```
52          mi = tab[i][m + 1];
53        }
54      }
55      if (x == -1) return false;
56      int y = -1;
57      mi = INFINITY;
58      for (int j = 0; j < m; ++j) {
59        if ((B[j] < 0 && tab[x][j] < -eps) || (B[j] >= 0 && tab[x][j] >
eps)) {
60          auto tmp = tab[n][j] / tab[x][j] + (B[j] < 0 ? -eps : eps);
61          if (tmp < mi) {
62            y = j;
63            mi = tmp;
64          }
65        }
66      }
67      if (B[y] < 0) {
68        --neg_count;
69        B[y] = ~B[y];
70        pivot(x, y);
71        tab[x][m + 1] += 1;
72      } else {
73        pivot(x, y);
74      }
75    }
76    return true;
77  }
78
79  // Second phase: find an optimal BFS.
80  // Return false if the problem is unbounded.
81  bool simplex() {
82    while (true) {
83      int x = -1;
84      long double mi = -eps;
85      for (int i = 0; i < n; ++i) {
86        if (tab[i][m] < mi) {
87          x = i;
88          mi = tab[i][m];
89        }
90      }
91      // No column with a negative reduced cost => Optimal.
92      if (x == -1) break;
93      int y = -1;
94      mi = INFINITY;
95      for (int j = 0; j < m; ++j) {
96        if (tab[x][j] <= eps) continue;
97        if (tab[n][j] / tab[x][j] < mi) {
98          y = j;
99          mi = tab[n][j] / tab[x][j];
100       }
101     }
102     // No row with a positive ratio => Unbounded.
```

```
    if (y == -1) return false;
    pivot(x, y);
  }
  return true;
}

int solve() {
  B.resize(m);
  N.resize(n);
  std::iota(B.begin(), B.end(), n);
  std::iota(N.begin(), N.end(), 0);
  return initialize() ? (simplex() ? 0 : 1) : -1;
}

int main() {
  std::ios::sync_with_stdio(false), std::cin.tie(nullptr);
  std::cout << std::fixed << std::setprecision(8);
  std::cin >> n >> m;
  tab.assign(n + 1, std::vector<long double>(m + 2));
  for (int i = 0; i < n; ++i) {
    std::cin >> tab[i][m];
    tab[i][m] = -tab[i][m];
  }
  for (int j = 0; j < m; ++j) {
    for (int i = 0; i <= n; ++i) {
      std::cin >> tab[i][j];
    }
  }
  switch (solve()) {
    case -1:
      std::cout << "Infeasible" << std::endl;
      break;
    case 1:
      std::cout << "Unbounded" << std::endl;
      break;
    case 0: {
      std::cout << tab[n][m] << std::endl;
      std::vector<long double> x(n);
      for (int j = 0; j < m; ++j) {
        if (B[j] < n) x[B[j]] = tab[n][j];
      }
      for (int i = 0; i < n; ++i) {
        std::cout << x[i] << ' ';
      }
      std::cout << std::endl;
      break;
    }
  }
  return 0;
}

```

例题

「NOI2008」志愿者招募[3]

总共 n 天活动需要招募志愿者，其中，第 i 天至少需要 b_i 位志愿者．总有 m 类志愿者，其中，第 j 类志愿者可以服务的日期为连续区间 $[l_j, r_j]$，且单位招募成本为 c_j．求最优的招募方案，使得招募志愿者的成本最低．

解答

设第 j 类志愿者招募 x_j 位．那么，可以列出线性规划问题为

$$\min_x \sum_{j=1}^{m} c_j x_j$$

$$\text{subject to} \sum_{j=1}^{m} a_{ij} x_j \geq b_i,\ i = 1, \cdots, n,$$

$$x_j \geq 0,\ j = 1, \cdots, m.$$

其中，系数

$$a_{ij} = \begin{cases} 1, l_j \leq i \leq r_j, \\ 0, \text{otherwise}. \end{cases}$$

原问题没有显然的初始可行解．因此，不妨考虑其 对偶问题→9.15.1@p1774：

$$\max_y \sum_{i=1}^{n} b_i y_i$$

$$\text{subject to} \sum_{i=1}^{n} a_{ij} y_i \leq c_j,\ j = 1, \cdots, m,$$

$$y_i \geq 0,\ i = 1, \cdots, n.$$

通过添加松弛变量，容易得到一组初始可行解，可以直接略过一阶段，通过单纯形法求解．根据对偶原理，得到的解就是原问题的解．

```cpp
#include <cmath>
#include <cstring>
#include <iostream>
using namespace std;
constexpr int M = 10005, N = 1005, INF = 1e9;

int n, m;
double a[M][N], b[M], c[N], v;

void pivot(int l, int e) {  // 转轴操作函数
  b[l] /= a[l][e];
  for (int j = 1; j <= n; j++)
    if (j != e) a[l][j] /= a[l][e];
  a[l][e] = 1 / a[l][e];

  for (int i = 1; i <= m; i++)
    if (i != l && fabs(a[i][e]) > 0) {
      b[i] -= a[i][e] * b[l];
      for (int j = 1; j <= n; j++)
        if (j != e) a[i][j] -= a[i][e] * a[l][j];
```

```
      a[i][e] = -a[i][e] * a[l][e];
    }

  v += c[e] * b[l];
  for (int j = 1; j <= n; j++)
    if (j != e) c[j] -= c[e] * a[l][j];
  c[e] = -c[e] * a[l][e];

  // swap(B[l],N[e])
}

double simplex() {
  while (true) {
    int e = 0, l = 0;
    for (e = 1; e <= n; e++)
      if (c[e] > (double)0) break;
    if (e == n + 1) return v;  // 此时v即为最优解
    double mn = INF;
    for (int i = 1; i <= m; i++) {
      if (a[i][e] > (double)0 && mn > b[i] / a[i][e]) {
        mn = b[i] / a[i][e];  // 找对这个e限制最紧的l
        l = i;
      }
    }
    if (mn == INF) return INF;  // unbounded
    pivot(l, e);                // 转动l,e
  }
}

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  cin >> n >> m;
  for (int i = 1; i <= n; i++) cin >> c[i];
  for (int i = 1; i <= m; i++) {
    int s, t;
    cin >> s >> t;
    for (int j = s; j <= t; j++) a[i][j] = 1;  // 表示第i种志愿者在j时间可以服务
    cin >> b[i];
  }
  cout << (int)(simplex() + 0.5);
}

```

习题

- Luogu P13337【模板】线性规划[4]
- UOJ#179. 线性规划[5]
- Luogu P4232 无意识之外的捉迷藏[6]
- Codeforces 1430 G. Yet Another DAG Problem[7]
- AtCoder Beginner Contest 231 H - Minimum Coloring[8]

参考资料

- 线性规划之单纯形法【超详解 + 图解】[9]
- 2016 国家集训队论文[10]
- 算法导论
- Matoušek, Jiří, and Bernd Gärtner. Understanding and using linear programming. Vol. 1. Berlin: Springer, 2007.
- Inayatullah, Syed, Nasir Touheed, and Muhammad Imtiaz. "A streamlined artificial variable free version of simplex method." PloS one 10, no. 3 (2015): e0116156.
- Floudas, Christodoulos A., and Panos M. Pardalos, eds. Encyclopedia of optimization. Springer Science & Business Media, 2008.

[1] 这份讲义

[2] Luogu P13337【模板】线性规划

[3] 「NOI2008」志愿者招募

[4] Luogu P13337【模板】线性规划

[5] UOJ#179. 线性规划

[6] Luogu P4232 无意识之外的捉迷藏

[7] Codeforces 1430 G. Yet Another DAG Problem

[8] AtCoder Beginner Contest 231 H - Minimum Coloring

[9] 线性规划之单纯形法【超详解 + 图解】

[10] 2016 国家集训队论文

9.16 抽象代数

9.16.1 基本概念

Authors: jifbt, billchenchina, Enter-tainer, Great-designer, iamtwz, ImpleLee, isdanni, Menci, ouuan, Tiphereth-A, warzone-oier, Xeonacid, c-forrest

本章节将简要介绍抽象代数的相关知识. 现阶段算法竞赛的主要内容并不直接考察抽象代数的知识, 但是在算法的描述或是问题的题解中常常会牵涉一些抽象代数的基本概念, 这使得掌握了基础抽象代数概念的读者能够更快速理解一些算法. 因此, 这部分内容并不是任何选手的必修知识, 而仅供那些感兴趣或者可能从中受益的读者参考使用. 同时, 本章节将避免过全过深的介绍抽象代数的知识[1], 而会集中在基础概念以及与 OI 其他部分知识联系最为紧密的部分. 想系统学习抽象代数知识的读者, 应当参考专业的抽象代数教科书学习.

为了更好帮助读者理解阅读本部分内容可能的收获, 列举一些算法竞赛中可能牵涉到抽象代数知识的例子:

- 数论和多项式的很多定理是抽象代数中结论的特例;
- 数据结构中, 线段树→10.13.1@p2098 等结构可以维护幺半群的信息, 而很多 DP 问题的递推关系可以抽象成这样的幺半群结构;
- 组合数学中, Pólya 计数原理→9.13.14@p1680 的严格表述和证明需要用到群论的相关概念.

基于此, 本章节将着重介绍无法跳过的基础知识和与这些应用直接相关的部分. 作为开始, 本文介绍群、环、域的基本概念.

群

群的定义如下.

[1] 因为 OI Wiki 不是百科全书→1.4@p21.

群

设 G 是非空集合，其上有二元运算 $\cdot : G \times G \to G$，如果它们满足以下性质，则称 $(G, \cdot)$ 是一个**群**（group）：

1. 结合律（associative property）：对于所有 $a, b, c \in G$，成立 $a \cdot (b \cdot c) = (a \cdot b) \cdot c$；
2. 有单位元：存在 $e \in G$，使得对于任意 $a \in G$，都成立 $a \cdot e = e \cdot a = a$．这里，$e$ 称为 G 的**单位元**（identity element），也称幺元；
3. 存在逆元：对于所有 $a \in G$，都存在相应的 $b \in G$ 使得 $a \cdot b = b \cdot a = e$．这里，$b$ 称为 a 的**逆元**（inverse element）．

关于定义中的封闭性条件

这里的二元运算就隐含了所谓的封闭性条件，即对于任何 $a, b \in G$，都有 $a \cdot b \in G$．有些文章会将其单独列出．

群的基本性质

对于群 $(G, \cdot)$，以下性质总是成立：

1. 对于任何有限长的列 $\{g_i\}_{i=1}^{k} \subseteq G$，乘积 $g_1 \cdot g_2 \cdot \cdots \cdot g_k$ 的运算结果与加括号的方式无关；
2. 单位元 e 总是唯一的；
3. 对于任何元素 $a \in G$，它的逆 a^{-1} 也是唯一的；
4. 消去律（cancellation law）：对于 $a, b, c \in G$，如果 $a \cdot c = b \cdot c$ 或 $c \cdot a = c \cdot b$，那么有 $a = b$.

群相当常见．通俗地说，所有不损失结构的变换都自动构成群．以常见的几种类型的群为例．

群的例子

- **对称群**（symmetric group）：集合 M 上的所有 置换→9.8@p1138，即自 M 到 M 自身的双射，就在映射的复合下构成群 S_M．单位元是恒等变换，逆元是逆映射（双射必然存在逆映射）．如果集合 M 有限，大小为 n，也常记作 S_n，称作 n 次对称群．
- 空间对称群（symmetry group）：对于一个几何图形，能够使其与自身重合的变换全体也在映射的复合下构成群．这描述了该几何图形的空间对称性．具体例子可以参考 常见空间对称群→9.13.14@p1680.
- 整数的加法群：整数集 $\mathbf{Z}$ 在加法 $+$ 运算下构成群 $(\mathbf{Z}, +)$．单位元是 0，逆元是相反数．
- 整数模 n 乘法群（multiplicative group of integers modulo n）：对于一个模数 n，所有与 n 互质的整数对应的 同余类→9.11.1@p1160，在乘法运算下构成群 $((\mathbf{Z}/n\mathbf{Z})^{\times}, \times)$．单位元是 $\bar{1}$，逆元就是模 n 的 乘法逆元→9.11.10@p1229（对应的同余类），其存在性由 裴蜀定理→9.11.8@p1218 保证．具体结构分析参考 整数模 n 乘法群→9.16.3@p1830.
- 一般线性群（general linear group）：数域 F 上的 n 维的全体可逆方阵在乘法运算下构成群 $GL_n(F)$．单位元是单位矩阵，逆元是逆矩阵．

要更好地理解群的定义，不妨对比着看几个不属于群的例子．

不是群的例子

- 所有 M 到自身的映射（不一定是双射），并不构成群．因为那些不是双射的映射不存在逆元．
- 整数在乘法下并不构成群，因为 2 在整数范围内没有乘法逆元．
- 正整数在加法下也不构成群，因为正整数没有加法单位元．

- 模 n 的所有非零同余类在乘法意义下往往不构成群．比如说 $(\mathbf{Z}/6\mathbf{Z})\setminus\{\overline{0}\}$ 中，$\overline{2}\times\overline{3}=\overline{0}$ 不属于这个集合，这意味着乘法不是这个集合上良定义的二元运算（或者说，它不满足封闭性）．

有时，也需要讨论这些更不完善的结构的性质．因此，可以定义如下概念，它们比群更宽泛．

自 半群

对于非空集合 G 和其上的二元运算 $\cdot$，如果该运算满足结合律，则称 $(G,\cdot)$ 是一个 **半群**（semigroup）．

自 幺半群

对于半群 $(G,\cdot)$，如果它还存在单位元，则称 $(G,\cdot)$ 是一个 **幺半群**（monoid）．

幺半群和半群的例子

上面的例子中，$(\mathbf{N}_+,+)$ 是半群，而 $(\mathbf{Z},\times)$ 是幺半群．

最后，很多熟悉的群上的运算除了满足结合律外，还满足交换律．这类群的结构相对简单，它们称作 Abel 群，也称作交换群．

自 Abel 群

对于群 $(G,\cdot)$，如果运算 $\cdot$ 还满足交换律（commutative property），即对于所有 $a,b\in G$，都成立 $a\cdot b=b\cdot a$，则称 $(G,\cdot)$ 是一个 **Abel 群**（Abelian group）或 **交换群**（commutative group）．

Abel 群和非 Abel 群的例子

- 整数加法群 $(\mathbf{Z},+)$ 就是一个 Abel 群．
- 当 $n\geq 3$ 时，对称群 S_n 并不是 Abel 群．

这些就是群论相关的基本定义．群论的更多内容，可以参考 群论→9.16.2@p1812 或相关书籍．

环

环的定义如下．

自 环

对于非空集合 R 和其上的两个二元运算 $+:R\times R\to R$ 和 $\cdot:R\times R\to R$，如果它们满足以下性质，则称 $(R,+,\cdot)$ 是一个 **环**（ring）：

1. $(R,+)$ 构成 Abel 群，其单位元记作 0，元素 $a\in R$ 在 $+$ 下的逆元记作 $-a$．
2. $(R,\cdot)$ 构成半群，即 $\cdot$ 满足结合律．
3. 分配律（distributive property）：对于所有 $a,b,c\in R$，成立 $a\cdot(b+c)=a\cdot b+a\cdot c$ 和 $(a+b)\cdot c=a\cdot c+b\cdot c$．

为表述方便，这两个二元运算 $+$ 和 $\cdot$ 常称作该环的加法和乘法，相应地，加法单位元称作 **零元**（zero），乘法单位元（如果存在）称作 **幺元**（identity）．应避免和具体的数集中的加法、乘法，以及自然数零和一产生混淆．

关于定义中是否要求乘法单位元

在有的定义中，环必须存在乘法单位元；相对地，不存在乘法单位元的则被称为 **伪环**（rng 或 pseudo-ring）. 遇到的时候需根据上下文加以判断. 维基百科采用的就是这种定义[2].

环的加法结构相当简单，但是乘法结构十分原始. 因而如果类比群，在乘法上做更多要求，可以得到如下相关定义.

幺环

对于环 $(R,+,\cdot)$，如果它含幺，即存在乘法单位元，记作 1，则称 $(R,+,\cdot)$ 是一个 **幺环**（ring with identity）.

除环

对于非零幺环 $(R,+,\cdot)$，如果对于所有非 0 元素 $a\in R$，都存在乘法逆元（记作 a^{-1}），则称 $(R,+,\cdot)$ 是一个 **除环**（division ring）.

交换环

对于环 $(R,+,\cdot)$，如果它的乘法满足交换律，则称 $(R,+,\cdot)$ 是一个 **交换环**（commutative ring）.

这里除环的定义中有趣的一点是，它将 0 视为乘法结构中的特殊元素. 这是因为 $0=0\cdot a=a\cdot 0$[3]. 也就是说，环中加法单位元乘以任何元素都得到其自身. 这样，它自然不会存在乘法逆元，除非它本身就是乘法单位元. 这样的环只有零环（见下面的例子）.

这里的启示是，理解一般的环的乘法结构时，要去除加法单位元的影响，考察 $R\setminus\{0\}$. 基于这一想法，有如下定义.

零因子

对于环 $(R,+,\cdot)$，如果存在 $b\in R$ 且 $b\neq 0$，成立 $a\cdot b=0$ 或 $b\cdot a=0$，则称非零元素 a 为一个 **零因子**（zero divisor）.

可逆元（单位）

对于环 $(R,+,\cdot)$，如果元素 a 有乘法逆元，即存在 $b\in R$，成立 $a\cdot b=b\cdot a=1$，则称元素 $a\in R$ 是一个 **可逆元**，或称 **单位**（unit）.

「单位」与「单位元」

请不要混淆这两个概念. 为避免混淆，抽象代数部分将使用「可逆元」的名称代替「单位」.

零因子不可能是可逆元，可逆元不可能是零因子. 但是，一个非零元素可以既不是零因子，也不是可逆元.

如果一个环没有零因子，就说明所有非零元素的集合在乘法运算下封闭，即 $(R\setminus\{0\},\cdot)$ 构成半群. 进一步地，如果还要求它成为交换幺半群，就可以得到整环的定义.

---

[2] Ring（mathematics）- Wikipedia[1]

[3] 该式的推导即 $0\cdot a+0=0\cdot a=(0+0)\cdot a=0\cdot a+0\cdot a$，这里，第一个等号和第二个等号是加法单位元的定义，第三个等号是分配律，最后的蕴涵关系是加法的消去律. 另一侧的乘法类似.

整环

对于非零环 $(R,+,\cdot)$，如果它是交换环，有乘法单位元，且无零因子，则称它为整环（integral domain）.

虽然整环中的元素不一定存在逆元，但是没有零因子这一特性已经足够在整环上建立消去律.

整环的消去律

设整环 R 有元素 $a,b,c\in R$ 且 $a\neq 0$，如果 $ab=ac$，则必然有 $b=c$.

对于一般的幺环，如果只考虑它的全体可逆元，那么同样可以得到群结构. 这称为环的乘法群或是单位群.

乘法群（单位群）

对于幺环 $(R,+,\cdot)$，设 $R^\times$ 为 R 中全体可逆元的集合，则 $(R^\times,\cdot)$ 构成群，称为幺环 R 的 **乘法群**（multiplicative group），或是 **单位群**（unit group）.

最简单的一些环的例子如下.

环的例子

- 零环（zero ring）：集合 $\{0\}$ 在通常意义的加法 $+$ 和乘法 $\times$ 下构成环，称为零环. 它是唯一的只有一个元素的环，也是唯一的加法单位元和乘法单位元相等的环.
- 整数环：整数集 $\mathbf{Z}$ 和其上通常定义的加法 $+$ 和乘法 $\times$ 构成了环 $(\mathbf{Z},+,\times)$. 实际上，这是一个整环，但是它不是除环.
- 多项式环：对于一个环 R，可以在上面定义 多项式环→9.16.3@p1830 $R[x]$. 如果 R 是整环，则该多项式环必然是整环.
- 四元数（quaternion）：类比复数，可以考虑集合 $\mathbf{H}=\{a+b\mathrm{i}+c\mathrm{j}+d\mathrm{k}:a,b,c,d\in\mathbf{R}\}$，并且定义其上的加法和乘法，这里，$\mathrm{i},\mathrm{j},\mathrm{k}$ 的乘法运算满足
  $$\mathrm{i}^2=\mathrm{j}^2=\mathrm{k}^2=-1,\ \mathrm{ij}=-\mathrm{ji}=\mathrm{k},\ \mathrm{jk}=-\mathrm{kj}=\mathrm{i},\ \mathrm{ki}=-\mathrm{ik}=\mathrm{j}.$$
  那么可以验证，$\mathbf{H}$ 构成环，而且，它是一个非交换的除环.
- 整数集的子集 $2\mathbf{Z}$，在通常意义的加法和乘法下构成环，它是交换环，没有零因子，但是并不含幺.
- 整数模 n 同余类 $\mathbf{Z}/n\mathbf{Z}$ 在同余类的加法和乘法运算下构成环，它是交换环，含幺（即 $\bar{1}$）. 这样的环含有零因子，当且仅当 n 是合数. 所以，当 n 是素数时，环 $(\mathbf{Z}/n\mathbf{Z},+,\times)$ 是整环；而且，此时它也是除环，所以它实际构成了一个域. 它的乘法群 $((\mathbf{Z}/n\mathbf{Z})^\times,\times)$ 就是整数模 n 乘法群.
- 矩阵环：环 R 上的全体 n 维方阵在矩阵的加法和乘法下构成一个环 $M_n(R)$. 一般地，这个环有零因子，且不是交换环.
- 对于一个集合 A 的全体子集 $\mathcal{P}(A)$，如果定义集合的对称差 $\triangle$ 和交 $\cap$ 分别为其加法和乘法运算，则 $(\mathcal{P}(A),\triangle,\cap)$ 构成环. 一般地，这个环含幺，有零因子，且是交换环.

