 -1 就是绝对值 */
}

```

- Python

```python
def Abs(n):
    return (n ^ (n >> 31)) - (n >> 31)
    """
    n>>31 取得 n 的符号，若 n 为正数，n>>31 等于 0，若 n 为负数，n>>31 等于 -1
    若 n 为正数 n^0=n，数不变，若 n 为负数有 n^(-1)
    需要计算 n 和 -1 的补码，然后进行异或运算，
    结果 n 变号并且为 n 的绝对值减 1，再减去 -1 就是绝对值
    """


```

取两个数的最大/最小值

在某些机器上，效率比 `a > b ? a : b` 高.

- C++

```cpp
// 如果 a >= b，(a - b) >> 31 为 0，否则为 -1
int max(int a, int b) { return (b & ((a - b) >> 31)) | (a & (~(a - b)
>> 31)); }

int min(int a, int b) { return (a & ((a - b) >> 31)) | (b & (~(a - b)
>> 31)); }

```

- Python

```python
# 如果 a >= b，(a - b) >> 31 为 0，否则为 -1
def max(a, b):
    return b & ((a - b) >> 31) | a & (~(a - b) >> 31)


def min(a, b):
    return a & ((a - b) >> 31) | b & (~(a - b) >> 31)


```

判断两非零数符号是否相同

- C++

```cpp
bool isSameSign(int x, int y) {  // 有 0 的情况例外
```

```
  return (x ^ y) >= 0;
}

```

- Python

```
# 有 0 的情况例外
def isSameSign(x, y):
    return (x ^ y) >= 0


```

交换两个数

> **该方法具有局限性**
>
> 这种方式只能用来交换两个整数，使用范围有限.
>
> 对于一般情况下的交换操作，推荐直接调用 `algorithm` 库中的 `std::swap` 函数.

```
void swap(int& a, int& b) { a ^= b ^= a ^= b; }

```

操作一个数的二进制位

获取一个数二进制的某一位:

- C++

```
// 获取 a 的第 b 位，最低位编号为 0
int getBit(int a, int b) { return (a >> b) & 1; }

```

- Python

```
# 获取 a 的第 b 位，最低位编号为 0
def getBit(a, b):
    return (a >> b) & 1


```

将一个数二进制的某一位设置为 0:

- C++

```
// 将 a 的第 b 位设置为 0 ，最低位编号为 0
int unsetBit(int a, int b) { return a & ~(1 << b); }

```

- Python

```
# 将 a 的第 b 位设置为 0 ，最低位编号为 0
def unsetBit(a, b):
    return a & ~(1 << b)


```

将一个数二进制的某一位设置为 1:

- C++

```cpp
// 将 a 的第 b 位设置为 1 ，最低位编号为 0
int setBit(int a, int b) { return a | (1 << b); }

```

- Python

```python
# 将 a 的第 b 位设置为 1 ，最低位编号为 0
def setBit(a, b):
    return a | (1 << b)

```

将一个数二进制的某一位取反：

- C++

```cpp
// 将 a 的第 b 位取反 ，最低位编号为 0
int flapBit(int a, int b) { return a ^ (1 << b); }

```

- Python

```python
# 将 a 的第 b 位取反 ，最低位编号为 0
def flapBit(a, b):
    return a ^ (1 << b)

```

这些操作相当于将一个 32 位整型变量当作一个长度为 32 的布尔数组.

汉明权重

汉明权重是一串符号中不同于（定义在其所使用的字符集上的）零符号（zero-symbol）的个数. 对于一个二进制数，它的汉明权重就等于它 1 的个数（即 `popcount`）.

求一个数的汉明权重可以循环求解: 我们不断地去掉这个数在二进制下的最后一位（即右移 1 位），维护一个答案变量，在除的过程中根据最低位是否为 1 更新答案.

代码如下：

```cpp
// 求 x 的汉明权重
int popcount(int x) {
  int cnt = 0;
  while (x) {
    cnt += x & 1;
    x >>= 1;
  }
  return cnt;
}

```

求一个数的汉明权重还可以使用 `lowbit` 操作：我们将这个数不断地减去它的 `lowbit`[1]，直到这个数变为 0.

代码如下：

```cpp
// 求 x 的汉明权重
int popcount(int x) {
```

[1]一个数二进制表示从低往高的第一个 1 连同后面的零, 如 $(1010)_2$ 的 `lowbit` 是 $(0010)_2$, 详见 树状数组→10.12@p2078.

```
  int cnt = 0;
  while (x) {
    cnt++;
    x -= x & -x;
  }
  return cnt;
}

```

构造汉明权重递增的排列

在 状压 DP→7.8@p744 中，按照 popcount 递增的顺序枚举有时可以避免重复枚举状态．这是构造汉明权重递增的排列的一大作用．

下面我们来具体探究如何在 $O(n)$ 时间内构造汉明权重递增的排列．

我们知道，一个汉明权重为 n 的最小的整数为 2^n-1．只要可以在常数时间构造出一个整数汉明权重相等的后继，我们就可以通过枚举汉明权重，从 2^n-1 开始不断寻找下一个数的方式，在 $O(n)$ 时间内构造出 $0\sim n$ 的符合要求的排列．

而找出一个数 x 汉明权重相等的后继有这样的思路，以 $(10110)_2$ 为例：

- 把 $(10110)_2$ 最右边的 1 向左移动，如果不能移动，移动它左边的 1，以此类推，得到 $(11010)_2$．
- 把得到的 $(11010)_2$ 最后移动的 1 原先的位置一直到最低位的所有 1 都移到最右边．这里最后移动的 1 原来在第三位，所以最后三位 010 要变成 001，得到 $(11001)_2$．

这个过程可以用位操作优化：

```
  int t = x + (x & -x);
  x = t | ((((t & -t) / (x & -x)) >> 1) - 1);
```

- 第一个步骤中，我们把数 x 加上它的 `lowbit`，在二进制表示下，就相当于把 x 最右边的连续一段 1 换成它左边的一个 1．如刚才提到的二进制数 $(10110)_2$，它在加上它的 `lowbit` 后是 $(11000)_2$．这其实得到了我们答案的前半部分．
- 我们接下来要把答案后面的 1 补齐，t 的 `lowbit` 是 x 最右边连续一段 1 最左边的 1 移动后的位置，而 x 的 `lowbit` 则是 x 最右边连续一段 1 最右边的位置．还是以 $(10110)_2$ 为例，$t=(11000)_2$，$\text{lowbit}(t)=(01000)_2$，$\text{lowbit}(x)=(00010)_2$．
- 接下来的除法操作是这种位操作中最难理解的部分，但也是最关键的部分．我们设 **原数** 最右边连续一段 1 最高位的 1 在第 r 位上（位数从 0 开始），最低位的 1 在第 l 位，t 的 `lowbit` 等于 `1 << (r+1)`，x 的 `lowbit` 等于 `1 << l`，`(((t&-t)/(x&-x))>>1)` 得到的，就是 `(1<<(r+1))/(1<<l)/2 = (1<<r)/(1<<l) = 1<<(r-l)`，在二进制表示下就是 1 后面跟上 $r-l$ 个零，零的个数正好等于连续 1 的个数减去 1．举我们刚才的数为例，$\frac{\text{lowbit(t)}/2}{\text{lowbit(x)}}=\frac{(00100)_2}{(00010)_2}=(00010)_2$．把这个数减去 1 得到的就是我们要补全的低位，或上原来的数就可以得到答案．

所以枚举 $0\sim n$ 按汉明权重递增的排列的完整代码为：

```
  for (int i = 0; (1 << i) - 1 <= n; i++) {
    for (int x = (1 << i) - 1, t; x <= n; t = x + (x & -x),
             x = x ? (t | ((((t & -t) / (x & -x)) >> 1) - 1)) : (n + 1)) {
      // 写下需要完成的操作
    }
  }
```

其中要注意 0 的特判，因为 0 没有相同汉明权重的后继．

C++ 中的相关类与函数

GCC 内建函数

GCC 中还有一些用于位操作的内建函数：

- `int __builtin_ffs(int x)`：返回 x 的二进制末尾最后一个 1 的位置，位置的编号从 1 开始（最低位编号为 1）．当 x 为 0 时返回 0．
- `int __builtin_clz(unsigned int x)`：返回 x 的二进制的前导 0 的个数．当 x 为 0 时，结果未定义．
- `int __builtin_ctz(unsigned int x)`：返回 x 的二进制末尾连续 0 的个数．当 x 为 0 时，结果未定义．
- `int __builtin_clrsb(int x)`：当 x 的符号位为 0 时返回 x 的二进制的前导 0 的个数减一，否则返回 x 的二进制的前导 1 的个数减一．
- `int __builtin_popcount(unsigned int x)`：返回 x 的二进制中 1 的个数．
- `int __builtin_parity(unsigned int x)`：判断 x 的二进制中 1 个数的奇偶性．

这些函数都可以在函数名末尾添加 `l` 或 `ll`（如 `__builtin_popcountll`）来使参数类型变为 `(unsigned)long` 或 `(unsigned)long long`（返回值仍然是 `int` 类型）．例如，我们有时候希望求出一个数以二为底的对数，如果不考虑 `0` 的特殊情况，就相当于这个数二进制的位数 `-1`，而一个 `N` 位整数 `n` 的二进制表示的位数可以使用 `N - __builtin_clz(n)` 表示，因此 `N - 1 - __builtin_clz(n)` 就可以求出 `n` 以二为底的对数．

由于这些函数是内建函数，经过了编译器的高度优化，运行速度十分快（有些甚至只需要一条指令）．

更多位数

如果需要操作的位序列非常长，可以使用 `std::bitset`→4.3.4@p389．

题目推荐

- Luogu P1225 黑白棋游戏[1]

参考资料与注释

1. 位运算技巧[2]
2. Bit Operation Builtins (Using the GNU Compiler Collection (GCC))[3]
3. Bitwise operation - Wikipedia[4]

[1] Luogu P1225 黑白棋游戏 

[2] 位运算技巧 

[3] Bit Operation Builtins (Using the GNU Compiler Collection (GCC)) 

[4] Bitwise operation - Wikipedia 

9.5 二进制集合操作

前置知识：位运算→9.4@p1099、整数与位序列→9.4@p1099．

一个数的二进制表示可以看作是一个集合（0 表示不在集合中，1 表示在集合中）．比如集合 $\{1,3,4,8\}$，可以表示成 $(100011010)_2$．而对应的位运算也就可以看作是对集合进行的操作．

操作	集合表示	位运算表示
交集	$a \cap b$	a AND b
并集	$a \cup b$	a OR b
补集	$\bar{a}$	NOT a（全集为二进制都是 1）

差集	$a \setminus b$	a AND NOT b
对称差	$a \triangle b$	a XOR b

在进一步介绍集合的子集遍历操作之前，先看位运算的有关应用例子.

模 2 的幂

一个数对 2 的非负整数次幂取模，等价于取二进制下一个数的后若干位，等价于和 $mod - 1$ 进行与操作.

- C++

```cpp
int modPowerOfTwo(int x, int mod) { return x & (mod - 1); }
```

- Python

```python
def modPowerOfTwo(x, mod):
    return x & (mod - 1)
```

于是可以知道，2 的非负整数次幂对它本身取模，结果为 0，即如果 n 是 2 的非负整数次幂，n 和 $n-1$ 的与操作结果为 0.

事实上，对于一个正整数 n，$n-1$ 会将 n 的最低 1 位置零，并将后续位数全部置 1. 因此，n 和 $n-1$ 的与操作等价于删掉 n 的最低 1 位.

借此可以判断一个数是不是 2 的非负整数次幂. 当且仅当 n 的二进制表示只有一个 1 时，n 为 2 的非负整数次幂.

- C++

```cpp
bool isPowerOfTwo(int n) { return n > 0 && (n & (n - 1)) == 0; }
```

- Python

```python
def isPowerOfTwo(n):
    return n > 0 and (n & (n - 1)) == 0
```

子集遍历

遍历一个二进制数表示的集合的全部子集，等价于枚举二进制数对应掩码的所有子掩码.

掩码是一串二进制码，用于和源码进行与运算，得到屏蔽源码的若干输入位后的新操作数.

掩码对于源码可以起到遮罩的作用，掩码中的 1 位意味着源码的相应位得到保留，掩码中的 0 位意味着源码的相应位进行置 0 操作. 将掩码的若干 1 位改为 0 位可以得到掩码的子掩码，掩码本身也是自己的子掩码.

给定一个掩码 m，希望有效迭代 m 的所有子掩码 s，可以考虑基于位运算技巧的实现.

```cpp
// 降序遍历 m 的非空子集
int s = m;
while (s > 0) {
  // s 是 m 的一个非空子集
  s = (s - 1) & m;
}
```

或者使用更紧凑的 for 语句:

```cpp
// 降序遍历 m 的非空子集
for (int s = m; s; s = (s - 1) & m)
// s 是 m 的一个非空子集
```

这两段代码都不会处理等于 0 的子掩码，要想处理等于 0 的子掩码可以使用其他办法，例如：

```
// 降序遍历 m 的子集
for (int s = m;; s = (s - 1) & m) {
  // s 是 m 的一个子集
  if (s == 0) break;
}
```

接下来证明，上面的代码访问了所有 m 的子掩码，没有重复，并且按降序排列.

假设有一个当前位掩码 s，并且想继续访问下一个位掩码. 在掩码 s 中减去 1，等价于删除掩码 s 中最右边的设置位，并将其右边的所有位变为 1.

为了使 $s-1$ 变为新的子掩码，需要删除掩码 m 中未包含的所有额外的 1 位，可以使用位运算 `(s - 1) & m` 来进行此移除.

这两步操作等价于切割掩码 $s-1$，以确定算术上可以取到的最大值，即按降序排列的 s 之后的下一个子掩码.

因此，该算法按降序生成该掩码的所有子掩码，每次迭代仅执行两个操作.

特殊情况是 $s=0$. 在执行 $s-1$ 之后得到 -1，其中所有位都为 1. 在 `(s - 1) & m` 操作之后将得到新的 s 等于 m. 因此，如果循环不以 $s=0$ 结束，算法的循环将无法终止.

使用 $\text{popcount}(m)$ 表示 m 二进制中 1 的个数，用这种方法可以在 $O(2^{\text{popcount}(m)})$ 的时间复杂度内遍历集合 m 的子集.

遍历所有掩码的子掩码

在使用状压 DP 的问题中，有时会希望对于每个掩码，遍历掩码的所有子掩码：

```
for (int m = 0; m < (1 << n); ++m)
  // 降序遍历 m 的非空子集
  for (int s = m; s; s = (s - 1) & m)
// s 是 m 的一个非空子集
```

这样做可以遍历大小为 n 的集合的每个子集的子集.

接下来证明，该操作的时间复杂度为 $O(3^n)$，n 为掩码总共的位数，即集合中元素的总数.

考虑第 i 位，即集合中第 i 个元素，有三种情况：

- 在掩码 m 中为 0，因此在子掩码 s 中为 0，即元素不在大小子集中.
- 在 m 中为 1，但在 s 中为 0，即元素只在大子集中，不在小子集中.
- 在 m 和 s 中均为 1，即元素同时在大小子集中.

总共有 n 位，因此有 3^n 个不同的组合.

还有一种证明方法是：

如果掩码 m 具有 k 个 1，那么它有 2^k 个子掩码. 对于给定的 k，对应有 $\binom{n}{k}$ 个掩码 m，那么所有掩码的总数为：

$$\sum_{k=0}^{n}\binom{n}{k}2^k$$

上面的和等于使用二项式定理对 $(1+2)^n$ 的展开，因此有 3^n 个不同的组合.

参考资料

本页面主要译自博文 Перебор всех подмасок данной маски[1] 与其英文翻译版 Submask Enumeration[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.

习题

- Atcoder - Close Group[3]
- Codeforces - Nuclear Fusion[4]
- Codeforces - Sandy and Nuts[5]
- UVa 1439 - Exclusive Access 2[6]
- UVa 11825 - Hackers' Crackdown[7]

外部参考资料

[1] Перебор всех подмасок данной маски 

[2] Submask Enumeration 

[3] Atcoder - Close Group 

[4] Codeforces - Nuclear Fusion 

[5] Codeforces - Sandy and Nuts 

[6] UVa 1439 - Exclusive Access 2 

[7] UVa 11825 - Hackers' Crackdown 

9.6 高精度计算

> 太长不看版：结尾自取模板……

定义

高精度计算（Arbitrary-Precision Arithmetic），也被称作大整数（bignum）计算，运用了一些算法结构来支持更大整数间的运算（数字大小超过语言内建整型）.

引入

高精度问题包含很多小的细节，实现上也有很多讲究.

所以今天就来一起实现一个简单的计算器吧.

任务

输入：一个形如 `a <op> b` 的表达式.

- `a`、`b` 分别是长度不超过 1000 的十进制非负整数;
- `<op>` 是一个字符（`+`、`-`、`*` 或 `/`），表示运算.
- 整数与运算符之间由一个空格分隔.

输出：运算结果.

- 对于 `+`、`-`、`*` 运算，输出一行表示结果;
- 对于 `/` 运算，输出两行分别表示商和余数.
- 保证结果均为非负整数.

存储

在平常的实现中，高精度数字利用字符串表示，每一个字符表示数字的一个十进制位. 因此可以说，高精度数值计算实际上是一种特别的字符串处理.

读入字符串时，数字最高位在字符串首（下标小的位置）. 但是习惯上，下标最小的位置存放的是数字的 **最低位**，即存储反转的字符串. 这么做的原因在于，数字的长度可能发生变化，但我们希望同样权值位始终保持对齐（例如，希望所有的个位都在下标 `[0]`，所有的十位都在下标 `[1]`……）；同时，加、减、乘的运算一般都从个位开始进行（回想小学的竖式运算），这都给了「反转存储」以充分的理由.

此后我们将一直沿用这一约定．定义一个常数 `LEN = 1004` 表示程序所容纳的最大长度．

由此不难写出读入高精度数字的代码：

```
void clear(int a[]) {
  for (int i = 0; i < LEN; ++i) a[i] = 0;
}

void read(int a[]) {
  static char s[LEN + 1];
  scanf("%s", s);

  clear(a);

  int len = strlen(s);
  // 如上所述，反转
  for (int i = 0; i < len; ++i) a[len - i - 1] = s[i] - '0';
  // s[i] - '0' 就是 s[i] 所表示的数码
  // 有些同学可能更习惯用 ord(s[i]) - ord('0') 的方式理解
}
```

输出也按照存储的逆序输出．由于不希望输出前导零，故这里从最高位开始向下寻找第一个非零位，从此处开始输出；终止条件 `i >= 1` 而不是 `i >= 0` 是因为当整个数字等于 0 时仍希望输出一个字符 `0`．

```
void print(int a[]) {
  int i;
  for (i = LEN - 1; i >= 1; --i)
    if (a[i] != 0) break;
  for (; i >= 0; --i) putchar(a[i] + '0');
  putchar('\n');
}
```

拼起来就是一个完整的复读机程序咯．

copycat.cpp

```
#include <cstdio>
#include <cstring>

constexpr int LEN = 1004;

int a[LEN];

void clear(int a[]) {
  for (int i = 0; i < LEN; ++i) a[i] = 0;
}

void read(int a[]) {
  static char s[LEN + 1];
  scanf("%s", s);

  clear(a);

  int len = strlen(s);
  for (int i = 0; i < len; ++i) a[len - i - 1] = s[i] - '0';
```

```
}

void print(int a[]) {
  int i;
  for (i = LEN - 1; i >= 1; --i)
    if (a[i] != 0) break;
  for (; i >= 0; --i) putchar(a[i] + '0');
  putchar('\n');
}

int main() {
  read(a);
  print(a);

  return 0;
}
```

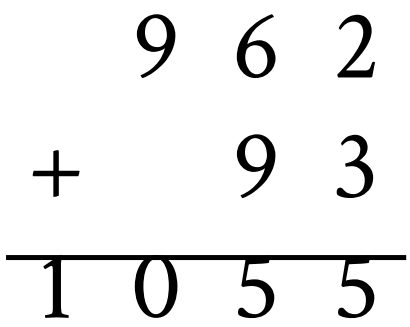
四则运算

四则运算中难度也各不相同．最简单的是高精度加减法，其次是高精度—单精度（普通的 `int`）乘法和高精度—高精度乘法，最后是高精度—高精度除法．

我们将按这个顺序分别实现所有要求的功能．

加法

高精度加法，其实就是竖式加法啦．



也就是从最低位开始，将两个加数对应位置上的数码相加，并判断是否达到或超过 10．如果达到，那么处理进位：将更高一位的结果上增加 1，当前位的结果减少 10．

```
void add(int a[], int b[], int c[]) {
  clear(c);

  // 高精度实现中，一般令数组的最大长度 LEN 比可能的输入大一些
  // 然后略去末尾的几次循环，这样一来可以省去不少边界情况的处理
  // 因为实际输入不会超过 1000 位，故在此循环到 LEN - 1 = 1003 已经足够
  for (int i = 0; i < LEN - 1; ++i) {
    // 将相应位上的数码相加
```

```
    c[i] += a[i] + b[i];
    if (c[i] >= 10) {
      // 进位
      c[i + 1] += 1;
      c[i] -= 10;
    }
  }
}
```

试着和上一部分结合，可以得到一个加法计算器.

adder.cpp

```cpp
#include <cstdio>
#include <cstring>

constexpr int LEN = 1004;

int a[LEN], b[LEN], c[LEN];

void clear(int a[]) {
  for (int i = 0; i < LEN; ++i) a[i] = 0;
}

void read(int a[]) {
  static char s[LEN + 1];
  scanf("%s", s);

  clear(a);

  int len = strlen(s);
  for (int i = 0; i < len; ++i) a[len - i - 1] = s[i] - '0';
}

void print(int a[]) {
  int i;
  for (i = LEN - 1; i >= 1; --i)
    if (a[i] != 0) break;
  for (; i >= 0; --i) putchar(a[i] + '0');
  putchar('\n');
}

void add(int a[], int b[], int c[]) {
  clear(c);

  for (int i = 0; i < LEN - 1; ++i) {
    c[i] += a[i] + b[i];
    if (c[i] >= 10) {
      c[i + 1] += 1;
      c[i] -= 10;
    }
  }
}

```

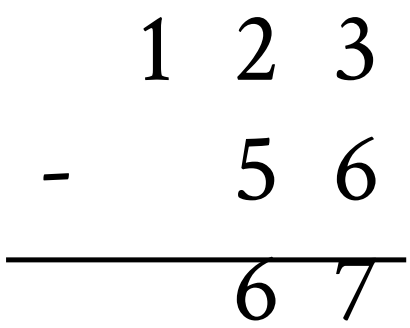
```cpp
int main() {
  read(a);
  read(b);

  add(a, b, c);
  print(c);

  return 0;
}
```

减法

高精度减法，也就是竖式减法啦.



从个位起逐位相减，遇到负的情况则向上一位借 1. 整体思路与加法完全一致.

```cpp
void sub(int a[], int b[], int c[]) {
  clear(c);

  for (int i = 0; i < LEN - 1; ++i) {
    // 逐位相减
    c[i] += a[i] - b[i];
    if (c[i] < 0) {
      // 借位
      c[i + 1] -= 1;
      c[i] += 10;
    }
  }
}
```

将上一个程序中的 `add()` 替换成 `sub()`，就有了一个减法计算器.

subtractor.cpp

```cpp
#include <cstdio>
#include <cstring>

constexpr int LEN = 1004;
```

```
5
6  int a[LEN], b[LEN], c[LEN];
7
8  void clear(int a[]) {
9    for (int i = 0; i < LEN; ++i) a[i] = 0;
10 }
11
12 void read(int a[]) {
13   static char s[LEN + 1];
14   scanf("%s", s);
15
16   clear(a);
17
18   int len = strlen(s);
19   for (int i = 0; i < len; ++i) a[len - i - 1] = s[i] - '0';
20 }
21
22 void print(int a[]) {
23   int i;
24   for (i = LEN - 1; i >= 1; --i)
25     if (a[i] != 0) break;
26   for (; i >= 0; --i) putchar(a[i] + '0');
27   putchar('\n');
28 }
29
30 void sub(int a[], int b[], int c[]) {
31   clear(c);
32
33   for (int i = 0; i < LEN - 1; ++i) {
34     c[i] += a[i] - b[i];
35     if (c[i] < 0) {
36       c[i + 1] -= 1;
37       c[i] += 10;
38     }
39   }
40 }
41
42 int main() {
43   read(a);
44   read(b);
45
46   sub(a, b, c);
47   print(c);
48
49   return 0;
50 }
```

试一试，输入 `1 2`——输出 `/9999999`，诶这个 OI Wiki 怎么给了我一份假的代码啊……

事实上，上面的代码只能处理减数 a 大于等于被减数 b 的情况．处理被减数比减数小，即 $a < b$ 时的情况很简单．

$a - b = -(b - a)$

要计算 $b-a$ 的值，因为有 $b>a$，可以调用以上代码中的 `sub` 函数，写法为 `sub(b,a,c)`. 要得到 $a-b$ 的值，在得数前加上负号即可.

乘法

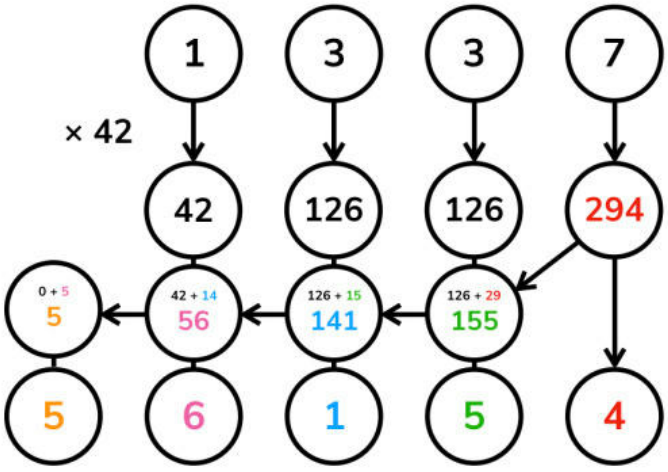
高精度—单精度

高精度乘法，也就是竖……等会儿等会儿!

先考虑一个简单的情况：乘数中的一个是普通的 `int` 类型. 有没有简单的处理方法呢?

一个直观的思路是直接将 a 每一位上的数字乘以 b. 从数值上来说，这个方法是正确的，但它并不符合十进制表示法，因此需要将它重新整理成正常的样子.

重整的方式，也是从个位开始逐位向上处理进位. 但是这里的进位可能非常大，甚至远大于 9，因为每一位被乘上之后都可能达到 $9b$ 的数量级. 所以这里的进位不能再简单地进行 -10 运算，而是要通过除以 10 的商以及余数计算. 详见代码注释，也可以参考下图展示的一个计算高精度数 1337 乘以单精度数 42 的过程.



当然，也是出于这个原因，这个方法需要特别关注乘数 b 的范围. 若它和 10^9（或相应整型的取值上界）属于同一数量级，那么需要慎用高精度—单精度乘法.

```
void mul_short(int a[], int b, int c[]) {
  clear(c);

  for (int i = 0; i < LEN - 1; ++i) {
    // 直接把 a 的第 i 位数码乘以乘数，加入结果
    c[i] += a[i] * b;

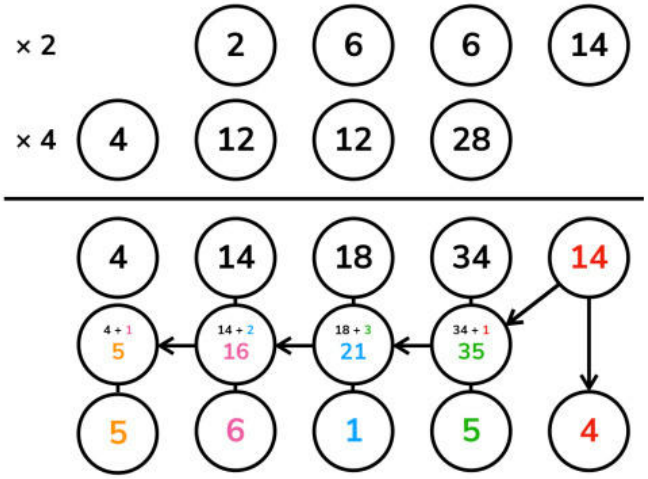
    if (c[i] >= 10) {
      // 处理进位
      // c[i] / 10 即除法的商数成为进位的增量值
      c[i + 1] += c[i] / 10;
      // 而 c[i] % 10 即除法的余数成为在当前位留下的值
      c[i] %= 10;
    }
  }
}
```

高精度—高精度

如果两个乘数都是高精度，那么竖式乘法又可以大显身手了.

回想竖式乘法的每一步，实际上是计算了若干 $a \times b_i \times 10^i$ 的和．例如计算 1337×42，计算的就是 $1337 \times 2 \times 10^0 + 1337 \times 4 \times 10^1$．

于是可以将 b 分解为它的所有数码，其中每个数码都是单精度数，将它们分别与 a 相乘，再向左移动到各自的位置上相加即得答案．当然，最后也需要用与上例相同的方式处理进位．

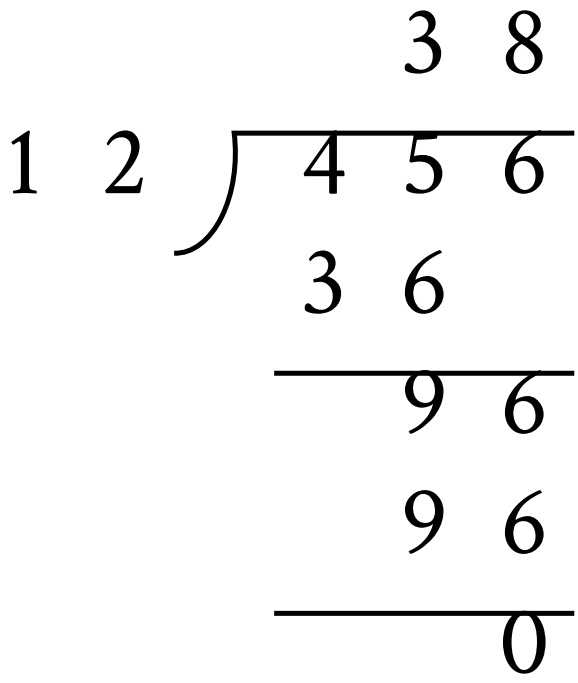


注意这个过程与竖式乘法不尽相同，我们的算法在每一步乘的过程中并不进位，而是将所有的结果保留在对应的位置上，到最后再统一处理进位，但这不会影响结果．

```
1  void mul(int a[], int b[], int c[]) {
2    clear(c);
3
4    for (int i = 0; i < LEN - 1; ++i) {
5      // 这里直接计算结果中的从低到高第 i 位，且一并处理了进位
6      // 第 i 次循环为 c[i] 加上了所有满足 p + q = i 的 a[p] 与 b[q] 的乘积之和
7      // 这样做的效果和直接进行上图的运算最后求和是一样的，只是更加简短的一种实现方式
8      for (int j = 0; j <= i; ++j) c[i] += a[j] * b[i - j];
9
10     if (c[i] >= 10) {
11       c[i + 1] += c[i] / 10;
12       c[i] %= 10;
13     }
14   }
15 }
```

除法

高精度除法的一种实现方式就是竖式长除法．



竖式长除法实际上可以看作一个逐次减法的过程. 例如上图中商数十位的计算可以这样理解: 将 45 减去三次 12 后变得小于 12，不能再减，故此位为 3.

为了减少冗余运算，我们提前得到被除数的长度 l_a 与除数的长度 l_b，从下标 $l_a - l_b$ 开始，从高位到低位来计算商. 这和手工计算时将第一次乘法的最高位与被除数最高位对齐的做法是一样的.

参考程序实现了一个函数`greater_eq()`用于判断被除数以下标`last_dg`为最低位, 是否可以再减去除数而保持非负. 此后对于商的每一位，不断调用`greater_eq()`，并在成立的时候用高精度减法从余数中减去除数，也即模拟了竖式除法的过程.