当然，对于环的结构的讨论远不止这些，要了解更多内容，可以参考 环论→9.16.3@p1830 或相关书籍.

域

域是一个比环性质更强的代数结构. 具体地，域是交换除环. 当然也可以写出它完整的定义.

域

对于非空集合 F 和其上的两个二元运算 $+ : F \times F \to F$ 和 $\cdot : F \times F \to F$，如果它们满足以下性质，则称 $(F, +, \cdot)$ 是一个 **域**（field）：

1. $(F, +)$ 构成 Abel 群，其单位元记作 0，元素 $a \in F$ 在 $+$ 下的逆元记作 $-a$.
2. $(F \setminus \{0\}, \cdot)$ 构成 Abel 群，其单位元记作 1，元素 $a \in F \setminus \{0\}$ 在 $\cdot$ 下的逆元记作 a^{-1}.
3. 分配律：对于所有 $a, b, c \in F$，成立 $a \cdot (b + c) = a \cdot b + a \cdot c$.

换句话说，域是对加、减、乘、除四则运算都封闭的代数结构.

常见的域的例子如下.

域的例子

- 数域：有理数集 $\mathbf{Q}$，实数集 $\mathbf{R}$ 和复数集 $\mathbf{C}$ 在通常意义的加法和乘法下都构成域.
- 有限域（finite field）：以质数 p 为模的整数同余类的集合 $\mathbf{Z}/p\mathbf{Z}$ 在同余类的加法和乘法下构成域. 当然，除此之外还有其他的有限域，它们的结构由其大小唯一确定，且大小必然是质数幂的形式.
- **分式域**（fraction field）：设 $(R, +, \cdot)$ 为整环，可以考虑形如 ab^{-1} 的元素构成的集合 Q. 严格地说，在集合 $R \times (R \setminus \{0\})$ 上定义等价关系：$(a_1, b_1) \sim (a_2, b_2)$ 当且仅当 $a_1 b_2 = a_2 b_1$. 那么，集合 Q 就是这一关系下的等价类构成的集合 $R \times (R \setminus \{0\}) / \sim$，其中，$(a, b)$ 所在等价类就记作 ab^{-1}. 如果定义它上面的运算为
$$a_1 b_1^{-1} + a_2 b_2^{-1} = (a_1 \cdot b_2 + a_2 \cdot b_1)(b_1 \cdot b_2)^{-1},$$
$$(a_1 b_1^{-1}) \cdot (a_2 b_2^{-1}) = (a_1 \cdot a_2)(b_1 \cdot b_2)^{-1}$$
则 $(Q, +, \cdot)$ 构成域，称为 R 的分式域. 例如，有理数域 $(\mathbf{Q}, +, \times)$ 就是整数环 $(\mathbf{Z}, +, \times)$ 的分式域.
- 二次域（quadratic field）：它是在有理数域 $\mathbf{Q}$ 中添加了 $\sqrt{d}$ 而扩张成的，这里 $d \neq 0, 1$ 且没有平方因子. 相关内容可以参考 二次域→9.11.32@p1495.

域相较于环，拥有着非常简单的加法和乘法结构. 所以，域本身的结构往往很简单. 这使得域的研究和环的研究大不相同，通常会转而研究域的扩张，以及相应的 Galois 理论. 在算法竞赛中，有时会需要在有理数域或者有限域的扩域上进行计算. 域论的相关内容，可以参考 域论→9.16.4@p1849 或相关书籍.

应用

最后，以下面的题目为例，说明抽象的代数对象是怎样辅助分析具体的问题的.

【模板】"动态 DP"& 动态树分治（加强版） [2]

给定大小为 n 的带点权的树，进行 m 次点权修改. 每次修改后要输出树上最大带权独立集的权值之和. 问题强制在线.

思路分析

这道题是动态 DP 模板，一种复杂度正确的代码实现需要用到 全局平衡二叉树→10.20.2@p2498，具体样例代码也在对应页面. 这里仅仅结合该题情景，分析建模的过程.

为了突出重点，这里暂不考虑全局平衡二叉树对于树形结构的处理，转而考虑链上的最大带权独立集的 DP 问题. 顺次考虑链 $[1, n]$ 上的每个点，对于点 i 可以选（1）或不选（0）. 分别设这两种情形下，$[1, i]$ 上子问题的最优解为 $f_{i,1}$ 和 $f_{i,0}$. 所以，可以写出 DP 方程为

$$f_{i,1} = w_i + f_{i-1,0},$$
$$f_{i,0} = \max\{f_{i-1,1}, f_{i-1,0}\}.$$

它的初值为 $(f_{0,1}, f_{0,0}) = (0, 0)$，而最终答案就是 $\max\{f_{n,1}, f_{n,0}\}$．要表示点 i 对于最终结果的影响，只需要注意到这一递归关系可以写作

$$(f_{i,1}, f_{i,0}) = g(f_{i-1,1}, f_{i-1,0}; w_i).$$

这是一连串 $\mathbf{R}^2$ 到 $\mathbf{R}^2$ 的映射，它将 $(f_{i-1,1}, f_{i-1,0})$ 映射到 $(f_{i,1}, f_{i,0})$，用群的语言描述，这些变换在映射的复合之下构成幺半群．这正是线段树可以维护的．

但是，这样的含参变换 $g(\cdot; w_i)$ 如果没有特殊的结构，一般的 $\mathbf{R}^2$ 到 $\mathbf{R}^2$ 的映射是不可能用有限维的数据描述的．这里就需要另一项观察，即如果在 $\mathbf{R} \cup \{-\infty\}$ 上，定义 max 作为加法、$+$ 作为乘法，那么 $\mathbf{R} \cup \{-\infty\}$ 构成一种类似环的结构，这里，$-\infty$ 是加法单位元，0 是乘法单位元．但是它不是环，因为其中的元素并非都有加法逆元．这样的结构叫做半环[4]，这里 $(\mathbf{R} \cup \{-\infty\}, \max, +)$ 形成的半环叫做 **热带半环**（tropical semiring）．

基于热带半环 $(R, \oplus, \otimes)$，可以定义它上面的矩阵乘法．即对于 $m \times n$ 维矩阵 $A = (a_{ij})$ 和 $n \times p$ 维矩阵 $B = (b_{jk})$，可以定义其乘积 AB 为 (c_{ik})，它的每项元素等于

$$c_{ik} = \bigoplus_{j=1}^{n} (a_{ij} \otimes b_{jk}) = \max_{1 \le j \le n} (a_{ij} + b_{jk}).$$

有了这些记号，可以将上述递推关系看作是热带半环上的线性变换，并用矩阵语言写作

$$\begin{pmatrix} f_{i,1} \\ f_{i,0} \end{pmatrix} = \begin{pmatrix} -\infty & w_i \\ 0 & 0 \end{pmatrix} \begin{pmatrix} f_{i-1,1} \\ f_{i-1,0} \end{pmatrix}.$$

由此，只要用线段树维护这一热带半环上的矩阵的乘积就可以回答多次修改的链上的动态 DP 问题．

现在回到该问题的树上版本．对于树上的节点 i，其子节点集合记作 $S(i)$，则该处的 DP 方程为

$$f_{i,1} = w_i + \sum_{j \in S(i)} f_{j,0},$$
$$f_{i,0} = \sum_{j \in S(i)} \max\{f_{j,1}, f_{j,0}\}.$$

首先，通过树链剖分将问题转化为链上版本．设 h 是 i 的重子节点，那么上述递推方程可以写作

$$f_{i,1} = w_i + f_{h,0} + g_{i,1},$$
$$f_{i,0} = \max\{f_{h,0}, f_{h,1}\} + g_{i,0},$$

这里，

$$g_{i,1} = \sum_{j \in S(i),\, j \ne h} f_{j,0},$$
$$g_{i,0} = \sum_{j \in S(i),\, j \ne h} \max\{f_{j,1}, f_{j,0}\}$$

总结了轻子节点的贡献．根据上文描述，这些变换都可以写作热带半环上的矩阵形式，所以，整个问题也就可以在树剖后的线段树上维护．但是，直接用树剖加线段树的单次修改是 $O(\log^2 n)$ 的，所以需要用到上文提到的全局平衡二叉树优化到 $O(\log n)$，当然也可以用 LCT 维护．

这里提到的热带半环以及上面的矩阵运算其实并不罕见．如果将上文中的 max 换作 min，则相应的热带半环常用于最短路问题中．如果 n 维方阵 A 给出了顶点数目为 n 的某个图的两点间的（最短）边权，那么，A^k 的 (i, j) 处的元素就是自点 i 经至多 k 条边到点 j 的最短距离；特别地，A^n 就是

[4] 半环（semiring）是在幺环的定义中放松了加法运算一定存在逆元的要求，即加法结构是交换幺半群、乘法结构是幺半群的代数结构．更多信息参见 Wikipedia[3].

该图的距离矩阵. 当然实际实现的时候并不会真的暴力计算这一矩阵的幂, 而是使用复杂度为 $O(n^3)$ 的 Floyd 算法.

参考资料与注释

- Dummit, D.S. and Foote, R.M. (2004) Abstract Algebra. 3rd Edition, John Wiley & Sons, Inc.
- Tropical semiring - Wikipedia[4]

[1] Ring（mathematics） - Wikipedia

[2] 【模板】"动态 DP"& 动态树分治（加强版）

[3] Wikipedia

[4] Tropical semiring - Wikipedia

9.16.2 群论

Authors: jifbt, billchenchina, Enter-tainer, Great-designer, iamtwz, ImpleLee, isdanni, Menci, ouuan, Tiphereth-A, warzone-oier, Xeonacid, c-forrest, cervoliu

前置知识：抽象代数基本概念→9.16.1@p1805、置换与排列→9.8@p1138

引入

群论（group theory）主要研究群这个 代数结构[1].

为了研究群的结构，需要掌握一些基本工具，这包括子群、群同态和群作用. 算法竞赛中，主要涉及到的群是数论相关的群（比如整数模 n 乘法群 $(\mathbf{Z}/n\mathbf{Z})^\times$）以及置换群，本文将着重介绍相关的概念. 本文未涉及的群论部分，比如有限群的结构理论和群的线性表示理论，有兴趣的读者应当参考专业书籍.

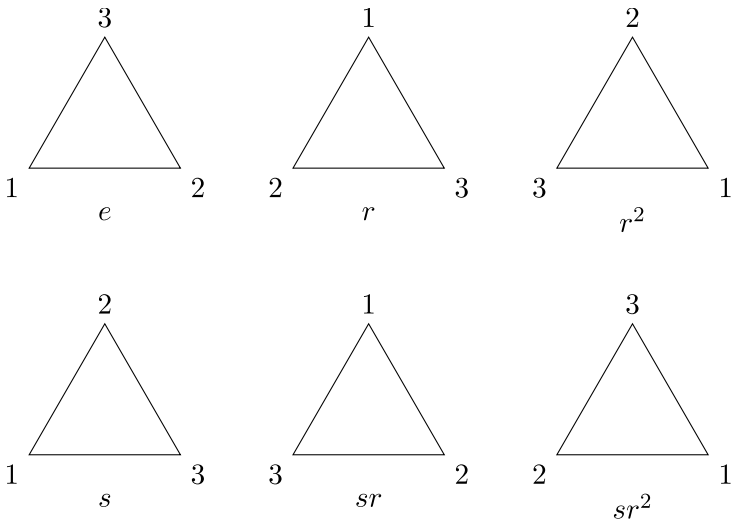
记号

在不引起歧义时，本文可能会将 $g \cdot h$ 写作 gh，也可能会将群 $(G, \cdot)$ 写作群 G.

理解抽象代数不能够离开实例. 作为理解下文概念的例子，这里讨论正三角形的空间对称群 D_6.

例子：正三角形的空间对称群 D_6

如图所示，对于给定正三角形，共计有六种不同的操作可以使得它与自身重合.



这里，使用 r 表示顺时针旋转，使用 s 表示沿顶点 1 到三角形中心的连线翻转，操作自右向左复合，即 sr 表示先旋转（r）再翻转（s）．两个操作不同，当且仅当某个三角形的顶点在两个操作之后所处位置不同．

记号	操作	阶	置换表示
e	恒等变换，即什么也不做	1	(1)
r	顺时针旋转 120°	3	(123)
r^2	顺时针旋转 240°	3	(132)
s	沿 1 到中心连线翻转	2	(23)
sr	先旋转 120°，再翻转	2	(13)
sr^2	先旋转 240°，再翻转	2	(12)

容易验证，这些操作确实构成了群．比如说，该群的单位元是 e，而 sr 的逆元是它自身．而且，群 D_6 并不是交换群，比如可以直接验证 $rs = sr^{-1}$．

表中记录的操作并不是该记号对应的唯一的对称操作．比如，「沿 2 到中心的连线翻转」也是三角形的对称操作，它不在表中，但它的结果和操作「先旋转 120°，再翻转」相同．表中的「阶」和「置换表示」等概念下文会给予说明．

子群

要理解给定群的结构，可以首先分析其子结构．群的子结构就是那些在同一运算下仍然成为一个群的该群的子集．由此，有如下定义．

自 子群

对于群 $(G, \cdot)$ 和它的一个子集 $H \subseteq G$，如果 $(H, \cdot)$ 也是一个群，则称子集 H 是 G 的一个 **子群**（subgroup），记作 $H \leq G$．

例子：正三角形的空间对称群 D_6（续）

在 D_6 中，容易验证它的子群有 $\{e\}$，$\{e, s\}$，$\{e, sr\}$，$\{e, sr^2\}$，$\{e, r, r^2\}$ 和 D_6 本身，共计六个．除群 D_6 外，这些子群的结构都更为简单，而且蕴含了关于原来群的部分信息．

要判断给定子集 $H \subseteq G$ 是不是子群，并不需要逐一验证群的定义：结合律自然成立；子集成为子群，只要保证它对二元运算封闭、有单位元且对取逆封闭就好了．事实上，这些条件可以总结在一起．

定理（子群判别法）

群 G 的非空子集 H 是子群，当且仅当，对于所有元素 $g, h \in H$ 都有 $g^{-1}h \in H$．

由子集生成的子群

一般地，给定群 G 中的子集 S，从 S 中的元素出发，重复进行乘法和取逆运算有限次，能够得到的所有结果的集合构成 G 的一个子群．这称为由子集 S 生成的子群．

自 由子集生成的子群

对于群 G 和它的非空子集 $S \subseteq G$，如果 H 是包含 S 的 G 的子群中（依包含关系）最小的，则子群 H 称为 **由子集 S 生成的子群**（subgroup generated by a subset），并记作 $\langle S \rangle$．特别地，如果 $S = \{x\}$ 是单元素集合，则 $\langle S \rangle$ 也记作 $\langle x \rangle$，称为 x 的幂的循环子群（cyclic subgroup of the powers of an element）．

可以证明，任给定这样的子集 S，总能找到这样的子群：$\langle S\rangle$ 可以构造为所有包含 S 的 G 的子群的交.

例子：正三角形的空间对称群 D_6（续）

在群 D_6 中选定旋转操作 r，重复应用它和它的逆操作，就得到子群 $\{e, r, r^2\}$. 它可以记作 $\langle r\rangle$. 群 D_6 中的所有非平凡子群都可以通过选定某个操作来生成.

有些子集生成的子群是群本身. 这样的子集尤为特殊，称为群的生成子集.

群的生成子集

如果群 $(G, \cdot)$ 的子集 $S \subseteq G$ 满足 $\langle S\rangle = G$，则称 S 是 G 的 **生成子集**（generating set of a group）. 生成子集 S 中的元素称为 **生成元**（generator）.

群是自身平凡的生成子集. 更为有趣的情形是，生成子集远小于群本身的规模.

例子：正三角形的空间对称群 D_6（续）

可以验证，$D_6 = \langle s, r\rangle$，这就是说，任何正三角形的对称操作都可以通过旋转和翻转的复合得到.

循环群

仅由一个元素生成的群的结构非常简单. 这样的群称为循环群.

循环群

对于群 G，如果存在 $x \in G$，成立 $G = \langle x\rangle$，则称 G 是一个 **循环群**（cyclic group）.

例子：正三角形的空间对称群 D_6（续）

所有 D_6 的非平凡子群以及 $\{e\}$ 都是循环群.

可以证明，循环群的结构由其大小唯一确定. 如果循环群无限，则它必然和整数的加法群 $(\mathbf{Z}, +)$ 具有相同的群结构，记作 C_∞ 或 $\mathbf{Z}$；否则，记群的元素个数为 $n \in \mathbf{N}_+$，则它必然和整数模 n 的同余类的加法群 $(\mathbf{Z}/n\mathbf{Z}, +)$ 具有相同的群结构，记作 C_n 或 $\mathbf{Z}_n$. 这一结论的严格叙述需要用到下文的 群同构→9.16@p1805 的概念，它严格描述了两个群结构相同这一事实.

循环群分类定理

大小为 n 的有限循环群 G 同构于 C_n；无限循环群 G 同构于 C_∞.

证明

给定循环群 $G = \langle x\rangle$，它总可以写作 $G = \{x^n : n \in \mathbf{Z}\}$. 如果群 G 有限，那么必然存在自然数 $n < m$ 满足 $x^n = x^m$，依消去律，可以得到 $x^{m-n} = e$. 此时，不妨取最小的正整数 $n \in \mathbf{N}_+$ 使得 $x^n = e$，那么列 $\{x^k\}$ 将会是长度为 n 的循环，且循环节内元素各不相同（否则违反 n 的最小性）. 此时，映射 $x^k \mapsto \bar{k}$ 就提供了同构映射 $G \to \mathbf{Z}/n\mathbf{Z}$，亦即 $G \cong C_n$. 反之，如果群 G 无限，那么群 G 内元素各不相同，映射 $x^k \mapsto k$ 就提供了同构映射 $G \to \mathbf{Z}$，亦即 $G \cong C_\infty$.

所有循环群都是 Abel 群. 本文的例子 D_6 说明，即使群的所有非平凡子群都是循环群，群本身也可能不是 Abel 群.

阶

群的阶就是群的元素的个数. 群内给定元素的阶就是该元素生成的循环子群的阶. 由此，有如下定义：

群的阶

群 G 的 **阶**（order）是它的元素个数，记作 $|G|$．无限群的阶也是无限．

元素的阶

群 G 中元素 $x \in G$ 的 **阶**（order）是最小的正整数 n 使得 $x^n = e$ 成立，记作 $|x|$；如果这样的 n 不存在，则称元素 x 的阶是无限，记作 $|x| = \infty$．

元素的阶总是不大于群的阶，事实上，下文即证明，元素的阶总是整除群的阶．但是，群的阶并非总是元素的阶的最大值，比如 D_6 是六阶群，但是元素的阶最大是 3．群的阶也并不是所有元素的阶的最小公倍数，比如 Klein 四元群[5] V_4 的阶是 4，但是里面只有 1 阶元和 2 阶元．

定理

有限循环群 $C_n = \langle x \rangle$ 中，元素 x^k 的阶是

$$\frac{n}{\gcd(k, n)}.$$

特别地，C_n 的生成元的数目是 $\varphi(n)$，这里，$\varphi(\cdot)$ 是 欧拉函数→9.11.5@p1196．

作为上述讨论的应用，注意到模 n 整数乘法群的阶是 $\varphi(n)$，群中的任何元素 a 的阶都整除它，故而必然有 $a^{\varphi(n)} = 1$．这就是 欧拉定理→9.11.9@p1223，因为该群中的元素就是所有与 n 互质的元素．

陪集

子群以外的元素在群中的结构也并非杂乱无章．

例子：正三角形的空间对称群 D_6（续）

观察 D_6 中的子群 $\langle r \rangle$，剩余的元素 $\{s, sr, sr^2\}$ 结构和 $\langle r \rangle$ 十分相似：它当中的每个操作都可以通过 s 和 $\langle r \rangle$ 中的操作复合得到．同理，考虑子群 $\langle s \rangle$，则群中的剩余元素可以分为两类，$\{r, sr\}$ 和 $\{r^2, sr^2\}$，它们可以通过 $\langle s \rangle$ 中的元素分别与 r 和 r^2 复合得到．这样的现象是普遍的．

给定子群，可以定义它的陪集．

陪集

设 G 是群，$H \leq G$ 是它的子群，则子群 H 的包含 g 的 **左陪集**（left coset）和 **右陪集**（right coset）分别定义为集合

$$gH = \{gh : h \in H\},$$
$$Hg = \{hg : h \in H\}.$$

陪集中的元素称为陪集的代表元（representative element）．

子群本身也是其陪集．给定子群，全体陪集构成群的一个分划，即群是全体陪集的不交并．分划总可以看作某个等价关系的等价类．对于左陪集的分划，这个等价关系就是 $g_1 \sim g_2$ 当且仅当 $g_1^{-1} g_2 \in H$；对于右陪集的分划，这个等价关系就是 $g_1 \sim g_2$ 当且仅当 $g_1 g_2^{-1} \in H$．

[5] 这个群可以表示为置换群 $\{(1), (12)(34), (13)(24), (14)(23)\}$，也可以写作 $C_2 \times C_2$．

例子：正三角形的空间对称群 D_6（续）

按照陪集的语言，上面的例子中，D_6 可以分别划分成 $\langle r\rangle \cup s\langle r\rangle$ 和 $\langle s\rangle \cup \langle s\rangle r \cup \langle s\rangle r^2$．这里，前者将群划分为若干左陪集，后者将群划分为若干右陪集．应该注意，代表元的选取并无特殊，比如可以验证 $s\langle r\rangle = sr\langle r\rangle$．陪集中的任何元素都是该陪集的代表元．