```
// 被除数 a 以下标 last_dg 为最低位，是否可以再减去除数 b 而保持非负
// len 是除数 b 的长度，避免反复计算
bool greater_eq(int a[], int b[], int last_dg, int len) {
  // 有可能被除数剩余的部分比除数长，这个情况下最多多出 1 位，故如此判断即可
  if (a[last_dg + len] != 0) return true;
  // 从高位到低位，逐位比较
  for (int i = len - 1; i >= 0; --i) {
    if (a[last_dg + i] > b[i]) return true;
    if (a[last_dg + i] < b[i]) return false;
  }
  // 相等的情形下也是可行的
  return true;
}

void div(int a[], int b[], int c[], int d[]) {
  clear(c);
  clear(d);

  int la, lb;
  for (la = LEN - 1; la > 0; --la)
```

```
    if (a[la - 1] != 0) break;
  for (lb = LEN - 1; lb > 0; --lb)
    if (b[lb - 1] != 0) break;
  if (lb == 0) {  // 除数不能为零
    puts("> <");
    return;
  }

  // c 是商
  // d 是被除数的剩余部分，算法结束后自然成为余数
  for (int i = 0; i < la; ++i) d[i] = a[i];
  for (int i = la - lb; i >= 0; --i) {
    // 计算商的第 i 位
    while (greater_eq(d, b, i, lb)) {
      // 若可以减，则减
      // 这一段是一个高精度减法
      for (int j = 0; j < lb; ++j) {
        d[i + j] -= b[j];
        if (d[i + j] < 0) {
          d[i + j + 1] -= 1;
          d[i + j] += 10;
        }
      }
      // 使商的这一位增加 1
      c[i] += 1;
      // 返回循环开头，重新检查
    }
  }
}
```

入门篇完成！

将上面介绍的四则运算的实现结合，即可完成开头提到的计算器程序.

calculator.cpp

```
#include <cstdio>
#include <cstring>

constexpr int LEN = 1004;

int a[LEN], b[LEN], c[LEN], d[LEN];

void clear(int a[]) {
  for (int i = 0; i < LEN; ++i) a[i] = 0;
}

void read(int a[]) {
  static char s[LEN + 1];
  scanf("%s", s);

  clear(a);

```

```
  int len = strlen(s);
  for (int i = 0; i < len; ++i) a[len - i - 1] = s[i] - '0';
}

void print(int a[]) {
  int i;
  for (i = LEN - 1; i >= 1; --i)
    if (a[i] != 0) break;
  for (; i >= 0; --i) putchar(a[i] + '0');
  putchar('\n');
}

void add(int a[], int b[], int c[]) {
  clear(c);

  for (int i = 0; i < LEN - 1; ++i) {
    c[i] += a[i] + b[i];
    if (c[i] >= 10) {
      c[i + 1] += 1;
      c[i] -= 10;
    }
  }
}

void sub(int a[], int b[], int c[]) {
  clear(c);

  for (int i = 0; i < LEN - 1; ++i) {
    c[i] += a[i] - b[i];
    if (c[i] < 0) {
      c[i + 1] -= 1;
      c[i] += 10;
    }
  }
}

void mul(int a[], int b[], int c[]) {
  clear(c);

  for (int i = 0; i < LEN - 1; ++i) {
    for (int j = 0; j <= i; ++j) c[i] += a[j] * b[i - j];

    if (c[i] >= 10) {
      c[i + 1] += c[i] / 10;
      c[i] %= 10;
    }
  }
}

bool greater_eq(int a[], int b[], int last_dg, int len) {
  if (a[last_dg + len] != 0) return true;
  for (int i = len - 1; i >= 0; --i) {
```

```
    if (a[last_dg + i] > b[i]) return true;
    if (a[last_dg + i] < b[i]) return false;
  }
  return true;
}

void div(int a[], int b[], int c[], int d[]) {
  clear(c);
  clear(d);

  int la, lb;
  for (la = LEN - 1; la > 0; --la)
    if (a[la - 1] != 0) break;
  for (lb = LEN - 1; lb > 0; --lb)
    if (b[lb - 1] != 0) break;
  if (lb == 0) {
    puts("> <");
    return;
  }

  for (int i = 0; i < la; ++i) d[i] = a[i];
  for (int i = la - lb; i >= 0; --i) {
    while (greater_eq(d, b, i, lb)) {
      for (int j = 0; j < lb; ++j) {
        d[i + j] -= b[j];
        if (d[i + j] < 0) {
          d[i + j + 1] -= 1;
          d[i + j] += 10;
        }
      }
      c[i] += 1;
    }
  }
}

int main() {
  read(a);

  char op[4];
  scanf("%s", op);

  read(b);

  switch (op[0]) {
    case '+':
      add(a, b, c);
      print(c);
      break;
    case '-':
      sub(a, b, c);
      print(c);
      break;
```

```
122     case '*':
123       mul(a, b, c);
124       print(c);
125       break;
126     case '/':
127       div(a, b, c, d);
128       print(c);
129       print(d);
130       break;
131     default:
132       puts("> <");
133   }
134
135   return 0;
136 }
```

压位高精度

引入

在一般的高精度加法，减法，乘法运算中，我们都是将参与运算的数拆分成一个个单独的数码进行运算.

例如计算 8192×42 时，如果按照高精度乘高精度的计算方式，我们实际上算的是 $(8000 + 100 + 90 + 2) \times (40 + 2)$.

在位数较多的时候，拆分出的数也很多，高精度运算的效率就会下降.

有没有办法作出一些优化呢?

注意到拆分数字的方式并不影响最终的结果，因此我们可以将若干个数码进行合并.

过程

还是以上面这个例子为例，如果我们每两位拆分一个数，我们可以拆分成 $(8100 + 92) \times 42$.

这样的拆分不影响最终结果，但是因为拆分出的数字变少了，计算效率也就提升了.

从 进位制→9.3.2@p1087 的角度理解这一过程，我们通过在较大的进位制（上面每两位拆分一个数，可以认为是在 100 进制下进行运算）下进行运算，从而达到减少参与运算的数字的位数，提升运算效率的目的.

这就是 **压位高精度** 的思想.

下面我们给出压位高精度的加法代码，用于进一步阐述其实现方法:

压位高精度加法参考实现

```
// 这里的 a,b,c 数组均为 p 进制下的数
// 最终输出答案时需要将数字转为十进制
void add(int a[], int b[], int c[]) {
  clear(c);

  for (int i = 0; i < LEN - 1; ++i) {
    c[i] += a[i] + b[i];
    if (c[i] >= p) {  // 在普通高精度运算下，p=10
      c[i + 1] += 1;
```

```
      c[i] -= p;
    }
  }
}
```

压位高精下的高效竖式除法

在使用压位高精时，如果试商时仍然使用上文介绍的方法，由于试商次数会很多，计算常数会非常大. 例如在万进制下，平均每个位需要试商 5000 次，这个巨大的常数是不可接受的. 因此我们需要一个更高效的试商办法.

我们可以把 double 作为媒介. 假设被除数有 4 位，是 a_4, a_3, a_2, a_1，除数有 3 位，是 b_3, b_2, b_1，那么我们只要试一位的商：使用 $base$ 进制，用式子 $\frac{a_4 base + a_3}{b_3 + b_2 base^{-1} + (b_1 + 1) base^{-2}}$ 来估商. 而对于多个位的情况，就是一位的写法加个循环. 由于除数使用 3 位的精度来参与估商，能保证估的商 q' 与实际商 q 的关系满足 $q - 1 \le q' \le q$，这样每个位在最坏的情况下也只需要两次试商. 但与此同时要求 $base^3$ 在 double 的有效精度内，即 $base^3 < 2^{53}$，所以在运用这个方法时建议不要超过 32768 进制，否则很容易因精度不足产生误差从而导致错误.

另外，由于估的商总是小于等于实际商，所以还有再进一步优化的空间. 绝大多数情况下每个位只估商一次，这样在下一个位估商时，虽然得到的商有可能因为前一位的误差造成试商结果大于等于 base，但这没有关系，只要在最后做统一进位便可. 举个例子，假设 base 是 10，求 395081/9876，试商计算步骤如下：

1. 首先试商计算得到 $3950/988 = 3$，于是 $395081 - (9876 \times 3 \times 10^1) = 98801$，这一步出现了误差，但不用管，继续下一步计算.
2. 对余数 98801 继续试商计算得到 $9880/988 = 10$，于是 $98801 - (9876 \times 10 \times 10^0) = 41$，这就是最终余数.
3. 把试商过程的结果加起来并处理进位，即 $3 \times 10^1 + 10 \times 10^0 = 40$ 便是准确的商.

方法虽然看着简单，但具体实现上很容易进坑，所以以下提供一个经过多番验证确认没有问题的实现供大家参考，要注意的细节也写在注释当中.

压位高精度高效竖式除法参考实现

```
// 完整模板和实现 https://baobaobear.github.io/post/20210228-bigint1/
// 对b乘以mul再左移offset的结果相减，为除法服务
BigIntSimple &sub_mul(const BigIntSimple &b, int mul, int offset) {
  if (mul == 0) return *this;
  int borrow = 0;
  // 与减法不同的是，borrow可能很大，不能使用减法的写法
  for (size_t i = 0; i < b.v.size(); ++i) {
    borrow += v[i + offset] - b.v[i] * mul - BIGINT_BASE + 1;
    v[i + offset] = borrow % BIGINT_BASE + BIGINT_BASE - 1;
    borrow /= BIGINT_BASE;
  }
  // 如果还有借位就继续处理
  for (size_t i = b.v.size(); borrow; ++i) {
    borrow += v[i + offset] - BIGINT_BASE + 1;
    v[i + offset] = borrow % BIGINT_BASE + BIGINT_BASE - 1;
    borrow /= BIGINT_BASE;
  }
  return *this;
```

```
}

BigIntSimple div_mod(const BigIntSimple &b, BigIntSimple &r) const {
  BigIntSimple d;
  r = *this;
  if (absless(b)) return d;
  d.v.resize(v.size() - b.v.size() + 1);
  // 提前算好除数的最高三位+1的倒数，若最高三位是a3,a2,a1
  // 那么db是a3+a2/base+(a1+1)/base^2的倒数，最后用乘法估商的每一位
  // 此法在BIGINT_BASE<=32768时可在int32范围内用
  // 但即使使用int64，那么也只有BIGINT_BASE<=131072时可用（受double的精度限制）
  // 能保证估计结果q'与实际结果q的关系满足q'<=q<=q'+1
  // 所以每一位的试商平均只需要一次，只要后面再统一处理进位即可
  // 如果要使用更大的base，那么需要更换其它试商方案
  double t = (b.get((unsigned)b.v.size() - 2) +
              (b.get((unsigned)b.v.size() - 3) + 1.0) / BIGINT_BASE);
  double db = 1.0 / (b.v.back() + t / BIGINT_BASE);
  for (size_t i = v.size() - 1, j = d.v.size() - 1; j <= v.size();) {
    int rm = r.get(i + 1) * BIGINT_BASE + r.get(i);
    int m = std::max((int)(db * rm), r.get(i + 1));
    r.sub_mul(b, m, j);
    d.v[j] += m;
    if (!r.get(i + 1))  // 检查最高位是否已为0，避免极端情况
      --i, --j;
  }
  r.trim();
  // 修正结果的个位
  int carry = 0;
  while (!r.absless(b)) {
    r.subtract(b);
    ++carry;
  }
  // 修正每一位的进位
  for (size_t i = 0; i < d.v.size(); ++i) {
    carry += d.v[i];
    d.v[i] = carry % BIGINT_BASE;
    carry /= BIGINT_BASE;
  }
  d.trim();
  d.sign = sign * b.sign;
  return d;
}

BigIntSimple operator/(const BigIntSimple &b) const {
  BigIntSimple r;
  return div_mod(b, r);
}

BigIntSimple operator%(const BigIntSimple &b) const {
  BigIntSimple r;
  div_mod(b, r);
  return r;
```

```
71 }
```

Karatsuba 乘法

记高精度数字的位数为 n，那么高精度—高精度竖式乘法需要花费 $O(n^2)$ 的时间．本节介绍一个时间复杂度更为优秀的算法，由前苏联（俄罗斯）数学家 Anatoly Karatsuba 提出，是一种分治算法．

考虑两个十进制大整数 x 和 y，均包含 n 个数码（可以有前导零）．任取 $0 < m < n$，记

$$x = x_1 \cdot 10^m + x_0,$$

$$y = y_1 \cdot 10^m + y_0,$$

$$x \cdot y = z_2 \cdot 10^{2m} + z_1 \cdot 10^m + z_0,$$

其中 $x_0, y_0, z_0, z_1 < 10^m$．可得

$$z_2 = x_1 \cdot y_1,$$

$$z_1 = x_1 \cdot y_0 + x_0 \cdot y_1,$$

$$z_0 = x_0 \cdot y_0.$$

观察知

$$z_1 = (x_1 + x_0) \cdot (y_1 + y_0) - z_2 - z_0,$$

于是要计算 z_1，只需计算 $(x_1 + x_0) \cdot (y_1 + y_0)$，再与 z_0、z_2 相减即可．

上式实际上是 Karatsuba 算法的核心，它将长度为 n 的乘法问题转化为了 3 个长度更小的子问题．若令 $m = \left\lceil \frac{n}{2} \right\rceil$，记 Karatsuba 算法计算两个 n 位整数乘法的耗时为 $T(n)$，则有 $T(n) = 3 \cdot T\left(\left\lceil \frac{n}{2} \right\rceil\right) + O(n)$，由主定理可得 $T(n) = \Theta(n^{\log_2 3}) \approx \Theta(n^{1.585})$．

整个过程可以递归实现．为清晰起见，下面的代码通过 Karatsuba 算法实现了多项式乘法，最后再处理所有的进位问题．

karatsuba_mulc.cpp

```cpp
int *karatsuba_polymul(int n, int *a, int *b) {
  if (n <= 32) {
    // 规模较小时直接计算，避免继续递归带来的效率损失
    int *r = new int[n * 2 + 1]();
    for (int i = 0; i <= n; ++i)
      for (int j = 0; j <= n; ++j) r[i + j] += a[i] * b[j];
    return r;
  }

  int m = n / 2 + 1;
  int *r = new int[m * 4 + 1]();
  int *z0, *z1, *z2;

  z0 = karatsuba_polymul(m - 1, a, b);
  z2 = karatsuba_polymul(n - m, a + m, b + m);

  // 计算 z1
  // 临时更改，计算完毕后恢复
  for (int i = 0; i + m <= n; ++i) a[i] += a[i + m];
  for (int i = 0; i + m <= n; ++i) b[i] += b[i + m];
  z1 = karatsuba_polymul(m - 1, a, b);
```

```
  for (int i = 0; i + m <= n; ++i) a[i] -= a[i + m];
  for (int i = 0; i + m <= n; ++i) b[i] -= b[i + m];
  for (int i = 0; i <= (m - 1) * 2; ++i) z1[i] -= z0[i];
  for (int i = 0; i <= (n - m) * 2; ++i) z1[i] -= z2[i];

  // 由 z0、z1、z2 组合获得结果
  for (int i = 0; i <= (m - 1) * 2; ++i) r[i] += z0[i];
  for (int i = 0; i <= (m - 1) * 2; ++i) r[i + m] += z1[i];
  for (int i = 0; i <= (n - m) * 2; ++i) r[i + m * 2] += z2[i];

  delete[] z0;
  delete[] z1;
  delete[] z2;
  return r;
}

void karatsuba_mul(int a[], int b[], int c[]) {
  int *r = karatsuba_polymul(LEN - 1, a, b);
  memcpy(c, r, sizeof(int) * LEN);
  for (int i = 0; i < LEN - 1; ++i)
    if (c[i] >= 10) {
      c[i + 1] += c[i] / 10;
      c[i] %= 10;
    }
  delete[] r;
}
```

关于 new 和 delete

见 内存池→2.6.4@p120.

但是这样的实现存在一个问题：在 b 进制下，多项式的每一个系数都有可能达到 $n \cdot b^2$ 量级，在压位高精度实现中可能造成整数溢出；而若在多项式乘法的过程中处理进位问题，则 $x_1 + x_0$ 与 $y_1 + y_0$ 的结果可能达到 $2 \cdot b^m$，增加一个位（如果采用 $x_1 - x_0$ 的计算方式，则不得不特殊处理负数的情况）．因此，需要依照实际的应用场景来决定采用何种实现方式．

基于多项式的高效大整数乘法

如果数据规模达到了 10^{10^5} 或更大，普通的高精度乘法可能会超时．本节将介绍用多项式优化此类乘法的方法．

对于一个 n 位的十进制整数 a，可以将它看作一个每位系数均为整数且不超过 10 的多项式 $A = a_0 10^0 + a_1 10^1 + \cdots + a_{n-1} 10^{n-1}$．这样，我们就将两个整数乘法转化为了两个多项式乘法．

普通的多项式乘法时间复杂度仍是 $O(n^2)$，但可以用多项式一节中的 快速傅里叶变换→9.12.3@p1527、快速数论变换→9.12.4@p1539 等算法优化，优化后的时间复杂度是 $O(n \log n)$．

封装类

这里[1] 有一个封装好的高精度整数类，以及 这里[2] 支持动态长度及四则运算的超迷你实现类．

这里是另一个模板

```cpp
constexpr int MAXN = 9999;
// MAXN 是一位中最大的数字
constexpr int MAXSIZE = 10024;
// MAXSIZE 是位数
constexpr int DLEN = 4;

// DLEN 记录压几位
struct Big {
  int a[MAXSIZE], len;
  bool flag;  // 标记符号'-'

  Big() {
    len = 1;
    memset(a, 0, sizeof a);
    flag = false;
  }

  Big(const int);
  Big(const char*);
  Big(const Big&);
  Big& operator=(const Big&);
  Big operator+(const Big&) const;
  Big operator-(const Big&) const;
  Big operator*(const Big&) const;
  Big operator/(const int&) const;
  // TODO: Big / Big;
  Big operator^(const int&) const;
  // TODO: Big ^ Big;

  // TODO: Big 位运算;

  int operator%(const int&) const;
  // TODO: Big ^ Big;
  bool operator<(const Big&) const;
  bool operator<(const int& t) const;
  void print() const;
};

Big::Big(const int b) {
  int c, d = b;
  len = 0;
  // memset(a,0,sizeof a);
  CLR(a);
  while (d > MAXN) {
    c = d - (d / (MAXN + 1)) * (MAXN + 1));
    d = d / (MAXN + 1);
    a[len++] = c;
  }
  a[len++] = d;
}

Big::Big(const char* s) {
```

```
  int t, k, index, l;
  CLR(a);
  l = strlen(s);
  len = l / DLEN;
  if (l % DLEN) ++len;
  index = 0;
  for (int i = l - 1; i >= 0; i -= DLEN) {
    t = 0;
    k = i - DLEN + 1;
    if (k < 0) k = 0;
    g(j, k, i) t = t * 10 + s[j] - '0';
    a[index++] = t;
  }
}

Big::Big(const Big& T) : len(T.len) {
  CLR(a);
  f(i, 0, len) a[i] = T.a[i];
  // TODO:重载此处?
}

Big& Big::operator=(const Big& T) {
  CLR(a);
  len = T.len;
  f(i, 0, len) a[i] = T.a[i];
  return *this;
}

Big Big::operator+(const Big& T) const {
  Big t(*this);
  int big = len;
  if (T.len > len) big = T.len;
  f(i, 0, big) {
    t.a[i] += T.a[i];
    if (t.a[i] > MAXN) {
      ++t.a[i + 1];
      t.a[i] -= MAXN + 1;
    }
  }
  if (t.a[big])
    t.len = big + 1;
  else
    t.len = big;
  return t;
}

Big Big::operator-(const Big& T) const {
  int big;
  bool ctf;
  Big t1, t2;
  if (*this < T) {
    t1 = T;
```

```
105     t2 = *this;
106     ctf = true;
107   } else {
108     t1 = *this;
109     t2 = T;
110     ctf = false;
111   }
112   big = t1.len;
113   int j = 0;
114   f(i, 0, big) {
115     if (t1.a[i] < t2.a[i]) {
116       j = i + 1;
117       while (t1.a[j] == 0) ++j;
118       --t1.a[j--];
119       // WTF?
120       while (j > i) t1.a[j--] += MAXN;
121       t1.a[i] += MAXN + 1 - t2.a[i];
122     } else
123       t1.a[i] -= t2.a[i];
124   }
125   t1.len = big;
126   while (t1.len > 1 && t1.a[t1.len - 1] == 0) {
127     --t1.len;
128     --big;
129   }
130   if (ctf) t1.a[big - 1] = -t1.a[big - 1];
131   return t1;
132 }
133
134 Big Big::operator*(const Big& T) const {
135   Big res;
136   int up;
137   int te, tee;
138   f(i, 0, len) {
139     up = 0;
140     f(j, 0, T.len) {
141       te = a[i] * T.a[j] + res.a[i + j] + up;
142       if (te > MAXN) {
143         tee = te - te / (MAXN + 1) * (MAXN + 1);
144         up = te / (MAXN + 1);
145         res.a[i + j] = tee;
146       } else {
147         up = 0;
148         res.a[i + j] = te;
149       }
150     }
151     if (up) res.a[i + T.len] = up;
152   }
153   res.len = len + T.len;
154   while (res.len > 1 && res.a[res.len - 1] == 0) --res.len;
155   return res;
156 }
```

```
157
158 Big Big::operator/(const int& b) const {
159   Big res;
160   int down = 0;
161   gd(i, len - 1, 0) {
162     res.a[i] = (a[i] + down * (MAXN + 1)) / b;
163     down = a[i] + down * (MAXN + 1) - res.a[i] * b;
164   }
165   res.len = len;
166   while (res.len > 1 && res.a[res.len - 1] == 0) --res.len;
167   return res;
168 }
169
170 int Big::operator%(const int& b) const {
171   int d = 0;
172   gd(i, len - 1, 0) d = (d * (MAXN + 1) % b + a[i]) % b;
173   return d;
174 }
175
176 Big Big::operator^(const int& n) const {
177   Big t(*this), res(1);
178   int y = n;
179   while (y) {
180     if (y & 1) res = res * t;
181     t = t * t;
182     y >>= 1;
183   }
184   return res;
185 }
186
187 bool Big::operator<(const Big& T) const {
188   int ln;
189   if (len < T.len) return true;
190   if (len == T.len) {
191     ln = len - 1;
192     while (ln >= 0 && a[ln] == T.a[ln]) --ln;
193     if (ln >= 0 && a[ln] < T.a[ln]) return true;
194     return false;
195   }
196   return false;
197 }
198
199 bool Big::operator<(const int& t) const {
200   Big tee(t);
201   return *this < tee;
202 }
203
204 void Big::print() const {
205   printf("%d", a[len - 1]);
206   gd(i, len - 2, 0) { printf("%04d", a[i]); }
207 }
208
```

```
209 void print(const Big& s) {
210   int len = s.len;
211   printf("%d", s.a[len - 1]);
212   gd(i, len - 2, 0) { printf("%04d", s.a[i]); }
213 }
214
215 char s[100024];
```

习题

- NOIP 2012 国王游戏[3]
- SPOJ - Fast Multiplication[4]
- SPOJ - GCD2[5]
- UVa - Division[6]
- UVa - Fibonacci Freeze[7]
- Codeforces - Notepad[8]

参考资料与链接

1. Karatsuba algorithm - Wikipedia[9]

外部参考资料

[1] 这里 

[2] 这里 

[3] NOIP 2012 国王游戏 

[4] SPOJ - Fast Multiplication 

[5] SPOJ - GCD2 

[6] UVa - Division 

[7] UVa - Fibonacci Freeze 

[8] Codeforces - Notepad 

[9] Karatsuba algorithm - Wikipedia 

9.7 快速幂

autor: iamtwz, billchenchina, CBW2007, CCXXXI, chinggg, Enter-tainer, eyedeng, FFjet, gaojude, Great-designer, H-J-Granger, Henry-ZHR, hsfzLZH1, Ir1d, kenlig, Konano, ksyx, luoguyuntianming, Marcythm, Menci, NachtgeistW, ouuan, Peanut-Tang, qwqAutomaton, sshwy, StudyingFather, Tiphereth-A, TrisolarisHD, TRSWNCA, Xeonacid, Yuuko10032, Zhangjiacheng2006, Zhoier, Hszzzx, shenshuaijie, kfy666

引入

快速幂（fast exponentiation），也称 **二进制取幂**（binary exponentiation）或 **平方取幂法**（exponentiation by squaring），是一个在 $\Theta(\log n)$ 的时间内计算 a^n 的小技巧，而暴力的计算需要 $\Theta(n)$ 的时间.

这个技巧可以应用于任何 a 的乘法满足结合律的场景中，例如模意义下取幂、矩阵幂等，详见后文 应用→9@p1079 一节.

过程

计算 a 的 n 次方表示将 n 个 a 乘在一起：$a^n = \underbrace{a \times a \cdots \times a}_{n \text{ 个 a}}$. 然而当 n 太大或单次乘法开销太大的时候，这种方法就不太适用了. 二进制取幂的想法是，将取幂的任务按照指数的 **二进制表示** 来分割成更小的任务.

例子

假设要计算 3^{13}．如果将它展开为连乘式，需要 $13-1=12$ 次乘法．但是，因为

$$3^{13}=3^{(1101)_2}=3^8\times 3^4\times 3^1,$$

所以，只要能快速计算出 $3^1,3^2,3^4,3^8$，就能通过 2 次乘法计算出 3^{13} 的值．于是，只需要知道一个快速的方法来计算上述 3 的 2^k 次幂的序列．这是容易的，因为序列中（除第一个）任意一个元素都是其前一个元素的平方．

根据这些分析，可以得到 3^{13} 的计算过程如下：

$$\begin{aligned}3^1&=3,\\ 3^2&=\left(3^1\right)^2=3^2=9,\\ 3^4&=\left(3^2\right)^2=9^2=81,\\ 3^8&=\left(3^4\right)^2=81^2=6561,\\ 3^{13}&=6561\times 81\times 3=1594323.\end{aligned}$$

过程中，只进行了 5 次乘法运算．

这就是快速幂的基本想法．至于具体实现，有两种常见的版本．

迭代版本

设 n 的二进制表示为 $(n_tn_{t-1}\cdots n_1n_0)_2$，也就是说，有

$$n=n_t2^t+n_{t-1}2^{t-1}+\cdots+n_12^1+n_02^0,$$

其中，$n_i\in\{0,1\}$．那么，就有

$$\begin{aligned}a^n&=a^{n_t2^t+n_{t-1}2^{t-1}+\cdots+n_12^1+n_02^0}\\ &=a^{n_02^0}\times a^{n_12^1}\times\cdots\times a^{n_{t-1}2^{t-1}}\times a^{n_t2^t}.\end{aligned}$$

注意，只有 $n_i=1$ 的项才会真正出现在乘积的计算中．

根据这一表达式，可以首先在 $\Theta(\log n)$ 时间内计算出 a 的 $\Theta(\log n)$ 个 2^k 次幂的取值，然后花费 $\Theta(\log n)$ 的时间选择等于 1 的二进制位对应的幂次乘到最终结果中．这就是快速幂的迭代版本实现．

伪代码如下：

Algorithm FastPow(a,n) :
Input. Base a and exponent n.
Output. Power a^n.
Method.

```
result ← Id
while n > 0 do
    if n mod 2 = 1 then
        result ← result · a
    end if
    a ← a · a
    n ← n/2
end while
return result
```

利用这一方法计算快速幂，需要进行 $\Theta(\log n)$ 次乘法运算．

递归版本

这一过程同样可以通过递归形式实现．注意到，指数 n 的二进制展开可以递归地写作

$$(n_tn_{t-1}\cdots n_1n_0)_2=2\times(n_tn_{t-1}\cdots n_1)_2+n_0.$$

因此，幂次 a^n 可以递归地计算为

$$a^n = \begin{cases} 1, & n = 0, \\ (a^{\lfloor n/2 \rfloor})^2, & n > 0 \text{ and } n \text{ is even}, \\ (a^{\lfloor n/2 \rfloor})^2 \cdot a, & n > 0 \text{ and } n \text{ is odd}. \end{cases}$$

这就是快速幂的递归版本实现.

伪代码如下:

Algorithm FastPow(a, n) :
Input. Base a and exponent n.
Output. Power a^n.
Method.
1 **if** $n = 0$ **then**
2 **return** Id
3 **end if**
4 $result \leftarrow$ FastPow$(a, n/2)$
5 **if** $n \bmod 2 = 0$ **then**
6 **return** $result \cdot result$
7 **else**
8 **return** $result \cdot result \cdot a$
9 **end if**

利用这一方法计算快速幂，需要递归 $\Theta(\log n)$ 次，同样需要 $\Theta(\log n)$ 次乘法运算. 尽管复杂度相同，由于递归本身有一定开销，所以实践中迭代版本的速度更快.

应用

模意义下取幂

洛谷 P1226【模板】快速幂[1]

给定三个整数 a, b, p，求 $a^b \bmod p$. 其中 $p \geq 2$.

这是一个非常常见的应用，例如它可以用于计算模意义下的乘法逆元. 既然我们知道取模的运算不会干涉乘法运算，因此我们只需要在计算的过程中取模即可.

首先我们可以直接按照上述递归方法实现:

参考实现

- C++

```cpp
long long binpow(long long a, long long b, long long p) {
  if (b == 0) return 1;
  long long res = binpow(a, b / 2, p);
  if (b % 2)
    return res * res % p * a % p;
  else
    return res * res % p;
}

```

- Python

```python
def binpow(a, b, p):
    if b == 0:
        return 1
    res = binpow(a, b // 2, p)
    if (b % 2) == 1:
        return res * res * a % p
```

```
    else:
        return res * res % p

```

第二种实现方法是非递归式的. 它在循环的过程中将二进制位为 1 时对应的幂累乘到答案中. 尽管两者的理论复杂度是相同的, 但第二种在实践过程中的速度是比第一种更快的, 因为递归会花费一定的开销.

参考实现

- C++

```cpp
long long binpow(long long a, long long b, long long p) {
  long long res = 1;
  while (b > 0) {
    if (b & 1) res = res * a % p;
    a = a * a % p;
    b >>= 1;
  }
  return res;
}

```

- Python

```python
def binpow(a, b, p):
    res = 1
    while b > 0:
        if b & 1:
            res = res * a % p
        a = a * a % p
        b >>= 1
    return res

```

注意

- 模数通常情况下大于 1. 在十分特殊的情况下, 模数 p 可能等于 1, 此时需要特殊考虑 $b = 0$ 的情况.
- 当指数很大时, 需利用 扩展欧拉定理→9.11.9@p1223 降幂后计算.

计算斐波那契数

根据斐波那契数列的递推式 $F_n = F_{n-1} + F_{n-2}$, 我们可以构建一个 2×2 的矩阵来表示从 F_i, F_{i+1} 到 F_{i+1}, F_{i+2} 的变换. 于是在计算这个矩阵的 n 次幂的时候, 我们使用快速幂的思想, 可以在 $\Theta(\log n)$ 的时间内计算出结果. 对于更多的细节参见 斐波那契数列→9.13.4@p1629, 矩阵快速幂的实现参见 矩阵加速递推→9.14.4@p1716 中的实现.

多次置换

问题描述

给你一个长度为 n 的序列和一个置换，把这个序列置换 k 次.

简单地把这个置换取 k 次幂，然后把它应用到序列上即可. 时间复杂度为 $O(n \log k)$. 对于更多的细节参见 置换的复合→9.8@p1138.

注意

对这个置换建图，然后在每一个环上分别做 k 次幂（事实上等价于 k 对环长取模），可以在 $O(n)$ 的时间复杂度下解决此问题.

加速几何中对点集的操作

HDU 4087 A Letter to Programmers[2]

给定三维空间中 n 个点 p_i，要求将 m 个操作都应用于这些点. 包含 3 种操作:

1. 沿某个向量移动点的位置（Shift）.
2. 按比例缩放这个点的坐标（Scale）.
3. 绕某条直线旋转（Rotate）.

还有一个特殊的操作，就是将某个操作序列重复 k 次（Repeat），Repeat 操作可以嵌套. 输出操作结束后每个点的坐标.

参考 向量与矩阵→9.14.2@p1707 中的内容，每一种操作都可以用一个变换矩阵表示，一系列连续的变换可以用矩阵的乘积来表示. 一个 Repeat 操作就相当于取一个矩阵的 k 次幂. 这样可以用 $O(m \log k)$ 的时间计算出整个变换序列最终形成的矩阵. 最后将它应用到 n 个点上，总复杂度 $O(n + m \log k)$.

定长路径计数

问题描述

给一个有向图（边权为 1），求任意两点 u, v 间从 u 到 v，长度为 k 的路径的条数.

我们把该图的邻接矩阵 M 取 k 次幂，那么 $M_{i,j}$ 就表示从 i 到 j 长度为 k 的路径的数目. 该算法的复杂度是 $O(n^3 \log k)$. 有关该算法的细节请参见 矩阵→9.14.4@p1716 页面.

模意义下的整数乘法

问题描述

给定非负整数 a, b 和正整数 m，计算 $a \times b \bmod m$，其中 $a, b \leq m \leq 10^{18}$.

与二进制取幂的思想一样，这次我们将其中的一个乘数表示为若干个 2 的整数次幂的和的形式. 因为在对一个数做乘 2 并取模的运算的时候，我们可以转化为加减操作防止整型溢出. 这样可以在 $O(\log m)$ 的时间复杂度下解决问题. 递归方法如下:

$$a \cdot b = \begin{cases} 0 & \text{if } a = 0 \\ 2 \cdot \frac{a}{2} \cdot b & \text{if } a > 0 \text{ and } a \text{ even} \\ 2 \cdot \frac{a-1}{2} \cdot b + b & \text{if } a > 0 \text{ and } a \text{ odd} \end{cases}$$

但在实际使用中，此方法由于引入了更大的计算复杂度导致时间效率不优. 实际编程中通常利用 快速乘→9.11.2@p1169 来进行模数范围在 `long long` 时的乘法操作.

高精度快速幂

前置技能：大整数乘法→9.6@p1110

洛谷 P1045 [NOIP 2003 普及组] 麦森数[3]

给定整数 P（$1000 < P < 3100000$），计算 $2^P - 1$ 的位数与最后 500 位数字（用十进制数表示），不足 500 位时高位补 0.

代码实现

```cpp
#include <cmath>
#include <cstring>
#include <iostream>
using namespace std;

const int M = 500;

int a[505], b[505], t[505];

// 大整数乘法
void mult(int x[], int y[]) {
  memset(t, 0, sizeof(t));
  for (int i = 1; i <= x[0]; i++) {
    for (int j = 1; j <= y[0]; j++) {
      if (i + j - 1 > M) continue;
      t[i + j - 1] += x[i] * y[j];
      t[i + j] += t[i + j - 1] / 10;
      t[i + j - 1] %= 10;
      t[0] = i + j;
    }
  }
  memcpy(b, t, sizeof(b));
}

// 快速幂
void binpow(int p) {
  if (p == 1) {
    memcpy(b, a, sizeof(b));
    return;
  }
  binpow(p / 2);  // (2^(p/2))^2=2^p
  mult(b, b);     // 对 b 平方
  if (p % 2 == 1) mult(b, a);
}

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  int p;
  cin >> p;
  a[0] = 1;  // 记录 a 数组的位数
  a[1] = 2;  // 对 2 进行平方
  b[0] = 1;  // 记录 b 数组的位数
  b[1] = 1;  // 答案数组
```

```
  binpow(p);
  cout << (int)(log10(2) * p) + 1 << '\n';
  b[1] -= 1;  // 最后一位减 1
  for (int i = M; i >= 1; i--) {
    cout << b[i];
    if ((i - 1) % 50 == 0) {
      cout << '\n';
    }
  }
}

```

底数固定的预处理快速幂

当底数 a 固定时，可以利用 分块思想→10.8.1@p2047，用一定的时间预处理后用 $O(1)$ 的时间回答一次幂询问．这一算法也常称为光速幂．过程如下：

1. 选定一个数 s，预处理出 $a^0, a^1, \cdots, a^{s-1}$ 与 $a^0, a^s, \cdots, a^{\lfloor p/s \rfloor s}$ 的值并存在两个数组里；
2. 对于每一次询问 a^b，将 b 拆分成 $\lfloor b/s \rfloor s + (b \bmod s)$，则 $a^b = a^{\lfloor b/s \rfloor s} \cdot a^{b \bmod s}$，就可以 $O(1)$ 求出答案．

假设指数 b 的范围是 $[0, n]$，那么块长 s 经常选择为 $\sqrt{n}$ 或者与之相近的 2 的幂次．选择 $\sqrt{n}$ 可以获得最优的预处理复杂度 $O(\sqrt{n})$，而选择 2 的幂次可以使用位操作简化计算．

特别地，对于模意义下幂的计算，底数 a 相同隐含着模数 m 也要相同这一要求．由于 扩展欧拉定理→9.11.9@p1223，对于任意模数 m，预处理的指数范围上界为 $n = 2\varphi(m)$；对于素模数 p，预处理的范围上界为 $n = p - 1$．这两种情形预处理的复杂度都是 $O(\sqrt{m})$．

参考代码

```
int a, mod, pow1[65536], pow2[65536];

void preproc() {
  pow1[0] = pow2[0] = 1;
  for (int i = 1; i < 65536; i++) pow1[i] = 1LL * pow1[i - 1] * a % mod;
  int pow65536 = 1LL * pow1[65535] * a % mod;
  for (int i = 1; i < 65536; i++) pow2[i] = 1LL * pow2[i - 1] * pow65536
% mod;
}

int query(int pows) {
  return 1LL * pow1[pows & 65535] * pow2[pows >> 16] % mod;
}

```

习题

- UVa 1230 - MODEX[4]
- UVa 374 - Big Mod[5]
- UVa 11029 - Leading and Trailing[6]
- Codeforces - Parking Lot[7]
- SPOJ - The last digit[8]
- SPOJ - Locker[9]

- SPOJ - Just add it[10]

本页面部分内容译自博文 Бинарное возведение в степень[11] 与其英文翻译版 Binary Exponentiation[12]. 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 P1226【模板】快速幂

[2] HDU 4087 A Letter to Programmers

[3] 洛谷 P1045 [NOIP 2003 普及组] 麦森数

[4] UVa 1230 - MODEX

[5] UVa 374 - Big Mod

[6] UVa 11029 - Leading and Trailing

[7] Codeforces - Parking Lot

[8] SPOJ - The last digit

[9] SPOJ - Locker

[10] SPOJ - Just add it

[11] Бинарное возведение в степень

[12] Binary Exponentiation

9.8 置换和排列

Authors: 2008verser, aofall, CoelacanthusHex, Early0v0, Great-designer, Marcythm, Persdre, shuzhouliu, Tiphereth-A, Enter-tainer, gavinliu266, gi-b716, hjsjhn, Ir1d, MegaOwIer, wjy-yy, c-forrest

置换和排列是各类问题中都很常见的概念.

⚠ 本文讨论的不是排列数

本文讨论的主题是全排列，而不是排列组合中的排列数. 排列数的相关内容应当参考 排列组合→9.13.1@p1612.

ⓘ 约定

本文中如果不加说明，总是讨论有限的集合.

定义

一个集合 X 到自身的双射（即一一对应）σ 称为 X 的一个 **置换**（permutation）. 如果集合 X 上还具有全序→9.20@p1951 关系，则它的一个置换也常称作一个 **（全）排列**. 这个全序关系称为集合上的自然顺序.

ⓘ 「置换」与「排列」

在中文语境下，「置换」通常指改变元素顺序，而「排列」通常指将元素排成一列. 当元素之间存在自然的顺序时，这两个概念是一回事：「排列」可以看作「置换」的结果；而相较于元素的自然顺序，「排列」中元素的顺序就指定了「置换」. 因为本文使用「排列」一词时，总假定集合上存在自然顺序，故而本文中不会特意区分这两个概念.

当然，没有自然顺序的元素也可以进行「排列」. 这种「排列」常常出现在组合计数问题中. 这超出了本文讨论的范畴.

设集合 X 的大小是 n，那么，X 上的全体置换的数目就是 $n!$. 特别地，$0! = 1$，即空集合上有且只有一个置换，也就是空置换.

记号

置换讨论的是元素间的对应关系, 而并不关心元素具体是什么. 因而, 当讨论大小为 n 的集合时, 通常假定讨论的集合就是 $\{1,2,\cdots,n\}$; 当集合上需要自然顺序的时候, 通常假定使用自然数上的自然顺序.

表示方法

置换有多种表示方法. 这里, 以如下置换为例, 讨论不同的置换表示方法.

$$\sigma(1)=2,\sigma(2)=6,\sigma(3)=5,\sigma(4)=4,\sigma(5)=3,\sigma(6)=1.$$

双行记号

集合 $X=\{x_1,x_2,\cdots,x_n\}$ 上的置换可以表示为

$$\sigma=\begin{pmatrix}x_1 & x_2 & \cdots & x_n\\ x_{p_1} & x_{p_2} & \cdots & x_{p_n}\end{pmatrix}.$$

它表示置换 σ 将元素 x_i 映射到 x_{p_i}. 这里, 当然需要 $X=\{x_{p_1},x_{p_2},\cdots,x_{p_n}\}$. 置换的双行记号表示中, 首行的元素出现顺序并不重要, 重要的是两行之间的对应关系.

比如说, 前文的例子可以按照双行记号写作

$$\sigma=\begin{pmatrix}1&2&3&4&5&6\\2&6&5&4&3&1\end{pmatrix},$$

当然, 也可以写作

$$\sigma=\begin{pmatrix}6&5&4&3&2&1\\1&3&4&5&6&2\end{pmatrix}.$$

单行记号

很多时候, 集合 X 上有自然顺序. 如果在双行记号中默认首行按照自然的顺序书写, 并省略首行, 那么, 置换可以表示为

$$\sigma=\sigma(1)\sigma(2)\cdots\sigma(n).$$

这更像自然语言中排列的概念. 所以, 有时也用「排列」称呼这个有序组.

前文的例子利用单行记号可以写作

$$\sigma=265431.$$

这种单行的记号, 常用来比较不同排列的大小.

轮换表示

置换还有一种更为紧凑的表达方式, 称为置换的轮换表示. 它将置换表示为一系列不相交的轮换的乘积. 下面描述将给定置换写成轮换表示的步骤.

给定一个置换 σ, 可以通过如下步骤写成轮换表示:

1. 如果 X 中还有未曾写下的元素, 就写下一个左括号, 并写下任意一个这样的元素;
2. 当前一个写下的元素是 x 时,
   - 如果 $\sigma(x)$ 已经在前面写下过, 就写上右括号, 并返回步骤 1;
   - 如果 $\sigma(x)$ 还没有写下过, 就写下 $\sigma(x)$, 并继续步骤 2;
3. 重复上述步骤, 直到 X 的所有元素都已经写下为止.

每一对括号中, 都是一个轮换. 括号中的元素个数, 称为对应轮换的长度. 实践中, 常常省略掉长度为一的轮换.

前文的例子利用轮换表示可以写作

$$\sigma = (126)(35)(4) = (126)(35).$$

恒等变换中所有的轮换长度都是一，常常记作 (1) 而不是全部省略.

复合

置换的复合就是映射的复合. 置换的复合也常常称作置换的乘法.

给定两个置换

$$\sigma = \begin{pmatrix} x_1 & x_2 & \cdots & x_n \\ x_{p_1} & x_{p_2} & \cdots & x_{p_n} \end{pmatrix}, \pi = \begin{pmatrix} x_{p_1} & x_{p_2} & \cdots & x_{p_n} \\ x_{q_1} & x_{q_2} & \cdots & x_{q_n} \end{pmatrix},$$

那么，它们的乘积 $\pi \circ \sigma$ 就是

$$\pi \circ \sigma = \begin{pmatrix} x_1 & x_2 & \cdots & x_n \\ x_{q_1} & x_{q_2} & \cdots & x_{q_n} \end{pmatrix}.$$

简单来说就是先经过 σ 的映射，再经过 π 的映射. 注意在上面的双行记号中，内层映射 σ 的第二行的顺序和外层映射 π 的第一行的顺序一致.

因为置换 σ 和 π 本质是两个映射，所以 $(\pi \circ \sigma)(x) = \pi(\sigma(x))$. 置换的复合的运算顺序是自右向左的. 置换的乘法并不满足交换律，所以使用错误的顺序计算可能会导致错误的结果.

同一置换的连续多次复合称作置换的幂，可以使用 快速幂→9.7@p1131 加速计算.

逆置换

因为置换是双射，所以置换总有相应的逆置换.

给定置换

$$\sigma = \begin{pmatrix} x_1 & x_2 & \cdots & x_n \\ x_{p_1} & x_{p_2} & \cdots & x_{p_n} \end{pmatrix},$$

它的逆置换就是

$$\sigma^{-1} = \begin{pmatrix} x_{p_1} & x_{p_2} & \cdots & x_{p_n} \\ x_1 & x_2 & \cdots & x_n \end{pmatrix}.$$

在轮换表示中，只要对每个轮换取逆，就能得到原来的置换的逆；而对每个轮换取逆，只要把元素的书写顺序倒过来就可以了. 比如说，上文中的例子 σ 的逆置换的轮换表示是

$$\sigma^{-1} = (621)(53) = (162)(35).$$

给定 $1 \sim n$ 的一个排列，它与每个元素在该排列中所处位置构成的序列互为逆排列.

轮换

轮换（cycle）本身是特殊的置换. 轮换的特性是，从轮换中的任何一点 x 出发，都能通过反复应用置换 σ 的方式得到轮换中的任何一点 y. 长度为 k 的轮换也称作 k-**轮换**（k-cycle）. 反复应用 k-轮换 k 次，将得到恒等变换，即每个元素都回到了最开始的位置.

置换的轮换表示可以看作将置换写成这些特殊置换（即轮换）的乘积，因而置换的轮换表示也可以看作是置换的 **轮换分解**（cycle decomposition）. 对于每个置换，它分解成轮换乘积的方式在不计顺序后都是唯一的. 轮换可以看作是构成置换的基本单元.

置换的轮换分解有着清晰的几何意义. 如果将集合 X 上的置换中的每个有序对 $(x, \sigma(x))$ 都看成以 X 为顶点的有向图的边，那么这些轮换就是这个图上面的环路. 如果置换 σ 能够分解为 m 个轮换，就意味着对应的有向图中共计有 m 个环路（包括自环）. 这些环路自然互不相交.

不动点

1-轮换中的元素就是置换的 **不动点**（fixed point）. 对于集合 X 上的置换 σ，通常用 X^σ 表示 σ 的不动点集合，即 $X^\sigma = \{x \in X : \sigma(x) = x\}$.

对换

2-轮换也称作 **对换**（transposition）. 也就是说，对换就是只交换了一对元素位置的置换. 它的轮换表示是 $(x_i x_j)$，表示它交换了 x_i 和 x_j 的位置.

任何置换都可以写作一系列对换的乘积. 这相当于说，任何顺序的排列都可以通过一系列交换两个元素的操作恢复成指定的正序排列. 这正是基于交换的排序算法在做的事情.

更进一步，冒泡排序算法→5.7.3@p571 的正确性其实说明，任何置换都可以写作一系列相邻对换的乘积. 这里的 **相邻对换**（adjacent transposition）指的是只交换相邻元素的对换.

性质

在应用中，常常需要关注单个置换的性质.

奇偶性

将轮换分解成对换的方式并不是唯一的. 比如，

$$(123) = (13)(12) = (12)(23) = (12)(13)(12)(13).$$

但是，置换分解成一系列对换时，需要的对换的数目的奇偶性是固定的. 一个置换的对换分解的数目的奇偶性也称作置换的 **奇偶性**（parity）.

能够分解成偶数个对换的乘积的置换叫做偶置换，能够分解成奇数个对换的乘积的置换叫做奇置换. 当 $n \geq 2$ 时，n 元置换中奇置换和偶置换的数目相同.

符号

根据置换的奇偶性，还可以定义置换的 **符号**（sign），记作 $\operatorname{sgn}\sigma$. 偶置换的符号定义为 $+1$，奇置换的符号定义为 -1.

置换的乘积的符号，等于它们的符号的乘积，即

$$\operatorname{sgn}(\pi \circ \sigma) = \operatorname{sgn}\pi \cdot \operatorname{sgn}\sigma.$$

也就是说，两个奇偶性相同的置换的复合是偶置换，两个奇偶性不同的置换的复合是奇置换. 这一结论，从对换分解的角度看是显然的.

特别地，单次对换必然改变置换的奇偶性. 这也正解释了为什么虽然分解成对换的方式不唯一，但是所需的对换的数目的奇偶性是确定的.

置换的符号出现在 行列式的 Leibniz 展开→9.14.6@p1729 中.

置换的阶

置换的 **阶**（order）是指满足如下条件的最小正整数 a：重复该置换 a 次后，所有元素都回到了原位. 即

$$\operatorname{ord}\sigma = \min\{a \in \mathbf{N}_+ : \sigma^a = (1)\}.$$

有限集合上，所有置换的阶都是有限的. 这意味着，从起始顺序出发，只要重复按照固定模式打乱给定序列，在有限时间内，总可以将排列恢复原样.

置换的型

将 n 个元素的置换做轮换分解，置换的 **型**（cycle type）就是分解中轮换长度的可重集合. 这些轮换的长度构成了 n 的一个整数分拆. 如果得到的分解中长度为 k 的轮换共计 α_k 个，那么置换的型常记作

$$1^{\alpha_1} 2^{\alpha_2} \cdots n^{\alpha_n},$$

且这些指数满足 $\sum_{k=1}^{n} k\alpha_k = n$.

给定置换的型，不同的置换的数目为

$$\frac{n!}{1^{\alpha_1}2^{\alpha_2}\cdots n^{\alpha_n}\alpha_{(1)!}\alpha_{(2)!}\cdots\alpha_{(n)!}}.$$

分析

这是因为，给定任何 $1\sim n$ 的排列，都可以按照置换的型分割成相应的轮换分解．但是，长度相同的轮换之间的顺序并不影响置换，所以总数需要除以 $\prod_k \alpha_{(k)!}$．另外，同一轮换内部实际是圆排列，起点的选取也不影响置换，所以需要除以 $\prod_k k^{\alpha_k}$．这就得到上式．

如果仅仅给定置换分解成的轮换个数 k，则不同的置换的数目为 第一类斯特林数→9.13.7@p1645 $\begin{bmatrix}n\\k\end{bmatrix}$．不同的型的个数为 n 的 分拆数→9.13.12@p1672 p_n．

从置换的型，可以方便地确定置换的阶和奇偶性等性质．

因为 k-轮换的阶是 k，不同的轮换又互不相交，所以置换 σ 的阶就是

$$\operatorname{lcm}\{k:\alpha_k>0\}.$$

同样地，因为 k-轮换的奇偶性与 k 的奇偶性相反，所以置换 σ 的奇偶性就是

$$\sum_k (k-1)\alpha_k=\sum_k k\alpha_k-\sum_k \alpha_k=n-c(\sigma)$$

的奇偶性．这里，$c(\sigma)$ 是轮换的个数（包括 1-轮换，即不动点）．

置换的型在 Pólya 计数→9.13.14@p1680 中有重要作用．

排列相关

如果集合 X 本身具有自然顺序，此时置换 σ 常称作排列，并用单行记号

$$\sigma(1)\sigma(2)\cdots\sigma(n)$$

表示．注意不要同轮换混淆．

逆序数

在一个排列中，如果某一个较大的数排在某一个较小的数前面，就说这两个数构成一个 **逆序**（inversion）或反序．这里的比较是在自然顺序下进行的．

在一个排列里出现的逆序的总个数，叫做这个排列的 **逆序数**．排列的逆序数是它恢复成正序序列所需要做相邻对换的最少次数．因而，排列的逆序数的奇偶性和相应的置换的奇偶性一致．这可以作为置换的奇偶性的等价定义．

求解逆序数的算法，可以使用 归并排序→5.7.8@p592 或 树状数组→10.12@p2078，时间复杂度均为 $O(n\log n)$．两种算法的解释详见对应章节，这里给出它们的参考实现．

参考实现

- 归并排序

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

// Merge sort and count inversions.
long long merge_sort(std::vector<int>& nums, int b, int e) {
  // Return 0 when length <= 1.
  if (e - b <= 1) return 0;
  long long res = 0;
  int m = (b + e) / 2;
```

```
  // Merge_sort two halves.
  res += merge_sort(nums, b, m);
  res += merge_sort(nums, m, e);
  // Temporary vector to store sorted array.
  std::vector<int> tmp(e - b);
  int i = b, j = m, k = 0;
  while (i < m && j < e) {
    if (nums[j] < nums[i]) {
      tmp[k] = nums[j++];
      // In this case, all elements in [i,m) are larger than element j.
      res += m - i;
    } else {
      tmp[k] = nums[i++];
    }
    ++k;
  }
  // Deal with remaining elements.
  for (; i < m; ++i, ++k) {
    tmp[k] = nums[i];
  }
  for (; j < e; ++j, ++k) {
    tmp[k] = nums[j];
  }
  // Copy back to original vector.
  for (i = b, k = 0; i < e; ++i, ++k) {
    nums[i] = tmp[k];
  }
  return res;
}

int main() {
  int n;
  std::cin >> n;
  std::vector<int> nums(n);
  for (int& num : nums) {
    std::cin >> num;
  }
  std::cout << merge_sort(nums, 0, n);
  return 0;
}

```

- 树状数组

```
#include <algorithm>
#include <iostream>
#include <unordered_map>
#include <vector>

// A simple BIT implementation.
class BIT {
  int n;
  std::vector<int> su;
```

```
10
11  public:
12   BIT(int n) : n(n), su(n + 1) {}
13
14   // Add v to the x-th number.
15   void add(int x, int v) {
16     for (; x <= n; x += x & (-x)) {
17       su[x] += v;
18     }
19   }
20
21   // Get the cumulative sum till the x-th number.
22   int query(int x) {
23     int res = 0;
24     for (; x; x &= x - 1) {
25       res += su[x];
26     }
27     return res;
28   }
29 };
30
31 // Count inversions.
32 long long solve(const std::vector<int>& nums) {
33   // Discretization.
34   std::vector<int> sorted(nums);
35   std::sort(sorted.begin(), sorted.end());
36   sorted.erase(std::unique(sorted.begin(), sorted.end()),
   sorted.end());
37   std::unordered_map<int, int> ids;
38   int m = sorted.size();
39   for (int i = 0; i < m; ++i) {
40     // Reverse the order.
41     // Now a smaller id means a larger element.
42     ids[sorted[i]] = m - i;
43   }
44   // Main part.
45   BIT bit(m);
46   long long res = 0;
47   for (int num : nums) {
48     int id = ids[num];
49     // Get inversion pair (i,j) with j the current element.
50     // Namely, count the number of elements larger than
51     //     the current one but located before it.
52     res += bit.query(id - 1);
53     // Insert the current element to the BIT.
54     bit.add(id, 1);
55   }
56   return res;
57 }
58
59 int main() {
60   int n;
```

```
61    std::cin >> n;
62    std::vector<int> nums(n);
63    for (int& num : nums) {
64      std::cin >> num;
65    }
66    std::cout << solve(nums);
67    return 0;
68  }
69
```

顺序

排列之间是可以比较大小的. 因为每个单行记号就是一个字符串, 排列的顺序就是这个字符串上的 字典序→8.2@p890.

在 C++ 标准库头文件 `<algorithm>` 中可以使用 `prev_permutation` 和 `next_permutation` 分别找到当前排列按照字典序的上一个和下一个排列.

排名

将 n 个元素的排列按照字典序从小到大列举出来，则某一排列在这个序列中的位次就是该排列的排名．它建立了排列和正整数之间的一一对应，常常用于排列相关问题的状态压缩.

在中文竞赛圈，这个排名常称作排列的「康托展开」，但这种名称并不规范．更为严谨的说法是，一个排列的排名的 **康托展开**（Cantor expansion），对应着该排列的 **Lehmer 码**（Lehmer code）.

关于「康托展开」

正如名字所暗示的那样, 康托展开是指一种将自然数展开为数列的方法. 它可以看作是一种特殊的进制，也叫做 阶乘进制→9.3.2@p1087．这种进制中，不同的数位对应的底数（radix）并不相同．比如，十进制数 463_{10} 可以在阶乘进制中表示为

$$463_{10} = 341010_!.$$

它表示如下含义

$$463 = 3 \times 5! + 4 \times 4! + 1 \times 3! + 0 \times 2! + 1 \times 1! + 0 \times 0!.$$

康托对于这类混合底数的进制进行了研究，故而自然数在这种进制下的数码表示也常称作自然数的康托展开.

示例

不熟悉排名的计算方法的读者，可以通过这个简单的例子理解下面的算法的基本思路.

要计算排列 $\sigma = 452631$ 的排名，就是要计算有多少排列的字典序小于 σ，再加一．这里的思想类似于 数位 DP→7.9@p747，都是逐位讨论.

- 第 1 位的选取要小于 σ，只能取自 $\{1, 2, 3\}$，后面 5 位可以任意选取，共 $3 \times 5!$ 个可选的排列;
- 如果第 1 位也选择 4，那么第 2 位的选取要小于 σ，只能取自 $\{1, 2, 3\}$（这里，4 已经选过了），后面的 4 位可以任意选取，共 $3 \times 4!$ 个可选的排列;
- 类似地，前 2 位的选取和 σ 相同时，第 3 位的选取要小于 σ，只能取自 $\{1\}$，后面的 3 位可以任意选取，共 $1 \times 3!$ 个可选的排列;
- 前 3 位的选取和 σ 相同时，第 4 位的选取要小于 σ，只能取自 $\{1, 3\}$，后面的 2 位可以任意选取，共 $2 \times 2!$ 个可选的排列;

- 前 4 位的选取和 σ 相同时，第 5 位的选取要小于 σ，只能取自 $\{1\}$，后面的 1 位可以任意选取，共 $1\times 1!$ 个可选的排列；
- 前 5 位的选取和 σ 相同时，第 6 位的选择无法得到比 σ 更小的排列，所以共 $0\times 0!$ 个可选的排列.

因此，排列 σ 的排名为

$$1+3\times 5!+3\times 4!+1\times 3!+2\times 2!+1\times 1!+0\times 0!=444.$$

对于不同的排列，关键的点在于确定阶乘前面的系数. 实际上，这些系数正是排在该位置之后却小于该位置元素的元素数目.

从例子中可以知道，求解给定排列的排名的算法，可以分为两步：

1. 将给定的长度为 n 的排列转化为它的 Lehmer 码，即长度为 n 的序列 L_σ，其中，第 i 位是
   $$L_\sigma(i)=\#\{j>i:\sigma(j)<\sigma(i)\},$$
   也就是在排列中，排在第 i 位后面，但是却比 $\sigma(i)$ 小的元素个数. 它等于尚未使用的元素中给定元素的排名（减一）.
2. 将 Lehmer 码看作是自然数的康托展开，求出原来的自然数，并加一. 也就是说，最终的排名等于 $\operatorname{rank}\sigma=1+L_\sigma(1)(n-1)!+L_\sigma(2)(n-2)!+\cdots+L_\sigma(n)0!$.

要求解这一问题的逆问题，即给定排名求解相应的排列，只要将上述过程反过来操作即可. 在这一过程中求得的 Lehmer 码中的数字之和，就是排列的逆序数.

编程实现时，关键是要能够快速计算「尚未使用的元素中给定元素的排名」（求排名时）和「尚未使用的元素中给定排名的元素」（求排列时），这些都可以通过 树状数组→10.12@p2078 或 线段树→10.13.1@p2098 等数据结构维护. 正反操作的时间复杂度均为 $O(n\log n)$.

参考实现

- 求给定排列的排名

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

// A simple BIT implementation.
class BIT {
  int n;
  std::vector<int> su;

 public:
  BIT(int n) : n(n), su(n + 1) {}

  // Add v to the x-th number.
  void add(int x, int v) {
    for (; x <= n; x += x & (-x)) {
      su[x] += v;
    }
  }

  // Get the cumulative sum till the x-th number.
  int query(int x) {
    int res = 0;
    for (; x; x &= x - 1) {
      res += su[x];
```

```
    }
    return res;
  }
};

// Get the rank of a permutation of 1~n.
long long find_rank(const std::vector<int>& nums) {
  int n = nums.size();
  BIT bit(n);
  long long fac = 1;
  long long res = 0;
  // Reverse iteration.
  for (int i = n - 1; i >= 0; --i) {
    // Count the number of elements smaller than the current one.
    res += bit.query(nums[i] - 1) * fac;
    // Insert the current element into the BIT.
    bit.add(nums[i], 1);
    // Update the factorial.
    fac *= n - i;
  }
  return res + 1;
}

int main() {
  int n;
  std::cin >> n;
  std::vector<int> nums(n);
  for (int& num : nums) {
    std::cin >> num;
  }
  std::cout << find_rank(nums);
  return 0;
}

```

- 求给定排名的排列

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

// A simple BIT implementation.
class BIT {
  int n;
  std::vector<int> su;

 public:
  BIT(int n) : n(n), su(n + 1) {}

  // Fill the BIT with one.
  void fill() {
    for (int x = 1; x <= n; ++x) {
      su[x] += x & (-x);
```

```
    }
  }

  // Add v to the x-th number.
  void add(int x, int v) {
    for (; x <= n; x += x & (-x)) {
      su[x] += v;
    }
  }

  // Get the k-th smallest element.
  int find_kth(int k) {
    int ps = 0, x = 0;
    for (int i = log2(n); i >= 0; --i) {
      x += 1 << i;
      if (x >= n || ps + su[x] >= k) {
        x -= 1 << i;
      } else {
        ps += su[x];
      }
    }
    return x + 1;
  }
};

// Find the k-th permutation of 1~n.
std::vector<int> find_permutation(int n, long long k) {
  --k;
  // Expand rank to Lehmer code.
  std::vector<int> lehmer(n);
  for (int i = 1; i <= n; ++i) {
    lehmer[n - i] = k % i;
    k /= i;
  }
  BIT bit(n);
  // Set all values in BIT to one.
  bit.fill();
  std::vector<int> res(n);
  for (int i = 0; i < n; ++i) {
    // Find the lehmer[i]-th smallest unused element.
    res[i] = bit.find_kth(lehmer[i] + 1);
    // Remove it from the BIT.
    bit.add(res[i], -1);
  }
  return res;
}

int main() {
  int n;
  long long k;
  std::cin >> n >> k;
  auto res = find_permutation(n, k);
```

```
  for (int num : res) {
    std::cout << num << ' ';
  }
  return 0;
}

```

参考资料与注释

- Permutation - Wikipedia[1]
- Lehmer code - Wikipedia[2]
- Factorial number system - Wikipedia[3]

外部参考资料

[1] Permutation - Wikipedia  [2] Lehmer code - Wikipedia  [3] Factorial number system - Wikipedia 

9.9 弧度制与坐标系

Authors: Ir1d, HeRaNO, Chrogeek, abc1763613206, mxdyzmx

角的定义

在小学或初中已经学习过角的 **静态定义**：具有公共端点的两条射线组成的图形叫做角.