同一子群的不同陪集大小都相同，都等于对应子群的大小．由于给定子群的全体陪集构成群的一个分划，有限群的阶必然是子群的阶的整数倍．这叫做 Lagrange 定理．

Lagrange 定理

对于有限群 G 和它的子群 $H \leq G$，成立 $|\ G\ | = [G : H]|\ H\ |$，这里，$[G : H]$ 表示 G 中子群 H 的左（右）陪集数，称为群 G 中子群 H 的 **指数**（index）．

证明

考察左乘以 g 的映射 $h \mapsto gh$，则它和映射 $h \mapsto g^{-1}h$ 互为逆映射，因而它们都是双射．这说明，$|\ H\ | = |\ gH\ |$ 总是成立．

注意到，元素的阶就是元素生成的循环子群的阶，所以元素的阶也必然整除群的阶．

正规子群

一般情况下，给定子群的左右陪集并不相同．

例子：正三角形的空间对称群 D_6（续）

群 D_6 中，$\langle s\rangle r = \{r, sr\}$，但是 $r\langle s\rangle = \{r, sr^2\}$．但是如果考虑子群 $\langle r\rangle$，那么左右陪集又总是相同的，因为此时群只有两个陪集，而子群作为一个陪集又必然重合．

左右陪集是否相同，反映了相应的子群的性质．

正规子群

设 $N \leq G$ 是群 G 的子群，如果对所有 $h \in N$ 和 $g \in G$，都成立 $ghg^{-1} \in N$，换言之，对所有 $g \in G$，都成立 $gNg^{-1} \subseteq N$，则称 N 是 G 的一个 **正规子群**（normal subgroup），记作 $N \trianglelefteq G$．

这个定义中的条件正等价于 $gN = Ng$ 永远成立．群 G 总有平凡的正规子群，即 $\langle e\rangle$ 和 G 自身．

例子：正三角形的空间对称群 D_6（续）

在群 D_6 中，$\langle s\rangle$ 不是正规子群，而 $\langle r\rangle$ 是正规子群．

商群

正规子群是非常重要的一类子群，原因之一就是基于正规子群可以定义商群．

对于群 G 和它的正规子群 $N \trianglelefteq G$，考虑全体陪集的集合

$$G/N = \{gN : g \in G\}.$$

此时，左右陪集相同，不必区分．可以从群 G 的运算出发，定义 G/N 上的二元运算 $\circ$，它满足

$$g_1N \circ g_2N = (g_1g_2)N.$$

[6]对于一般的子群 $H \leq G$，也可以在全体左（右）陪集上尝试定义类似的运算．但是，这样的运算是良定义的，当且仅当 H 是 G 的正规子群．

可以证明，运算的结果与代表元的选取无关[6]．此时，$(G/N, \circ)$ 确实具有群的结构，它称为 G 模 N 的**商群**（quotient group）．商群 G/N 并不是群 G 的子群，它的每个元素都是群 G 的一个子集．

例子：正三角形的空间对称群 D_6（续）

在群 D_6 中，商群 $G/\langle r\rangle$ 的意义非常显然．它相当于在所有这些对称操作中，忽视操作中将三角形旋转的角度，而只关注它是否将三角形翻转．两个将三角形翻转的操作的复合相当于没有翻转原三角形；但是，如果两个操作一个翻转了三角形而另一个没有，那么复合也必然翻转了三角形．翻转与否也具有群的结构．从群 D_6 中忽视旋转的细节而只考虑翻转的有无，在代数上就是将复杂的群 D_6 模掉 $\langle r\rangle$ 而得到商群 $D_6/\langle r\rangle$．这些论述对子群 $\langle s\rangle$ 是无效的，因为如果忽视翻转的有无，那么旋转的度数就没有办法明确判断，这也是为什么 $G/\langle s\rangle$ 不具有商群的结构．

群的商群可以将复杂的群简化，允许观察群的部分结构来了解原来群的结构．这也是商群也称为**因子群**（factor group）的原因．除平凡正规子群外，再不含其他正规子群的群称为**单群**（simple group），这些群没有办法简化为更小的群．如同素数一样，它们是组成更复杂的群结构的基石．

群同态

理解给定群结构的第二种方法，是将两个群的结构相互比较．

对于两个群，要比较它们的结构，就是要构造两个群之间的映射．但是，这样的映射并不能是任意的，它们要保持群的结构，也就是要保持群的运算在映射前后一致．这样的映射称为群的同态．

群同态

设映射 $\varphi : G \to H$ 是自群 $(G, \cdot)$ 到群 $(H, \odot)$ 的映射，如果 φ 保持群的运算，即对所有 $g_1, g_2 \in G$ 都成立 $\varphi(g_1 \cdot g_2) = \varphi(g_1) \odot \varphi(g_2)$，则称映射 φ 是一个自群 G 到群 H 的**同态**（homomorphism）．

群同态必然将单位元映射到单位元，也必然将逆元映射到逆元．

记号

在下文中，如果不引起歧义，不会区分群 G 和 H 中的运算的记号，并且为表述简便，将省略这些记号．

群同构

对于自群 G 到群 H 的同态 $\varphi : G \to H$，一个自然的问题是，这一同态在多大程度上反映了群 G 和群 H 的结构是一致的．为此，考察群 G 在同态 φ 下的像 $\varphi(G)$，它将群 G 的结构映射到群 H 中．一方面，$\varphi(G)$ 是群 H 的一个子群；但若同态 φ 不是满射，$\varphi(G)$ 与群 H 并不相同．另一方面，同态 φ 也未必是单射；如果 φ 不是单射，那么 $\varphi(G)$ 只反映了群 G 的部分结构．只有当同态 φ 是双射时，群 G 和群 H 的结构才是完全一致的．这种特殊的群同态叫做群同构．

群同构

设 $\varphi : G \to H$ 是自群 G 到群 H 的同态，如果 φ 是双射，则称 φ 是群 G 和群 H 之间的**同构**（isomorphism），记作 $G \cong H$．

同构的两个群结构完全一致．如果只关心群的结构，两个同构的群没有必要区分．前文关于循环群的分类定理就是在同构的意义下给出的．

例子：正三角形的空间对称群 D_6（续）

回到前文的例子，正三角形的每个对称操作都唯一对应了顶点集合上的置换操作. 顶点集合上的全体置换也构成群，即 S_3. 容易验证，这样得到的映射 $\varphi : D_6 \to S_3$ 是群同态；进一步地，它也是群同构. 所以，$D_6 \cong S_3$. 事实上，六阶群要么同构于 C_6，要么同构于 S_3（证明见下文）.

对给定阶的有限群的结构进行分类，是群论的重要研究内容，但超出了本文的范畴.

同态的核

对于一般的同态，可以进一步讨论有多少关于群的结构的信息损失在了同态中. 延续上文的记号. 已知 $\varphi(G)$ 是群 H 的子群，现在问题的关键在于 $\varphi(G)$ 和群 G 之间的关系.

例子：正三角形的空间对称群 D_6（续）

考虑如下定义的映射 $\varphi : D_6 \to C_2 = \langle x \rangle$：

$$\varphi(e) = \varphi(r) = \varphi(r^2) = e,\ \varphi(s) = \varphi(sr) = \varphi(sr^2) = x.$$

容易验证，φ 是群同态；它是满射，但不是单射. 它的意义很明显，就是将群的每一个对称操作映射到其翻转的有无. 这样的同态压缩的群 D_6 中的信息，就是有关它旋转角度的信息. 比如说，如果没有翻转，任何角度的旋转都映射到了群 C_2 中的单位元.

这个例子启发使用 $\{e\}$ 的原像衡量同态中损失的结构信息. 为此，有如下定义.

同态的核

设 $\varphi : G \to H$ 是自群 G 到群 H 的同态，则同态 φ 的 **核**（kernel）是 $\ker \varphi = \{g \in G : \varphi(g) = e\}$，这里，$e$ 是 H 的单位元.

同态基本定理

同态的核 $\ker \varphi$ 的确刻画了群同态中损失的结构信息. 这一结论的精确表述就是 **同态基本定理**（亦称 **第一同构定理**）（fundamental theorem of group homomorphism, a.k.a., first isomorphism theorem）.

同态基本定理（第一同构定理）

设 $\varphi : G \to H$ 是自群 G 到群 H 的同态，则 $\ker \varphi \trianglelefteq G$，且 $G/\ker \varphi \cong \varphi(G) \leq H$.

证明

首先，$N = \ker \varphi$ 是正规子群，因为对于任意 $h \in N$ 都有 $\varphi(ghg^{-1}) = \varphi(g)\varphi(h)\varphi(g)^{-1} = \varphi(g)\varphi(g)^{-1} = e$，亦即 $ghg^{-1} \in \ker \varphi$. 然后，考察映射 $\Phi : G/N \to \varphi(G)$，它满足 $\Phi(gN) = \varphi(g)$. 映射是良定义的，因为如果 $g_1 N = g_2 N$，那么 $g_1^{-1} g_2 \in N$，则 $\varphi(g_1^{-1} g_2) = e$，即 $\varphi(g_1) = \varphi(g_2)$. 映射 Φ 显然是满射；它也是单射，因为 $\ker \Phi = \{gN : \varphi(g) = e\} = \{N\}$. 故而，$\Phi$ 是群同构. 最后，$\varphi(g_1)\varphi(g_2)^{-1} = \varphi(g_1 g_2^{-1}) \in \varphi(G)$，根据子群判别法，$\varphi(G)$ 必然是子群.

推论

同态 $\varphi : G \to H$ 是单射，当且仅当 $\ker \varphi = \{e\}$. 此时，G 同构于 H 的一个子群，即 $G \cong \varphi(G) \leq H$.

也就是说，同态 φ 的核是群 G 的正规子群，而模 $\ker \varphi$ 得到的商群 $G/\ker \varphi$ 同构于同态的像 $\varphi(G)$，而这一同态的像正是群 H 的子群.

例子：正三角形的空间对称群 D_6（续）

上文给出的群同态 $\varphi : D_6 \to C_2$ 的核是 $\langle r \rangle$，前文讨论正规子群时也已经说明 $D_6/\langle r \rangle$ 的确同构于 C_2.

自然同态

得到这样的结论并不为奇．这是因为在构造同态 $\varphi : D_6 \to C_2$ 时，利用的正是商群 $D_6/\langle r \rangle$ 的几何意义．这样的现象并不罕见．事实上，对每个商群，都可以构造出群同态，使得同态的像同构于给定的商群．

自然同态

对于群 G 和其正规子群 $N \trianglelefteq G$，由 $\pi(g) = gN$ 给出的映射 $\pi : G \to G/N$ 是自 G 到 G/N 的满同态，称为自群 G 到商群 G/N 的 **自然同态**（natural homomorphism）或 **自然映射**.

这一结论也说明，对于任何给定群的正规子群，都能够找到对应的群同态，使得这一同态的核就是给定的正规子群．前文同态基本定理则说明，任何同态的核都是正规子群．故而，正规子群和同态的核是一体两面.

利用自然映射的概念，群的同态基本定理其实是在说如下的 **交换图**（commutative diagram）成立.

$$\begin{array}{ccc} G & \xrightarrow{\quad\varphi\quad} & H \\ \Big\downarrow{\scriptstyle \pi} & & \Big\uparrow{\scriptstyle \iota} \\ G/N & \xrightarrow{\quad\Phi\quad} & \varphi(G) \end{array}$$

这里，所有箭头都是群同态，且 $N = \ker \varphi$ 是同态 φ 的核，$\varphi(G)$ 是同态 φ 的像．这些映射依次是，$\pi : g \mapsto gN$ 为群 G 到商群 G/N 的自然映射（满同态），$\Phi : gN \mapsto \varphi(g)$ 是同构映射，ι 则是嵌入映射（单同态）．交换图意味着，图中从 G 出发到 H 结束的两条不同路径上的映射的复合得到的结果是一致的，即 $\varphi = \iota \circ \Phi \circ \pi$．交换图清晰地说明，同态 φ 损失的信息就反映在 π 和 ι 中.

群的同构定理

用于理解群的结构的有力工具，是群的同构定理．前文已经给出第一同构定理．为了内容完整，这里再给出其他常见的同构定理.

第二同构定理涉及到子群的乘积的概念.

子集的乘积

对于群 G 和它的子集 $A, B \subseteq G$，子集 A 和 B 的 **乘积**（product）是子集 $AB = \{ab : a \in A, b \in B\}$.

子群的乘积并不总是子群．比如，群 D_6 的子群 $A = \langle s \rangle$ 和子群 $B = \langle sr \rangle$ 的乘积等于 $AB = \{e, s, r, sr\}$，这并不是 G 的子群，因为 $(sr)s = r^2 \notin AB$．其实，这种 $a \in A$，$b \in B$，但 $ba \notin AB$ 形式的反例正是乘积不是子群的根本原因．对此，有如下定理.

定理

对于群 G 和它的子群 $A, B \leq G$，乘积 AB 是子群，当且仅当 $AB = BA$.

证明

乘积 AB 是子群，则必然有 $ba=(a^{-1}b^{-1})^{-1}\in AB$ 对任意 $a\in A$ 和 $b\in B$ 都成立，所以 $BA\subseteq AB$. 反过来，如果 $AB=BA$，则对于任意 $a_1,a_2\in A$ 和 $b_1,b_2\in B$，都有 $(a_1b_1)(a_2b_2)^{-1}=a_1b_1b_2^{-1}a_2^{-1}\in a_1BA=a_1AB=AB$，则根据子群判别法，必然有 AB 是子群.

第二同构定理（second isomorphism theorem, a.k.a., diamond isomorphism theorem）则给出了子群乘积仍是子群的更为简单的充分条件，并且进一步确定了其结构.

第二同构定理

设群 G 和子群 $A,B\leq G$ 满足 $A\leq N_G(B)$，那么，$AB\leq G$，且 $B\trianglelefteq AB$，$A\cap B\trianglelefteq A$，$AB/B\cong A/(A\cap B)$. 这里，$N_G(B)$ 是 B 的 正规化子→9.16@p1805. 特别地，$A\leq N_G(B)$ 的一个充分条件是 $B\trianglelefteq G$.

证明

因为 $A\leq N_G(B)$，必然有 $aBa^{-1}=B$ 对于所有 $a\in A$ 都成立，此即 $aB=Ba$. 因此，必然有 $AB=BA$，则由上述定理知 AB 是子群. 子群 B 作为 AB 的子群，左右陪集相同，因而 $B\trianglelefteq AB$.

考察映射 $\varphi:A\to AB/B$ 满足 $\varphi(a)=aB$，则它是满射，且它的核 $\ker\varphi=\{a\in A:aB=B\}=A\cap B$. 应用同态基本定理就可得证.

第三同构定理（third isomorphism theorem）则给出了商群的正规子群和商群与原来群的正规子群和商群之间的对应关系. 它解释了将商群进一步分解这一想法的合理性.

第三同构定理

设群 G 有正规子群 $H,K\trianglelefteq G$，且 $H\leq K$，则 $K/H\trianglelefteq G/H$，且 $(G/H)/(K/H)\cong G/K$.

证明

考察映射 $\varphi:G/H\to G/K$ 满足 $\varphi(gH)=gK$，则它是满的群同态，且 $\ker\varphi=\{gH:g\in K\}=K/H$. 应用同态基本定理就可得证.

这一结论可以推广到第四同构定理，或称 **对应定理**（correspondence theorem），它进一步给出了群的子群格和商群的子群格之间的对应关系.

对应定理

设群 G 有正规子群 $N\trianglelefteq G$，则全体包含 N 的群 G 的子群 $\mathcal{H}=\{H:N\subseteq H\subseteq G\}$ 和商群 G/N 的全体子群 $\mathcal{S}=\{S:S\leq G/N\}$ 之间存在双射 $\varphi:\mathcal{H}\to\mathcal{S}$，它将 $H\in\mathcal{H}$ 映射至 $H/N\in\mathcal{S}$. 这个双射保持子群的包含关系，且 G 的正规子群总是映射到 G/N 的正规子群.

关于同构定理的内容

不同的教材中，群的同构定理的内容和名称可能有所差异. 这里选取的是常见的一个版本. 维基百科[2] 总结了常见教材中同构定理内容和名称的差异.

群作用

理解给定群结构的第三种方法，是考察群在集合上的作用.

比如说，本文考察的正三角形的空间对称群就是通过群的元素（即对称操作）在三角形上的作用来定义的．再比如说，对称群 S_M 的定义可以通过它的元素在集合 M 上的作用给出．这里所谓的作用，指的是每个群的元素都对应一个集合上的置换．

自 群在集合上的作用

对于群 G 和集合 X 以及映射 $G \times X \to X$，记 (g, x) 在该映射下的像为 $g \cdot x$，如果该映射对所有 $g_1, g_2 \in G$ 和 $x \in X$ 都满足条件 $g_1 \cdot (g_2 \cdot x) = (g_1 g_2) \cdot x$ 和 $e \cdot x = x$，那么就称该映射为群 G 在集合 X 上的 **群作用**（group action）．

「左作用」和「右作用」

这里采取的群作用的定义在有些地方[7]会称为左作用（left action），因为在记号 $g \cdot x$ 中，群的元素写在了集合的元素的左侧．相应地，此时群元素的复合 $g_1 g_2$ 作用在集合上时，需要先进行 g_2 的作用，再进行 g_1 的作用．当然，也可以相应地定义右作用，记作 $x \cdot g$．此时，群元素的复合顺序与左作用相反，即 $x \cdot (g_1 g_2) = (x \cdot g_1) \cdot g_2$．两者只有记号上的区别，而无本质区别，因而本文默认采用左作用的记号．

对于满足上述定义的群作用，自然有如下构造

$$\begin{aligned} \varphi : G &\to S_X \\ g &\mapsto \varphi_g : X \to X \\ x &\mapsto g \cdot x. \end{aligned}$$

这一映射，将每个群 G 中的元素 g 都对应到集合 X 上的一个置换 φ_g，且置换 φ_g 将元素 x 映射到 $g \cdot x$．

根据定义，群中的单位元 e 对应的双射 φ_e 是 X 上的恒等映射，而群中元素 g 对应的映射 φ_g 和其逆元 g^{-1} 对应的映射 $\varphi_{g^{-1}}$ 互为逆映射（这也说明为什么 φ_g 总是双射）．可以验证，$\varphi_{g_1 g_2} = \varphi_{g_1} \varphi_{g_2}$，即 φ 是群 G 到群 S_X 的群同态．

这一群同态 φ 称为该群作用的 **置换表示**（permutation representation），它将群 G 映射到了某个置换群上．

自 置换群

如果群 G 是某个对称群的子群，则称群 G 是一个 **置换群**（permutation group）．

该群同态的核也称为该群作用的核．如果这一群同态的核是平凡的，即这个同态是单射，则称该群作用是 **忠实的**（faithful），即该作用的置换表示忠实地反映了群结构的信息．此时，群 G 与置换表示得到的置换群同构．

记号

下文中，为表述方便，将省略群作用中的 $\cdot$ 记号．

轨道

群作用是二元映射．固定群中的元素 g，则可以得到集合上的置换 φ_g．而如果固定集合上的元素 x，则可以得到群对该元素作用的所有可能的结果．

[7] 比如 Group action - Wikipedia[3].

轨道

对于群 G 在集合 X 上的作用和 $x \in X$，称 x 在群 G 作用下的 **轨道**（orbit）是子集 $Gx = \{gx : g \in G\}$.

例子：正三角形的空间对称群 D_6（续）

比如说，如果考虑群 $\langle s \rangle \leq D_6$ 在正三角形顶点集合上的作用，则顶点 1 的轨道是 $\{1\}$，而顶点 2 和 3 的轨道是 $\{2, 3\}$. 但是，群 $\langle r \rangle \leq D_6$ 在顶点集合上的作用只有一个轨道，即全体顶点集.

容易证明，群 G 的作用下，集合 X 的全体轨道构成了该集合的一个分划，记作 X/G. 但是和陪集不同，这些轨道并不一定是等长的.

稳定化子

群作用下，集合中的一个元素的轨道长度取决于有多少群里的元素对应的置换以它为不动点.

例子：正三角形的空间对称群 D_6（续）

比如说，之所以在群 $\langle s \rangle \leq D_6$ 的作用下，顶点 1 的轨道长是一，是因为所有群里的元素都将顶点 1 映到其自身；而顶点 2 的轨道长是二，是因为只有单位元 e 将顶点 2 映射到其自身.

这启发了如下的定义.

稳定化子

对于群 G 在集合 X 上的作用和 $x \in X$，称群 G 中 x 的 **稳定化子**（stabilizer）是子群 $G_x = \{g \in G : gx = x\}$.

群作用的核就是集合中所有元素的稳定化子的交.

例子：正三角形的空间对称群 D_6（续）

考虑群 D_6 在顶点集合上的群作用，则顶点 1 的稳定化子是 $\{e, s\} = \langle s \rangle$. 这是 D_6 的子群. 因为 D_6 可以划分成左陪集 $\langle s \rangle$，$r\langle s \rangle$ 和 $r^2\langle s \rangle$，容易发现，每个左陪集对顶点 1 作用的结果是一样的.

这一例子说明，轨道上的元素，都和稳定化子的左陪集一一对应. 这说明如下结果.

定理

对于群 G 在集合 X 上的作用，元素 $x \in X$ 的稳定化子 G_x 是 G 的子群，且子群 G_x 的左陪集与轨道 Gx 存在双射.

证明

验证映射 $gG_x \mapsto gx$ 是良定义的双射即可.

利用 Lagrange 定理，可以将轨道长和稳定化子的陪集数目联系起来. 这就是 **轨道稳定化子定理**（orbit-stabilizer theorem）.

轨道稳定化子定理

对于有限群 G 在集合 X 上的作用和 $x \in X$，有 $|\, Gx \,| = [G : G_x] = |\, G \,| / |\, G_x|$.