但是该定义将角度限制在了 $[0, 360^\circ]$, 这给深入研究带来了一定的困难, 还有其他的问题无法解释清, 比如：旋转 720° 是什么意思?

在高中数学, 讲了角的 **动态定义**: 平面内一条射线绕其端点从一个位置旋转到另一个位置形成的图形叫做角.

开始的位置称为 **始边**，结束的位置称为 **终边**. 并规定:

- 按 **逆时针** 方向旋转形成的角叫做 **正角**，其角度为正;
- 按 **顺时针** 方向旋转形成的角叫做 **负角**，其角度为负;
- 终边相对于始边没有做任何旋转的角叫做 **零角**，其角度为 0°.

这样就把角的概念推向了 **任意角**.

注意

零角始边和终边重合，但始边和终边重合的角并不都是零角，如以 360° 为倍数的角.

弧度制

实际应用中经常有角度到各种参数的转换，而使用弧度制描述角可以减少系数的使用. 所以接下来, 介绍 **弧度制**:

把长度等于半径长的弧所对的圆心角称为 1 弧度的角，用符号 rad 表示，读作：弧度.

根据前面的规定，正角的弧度为正，负角的弧度为负，零角的弧度为 0，如果半径为 r 的圆的圆心角 α 所对弧长为 l，则:

$$|\alpha| = \frac{l}{r}$$

利用这个公式还可以写出弧长和扇形面积公式，在此略过.

于是，360° 角的弧度为 2π，这样有了对应关系之后就可以进行角度值和弧度制的转化了：

$$k\,\mathrm{rad} = \frac{\pi}{180^\circ} n^\circ$$

考虑一个角，将其终边再旋转一周，甚至多周，始边位置不动，那么终边位置永远是相同的，称这些角为终边位置相同的角.

与角 α 终边位置相同的角的集合很容易得出，为 $\{\varphi \mid \varphi = \alpha + 2k\pi, k \in \mathbf{Z}\}$.

可以理解为：给这个角的边不停加转一圈，终边位置不变.

π 和 τ 两个数学常数

目前西方数学界有一些观点认为，「真正的圆周率」应为 2π，将这个值记为希腊字母 τ. 新圆周率的支持者们选择在 6 月 28 日庆祝「真正的」圆周率日.

比如，在弧度制下，一个周角是 2π，直接对 2π 进行等分可以得到周角的等分. 又例如，在复变函数中频繁出现 2π 的组合，等等.

为了迎合中国各地区约定俗成的习惯，在 OI Wiki，采用参数 π 表示圆周率.

编程中圆周率的习惯写法

在 C/C++ 语言中，一般取 π 为 `acos(-1)`，只有这个值是最接近 π 的浮点数. 使用 `acos(-1)` 或者 `4 * atan(1)` 写出来的 π 是 3.14159265358979310000.

采用其他值，例如 `acos(-1.0/2.0)`，`acos(1.0/2.0)`，`asin(1.0/2.0)` 等等，写出来的 π 是 3.14159265358979360000，这就不是最接近 π 的浮点数了.

如果你背得下来，也可以直接写 3.1415926535897932.

平面直角坐标系

在同一个平面上互相垂直且有公共原点的两条数轴构成平面直角坐标系（Rectangular Coordinates）.

通常，两条数轴分别置于水平位置与垂直位置，取向右与向上的方向分别为两条数轴的正方向. 水平的数轴叫做 x 轴（x-axis）或横轴，垂直的数轴叫做 y 轴（y-axis）或纵轴，x 轴 y 轴统称为坐标轴，它们的公共原点 O 称为平面直角坐标系的原点（origin），以点 O 为原点的平面直角坐标系记作平面直角坐标系 xOy.

x 轴 y 轴将坐标平面分成了四个象限（quadrant），右上方的部分叫做第一象限，其他三个部分按逆时针方向依次叫做第二象限、第三象限和第四象限. 象限以数轴为界，横轴、纵轴上的点及原点不在任何一个象限内. 一般情况下，x 轴 y 轴取相同的单位长度，但在特殊的情况下，也可以取不同的单位长度.

平面直角坐标系下位置的描述

在平面直角坐标系中，对于平面上的任意一点，都有唯一的一个有序数对（即点的坐标（coordinates））与它对应；反过来，对于任意一个有序数对，都有平面上唯一的一点与它对应.

对于平面内任意一点 C，过点 C 分别向 x 轴、y 轴作垂线，垂足在 x 轴、y 轴上的对应点 a, b 分别叫做点 C 的横坐标、纵坐标，有序数对（ordered pair）(a, b) 叫做点 C 的直角坐标. 一个点在不同的象限或坐标轴上，其坐标都不一样.

平面极坐标系

考虑实际情况，比如航海，说「点 B 在点 A 的北偏东 30° 方向上，距离为 100 米」，而不是「以 A 为原点建立平面直角坐标系，$B(50, 50\sqrt{3})$」.

这样：

1. 在平面上选一定点 O，称为 **极点**；
2. 自极点引出一条射线 Ox，称为 **极轴**；
3. 选择一个单位长度（在数学问题中通常为 1），一个角度单位（通常为弧度）及其正方向（通常为逆时针方向）；

就建立了 **极坐标系**.

极坐标系下位置的描述

设 A 为平面上一点.

- 极点 O 与 A 之间的距离 $|\,OA\,|$ 称为 **极径**，记为 ρ；
- 以极轴为始边，OA 为终边的角 $\angle xOA$ 称为 **极角**，记为 φ；

那么有序数对 (ρ,φ) 即为 A 的 **极坐标**.

由终边相同的角的定义可知，(ρ,φ) 与 $(\rho,\varphi+2k\pi)$ $(k\in\mathbf{Z})$ 其实表示的是一样的点. 特别地，极点的极坐标为 $(0,\varphi)$ $(\varphi\in\mathbf{R})$，于是平面内的点的极坐标表示有无数多种.

如果规定 $\rho\geq 0, 0\leq\varphi<2\pi$，那么除极点外，其他平面内的点可以用唯一有序数对 (ρ,φ) 表示，而极坐标 (ρ,φ) 表示的点是唯一确定的.

平面直角坐标系与极坐标系的相互转换

当然, 有时候研究极坐标系下的图形有些不方便. 要想转到直角坐标系下研究, 有互化公式. 点 $A(\rho,\varphi)$ 的直角坐标 (x,y) 可以如下表示：

$$\begin{aligned} x&=\rho\cos\varphi\\ y&=\rho\sin\varphi \end{aligned}$$

进而可知：

$$\begin{aligned} \rho^2&=x^2+y^2\\ \tan\varphi&=\frac{y}{x}\quad(x\neq 0) \end{aligned}$$

于是有 $\rho=\sqrt{x^2+y^2}$.

但具有相同 $\dfrac{y}{x}$ 的 $\tan\varphi$ 有两个可能的 φ 的值，此时还需要根据 x,y 的值来确定方向. 具体地，定义函数：

$$\operatorname{atan2}(y,x)=\begin{cases} \arctan(\frac{y}{x}) & \text{if } x>0\\ \arctan(\frac{y}{x})+\pi & \text{if } y\geq 0, x<0\\ \arctan(\frac{y}{x})-\pi & \text{if } y<0, x<0\\ \pi/2 & \text{if } y>0, x=0\\ -\pi/2 & \text{if } y<0, x=0\\ \text{any} & \text{if } y=0, x=0 \end{cases}$$

则 $\varphi=\operatorname{atan2}(y,x)$. 注意上述函数的值域为 $(-\pi,\pi]$.

在 C/C++ 语言的 `<math.h>` 或 `<cmath>` 库里定义了 该函数[1]，调用 `atan2(y, x)` 即可.

空间直角坐标系

使用如下方法建立空间直角坐标系：

1. 在空间内选定一点 O；
2. 过点 O 作三条互相垂直的数轴 $\overrightarrow{Ox},\overrightarrow{Oy},\overrightarrow{Oz}$，分别称作 x 轴（横轴），y 轴（纵轴），z 轴（竖轴），统称为坐标轴；它们的正方向符合右手规则，即以右手握住 z 轴，当右手的四个手指 x 轴的正向以角度转向 y 轴正向时，大拇指的指向就是 z 轴的正向；
3. 设定各轴上的长度单位，通常都设为 1.

这样就构成了一个空间直角坐标系，称为空间直角坐标系 $O-xyz$. 定点 O 称为该坐标系的原点.

任意两条坐标轴确定一个平面，这样可确定三个互相垂直的平面，统称为坐标面．其中 x 轴与 y 轴所确定的坐标面称为 xOy 面，类似地有 yOz 面和 zOx 面．三个坐标面把空间分成八个部分，每一部分称为一个卦限．

空间直角坐标系下位置的描述

取定空间直角坐标系 $O-xyz$ 后，就可以建立空间的点与三元组之间的一一对应关系．

设点 M 为空间的一点，过点 M 分别作垂直于 x 轴、y 轴和 z 轴的平面．设三个平面与 x 轴、y 轴和 z 轴的交点依次为 P,Q,R，点 P,Q,R 分别称为点 M 在 x 轴、y 轴和 z 轴上的投影．又设点 P,Q,R 在 x 轴、y 轴和 z 轴上的坐标依次为 x,y,z，于是点 M 确定了一个三元组 (x,y,z)．

反之，如果给定一个三元组 (x,y,z)，可以在 x 轴上取坐标为 x 的点 P，在 y 轴上取坐标为 y 的点 Q，在 z 轴上取坐标为 z 的点 R，然后点 P,Q,R 分别作垂直于 x 轴、y 轴和 z 轴的三个平面，它们相交于空间的一点 M，点 M 就是由三元组 (x,y,z) 所确定的点．

这样一来，空间的点 M 与三元组 (x,y,z) 之间就建立了一一对应的关系．把三元组 (x,y,z) 称为点 M 的坐标，记作 $M(x,y,z)$，其中 x 称为横坐标、y 称为纵坐标、z 称为竖坐标．

空间柱坐标系

空间柱坐标系，将极坐标扩展为三维的方式：从应用于平面工作中的极坐标系开始，然后过极点 O 添加垂直于该平面的 z 轴，方向朝上．

为了找到由柱坐标 (ρ,φ,z) 所描述的点，可以首先在极坐标系下处理 ρ 和 φ，然后根据 z 坐标沿着 z 轴「向上」或「向下」移动．

柱坐标系与空间直角坐标系的相互转换

两坐标系下 z 的值是相同的．

(x,y) 与 (ρ,φ) 的相互转换参见上文 平面直角坐标系与极坐标系的相互转换→9@p1079．

空间球坐标系

球坐标可以通过以下方法确定：

1. 站在原点，面向水平极轴的方向；垂直轴的指向是从脚指向头部；
2. 手臂向上，指向垂直极轴方向；
3. 逆时针旋转角度 φ；
4. 将手臂向下旋转角度 ϑ，手臂指向 φ 和 ϑ 指定的方向；
5. 沿着该方向从原点移位距离 r．

这样即可到达球坐标 (r,ϑ,φ) 所描述的点．其中 ϑ 称为 **天顶角**，φ 称为 **方位角**．

> ⚠ **Warning**
>
> 由于诸多原因，有的地方使用 ϕ 表示天顶角，用 θ 表示方位角．阅读文章遇到球坐标系时请务必注意这一点．
>
> 同时，在写文章时，如果用到了球坐标系，建议提前声明清楚使用什么符号表示天顶角和方位角．

柱坐标系与球坐标系的相互转换

两坐标系下 φ 的值是相同的．

从柱坐标系到球坐标系：

$$r=\sqrt{\rho^2+z^2}$$

$$\vartheta=\begin{cases}\arctan\left(\frac{\rho}{z}\right) & \text{if } z>0\\ \pi/2 & \text{if } z=0,\rho\neq 0\\ \arctan\left(\frac{\rho}{z}\right)+\pi & \text{if } z<0\end{cases}$$

注意对于柱坐标系下的点 $(0,0,0)$，其球坐标的 ϑ 不明确.

从球坐标系到柱坐标系：

$$\rho=r\sin\vartheta$$

$$z=r\cos\vartheta$$

空间直角坐标系与球坐标系的相互转换

可以结合上文 平面直角坐标系与极坐标系的相互转换→9@p1079 和上文 柱坐标系与球坐标系的相互转换→9@p1079 一起使用，或直接使用下面的公式：

从空间直角坐标系到球坐标系：

$$r=\sqrt{x^2+y^2+z^2}$$

$$\vartheta=\arccos\left(\frac{z}{\sqrt{x^2+y^2+z^2}}\right)$$

$$\varphi=\operatorname{atan2}(y,x)$$

其中 atan2 的定义见 平面直角坐标系与极坐标系的相互转换→9@p1079.

注意对于空间直角坐标系下的点 $(0,0,0)$，其球坐标的 ϑ 和 φ 取值不明确.

从球坐标系到空间直角坐标系：

$$x=r\sin\vartheta\cos\varphi$$

$$y=r\sin\vartheta\sin\varphi$$

$$z=r\cos\vartheta$$

外部参考资料

[1] 该函数

9.10 复数

如果您已经学习过复数相关知识，请跳过本页面.

学习复数知识需要一部分向量基础，如果并未学习过向量知识请移步 向量页面→9.14.2@p1707.

复数

引入

> **注**
>
> 下面的引入方法来自人教版高中数学 A 版必修二.

从方程的角度看，负实数能不能开平方，就是方程 $x^2+a=0(a>0)$ 有没有解，进而可以归结为方程 $x^2+1=0$ 有没有解.

回顾已有的数集扩充过程，可以看到，每次扩充都与实际需求密切相关. 例如，为了解决正方形对角线的度量，以及 $x^2-2=0$ 这样的方程在有理数集中无解的问题，人们把有理数集扩充到了实数集. 数集

扩充后，在实数集中规定的加法运算、乘法运算，与原来在有理数集中规定的加法运算、乘法运算协调一致，并且加法和乘法都满足交换律和结合律，乘法对加法满足分配律.

依照这种思想, 为了解决 $x^2+1=0$ 这样的方程在实数系中无解的问题, 我们设想引入一个新数 i, 使得 $x=\mathrm{i}$ 是方程 $x^2+1=0$ 的解，即使得 $\mathrm{i}^2=-1$.

思考：把新引进的数 i 添加到实数集中，我们希望数 i 和实数之间仍然能像实数那样进行加法和乘法运算, 并希望加法和乘法都满足交换律、结合律, 以及乘法对加法满足分配律. 那么, 实数系经过扩充后, 得到的新数系由哪些数组成呢?

依照以上设想，把实数 b 与 i 相乘，结果记作 $b\mathrm{i}$；把实数 a 与 $b\mathrm{i}$ 相加，结果记作 $a+b\mathrm{i}$. 注意到所有实数以及 i 都可以写成 $a+b\mathrm{i}(a,b\in\mathbf{R})$ 的形式，从而这些数都在扩充后的新数集中.

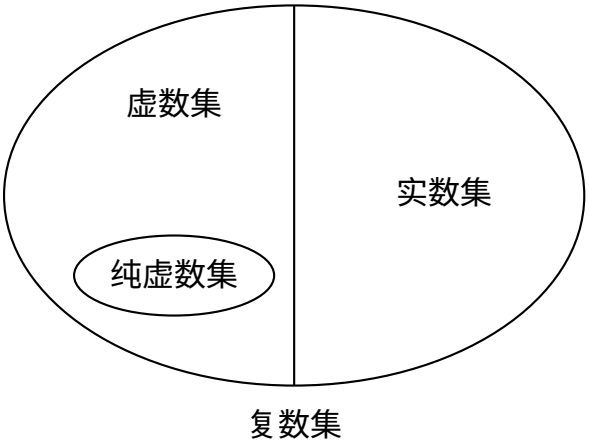
定义

我们定义形如 $a+b\mathrm{i}$，其中 $a,b\in\mathbf{R}$ 的数叫做 **复数**，其中 i 被称为 **虚数单位**，全体复数的集合叫做 **复数集**，记作 $\mathbf{C}$.

复数通常用 z 表示，即 $z=a+b\mathrm{i}$. 这种形式被称为 **复数的代数形式**. 其中 a 称为复数 z 的 **实部**，记作 $\mathrm{Re}(z)$，b 称为复数 z 的 **虚部**，记作 $\mathrm{Im}(z)$. 如无特殊说明，都有 $a,b\in\mathbf{R}$.

对于一个复数 z，当且仅当 $b=0$ 时，它是实数，当 $b\neq0$ 时，它是虚数，当 $a=0$ 且 $b\neq0$ 时，它是纯虚数.

纯虚数，虚数，实数，复数的关系如下图所示.



性质与运算

几何意义

我们知道了 $a+b\mathrm{i}$ 这样类似的形式的数被称为复数, 并且给出了定义和分类, 我们还可以挖掘一下更深层的性质.

我们把所有实数都放在了数轴上，并且发现数轴上的点与实数一一对应. 我们考虑对复数也这样处理.

首先我们定义 **复数相等**：两个复数 $z_1=a+b\mathrm{i},z_2=c+d\mathrm{i}$ 是相等的，当且仅当 $a=c$ 且 $b=d$.

这么定义是十分自然的，在此不做过多解释.

也就是说，我们可以用唯一的有序实数对 (a,b) 表示一个复数 $z=a+b\mathrm{i}$. 这样，联想到平面直角坐标系，我们可以发现 **复数集与平面直角坐标系中的点集一一对应**. 好了，我们找到了复数的一种几何意义.

那么这个平面直角坐标系就不再一般，因为平面直角坐标系中的点具有了特殊意义——表示一个复数，所以我们把这样的平面直角坐标系称为 **复平面**，x 轴称为 **实轴**，y 轴称为 **虚轴**. 我们进一步地说：**复数集与复平面内所有的点所构成的集合是一一对应的**.

我们考虑到学过的平面向量的知识，发现向量的坐标表示也是一个有序实数对 (a,b)，显然，复数 $z=a+b\mathrm{i}$ 对应复平面内的点 $Z(a,b)$，那么它还对应平面向量 $\overrightarrow{OZ}=(a,b)$，于是我们又找到了复数的另一种几何意义：**复数集与复平面内的向量所构成的集合是一一对应的（实数 0 与零向量对应）**.

于是，我们由向量的知识迁移到复数上来，定义 **复数的模** 就是复数所对应的向量的模. 复数 $z=a+b\mathrm{i}$ 的模 $\mid z\mid=\sqrt{a^2+b^2}$.

于是为了方便，我们常把复数 $z=a+b\mathrm{i}$ 称为点 Z 或向量 $\overrightarrow{OZ}$，并规定相等的向量表示同一个复数.

并且由向量的知识我们发现，虚数不可以比较大小（但是实数是可以的）.

加法与减法

对复数 $z_1=a+b\mathrm{i},z_2=c+d\mathrm{i}$，定义加法规则如下：

$$z_1+z_2=(a+c)+(b+d)\mathrm{i}$$

很明显，两个复数的和仍为复数.

考虑到向量的加法运算，我们发现复数的加法运算符合向量的加法运算法则，这同样证明了复数的几何意义的正确性.

同样可以验证，复数的加法满足 **交换律** 和 **结合律**. 即：

$$z_1+z_2=z_2+z_1$$

$$(z_1+z_2)+z_3=z_1+(z_2+z_3)$$

减法作为加法的逆运算，我们可以通过加法法则与复数相等的定义来推导出减法法则：

$$z_1-z_2=(a-c)+(b-d)\mathrm{i}$$

这同样符合向量的减法运算.

乘法、除法与共轭

对复数 $z_1=a+b\mathrm{i},z_2=c+d\mathrm{i}$，定义乘法规则如下：

$$\begin{aligned}z_1z_2&=(a+b\mathrm{i})(c+d\mathrm{i})\\&=ac+bc\mathrm{i}+ad\mathrm{i}+bd\mathrm{i}^2\\&=(ac-bd)+(bc+ad)\mathrm{i}\end{aligned}$$

可以看出，两个复数相乘类似于两个多项式相乘，只需要把 i^2 换成 -1，并将实部与虚部分别合并即可.

复数的乘法与向量的向量积形式类似.

易得复数乘法满足 **交换律**，**结合律** 和 **对加法的分配律**，即：

- $z_1z_2=z_2z_1$
- $(z_1z_2)z_3=z_1(z_2z_3)$
- $z_1(z_2+z_3)=z_1z_2+z_1z_3$

由于满足运算律，我们可以发现实数域中的 **乘法公式在复数域中同样适用**.

除法运算是乘法运算的逆运算，我们可以推导一下：

$$\begin{aligned}\frac{a+b\mathrm{i}}{c+d\mathrm{i}}&=\frac{(a+b\mathrm{i})(c-d\mathrm{i})}{(c+d\mathrm{i})(c-d\mathrm{i})}\\&=\frac{ac+bd}{c^2+d^2}+\frac{bc-ad}{c^2+d^2}\mathrm{i}(c+d\mathrm{i}\neq0)\end{aligned}$$

由于向量没有除法，这里不讨论与向量的关系.

为了分母实数化，我们乘了一个 $c-d\mathrm{i}$，这个式子很有意义.

对复数 $z=a+b\mathrm{i}$，称 $a-b\mathrm{i}$ 为 z 的 **共轭复数**，通常记为 $\bar{z}$. 我们可以发现，若两个复数互为共轭复数，那么它们 **关于实轴对称**.

对复数 z,w，复数共轭有如下性质

- $z\cdot\bar{z}=|\,z\,|^2$
- $\bar{\bar{z}}=z$
- $\mathrm{Re}(z)=\dfrac{z+\bar{z}}{2}$，$\mathrm{Im}(z)=\dfrac{z-\bar{z}}{2}$
- $\overline{z\pm w}=\bar{z}\pm\bar{w}$
- $\overline{zw}=\bar{z}\bar{w}$
- $\overline{z/w}=\bar{z}/\bar{w}$

辐角和辐角主值

如果设定实数单位 1 作为水平正方向，虚数单位 i 作为竖直正方向，得到的就是直角坐标视角下的复平面.

表示复数 z 的位置，也可以借助于极坐标 (r,θ) 确定. 前文已经提到了 r 为复数 z 的模.

从实轴正向到 **非零** 复数 $z=x+\mathrm{i}y$ 对应向量的夹角 θ 满足关系：

$$\tan\theta=\frac{y}{x}$$

称为复数 z 的 **辐角**，记为：

$$\theta=\arg z$$

任一个 **非零** 复数 z 有无穷多个辐角，故 $\arg z$ 事实上是一个集合. 借助开头大写的 $\mathrm{Arg}\,z$ 表示 **其中一个特定值**，满足条件：

$$-\pi<\mathrm{Arg}\,z\le\pi$$

称 $\mathrm{Arg}\,z$ 为 **辐角主值** 或 **主辐角**. 辐角就是辐角主值基础上加若干整数个（可以为零或负整数）$2k\pi$，即 $\arg z=\{\mathrm{Arg}\,z+2k\pi\mid k\in\mathbf{Z}\}$.

需要注意的是两个辐角主值相加后不一定还是辐角主值，而两个辐角相加一定还是合法的辐角.

称模小于 1 的复数，在复平面上构成的图形为 **单位圆**. 称模等于 1 的复数为 **单位复数**，全体单位复数在复平面上构成的图形为 **单位圆周**. 在不引起混淆的情况下，有时单位圆周也简称单位圆.

在极坐标的视角下，复数的乘除法变得很简单. 复数乘法，模相乘，辐角相加. 复数除法，模相除，辐角相减.

欧拉公式

欧拉公式（Euler's formula）[1]

对任意实数 x，有

$$\mathrm{e}^{\mathrm{i}x}=\cos x+\mathrm{i}\sin x$$

在补充 复指数函数与复三角函数→9@p1079 的定义后，该公式可推广至全体复数.

指数函数与三角函数

对于复数 $z=x+\mathrm{i}y$，函数 $f(z)=\mathrm{e}^x(\cos y+\mathrm{i}\sin y)$ 满足 $f(z_1+z_2)=f(z_1)f(z_2)$. 由此给出 **复指数函数** 的定义：

$$\exp z=\mathrm{e}^x(\cos y+\mathrm{i}\sin y)$$

[1]有关欧拉公式的更多介绍，可以参考两个视频：欧拉公式与初等群论[1]、微分方程概论 - 第五章：在 3.14 分钟内理解 $\mathrm{e}^{\mathrm{i}\pi}$[2].

复指数函数在实数集上与实指数函数的定义完全一致. 在复平面上拥有性质:

- 模恒正: $|\exp z| = \exp x > 0$.
- 辐角: $\arg(\exp z) = \{y + 2k\pi \mid k \in \mathbf{Z}\}$.
- 加法定理: $\exp(z_1 + z_2) = \exp(z_1)\exp(z_2)$.
- 周期性: $\exp z$ 是以 $2\pi\mathrm{i}$ 为基本周期的周期函数. 如果一个函数 $f(z)$ 的周期是某一周期的整倍数, 称该周期为 **基本周期**.

复三角函数（也简称 **三角函数**）的定义如下:

$$\cos z = \frac{\exp(\mathrm{i}z) + \exp(-\mathrm{i}z)}{2}$$

$$\sin z = \frac{\exp(\mathrm{i}z) - \exp(-\mathrm{i}z)}{2\mathrm{i}}$$

若取 $z \in \mathbf{R}$, 则由 欧拉公式→9@p1079 有:

$$\cos z = \mathrm{Re}(\mathrm{e}^{\mathrm{i}z})$$

$$\sin z = \mathrm{Im}(\mathrm{e}^{\mathrm{i}z})$$

复三角函数在实数集上与实三角函数的定义完全一致. 在复平面上拥有性质:

- 奇偶性: 正弦函数是奇函数, 余弦函数是偶函数.
- 三角恒等式: 通常的三角恒等式都成立, 例如平方和为 1, 或者角的和差公式等.
- 周期性: 正弦与余弦函数以 2π 为基本周期.
- 零点: 实正弦与实余弦函数的全体零点, 构成了复正弦与复余弦函数的全体零点. 这个推广没有引进新的零点.
- 模的无界性: 复正弦与复余弦函数, 模长可以大于任意给定的正数, 不再像实正弦与实余弦函数一样被限制在 1 的范围内.

复数的三种形式

借助直角坐标系的视角以及极坐标系的视角, 可以写出复数的三种形式.

复数的 **代数形式** 用于表示任意复数.

$$z = x + y\mathrm{i}$$

代数形式用于计算复数的加减乘除四个运算比较方便.

复数的 **三角形式** 和 **指数形式**, 用于表示非零复数.

$$z = r(\cos\theta + \mathrm{i}\sin\theta) = r\exp(\mathrm{i}\theta)$$

这两种形式用于计算复数的乘除两个运算以及后面的运算较为方便. 如果只用高中见过的函数, 可以使用三角形式. 如果引入了复指数函数, 写成等价的指数形式会更加方便.

单位根

考察方程 $x^n = 1$ 在复数意义下的解. 显然, 这样的解有 n 个, 称这 n 个解都是 n **次单位（复）根**（n-th root of unity）. 根据复平面的知识, n 次单位根把单位圆 n 等分.

设 $\omega_n = \exp\dfrac{2\pi\mathrm{i}}{n}$ (即幅角为 $2\pi/n$ 的单位复数), 则 $x^n = 1$ 的解集表示为 $\{\omega_n^k \mid k = 0, 1\cdots, n-1\}$, 其中,

$$w_n^k = \exp\frac{2\pi k\mathrm{i}}{n} = \cos\frac{2\pi k}{n} + \mathrm{i}\sin\frac{2\pi k}{n}.$$

如果不加说明, 一般叙述中的 n 次单位根, 是指从 1 开始逆时针方向的第一个解, 即上述 ω_n, 其它解均可以用 ω_n 的幂表示.

为什么通常提到 n 次单位根，总是特指第一个？

主要是为了应用时方便．所有 n 次单位根都可以表示为第一个 n 次单位根 ω_n 的幂次；而且，对于任意 $k < n$，复数 ω_n 都不是 k 次单位根．

本原单位根

事实上，n 次单位根中满足类似性质的不止 ω_n 一个．称集合

$$\{\omega_n^k \mid 0 \le k < n,\ \gcd(n,k) = 1\}$$

中的元素为 n **次本原单位根**（n-th primitive root of unity）．根据上述表达式可知，全体 n 次本原单位根共有 $\varphi(n)$ 个，其中，$\varphi(n)$ 为 欧拉函数→9.11.5@p1196．

任意一个本原单位根 ω，都与上述 ω_n 具有相同的性质：对于任意的 $0 < k < n$，ω 的 k 次幂不为 1，也就是说，ω 不是 k 次单位根．因此，借助任意一个本原单位根，都可以生成全体单位根．

为了理解 n 次本原单位根的结构，需要考虑单位根的如下性质：

性质

对于整数 n 和 k，设 $d = \gcd(n,k)$，有 $\omega_n^k = \omega_{n/d}^{k/d}$．

证明

直接计算可知

$$w_n^k = \exp\frac{2\pi k\mathrm{i}}{n} = \exp\frac{2\pi(k/d)\mathrm{i}}{n/d} = \omega_{n/d}^{k/d}.$$

这说明，只要 $\gcd(n,k) \neq 1$，那么，ω_n^k 就一定是 $\dfrac{n}{\gcd(n,k)}$ 次（本原）单位根．因此，满足前述性质的单位根 ω_n^k 一定是满足 $\gcd(n,k) = 1$．这正是本原单位根具有上述定义的原因．

另外，作为这些分析的简单推论，有：

定理

当 k 遍历 n 的因数，所有 k 次本原单位根恰构成 n 次单位根的一个划分．而且，对于 $\ell \perp n$，映射 $x \mapsto x^\ell$ 给出 n 次单位根之间的双射，且保持上述划分不变：它将 $k \mid n$ 次本原单位根仍然映射到 k 次本原单位根．

尽管本原单位根有很多选择，但是由于第一个根 ω_n 形式最为简单，算法竞赛中还是 ω_n 最为常用．对于部分场景，为提高计算效率，还可以考虑用某一模数下的 本原单位根→9.11.20@p1291 代替复数域中的 ω_n．

编程语言中的复数

C 中的复数

在 C99 标准中，有 `<complex.h>` 头文件．

在 `<complex.h>` 头文件中，提供了 `double complex`、`float complex` 和 `long double complex` 三种类型．

算术运算符'+'、'-'、'*'和'/'，可以用于浮点数和复数的任意混合．当表达式两端有一个为复数时，计算结果为复数．

头文件 `<complex.h>` 提供了虚数单位 `I`，引入此头文件时，大写字母 `I` 不可以作为变量名使用．

对于单个复数，`<complex.h>` 提供了若干操作：`creal` 函数用于提取实部，`cimag` 函数用于提取虚部，`cabs` 函数用于计算模，`carg` 函数用于计算辐角主值.