可以在上面的例子中验证这一结论.

Burnside 引理

这一引理给出了群作用的轨道个数公式.

Burnside 引理

对于群 G 在集合 X 上的作用，轨道的个数等于群中每个元素对应置换的不动点的平均个数，即

$$|X/G| = \frac{1}{|G|}\sum_{g\in G}|X^g| .$$

这里，$X^g = \{x \in X : gx = x\}$ 是元素 $g \in G$ 对应置换的不动点集合.

证明

这一定理的证明十分简明. 注意到，轨道个数可以写作

$$|X/G| = \sum_{o\in X/G} 1 = \sum_{x\in X}\frac{1}{|Gx|} = \frac{1}{|G|}\sum_{x\in X}|G_x| .$$

最后一个等号就是上面的推论; 而右式和所要求证的只差一个 Fubini 定理, 因为它们中的求和式都是对集合 $\{(g,x) \in G \times X : gx = x\}$ 的计数, 只不过右式先对 g 求和, 而所求证的式子先对 x 求和.

这一定理在组合数学中有很多用处，可以用于统计「本质不同」的对象的数目. 更多例子和讨论可以参考 Pólya 计数→9.13.14@p1680.

Cayley 定理

利用群作用研究群的结构，需要选取合适的集合. 事实上，群自身就是这样一个集合. 因此，接下来考虑群对自身的两类常见的群作用，并借此分析群的结构.

第一个这样的群作用是群对自身的 **左乘作用**（left multiplication）. 它的置换表示如下.

$$\begin{aligned} \varphi : G &\to S_G \\ g \mapsto \varphi_g : G &\to G \\ x &\mapsto gx \end{aligned}$$

群的左乘作用必然是忠实的，因为群满足消去律. 所以，由同态基本定理，G 可以嵌入对称群 S_G 中. 这意味着，每个群都同构于某个置换群 [8].

Cayley 定理

群 G 同构于对称群 S_G 的子群.

这个群作用只有一个轨道，而且，每个元素的稳定化子都是 $\{e\}$.

共轭作用

第二个群到自身的作用叫做 **共轭作用**（conjugation）. 它的置换表示如下.

$$\begin{aligned} \varphi : G &\to S_G \\ g \mapsto \varphi_g : G &\to G \\ x &\mapsto gxg^{-1} \end{aligned}$$

在群的共轭作用下，轨道和稳定化子都有特别的名字.

[8] Cayley 定理本身没有反映太多群本身结构的信息，因为 S_G 这个群的规模通常很庞大，很难讲它的某个大小恰为 $|G|$ 的子群拥有哪些确定的性质. 但是，早期群论的发展主要集中在置换群上. 所以，Cayley 定理其实是在说，所有可能的群结构都是这些已经充分研究过的对象. 尽管在实际研究时，需要更为精细的工具.

共轭类

对于群 G 和 $g \in G$，元素 g 在群 G 中的 **共轭类**（conjugacy class）是共轭作用下 g 的轨道．如果元素 g 和 h 处在同一共轭类中，则称 g 和 h **共轭**（conjugate）．

中心化子

对于群 G 和 $a \in G$，元素 a 在群 G 中的 **中心化子**（centralizer）是 $C_G(a) = \{g \in G : ga = ag\}$.

中心

对于群 G，群 G 的 **中心**（center）为 $Z(G) = \cap_{a\in G} C_G(a) = \{g \in G : \forall a \in G(ga = ag)\}$.

群的中心，是与群中所有元素都交换的元素的集合；因为它是群作用的核，它必然是正规子群．群的中心的大小，表明了它和交换群之间的差距．给定元素的中心化子，是与该元素交换的所有元素的集合；也是所有以它为中心元素的子群中最大的，这也就是它的名字来源；同时，因为它是共轭作用下某元素的稳定化子，所以它是子群．共轭类一般不是子群．

例子：正三角形的空间对称群 D_6（续）

回到群 D_6，它的中心是 $\{e\} \neq G$，这说明它不是交换群．元素 r 的中心化子 $C_G(r)$ 是 $\langle r\rangle$，元素 s 的中心化子 $C_G(s)$ 为 $\langle s\rangle$．一般地，对所有 $g \in G$，总成立 $\langle g\rangle \leq C_G(g)$．群 D_6 的共轭类共三个，即 $\{e\}, \{r, r^2\}, \{s, sr, sr^2\}$．容易发现，共轭的元素都是同阶的[9]．

群在共轭作用下划分成若干个共轭类．所以，可以写出 **类方程**（class equation）．

类方程

对于群 G，设 $\{\mathcal{K}_i\}_{i=1}^r$ 为全体长度大于一的共轭类，且 g_i 是 $\mathcal{K}_i$ 的代表元，成立

$$|G| = |Z(G)| + \sum_{i=1}^{r} [G : C_G(g_i)].$$

它可以用于分析群的结构，比如用于证明下文的 Sylow 定理→9.16@p1805.

正规化子和中心化子

事实上，群的幂集也可以是群作用的对象．这里着重讨论群在它的全体子集 $X = \mathcal{P}(G)$ 上的共轭作用．它的置换表示如下．

$$\begin{aligned} \varphi : G &\to S_X \\ g &\mapsto \varphi_g : X \to X \\ S &\mapsto gSg^{-1} = \{gsg^{-1} : s \in S\} \end{aligned}$$

在这一共轭作用下，同样可以定义群的子集的共轭类．它的核依然是群的中心．它的稳定化子则称为正规化子．

正规化子

对于群 G 和它的子集 $S \subseteq G$，子集 S 在群 G 中的 **正规化子**（normalizer）是 $N_G(S) = \{g \in G : gSg^{-1} = S\}$.

[9] 更一般地，置换群中共轭的元素必然有着相同的 型→9.8@p1138.

正规化子是共轭作用下 S 的稳定化子，必然是 G 的子群；S 的正规化子是能够使得 S 包含在其正规子群中的子群中最大的，这也是正规化子的名字来源．特别地，对于单元素集合 $\{a\}$，$N_G(\{a\}) = C_G(a)$．

中心化子也可以类似地推广到子集的情形．

自 中心化子

对于群 G 和它的子集 $S \subseteq G$，子集 S 在群 G 中的 **中心化子**（centralizer）是 $C_G(S) = \{g \in G : \forall s \in S(gsg^{-1} = s)\}$．

中心化子实际上是群 $N_G(S)$ 在 S 的共轭作用的核．所以，必然有 $C_G(S) \trianglelefteq N_G(S) \leq G$，所以由子群的传递性，中心化子也是子群．特别地，群的中心 $Z(G) = C_G(G) \trianglelefteq N_G(G) = G$ 必然是正规子群．

Sylow 定理

进一步地，对有限群的共轭作用进行分析，可以得到 Sylow 定理．它是处理有限群的结构的有力工具，可以迅速地得到大量小阶群的结构．

自 p-群

对于群 G，如果存在素数 p 和正整数 α 使得 $|G| = p^\alpha$，则称它为 **p-群**（p-group）．

关于 p-群的定义

p-群还有另外一种定义，即所有元素的阶都是素数的幂的群．这两种定义对于有限群是等价的；但是，第二种定义同样适用于无限群的情形．另外，关于 p-群的定义，不同的文献可能对于是否将 $\{e\}$ 算作 p-群存在分歧，读者在阅读时应当加以分辨．

自 p-子群

对于群 G 和它的子群 $P \leq G$，如果 P 本身是一个 p-群，则称 P 为 **p-子群**（p-subgroup）．

Cauchy 定理

如果素数 p 能够整除群 G 的阶，则必然存在 p 阶元．

Cauchy 定理保证了 p-子群的存在性．其实，通过更为细致的分析，能够得到更强的结论，即 Sylow p-子群的存在性．

自 Sylow p-子群

对于群 G 和它的子群 $P \leq G$，如果 $|G| = p^\alpha m$，$p \perp m$ 且 $|P| = p^\alpha$，则称 P 为 **Sylow p-子群**（Sylow p-subgroup）．

也就是说，Sylow p-子群是极大的 p-子群．Sylow 定理断言了 Sylow p-子群的存在性，这提供了 Lagrange 定理的一定程度的逆命题．

Sylow 定理

设有限群 G 的阶 $|G|$ 能表达成 $p^\alpha m$ 的形式，其中，p 是质数且 p 不能整除 m，那么有如下结论：

1. 存在 Sylow p-子群；
2. 对于一个 Sylow p-子群 P 和一个群 G 的 p-子群 Q，存在 $g \in G$ 使得 $Q \leq gPg^{-1}$ 成立，特别地，所有 Sylow p-子群都共轭；

3. 群 G 中的 Sylow p-子群的数目 n_p 满足 $n_p \equiv 1 \pmod p$，$n_p \mid m$ 且 $n_p = [G : N_G(P)]$，其中 P 是任意的 Sylow p-子群.

证明

为证明第一部分，对 $|G|$ 进行归纳. 考虑类方程

$$|G| = |Z(G)| + \sum_{i=1}^{r} [G : C_G(g_i)].$$

如果 $p \mid |Z(G)|$，那么可以任取中心的 p 阶子群 $N \leq Z(G)$，因为 $N \leq Z(G)$，它与 G 中所有元素交换，故而 $N \trianglelefteq G$，因而，G/N 是更小的群. 根据归纳假设，它有 Sylow p-子群 P/N，则自然同态下相应的原像 P 就是原来的群 G 的 Sylow p-子群. 于是，只要考虑 p 不能整除 $|Z(G)|$ 的情形. 此时，可以取 $g_i \in G$，使得 p 不能整除 $[G : C_G(g_i)]$，因而，$|G|$ 中 p 的全部幂次都出现在 $|C_G(g_i)|$ 中. 再次根据归纳假设，$C_G(g_i)$ 有 Sylow p-子群 P，它正是原来的群 G 的 Sylow p-子群.

为证明第二部分和第三部分，设群 G 的 Sylow p-子群 P 的共轭类为 $\mathcal{S} = \{gPg^{-1} : g \in G\}$，并考察群 G 在 $\mathcal{S}$ 上的共轭作用限制在某一 p-子群 Q 上的结果. 集合 $\mathcal{S}$ 在该作用下可以划分成若干轨道. 设总共有 r 个轨道，每个轨道的代表元为 P_i，则有恒等式

$$|\mathcal{S}| = \sum_{i=1}^{r} [Q : Q \cap N_G(P_i)].$$

这里，$Q \cap N_G(P_i)$ 是 P_i 在上述群作用下的稳定化子. 设 $H = Q \cap N_G(P_i)$，则显然 $H \leq N_G(P_i)$，故而根据群的第二同构定理，有 $P_iH/P_i \cong H/(P_i \cap H)$. 所以，$|P_iH| = |P_i| \, |H| \, / \, |P_i \cap H|$，而右侧是 p 的幂，左侧必然也是 p 的幂，而且需要成立 $|P_i| \leq |P_iH|$. 但是，P_i 已经是 G 的子群中 p 的幂次最大的，所以 $P_i = P_iH$. 代入前文的同构关系中，就有 $H = P_i \cap H = Q \cap P_i$. 这里用到了显然的关系 $P_i \leq N_G(P_i)$. 这样，就知道稳定化子 $Q \cap N_G(P_i)$ 实际上就是 $Q \cap P_i$. 所以，上式可以写作

$$|\mathcal{S}| = \sum_{i=1}^{r} [Q : Q \cap P_i].$$

这一等式对于所有 p-子群 Q 都成立.

自然，Sylow p-子群 P 也是 p-子群. 将 P 代入上式中的 Q，则右侧有且仅有一项的大小是 1，其余都是 p 的倍数. 因此，Sylow p-子群 P 的共轭类的大小 $|\mathcal{S}|$ 必然模 p 余 1. 同时，因为这一式子对于任何 p-子群 Q 都成立，就自然存在 $g \in G$，使得 $Q \leq gPg^{-1}$；否则，对所有上述等式中的 P_i 都有 $|Q \cap P_i| < |Q|$，就与 $|\mathcal{S}| \equiv 1 \pmod p$ 矛盾. 如果此时再考虑 Q 是某个不同于 P 的 Sylow p-子群，则必然有 $Q = gPg^{-1}$ 对某个 $g \in G$ 成立，即所有 Sylow p-子群都共轭. 因而，Sylow p-子群的数目 n_p 就是 $|\mathcal{S}|$，它必然满足 $n_p \equiv 1 \pmod p$. 最后，注意到 $n_p = |\mathcal{S}| = [G : N_G(P)]$，且 $P \leq N_G(P)$，就成立 $n_p \mid m$. 这样就完整地证明了定理的第二部分和第三部分.

应用：同构意义下，六阶群只有 C_6 和 S_3

设 G 的阶是 6. 那么，根据 Sylow 定理，它有 Sylow 2-子群，且它的数目满足 $n_2 \equiv 1 \pmod 2$ 和 $n_2 \mid 3$，所以，只有两个情形：$n_2 = 1$ 或 $n_2 = 3$. 同理，可以证明群 G 有且只有一个 Sylow 3-子群，即 $n_3 = 1$.

对于 $n_2 = 1$ 的情形，可以发现群 G 中有一个 Sylow 2-子群，故而有一个 2 阶元；又有一个 Sylow 3-子群，故而有两个 3 阶元. 群 G 还有一个单位元，而剩下的元素，根据 Lagrange 定理，它的阶数必须整除 6. 又不能是新的 2 阶或 3 阶元，否则会出现与前文不同的新的 Sylow p-子群；所以，剩下的元素只能是 6 阶元. 存在和群的阶数相同的元素，这意味着群 G 是循环群，所以 $G \cong C_6$.

对于 $n_2 = 3$ 的情形，群 G 有三个共轭的 Sylow 2-子群．考虑群 G 在这三个 Sylow 2-子群上的共轭作用．对于任何一个 Sylow 2-子群 P，根据 Sylow 定理，有 $|N_G(P)| = 2$；但又有 $P \leq N_G(P)$，所以 $P = N_G(P)$．因此，这三个 Sylow 2-子群的正规化子的交集，即这个共轭作用的核，是平凡的．所以，这个作用是忠实的，它将 G 嵌入到了这三个 Sylow 2-子群上的置换群 S_3．但是，因为 $|G| = |S_3|$，必然有 $G \cong S_3$．

有限生成 Abel 群

在掌握分析群结构的基本工具后，现在重点讨论一类群的结构．

在概述中提到，Abel 群因为元素可以交换，结构相较于其它群更为简单．其中尤为简单的是那些可以通过有限多个元素生成的 Abel 群．

自 有限生成

如果群 G 有一个有限的生成子集，则称群 G 是 **有限生成的**（finitely generated）．

本节的分类定理说明，有限生成的 Abel 群可以看作是有限多个的循环群的简单组合．算法竞赛中涉及的群多为有限群．有限 Abel 群必然是有限生成的，因此总是适用这一结论．

直积

前文对群的分析主要集中在如何将群分解为更小的群；相反地，自然可以讨论如何将两个群组合成更大的群．在所有可能的组合方式中，群的直积是最为简单的一种．

群的直积的基本想法是，给定两个群 G 和 H，考虑其笛卡尔积 $G \times H$，二元对 (g, h) 的运算定义为对两分量分别运算，分量之间互不影响．这样得到的结果显然是更大的群，且原来的两个群可以平凡地嵌入新的群中．

自 直积

群 $(G, \cdot_G)$ 和群 $(H, \cdot_H)$ 的 **直积**（direct product）是群 $(G \times H, \cdot)$，其中，二元运算 $\cdot : (G \times H) \times (G \times H) \to G \times H$ 定义为 $(g_1, h_1) \cdot (g_2, h_2) = (g_1 \cdot_G g_2, h_1 \cdot_H h_2)$．群 G 和群 H 的直积记为 $G \times H$．

对于直积 $G \times H$，平凡地有嵌入映射 $g \mapsto (g, e_H)$ 和 $h \mapsto (e_G, h)$．反过来，映射 $(g, h) \mapsto g$ 和映射 $(g, h) \mapsto h$ 是群同态，同态的核分别是 $\{e_G\} \times H$ 和 $G \times \{e_H\}$．这两个核恰好是上述嵌入映射的像，且它们的交集是平凡的，即 $\{(e_G, e_H)\}$．所以，群的直积 $G \times H$ 中有两个子群，都是正规子群，交集是平凡的，且它们的乘积就是直积 $G \times H$ 本身．

这样的分析其实给出了一个群能够写成它的两个子群的直积的充分必要条件．

定理

对于群 G 和它的子群 $H_1, H_2 \leq G$，$G \cong H_1 \times H_2$ 当且仅当 $H_1, H_2 \trianglelefteq G$，$H_1 \cap H_2 = \{e\}$ 且 $G = H_1 H_2$．

证明

这些条件的必要性在正文中已经讨论过，这里证明它们的充分性．考察映射 $\varphi : G \to H_1 \times H_2$ 满足 $h_1 h_2 \mapsto (h_1, h_2)$．映射 φ 是良定义的，因为对于任意 $h_1, k_1 \in H_1$ 和 $h_2, k_2 \in H_2$，满足 $h_1 h_2 = k_1 k_2$ 就意味着 $h_1 = k_1$ 和 $h_2 = k_2$；这是因为 $k_1^{-1} h_1 = k_2 h_2^{-1} \in H_1 \cap H_2 = \{e\}$．要说明 φ 是群同态，则就是要说明 $(h_1 h_2)(k_1 k_2) = h_1 k_1 h_2 k_2$，这等价于 k_1 和 h_2 是可交换的，亦即 $k_1 h_2 k_1^{-1} h_2^{-1} = e$．要证明这一关系，只要注意到 $k_1 h_2 k_1^{-1} h_2^{-1} = (k_1 h_2 k_1^{-1}) h_2^{-1} \in (k_1 H_2 k_1^{-1}) H_2 = H_2$，同理也有 $k_1 (h_2 k_1^{-1} h_2^{-1}) \in$

H_1，故而 $k_1h_2k_1^{-1}h_2^{-1} \in H_1 \cap H_2 = \{e\}$．这些就证明了 φ 是群同态．它显然是双射，故而它是同构，即 $G \cong H_1 \times H_2$．

在直积中，两个直积因子的元素必然是可以交换的，这是因为 $hg = (e_G, h)(g, e_H) = (g, h) = gh$．所以，如果两个直积因子都是 Abel 群，那么直积也必然是 Abel 群．

并不是所有的群都可以写成两个非平凡子群的直积．

例子：正三角形的空间对称群 D_6（续）

例如，群 $D_6 = \langle r, s\rangle$ 就不同构于 $\langle r\rangle \times \langle s\rangle$，因为作为两个循环群的直积，后者必然是 Abel 群．

下面的分类定理则说明，有限生成的 Abel 群都可以写作有限多个循环群的直积．

分类定理

对于有限生成的 Abel 群，有如下分类定理．它称为 **有限生成 Abel 群基本定理**（fundamental theorem of finitely generated Abelian groups）．

有限生成 Abel 群基本定理

对于有限生成的 Abel 群 G，存在整数 $r \geq 0$ 和 $n_1, \cdots, n_s \geq 2$，使得

$$G \cong C_\infty^r \times C_{n_1} \times \cdots \times C_{n_s}.$$

特别地，r 是唯一确定的，称为群 G 的 **秩**（rank），而且

- 可以选取整数 $n_1, \cdots, n_s$ 使其满足 $n_1 \geq 2$, $n_1 \mid n_2, \cdots, n_{s-1} \mid n_s$，此时，整数 $n_1, \cdots, n_s$ 唯一确定，因子 C_{n_i} 称为群 G 的 **不变因子**（invariant factor）；
- 也可以选取整数 $n_1, \cdots, n_s$ 使其都是素数幂的形式，此时，这些素数幂也都唯一确定，因子 C_{n_i} 称为群 G 的 **初等因子**（elementary divisor）．

定理首先断言，有限生成的 Abel 群一定是有限多个循环群的直积．

证明

这里给出一个形式简单的证明[10]；更深刻的证明应当参考主理想整环上的有限生成模的结构定理[11]．在证明中，为书写简便，将使用加法记号代替一般的群中的乘法记号，此时 0 代表单位元，而记号 mx 代表 x 的 m 次幂．

设 G 最少可以由 k 个元素生成．定理的证明需要对 k 进行归纳．当 $k = 1$ 时结论是平凡的．当 $k > 1$ 时，取 G 的全体生成元组 $(x_1, x_2, \cdots, x_k)$ 中 x_1 的阶最小的那个．下面要说明 $G = \langle x_1\rangle \times \langle x_2, \cdots, x_k\rangle$，后者则根据归纳假设已经可以分解成 $(k-1)$ 个循环群的直积，故归纳步骤得证．

根据群的直积的刻画，如果直积分解不成立，必然存在关系 $m_1x_1 + m_2x_2 + \cdots + m_kx_k = 0$，且 $m_1x_1 \neq 0$．对于负的系数 m_i，可以用逆元 $-x_i$ 代替 x_i，则所有系数 m_i 都可以取作非负整数．而且，此时可以取 $0 < m_1 < |x_1|$．如果再取 $d = \gcd(m_1, m_2, \cdots, m_k)$ 和 $c_i = m_i/d$，则必然有 $y_1 = c_1x_1 + c_2x_2 + \cdots + c_kx_k$ 满足 $dy_1 = 0$，因而 $|y_1| \leq d \leq m_1 < |x_1|$，即 y_1 是比 x_1 阶更小的元素．

下面证明，y_1 可以扩张成 G 的一组生成元．也就是说，存在元素 $y_2, \cdots, y_k \in G$ 满足 $G = \langle y_1, y_2, \cdots, y_k\rangle$．这里唯一的已知条件是 y_1 本身可以写作 $c_1x_1 + c_2x_2 + \cdots + c_kx_k$，其中，系数 c_i 都是自然数且它们的最大公约数是一．不妨假设系数（非严格）递减排列，则 y_1 也可以写作 $(c_1 - c_2)x_1 + c_2(x_1 + x_2) + \cdots + c_kx_k$．此时，对比之前的条件，可以发现系数依然全部是自然数，且最大公约数是一，而且 $G = \langle x_1, x_1 + x_2, x_3, \cdots, x_k\rangle$，但是全体系数的和减少了 c_2．如果 $c_2 = 0$，则必然有 $c_1 = 1$，

[10] 参见 Milne, J.S. (2021) Group Theory 第 25 页.
[11] 参见 Structure theorem for finitely generated modules over a principal ideal domain - Wikipedia[4].

结论是平凡的；否则，系数的和严格地减少了．这意味着，如果对系数的和进行归纳，就可以证明满足上述条件的 y_1 总可以扩张成 G 的一组生成元．

进而，此时找到了比 x_1 更小阶的生成元 y_1，这与 x_1 的选取矛盾．所以，直积分解必然成立，根据归纳原理就知道所求证的结论成立．

当然，循环群可能是无限阶的或是有限阶的，它们分别是上述分解的 C_∞ 部分和 C_{n_i} 部分．然后，定理给出了有限阶循环群 C_n 的结构．定理的初等因子分解的部分其实依赖于如下观察．

引理

如果 m 与 n 互质，那么 $C_{mn} \cong C_m \times C_n$.

证明

设 x 和 y 分别是 C_m 和 C_n 的生成元，那么因为 $(m,n)=1$，就有 $|(x,y)| = mn = |C_m \times C_n|$. 所以，$(x,y)$ 就是 $C_m \times C_n$ 的生成元，亦即 $C_m \times C_n = \langle (x,y) \rangle$．所以，它是 mn 阶循环群，必然同构于 C_{mn}.

给定循环群 C_n，如果根据算术基本定理有 $n = p_1^{r_1} \cdots p_k^{r_k}$，那么重复利用引理，就可以证明 $C_n \cong C_{p_1^{r_1}} \times \cdots \times C_{p_k^{r_k}}$．在这两步分解之后，实际上就已经得到了定理中的初等因子分解．再根据引理，重组这些素数幂阶循环群，使得它们的阶数符合要求，就可以得到定理中的不变因子分解．定理中的唯一性可以归纳证明得出．

例子：24 阶 Abel 群的分类

作为示例，可以通过定理得知所有的 24 阶 Abel 群共三种，列举如下．

不变因子分解	初等因子分解
C_{24}	$C_3 \times C_8$
$C_2 \times C_{12}$	$C_2 \times C_3 \times C_4$
$C_2 \times C_2 \times C_6$	$C_2 \times C_2 \times C_2 \times C_3$

参考资料与注释

- Dummit, D.S. and Foote, R.M. (2004) Abstract Algebra. 3rd Edition, John Wiley & Sons, Inc.
- Milne, J.S. (2021) Group Theory[5].
- Group (mathematics) - Wikipedia[6]
- Group theory - Wikipedia[7]
- Group - Wolfram MathWorld[8]
- Visual Group Theory[9]

[1] 代数结构

[2] 维基百科

[3] Group action - Wikipedia

[4] Structure theorem for finitely generated modules over a principal ideal domain - Wikipedia

[5] Milne, J.S. (2021) Group Theory

[6] Group (mathematics) - Wikipedia

[7] Group theory - Wikipedia

[8] Group - Wolfram MathWorld

[9] Visual Group Theory

9.16.3 环论

前置知识：抽象代数基本概念→9.16.1@p1805、群论→9.16.2@p1812

引入

环论（ring theory）研究形形色色的环.

本文涉及的环论的内容，与数论中的整除理论密不可分. 首先，类似群论中的正规子群，本文首先介绍环同态的核，它称作环的理想；其实，这是数论中的数的概念在一般环的推广. 然后，考虑将整数环上的素数、辗转相除法、质因子分解等概念推广到一般的环上，就有了不同类型的整环的概念.

数论中的很多结论在其它常见的环上，都依然成立. 可以说，环论的一部分工作，就是在讨论使得这些数论中的结论在一般的环上能否成立；如果不能，需要给环施加怎样的限制才能够使这些结论成立.

记号

在不引起歧义时，本文可能会省略掉环的乘法记号，并且会将环 $(R,+,\cdot)$ 写作环 R. 环 R 中，加法单位元也称作零元，记作 0；乘法单位元也称作幺元，记作 1.

本文的环的定义不要求有幺元

注意，本文的环的定义不要求含幺. 有些文章要求环的定义含幺，则本文部分结论的叙述需要稍作调整. 比如说，本文中理想可以基于子环定义，但是其它文章中可能需要基于加法子群定义.

理想

类似群的情形，可以建立子环和环同态的概念.

子环

对于环 $(R,+,\cdot)$ 和它的子集 S，如果 $(S,+,\cdot)$ 也是一个环，则称 S 是 R 的 **子环**（subring）.

例子：整数环 $\mathbf{Z}$

对于任何整数 n，都有 $n\mathbf{Z}=\{nk:k\in\mathbf{Z}\}$ 是 $\mathbf{Z}$ 的一个子环.

环同态

对于环 $(R,+,\cdot)$ 和 $(S,\oplus,\odot)$，如果 π 保持环的加法和乘法运算，即对所有 $r_1,r_2\in R$ 都成立 $\pi(r_1+r_2)=\pi(r_1)\oplus\pi(r_2)$ 和 $\pi(r_1\cdot r_2)=\pi(r_1)\odot\pi(r_2)$，则称映射 $\pi:R\to S$ 是自环 R 到环 S 的 **同态**（homomorphism）.

环的定义要求含幺的情形

如果环的定义要求含有幺元，那么，环同态的定义也常常要求将幺元映射至幺元. 对于非零幺环间的同态来说，这个额外的要求仅仅是保证了同态不会将整个幺环映射到零元.

例子：整数环 $\mathbf{Z}$（续）

对任何非零整数 n 取模的映射，即 $\pi:\mathbf{Z}\to\mathbf{Z}/n\mathbf{Z}$，其中，$\pi(a)=\bar{a}$，都是环同态.

对群同态的核和像的 讨论→9.16.2@p1812 可以几乎原封不动地搬到此处. 同态的像的（相对）大小决定了同态是否是满射，而同态的核的平凡与否则决定了同态是否是单射. 环同态的核定义如下：

自 同态的核

自环 R 到环 S 的同态 $\pi : R \to S$ 的 **核**（kernel）是 $\{r \in R : \pi(r) = 0\}$，记作 $\ker \pi$，其中，0 是 S 的加法单位元.

显然，环同态的核和像都是子环. 反过来，并不是所有子环都可以成为某个环同态的核. 能够成为环同态的核的子环称为环的理想.

自 理想

对于环 R 和它的子环 I，则称 I 是 R 的

- **左理想**（left ideal），如果对于所有 $r \in R$，都有 $rI \subseteq I$，这里，$rI = \{ra : a \in I\}$；
- **右理想**（right ideal），如果对于所有 $r \in R$，都有 $Ir \subseteq I$，这里，$Ir = \{ar : a \in I\}$；
- **理想**（ideal），如果 I 既是 R 的左理想，也是 R 的右理想.

这里要求理想 I 对环 R 的左乘和右乘都封闭. 这个条件是自然的. 因为，理想中的元素在环同态中会映射到零元，而任何数左乘或右乘以零都应该等于零，这就是所要求的封闭性. 除此之外，因为环的加法结构是 Abel 群，任何子群都是正规子群；而环的乘法结构又十分原始，不会对子结构施加额外的限制. 这就说明，对左乘和右乘封闭这个条件也是充分的.

例子：整数环 $\mathbf{Z}$（续）

作为例子，前面提到的子环 $n\mathbf{Z}$ 其实是 $\mathbf{Z}$ 的理想. 它是所有 n 的倍数构成的集合. 一个 n 的倍数，与任何整数相乘，都会得到 n 的倍数. 事实上，$\mathbf{Z}$ 的全部理想都是这样的形式，这样的环称为 主理想整环→9.16@p1805. 对于一般的环，有些理想并不是某个元素的倍数的集合；这样的一般的环的存在，也正是研究理想（而不是简单地研究倍数）的最初动机[12].

商环

和群一样，基于环的理想，可以在全体（加法群意义上的）陪集的集合上定义 **商环**（quotient ring）. 考虑集合

$$R/I = \{a + I : a \in R\},$$

这里，陪集 $a + I = \{a + b : b \in I\}$. 可以证明当且仅当 I 是理想时，运算

$$(a + I) + (b + I) = (a + b) + I,$$

$$(a + I)(b + I) = (ab) + I$$

是良定义的，即这些运算的结果和陪集中代表元的选取无关. 在这些运算下，R/I 构成环. 再次和群的情形一致，可以建立环的 **第一同构定理**（first isomorphism theorem），并存在环到其商环的自然同态. 这些证明，环的理想和群的正规子群，在相应结构的同态中起到了一样的作用.

第一同构定理

设 $\pi : R \to S$ 是自环 R 到环 S 的同态，则 $\ker \pi$ 是 R 的理想，且 $R/\ker \pi \cong \pi(R)$ 是 S 的子环.

自 自然同态

对于环 R 和它的理想 I，则由 $\pi(r) = r + I$ 给出的映射 $\pi : R \to R/I$ 是自 R 到 R/I 的满同态，称为自环 R 到商环 R/I 的 **自然同态**（natural homomorphism）.

[12] https://en.wikipedia.org/wiki/Ideal_(ring_theory)#History[1]

例子：整数环 $\mathbf{Z}$（续）

作为例子，整数模 n 的同余类构成的环 $\mathbf{Z}/n\mathbf{Z}$ 是 $\mathbf{Z}$ 模它的理想 $n\mathbf{Z}$ 得到的商环．这也解释了符号 $\mathbf{Z}/n\mathbf{Z}$ 的含义．上面提到的模 n 的映射 $\pi : \mathbf{Z} \to \mathbf{Z}/n\mathbf{Z}$ 就是这里提到的自然映射，相应的核正是理想 $n\mathbf{Z}$.

在环的情形，同样成立其他同构定理.

第二同构定理

设环 R 有子环 A 和理想 B，那么 $A+B=\{a+b : a\in A, b\in B\}$ 同样是 R 的子环，而 $A\cap B$ 是 A 的理想，B 是 $A+B$ 的理想，并且 $(A+B)/B \cong A/(A\cap B)$.

第三同构定理

设环 R 有理想 I, J 且 $I\subseteq J$，那么 J/I 也是 R/I 的理想，并且 $(R/I)/(J/I)\cong R/J$.

对应定理

设环 R 有理想 I，则全体包含 I 的环 R 的子环 $\mathcal{S}=\{S : I\subseteq S\subseteq R\}$ 和商环 R/I 的全体子环 $\mathcal{T}=\{T : T \text{ is a subring of } R/I\}$ 之间存在双射 $\varphi : \mathcal{S}\to\mathcal{T}$，它将 $S\in\mathcal{S}$ 映射至 $S/I\in\mathcal{T}$．这个双射保持子环的包含关系，且环 R 的理想总是映射到 R/I 的理想.

这些定理在后文中讨论环和理想的结构时将起到基础的作用.

理想的运算

环的理想上可以定义各种运算．这类似于整数的整除结构上可以定义最大公约数、最小公倍数等概念.

自 理想的运算

设环 R 有理想 I, J，可以定义如下运算：

- 理想的 **和**（sum）：$I+J=\{a+b : a\in I, b\in J\}$；
- 理想的 **乘积**（product）：$IJ=\{\sum_{i=1}^{n} a_i b_i : a_i\in I, b_i\in J\}$，即全体 ab 形式乘积的有限和构成的集合；
- 理想的 **交**（intersection）：$I\cap J$.

容易验证，这些运算的结果都依然是环的理想.

例子：整数环 $\mathbf{Z}$（续）

考虑整数环 $\mathbf{Z}$ 的情形．对于理想 $n\mathbf{Z}$ 和 $m\mathbf{Z}$，可以得到

$$n\mathbf{Z}+m\mathbf{Z}=\gcd(m,n)\mathbf{Z},$$
$$(n\mathbf{Z})(m\mathbf{Z})=(mn)\mathbf{Z},$$
$$(n\mathbf{Z})\cap(m\mathbf{Z})=\operatorname{lcm}(m,n)\mathbf{Z}.$$

一般地，对于环 R 和它的理想 I 和 J，总有

$$IJ\subseteq I\cap J\subseteq I, J\subseteq I+J.$$

利用这些定义，可以将整数的中国剩余定理推广到一般的环上．但在此之前，还需要进一步将诸如素数和互素等概念推广到一般的环上.

极大理想

通过环的理想的结构，可以理解环的性质.

非零环 R 总有两个平凡的理想，即 $\{0\}$ 和 R. 如果环 R 还是交换的非零幺环，那么只有这两个理想的环能且只能是域[13].

定理

设 R 是交换的非零幺环，那么 R 是域，当且仅当 R 只有平凡理想 $\{0\}$ 和 R.

证明

如果 R 是域，则对于任何非零理想 I 都可以任取非零元素 $a \in I$，于是，任何域中的元素 $r \in R$ 都有 $r = (ra^{-1})a \in (ra^{-1})I \subseteq I$，故而 $I = R$. 反过来，对于任何 $a \in R$ 且 $a \neq 0$，可以验证 $aR = \{ar : r \in R\}$ 是理想，它必然等于 R，因而存在 $b \in R$ 使得 $ab = 1$，这就说明 a 存在逆元，故而有 R 是域.

这里交换环的条件是必要的；不然，需要同时限制左理想和右理想都是平凡的，才能保证环是除环.

这里的结论可以推广到环本身不是域的情形. 但是，此时需要转而考虑商环，讨论交换非零幺环的商环是域的条件. 商环 R/I 是域，这意味着商环 R/I 中只有平凡理想，根据对应定理可知，原来的环 R 中没有严格介于模掉的理想 I 和原来的环 R 之间的理想. 这样的理想 I 称为极大理想.

极大理想

对于环 R 和它的理想 M，如果 $M \neq R$，且包含 M 的 R 的理想只有 M 和 R 两个，则称理想 M 是一个 **极大理想**（maximal ideal）.

定理

设交换非零幺环 R 有理想 M，那么商环 R/M 是域，当且仅当 M 是极大理想.

例子：整数环 Z（续）

例如，整数环 $\mathbf{Z}$ 中的理想 $n\mathbf{Z}$ 是极大理想，当且仅当 n 是素数. 对于素数 p，商环 $\mathbf{Z}/p\mathbf{Z}$ 是域，也记作 $\mathbf{F}_p$.

并不是所有的环都有极大理想，但是非零幺环中总是有极大理想.

定理（Krull）

对于非零幺环 R 的理想 $I \neq R$，总有 R 的极大理想 M 使得 $I \subseteq M$ 成立.

证明

思路是利用 Zorn 引理. 考察全体包含 I 的 R 的真理想（即不等于 R 的理想）的集合 $\mathcal{S}$. 因为 $I \in \mathcal{S}$，它非空，且在包含关系下形成偏序集. 对于其中的任何链 $J_0 \subseteq J_1 \subseteq \cdots \subseteq J_n \subseteq \cdots$，设它们的并集为 J，则容易验证这也是理想. 而且，$J \neq R$，否则 $1 \in J$，亦即存在 n 使得 $1 \in J_n$，这与 J_n 是真理想相矛盾. 由此，依 Zorn 引理，存在极大理想 $M \supseteq I$.

[13]和群的情形一致，这样的环叫做 **单环**（simple ring）. 交换单环只能是域，非交换单环的情形则复杂得多.

极大理想，类比到整除理论中就是不可约元．这是因为，理想的包含关系就是整数的整除关系；没有作为超集的理想，就相当于没有可以除尽的因子．但是，极大理想的概念比不可约元更为宽泛，这是因为并不是所有的理想都是主理想．

素理想

域的条件比整环的更为苛刻．能够确保商环是整环的理想称为素理想，它类似于整除理论中的素数的概念．

自 素理想

对于交换环 R 和它的理想 P，如果 $P \neq R$，且对于环中任意元素 $a, b \in R$，每当 $ab \in P$ 成立时总有 $a \in P$ 或 $b \in P$，则称理想 P 是一个 **素理想**（prime ideal）．

这个定义看起来稍显突兀，但是对比 素数的定义→9.11.1@p1160，这个素理想的定义也是自然的．

定理

设交换非零幺环 R 有理想 P，那么商环 R/P 是整环，当且仅当 P 是素理想．

证明

对于交换非零幺环 R，商环 R/P 是整环，当且仅当 R/P 没有零因子．将陪集 $a+P$ 记作 $\bar{a}$．商环 R/P 没有零因子，就等价于 $\bar{a}\bar{b}=\bar{0}$ 总能推出 $\bar{a}=\bar{0}$ 或 $\bar{b}=\bar{0}$．根据商环的定义，这就等价于 $ab \in P$ 总能推出 $a \in P$ 或 $b \in P$．

在整数环 $\mathbf{Z}$ 中，$n\mathbf{Z}$ 是极大理想，当且仅当 n 是素数；而 $n\mathbf{Z}$ 是素理想，当且仅当 n 是素数或 $n=0$．在一般的交换环中，极大理想总能推出素理想，当然反之未必成立；这从它们对应的商环的性质上可以看出来．

定理

对于交换非零幺环 R，那么它的极大理想必然是素理想．

稍后要看到，只有在那些具有良好性质、足够与整数环相似的环中，逆命题才成立．

主理想

类似子群的概念，在环的讨论中也常需要考虑由某个子集生成的理想．

自 由子集生成的理想

对于非零幺环 R 和它的非空子集 $A \subseteq R$，如果 I 是包含 A 的 R 的理想中（依包含关系）最小的，则理想 I 称为 **由子集 A 生成的理想**（ideal generated by a subset），并记作 (A)．此时，A 称为 (A) 的 **生成子集**（generating set）．

自 主理想

由单个元素 $a \in R$ 生成的理想称为 **主理想**（principal ideal），记作 (a)．此时，a 称为 (a) 的 **生成元**（generator）．

对于集合 A，可以给出其生成的理想的构造．首先，有如下定义

$$RA = \{r_1a_1 + \cdots + r_na_n : r_i \in R, a_i \in A, n \in \mathbf{N}_+\},$$

$$AR = \{a_1r_1 + \cdots + a_nr_n : r_i \in R, a_i \in A, n \in \mathbf{N}_+\}.$$

其实它们分别是 A 生成的左理想和右理想. 然后, 由子集 A 生成的理想就是 RAR. 对于交换环, 定义出来的这些结构都相同.

整数环中的所有理想 $n\mathbf{Z}$ 都是主理想, 下文中常记作 (n).

整环

整环是交换、含幺、无零因子的非零环. 这个概念正是整数环的推广. 但是, 这样得到的环性质未必足够好到允许将整数的整除理论中的每个结论都原样照搬过来. 为了能够推广数论中的结论, 可以在整环上进一步作出限制. 其中, 最为常见的三种整环分别是欧几里得整环、主理想整环和唯一分解整环; 前面的概念严格地包含在后面的概念中.

整除关系

首先, 这里将整数的整除理论中的相关概念推广到一般的交换环上.

自 整除

设交换环 R 有元素 $a,b\in R$, 如果存在 $x\in R$, 满足 $a=bx$, 则称 b **整除** (divide) a, 记作 $b\mid a$. 此时称 b 是 a 的 **因子** (divisor).

自 相伴

设交换环 R 有元素 $a,b\in R$, 如果它们只相差了一个可逆元, 即存在可逆元 $u\in R$, 满足 $a=bu$, 则称 a 和 b 是 **相伴的** (associate).

整除关系是环上的 偏序→9.20@p1951 关系, 而相伴关系是环上的等价关系. 从理想的角度看, $a\mid b$ 等价于 $(b)\subseteq(a)$, a 和 b 相伴等价于 $(a)=(b)$. 因而, 在讨论环中的元素时, 通常不计较相伴元之间的差异. 和整数的情形类似, 交换环中 a 和 b 的最大公因子就定义为 $\{a,b\}$ 的下确界.

自 最大公因子

对于交换环 R 和它的元素 $a,b\in R$, 如果存在非零元素 $d\in R$, 它满足 $d\mid a$ 和 $d\mid b$, 且对于任何满足 $d'\mid a$ 和 $d'\mid b$ 的 d' 都成立 $d'\mid d$, 则称 d 是 a 和 b 的 **最大公因子** (greatest common divisor), 记作 $\gcd(a,b)$.

在整环中, 最大公因子在相伴意义下是唯一确定的. 下面的讨论就限制在整环中.

整环中还可以建立类似素数的概念. 在整数理论中, 素数存在着两个等价的定义, 但是在一般的整环中, 这两个定义对应着不同的概念:

自 素元

设整环 R 有非零元素 $p\in R$, 如果 (p) 是素理想, 也就是说, p 不是可逆元, 且 $p\mid ab$ 总能推出 $p\mid a$ 或 $p\mid b$, 则称 p 是 **素元** (prime).

自 不可约元

设整环 R 有非零元素 $r\in R$, 如果 r 不是可逆元, 而且对于任何 $a,b\in R$ 且 $r=ab$ 都有 a 或 b 是可逆元, 则称 r 是 **不可约元** (irreducible), 或称 r 不可约. 反过来, 如果 $r=ab$ 且 $a,b\in R$ 都不是可逆元, 则称 r 可约.

可以说明, 不可约元 r 对应的主理想 (r) 一定是环的所有真主理想中极大的; 但是, 一般的整环中, 并非所有理想都是主理想, 所以不可约元和极大理想的概念并不等价.