所有的函数根据类型不同，都有三个. 例如 `creal` 函数有 `creal`、`crealf`、`creall` 三个，用于处理对应的 `double`、`float` 和 `long double` 三种类型. 末尾什么都不带的默认处理 `double` 类型. 以下所有函数均遵从此规律，不再特别说明.

这些函数返回值都是一般的浮点数. 可以将普通浮点数直接赋值给复数，但是不可以将复数直接赋值给浮点数，而是需要使用上述提取操作.

函数 `conj` 用于计算共轭复数，返回值是复数.

函数 `cexp` 计算复指数，`clog` 计算对数主值，`csin` 计算正弦，`ccos` 计算余弦，`ctan` 计算正切.

函数 `cpow` 计算幂函数，`csqrt` 计算平方根，`casin` 计算反正弦，`cacos` 计算反余弦，`catan` 计算反正切. 这部分函数计算的全部都是多值函数的主值.

C++ 中的复数

在 C 里面的 `<ctype.h>`，到 C++ 会变成 `<cctype>`，几乎所有的头文件遵从这个命名规律.

但是，`<complex.h>` 不遵守，C++ 没有 `<ccomplex>` 头文件. C++ 的复数直接是 `<complex>`，并且装的东西和 C 完全不一样.

很有趣. 这是因为，在 C++ 的第一个版本 C++98，即已经有了 `<complex>`，而 C 语言在 C99 才添加.

在 C++ 中，复数类型定义使用 `complex<float>`、`complex<double>` 和 `complex<long double>`. 由于面向对象的多态性，下面函数的名字都是唯一的，无需 f 或 l 的后缀.

一个复数对象拥有成员函数 `real` 和 `imag`，可以访问实部和虚部.

一个复数对象拥有非成员函数 `real`、`imag`、`abs`、`arg`，返回实部、虚部、模和辐角.

一个复数对象还拥有非成员函数：`norm` 为模的平方，`conj` 为共轭复数.

一个复数对象还拥有非成员函数 `exp`、`log`（底为 e 的对数主值）、`log10`（底为 10 的对数主值，C 中没有）、`pow`、`sqrt`、`sin`、`cos`、`tan`，含义与 C 中的含义相同.

在 C++14 及以后的版本中，定义了 字面量运算符 `std::literals::complex_literals::""if, ""i, ""il`[3]. 例如输入 `100if`、`100i` 和 `100il`，三者将分别返回 `std::complex<float>{0.0f, 100.0f}`、`std::complex<double>{0.0, 100.0}` 以及 `std::complex<long double>{0.0l, 100.0l}`. 这使得我们可以方便地书写形如 `auto z = 4.0 + 3i` 的复数声明.

参考资料与链接

- Complex number - Wikipedia[4]
- Euler's formula - Wikipedia[5]
- Complex number arithmetic - cppreference.com[6]
- std::complex - cppreference.com[7]

[1] 欧拉公式与初等群论

[2] 微分方程概论 - 第五章：在 3.14 分钟内理解 $e^{i\pi}$

[3] 字面量运算符 `std::literals::complex_literals::""i, ""il`

[4] Complex number - Wikipedia

[5] Euler's formula - Wikipedia

[6] Complex number arithmetic - cppreference.com

[7] std::complex - cppreference.com

9.11 数论

9.11.1 数论基础

本文对于数论的开头部分做一个简介.

整除

定义

设 $a,b\in\mathbf{Z}$，$a\neq 0$. 如果 $\exists q\in\mathbf{Z}$，使得 $b=aq$，那么就说 b 可被 a **整除**，记作 $a\mid b$；b 不被 a 整除记作 $a\nmid b$.

整除的性质：

- $a\mid b\iff -a\mid b\iff a\mid -b\iff \mid a\mid\mid b\mid$
- $a\mid b\wedge b\mid c\Longrightarrow a\mid c$
- $a\mid b\wedge a\mid c\iff \forall x,y\in\mathbf{Z},a\mid(xb+yc)$
- $a\mid b\wedge b\mid a\Longrightarrow b=\pm a$
- 设 $m\neq 0$，那么 $a\mid b\iff ma\mid mb$.
- 设 $b\neq 0$，那么 $a\mid b\Longrightarrow \mid a\mid\leq\mid b\mid$.
- 设 $a\neq 0,b=qa+c$，那么 $a\mid b\iff a\mid c$.

约数

定义

若 $a\mid b$，则称 b 是 a 的 **倍数**，a 是 b 的 **约数**.

0 是所有非 0 整数的倍数. 对于整数 $b\neq 0$，b 的约数只有有限个.

平凡约数（平凡因数）：对于整数 $b\neq 0$，± 1、$\pm b$ 是 b 的平凡约数. 当 $b=\pm 1$ 时，b 只有两个平凡约数.

对于整数 $b\neq 0$，b 的其他约数称为真约数（真因数、非平凡约数、非平凡因数）.

约数的性质：

- 设整数 $b\neq 0$. 当 d 遍历 b 的全体约数的时候，$\dfrac{b}{d}$ 也遍历 b 的全体约数.
- 设整数 $b>0$，则当 d 遍历 b 的全体正约数的时候，$\dfrac{b}{d}$ 也遍历 b 的全体正约数.

在具体问题中，**如果没有特别说明，约数总是指正约数.**

带余数除法

余数

设 a,b 为两个给定的整数，$a\neq 0$. 设 d 是一个给定的整数. 那么，一定存在唯一的一对整数 q 和 r，满足 $b=qa+r,d\leq r<\mid a\mid+d$.

无论整数 d 取何值，r 统称为余数. $a\mid b$ 等价于 $a\mid r$.

一般情况下，d 取 0，此时等式 $b=qa+r,0\leq r<\mid a\mid$ 称为带余数除法（带余除法）. 这里的余数 r 称为最小非负余数.

余数往往还有两种常见取法：

- 绝对最小余数：d 取 a 的绝对值的一半的相反数. 即 $b=qa+r,-\dfrac{\mid a\mid}{2}\leq r<\mid a\mid-\dfrac{\mid a\mid}{2}$.
- 最小正余数：d 取 1. 即 $b=qa+r,1\leq r<\mid a\mid+1$.

带余数除法的余数只有最小非负余数. **如果没有特别说明，余数总是指最小非负余数.**

余数的性质：

- 任一整数被正整数 a 除后，余数一定是且仅是 0 到 $(a-1)$ 这 a 个数中的一个.
- 相邻的 a 个整数被正整数 a 除后，恰好取到上述 a 个余数. 特别地，一定有且仅有一个数被 a 整除.

最大公约数与最小公倍数

关于公约数、公倍数、最大公约数与最小公倍数，四个名词的定义，见 最大公约数→9.11.4@p1188.

> ⚠ **Warning**
>
> 一些作者认为 0 和 0 的最大公约数无定义，其余作者一般将其视为 0. C++ STL 的实现中采用后者，即认为 0 和 0 的最大公约数为 0[1].

最大公约数有如下性质：

- $(a_1,...,a_n)=(|a_1|,...,|a_n|)$;
- $(a,b)=(b,a)$;
- 若 $a\neq 0$，则 $(a,0)=(a,a)=|a|$;
- $(bq+r,b)=(r,b)$;
- $(a_1,...,a_n)=((a_1,a_2),a_3,...,a_n)$. 进而 $\forall 1<k<n-1$，$(a_1,...,a_n)=((a_1,...,a_k),(a_{k+1},...,a_n))$;
- 对不全为 0 的整数 $a_1,...,a_n$ 和非零整数 m，$(ma_1,...,ma_n)=|m|(a_1,...,a_n)$;
- 对不全为 0 的整数 $a_1,...,a_n$，若 $(a_1,...,a_n)=d$，则 $(a_1/d,...,a_n/d)=1$;
- $(a^n,b^n)=(a,b)^n$.

最大公约数还有如下与互素相关的性质：

- 若 $b\mid ac$ 且 $(a,b)=1$，则 $b\mid c$;
- 若 $b\mid c$、$a\mid c$ 且 $(a,b)=1$，则 $ab\mid c$;
- 若 $(a,b)=1$，则 $(a,bc)=(a,c)$;
- 若 $(a_i,b_j)=1$，$\forall 1\leq i\leq n,1\leq j\leq m$，则 $\left(\prod_i a_i,\prod_j b_j\right)=1$. 特别地，若 $(a,b)=1$，则 $(a^n,b^m)=1$;
- 对整数 $a_1,...,a_n$，若 $\exists v\in\mathbf{Z}$，$\prod_i a_i=v^m$，且 $(a_i,a_j)=1$，$\forall i\neq j$，则 $\forall 1\leq i\leq n$，$\sqrt[m]{a_i}\in\mathbf{Z}$.

最小公倍数有如下性质：

- $[a_1,...,a_n]=[|a_1|,...,|a_n|]$;
- $[a,b]=[b,a]$;
- 若 $a\neq 0$，则 $[a,1]=[a,a]=|a|$;
- 若 $a\mid b$，则 $[a,b]=|b|$;
- $[a_1,...,a_n]=[[a_1,a_2],a_3,...,a_n]$. 进而 $\forall 1<k<n-1$，$[a_1,...,a_n]=[[a_1,...,a_k],[a_{k+1},...,a_n]]$;
- 若 $a_i\mid m$，$\forall 1\leq i\leq n$，则 $[a_1,...,a_n]\mid m$;
- $[ma_1,...,ma_n]=|m|[a_1,...,a_n]$;
- $[a,b,c][ab,bc,ca]=[a,b][b,c][c,a]$;
- $[a^n,b^n]=[a,b]^n$.

最大公约数和最小公倍数可以组合出很多奇妙的等式，如：

- $(a,b)[a,b]=|ab|$;
- $(ab,bc,ca)[a,b,c]=|abc|$;
- $\dfrac{(a,b,c)^2}{(a,b)(b,c)(a,c)}=\dfrac{[a,b,c]^2}{[a,b][b,c][a,c]}$.

这些性质均可通过定义或 唯一分解定理→9.11@p1160 证明，其中使用唯一分解定理的证明更容易理解.

[1] std::gcd - cppreference.com[1]

互素

定义

若 $(a_1, a_2) = 1$，则称 a_1 和 a_2 **互素（既约）**.

若 $(a_1, ..., a_k) = 1$，则称 $a_1, ..., a_k$ **互素（既约）**.

多个整数互素，不一定两两互素. 例如 6、10 和 15 互素，但是任意两个都不互素.

互素的性质与最大公约数理论：裴蜀定理（Bézout's identity）. 见 裴蜀定理→9.11.8@p1218.

辗转相除法

辗转相除法是一种算法，也称 Euclid 算法. 见 最大公约数→9.11.4@p1188.

素数与合数

关于素数的算法见 素数→9.11.3@p1179.

定义

设整数 $p \neq 0, \pm 1$. 如果 p 除了平凡约数外没有其他约数，那么称 p 为 **素数（不可约数）**.

若整数 $a \neq 0, \pm 1$ 且 a 不是素数，则称 a 为 **合数**.

p 和 $-p$ 总是同为素数或者同为合数. **如果没有特别说明，素数总是指正的素数.**

整数的因数是素数，则该素数称为该整数的素因数（素约数）.

素数与合数的简单性质：

- 大于 1 的整数 a 是合数，等价于 a 可以表示为整数 d 和 e（$1 < d, e < a$）的乘积.
- 如果素数 p 有大于 1 的约数 d，那么 $d = p$.
- 大于 1 的整数 a 一定可以表示为素数的乘积.
- 对于合数 a，一定存在素数 $p \leq \sqrt{a}$ 使得 $p \mid a$.
- 素数有无穷多个.
- 所有大于 3 的素数都可以表示为 $6n \pm 1$ 的形式[2].

算术基本定理

算术基本引理

设 p 是素数，$p \mid a_1 a_2$，那么 $p \mid a_1$ 和 $p \mid a_2$ 至少有一个成立.

算术基本引理的逆命题稍加修改也可以得到素数的另一种定义.

素数的另一种定义

对整数 $p \neq 0, \pm 1$，若对任意满足 $p \mid a_1 a_2$ 的整数 a_1, a_2 均有 $p \mid a_1$ 或 $p \mid a_2$ 成立，则称 p 是素数.

Tip

这个定义的动机可以从 素理想→9.16.3@p1830 中找到.

[2] Are all primes (past 2 and 3) of the forms 6n+1 and 6n-1?[2]

算术基本定理（唯一分解定理）

设正整数 a，那么必有表示：

$$a = p_1 p_2 \cdots p_s$$

其中 $p_j (1 \le j \le s)$ 是素数．并且在不计次序的意义下，该表示唯一．

标准素因数分解式

将上述表示中，相同的素数合并，可得：

$$a = p_1^{\alpha_1} p_2^{\alpha_2} \cdots p_s^{\alpha_s}, p_1 < p_2 < \cdots < p_s$$

称为正整数 a 的标准素因数分解式．

算术基本定理和算术基本引理，两个定理是等价的．

同余

定义

设整数 $m \neq 0$．若 $m \mid (a-b)$，称 m 为 **模数（模）**，a 同余于 b 模 m，b 是 a 对模 m 的 **剩余**．记作 $a \equiv b \pmod{m}$．

否则，a 不同余于 b 模 m，b 不是 a 对模 m 的剩余．记作 $a \not\equiv b \pmod{m}$．

这样的等式，称为模 m 的同余式，简称 **同余式**．

根据整除的性质，上述同余式也等价于 $a \equiv b \pmod{(-m)}$．

后文中，如果没有特别说明，模数总是 **正整数**．

式中的 b 是 a 对模 m 的剩余，这个概念与余数完全一致．通过限定 b 的范围，相应的有 a 对模 m 的最小非负剩余、绝对最小剩余、最小正剩余．

同余的性质：

- 同余是 等价关系→9.20@p1951，即同余具有
  - 自反性：$a \equiv a \pmod{m}$．
  - 对称性：若 $a \equiv b \pmod{m}$，则 $b \equiv a \pmod{m}$．
  - 传递性：若 $a \equiv b \pmod{m}, b \equiv c \pmod{m}$，则 $a \equiv c \pmod{m}$．
- 线性运算：若 $a, b, c, d \in \mathbf{Z}, m \in \mathbf{N}^*, a \equiv b \pmod{m}, c \equiv d \pmod{m}$ 则有：
  - $a \pm c \equiv b \pm d \pmod{m}$．
  - $a \times c \equiv b \times d \pmod{m}$．
- 设 $f(x) = \sum_{i=0}^{n} a_i x^i$ 和 $g(x) = \sum_{i=0}^{n} b_i x^i$ 是两个整系数多项式，$m \in \mathbf{N}^*$，且 $a_i \equiv b_i \pmod{m}$，$0 \le i \le n$，则对任意整数 x 均有 $f(x) \equiv g(x) \pmod{m}$．进而若 $s \equiv t \pmod{m}$，则 $f(s) \equiv g(t) \pmod{m}$．
- 若 $a, b \in \mathbf{Z}, k, m \in \mathbf{N}^*, a \equiv b \pmod{m}$，则 $ak \equiv bk \pmod{mk}$．
- 若 $a, b \in \mathbf{Z}, d, m \in \mathbf{N}^*, d \mid a, d \mid b, d \mid m$，则当 $a \equiv b \pmod{m}$ 成立时，有 $\frac{a}{d} \equiv \frac{b}{d} \left(\mod \frac{m}{d}\right)$．
- 若 $a, b \in \mathbf{Z}, d, m \in \mathbf{N}^*, d \mid m$，则当 $a \equiv b \pmod{m}$ 成立时，有 $a \equiv b \pmod{d}$．
- 若 $a, b \in \mathbf{Z}, d, m \in \mathbf{N}^*$，则当 $a \equiv b \pmod{m}$ 成立时，有 $(a, m) = (b, m)$．若 d 能整除 m 及 a, b 中的一个，则 d 必定能整除 a, b 中的另一个．

还有性质是乘法逆元．见 乘法逆元→9.11.10@p1229．

同余类与剩余系

为方便讨论，对集合 A, B 和元素 r，我们引入如下记号：

- $r+A:=\{r+a:a\in A\}$;
- $rA:=\{ra:a\in A\}$;
- $A+B:=\{a+b:a\in A,b\in B\}$;
- $AB:=\{ab:a\in A,b\in B\}$.

同余类

对非零整数 m，把全体整数分成 $|m|$ 个两两不交的集合，且同一个集合中的任意两个数模 m 均同余，我们把这 $|m|$ 个集合均称为模 m 的 **同余类** 或 **剩余类**. 用 $r \bmod m$ 表示含有整数 r 的模 m 的同余类.

不难证明对任意非零整数 m，上述划分方案一定存在且唯一.

由同余类的定义可知:

- $r \bmod m=\{r+km:k\in\mathbf{Z}\}$;
- $r \bmod m=s \bmod m\iff r\equiv s\pmod m$;
- 对任意 $r,s\in\mathbf{Z}$，要么 $r \bmod m=s \bmod m$，要么 $(r \bmod m)\cap(s \bmod m)=\varnothing$;
- 若 $m_1\mid m$，则对任意整数 r 均有 $r+m\mathbf{Z}\subseteq r+m_1\mathbf{Z}$.

注意到同余是等价关系，所以同余类即为同余关系的等价类.

我们把模 m 的同余类全体构成的集合记为 $\mathbf{Z}_m$，即

$$\mathbf{Z}_m:=\{r \bmod m:0\le r<m\}$$

不难发现:

- 对任意整数 a，$a+\mathbf{Z}_m=\mathbf{Z}_m$;
- 对任意与 m 互质的整数 b，$b\mathbf{Z}_m=\mathbf{Z}_m$.

由 商群→9.16.2@p1812 的定义可知 $\mathbf{Z}_m=\mathbf{Z}/m\mathbf{Z}$，所以有时我们也会用 $\mathbf{Z}/m\mathbf{Z}$ 表示 $\mathbf{Z}_m$.

由 抽屉原理→9.13.2@p1618 可知:

- 任取 $m+1$ 个整数，必有两个整数模 m 同余.
- 存在 m 个两两模 m 不同余的整数.

由此我们给出完全剩余系的定义:

（完全）剩余系

对 m 个整数 $a_1,a_2,...,a_m$，若对任意的数 x，有且仅有一个数 a_i 使得 x 与 a_i 模 m 同余，则称这 m 个整数 $a_1,a_2,...,a_m$ 为模 m 的 **完全剩余系**，简称 **剩余系**.

我们还可以定义模 m 的:

- 最小非负（完全）剩余系：$0,...,m-1$;
- 最小正（完全）剩余系：$1,...,m$;
- 绝对最小（完全）剩余系：$-\lfloor m/2\rfloor,...,-\lfloor -m/2\rfloor-1$;
- 最大非正（完全）剩余系：$-m+1,...,0$;
- 最大负（完全）剩余系：$-m,...,-1$.

若无特殊说明，一般我们只用最小非负剩余系.

我们注意到如下命题成立:

- 在模 m 的任意一个同余类中，任取两个整数 a_1,a_2 均有 $(a_1,m)=(a_2,m)$.

考虑同余类 $r \bmod m$，若 $(r,m)=1$，则该同余类的所有元素均与 m 互质，这说明我们也许可以通过类似方式得知所有与 m 互质的整数构成的集合的结构.

既约同余类

对同余类 $r \bmod m$，若 $(r,m)=1$，则称该同余类为 **既约同余类** 或 **既约剩余类**.

我们把模 m 既约剩余类的个数记作 $\varphi(m)$，称其为 Euler 函数→9.11.5@p1196.

我们把模 m 的既约同余类全体构成的集合记为 $\mathbf{Z}_m^*$，即

$$\mathbf{Z}_m^* := \{r \bmod m : 0 \le r < m, (r,m)=1\}$$

Warning

对于任意的整数 a 和与 m 互质的整数 b，$b\mathbf{Z}_m^* = \mathbf{Z}_m^*$，但是 $a+\mathbf{Z}_m^*$ 不一定为 $\mathbf{Z}_m^*$. 这一点与 $\mathbf{Z}_m$ 不同.

由 抽屉原理→9.13.2@p1618 可知:

- 任取 $\varphi(m)+1$ 个与 m 互质的整数，必有两个整数模 m 同余.
- 存在 $\varphi(m)$ 个与 m 互质且两两模 m 不同余的整数.

由此我们给出既约剩余系的定义:

既约剩余系

对 $t=\varphi(m)$ 个整数 $a_1,a_2,...,a_t$，若 $(a_i,m)=1,\ \forall 1\le i\le t$，且对任意满足 $(x,m)=1$ 的数 x，有且仅有一个数 a_i 使得 x 与 a_i 模 m 同余，则称这 t 个整数 $a_1,a_2,...,a_t$ 为模 m 的 **既约剩余系**、**缩剩余系** 或 **简化剩余系**.

类似地，我们也可以定义最小非负既约剩余系等概念.

若无特殊说明，一般我们只用最小非负既约剩余系.

剩余系的复合

对正整数 m，我们有如下定理:

- 若 $m=m_1m_2,\ 1\le m_1,m_2$，令 Z_{m_1},Z_{m_2} 分别为模 m_1,m_2 的 **完全** 剩余系，则对任意与 m_1 互质的 a 均有:

  $Z_m = aZ_{m_1} + m_1Z_{m_2}$.

  为模 m 的 **完全** 剩余系. 进而，若 $m=\prod_{i=1}^k m_i,\ 1\le m_1,m_2,...,m_k$，令 $Z_{m_1},...,Z_{m_k}$ 分别为模 $m_1,...,m_k$ 的 **完全** 剩余系，则:

  $$Z_m = \sum_{i=1}^{k}\left(\prod_{j=1}^{i-1} m_j\right) Z_{m_i}.$$

  为模 m 的 **完全** 剩余系.

证明

只需证明对任意满足 $ax+m_1y \equiv ax'+m_1y' \pmod{m_1m_2}$ 的 $x,x'\in Z_{m_1},\ y,y'\in Z_{m_2}$，都有:

$$ax+m_1y = ax'+m_1y'.$$

实际上，由 $m_1 \mid m_1m_2$，我们有 $ax+m_1y \equiv ax'+m_1y' \pmod{m_1}$，进而 $ax\equiv ax' \pmod{m_1}$，由 $(a,m_1)=1$ 可知 $x\equiv x' \pmod{m_1}$，进而有 $x=x'$.

进一步，$m_1y\equiv m_1y' \pmod{m_1m_2}$，则 $y\equiv y' \pmod{m_2}$，即 $y=y'$.

因此，

$$ax+m_1y = ax'+m_1y'.$$

- 若 $m=m_1m_2$，$1\le m_1,m_2,(m_1,m_2)=1$，令 $Z_{m_1}^*,Z_{m_2}^*$ 分别为模 m_1,m_2 的 **既约** 剩余系，则：$Z_m^*=m_2Z_{m_1}^*+m_1Z_{m_2}^*$.

  为模 m 的 **既约** 剩余系.

Tip

该定理等价于证明 Euler 函数为 积性函数→9.11@p1160.

证明

令 Z_{m_1},Z_{m_2} 分别为模 m_1,m_2 的完全剩余系，我们已经证明了

$$Z_m=m_2Z_{m_1}+m_1Z_{m_2}$$

为模 m 的完全剩余系. 令 $M=\{a\in Z_m:(a,m)=1\}\subseteq Z_m$，显然 M 为模 m 的既约剩余系，所以我们只需证明 $M=Z_m^*$ 即可.

显然 $Z_m^*\subseteq Z_m$.

任取 $m_2x+m_1y\in M$，其中 $x\in Z_{m_1}$ 且 $y\in Z_{m_2}$，有 $(m_2x+m_1y,m_1m_2)=1$，由 $(m_1,m_2)=1$ 可得

$$1=(m_2x+m_1y,m_1)=(m_2x,m_1)=(x,m_1),$$
$$1=(m_2x+m_1y,m_2)=(m_1y,m_2)=(y,m_2).$$

因此可得 $x\in Z_{m_1}^*$ 且 $y\in Z_{m_2}^*$，即 $M\subseteq Z_m^*$.

任取 $m_2x+m_1y\in Z_m^*$，其中 $x\in Z_{m_1}^*$ 且 $y\in Z_{m_2}^*$，有 $(x,m_1)=1$ 且 $(y,m_2)=1$，由 $(m_1,m_2)=1$ 可得

$$(m_2x+m_1y,m_1)=(m_2x,m_1)=(x,m_1)=1,$$
$$(m_2x+m_1y,m_2)=(m_1y,m_2)=(x,m_2)=1,$$

因此可得 $(m_2x+m_1y,m_1m_2)=1$，即 $Z_m^*\subseteq M$.

综上所述，

$$Z_m^*=m_2Z_{m_1}^*+m_1Z_{m_2}^*.$$

为模 m 的 **既约** 剩余系.

数论函数

数论函数（也称算术函数）指定义域为正整数的函数. 数论函数也可以视作一个数列.

积性函数

定义

在数论中，若函数 $f(n)$ 满足 $f(1)=1$，且 $f(xy)=f(x)f(y)$ 对任意互质的 $x,y\in\mathbf{N}^*$ 都成立，则 $f(n)$ 为 **积性函数**.

在数论中，若函数 $f(n)$ 满足 $f(1)=1$ 且 $f(xy)=f(x)f(y)$ 对任意的 $x,y\in\mathbf{N}^*$ 都成立，则 $f(n)$ 为 **完全积性函数**.

性质

若 $f(x)$ 和 $g(x)$ 均为积性函数，则以下函数也为积性函数：

$$h(x)=f(x^p)$$
$$h(x)=f^p(x)$$
$$h(x)=f(x)g(x)$$
$$h(x)=\sum_{d|x}f(d)g\left(\frac{x}{d}\right)$$

对正整数 x，设其唯一质因数分解为 $x=\prod p_i^{k_i}$，其中 p_i 为质数.

若 $F(x)$ 为积性函数，则有 $F(x)=\prod F(p_i^{k_i})$.

若 $F(x)$ 为完全积性函数，则有 $F(x)=\prod F(p_i^{k_i})=\prod F(p_i)^{k_i}$.

例子

- 单位函数：$\varepsilon(n)=[n=1]$.（完全积性）
- 恒等函数：$\mathrm{id}_k(n)=n^k$，$\mathrm{id}_1(n)$ 通常简记作 $\mathrm{id}(n)$.（完全积性）
- 常数函数：$1(n)=1$.（完全积性）
- 除数函数：$\sigma_k(n)=\sum_{d|n}d^k$. $\sigma_0(n)$ 通常简记作 $d(n)$ 或 $\tau(n)$，$\sigma_1(n)$ 通常简记作 $\sigma(n)$.
- 欧拉函数：$\varphi(n)=\sum_{i=1}^{n}[(i,n)=1]$.
- 莫比乌斯函数：$\mu(n)=\begin{cases}1 & n=1\\ 0 & \exists d>1,d^2|n\\ (-1)^{\omega(n)} & \text{otherwise}\end{cases}$，其中 $\omega(n)$ 表示 n 的本质不同质因子个数.

加性函数

定义

在数论中，若函数 $f(n)$ 满足 $f(1)=0$ 且 $f(xy)=f(x)+f(y)$ 对任意互质的 $x,y\in\mathbf{N}^*$ 都成立，则 $f(n)$ 为 **加性函数**.

在数论中，若函数 $f(n)$ 满足 $f(1)=0$ 且 $f(xy)=f(x)+f(y)$ 对任意的 $x,y\in\mathbf{N}^*$ 都成立，则 $f(n)$ 为 **完全加性函数**.

加性函数

本节中的加性函数指数论上的加性函数 (Additive function)，应与代数中的 Additive map 做区分.

性质

对正整数 x，设其唯一质因数分解为 $x=\prod p_i^{k_i}$，其中 p_i 为质数.

若 $F(x)$ 为加性函数，则有 $F(x)=\sum F(p_i^{k_i})$.

若 $F(x)$ 为完全加性函数，则有 $F(x)=\sum F(p_i^{k_i})=\sum F(p_i)\cdot k_i$.

例子

为方便叙述，令所有质数组成的集合为 $\mathbf{P}$.

- 素因数分解中 p 的重数：$\nu_p(n)=\max\{k\in\mathbf{N}:p^k\mid n\}$，其中，$p\in\mathbf{P}$.（完全加性）
- 所有质因子数目：$\Omega(n)=\sum_{p\in\mathbf{P}}\nu_p(n)$.（完全加性）
- 相异质因子数目：$\omega(n)=\sum_{p\in\mathbf{P}}[p\mid n]$.
- 所有质因子之和：$a_0(n)=\sum_{p\in\mathbf{P}}\nu_p(n)\cdot p$.（完全加性）
- 相异质因子之和：$a_1(n)=\sum_{p\in\mathbf{P}}[p\mid n]\cdot p$.

取整函数

对于实数 x，定义 **下取整函数**（floor function）和 **上取整函数**（ceiling function）分别为

$$\lfloor x\rfloor=\max\{k\in\mathbf{Z}:k\le x\},\ \lceil x\rceil=\min\{k\in\mathbf{Z}:k\ge x\}.$$

利用下取整函数，一个实数可以分解为整数部分和小数部分：$x=\lfloor x\rfloor+\{x\}$．其中，$\{x\}$ 表示 x 的小数部分．

取整函数有如下基本性质：（$x\in\mathbf{R},\ n\in\mathbf{Z}$）

- $x\in\mathbf{Z}\iff x=\lfloor x\rfloor=\lceil x\rceil$．
- $\lceil x\rceil-\lfloor x\rfloor=[x\notin\mathbf{Z}]$．
- $x-1<\lfloor x\rfloor\le x\le\lceil x\rceil<x+1$．
- $\lfloor -x\rfloor=-\lceil x\rceil,\ \lceil -x\rceil=-\lfloor x\rfloor$．
- $\lfloor x+n\rfloor=\lfloor x\rfloor+n,\ \lceil x+n\rceil=\lceil x\rceil+n$．
- $\lfloor x\rfloor$ 和 $\lceil x\rceil$ 都是关于 x 的单调弱增函数．

证明关于下（上）取整函数的等式经常用到如下等价形式：（$x\in\mathbf{R},\ n\in\mathbf{Z}$）

- $\lfloor x\rfloor=n\iff n\le x<n+1\iff x-1<n\le x$．
- $\lceil x\rceil=n\iff n-1<x\le n\iff x\le n<x+1$．

证明关于下（上）取整函数的不等式经常用到如下等价形式：（$x\in\mathbf{R},\ n\in\mathbf{Z}$）

- $x<n\iff\lfloor x\rfloor<n$．
- $n<x\iff n<\lceil x\rceil$．
- $x\le n\iff\lceil x\rceil\le n$．
- $n\le x\iff n\le\lfloor x\rfloor$．

涉及和、差的性质如下：（$x,y\in\mathbf{R}$）

- $\lfloor x\rfloor+\lfloor y\rfloor\le\lfloor x+y\rfloor\le\lfloor x\rfloor+\lfloor y\rfloor+1$，且恰有一个等号成立．
- $\lceil x\rceil+\lceil y\rceil-1\le\lceil x+y\rceil\le\lceil x\rceil+\lceil y\rceil$，且恰有一个等号成立．
- $\lfloor|\,x-y\,|\rfloor\le|\,\lfloor x\rfloor-\lfloor y\rfloor\,|\le\lceil|\,x-y\,|\rceil$．
- $\lfloor|\,x-y\,|\rfloor\le|\,\lceil x\rceil-\lceil y\rceil\,|\le\lceil|\,x-y\,|\rceil$．

涉及商的性质如下：（$x\in\mathbf{R},\ n\in\mathbf{Z},\ m\in\mathbf{Z}_+$）

- $\left\lfloor\dfrac{n}{m}\right\rfloor=\left\lfloor\dfrac{n+m-1}{m}\right\rfloor,\ \left\lceil\dfrac{n}{m}\right\rceil=\left\lceil\dfrac{n-m+1}{m}\right\rceil$．
- $\left\lfloor\dfrac{x+n}{m}\right\rfloor=\left\lfloor\dfrac{\lfloor x\rfloor+n}{m}\right\rfloor,\ \left\lceil\dfrac{x+n}{m}\right\rceil=\left\lceil\dfrac{\lceil x\rceil+n}{m}\right\rceil$．
- $\left\lfloor\dfrac{\lfloor x/n\rfloor}{m}\right\rfloor=\left\lfloor\dfrac{x}{nm}\right\rfloor,\ \left\lceil\dfrac{\lceil x/n\rceil}{m}\right\rceil=\left\lceil\dfrac{x}{nm}\right\rceil$．
- 对于 $x>0$，有 $\left\lfloor\dfrac{x}{m}\right\rfloor=\sum\limits_{k=1}^{\lfloor x\rfloor}[m\mid k]$．

其中，第二条和第三条性质都可以看作是如下结论的直接推论：

- 设 f 为连续单增函数，且只要 $f(x)\in\mathbf{Z}$，就有 $x\in\mathbf{Z}$，那么 $\lfloor f(x)\rfloor=\lfloor f(\lfloor x\rfloor)\rfloor,\ \lceil f(x)\rceil=\lceil f(\lceil x\rceil)\rceil$．

证明

由对称性，只需要证明第一个等式．如果 x 是整数，那么命题显然．否则，$\lfloor x\rfloor<x$．由 f 和下取整函数的单调性可知，$\lfloor f(x)\rfloor\ge\lfloor f(\lfloor x\rfloor)\rfloor$．如果等号不成立，那么设 $y=\lfloor f(x)\rfloor$，它满足 $\lfloor f(\lfloor x\rfloor)\rfloor<y\le\lfloor f(x)\rfloor$，这等价于 $f(\lfloor x\rfloor)<y\le f(x)$．由 f 的连续性可知，存在 $\lfloor x\rfloor<x_0\le x$ 使得 $f(x_0)=y$．因为 $y\in\mathbf{Z}$，所以 $x_0\in\mathbf{Z}$，这与 $\lfloor x\rfloor$ 的定义矛盾．故而，等号成立，即 $\lfloor f(x)\rfloor=\lfloor f(\lfloor x\rfloor)\rfloor$．

最后是一组关于带有取整函数的求和式的结论：（$x\in\mathbf{R},\ n\in\mathbf{Z},\ m\in\mathbf{Z}_+$）

- $n=\left\lfloor\dfrac{n}{2}\right\rfloor+\left\lceil\dfrac{n}{2}\right\rceil$．
- $n=\left\lfloor\dfrac{n}{m}\right\rfloor+\left\lfloor\dfrac{n+1}{m}\right\rfloor+\cdots+\left\lfloor\dfrac{n+m-1}{m}\right\rfloor$．
- $n=\left\lfloor\dfrac{n}{m}\right\rfloor+\left\lfloor\dfrac{n-1}{m}\right\rfloor+\cdots+\left\lfloor\dfrac{n-m+1}{m}\right\rfloor$．
- $\lfloor mx\rfloor=\lfloor x\rfloor+\left\lfloor x+\dfrac{1}{m}\right\rfloor+\cdots+\left\lfloor x+\dfrac{m-1}{m}\right\rfloor$．

- $\lceil mx \rceil = \lceil x \rceil + \left\lceil x - \frac{1}{m} \right\rceil + \cdots + \left\lceil x - \frac{m-1}{m} \right\rceil$.
- 当 $m \perp n$ 时，$\sum_{k=1}^{m-1} \left\lfloor \frac{kn}{m} \right\rfloor = \frac{1}{2}(n-1)(m-1)$.
- 当 $m \perp n$ 时，$\sum_{k=1}^{m-1} \left\lfloor \frac{kn}{m} \right\rfloor = \frac{1}{2}(n+1)(m-1)$.

这些乃至更一般的类似形式求和式的推导可以参考 类欧几里得算法→9.11.29@p1410 页面.

取整函数的更多性质以及应用可以参考如下页面：

- 取模运算：$n \bmod m = n - \left\lfloor \frac{n}{m} \right\rfloor m$. 它可以用于 优化整数取模运算→9.11.2@p1169.
- 利用 Gauss 引理证明 二次互反律→9.11.17@p1263.
- 数论分块→9.11.21@p1314，尤其是它的性质证明部分.
- 计算阶乘中素数因子幂次的 Legendre 公式→9.11.14@p1242.
- Beatty 数列→9.18.2@p1903、Rayleigh 定理以及 Wythoff 博弈.

参考资料与注释

- 潘承洞，潘承彪. 初等数论. 北京大学出版社.
- Floor and ceiling functions - Wikipedia[3]
- Graham, Ronald L., Donald E. Knuth, and Oren Patashnik. "Concrete mathematics: a foundation for computer science." (1989).

[1] std::gcd - cppreference.com

[2] Are all primes (past 2 and 3) of the forms 6n+1 and 6n-1?

[3] Floor and ceiling functions - Wikipedia

9.11.2 模算术简介

> **Authors:** 383494, buuzzing, c-forrest, cr4c1an, Emp7iness, Enter-tainer, Great-designer, HeRaNO, jifbt, Kaiser-Yang, Koishilll, ksyx, Marcythm, Qiu-Quanzhi, Saisyc, sshwy, StarryReverie, StudyingFather, Tiphereth-A, Xeonacid, xyf007

算法竞赛中，数论部分的一个重要组成部分就是 **模算术**（modular arithmetic），也就是在某一模数下进行各种整数运算. 除了基础的四则运算和求幂外，还可以方便地进行取对数、开各次方、求阶乘和组合数等运算.

模算术常见于各类问题中，而不仅仅局限于数论部分. 很多问题的实际答案可能非常大，超过了常见的整型变量的存储范围. 此时，为了避免引入大整数运算和输出长数字，题目常常要求对答案取模后输出. 这就要求熟练掌握各类模算术技巧.

C/C++ 的整数除法和取模运算

在 C/C++ 中，整数除法和取模运算，与数学上习惯的取模和除法不一致.

对于所有标准版本的 C/C++，规定在整数除法中：

1. 当除数为 0 时，行为未定义；
2. 否则 `(a / b) * b + a % b` 的运算结果与 `a` 相等.

也就是说，取模运算的符号取决于除法如何取整；而除法如何取整，这是实现定义的（由编译器决定）.

从 C99[1] 和 C++11[2] 标准版本起，规定 **商向零取整**（舍弃小数部分）；取模的符号就与被除数相同. 从此，以下断言保证为真：

```
assert(5 % 3 == 2);
assert(5 % -3 == 2);
assert(-5 % 3 == -2);
assert(-5 % -3 == -2);
```

模整数类

模算术可以看作是对某模数下的同余类→9.11.1@p1160 进行各种运算. 如果用一个结构体来表示一个同余类, 并且将同余类之间的加、减、乘等运算封装为结构体的方法或运算符重载, 那么模算术就可以自然地实现为一个模整数类. 下面给出一个简单的例子, 它支持模数 $M < 2^{30}$ 下 32 位带符号整数的加法、减法、乘法以及快速幂运算:

一个简单的模整数类

```cpp
// A simple ModInt implementation.
template <int M>
struct ModInt {
  struct skip_mod {};

  ModInt(int v, skip_mod) : v(v) {}

  int v;

  ModInt() : v(0) {}

  // Initialization: find remainder.
  // Equivalent to: v = int((x % M + M) % M)
  ModInt(long long x) {
    x %= M;
    if (x < 0) x += M;
    v = int(x);
  }

  // Addition.
  // Equivalent to: ModInt((l.v + r.v) % M)
  friend ModInt operator+(ModInt l, ModInt r) {
    int res = l.v + r.v;
    if (res >= M) res -= M;
    return ModInt(res, skip_mod{});
  }

  // Subtraction.
  // Equivalent to: ModInt((l.v - r.v + M) % M)
  friend ModInt operator-(ModInt l, ModInt r) {
    int res = l.v - r.v;
    if (res < 0) res += M;
    return ModInt(res, skip_mod{});
  }

  // Multiplication.
  friend ModInt operator*(ModInt l, ModInt r) {
    return ModInt(1LL * l.v * r.v % M, skip_mod{});
  }

  // Exponentiation.
  ModInt pow(long long b) const {
    ModInt res{1}, po{*this};
    for (; b; b >>= 1) {
      if (b & 1) res = res * po;
```

```
46        po = po * po;
47      }
48      return res;
49    }
50 };
51
```

这个实现有意地减少了取模运算的次数，因为取模操作通常比普通的加、减、乘或比较运算要耗时得多. 代码注释中提供了等价且更为直接的实现方式. 这些简单优化的主要思路是，两个 $[0, M)$ 内的整数做加减运算时，结果一定落在区间 $(-M, 2M)$ 内，因此可以通过一次加减法调整回区间 $[0, M)$. 该实现中的取幂运算用到了 快速幂→9.7@p1131 的技巧.

除了这些基础运算外，还可以在各种模数下做如下运算：

- 逆元→9.11.10@p1229
- 除法→9.11.11@p1233
- 阶乘→9.11.14@p1242
- 组合数→9.11.15@p1251
- 开平方→9.11.17@p1263
- 取对数→9.11.19@p1286
- 开方→9.11.20@p1291

这些运算通常在素数模数下比较容易. 对于合数模数，往往需要用到对应算法的扩展版本和 中国剩余定理→9.11.12@p1236. 这些模意义下的运算大多可以看作求解某种同余方程. 对于求解同余方程的一般方法，可以参考 同余方程→9.11.16@p1258 页面.

相关算法

本节将介绍几种在模意义下优化取模、乘法和快速幂运算的方法. 对于绝大多数题目来说，前文提供的简单实现已经足够高效. 然而，当题目对算法常数有严格要求时，这些优化方法就可以发挥作用，通过减少不必要的计算和取模操作，进一步降低算法的时间开销.

快速乘

在素性测试与质因数分解中，经常会遇到模数在 `long long` 范围内的乘法取模运算. 为了避免运算中的整型溢出问题，本节介绍一种可以处理模数在 `long long` 范围内，不需要使用 `__int128` 且复杂度为 $O(1)$ 的「快速乘」. 本算法要求测评系统中，`long double` 至少表示为 80 位扩展精度浮点数[3].

假设 $0 \le a, b < m$，要计算 $ab \bmod m$. 注意到：

$$ab \bmod m = ab - \left\lfloor \frac{ab}{m} \right\rfloor m.$$

利用 `unsigned long long` 的自然溢出：

$$ab \bmod m = ab - \left\lfloor \frac{ab}{m} \right\rfloor m = \left(ab - \left\lfloor \frac{ab}{m} \right\rfloor m \right) \bmod 2^{64}.$$

只要能算出商 $\left\lfloor \frac{ab}{m} \right\rfloor$，最右侧表达式中的乘法和减法运算都可以使用 `unsigned long long` 直接计算.

接下来，只需要考虑如何计算 $\left\lfloor \frac{ab}{m} \right\rfloor$. 解决方案是先使用 `long double` 算出 $\frac{a}{m}$ 再乘上 b. 既然使用了 `long double`，就无疑会有精度误差. 假设 `long double` 表示为 80 位扩展精度浮点数（即符号为 1 位，指数为 15 位，尾数为 64 位），那么 `long double` 最多能精确表示的有效位数为 64[4]. 所以 $\frac{a}{m}$ 最差从第 65

[3] 这适用于大多数 64 位系统上的 GCC 或 Clang 编译器.
[4] 参见 Extended precision - Wikipedia[3].

位开始出错，误差范围[5]为 $(-2^{-64}, 2^{-64})$．乘上 b 这个 64 位带符号整数，误差范围为 $(-0.5, 0.5)$．为了简化后续讨论，可以先加一个 0.5 再取整，最后的误差范围是 $\{0, 1\}$．

最后，代入上式计算时，需要乘以 $-m$，所以最后的误差范围是 $\{0, -m\}$．因为 m 在 `long long` 范围内，所以当结果 $r \in [0, m)$ 时，直接返回 r，否则返回 $r + m$．

代码实现如下：

参考实现

```
long long mul(long long a, long long b, long long m) {
  long long c = (unsigned long long)a * b -
                (unsigned long long)((long double)a / m * b + 0.5L) * m;
  return c < 0 ? c + m : c;
}

```

如今，绝大多数测评系统所配备的 C/C++ 编译器已支持 `__int128` 类型[6]，因此也可以直接将乘数类型提升至 `__int128` 后取模计算：

参考实现

```
long long mul(long long a, long long b, long long m) {
  return (__int128)a * b % m;
}

```

当然，`__int128` 的取模运算耗时并不少．如果需要进一步卡常，可以考虑接下来两节介绍的方法．

Barrett 约减

前文提到，除法和取模运算通常比其他四则运算更为耗时．为了减少取模运算的开销，有一些算法可以在不直接做取模的情况下得到相同的结果．本节要介绍的 Barrett 约减算法就是其中之一．

设 m 为固定模数，假设要对不同的 $a > 0$ 多次计算 $a \bmod m$．由带余除法可知

$$z = a \bmod m = a - \left\lfloor \frac{a}{m} \right\rfloor m.$$

关键在于商数 $\left\lfloor \frac{a}{m} \right\rfloor$ 的计算．设 R 是某个常数，就有[7]

$$\left\lfloor \frac{a}{m} \right\rfloor = \left\lfloor a \frac{R}{m} / R \right\rfloor \approx \left\lfloor a \left\lfloor \frac{R}{m} \right\rfloor / R \right\rfloor.$$

如果选取 $R = 2^k$，那么，右式中 $\left\lfloor \frac{R}{m} \right\rfloor$ 可以预处理，除以 R 的操作可以通过移位运算进行．所以，用右式计算商数，仅需要一次乘法和一次移位操作．再代入 $a \bmod m$ 的表达式，就得到所求余数的估计 z'．

现在分析这样做的误差．取整函数→9.11.1@p1160 具有性质：对于 $x > y > 0$ 都有 $\lfloor x \rfloor - \lfloor y \rfloor \le \lceil x - y \rceil$．所以，误差

[5]此处用到了条件 $a < m$，即 $a/m \in [0, 1)$．

[6]在目前的主流编译环境中，只有 Windows 平台上的 MSVC 不支持 `__int128` 类型．若需要编写可在多平台上兼容的代码，可以通过宏 `_MSC_VER` 检测 MSVC 编译环境，并在该条件下包含 `<intrin.h>`[4] 头文件，利用其提供的内建函数（如 `_umul128` 等）来间接实现 128 位整数运算（仅在 64 位平台上可用）．

[7]此处 $\left\lfloor \frac{R}{m} \right\rfloor$ 也可以替换成 $\frac{R}{m}$ 的其他整数估计，例如上取整函数 $\left\lceil \frac{R}{m} \right\rceil$ 和四舍五入取整函数 $\left\lfloor \frac{R}{m} \right\rceil$ 等，只要相应地调整对估计值的误差修正步骤．

$$\Delta = |\, z' - z \,| = m\left|\left\lfloor \frac{a}{m} \right\rfloor - \left\lfloor a\left\lfloor \frac{R}{m} \right\rfloor / R \right\rfloor\right|$$
$$\leq m\left\lceil a\left(\frac{R}{m} - \left\lfloor \frac{R}{m} \right\rfloor\right)/R \right\rceil \leq m\left\lceil \frac{a}{R} \right\rceil.$$

只要 $a \leq R$，误差 Δ 就不超过 m．由于 $z' \geq z$，所以估计值 z' 只能是 z 或 $z + m$．只要在得到估计值后，在 $z' \geq m$ 时再减去多余的 m，就可以保证答案正确．

在 Barrett 约减的计算过程中，仅使用了两次乘法、一次移位操作和至多两次减法，就完成了整数取模．但效率的提升并非毫无成本，实际上，Barrett 约减涉及的中间变量长度往往长于输入变量长度．容易发现，Barrett 约减中涉及的最长中间变量为 $a\left\lfloor \frac{R}{m} \right\rfloor$．设 $\ell(x)$ 为整数 x 的二进制表示长度．那么，有

$$\ell\left(a\left\lfloor \frac{R}{m} \right\rfloor\right) \approx \ell(a) + \ell(R) - \ell(m).$$

由于 R 的选取需要满足条件 $a \leq R$，这一长度至少为 $2\ell(a) - \ell(m)$．但是，需要取模时，一般都有 $\ell(m) \leq \ell(a)$，因此，这一中间变量的长度可能大于输入长度 $\ell(a)$．例如，如果需要将 64 位整数对 32 位整数取模，实际上中间变量需要 $64 \times 2 - 32 = 96$ 位整数．

Barrett 约减的一个应用场景就是计算乘积的余数 $ab \bmod m$．如果其中一个乘数固定，比如 b 固定时，可以通过

$$ab \bmod m = ab - \left\lfloor a\left\lfloor \frac{bR}{m} \right\rfloor / R \right\rfloor m$$

进行与上文类似的估计，只要预处理出 $\left\lfloor \frac{bR}{m} \right\rfloor$ 的值即可．这种 b 固定的情形有时也称为 Shoup 模乘[8]．

更为常见的情形是 a, b 都不固定．此时，需要首先计算 ab 的值，再利用 Barrett 约减得到 $ab \bmod m$．例如，实现模意义下乘法时，需要对 $0 \leq a, b < m$ 计算 $ab \bmod m$．此时，选取的 R 需要满足 $ab \leq R$．根据前文分析，计算过程涉及的最长中间变量长度为 $2\ell(ab) - \ell(m)$．当 $\ell(a) \approx \ell(b) \approx \ell(m)$ 时，该长度为 $3\ell(m)$．也就是说，如果要用 Barrett 约减实现 32 位整数的模乘，中间变量需要 96 位整数．这也是 Barrett 约减在算法竞赛中实际应用时的一个限制．

作为示例，利用 Barrett 约减实现 32 位有符号整数模乘的参考实现如下：

参考实现

```cpp
// Modular multiplication of int32_t using Barrett reduction.
class Barrett {
  int32_t m;
  uint64_t r;

 public:
  Barrett(int32_t m) : m(m), r((uint64_t)(-m) / m + 1) {}

  // Barrett reduction: a % m.
  int32_t reduce(int64_t a) const {
    int64_t q = (__int128)a * r >> 64;
    a -= q * m;
    return a >= m ? a - m : a;
  }

  // Modular multiplication: (a * b) % m;
  // Assume that 0 <= a, b < m.
```

[8] Shoup 在他的数论计算库 NTL[5] 中实现了 Barrett 约减的这一扩展，因此得名．

```
18   int32_t mul(int32_t a, int32_t b) const { return reduce((int64_t)a * b); }
19 };
20
```

实现中需要用到 128 位整数[9].

Montgomery 模乘

Montgomery 模乘算法的功能和 Barrett 算法十分类似，它同样可以减少模整数运算过程中取模运算的开销．与前两个算法都是在近似计算商数不同，Montgomery 模乘将所有整数都映射到 Montgomery 空间上，而 Montgomery 空间中的运算相对容易，进而降低了整体计算成本．

设模数 m 为奇数，并选取 $R = 2^k > m$．那么，同余类 $a \bmod m$ 对应的 Montgomery 形式就是

$$aR \bmod m.$$

因为 $R \perp m$，所以同余类 $a \bmod m$ 与它的 Montgomery 形式 $aR \bmod m$ 之间存在双射．因此，可以将整数转换为 Montgomery 形式后，进行若干模 m 的运算，再将得到的 Montgomery 形式转换回整数，结果总是正确的．

利用 Montgomery 形式可以方便地进行很多模整数的运算．刚刚已经说明，比较两个同余类是否相同，只要比较它们的 Montgomery 形式．又因为

$$(a \pm b)R \bmod m = ((aR \bmod m) \pm (bR \bmod m)) \bmod m,$$

所以同余类的加法、减法就对应它们的 Montgomery 形式的加法、减法．但是，要计算同余类的乘法，并不能直接将两个 Montgomery 形式相乘．因为

$$(ab)R \bmod m = ((aR \bmod m)(bR \bmod m)R^{-1}) \bmod m,$$

所以，计算两个 Montgomery 形式的乘法时，需要对它们的乘积 x 作如下 **Montgomery 约减**（Montgomery reduction）操作：

$$\mathrm{REDC} : x \mapsto xR^{-1} \bmod m.$$

利用这一操作，乘积 ab 的 Montgomery 形式就是 $\mathrm{REDC}((aR \bmod m)(bR \bmod m))$．Montgomery 约减操作是 Montgomery 模乘的核心操作：

- 将 a 转换为它的 Montgomery 形式就是 $\mathrm{REDC}((a \bmod m)(R^2 \bmod m))$．
- 将 a 的 Montgomery 形式转换回 $a \bmod m$ 就是 $\mathrm{REDC}(aR \bmod m)$．
- 模逆元 $a^{-1} \bmod m$ 对应的 Montgomery 形式就是 $\mathrm{REDC}((aR \bmod m)^{-1}(R^3 \bmod m))$．

现在讨论 Montgomery 约减操作 REDC 的实现方法．在计算 $\mathrm{REDC}(x)$ 时，总是假定 $0 \le x < m^2$，这对于以上情形都是成立的．因为 $R \perp m$，所以由 裴蜀定理→9.11.8@p1218，存在整数 R^{-1}, m' 使得

$$RR^{-1} + mm' = 1.$$

所以，设 $q = \lfloor xm'/R \rfloor$，就有

$$xR^{-1} = x\frac{1 - mm'}{R} \equiv \frac{x - xmm' + qmR}{R} = \frac{x - m(xm' \bmod R)}{R} \pmod m.$$

因为 $0 \le x < m^2 < mR$ 且 $0 \le xm' \bmod R < R$，所以

$$-m < \frac{x - m(xm' \bmod R)}{R} < m.$$

也就是说，这个商和 $xR^{-1} \bmod m$ 之间至多差一个 m．只要在商小于零时，再加上 m 就可以得到 $\mathrm{REDC}(x)$．计算这个商，只需要两次整数乘法、一次整数减法和两次位操作（分别是对 $R = 2^k$ 取模和做除法）．因此，Montgomery 约减操作可以高效进行．

[9] 在目前的主流编译环境中，只有 Windows 平台上的 MSVC 不支持 `__int128` 类型．若需要编写可在多平台上兼容的代码，可以通过宏 `_MSC_VER` 检测 MSVC 编译环境，并在该条件下包含 `<intrin.h>`[6] 头文件，利用其提供的内建函数（如 `_umul128` 等）来间接实现 128 位整数运算（仅在 64 位平台上可用）．

为了进行 Montgomery 模乘操作，需要预处理出一系列常数．首先，Montgomery 约减中会用到 $m' = m^{-1} \bmod R$，可以通过 下文→9.11@p1160 介绍的 Newton–Hensel 方法计算．其次，将不同操作归约为 Montgomery 约减操作时，还涉及诸如 $R^2 \bmod m$ 这样的常数．为了得到它，需要计算一次 $R \bmod m$，将它与自身相加就得到 $2R \bmod m$．随后，将它看作 2 的 Montgomery 形式，直接计算快速幂，就可以得到 $2^k R \bmod m = R^2 \bmod m$．

作为示例，32 位有符号整数的 Montgomery 模乘实现如下：

参考实现

```cpp
// Montgomery modular multiplication.
// The modulus m must be odd. The constant r is 2^32.
class Montgomery {
  int32_t m;
  uint32_t mm, r2;

 public:
  Montgomery(int32_t m) : m(m), mm(1), r2(-m) {
    // Compute mm as inv(m) mod r.
    for (int i = 0; i < 5; ++i) {
      mm *= 2 - mm * m;
    }
    // Compute r2 as r * r mod m.
    // If allowed to use modular operation for uint64_t, simply use:
    //   r2 = (uint64_t)(-m) % m;
    r2 %= m;
    r2 <<= 1;
    if (r2 >= (uint32_t)m) r2 -= m;
    for (int i = 0; i < 5; ++i) {
      r2 = mul(r2, r2);
    }
  }

  // Montgomery reduction: x * inv(r) % m.
  // Also used to transform x from Montgomery space to the normal space.
  int32_t reduce(int64_t x) {
    uint32_t u = (uint32_t)x * mm;
    int32_t ans = (x - (int64_t)m * u) >> 32;
    return ans < 0 ? ans + m : ans;
  }

  // Multiplication in Montgomery space: x * y * inv(r) % m.
  int32_t mul(int32_t x, int32_t y) { return reduce((int64_t)x * y); }

  // Transform x from the normal space to Montgomery space.
  int32_t init(int32_t x) { return mul(x, r2); }
};

```

相对于 Barrett 约减实现模意义下乘法，Montgomery 模乘的计算涉及转换、Montgomery 形式的乘法、逆转换等多个步骤．因此，只有在转换和逆转换之间的模运算次数足够多时，转换和逆转换的成本才可以摊平，进而获得较高的整体效率．但是，由于 Montgomery 模乘的实现过程中只涉及长度为 $2\ell(m)$ 的中间

变量，所以实现起来更为灵活．例如，32 位整数的模乘仅需要 64 位整数的中间变量．所以，如果需要实现一个模整数类用于各种数论计算，Montgomery 模乘更为合适．

模 2 的幂次的整数类

本节讨论模数是 2 的幂次时，模整数类的实现．在这一特殊情形中，除法和取模运算可以通过位操作实现，计算效率很高．Barrett 约减和 Montgomery 模乘都是利用了 2^e 作为除数和模数时的这一特性来加速运算．特别地，当模数恰为 2^{32} 和 2^{64} 等特殊数字时，可以利用相应位长的无符号整数结合自然溢出实现模整数类，无需任何显式的取模运算．即使模数并非恰好如此，也可以转化为这些特殊模数的情形．例如模数为 2^{58} 时，可以在模数 2^{64} 下完成中间计算，最后再将结果对 2^{58} 取模．除了取模方便外，模 2^e 整数类的其他操作也有很多特殊实现．本节重点介绍逆元和取幂操作的实现方式．

首先是取逆操作：给定奇数 a 和模数 $m = 2^e$ $(e > 2)$，需要求出 $a^{-1} \bmod m$．求逆元的常见方法包括扩展欧几里得算法和快速幂法．扩展欧几里得算法的过程涉及对一般模数取模；普通的快速幂法需要计算 $a^{\varphi(m)-1} \bmod m$，这需要 $\Theta(e)$ 次整数乘法．更为高效的方法是 Newton–Hensel 方法→9.12.7@p1554．具体地，考虑应用如下结论：[10]

$$ax \equiv 1 \pmod{2^e} \Longrightarrow ax(2-ax) \equiv 1 \pmod{2^{2e}}.$$

根据这一表达式，只要从 $x = 1$ 开始，反复应用 $x \leftarrow x(2-ax)$，就可以在 $\lceil \log_2 e \rceil$ 次迭代后得到 $a^{-1} \bmod m$．

作为示例，模 2^{32} 整数取逆操作参考实现如下：

参考实现

```
// Compute 1/v mod 2^32 for odd v.
uint32_t inv_mod_2_32(uint32_t v) {
  uint32_t x = 1;
  for (int i = 0; i != 5; ++i) {
    x *= 2 - v * x;
  }
  return x;
}

```

接下来，讨论取幂操作：给定 x, a, b 和模数 $m = 2^e$ $(e > 2)$，需要求出 $xa^b \bmod m$，其中，a 是奇数．根据对模 2^e 整数乘法结构的 分析→9.11.18@p1275 可知，a 总是可以写成 $\pm g^\ell$ 的形式[11]，且负号出现且仅出现在 $a \equiv 3 \pmod 4$ 的情形．对于这种情况，可以将 a 替换成 $-a$，并将最终结果再乘上 $(-1)^b$．因此，接下来不妨假设 $a \equiv 1 \pmod 4$ 成立．此时，算法的核心想法是，将 a 写成 $g^{L(a)} \bmod m$ 的形式，然后用 $xg^{bL(a)} \bmod m$ 计算所求的幂．

计算 $L(a)$ 就是计算离散对数 $\operatorname{ind}_g a$．注意到，只要 $a \equiv 1 \pmod 4$，那么 a 总能写成如下形式：

$$a \equiv (2^{e_1}+1)(2^{e_2}+1)\cdots(2^{e_s}+1) \pmod m,$$

其中，$1 < e_1 < e_2 < \cdots < e_s < e$．这是因为直接将这一乘积展开可以发现，$a$ 的二进制表示中等于 1 的次低位就是第 e_1 位（下标从 0 开始），由此就可以递归地找到这一表示．根据离散对数的 性质→9.11.19@p1286 可知，有

$$4L(a) \equiv 4L(2^{e_1}+1) + 4L(2^{e_2}+1) + \cdots + 4L(2^{e_s}+1) \pmod m.$$

[10] 直接验证：由 $ax \equiv 1 \pmod{2^e}$，可以设 $ax = 1 + \lambda 2^e$，那么就有 $ax(2-ax) = (1+\lambda 2^e)(1-\lambda 2^e) = 1 - \lambda^2 2^{2e} \equiv 1 \pmod{2^{2e}}$．

[11] 文中所引页面仅证明了 g 可以取 5．实际上，完全重复该证明，可以说明 g 可以取任何模 8 余 5 的整数．后文会讨论 g 的选取方法．

由于离散对数的模数等于阶 $\delta_m(g) = 2^{e-2} = m/4$，所以此处直接将整个同余式都乘以 4，以保证计算可以在模 m 剩余类中进行. 由此，只需要对 $1 < d < e$ 预处理出所有的 $4L(2^d + 1)$，就可以快速计算 $4L(a)$ 的值.