类似于证明主理想整环中, 素理想一定是极大理想, 一般地可以证明如下结论:

定理

设 R 是整环，如果 $a \in R$ 是素元，那么 a 也一定是不可约元.

证明

设 $r \in R$ 是素元，且 $a, b \in R$ 满足 $r = ab$. 因为 r 是素元，不妨设 $r \mid a$ 成立，则 $a = cr = cba$. 因为整环上成立消去律，有 $1 = bc$，故而，b 有逆元 c. 这就说明 r 是不可约元.

反过来，这一结论并不成立.

反例

在二次整数环 $\mathbf{Z}[\sqrt{-5}]$ 中，3 是不可约元，但是 $9 = 3 \cdot 3 = (2+\sqrt{-5})(2-\sqrt{-5})$，所以它不是素元.

这里给出这一反例的证明，不熟悉二次整数环的读者请先阅读 二次整数环→9.16@p1805 部分. 设 $N(\cdot)$ 是二次整数环上的范数. 对于任何分解 $3 = ab$ 都有 $N(a)N(b) = N(3) = 9$. 如果 a, b 都不是可逆元，则 $N(a)$ 和 $N(b)$ 都大于 1，因而必然有 $N(a) = N(b) = 3$. 但是 $\mathbf{Z}[\sqrt{-5}]$ 上没有这样的元素，亦即 $x^2 + 5y^2 = 3$ 没有整数解. 这就说明 3 是不可约元. 至于 3 不是素元，就是要证明 3 不能整除 $2 \pm \sqrt{-5}$，这显然.

欧几里得整环

相关阅读：（扩展）欧几里得算法→9.11.4@p1188、裴蜀定理→9.11.8@p1218

欧几里得整环是允许做辗转相除法（即欧几里得算法）的整环.

欧几里得整环

对于整环 R，如果存在映射 $N : R \setminus \{0\} \to \mathbf{N}$，满足对于任意 $a, b \in R$ 且 $b \neq 0$，都存在 $q, r \in R$ 使得 $a = qb + r$ 成立且 $r = 0$ 或 $N(r) < N(b)$，则称整环 R 为 **欧几里得整环**（Euclidean domain, ED）. 映射 N 称为欧几里得整环中元素的范数（norm）.

其他等价定义

本文采用的定义仅仅在非零元素处定义了范数. 不同文本可能对欧几里得整环的定义有不同处理. 比如，有的文本可能补充定义 $N(0) = 0$；但是随后的带余除法中并没有用到 $N(0)$ 的值，所以这无关紧要. 再比如，Wikipedia[2] 的定义中还要求范数 N 满足性质：对于任何非零 $a, b \in R$ 都有 $N(a) \leq N(ab)$. 但是，容易验证，如果欧几里得整环 R 有范数 $N(\cdot)$ 满足本文定义所要求的性质，那么，可以定义范数 $N'(a) = \min_{b \in R \setminus \{0\}} N(ab)$ 使得它满足额外的性质 $N'(a) \leq N'(ab)$. 因此，这些不同的定义都是等价的.

这个定义其实就是整数中的带余除法的推广. 范数的存在使得能够衡量余数和除数的相对大小. 这样在辗转相除的时候，对应的余数的范数也在不断下降；因为范数取值在自然数上，这样的过程必然结束在 $r = 0$ 时. 这样，就得到了欧几里得整环上的辗转相除法.

能够做辗转相除法，这意味着欧几里得整环上能够高效地计算最大公因子. 完全类比整数的整除理论，可以证明，辗转相除法的结果一定是最大公因子，而且裴蜀定理成立，其中的系数可以通过扩展欧几里得算法确定.

定理

对于欧几里得整环 R 和它的元素 $a,b\in R$，对 a 和 b 做辗转相除法得到的结果 d 是 a 和 b 的最大公约数，且存在 $x,y\in R$ 使得 $d=ax+by$ 成立；反过来，任何 $ax+by$ 形式的元素都是 d 的倍数.

注意到，在环论的语言中，所有形如 $ax+by$ 的元素正是理想 (a,b) 中的元素，而这一定理就说明了 (a,b) 一定是主理想 (d).

其实，欧几里得整环中的理想一定是主理想.

定理

欧几里得整环中的理想一定是主理想.

证明

设 R 是欧几里得整环，且 I 是它的理想. 如果 $I=\{0\}$，它显然是主理想. 设 I 是非零理想. 依定义，环 R 上有范数 $N(\cdot)$，于是可以取 I 中范数最小的非零元素 d. 此时，对于任何 $a\in I$，都有 $a=qd+r$ 满足 $r=0$ 或 $N(r)<N(d)$. 又因为 $r=a-qd\in I$，所以依 d 的选取方式就可知 $r=0$，也就说 $a=qd\in(d)$. 这就说明 I 必然是主理想.

主理想整环

所有理想都是主理想的整环叫做主理想整环. 这是性质相当良好，也十分常见的一类整环. 在这些整环中，环中理想的概念就等同于整数中倍数的概念.

自 主理想整环

对于整环 R，如果它的每个理想都是主理想，则称它为 **主理想整环**（principal ideal domain, PID）.

因而，上一节最后一个定理就可以复述如下：

定理

欧几里得整环一定是主理想整环.

在主理想整环中，极大理想就等价于不可约元生成的理想. 类似整数中素数和不可约元是等价的，主理想整环中，这两个概念也是等价的，故而极大理想和素理想也是完全等价的.

定理

设主理想整环 R 有非零理想 I，则 I 是素理想，当且仅当 I 是极大理想.

证明

只需要证明素理想都是极大理想. 设主理想整环 R 中有非零素理想 (p)，且同时有理想 (a) 满足 $(p)\subseteq(a)\subseteq R$. 这说明 $a\mid p$，故而存在 $b\in R$ 使得 $p=ab$. 但由于 (p) 是素理想，$ab\in(p)$ 就意味着 $a\in(p)$ 或 $b\in(p)$. 如果 $a\in(p)$，就说明 $(a)\subseteq(p)$，故而 $(a)=(p)$；如果 $b\in(p)$，就说明 $b=cp$，故而 $p=acp$，又因 $p\neq0$，有 $1=ac$，即 a 存在逆元 c，于是 $(a)=R$. 这就说明，(p) 是极大理想.

推论

设主理想整环 R 有非零元素 r，则 r 是素元，当且仅当 r 是不可约元.

上一节中对裴蜀定理的分析可以迁移到主理想整环上.

定理

设 R 是主理想整环，且 $a,b\in R$ 是非零元素. 设 $d\in R$ 是理想 (a,b) 的生成元. 那么，a 和 b 的最大公因子是 d，且在相伴意义下唯一；而且，存在 $x,y\in R$ 使得 $ax+by=d$ 成立.

也就是说，主理想整环中 裴蜀定理→9.11.8@p1218 依然成立. 同样是存在最大公因子，欧几里得整环和主理想整环的最大区别在于在前者中，最大公因子可以通过辗转相除法高效地计算，但是主理想整环中一般并没有这样的高效算法.

唯一分解整环

比主理想整环更一般的概念是唯一分解整环. 整数的唯一分解定理称为 算术基本定理→9.11.1@p1160. 类似的唯一分解定理其实在一些并非主理想整环的整环中依然成立. 这样的整环叫做唯一分解整环.

自 唯一分解整环

对于整环 R，如果任何非零且不可逆的元素 r 都能写作 $r=p_1\cdots p_n$ 的形式，这里的 $p_1,\cdots,p_n$ 是可能重复的不可约元，且这样的分解在相伴和重新排列的意义下唯一，则称整环 R 是 **唯一分解整环**（unique factorization domain, UFD）.

算术基本定理说明，整数环 $\mathbf{Z}$ 是唯一分解整环.

前文给出了不可约元不是素元的反例，其中涉及的整环 $\mathbf{Z}[\sqrt{-5}]$ 中唯一分解定理不再成立. 但是，在所有唯一分解整环上，不可约元和素元都是等价的.

定理

对于唯一分解整环 R 和它的非零元素 $a\in R$，则 a 是素元，当且仅当 a 是不可约元.

证明

只需要证明不可约元都是素元. 对于不可约元 r，如果 $r\mid ab$，那么就存在 $c\in R$ 使得 $ab=rc$ 成立. 因为 R 是唯一分解整环，所以可以对 $a,b,c\in R$ 都做分解成不可约元的乘积. 比较左右两边，根据分解的唯一性可知，r 必然和 a 或者 b 的某个不可约因子相伴，故而 r 整除 a 或 b 中的一个. 这就说明 r 也是素元.

所有的主理想整环都是唯一分解整环.

定理

主理想整环一定是唯一分解整环.

证明

设 R 是主理想整环，且 $r\in R$ 不是零元，也不是可逆元. 要说明 r 可以唯一分解为一系列不可约元的乘积，可以分为两步：首先证明分解的存在性，再证明分解的唯一性.

分解的存在性比较自然. 如果 r 已经是不可约元，就不必继续分解；否则，必然存在 r_1,r_2 使得 $r=r_1r_2$ 且 r_1,r_2 都不是可逆元. 进而，如果 r_1 和 r_2 都是不可约元，那么也不必继续分解；否则，对 r_1 和 r_2 中不是不可约元的，可以进一步分解，r 也就可以写成更多元素的乘积. 由此，只要乘积中不全是不可约元，就可以将分解过程不断进行下去. 分解必然在有限步后终止. 不然，选择公理保证可以从 R 中取出无限长的元素链 $\{r_{(i)}\}_{i=0}^{\infty}$ 满足 $r_{(0)}=r$ 且 $r_{(i+1)}\mid r_{(i)}$ 对所有 $i\in\mathbf{N}$ 都成立，且这些整

除关系都是严格的，即链中不存在相伴元．用理想的语言说，这对应着严格无穷递增的理想列：$I_0 \subset I_1 \subset \cdots \subset I_i \subset \cdots \subset R$，其中，$I_i = (r_{(i)})$．容易验证，这些理想的并 $I = \bigcup_{i=0}^{\infty} I_i$ 还是理想，因而必然是主理想．令 a 为主理想 I 的生成元，因而，存在 $n \in \mathbf{N}$ 满足 $a \in I_n$．所以，$I = (a) \subseteq I_n$．这说明，这个严格无穷递增的理想列并不存在，故而上述分解过程必然在有限步内终止．

然后证明分解的唯一性．可以对分解中因子的个数做归纳．归纳的关键步骤在于验证，如果 $r = p_1 p_2 \cdots p_n = q_1 q_2 \cdots q_m$ 且 $n \leq m$，则必然有 p_1 与某个 q_j 相伴．这里需要用到之前的结论：主理想整环中不可约元都是素元．已知 p_1 是 R 中的不可约元，故而它也是素元，所以对右侧的乘积可以归纳地说明，必然存在某个元素 q_j 使得 $p_1 \mid q_j$．所以，存在 $c \in R$ 使得 $q_j = p_1 c$，而 q_j 是不可约元，p_1 也是不可约元，依定义只能有 c 是可逆元，故而 p_1 与 q_j 相伴．这样就可以利用消去律在左右两侧分别消去 p_1 和 q_j，并将两者相差的相伴元乘到任意一个剩余元素上．根据归纳假设，必然有 $p_2 \cdots p_n$ 和 $q_1 \cdots q_{j-1} q_{j+1} \cdots q_m$ 中不可约元个数相等，且在相伴意义下是一样的．定理得证．

最后，最大公因子的存在性在唯一分解整环上依然成立．

定理

设唯一分解整环 R 有非零元素 $a, b \in R$，且它们可以分解成 $a = u p_1^{r_1} \cdots p_n^{r_n}$ 和 $b = v p_1^{s_1} \cdots p_n^{s_n}$ 的形式，其中，u, v 是可逆元，$p_1, \cdots, p_n$ 是各不相同的不可约元，r_i, s_i 都是自然数，那么，它们的一个最大公约数是 $d = p_1^{\min\{r_1, s_1\}} \cdots p_n^{\min\{r_n, s_n\}}$．

这其实说明，最大公因子存在这个性质比唯一分解定理成立还要弱[14]．

例子：二次整数环

相关阅读：二次域→9.11.32@p1495

抽象代数的理解不能离开例子．正是因为费马大定理的研究过程需要研究一类代数整数的性质，才逐渐发展出了今天的环论[15]．这里讨论最简单的代数整数，即二次整数．这部分的很多结论的证明需要用到复杂的代数数论知识，故而略去．

二次整数（quadratic integer）指的是二次项系数为一的整系数二次方程 $\alpha^2 + b\alpha + c = 0$ 的复根．所有二次整数能且仅能有形式

$$\alpha = a + b\omega, \ (a, b \in \mathbf{Z})$$

这里，

$$\omega = \begin{cases} \dfrac{1+\sqrt{D}}{2}, & D \equiv 1 \pmod 4, \\ \sqrt{D}, & D \equiv 2, 3 \pmod 4, \end{cases}$$

其中，D 没有平方因子．

分析

根据二次方程求根公式，可以知道这个方程的根一定可以写成

$$\alpha = \frac{-b \pm \sqrt{b^2 - 4c}}{2}.$$

当 $b = 2k + 1$ 是奇数时，这个根可以写作

$$\alpha = -k - \frac{1 \pm \sqrt{4(k^2 + k - c) + 1}}{2}.$$

[14] 最大公因子存在的整环叫做 最大公因子整环[3].
[15] 环论的简要历史可以参看 这里[4].

否则，当 $b=2k$ 是偶数时，这个根可以写作

$$\alpha=-k\pm\sqrt{k^2-c}.$$

从而可以归纳得知二次整数必然有上述形式.

容易验证, 对于这样的 ω, 集合 $\mathbf{Z}[\omega]=\{a+b\omega:a,b\in\mathbf{Z}\}$ 构成环. 这称为 **二次整数环** (quadratic integer ring)，它的分式域就是二次域 $\mathbf{Q}(\sqrt{D})$. 当 $D>0$ 时，所有二次整数都是实数，故而也称作 **实二次整数环**; 当 $D<0$ 时，除了整数外的二次整数都是复数，故而也称作 **虚二次整数环**.

所有二次整数环 $\mathbf{Z}[\omega]$ 都是整环. 其中，当 $D=-1$ 时，$\mathbf{Z}[\sqrt{-1}]$（或记作 $\mathbf{Z}[\mathrm{i}]$）也称 Gauss 整数环；当 $D=-3$ 时，$\mathbf{Z}\left[\dfrac{1+\sqrt{-3}}{2}\right]$ 则称为 Eisenstein 整数环.

对于二次整数 $a+b\omega$，可以定义它的 **共轭**（conjugate）是 $a+b\bar{\omega}$，其中，

$$\bar{\omega}=\begin{cases}\dfrac{1-\sqrt{D}}{2}, & D\equiv 1 \pmod 4,\\ -\sqrt{D}, & D\equiv 2,3 \pmod 4,\end{cases}$$

注意，因为 $D>0$ 时，二次整数是实数，所以这里的共轭的概念和复数的共轭的概念并不是完全一致的，但它们都是域论中代数元的共轭的概念的特例. 共轭的二次整数是同一个整系数二次方程的根.

在二次整数环上可以定义 **范数**

$$\begin{aligned}N(a+b\omega)&=(a+b\omega)(a+b\bar{\omega})\\&=\begin{cases}a^2+ab+\dfrac{1-D}{4}b^2, & D\equiv 1 \pmod 4,\\ a^2-Db^2, & D\equiv 2,3 \pmod 4.\end{cases}\end{aligned}$$

二次整数的范数一定是整数. 特别地，当 $D<0$ 时，范数一定是自然数. 范数保持乘法结构，即 $N(ab)=N(a)N(b)$.

二次整数环中的可逆元（单位）能且仅能是那些范数是 ±1 的元素. 对于 $D>0$ 的情形，这就相当于考虑 Pell 方程→9.11.33@p1503 $x^2-Dy^2=\pm1$ 或 $x^2-Dy^2=\pm4$ 的解. 对于 $D<0$ 的情形，容易验证，除了 Gauss 整数环 $\mathbf{Z}[\mathrm{i}]$ 中可逆元是 $\{\pm1,\pm\mathrm{i}\}$ 和 Eisenstein 整数环 $\mathbf{Z}[\omega]$ 中可逆元是 $\{\pm1,\pm\omega,\pm\omega^2\}$ 这两种特殊情形外，其余的可逆元都只有 $\{\pm1\}$.

二次整数环上定义的范数 $N(\alpha)$ 可以用来证明它是欧几里得整环. 对于 $D>0$ 的情形，需要使用它的绝对值 $|N(\alpha)|$ 来作为欧几里得整环定义中的范数. 利用这样得到的范数，能够证明在 $D<0$ 时，

$$D=-1,-2,-3,-7,-11$$

或者在 $D>0$ 时，

$$D=2,3,5,6,7,11,13,17,19,21,29,33,37,41,57,73$$

这些整数对应的二次整数环是在模 $|N(\cdot)|$ 下的欧几里得整环. 但是，欧几里得整环定义中的范数未必是上述定义的范数. 比如在 $D=14,69$ 时，相应的二次整数环也是欧几里得整环，但是需要用到别的范数. 对于 $D<0$ 的情形，可以证明上面给出的情形就是二次整数环中所有的欧几里得整环.

利用更为复杂的方法，还可以判断某个二次整数环是否是主理想整环. 可以证明当 $D<0$ 时，只有

$$D=-1,-2,-3,-7,-11,-19,-43,-67,-163$$

对应的二次整数环是主理想整环. 比较上面的结果，可以知道诸如 $D=-19$ 的情形提供了主理想整环不是欧几里得整环的例子. 当 $D>0$ 时，目前尚没有完整的结果.

但是，可以证明，在二次整数环中，唯一分解整环和主理想整环是等价的. 上面的结果说明，比如说 $\mathbf{Z}[\sqrt{-5}]$ 就不是主理想整环，因而也不是唯一分解整环. 之前已经通过例子实际证明过了它不能唯一分解，即

$$9=3\times3=(2+\sqrt{-5})\times(2-\sqrt{-5}).$$

利用同样的例子，可以说明理想 $(3, 2+\sqrt{-5})$ 也不是主理想．稍后会看到，是唯一分解整环但不是主理想整环的一个简单例子是多项式环 $\mathbf{Z}[x]$.

尽管很多二次整数环并不是唯一分解整环，但是它们都是 Dedekind 整环[5]．这意味着，所有二次整数环中的非平凡理想都可以唯一分解为一系列素理想之积．但如果二次整数环本身并非主理想整环，这些素理想因子并不一定对应着素元，因而唯一分解定理（即数分解成素数的乘积）不再成立：这也是研究理想而不是数的最初动机．

多项式环

相关阅读：多项式技术简介→9.12.1@p1519

在算法竞赛中，时常会遇到多项式的各种运算．多项式的乘法、取逆、取余等运算可以看作数的运算在多项式环上的推广．利用抽象代数的语言，可以更快地理解多项式环上相关运算的性质．

自 多项式

对于非零交换幺环 R，一个 R 上的 **多项式**（polynomial）是指形式和

$$\sum_{k=0}^{n} a_k x^k = a_0 + a_1 x + \cdots + a_{n-1} x^{n-1} + a_n x^n,$$

其中，$n \in \mathbf{N}$，且对于每个 k，都有 $a_k \in R$．这些 a_k 称为多项式的 **系数**（coefficient），相应的 $a_k x^k$ 称为多项式的 **项**（term）．项 $a_k x^k$ 中的 k 称为该项的 **次数**（degree）．

所有系数都为零（即零元）的多项式称为 **零多项式**（zero polynomial），记作 0．对于其它多项式，不妨设 $a_n \neq 0$，即 $a_n x^n$ 是系数不为零的项中次数最高的项．此时，自然数 n 称为多项式的 **次数**（degree），而它所在的项 $a_n x^n$ 称为 **最高次项**（leading term），a_n 也称为 **最高次项系数**（leading coefficient）．最高次项系数等于一（即幺元）的多项式称为 **首一**（monic）多项式．零多项式的次数不予指定，或者规定为 $-\infty$.

多项式记号中出现的 x 称为多项式的 **不定元**（indeterminate）．它本身没有任何含义，也没有取值范围．它的存在，仅仅是通过它的指数标记系数的位置．所以，多项式也可以写作 R 上的数列

$$(a_0, a_1, \ldots, a_{n-1}, a_n, 0, 0, \cdots).$$

但是，这样的数列只能出现有限多个不为零的项．如果两个多项式对应的系数数列一样，就称两个多项式相等．这相当于在补齐系数为零的项后，它们的形式和完全一致．下文中，不再区分相等的多项式的形式和的记号：如果必要，读者可以自行补齐系数中空缺的零.

有时候需要将环中的元素代入多项式中的不定元．比如设 $f(x)$ 是 R 的多项式且 $a \in R$，则将 a 代入多项式 $f(x)$ 的结果就是 $f(a)$．它的含义是：在多项式的形式和中，将 x 替换成 a，就能得到 R 中的算术表达式，而 $f(a)$ 就是这个表达式在 R 中运算得到的结果.

「多项式」和「多项式函数」

读者不应混淆这两个概念．多项式只是有限长的系数数列，它并不自动成为函数．尽管这里将环中元素代入不定元的操作确实将多项式映射为多项式函数，但是这样的映射未必是单射．比如，$f(x) = x^p - x$ 作为域 $\mathbf{F}_p$ 上的多项式，显然不等于零多项式；但是 $f(x)$ 作为多项式函数 $\mathbf{F}_p \to \mathbf{F}_p$，恒等于零（即费马小定理）．虽然两者概念不同，很多多项式函数的概念都可以推广到多项式的情形，比如可以仿照多项式函数的微分、不定积分、复合等定义多项式的（形式的）导数→9.12.1@p1519、不定积分→9.12.1@p1519、复合→9.12.1@p1519 等．这些形式运算并不依赖于任何拓扑结构，但是仍然成立很多运算法则.