反过来，从 $L(a)$ 也很容易得到 $g^{L(a)} \bmod m$ 的值. 根据 二项式定理→9.13.1@p1612 可知，对于 $1 < d < e$，都有

$$(2^d + 1)^{2^{e-d}} \equiv 1 \pmod m, \quad (2^d + 1)^{2^{e-d-1}} \equiv 1 + 2^{e-1} \pmod m,$$

所以，$\delta_m(2^d + 1) = 2^{e-d}$. 根据阶的性质可知

$$\delta_m(2^d + 1) = \frac{\delta_m(g)}{\gcd(\delta_m(g), \operatorname{ind}_g(2^d + 1))}.$$

所以，$\gcd(\delta_m(g), \operatorname{ind}_g(2^d + 1)) = 2^{d-2}$. 这说明 $L(2^d + 1) = \operatorname{ind}_g(2^d + 1) = 2^{d-2}r$，其中，$2 \nmid r$. 所以，$4L(2^d + 1)$ 的二进制表示中等于 1 的最低位恰为第 d 位（下标从 0 开始）. 因此，同样可以通过二进制表示递归地将 $4L(a)$ 分解为形如 $4L(2^d + 1)$ 的和. 由此，就可以得到 a 的值.

具体实现时，有一些可以进一步优化的点. 首先，将 a 分解为乘积形式时，还是需要用到除法. 更方便的是计算 a^{-1} 的分解，即寻找 $1 < e_1 < e_2 < \cdots < e_s < e$ 使得

$$a(2^{e_1} + 1)(2^{e_2} + 1)\cdots(2^{e_s} + 1) \equiv 1 \pmod m$$

成立. 同样是通过寻找等于 1 的次低位来确定 e_1，但是要在 a^{-1} 中消去 $2^{e_1} + 1$ 因子，只需要在 a 上乘以 $2^{e_1} + 1$ 即可，这可以通过位操作进行. 又因为 $4L(a^{-1}) = -4L(a)$，所以统计 $4L(a)$ 时，需要用减法代替加法. 其次，对于特殊选择的基底 g，迭代无需进行到 $d = e - 1$，而只要进行到 $d = \lceil e/2 \rceil - 1$ 即可. 为此，需要选择 g 使得

$$4L(2^{\lceil e/2 \rceil} + 1) = 2^{\lceil e/2 \rceil}.$$

对于 $d \geq e/2$，都有

$$(2^d + 1)^2 = 2^{2d} + 2^{d+1} + 1 \equiv 2^{d+1} + 1 \pmod m.$$

所以，从 $d = \lceil e/2 \rceil$ 开始归纳可知，$4L(2^d + 1) = 2^d$ 对于所有 $d \geq e/2$ 都成立. 进而，只要 $e/2 \leq e_1 < e_2 < \cdots < e_s < e$，就有

$$(2^{e_1} + 1)(2^{e_2} + 1)\cdots(2^{e_s} + 1) \equiv 1 + 2^{e_1} + 2^{e_2} + \cdots + 2^{e_s} \pmod m$$

以及

$$4L((2^{e_1} + 1)(2^{e_2} + 1)\cdots(2^{e_s} + 1)) = 2^{e_1} + 2^{e_2} + \cdots + 2^{e_s}.$$

因此，处理完所有 $d < e/2$ 的二进制位后，可以直接得到剩余部分的离散对数，而无需逐位计算. 应用第一个优化后，整个取幂操作只需要 $O(e)$ 次加减法和位操作以及 1 次乘法操作；应用第二个优化后，可以省去约一半的加减法和位操作，但需要额外 1 次乘法操作.

作为示例，模 2^{32} 整数取幂操作参考实现如下：

参考实现

```cpp
// Store 4L(a) for a = 2^d + 1, where L(a) is disc. log. base 388251981.
// The first two values are never used and thus set to zero.
// The base is chosen such that 4L(2^16+1) = 2^16.
constexpr uint32_t log_table[16] = {
    0x00000000, 0x00000000, 0xbba0267c, 0x49b9d1e8, 0xf0026f90, 0xd6e17e20,
    0xe78bf840, 0x039fe080, 0xaf7f8100, 0x60fe0200, 0xd1f80400, 0x23e00800,
    0x47801000, 0x8e002000, 0x18004000, 0x20008000,
};

// Compute 4L(v).
uint32_t log_mod_2_32(uint32_t x, uint32_t v) {
```

```
  for (int i = 2; i != 16; ++i) {
    if ((v >> i) & 1) {
      v += v << i;
      x -= log_table[i];
    }
  }
  x += v ^ 1;
  return x;
}

// Compute x*a for 4L(a) = v.
uint32_t exp_mod_2_32(uint32_t x, uint32_t v) {
  for (int i = 2; i != 16; ++i) {
    if ((v >> i) & 1) {
      x += x << i;
      v -= log_table[i];
    }
  }
  x *= v ^ 1;
  return x;
}

// Compute x*a^b for odd a.
uint32_t pow_odd_mod_2_32(uint32_t a, uint32_t b, uint32_t x) {
  if (a & 2) {
    a = -a;
    if (b & 1) {
      x = -x;
    }
  }
  return exp_mod_2_32(x, log_mod_2_32(0, a) * b);
}

// Compute x*a^b mod 2^32.
uint32_t pow_mod_2_32(uint32_t a, uint32_t b, uint32_t x = 1) {
  if (!a) return b == 0 ? x : 0;
  auto d = __builtin_ctz(a);
  if ((uint64_t)d * b >= 32) return 0;
  return pow_odd_mod_2_32(a >> d, b, x) << (d * b);
}

```

离散对数的预处理可以通过 Pohlig–Hellman 算法进行，基底 g 可以选择为

$$5^{\operatorname{ind}_5(2^{\lceil e/2\rceil}+1)/2^{\lceil e/2\rceil-2}} \bmod 2^e.$$

参考资料与注释

- Fast modular multiplication by orz - Codeforces[7]
- Barrett Reduction - Wikipedia[8]
- Barrett Reduction - A41[9]
- Barrett 约减原理及正确性证明 by Chen - 知乎专栏[10]
- Montgomery Multiplication - CP Algorithms[11]
- Montgomery 模乘 by Chen - 知乎专栏[12]

- Binary Exponentiation by Factoring - CP Algorithms[13]
- Barrett, Paul. "Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor." In Conference on the Theory and Application of Cryptographic Techniques, pp. 311-323. Berlin, Heidelberg: Springer Berlin Heidelberg, 1986.
- Becker, Hanno, Vincent Hwang, Matthias J. Kannwischer, Bo-Yin Yang, and Shang-Yi Yang. "Neon NTT: Faster Dilithium, Kyber, and Saber on Cortex-A72 and Apple M1." IACR Transactions on Cryptographic Hardware and Embedded Systems (2022): 221-244.
- Montgomery, Peter L. "Modular multiplication without trial division." Mathematics of computation 44, no. 170 (1985): 519-521.

[1] C99 [2] C++11 [3] Extended precision - Wikipedia

[4] `<intrin.h>` [5] NTL [6] `<intrin.h>`

[7] Fast modular multiplication by orz - Codeforces [8] Barrett Reduction - Wikipedia [9] Barrett Reduction - A41

[10] Barrett 约减原理及正确性证明 by Chen - 知乎专栏 [11] Montgomery Multiplication - CP Algorithms [12] Montgomery 模乘 by Chen - 知乎专栏

[13] Binary Exponentiation by Factoring - CP Algorithms

9.11.3 素数

> **Authors:** Ir1d, Tiphereth-A, c-forrest, Xeonacid, Enter-tainer, StudyingFather, iamtwz, ksyx, Marcythm, MegaOwIer, 383494, Alpacabla, HeRaNO, abc1763613206, alphagocc, Backl1ght, CCXXXI, drkelo, Early0v0, Great-designer, greyqz, GuanghaoYe, H-J-Granger, HHH2309, isdanni, kenlig, lazyasn, Menci, ouuan, r-value, shawlleyw, shopee-jin, shuzhouliu, Siger Young, TrisolarisHD, untitledunrevised, void-mian, Voileexperiments, weilycoder, xtlsoft, yusancky, YuzhenQin1, sun2snow

素数与合数的定义，见 数论基础→9.11.1@p1160.

素数计数函数：小于或等于 x 的素数的个数，用 $\pi(x)$ 表示. 随着 x 的增大，有这样的近似结果：$\pi(x) \sim \dfrac{x}{\ln(x)}$.

素性测试

素性测试（Primality test）可以用于判定所给自然数是否为素数.

素性测试有两种：

1. 确定性测试：绝对确定一个数是否为素数. 常见例子包括试除法、Lucas–Lehmer 测试和椭圆曲线素性证明.
2. 概率性测试：通常比确定性测试快很多，但有可能（尽管概率很小）错误地将 合数→9.11.1@p1160 识别为素数（尽管反之则不会）. 因此，通过概率素性测试的数字被称为 **可能素数**，直到它们的素性可以被确定性地证明. 而通过测试但实际上是合数的数字则被称为 **伪素数**. 有许多特定类型的伪素数，最常见的是费马伪素数，它们是满足费马小定理的合数. 概率性测试的常见例子包括 Miller–Rabin 测试.

试除法

暴力做法自然可以枚举从小到大的每个数看是否能整除.

参考实现

- C++

```cpp
bool isPrime(int a) {
  if (a < 2) return false;
  for (int i = 2; i < a; ++i)
    if (a % i == 0) return false;
  return true;
}
```

- Python

```python
def isPrime(a):
    if a < 2:
        return False
    for i in range(2, a):
        if a % i == 0:
            return False
    return True
```

这样做是十分稳妥了，但是真的有必要每个数都去判断吗?

很容易发现这样一个事实：如果 x 是 a 的约数，那么 $\frac{a}{x}$ 也是 a 的约数.

这个结论告诉我们，对于每一对 $(x, \frac{a}{x})$，只检验其中的一个就足够了．为了方便起见，我们只考察每一对的较小数．不难发现，所有这些较小数都在 $[1, \sqrt{a}]$ 这个区间里.

由于 1 肯定是约数，所以不检验它.

参考实现

- C++

```cpp
bool isPrime(int a) {
  if (a < 2) return 0;
  for (int i = 2; (long long)i * i <= a; ++i)  // 防溢出
    if (a % i == 0) return 0;
  return 1;
}
```

- Python

```python
def isPrime(a):
    if a < 2:
        return False
    for i in range(2, int(sqrt(a)) + 1):
        if a % i == 0:
            return False
    return True
```

Fermat 素性测试

Fermat 素性检验 是最简单的概率性素性检验.

我们可以根据 费马小定理→9.11.9@p1223 得出一种检验素数的思路:

基本思想是不断地选取在 $[2, n-1]$ 中的基底 a，并检验是否每次都有 $a^{n-1} \equiv 1 \pmod{n}$.

参考实现

- C++

```cpp
bool fermat(int n) {
  if (n < 3) return n == 2;
  // test_time 为测试次数,建议设为不小于 8
  // 的整数以保证正确率,但也不宜过大,否则会影响效率
  for (int i = 1; i <= test_time; ++i) {
    int a = rand() % (n - 2) + 2;
    if (quickPow(a, n - 1, n) != 1) return false;
  }
  return true;
}
```

- Python

```python
def fermat(n):
    if n < 3:
        return n == 2
    # test_time 为测试次数,建议设为不小于 8
    # 的整数以保证正确率,但也不宜过大,否则会影响效率
    for i in range(1, test_time + 1):
        a = random.randint(0, 32767) % (n - 2) + 2
        if quickPow(a, n - 1, n) != 1:
            return False
    return True
```

如果 $a^{n-1} \equiv 1 \pmod n$ 但 n 不是素数，则称 n 为以 a 为底的 **Fermat 伪素数**. 我们在实践中观察到，如果 $a^{n-1} \equiv 1 \pmod n$，那么 n 通常是素数. 但其实存在反例：对于 $n = 341$ 且 $a = 2$，虽然有 $2^{340} \equiv 1 \pmod{341}$，但是 $341 = 11 \cdot 31$ 是合数. 事实上，对于任何固定的基底 a，这样的反例都有无穷多个[12].

既然对于单个基底，Fermat 素性测试无法保证正确性，一个自然的想法就是多检查几组基底. 但是，即使检查了所有可能的与 n 互素的基底 a，依然无法保证 n 是素数. 也就是说，费马小定理的逆命题并不成立：即使对于所有 $a \perp n$，都有 $a^{n-1} \equiv 1 \pmod n$，n 也不一定是素数. 这样的数称为 Carmichael 数 →9.11.18@p1275. 它也有无穷多个. 这迫使我们寻找更为严格的素性测试.

Miller–Rabin 素性测试

Miller–Rabin 素性测试（Miller–Rabin primality test）是更好的素数判定方法. 它是由 Miller 和 Rabin 二人根据 Fermat 素性测试优化得到的. 和其它概率性素数测试一样，它也只能检测出可能素数. 要确保是素数，需要用慢得多的确定性算法. 然而，目前还没有发现能够通过 Miller–Rabin 测试等高级概率性测试的合数，因此我们可以放心使用.

在不考虑乘法的复杂度时，对数 n 进行 k 轮测试的时间复杂度是 $O(k \log n)$. Miller–Rabin 素性测试常用于对高精度数进行测试，此时时间复杂度是 $O(k \log^3 n)$，利用 FFT 等技术可以优化到 $O(k \log^2 n \log\log n \log\log\log n)$[1].

为了解决 Carmichael 数带来的挑战，Miller–Rabin 素性测试进一步考虑了素数的如下性质：

[12] Pomerance, Carl, John L. Selfridge, and Samuel S. Wagstaff. "The pseudoprimes to 25· 10⁹." Mathematics of Computation 35, no. 151 (1980): 1003-1026. 的定理 1 说明了，对于固定的基底 a，能够通过更强的 Miller–Rabin 素性测试的合数也是无穷多的.

二次探测定理

如果 p 是奇素数，则 $x^2 \equiv 1 \pmod p$ 的解为 $x \equiv 1 \pmod p$ 或者 $x \equiv p-1 \pmod p$.

证明

容易验证，p 为奇素数时，$x \equiv 1 \pmod p$ 和 $x \equiv p-1 \pmod p$ 都可以使得上式成立．由 Lagrange 定理→9.11.16@p1258 可知，这就是该方程的所有解.

将费马小定理和二次探测定理结合起来使用，就得到 Miller–Rabin 素性测试：

1. 将 $a^{n-1} \equiv 1 \pmod n$ 中的指数 $n-1$ 分解为 $n-1 = u \times 2^t$，其中 u 为奇数；
2. 在每轮测试中对随机出来的 a 先求出 $v = a^u \bmod n$，之后对这个值执行最多 t 次平方操作；
3. 在整个过程中，如果发现 1 的非平凡平方根（即除了 ± 1 之外的其他根），就可以判断该数不是素数；
4. 否则，再使用 Fermat 素性测试判断.

还有一些实现上的小细节：

- 对于一轮测试，如果某一时刻 $a^{u \times 2^s} \equiv n-1 \pmod n$，则之后的平方操作全都会得到 1，则可以直接通过本轮测试.
- 如果找出了一个非平凡平方根，即 $a^{u \times 2^s} \not\equiv n-1 \text{ or } 1 \pmod n$ 而 $a^{u \times 2^{s+1}} \equiv 1 \pmod n$，则之后的平方操作全都会得到 1．可以选择直接返回 `false`，也可以放到 t 次平方操作后再返回 `false`.

这样得到了较正确的 Miller–Rabin：（来自 fjzzq2002）

参考实现

- C++

```cpp
bool millerRabin(int n) {
  if (n < 3 || n % 2 == 0) return n == 2;
  if (n % 3 == 0) return n == 3;
  int u = n - 1, t = 0;
  while (u % 2 == 0) u /= 2, ++t;
  // test_time 为测试次数，建议设为不小于 8
  // 的整数以保证正确率，但也不宜过大，否则会影响效率
  for (int i = 0; i < test_time; ++i) {
    // 0, 1, n-1 可以直接通过测试，a 取值范围 [2, n-2]
    int a = rand() % (n - 3) + 2, v = quickPow(a, u, n);
    if (v == 1) continue;
    int s;
    for (s = 0; s < t; ++s) {
      if (v == n - 1) break;  // 得到平凡平方根 n-1，通过此轮测试
      v = (long long)v * v % n;
    }
    // 如果找到了非平凡平方根，则会由于无法提前 break；而运行到 s == t
    // 如果 Fermat 素性测试无法通过，则一直运行到 s == t 前 v 都不会等于 -1
    if (s == t) return 0;
  }
  return 1;
}
```

- Python

```python
def millerRabin(n):
    if n < 3 or n % 2 == 0:
```

```
        return n == 2
    if n % 3 == 0:
        return n == 3
    u, t = n - 1, 0
    while u % 2 == 0:
        u = u // 2
        t = t + 1
    # test_time 为测试次数,建议设为不小于 8
    # 的整数以保证正确率,但也不宜过大,否则会影响效率
    for i in range(test_time):
        # 0, 1, n-1 可以直接通过测试, a 取值范围 [2, n-2]
        a = random.randint(2, n - 2)
        v = pow(a, u, n)
        if v == 1:
            continue
        s = 0
        while s < t:
            if v == n - 1:
                break
            v = v * v % n
            s = s + 1
        # 如果找到了非平凡平方根，则会由于无法提前 break; 而运行到 s == t
        # 如果 Fermat 素性测试无法通过，则一直运行到 s == t 前 v 都不会等于 -1
        if s == t:
            return False
    return True
```

可以证明[13]，奇合数 $n > 9$ 通过随机选取的一个基底 a 的 Miller–Rabin 素性测试的概率至多为四分之一．因此，随机选取 k 个基底后，仍将合数误判为素数的概率不超过 $1/4^k$.

证明

设 $n-1=u2^t$，其中，u 是奇数且 t 是正整数．那么，整数 n 可以通过基底为 a 的 Miller–Rabin 素性测试说明

$$a^u \equiv 1 \pmod{n}, \text{or } a^{u2^i} \equiv -1 \pmod{n} \text{ for some } 0 \le i < t.$$

记这样的 a（的同余类）集合为 S，要说明的是

$$|S| \le \frac{1}{4}\varphi(n).$$

其中，$\varphi(n)$ 是 欧拉函数→9.11.5@p1196．证明分为三步．

第一步：设 ℓ 是使得 $2^\ell \mid p-1$ 对所有 n 的素因子 p 都成立的最大正整数．那么，可以证明

$$S \subseteq S' = \{a \bmod n : a^{u2^{\ell-1}} \equiv \pm 1 \pmod{n}\}.$$

集合 S 中的元素 a 只有两种可能．如果 $a^u \equiv 1 \pmod{n}$，那么，显然 $a^{u2^{\ell-1}} \equiv 1 \pmod{n}$ 也成立，亦即 $a \in S'$．如果对于 $0 \le i < t$ 成立 $a^{u2^i} \equiv -1 \pmod{n}$，那么，对于任意素因子 $p \mid n$，都有 $a^{u2^i} \equiv -1 \pmod{p}$．设 $\delta_p(a)$ 是 a 模 p 的 阶→9.11.18@p1275，那么，显然有 $\delta_p(a) \mid u2^{i+1}$ 但是 $\delta_p(a) \nmid u2^i$，这说明，$\delta_p(a)$ 的素因数分解中，2 的指数恰为 $i+1$，因而 $2^{i+1} \mid \delta_p(a)$．由费马小定理可知，$\delta_p(a) \mid$

[13]本结论及其证明参考了 Crandall, Richard, and Carl Pomerance. Prime numbers: a computational perspective. New York, NY: Springer New York, 2005. 的第 3.5 节.

$p-1$，所以，$2^{i+1} \mid p-1$．这一点对于 n 的所有素因子 p 都成立．因此，$i+1 \le \ell$．这说明 $a^{u2^{\ell-1}} = (a^{u2^i})^{2^{\ell-1-i}} \equiv \pm 1 \pmod{n}$，同样有 $a \in S'$．综合两种可能，就得到 $S \subseteq S'$．

第二步：计算 $|S'|$ 的大小．

假设 n 有素因数分解 $n = p_1^{e_1} p_2^{e_2} \cdots p_k^{e_k}$，那么，由 中国剩余定理→9.11.12@p1236 可知，条件 $a^{u2^{\ell-1}} \equiv 1 \pmod{n}$ 等价于 $a^{u2^{\ell-1}} \equiv 1 \pmod{p_i^{e_i}}$ 对所有 $p_i^{e_i}$ 都成立．由于模奇素数幂 $p_i^{e_i}$ 的 原根→9.11.18@p1275 总是存在的，所以，同余方程 $a^{u2^{\ell-1}} \equiv 1 \pmod{p_i^{e_i}}$ 的 解的数量→9.11.20@p1291 为

$$\gcd(u2^{\ell-1}, p_i^{e_i-1}(p_i-1)) = \gcd(u2^{\ell-1}, p_i - 1) = 2^{\ell-1}\gcd(u, p_i - 1).$$

第一个等号成立，是因为 u 是 $n-1$ 的因子，不可能是 p_i 的倍数；第二个等号成立，是因为 ℓ 的选取方式．所以，由中国剩余定理可知，同余方程 $a^{u2^{\ell-1}} \equiv 1 \pmod{n}$ 的解的数量为

$$\prod_{p \mid n} 2^{\ell-1}\gcd(u, p-1).$$

同理，条件 $a^{u2^{\ell-1}} \equiv -1 \pmod{n}$ 等价于 $a^{u2^{\ell-1}} \equiv -1 \pmod{p_i^{e_i}}$ 对所有 $p_i^{e_i}$ 都成立．对于任意因子 $p_i^{e_i}$，条件 $a^{u2^{\ell-1}} \equiv -1 \pmod{p_i^{e_i}}$ 都等价于 $a^{u2^{\ell-1}} \not\equiv 1 \pmod{p_i^{e_i}}$ 且 $a^{u2^{\ell}} \equiv 1 \pmod{p_i^{e_i}}$ 成立．类似上文，可以计算出同余方程 $a^{u2^{\ell}} \equiv 1 \pmod{p_i^{e_i}}$ 的解的数量为 $2^{\ell}\gcd(u, p_i-1)$，因此，同余方程 $a^{u2^{\ell-1}} \equiv -1 \pmod{p_i^{e_i}}$ 的解的数量也等于

$$2^{\ell}\gcd(u, p_i-1) - 2^{\ell-1}\gcd(u, p_i-1) = 2^{\ell-1}\gcd(u, p_i-1).$$

再次应用中国剩余定理，就得到同余方程 $a^{u2^{\ell-1}} \equiv -1 \pmod{n}$ 的解的数量等于

$$\prod_{p \mid n} 2^{\ell-1}\gcd(u, p-1).$$

因此，综合两种情形，有

$$|S'| = 2\prod_{p \mid n} 2^{\ell-1}\gcd(u, p-1).$$

第三步：证明 $|S'| \le \varphi(n)/4$．

结合欧拉函数的表达式 $\varphi(n) = \prod_i p_i^{e_i-1}(p_i-1)$ 可知

$$\frac{\varphi(n)}{|S'|} = \frac{1}{2}\prod_i p_i^{e_i-1}\frac{p_i-1}{2^{\ell-1}\gcd(u, p_i-1)}.$$

对于每一个 i，相应的因子 $p_i^{e_i-1}\dfrac{p_i-1}{2^{\ell-1}\gcd(u, p_i-1)}$ 都是一个偶数，所以，$\varphi(n)/|S'|$ 是一个整数．假设 $|S'| \le \varphi(n)/4$ 不成立．必然有 $\varphi(n)/|S'| = 1, 2, 3$，亦即

$$\prod_i p_i^{e_i-1}\frac{p_i-1}{2^{\ell-1}\gcd(u, p_i-1)} = 2, 4, 6.$$

由于连乘式中的每个因子都是偶数，所以，这个连乘式要么只有一个因子且这个因子就等于 $2, 4, 6$，要么就只有两个因子且都等于 2．

首先考虑有两个因子的情形．此时，两个因子都没有奇素因子，所以，$p_i^{e_i-1} = 1$，亦即 n 没有平方因子．不妨设 $n = p_1 p_2$ 且 $p_1 < p_2$ 都是素数．两个因子都等于 2，所以，总有 $p_i - 1 = 2^{\ell}\gcd(u, p_i - 1)$．因此，$p_i = 1 + 2^{\ell} m_i$，其中，$m_i$ 是奇数，而且 $m_i \mid u$．将 $p_1 p_2 = n = 1 + u2^t$ 对 m_1 取模就得到 $p_1 p_2 \equiv 1 \pmod{m_1}$，故而 $p_2 \equiv 1 \pmod{m_1}$，这说明，$m_1 \mid m_2$．反过来也成立．这就说明 $m_1 = m_2$，也就是 $p_1 = p_2$．这与 $p_1 < p_2$ 矛盾．这一情形不成立．

最后，考虑只有一个因子的情形，亦即合数 $n = p^e$ 且 $e > 1$．此时，必然有 $p^{e-1} \mid 2, 4, 6$．因此，唯一的情形是 $p = 3, e = 2$，亦即 $n = 9$，与命题所设相矛盾．这一情形也不成立．

综合所有情形可知，$|S'| \le \varphi(n)/4$ 成立．

结合上述三个步骤可知，$|S| \le |S'| \le \varphi(n)/4$ 对于所有奇合数 $n > 9$ 都成立．

另外，假设 广义 Riemann 猜想[2]（generalized Riemann hypothesis, GRH）成立，则对数 n 最多只需要测试 $[2, \min\{n-2, \lfloor 2\ln^2 n \rfloor\}]$ 中的全部整数即可 **确定** 数 n 的素性. [14]

而在 OI 范围内，通常都是对 $[1, 2^{64})$ 范围内的数进行素性检验. 对于 $[1, 2^{32})$ 范围内的数，选取 $\{2, 7, 61\}$ 三个数作为基底进行 Miller–Rabin 素性检验就可以确定素性；对于 $[1, 2^{64})$ 范围内的数，选取 $\{2, 325, 9375, 28178, 450775, 9780504, 1795265022\}$ 七个数作为基底进行 Miller–Rabin 素性检验就可以确定素性. [15]

也可以选取 $\{2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37\}$（即前 12 个素数）检验 $[1, 2^{64})$ 范围内的素数.

注意如果要使用上面的数列中的数 a 作为基底判断 n 的素性：

- 所有的数都要取一遍，不能只选小于 n 的；
- 把 a 换成 $a \bmod n$；
- 如果 $a \equiv 0 \pmod n$ 或 $a \equiv \pm 1 \pmod n$，则直接通过该轮测试.

反素数

顾名思义，素数就是因子只有两个的数，那么反素数，就是因子最多的数（并且因子个数相同的时候值最小），所以反素数是相对于一个集合来说的.

一种符合直觉的反素数定义是：在一个正整数集合中，因子最多并且值最小的数，就是反素数.

自 反素数

对于某个正整数 n，如果任何小于 n 的正整数的约数个数都小于 n 的约数个数，则称 n 为 **反素数**（anti-prime, a.k.a. highly composite number）.

⚠ 注意

注意区分 emirp[5]，它表示的是逐位反转后是不同素数的素数（如 149 和 941 均为 emirp，101 不是 emirp）.

过程

那么，如何来求解反素数呢?

首先，既然要求因子数，首先要做的就是素因子分解. 把 n 分解成 $n = p_1^{k_1} p_2^{k_2} \cdots p_m^{k_m}$ 的形式，其中 p 是素数，k 为它的指数. 这样的话总因子个数就是 $(k_1+1) \times (k_2+1) \times \cdots \times (k_m+1)$.

但是显然素因子分解的复杂度是很高的，并且前一个数的结果不能被后面利用. 所以要换个方法.

我们来观察一下反素数的特点.

1. 反素数肯定是从 2 开始的连续素数的幂次形式的乘积.
2. 数值小的素数的幂次大于等于数值大的素数的幂次，即 $n = p_1^{k_1} p_2^{k_2} \cdots p_m^{k_m}$ 中，有 $k_1 \geq k_2 \geq k_3 \geq \cdots \geq k_m$.

解释：

1. 如果不是从 2 开始的连续素数，那么如果幂次不变，把素数变成数值更小的素数，那么此时因子个数不变，但是 n 的数值变小了. 交换到从 2 开始的连续素数的时候 n 值最小.
2. 如果数值小的素数的幂次小于数值大的素数的幂，那么如果把这两个素数交换位置（幂次不变），那么所得的 n 因子数量不变，但是 n 的值变小.

另外还有两个问题，

1. 对于给定的 n，要枚举到哪一个素数呢?

[14] Bach, Eric , "Explicit bounds for primality testing and related problems[3]", Mathematics of Computation, 55:191 (1990) pp 355–380.

[15] 更多类似的结果请参考 Deterministic variant of the Miller–Rabin primality test[4].

最极端的情况大不了就是 $n = p_1 p_2 \cdots p_m$，所以只要连续素数连乘到刚好小于等于 n 即可．如果枚举到更大的素数，则意味着必定某个之前素数的幂次为 0，那么就不可能成为反素数．

2. 我们要枚举到多少次幂呢?

   我们考虑一个极端情况, 当我们最小的素数的某个幂次已经比所给的 n（的最大值）大的话, 那么展开成其他的形式, 最大幂次一定小于这个幂次. 极端情况下 n 分解为 2 的次幂, 那么枚举到 $\lfloor \log_2 n \rfloor$ 即可.

细节有了，那么我们具体如何实现呢?

我们可以把当前走到每一个素数前面的时候列举成一棵树的根节点, 然后一层层地去找. 找到什么时候停止呢?

1. 当前走到的数字已经大于我们想要的数字了;
2. 当前枚举的因子已经用不到了;
3. 当前因子大于我们想要的因子了;
4. 当前因子正好是我们想要的因子（此时判断是否需要更新最小 ans）.

然后 dfs 里面不断一层一层枚举次数继续往下迭代即可.

例题

Codeforces 27E. A number with a given number of divisors[6]

求具有给定除数个数的最小自然数．答案保证不超过 10^{18}.

解题思路

对于这种题，我们只要以因子数为 dfs 的返回条件基准，不断更新找到的最小值就可以了.

参考代码

```cpp
#include <iostream>
constexpr int p[15] = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41,
43, 47};
// p 中的素数乘积超过了 1e18

int n;
long long ans = 2e18;

// u: 当前考虑的质数在 p 中的下标
// num: 当前构成的数值
// cnt: 当前数值的因数个数
// pre: 上一个因子的幂次，限定本次选择的幂次
void dfs(int u, long long num, long long cnt, int pre) {
  if (cnt > n || u >= 15) return;
  if (cnt == n) return ans = std::min(ans, num), void();
  for (int i = 1; i <= pre; ++i) {
    if (num * p[u] > ans) break;  // 剪枝
    dfs(u + 1, num *= p[u], cnt * (i + 1), i);
  }
}

int main() {
  std::cin >> n;
```

```
  dfs(0, 1, 1, 59);  // floor(log2(1e18))=19
  std::cout << ans << std::endl;
  return 0;
}

```

ZOJ 2562 More Divisors[7]

求不超过 n 的数中，除数最多的数.

解题思路

思路同上，只不过要改改 dfs 的返回条件．注意这样的题目的数据范围，32 位整数可能溢出.

参考代码

```
#include <iostream>

int p[16] = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53};
unsigned long long n;
unsigned long long ans,
    ans_num;  // ans 为 n 以内的最大反素数（会持续更新），ans_sum 为
              // ans的因子数。

// depth: 当前在枚举第几个素数
// temp: 当前因子数量为 num 的时候的数值
// num: 当前因子数
// up: 上一个素数的幂，限制当前因子幂次上界
void dfs(int depth, unsigned long long temp, unsigned long long num, int
up) {
  if (depth >= 16 || temp > n) return;
  if (num > ans_num) {  // 更新答案
    ans = temp;
    ans_num = num;
  }
  if (num == ans_num && ans > temp) ans = temp;  // 更新答案
  for (int i = 1; i <= up; i++) {
    if (temp * p[depth] > n)
      break;  // 剪枝：如果加一个这个乘数的结果比ans要大，则必不是最佳方案
    dfs(depth + 1, temp *= p[depth], num * (i + 1),
        i);  // 取一个该乘数，进行对下一个乘数的搜索
  }
  return;
}

using std::cin;
using std::cout;

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
```

```
  while (cin >> n) {
    ans_num = 0;
    dfs(0, 1, 1, 60);
    cout << ans << '\n';
  }
  return 0;
}

```

参考资料与注释

1. Rui-Juan Jing, Marc Moreno-Maza, Delaram Talaashrafi, "Complexity Estimates for Fourier-Motzkin Elimination[8]", Journal of Functional Programming 16:2 (2006) pp 197-217.
2. 数论部分第一节：素数与素性测试[9]
3. Miller–Rabin 与 Pollard–Rho 学习笔记 - Bill Yang's Blog[10]
4. Primality test - Wikipedia[11]
5. Fermat pseudoprime - Wikipedia[12]
6. 桃子的算法笔记——反素数详解（acm/OI）[13]
7. The Rabin-Miller Primality Test[14]
8. Highly composite number - Wikipedia[15]

[1] $O(k \log^2 n \log\log n \log\log lo$

[2] 广义 Riemann 猜想

[3] Explicit bounds for primality testing and related problems

[4] Deterministic variant of the Miller–Rabin primality test

[5] emirp

[6] Codeforces 27E. A number with a given number of divisors

[7] ZOJ 2562 More Divisors

[8] Complexity Estimates for Fourier-Motzkin Elimination

[9] 数论部分第一节：素数与素性测试

[10] Miller–Rabin 与 Pollard–Rho 学习笔记 - Bill Yang's Blog

[11] Primality test - Wikipedia

[12] Fermat pseudoprime - Wikipedia

[13] 桃子的算法笔记——反素数详解（acm/OI）

[14] The Rabin-Miller Primality Test

[15] Highly composite number - Wikipedia

9.11.4 最大公约数

定义

最大公约数即为 Greatest Common Divisor，常缩写为 gcd.

一组整数的公约数，是指同时是这组数中每一个数的约数的数．± 1 是任意一组整数的公约数.

一组整数的最大公约数，是指所有公约数里面最大的一个.

对不全为 0 的整数 a, b，将其最大公约数记为 $\gcd(a, b)$，不引起歧义时可简写为 (a, b).

对不全为 0 的整数 $a_1, ..., a_n$，将其最大公约数记为 $\gcd(a_1, ..., a_n)$，不引起歧义时可简写为 $(a_1, ..., a_n)$.

最大公约数与最小公倍数的性质见 数论基础→9.11.1@p1160.

那么如何求最大公约数呢？我们先考虑两个数的情况.

欧几里得算法

过程

如果我们已知两个数 a 和 b，如何求出二者的最大公约数呢?

不妨设 $a > b$.

我们发现如果 b 是 a 的约数，那么 b 就是二者的最大公约数. 下面讨论不能整除的情况，即 $a = b \times q + r$，其中 $r < b$.

我们通过证明可以得到 $\gcd(a, b) = \gcd(b, a \bmod b)$，过程如下:

证明

设 $a = bk + c$，显然有 $c = a \bmod b$. 设 $d \mid a$，$d \mid b$，则 $c = a - bk$, $\frac{c}{d} = \frac{a}{d} - \frac{b}{d}k$.

由右边的式子可知 $\frac{c}{d}$ 为整数，即 $d \mid c$，所以对于 a, b 的公约数，它也会是 $b, a \bmod b$ 的公约数.

反过来也需要证明:

设 $d \mid b$，$d \mid (a \bmod b)$，我们还是可以像之前一样得到以下式子 $\frac{a \bmod b}{d} = \frac{a}{d} - \frac{b}{d}k$，$\frac{a \bmod b}{d} + \frac{b}{d}k = \frac{a}{d}$.

因为左边式子显然为整数，所以 $\frac{a}{d}$ 也为整数，即 $d \mid a$，所以 $b, a \bmod b$ 的公约数也是 a, b 的公约数.

既然两式公约数都是相同的，那么最大公约数也会相同.

所以得到式子 $\gcd(a, b) = \gcd(b, a \bmod b)$

既然得到了 $\gcd(a, b) = \gcd(b, r)$，这里两个数的大小是不会增大的，那么我们也就得到了关于两个数的最大公约数的一个递归求法.

实现

- C++

```cpp
// Version 1
int gcd(int a, int b) {
  if (b == 0) return a;
  return gcd(b, a % b);
}

// Version 2
int gcd(int a, int b) { return b == 0 ? a : gcd(b, a % b); }
```

- Java

```java
// Version 1
public int gcd(int a, int b) {
    if (b == 0) return a;
    return gcd(b, a % b);
}

// Version 2
public int gcd(int a, int b) {
    return b == 0 ? a : gcd(b, a % b);
}
```

- Python

```python
def gcd(a, b):
    if b == 0:
        return a
    return gcd(b, a % b)
```

递归至 `b == 0`（即上一步的 `a % b == 0`）的情况再返回值即可.

根据上述递归求法，我们也可以写出一个迭代求法:

- C++

```cpp
int gcd(int a, int b) {
  while (b != 0) {
    int tmp = a;
    a = b;
    b = tmp % b;
  }
  return a;
}
```

- Java

```java
public int gcd(int a, int b) {
    while(b != 0) {
        int tmp = a;
        a = b;
        b = tmp % b;
    }
    return a;
}
```

- Python

```python
def gcd(a, b):
    while b != 0:
        a, b = b, a % b
    return a
```

上述算法都可被称作欧几里得算法（Euclidean algorithm）.

另外，对于 C++17，我们可以使用 `<numeric>`[1] 头中的 `std::gcd`[2] 与 `std::lcm`[3] 来求最大公约数和最小公倍数.

> **⚠ 注意**
>
> 在部分编译器中，C++14 中可以用 `std::__gcd(a,b)` 函数来求最大公约数，但是其仅作为 `std::rotate` 的私有辅助函数.[16] 使用该函数可能会导致预期之外的问题, 故一般情况下不推荐使用.

如果两个数 a 和 b 满足 $\gcd(a,b)=1$，我们称 a 和 b 互质.

性质

欧几里得算法的时间效率如何呢? 下面我们证明，在输入为两个长为 n 的二进制整数时，欧几里得算法的时间复杂度为 $O(n)$.（换句话说，在默认 a,b 同阶的情况下，时间复杂度为 $O(\log \max(a,b))$.）

> **证明**
>
> 当我们求 $\gcd(a,b)$ 的时候，会遇到两种情况:

[16] libstdc++: std Namespace Reference[4]

- $a < b$，这时候 $\gcd(a,b) = \gcd(b,a)$；
- $a \geq b$，这时候 $\gcd(a,b) = \gcd(b, a \bmod b)$，而对 a 取模会让 a 至少折半．这意味着这一过程最多发生 $O(\log a) = O(n)$ 次．

第一种情况发生后一定会发生第二种情况，因此第一种情况的发生次数一定 **不多于** 第二种情况的发生次数．

从而我们最多递归 $O(n)$ 次就可以得出结果．

事实上，假如我们试着用欧几里得算法去求 斐波那契数列→9.13.4@p1629 相邻两项的最大公约数，会让该算法达到最坏复杂度．

更相减损术

大整数取模的时间复杂度较高，而加减法时间复杂度较低．针对大整数，我们可以用加减代替乘除求出最大公约数．

过程

已知两数 a 和 b，求 $\gcd(a,b)$．

不妨设 $a \geq b$，若 $a = b$，则 $\gcd(a,b) = a = b$．否则，$\forall d \mid a, d \mid b$，可以证明 $d \mid a - b$．

因此，a 和 b 的 **所有** 公因数都是 $a - b$ 和 b 的公因数，$\gcd(a,b) = \gcd(a-b,b)$．

Stein 算法的优化

如果 $a \gg b$，更相减损术的 $O(n)$ 复杂度将会达到最坏情况．

考虑一个优化，若 $2 \mid a, 2 \mid b$，$\gcd(a,b) = 2\gcd\left(\frac{a}{2}, \frac{b}{2}\right)$．

否则，若 $2 \mid a$（$2 \mid b$ 同理），因为 $2 \mid b$ 的情况已经讨论过了，所以 $2 \nmid b$．因此 $\gcd(a,b) = \gcd\left(\frac{a}{2}, b\right)$．

优化后的算法（即 Stein 算法）时间复杂度是 $O(\log n)$．

证明

若 $2 \mid a$ 或 $2 \mid b$，每次递归至少会将 a, b 之一减半．

否则，$2 \mid a - b$，回到了上一种情况．

算法最多递归 $O(\log n)$ 次．

实现

高精度模板见 高精度计算→9.6@p1110．

高精度运算需实现：减法、大小比较、左移、右移（可用低精乘除代替）、二进制末位 0 的个数（可以通过判断奇偶暴力计算）．

C++

```cpp
Big gcd(Big a, Big b) {
  if (a == 0) return b;
  if (b == 0) return a;
  // 记录a和b的公因数2出现次数，countr_zero表示二进制末位0的个数
  int atimes = countr_zero(a);
  int btimes = countr_zero(b);
  int mintimes = min(atimes, btimes);
  a >>= atimes;
```

```
  for (;;) {
    // a和b公因数中的2已经计算过了，后面不可能出现a为偶数的情况
    b >>= btimes;
    // 确保 a<=b
    if (a > b) swap(a, b);
    b -= a;
    if (b == 0) break;
    btimes = countr_zero(b);
  }
  return a << mintimes;
}
```

上述代码参考了 libstdc++[5] 和 MSVC[6] 对 C++17 `std::gcd` 的实现. 在 `unsigned int` 和 `unsigned long long` 的数据范围下，如果可以以极快的速度计算 `countr_zero`，则 Stein 算法比欧几里得算法来得快，但反之则可能比欧几里得算法慢.

关于 countr_zero

1. gcc 有 内建函数→9.4@p1099 `__builtin_ctz`（32 位）或 `__builtin_ctzll`（64 位）可替换上述代码的 `countr_zero`;
2. 从 C++20 开始，头文件 `<bit>` 包含了 `std::countr_zero`[7];
3. 如果不使用不在标准库的函数，又无法使用 C++20 标准，下面的代码是一种在 Word-RAM with multiplication 模型下经过预处理后 $O(1)$ 的实现:

```
constexpr int loghash[64] = {0,  32, 48, 56, 60, 62, 63, 31, 47, 55, 59,
61, 30,
                             15, 39, 51, 57, 28, 46, 23, 43, 53, 58, 29,
14, 7,
                             35, 49, 24, 44, 54, 27, 45, 22, 11, 37, 50,
25, 12,
                             38, 19, 41, 52, 26, 13, 6,  3,  33, 16, 40,
20, 42,
                             21, 10, 5,  34, 17, 8,  36, 18, 9,  4,  2,  1};

int countr_zero(unsigned long long x) {
  return loghash[(x & -x) * 0x9150D32D8EB9EFC0Ui64 >> 58];
}
```

而对于高精度运算，如果实现方法类似 `bitset`，则搭配上述对 `countr_zero` 的实现可以在 `O(n / w)` 的时间复杂度下完成. 但如果不便按二进制位拆分，则只能暴力判断最大的 2 的幂因子，时间复杂度取决于实现. 比如:

```
// 以小端序实现的二进制 Big，要求能枚举每一个元素
int countr_zero(Big a) {
  int ans = 0;
  for (auto x : a) {
    if (x != 0) {
      ans += 32;  // 每一位数据类型的位长
    } else {
      return ans + countr_zero(x);
    }
```

```
  }
  return ans;
}

// 暴力计算，如需使用建议直接写进 gcd 加快常数
int countr_zero(Big a) {
  int ans = 0;
  while ((a & 1) == 0) {
    a >>= 1;
    ++ans;
  }
  return ans;
}
```

更多关于 `gcd` 实现上快慢的讨论可阅读 Fastest way to compute the greatest common divisor[8].

多个数的最大公约数

那怎么求多个数的最大公约数呢？显然答案一定是每个数的约数，那么也一定是每相邻两个数的约数．我们采用归纳法，可以证明，每次取出两个数求出答案后再放回去，不会对所需要的答案造成影响．

最小公倍数

接下来我们介绍如何求解最小公倍数（Least Common Multiple, LCM）．

定义

一组整数的公倍数，是指同时是这组数中每一个数的倍数的数．0 是任意一组整数的公倍数．

一组整数的最小公倍数，是指所有正的公倍数里面，最小的一个数．

对整数 a, b，将其最小公倍数记为 $\operatorname{lcm}(a, b)$，不引起歧义时可简写为 $[a, b]$.

对整数 $a_1, ..., a_n$，将其最小公倍数记为 $\operatorname{lcm}(a_1, ..., a_n)$，不引起歧义时可简写为 $[a_1, ..., a_n]$.

两个数

设 $a = p_1^{k_{a_1}} p_2^{k_{a_2}} \cdots p_s^{k_{a_s}}$，$b = p_1^{k_{b_1}} p_2^{k_{b_2}} \cdots p_s^{k_{b_s}}$

我们发现，对于 a 和 b 的情况，二者的最大公约数等于

$p_1^{\min(k_{a_1}, k_{b_1})} p_2^{\min(k_{a_2}, k_{b_2})} \cdots p_s^{\min(k_{a_s}, k_{b_s})}$

最小公倍数等于

$p_1^{\max(k_{a_1}, k_{b_1})} p_2^{\max(k_{a_2}, k_{b_2})} \cdots p_s^{\max(k_{a_s}, k_{b_s})}$

由于 $k_a + k_b = \max(k_a, k_b) + \min(k_a, k_b)$

所以得到结论是 $\gcd(a, b) \times \operatorname{lcm}(a, b) = a \times b$

要求两个数的最小公倍数，先求出最大公约数即可．

多个数

可以发现，当我们求出两个数的 gcd 时，求最小公倍数是 $O(1)$ 的复杂度．那么对于多个数，我们其实没有必要求一个共同的最大公约数再去处理，最直接的方法就是，当我们算出两个数的 gcd，或许在求多个数的 gcd 时候，我们将它放入序列对后面的数继续求解，那么，我们转换一下，直接将最小公倍数放入序列即可．

扩展欧几里得算法

扩展欧几里得算法（Extended Euclidean algorithm, EXGCD），常用于求 $ax+by=\gcd(a,b)$ 的一组可行解.

过程

设

$ax_1+by_1=\gcd(a,b)$

$bx_2+(a \bmod b)y_2=\gcd(b,a \bmod b)$

由欧几里得定理可知：$\gcd(a,b)=\gcd(b,a \bmod b)$

所以 $ax_1+by_1=bx_2+(a \bmod b)y_2$

又因为 $a \bmod b=a-(\lfloor \frac{a}{b} \rfloor \times b)$

所以 $ax_1+by_1=bx_2+(a-(\lfloor \frac{a}{b} \rfloor \times b))y_2$

$ax_1+by_1=ay_2+bx_2-\lfloor \frac{a}{b} \rfloor \times by_2=ay_2+b(x_2-\lfloor \frac{a}{b} \rfloor y_2)$

因为 $a=a,b=b$，所以 $x_1=y_2,y_1=x_2-\lfloor \frac{a}{b} \rfloor y_2$

将 x_2,y_2 不断代入递归求解直至 b 为 0 递归 $x=1,y=0$ 回去求解.

实现

- C++

```cpp
int Exgcd(int a, int b, int &x, int &y) {
  if (!b) {
    x = 1;
    y = 0;
    return a;
  }
  int d = Exgcd(b, a % b, x, y);
  int t = x;
  x = y;
  y = t - (a / b) * y;
  return d;
}
```

- Python

```python
def Exgcd(a, b):
    if b == 0:
        return a, 1, 0
    d, x, y = Exgcd(b, a % b)
    return d, y, x - (a // b) * y
```

函数返回的值为 gcd，在这个过程中计算 x,y 即可.

值域分析

$ax+by=\gcd(a,b)$ 的解有无数个，显然其中有的解会爆 long long.

万幸的是，若 $b \neq 0$，扩展欧几里得算法求出的可行解必有 $|x| \leq b,|y| \leq a$.

下面给出这一性质的证明.

证明

- $\gcd(a,b)=b$ 时，$a \bmod b=0$，必在下一层终止递归.

得到 $x_1 = 0, y_1 = 1$，显然 $a, b \geq 1 \geq |x_1|, |y_1|$.

- $\gcd(a, b) \neq b$ 时，设 $|x_2| \leq (a \bmod b), |y_2| \leq b$.

  因为 $x_1 = y_2, y_1 = x_2 - \left\lfloor \frac{a}{b} \right\rfloor y_2$

  所以 $|x_1| = |y_2| \leq b, |y_1| \leq |x_2| + \left| \left\lfloor \frac{a}{b} \right\rfloor y_2 \right| \leq (a \bmod b) + \left\lfloor \frac{a}{b} \right\rfloor |y_2|$

  $\leq a - \left\lfloor \frac{a}{b} \right\rfloor b + \left\lfloor \frac{a}{b} \right\rfloor |y_2| \leq a - \left\lfloor \frac{a}{b} \right\rfloor (b - |y_2|)$

  $a \bmod b = a - \left\lfloor \frac{a}{b} \right\rfloor b \leq a - \left\lfloor \frac{a}{b} \right\rfloor (b - |y_2|) \leq a$

  因此 $|x_1| \leq b, |y_1| \leq a$ 成立.

迭代法编写扩展欧几里得算法

首先，当 $x = 1$，$y = 0$，$x_1 = 0$，$y_1 = 1$ 时，显然有：

$$\begin{cases} ax + by = a \\ ax_1 + by_1 = b \end{cases}$$

成立.

已知 $a \bmod b = a - (\lfloor \frac{a}{b} \rfloor \times b)$，下面令 $q = \lfloor \frac{a}{b} \rfloor$. 参考迭代法求 gcd，每一轮的迭代过程可以表示为：

$$(a, b) \to (b, a - qb)$$

将迭代过程中的 a 替换为 $ax + by = a$，b 替换为 $ax_1 + by_1 = b$，可以得到：

$$\begin{cases} ax + by = a \\ ax_1 + by_1 = b \end{cases}$$

$$\to \begin{cases} ax_1 + by_1 = b \\ a(x - qx_1) + b(y - qy_1) = a - qb \end{cases}$$

据此就可以得到迭代法求 exgcd.

因为迭代的方法避免了递归，所以代码运行速度将比递归代码快一点.

```
int gcd(int a, int b, int& x, int& y) {
  x = 1, y = 0;
  int x1 = 0, y1 = 1, a1 = a, b1 = b;
  while (b1) {
    int q = a1 / b1;
    tie(x, x1) = make_tuple(x1, x - q * x1);
    tie(y, y1) = make_tuple(y1, y - q * y1);
    tie(a1, b1) = make_tuple(b1, a1 - q * b1);
  }
  return a1;
}
```

如果你仔细观察 a_1 和 b_1，你会发现，他们在迭代版本的欧几里德算法中取值完全相同，并且以下公式无论何时（在 while 循环之前和每次迭代结束时）都是成立的：$x \cdot a + y \cdot b = a_1$ 和 $x_1 \cdot a + y_1 \cdot b = b_1$. 因此，该算法肯定能正确计算出 gcd.

最后我们知道 a_1 就是要求的 gcd，有 $x \cdot a + y \cdot b = g$.

矩阵的解释

对于正整数 a 和 b 的一次辗转相除即 $\gcd(a, b) = \gcd(b, a \bmod b)$ 使用矩阵表示如

$$\begin{bmatrix} b \\ a \bmod b \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & -\lfloor a/b \rfloor \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix}$$

其中向下取整符号 $\lfloor c \rfloor$ 表示不大于 c 的最大整数. 我们定义变换 $\begin{bmatrix} a \\ b \end{bmatrix} \mapsto \begin{bmatrix} 0 & 1 \\ 1 & -\lfloor a/b \rfloor \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix}$.

易发现欧几里得算法即不停应用该变换，有

$$\begin{bmatrix}\gcd(a,b)\\0\end{bmatrix}=\left(\cdots\begin{bmatrix}0&1\\1&-\lfloor a/b\rfloor\end{bmatrix}\begin{bmatrix}1&0\\0&1\end{bmatrix}\right)\begin{bmatrix}a\\b\end{bmatrix}$$

令

$$\begin{bmatrix}x_1&x_2\\x_3&x_4\end{bmatrix}=\cdots\begin{bmatrix}0&1\\1&-\lfloor a/b\rfloor\end{bmatrix}\begin{bmatrix}1&0\\0&1\end{bmatrix}$$

那么

$$\begin{bmatrix}\gcd(a,b)\\0\end{bmatrix}=\begin{bmatrix}x_1&x_2\\x_3&x_4\end{bmatrix}\begin{bmatrix}a\\b\end{bmatrix}$$

满足 $a\cdot x_1+b\cdot x_2=\gcd(a,b)$ 即扩展欧几里得算法，注意在最后乘了一个单位矩阵不会影响结果，提示我们可以在开始时维护一个 2×2 的单位矩阵编写更简洁的迭代方法如

```cpp
int exgcd(int a, int b, int &x, int &y) {
  int x1 = 1, x2 = 0, x3 = 0, x4 = 1;
  while (b != 0) {
    int c = a / b;
    std::tie(x1, x2, x3, x4, a, b) =
        std::make_tuple(x3, x4, x1 - x3 * c, x2 - x4 * c, b, a - b * c);
  }
  x = x1, y = x2;
  return a;
}
```

这种表述相较于递归更简单.

应用

- 10104 - Euclid Problem[9]
- GYM - (J) once upon a time[10]
- UVa - 12775 - Gift Dilemma[11]

参考资料与链接

[1] `<numeric>` [2] `std::gcd` [3] `std::lcm`

[4] libstdc++: std Namespace Reference [5] libstdc++ [6] MSVC

[7] `std::countr_zero` [8] Fastest way to compute the greatest common divisor [9] 10104 - Euclid Problem

[10] GYM - (J) once upon a time [11] UVa - 12775 - Gift Dilemma

9.11.5 欧拉函数

> **Authors:** iamtwz, Chrogeek, Enter-tainer, StudyingFather, aofall, CCXXXI, CoelacanthusHex, frank-xjh, Great-designer, greyqz, guodong2005, henrytbtrue, Ir1d, kZime, lihaoyu1234, Marcythm, MegaOwIer, Menci, nalemy, orzAtalod, ouuan, Persdre, segment-tree, ShaoChenHeng, shuzhouliu, sshwy, Struggler-q, Tiphereth-A, TrisolarisHD, Xeonacid, yuhuoji

定义

欧拉函数（Euler's totient function），即 $\varphi(n)$，表示的是小于等于 n 且与 n 互质的数的个数.

比如说 $\varphi(1)=1$.

当 n 是质数的时候，显然有 $\varphi(n)=n-1$.

性质

- 欧拉函数是 积性函数→9.11.1@p1160.

  即对任意满足 $\gcd(a,b)=1$ 的整数 a,b，有 $\varphi(ab)=\varphi(a)\varphi(b)$.

  特别地，当 n 是奇数时 $\varphi(2n)=\varphi(n)$.

  证明参见 剩余系的复合→9.11.1@p1160.

- $n=\sum_{d|n}\varphi(d)$.

  证明

  利用 莫比乌斯反演→9.11.23@p1331 相关知识可以得出.

  也可以这样考虑：如果 $\gcd(k,n)=d$，那么 $\gcd(\frac{k}{d},\frac{n}{d})=1,(k<n)$.

  如果我们设 $f(x)$ 表示 $\gcd(k,n)=x$ 的数的个数，那么 $n=\sum_{i=1}^{n}f(i)$.

  根据上面的证明，我们发现，$f(x)=\varphi(\frac{n}{x})$，从而 $n=\sum_{d|n}\varphi(\frac{n}{d})$. 注意到约数 d 和 $\frac{n}{d}$ 具有对称性，所以上式化为 $n=\sum_{d|n}\varphi(d)$.

- 若 $n=p^k$，其中 p 是质数，那么 $\varphi(n)=p^k-p^{k-1}$. （根据定义可知）
- 由唯一分解定理，设 $n=\prod_{i=1}^{s}p_i^{k_i}$，其中 p_i 是质数，有 $\varphi(n)=n\times\prod_{i=1}^{s}\frac{p_i-1}{p_i}$.

  证明

  - 引理：设 p 为任意质数，那么 $\varphi(p^k)=p^{k-1}\times(p-1)$.

    证明：显然在从 1 到 p^k 的所有数中，除了 p^{k-1} 个 p 的倍数以外其它数都与 p^k 互素，故 $\varphi(p^k)=p^k-p^{k-1}=p^{k-1}\times(p-1)$，证毕.

  接下来我们证明 $\varphi(n)=n\times\prod_{i=1}^{s}\frac{p_i-1}{p_i}$. 由唯一分解定理与 $\varphi(x)$ 函数的积性可知

  $$\begin{aligned}\varphi(n)&=\prod_{i=1}^{s}\varphi(p_i^{k_i})\\&=\prod_{i=1}^{s}(p_i-1)\times p_i^{k_i-1}\\&=\prod_{i=1}^{s}p_i^{k_i}\times(1-\frac{1}{p_i})\\&=n\prod_{i=1}^{s}(1-\frac{1}{p_i})\end{aligned}\qquad\square$$

- 对任意正整数 m,n，$\varphi(mn)\varphi(\gcd(m,n))=\varphi(m)\varphi(n)\gcd(m,n)$.

  可由上一条直接计算得出.

实现

如果只要求一个数的欧拉函数值，那么直接根据定义，在质因数分解的同时求就好了. 这个过程可以用 Pollard Rho→9.11.7@p1209 算法优化.

参考实现

- C++

```cpp
#include <cmath>

int euler_phi(int n) {
  int ans = n;
  for (int i = 2; i * i <= n; i++)
    if (n % i == 0) {
      ans = ans / i * (i - 1);
      while (n % i == 0) n /= i;
    }
  if (n > 1) ans = ans / n * (n - 1);
  return ans;
}
```

- Python

```python
import math


def euler_phi(n):
    ans = n
    for i in range(2, math.isqrt(n) + 1):
        if n % i == 0:
            ans = ans // i * (i - 1)
            while n % i == 0:
                n = n // i
    if n > 1:
        ans = ans // n * (n - 1)
    return ans
```

如果是多个数的欧拉函数值，可以利用后面会提到的线性筛法来求得.

详见：筛法求欧拉函数→9.11.6@p1199

应用

欧拉函数常常用于化简一列最大公约数的和．国内有些文章称它为 **欧拉反演** [17].

在结论

$$n = \sum_{d|n} \varphi(d)$$

中代入 $n = \gcd(a, b)$，则有

$$\gcd(a,b) = \sum_{d|\gcd(a,b)} \varphi(d) = \sum_{d} [d \mid a][d \mid b]\varphi(d),$$

其中 $[\cdot]$ 为 Iverson 括号．对上式求和，就可以得到

$$\sum_{i=1}^{n} \gcd(i,n) = \sum_{d} \sum_{i=1}^{n} [d \mid i][d \mid n]\varphi(d) = \sum_{d} \left\lfloor \frac{n}{d} \right\rfloor [d \mid n]\varphi(d) = \sum_{d|n} \left\lfloor \frac{n}{d} \right\rfloor \varphi(d).$$

这里关键的观察是 $\sum_{i=1}^{n} [d \mid i] = \left\lfloor \frac{n}{d} \right\rfloor$，即在 1 和 n 之间能够被 d 整除的 i 的个数是 $\left\lfloor \frac{n}{d} \right\rfloor$.

利用这个式子，就可以遍历约数求和了．需要多组查询的时候，可以预处理欧拉函数的前缀和，利用数论分块查询.

[17] 这一说法并未见于学术期刊或国外的论坛中，在使用该说法时应当注意.

GCD SUM[1]

给定 $n \leq 100000$，求

$$\sum_{i=1}^{n}\sum_{j=1}^{n}\gcd(i,j).$$

思路

仿照上文的推导，可以得出

$$\sum_{i=1}^{n}\sum_{j=1}^{n}\gcd(i,j)=\sum_{d=1}^{n}\left\lfloor\frac{n}{d}\right\rfloor^2\varphi(d).$$

此时需要从 1 遍历到 n 求欧拉函数，用线性筛做就可以 $O(n)$ 得到答案.

欧拉定理

与欧拉函数紧密相关的一个定理就是欧拉定理. 其描述如下:

若 $\gcd(a,m)=1$，则 $a^{\varphi(m)}\equiv 1 \pmod m$.

扩展欧拉定理

当然也有扩展欧拉定理，用于处理一般的 a 和 m 的情形.

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

证明和习题详见 欧拉定理→9.11.9@p1223.

习题

- SPOJ ETF. Euler Totient Function[2]
- UVa 10179. Irreducible Basic Fractions[3]
- UVa 10299. Relatives[4]
- UVa 11327. Enumerating Rational Numbers[5]
- TIMUS 1673. Admission to Exam[6]
- Luogu P1390 公约数的和[7]
- Luogu P2155 [SDOI2008] 沙拉公主的困惑[8]
- Luogu P2568 GCD[9]

参考资料与注释

[1] GCD SUM

[2] SPOJ ETF. Euler Totient Function

[3] UVa 10179. Irreducible Basic Fractions

[4] UVa 10299. Relatives

[5] UVa 11327. Enumerating Rational Numbers

[6] TIMUS 1673. Admission to Exam

[7] Luogu P1390 公约数的和

[8] Luogu P2155 [SDOI2008] 沙拉公主的困惑

[9] Luogu P2568 GCD

9.11.6 筛法

Authors: inkydragon, TravorLZH, YOYO-UIAT, wood3, shuzhouliu, Mr-Python-in-China, HeRaNO, weilycoder

素数筛法

引入

如果我们想要知道小于等于 n 有多少个素数呢?

一个自然的想法是对于小于等于 n 的每个数进行一次质数检验．这种暴力的做法显然不能达到最优复杂度.

埃拉托斯特尼筛法

过程

考虑这样一件事情：对于任意一个大于 1 的正整数 n，那么它的 x 倍就是合数（$x > 1$）．利用这个结论，我们可以避免很多次不必要的检测.

如果我们从小到大考虑每个数，然后同时把当前这个数的所有（比自己大的）倍数记为合数，那么运行结束的时候没有被标记的数就是素数了.

实现

- 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;
  for (int i = 2; i <= n; ++i) {
    if (is_prime[i]) {
      prime.push_back(i);
      if ((long long)i * i > n) continue;
      for (int j = i * i; j <= n; j += i)
        // 因为从 2 到 i - 1 的倍数我们之前筛过了，这里直接从 i
        // 的倍数开始，提高了运行速度
        is_prime[j] = false;  // 是 i 的倍数的均不是素数
    }
  }
}
```

- 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
    for i in range(2, n + 1):
        if is_prime[i]:
            prime.append(i)
            if i * i > n:
                continue
            for j in range(i * i, n + 1, i):
                is_prime[j] = False
```

以上为 **Eratosthenes 筛法**（埃拉托斯特尼筛法，简称埃氏筛法），时间复杂度是 $O(n \log \log n)$.