对于多项式

$$f(x) = a_0 + a_1 x + \cdots + a_{n-1}x^{n-1} + a_n x^n,$$
$$g(x) = b_0 + b_1 x + \cdots + b_{n-1}x^{n-1} + b_n x^n,$$

多项式的加法运算定义为

$$f(x) + g(x) = (a_0 + b_0) + (a_1 + b_1)x + \cdots + (a_{n-1} + b_{n-1})x^{n-1} + (a_n + b_n)x^n,$$

而多项式的乘法运算定义为

$$f(x)g(x) = a_0 b_0 + (a_1 b_0 + a_0 b_1)x + (a_2 b_0 + a_1 b_1 + a_0 b_2)x^2 + \cdots,$$

其中，x^k 项的系数为 $\sum_{i=0}^{k} a_{k-i} b_i$. 在这样定义的加法和乘法运算下，可以证明，$R$ 上的全体多项式构成环，记作 $R[x]$.

多项式 $f(x)$ 的次数记作 $\deg f(x)$. 那些次数为零的多项式是常数多项式，它们以及零多项式相当于 R 在 $R[x]$ 中的嵌入. 显然，R 有零因子当且仅当 $R[x]$ 有零因子.

定理

多项式环 $R[x]$ 是整环，当且仅当 R 是整环.

整环 R 上的多项式环 $R[x]$ 中，加法和乘法的结果满足

$$\deg(f(x) + g(x)) \le \max\{\deg f(x), \deg g(x)\},$$
$$\deg(f(x)g(x)) = \deg f(x) + \deg g(x).$$

这里设 $\deg 0 = -\infty$. 所以，多项式环中的可逆元也一定是它的常数多项式中的那些可逆元. 任何一次及以上的多项式都不是可逆的.

下文的讨论将仅限于整环上的多项式.

约定

下文中，将不加区分地使用「环 R 上的多项式」和「多项式环 $R[x]$ 中的多项式」两种说法. 比如，多项式在环 R 上不可约，就是指多项式在环 $R[x]$ 中不可约. 而且，如果 R 是 S 的子环，那么 R 上的多项式将自动成为 S 上的多项式；对此也不再多加说明.

域上的多项式环

整环上的多项式环中性质最为简单的，当然是域上的多项式环. 域 F 上的多项式环 $F[x]$ 因为系数可以做除法，所以可以定义带余除法. 不妨设非零多项式 $f(x)$ 的范数 $N(f(x)) = \deg f(x)$. 那么，对于 $F[x]$ 中的多项式 $f(x)$ 和非零多项式 $g(x)$，显然可以做带余除法

$$f(x) = g(x)q(x) + r(x),$$

其中，$q(x), r(x) \in F[x]$，且 $r(x) = 0$ 或 $\deg r(x) < \deg g(x)$. 这说明，域上的多项式环都是欧几里得整环.

定理

域 F 上的多项式环 $F[x]$ 是欧几里得整环，也是主理想整环，也是唯一分解整环.

算法竞赛中，由于计算精度原因，常常考虑的是多项式环 $\mathbf{F}_p[x] = (\mathbf{Z}/p\mathbf{Z})[x]$，此时的模数 p 要求是质数. 这样的环容许辗转相除法等操作. 但是，任意模数 n 对应的多项式环 $(\mathbf{Z}/n\mathbf{Z})[x]$ 甚至都不是整环.

成立带余除法意味着多项式的根总对应着它的一个一次因子.

根

多项式 $f(x)$ 的 **根**（root）指的是使得 $f(\xi) = 0$ 成立的元素 $\xi \in F$.

定理

对于域 F 上的多项式 $f(x)$ 和域中的元素 $\xi \in F$，那么 ξ 是 $f(x)$ 的根，当且仅当 $f(x)$ 有一次因子 $(x-\xi)$.

证明

带余除法说明存在 $q(x), r(x)$，成立 $f(x)=q(x)(x-\xi)+r(x)$ 且 $\deg r(x)<\deg(x-\xi)=1$. 因而，$r(x)$ 是常数多项式或者零多项式，令 $r(x)=c$，则必然有 $f(x)=q(x)(x-\xi)+c$. 代入 $x=\xi$，故而有 $0=f(\xi)=c$，即 $f(x)=q(x)(x-\xi)$.

根的概念可以推广到重根的情形.

自 重根

如果多项式 $f(x)$ 有因子 $(x-\xi)^k$，且 $(x-\xi)^{k+1}$ 不能整除 $f(x)$，则称 ξ 是 $f(x)$ 的 k **重根**（root of multiplicity k）. 如果 $k>1$，则根 ξ 称为 $f(x)$ 的 **重根**（multiple root）；如果 $k=1$，则根 ξ 称为 $f(x)$ 的 **单根**（simple root）.

定理

如果域 F 上的多项式 $f(x)$ 有（可能重复的）根 $\xi_1, \cdots, \xi_k$，那么，它必然有因子 $(x-\xi_1)\cdots(x-\xi_k)$. 进而，域 F 上的多项式 $f(x)$ 次数为 n，那么它至多有 n 个根（计重数）.

证明

注意到 $F[x]$ 是唯一分解整环即可.

虽然域上的多项式成立唯一分解定理，但是并没有一般的办法判断给定的多项式是否可约. 次数比较小的情形相对容易. 比如说，所有的一次多项式都是不可约多项式. 在特殊的域上，所有的不可约多项式都是一次多项式. 这样的域称为 代数闭域→9.16.4@p1849. 在这样的域上，所有次数大于零的多项式都有根，因而任何大于一次的多项式都可以进一步分解. 一个这样的例子是复数域 $\mathbf{C}$. 而实数域 $\mathbf{R}$ 上，则存在二次的不可约多项式；有理数域 $\mathbf{Q}$ 上，不可约多项式的结构就更为复杂. 域论→9.16.4@p1849 页面对于有理数域和有限域上的多项式有更多的讨论.

以上结论都是关于域上的多项式. 更一般的整环上的多项式，常常可以转化为这样的情形.

下面考虑唯一分解整环 R 上的多项式环 $R[x]$. 直接在 $R[x]$ 中做运算，因为系数时常不能做除法，很多运算受到限制. 不妨考虑将 R 扩充到它的分式域 F，进而考虑将 $R[x]$ 中的多项式 $f(x)$ 在 $F[x]$ 中做分解. 已知 $F[x]$ 是唯一分解整环，那就可以通过 $F[x]$ 中 $f(x)$ 的分解反推出 $R[x]$ 中的分解. 幸而这样的想法总是可行的.

Gauss 引理

对于唯一分解整环 R 和它的分式域 F，如果 $f(x)\in R[x]$，那么如果在 $F[x]$ 中 $f(x)=A(x)B(x)$，那么必然存在 $s,t\in F$，使得 $a(x)=sA(x)\in R[x]$，$b(x)=tB(x)\in R[x]$，且 $f(x)=a(x)b(x)$. 因此，如果次数大于零的 $f(x)$ 在 $R[x]$ 中不可约，那么它在 $F[x]$ 中不可约.

证明

设 $f(x) \in R[x]$ 在 $F[x]$ 中可约，且 $f(x) = A(x)B(x)$．设 r_a 和 r_b 分别为 $A(x)$ 和 $B(x)$ 中所有系数的分母的最小公倍数，则有 $\tilde{a}(x) = r_a A(x)$ 和 $\tilde{b}(x) = r_b B(x)$ 都是 R 上多项式．令 $r = r_a r_b$，就有 $rf(x) = \tilde{a}(x)\tilde{b}(x)$．如果 r 是 R 中可逆元，则可以取分解 $f(x) = (r^{-1}\tilde{a}(x))\tilde{b}(x)$，显然满足引理的要求．

否则，如果 r 中存在不可约元因子 p，这里要证明，等式两侧可以消去这个因子，且保证所有系数仍旧在整环 R 中．注意到 p 必然也是素元，因而 (p) 是素理想．等式左右两边同时模去 p，则得到 $(R/(p))[x]$ 上的多项式 $0 = \bar{a}(x)\bar{b}(x)$，这里，$\bar{a}$ 和 $\bar{b}$ 是取模后的多项式．因为 $R/(p)$ 是整环，$(R/(p))[x]$ 也是整环，故而可以设 $\bar{a}(x) = 0$．这说明，p 可以整除 $\tilde{a}(x)$ 的全部系数．因而，等式两侧可以直接消去因子 p．

根据唯一分解整环的定义，r 至多有有限个这样的不可约元因子，故而有限次消去它们后就转化为了 r 是 R 中可逆元的情形．引理就得以证明．

推论

对于唯一分解整环 R 和它的分式域 F，如果 $f(x) \in R[x]$ 且 $f(x)$ 的所有系数互素（即最大公因子是 R 中幺元），则 $f(x)$ 在 $R[x]$ 中不可约，当且仅当 $f(x)$ 在 $F[x]$ 中不可约．

也就是说，整系数多项式环 $\mathbf{Z}[x]$ 中的不可约元都是 $\mathbf{Q}[x]$ 中的不可约元．判断整系数多项式是否不可约的一个有效方法是 Eisenstein 判别法．根据 Gauss 引理，它也提供了判断有理系数多项式是否不可约的方法．

Eisenstein 判别法

设 n 次整系数多项式 $f(x) = a_0 + a_1 x + \cdots + a_{n-1}x^{n-1} + a_n x^n$，如果存在质数 p 满足 $p \mid a_i$ 对所有 $i = 0, 1, \cdots, n-1$ 都成立，且 p 不能整除 a_n，p^2 不能整除 a_0，则多项式 $f(x)$ 在有理数域 $\mathbf{Q}$ 上不可约．如果 $\gcd(a_0, a_1, \cdots, a_n) = 1$，则多项式 $f(x)$ 也在整数环 $\mathbf{Z}$ 上不可约．

证明

利用 Gauss 引理可知，如果多项式 $f(x)$ 在有理数域 $\mathbf{Q}$ 上可约，则它也在整数环 $\mathbf{Z}$ 上可约．设 $f(x) = b(x)c(x)$ 是它在 $\mathbf{Z}[x]$ 中的分解．将等式的左右两边对质数 p 取模，则得到 $\mathbf{F}_p[x]$ 中的分解，$\overline{f}(x) = \overline{b}(x)\overline{c}(x)$．但是定理的条件说明，$\overline{f}(x) = x^n$，则必然存在整数 m 使得 $\overline{b}(x) = x^m$ 且 $\overline{c}(x) = x^{n-m}$，其中，$0 < m < n$．故而，因子 $b(x)$ 和 $c(x)$ 的常数项 b_0 和 c_0 都是 p 的倍数．故而，$f(x)$ 的常数项 $a_0 = b_0 c_0$ 必然是 p^2 的倍数．这与所给条件相矛盾．

例子

1. 多项式 $x^3 - 2$ 在 $\mathbf{Q}[x]$ 中不可约．对 $p = 2$ 应用 Eisenstein 判别法即可．
2. 多项式 $x^4 + 1$ 在 $\mathbf{Q}[x]$ 中不可约．否则，$(x+1)^4 + 1 = x^4 + 4x^3 + 6x^2 + 4x + 2$ 也可约．但是，对 $p = 2$ 应用 Eisenstein 判别法可知，后者并不可约．

对于唯一分解整环 R，因为相应的分式域 F 上的多项式环是唯一分解整环，而 Gauss 引理说明，分式域 F 上的多项式和原来的整环 R 上的多项式的分解是相互对应的，所以 $R[x]$ 也是唯一分解整环．因此，有如下定理：

定理

多项式环 $R[x]$ 是唯一分解整环，当且仅当 R 是唯一分解整环.

这里的 $\mathbf{Z}[x]$ 提供了唯一分解整环不一定是主理想整环的例子. 例如，在 $\mathbf{Z}[x]$ 中，$(2,x)$ 并不是主理想.

有很多方法可以将多项式环扩充到更大的集合. 比如说，对于整环上的多项式环 $R[x]$，可以将它扩充到它的分式域，记作 $R(x)$. 这个分式域常称作 **有理分式域**（field of rational fractions），其中的元素的基本形式为 $\dfrac{f(x)}{g(x)}$，这里，$f(x)$ 和 $g(x)$ 都是多项式.

多元多项式环

多项式环可以推广到含有多个不定元的情形. 对于交换幺环 R，可以定义 R 上的多项式环，即一元多项式环 $R[x]$. 进而，可以定义 $R[x]$ 上的多项式环 $R[x][y]$，它可以看作是 R 上的二元多项式环 $R[x,y]$. 由此，可以归纳地定义 R 上的 k 元多项式环 $R[x_1,\cdots,x_k]$. 当 R 是整环的时候，它上面的任意多元多项式环都是整环；类似地，唯一分解整环的性质也可以传递到任意多元多项式环.

形式幂级数环

也可以考虑形式和中可以有任意多项系数不为零的情形. 交换幺环 R 上的 **形式幂级数**（formal power series）定义为

$$\sum_{k=0}^{\infty} a_k x^k = a_0 + a_1 x + a_2 x^2 + \cdots.$$

利用与多项式环 $R[x]$ 一致的方式可以定义幂级数间的加法和乘法运算. 并且，此时的形式幂级数也构成环，记作 $R[[x]]$. 这里的形式幂级数并不需要考虑其敛散性，因为实际上每个形式幂级数只是它的系数序列，而并没有赋予更多的拓扑结构.

形式幂级数环的结构很有趣. 整环上的多项式环中，可逆元只能是常数. 但是，在形式幂级数环中，却可以有

$$(1-x)^{-1} = \sum_{k=0}^{\infty} x^k = 1 + x + x^2 + \cdots.$$

这个现象是普遍的. 只要一个形式幂级数的常数项 a_0 是 R 中的可逆元，就一定有 $\sum_{k=0}^{\infty} a_k x^k$ 也是可逆的. 这是因为如果设

$$\left(\sum_{k=0}^{\infty} a_k x^k\right)\left(\sum_{k=0}^{\infty} b_k x^k\right) = 1,$$

那么，列出系数需要满足的方程组，可以递归地求得 b_k 的表达式，其中只涉及到 a_0 的逆.

在形式幂级数环上可以定义各种运算，诸如取逆、除法、复合逆、形式导数、初等函数等，详见 多项式技术简介→9.12.1@p1519.

形式洛朗级数环

形式幂级数环还可以进一步拓展，使得它允许负次数的项. 交换幺环 R 上的 **形式洛朗级数**（formal Laurent series）定义为

$$\sum_{k=N}^{\infty} a_k x^k,$$

这里，$N \in \mathbf{Z}$. 因此，形式洛朗级数可以有有限多个负次数的项. 将之前的加法和乘法拓展到形式洛朗级数上，就能得到形式洛朗级数环，记作 $R((x))$. 如果 F 是域，那么 $F((x))$ 也是域.

形式洛朗级数环在 Lagrange 反演→9.12.13@p1587 中有应用.

中国剩余定理

相关阅读：中国剩余定理→9.11.12@p1236

在数论中，中国剩余定理常用来求解数论方程组. 对于一般的交换幺环，同样可以建立中国剩余定理. 每个同余方程都相当于指定了未知元在某个商环里的像，那么，交换幺环中的中国剩余定理就相当于通过这些商环里的像确定环中的元素.

这个讨论可以转化为形式语言. 对于非零交换幺环 R 和它的理想 $I_1,\cdots,I_n$，考虑环同态 $\varphi:R\to R/I_1\times\cdots\times R/I_n$，它将 r 映射至 $(r+I_1,\cdots,r+I_n)$. 其中，$r+I_i$ 是陪集，而 $\times$ 表示环的直积：

自 直积

对于环 R_1 和 R_2，它们的加法群的直积 $R_1\times R_2$ 上可以定义乘法为各个分量分别相乘，则 $R_1\times R_2$ 就成为环，称为环 R_1 和 R_2 的 **直积**（direct product），仍记作 $R_1\times R_2$.

同态 φ 的核是 $\ker\varphi=I_1\cap\cdots\cap I_n$. 中国剩余定理要回答的问题就是这样的映射在什么条件下是满射.

在数论的情形下，定理的成立需要这些模数互质. 这个条件可以推广到环论的情形.

自 互素

设环 R 有理想 I 和 J，如果 $I+J=R$，则称 I 和 J **互素**（comaximal）.

对于幺环的情形，如果考虑主理想 (a) 和 (b)，这个条件就相当于存在 $x,y\in R$ 使得 $ax+by=1$，这类似于整数互素时的裴蜀定理. 利用这个定义，可以完全仿照整数环的情形，建立交换幺环上的 **中国剩余定理**（Chinese remainder theorem）.

中国剩余定理

设非零交换幺环 R 有理想 $I_1,\cdots,I_n$. 如果它们两两互素，那么上述定义的环同态 φ 是满射，它的核等于这些理想的乘积 $\ker\varphi=I_1\cap\cdots\cap I_n=I_1\cdots I_n$，因此，

$$R/(I_1\cdots I_n)=R/(I_1\cap\cdots\cap I_n)\cong R/I_1\times\cdots\times R/I_n.$$

证明

定理内容很丰富，但仍需证明的结论只有两个，即 φ 是满射和 $I_1\cap\cdots\cap I_n=I_1\cdots I_n$. 关键在于利用好互素的条件.

首先证明 $n=2$ 的情形. 因为理想 I_1 和 I_2 互素，即 $I_1+I_2=R$，所以，R 中幺元 1 可以写成 a_1+a_2 的形式，其中，$a_i\in I_i$. 因为 $a_1\in I_1$ 且 $a_1=1-a_2\in 1+I_2$，所以 $\varphi(a_1)=(I_1,1+I_2)$；同理，$\varphi(a_2)=(1+I_1,I_2)$. 因而，$(\varphi(a_2),\varphi(a_1))$ 起到了类似向量空间中的「基」的作用. 故而，对任意像 (r_1+I_1,r_2+I_2)，都能找到同态 φ 下的原像 $r_1a_2+r_2a_1$. 这说明 φ 是满射.

还需要证明 $I_1\cap I_2=I_1I_2$. 对于一般的环总有 $I_1I_2\subseteq I_1\cap I_2$，关键在于其反面. 对于任意 $r\in I_1\cap I_2$，都有 $r=r(a_1+a_2)=ra_1+ra_2\in I_1I_2$. 故而也成立 $I_1\cap I_2\subseteq I_1I_2$. 所以，所求得证.

对于 $n>2$ 的情形，需要使用数学归纳法. 归纳步骤的关键在于证明对于两两互素的理想 $I_1,\cdots,I_n$ 总有理想 I_1 和 $I_2\cdots I_n$ 互素. 由于 I_1 与 $I_2,\cdots,I_n$ 都互素，故而对于每个 $i=2,\cdots,n$ 都存在 $a_i\in I_1$ 和 $b_i\in I_i$ 使得 $1=a_i+b_i$ 成立. 因而，有 $1=(a_2+b_2)\cdots(a_n+b_n)$ 成立. 所以，$1\in(b_2\cdots b_n)+I_1\subseteq I_1+(I_2\cdots I_n)$. 这说明，理想 I_1 和 $I_2\cdots I_n$ 互素.

应用：Lagrange 插值公式

相关阅读：Lagrange 插值→9.19.1@p1930、多项式快速插值→9.12.8@p1558

插值（interpolation）问题是指，给定一系列点值 $\{(x_i, y_i)\}_{i=1}^n$，寻找域 F 上的多项式 $f(x)$ 使其满足 $f(x_i) = y_i$ 对所有 $i = 1, \cdots, n$ 都成立．当然假设所有 x_i 互不相同．Lagrange 插值公式给出了这类问题的通解．

对于域 F 上的多项式 $f(x)$，条件 $f(x_i) = y_i$ 等价于 x_i 是多项式 $f(x) - y_i$ 的一个根，因而等价于 $(x - x_i) \mid (f(x) - y_i)$，也就是 $f(x) \equiv y_i \pmod{x - x_i}$．所以，插值问题就等价于求解同余方程组

$$\begin{cases} f(x) \equiv y_1 \pmod{x - x_1}, \\ f(x) \equiv y_2 \pmod{x - x_2}, \\ \cdots \\ f(x) \equiv y_n \pmod{x - x_n}. \end{cases}$$

这些一次多项式 $\{x - x_i\}_{i=1}^n$ 两两互质．根据中国剩余定理可知，问题的解应当具有形式

$$f(x) = \sum_{i=1}^n y_i M_i(x),$$

这里，$M_i(x) = m_i(x) \prod_{j \neq i} (x - x_j)$ 且 $M_i(x) \equiv 1 \pmod{x - x_i}$．根据前文推得的等价性可知，这等价于 $M_i(x_i) = 1$，亦即

$$m_i(x_i) \prod_{j \neq i} (x_i - x_j) = 1.$$

不妨取 $m_i(x)$ 是常数多项式，即

$$m_i(x) = \frac{1}{\prod_{j \neq i} (x_i - x_j)}.$$

由此，就得到 Lagrange 插值公式

$$f(x) = \sum_{i=1}^n y_i \frac{\prod_{j \neq i} (x - x_j)}{\prod_{j \neq i} (x_i - x_j)}.$$

一般地，将这种方法推广，还可以导出 Hermite 插值公式[6]，它允许限制多项式在各点处的若干项导数值．

应用：整数同余类的乘法群

相关阅读：原根→9.11.18@p1275、有限生成 Abel 群基本定理→9.16.2@p1812

作为中国剩余定理和群论相关内容的一个应用，这里讨论整数模 n 乘法群的结构．本节略去同余类的横线记号．

整数模 n 乘法群（multiplicative group of integers modulo n）指的是 $(\mathbf{Z}/n\mathbf{Z})^\times$，即商环 $\mathbf{Z}/n\mathbf{Z}$ 中的可逆元的乘法群（也称单位群）．群 $(\mathbf{Z}/n\mathbf{Z})^\times$ 的阶是 $\varphi(n)$，因为存在逆元的充要条件就是与 n 互质．这里的 $\varphi(n)$ 是 欧拉函数→9.11.5@p1196．而且，群 $(\mathbf{Z}/n\mathbf{Z})^\times$ 总是 Abel 群．

根据算术基本定理，模数 n 可以分解为不同的质数的幂的乘积：

$$n = p_1^{\alpha_1} \cdots p_s^{\alpha_s}.$$

容易验证，对于整数环的理想，理想互素的条件等价于理想的生成元互素．所以，应用中国剩余定理可以得到

$$\mathbf{Z}/n\mathbf{Z} \cong \mathbf{Z}/p_1^{\alpha_1}\mathbf{Z} \times \cdots \times \mathbf{Z}/p_s^{\alpha_s}\mathbf{Z}.$$

环的同构意味着相应的乘法结构也同构，所以

$$(\mathbf{Z}/n\mathbf{Z})^\times \cong (\mathbf{Z}/p_1^{\alpha_1}\mathbf{Z})^\times \times \cdots \times (\mathbf{Z}/p_s^{\alpha_s}\mathbf{Z})^\times.$$

这说明 $\varphi(n) = \varphi(p_1^{\alpha_1}) \cdots \varphi(p_s^{\alpha_s})$，即欧拉函数是积性函数．

因此，要研究一般的模数的情形，只要考虑素数幂 p^k 作为模数的情形就可以了．对于素数幂的情形，需要分别考虑 $p = 2$ 和 p 为奇素数的两种情形：

- 对于 $p=2$ 的情形，直接验证可知 $(\mathbf{Z}/2\mathbf{Z})^\times \cong C_1$ 和 $(\mathbf{Z}/4\mathbf{Z})^\times \cong C_2$. 对于 $k\geq 3$ 的情形，有 $(\mathbf{Z}/2^k\mathbf{Z})^\times \cong C_2 \times C_{2^{k-2}}$.

  证明

  利用二项式定理直接计算可以知道

  $$5^{2^{k-2}} = (1+2^2)^{2^{k-2}} \equiv 1 \pmod{2^k},$$

  $$5^{2^{k-3}} = (1+2^2)^{2^{k-3}} \equiv 1+2^{k-1} \pmod{2^k}.$$

  所以，5 是 $(\mathbf{Z}/2^k\mathbf{Z})^\times$ 中的 2^{k-2} 阶元. 同时，-1 和 $5^{2^{k-3}}$ 是两个不同的二阶元，所以，$-1\notin\langle 5\rangle$. 所以，$\langle -1\rangle$ 和 $\langle 5\rangle$ 交集是平凡的，故而根据群的直积的刻画可知 $(\mathbf{Z}/2^k\mathbf{Z})^\times \cong \langle -1\rangle \times \langle 5\rangle \cong C_2 \times C_{2^{k-2}}$.

- 对于 p 为奇素数的情形，可以证明 $(\mathbf{Z}/p^k\mathbf{Z})^\times$ 同构于循环群 $C_{\varphi(p^k)}$.

  证明

  要证明 $(\mathbf{Z}/p^k\mathbf{Z})^\times$ 是循环群, 利用有限 Abel 群基本定理可知, 只要证明它的每个 Sylow q-子群都是循环群. 首先，对于 Sylow p-子群，直接计算可知

  $$(1+p)^{p^{k-1}} \equiv 1 \pmod{p^k},$$

  $$(1+p)^{p^{k-2}} \equiv 1+p^{k-1} \pmod{p^k}.$$

  故而，$(1+p)$ 是 p^{k-1} 阶元. 也就是说，$(\mathbf{Z}/p^k\mathbf{Z})^\times$ 的唯一的 Sylow p-子群是循环群 $\langle 1+p\rangle$.

  对于其它的 Sylow q-子群（$q\neq p$），可以通过群同态将它转化为 $k=1$ 的情形. 考虑群同态 $\varphi:(\mathbf{Z}/p^k\mathbf{Z})^\times \to (\mathbf{Z}/p\mathbf{Z})^\times$，它将陪集 $r+p^k\mathbf{Z}$ 映射到陪集 $r+p\mathbf{Z}$. 这个映射的核的大小是 p^{k-1}，所以，将映射 φ 限制在 $(\mathbf{Z}/p^k\mathbf{Z})^\times$ 的 Sylow q-子群（$q\neq p$）上，限制后的映射的核都是平凡的，所以这个 Sylow q-子群同构于映射的像，即 $(\mathbf{Z}/p\mathbf{Z})^\times$ 的 Sylow q-子群. 因此，只要证明 $(\mathbf{Z}/p\mathbf{Z})^\times$ 的 Sylow q-子群都是循环群就可以了.

  最后，证明 $(\mathbf{Z}/p\mathbf{Z})^\times$ 的 Sylow q-子群都是循环群. 因为 $(\mathbf{Z}/p\mathbf{Z})^\times$ 是有限 Abel 群，可以将它按照不变因子分解为 $C_{n_1}\times\cdots\times C_{n_r}$.

  这里，$n_1 \mid n_2 \mid \cdots \mid n_r$. 所以，每个直积因子中都有 n_1 个元素的阶整除 n_1. 如果 $r>1$，则必然有严格多于 n_1 个元素满足方程 $x^{n_1}=1$. 但是，$\mathbf{Z}/p\mathbf{Z}$ 是域，而域上的 n_1 次多项式至多 n_1 个根，所以 $r=1$. 也就是说，$(\mathbf{Z}/p\mathbf{Z})^\times \cong C_{p-1}$.

  这样就证明 $(\mathbf{Z}/p^k\mathbf{Z})^\times \cong C_{p^{k-1}}\times C_{p-1} \cong C_{\varphi(p^k)}$.

一般的模数的情形的乘法群的结构也随之确定. 从现有的结果能够知道整数模 n 乘法群是循环群有且只有模数 n 取

$$1, 2, 4, p^k, 2p^k$$

时，其中，p 是奇素数；否则，整数模 n 乘法群一定有子群 $C_2\times C_2$，不可能是循环群. 当乘法群是循环群的时候，乘法群的生成元就称为该模的 **原根**（primitive root）. 因此，这里的定理给出的正是原根存在的充要条件.

当然, 对乘法群结构的分析蕴含着比原根存在的条件更多的信息. 它清楚地反映了乘法群中不同元素的阶. 群 $(\mathbf{Z}/n\mathbf{Z})^\times$ 中，满足 $x^k=1$ 的元素 x，也就是同余方程 $x^k\equiv 1 \pmod n$ 的解，它称为 **模 n 的 k 次单位根**（k-th root of unity modulo n）；阶恰为 k 的元素，则称为 **模 n 的 k 次本原单位根**（primitive k-th root of unity modulo n）. 利用乘法群的结构, 这些单位根的存在性和数目都可以得到精确的计算. 最后，群 $(\mathbf{Z}/n\mathbf{Z})^\times$ 中所有元素的阶的最小公倍数，即对所有 $x\in(\mathbf{Z}/n\mathbf{Z})^\times$ 都满足 $x^k=1$ 的最小正整数 k，表示为 n 的函数，就是 Carmichael 函数→9.11.18@p1275. 它的一系列性质，都可以从乘法群的结构中获得.

参考资料和注释

- Dummit, D.S. and Foote, R.M. (2004) Abstract Algebra. 3rd Edition, John Wiley & Sons, Inc.
- Quadratic integer - Wikipedia[7]
- Formal power series - Wikipedia[8]
- Multiplicative group of integers modulo n- Wikipedia[9]

[1] https://en.wikipedia.org/wiki/Ideal_(ring_theory)#History

[2] Wikipedia

[3] 最大公因子整环

[4] 这里

[5] Dedekind 整环

[6] Hermite 插值公式

[7] Quadratic integer - Wikipedia

[8] Formal power series - Wikipedia

[9] Multiplicative group of integers modulo n- Wikipedia

9.16.4 域论

前置知识：抽象代数基本概念→9.16.1@p1805、群论→9.16.2@p1812、环论→9.16.3@p1830

引入

域论（field theory）是关于域的理论.

本文涉及的域论主要是域的扩张理论. 域是对加、减、乘、除都封闭的代数结构，算法竞赛中经常需要对质数 p 取模，就相当于在有限域 $\mathbf{F}_p$ 上进行运算. 和实数域 $\mathbf{R}$ 的情形类似，有些问题的求解在更大的域（即复数域 $\mathbf{C}$）上进行计算更为方便，常见的例子比如利用 快速傅里叶变换→9.12.3@p1527 加速实系数多项式的乘法. 对于有限域也可以做类似的操作. 多数读者对于有限域的扩张相对陌生，因而，了解一般的域上的扩张理论是有益的. 文末给出了一些需要对有限域进行扩张的算法应用，同时也简单地讨论了部分应用中可能需要的整数环上的扩张.

与域论紧密相关的是 Galois 理论. 它将域的扩张与其自同构群联系起来，从而可以通过群论的工具来理解域的扩张的性质. 尽管这一理论也往往是相关代数课程的核心内容，但是与算法竞赛的内容相去甚远，故而本文不做过多介绍. 有兴趣的读者应当阅读相关专业书籍.

记号

在不引起歧义时，本文可能会省略掉环和域的乘法记号，并且会将环 $(R,+,\cdot)$ 写作环 R，将域 $(F,+,\cdot)$ 写作域 F. 环和域的加法单位元称为零元，乘法单位元称为幺元. 而且，本文中的 p 总是素数，而 q 总是素数幂，且可以写作 p^n，其中，n 是正整数.

域的扩张

类似群、环的情形，可以建立子域和域同态的概念.

子域

对于域 F，如果它的子环 E 也是域，那么称 E 是域 F 的 **子域**（subfield）.

其中，无论环和子环的定义对于幺元的处理如何，子域 E 必然包含域 F 的幺元[16].

域同态

自域 F 到域 E 的环同态 $\varphi: F \to E$ 也称为自域 F 到域 E 的 **域同态**（field homomorphism）.

[16]这是因为域 E 的幺元 1_E 必然满足 F 上的关系 $x^2 - x = 0$，而后者在域 F 内只有两个根 0_F 和 1_F，由于域的定义要求 $1_E \neq 0_E$，就必然有 $1_E = 1_F$ 和 $0_E = 0_F$.

域同态对幺元的处理

如果与本文的定义不同，环同态要求幺元映射到幺元，那么域同态自然也要求幺元映射到幺元. 否则，幺元也可能映射到零元.

因为域只有平凡的理想，所以域同态要么将整个域映射到零元，要么必然是嵌入映射. 这说明，域同态的讨论可以转化为子域的讨论.

在域的情形，往往更小的域是更为熟悉的域，所以，通常会转而以子域作为基点来考察更大的域. 这就是域的扩张的概念.

域扩张

对于域 F，如果 F 是 E 的子域，则称域 E 是域 F 的 **扩张**（extension），或称 **扩域**，记作 E/F.

域扩张的记号

尽管形式上一致，但是域扩张的概念和商环并没有关系，不应混淆.

例子

复数域 $\mathbf{C}$ 就是实数域 $\mathbf{R}$ 的扩张，而实数域 $\mathbf{R}$ 又是有理数域 $\mathbf{Q}$ 的扩张.

域扩张的次数

对于域扩张 E/F，域 E 总是域 F 上的 线性空间→9.14.7@p1731. 这个线性空间的维度就是域扩张的次数.

域扩张的次数

域扩张 E/F 的 **次数**（degree），是指将 E 看作是域 F 上线性空间时的维度，即 $\dim_F(E)$，记作 $[E:F]$. 如果域扩张的次数是有限的，就称域扩张为 **有限扩张**（finite extension），否则就称为 **无限扩张**（infinite extension）.

例子

域扩张 $\mathbf{C}/\mathbf{R}$ 的次数 $[\mathbf{C}:\mathbf{R}]$ 等于 2，所以是有限扩张. 域扩张 $\mathbf{R}/\mathbf{Q}$ 是无限扩张.

域的扩张次数满足乘法原理.

定理

设 $F \subseteq K \subseteq E$ 都是域，则它们之间的扩张次数满足 $[E:F] = [E:K][K:F]$.

证明

对于扩张次数是无限的情形，这是显然的；否则，如果 $\{\alpha_i\}$ 是 E 作为 K 上线性空间的一组基，且 $\{\beta_j\}$ 是 K 作为 F 上线性空间的一组基，那么可以验证，$\{\alpha_i\beta_j\}$ 是 E 作为 F 上线性空间的一组基.

本文讨论的情形主要是域的有限扩张.

域的特征

对域扩张的研究，有一个自然的起点，就是包含域 F 幺元的最小子域，这个域也称为域 F 的 **素子域**（prime subfield）.

素子域的结构，由域幺元的性质唯一确定. 域的特征就概括了这样的性质.

自 域的特征

域 F 的 **特征**（characteristic）是使得 $n \cdot 1 = 0$ 成立的最小正整数 n；如果这样的 n 不存在，则称域 F 的特征是 0. 其中，$n \cdot 1$ 指 n 个幺元 1 相加的结果. 如果域 F 的特征不为 0，那么就称域 F 是 **有限特征的**（finite characteristic）.

域的特征可以通过环同态来理解. 整数环 $\mathbf{Z}$ 就是自 0 和 1 出发，反复施加加、减、乘等运算得到的封闭结构. 它可以看作某种「原型」，所有包含幺元的环都应当「继承」了整数环的部分结构[17]. 因而，对于域 F，可以考察环同态 $\varphi : \mathbf{Z} \to F$ 并要求 $\varphi(1) = 1$. 这样的环同态是唯一确定的，它将 $n \in \mathbf{N}_+$ 映射到 $n \cdot 1$，即 n 个幺元 1 相加. 该同态的像 $\varphi(\mathbf{Z})$ 嵌入了域 F 中，必然含幺、交换、无零因子，故而是整环. 所以，同态的核 $\ker \varphi$ 必然是素理想. 整数环 $\mathbf{Z}$ 的素理想只能是 (n) 的形式，其中 $n = 0$ 或者 n 是素数. 这样得到的 n 就是该域的特征.

域的特征确定了素子域的结构：

1. 当特征为 0 时，同态 φ 是单的，整数环 $\mathbf{Z}$ 嵌入了域 F 中. 有理数域 $\mathbf{Q}$ 作为最小的包含整数环的域必然也可以嵌入域 F 中，它就是域 F 的素子域；
2. 当特征为素数 p 时，同态 φ 的像 $\mathbf{Z}/p\mathbf{Z}$ 嵌入了域 F 中. 此时，$\mathbf{Z}/p\mathbf{Z}$ 已经是域，记作 $\mathbf{F}_p$，它就是域 F 的素子域.

这些讨论实际上说明了如下结论：

定理

域 F 的特征只能是 0 或素数 p. 特征为 0 的域对应的素子域是 $\mathbf{Q}$，特征为素数 p 的域对应的素子域是 $\mathbf{F}_p$.

定理中的 $\mathbf{Q}$ 和 $\mathbf{F}_p$ 也称为 **素域**（prime field），即子域只有它自身的域. 有限域必然是有限特征的，因为特征为 0 的域至少包含子域 $\mathbf{Q}$.

特征有限的域和零特征的域性质往往不同. 比如，有限特征的域有如下性质：

定理

设 F 的特征为 p，则有：

1. 域 F 的加法群中，所有非零元素的阶都是 p，即对所有 $x \in F$ 都有 $px = 0$；
2. 「新手之梦」（freshman's dream），即对所有 $x, y \in F$ 都有 $(x + y)^p = x^p + y^p$. 进而，映射 $x \mapsto x^p$ 是 F 上的单自同态，叫做 **Frobenius 自同态**（Frobenius endomorphism）.

证明

对于第一条性质，只要注意到 $px = (p1)x = 0x = 0$ 即可. 对于第二条性质，只需要注意到 $(x + y)^p$ 的二项式展开中，除了 x^p 和 y^p 外的全部其他项的系数都是 p 的倍数，故而根据第一条性质就有 $(x + y)^p = x^p + y^p$. 至于验证 $x \mapsto x^p$ 是自同态，只需要再验证 $(xy)^p = x^p y^p$，这是因为域的乘法满足交换律. 最后，域之间的环同态将幺元映射到幺元，则必然是单射.

当然，对于有限域，Frobenius 自同态必然也是满的，因而是域的自同构.

[17] 用范畴论的语言来说，就是 $\mathbf{Z}$ 是幺环范畴的 始对象[1].

单扩张

类似于实数域扩张到复数域的情形，很多扩张可以通过向域中添加额外的元素，并规定其运算性质来完成．在一般的情形，为避免规定运算性质引起的麻烦，不妨考虑在域扩张 E/F 中，将 $E\setminus F$ 中的元素附加到 F 上的情形，此时这些额外的元素与域 F 中元素的运算的规则已经在更大的域 E 中确定了．

自 由子集生成的域扩张

设 E/F 是域扩张，$S\subseteq E$，那么 **由 S 生成的域 F 上的扩张**（extension generated by S over F）就是指同时包含 F 和 S 的，最小的 E 的子域，记作 $F(S)$．

最为简单的情形自然是集合 S 中的元素很少的情形．

自 有限生成扩张

设 E/F 是域扩张，如果存在有限集 $S=\{\alpha_1,\cdots,\alpha_n\}\subseteq E$ 使得 $E=F(S)$ 成立，则称 E 为域 F 的 **有限生成扩张**（finitely generated extension），也记作 $F(\alpha_1,\cdots,\alpha_n)$．

自 单扩张

设 E/F 是域扩张，如果存在 $\alpha\in E$ 使得 $E=F(\alpha)$ 成立，则称域 E 是域 F 的 **单扩张**（simple extension）．其中，元素 α 称为这个单扩张的 **本原元**（primitive element）．

例子

这些例子都是向 $\mathbf{Q}$ 中添加 $\mathbf{C}$ 中的元素得到的．

1. 对于无平方因子的整数 $D\neq 0,1$，二次域 $\mathbf{Q}(\sqrt{D})$ 就是在域 $\mathbf{Q}$ 中添加了 $\sqrt{D}\in\mathbf{C}\setminus\mathbf{Q}$ 得到的单扩张．它的扩张次数是 2，因为 $\{1,\sqrt{D}\}$ 构成了一组基．
2. 域 $\mathbf{Q}(\sqrt{2},\sqrt{3})$ 就是在域 $\mathbf{Q}$ 中添加了 $\sqrt{2}$ 和 $\sqrt{3}$ 得到的扩张．当然有 $\mathbf{Q}(\sqrt{2},\sqrt{3})=\mathbf{Q}(\sqrt{2})(\sqrt{3})=\mathbf{Q}(\sqrt{3})(\sqrt{2})$，即最后的扩张与元素的添加顺序和方式无关．这也是单扩张，因为 $\mathbf{Q}(\sqrt{2},\sqrt{3})=\mathbf{Q}(\sqrt{2}+\sqrt{3})$．它的扩张次数是 4，因为 $\{1,\sqrt{2},\sqrt{3},\sqrt{6}\}$ 构成了一组基．
3. 域 $\mathbf{Q}(\pi)$ 也是单扩张，其中，π 是圆周率．它是无限扩张，因为 $\mathbf{Q}[\pi]\subseteq\mathbf{Q}(\pi)$ 已经有一组基 $\{1,\pi,\pi^2,\cdots\}$．
4. 有限生成的扩张未必是单扩张．例如，设 x,y 是两个不定元，则有理分式域 $\mathbf{Q}(x,y)$ 是 $\mathbf{Q}$ 的有限生成扩张，但它的超越次数是 2，故而不是单扩张．

这些例子说明，单扩张的性质可能相差悬殊．这取决于添加的元素的性质．

代数扩张

为了分析向域中添加元素可能出现的所有情形，不妨仿照前文对域的特征的讨论，考察多项式环 $F[x]$ 到扩张 E/F 的环同态．此处的 $F[x]$ 起到了前文的整数环 $\mathbf{Z}$ 的作用：它正是在域 F 中添加不定元 x 后且对加、减、乘封闭的结构的「原型」[18]．

设环同态 $\varphi:F[x]\to E$ 满足 φ 限制在 F 上是恒等映射，且 $\varphi(x)=\alpha$，即将不定元映射到扩张 E 中的某个元素．此时，因为像 $\varphi(F[x])=F[\alpha]$ 必然是整环，同态的核 $\ker\varphi$ 必然是多项式环 $F[x]$ 的素理想．域上的多项式环是主理想整环，因而它必然有 $(f(x))$ 的形式，其中 $f(x)=0$ 或 $f(x)$ 是 $F[x]$ 中的不可约元．对此有如下讨论：

1. 当同态的核 $\ker\varphi=\{0\}$ 时，多项式环 $F[x]$ 嵌入到 E 中，它的像 $F[\alpha]$ 是整环．因而，域 E 中同时包含 F 和 α 的最小的域就是 $F[\alpha]$ 的分式域，即 $F(\alpha)$．这个记号，既可以解释为将有理分式域

[18] 严格地说，这里指的是多项式环 $R[x]$ 的 万有性质[2]（universal property）．

$F(x)$ 中的不定元代入 α 的结果，也可以解释为域 F 上由 α 生成的单扩张：这两个解释在这个语境下得到的结果是一致的；

2. 当同态的核 $\ker\varphi=(f(x))$，且 $f(x)$ 为不可约元时，就成立 $\varphi(f(x))=f(\alpha)=0$，即 $\alpha\in E$ 是 F 上的多项式 $f(x)$ 的根. 因为 F 是域，不妨设 $f(x)$ 是首一多项式. 此时同态 φ 的像是域 $F(\alpha)$，故而有 $F[x]/(f(x))\cong F(\alpha)$.

   此时又可以分为两种情形：

   1. 如果 $f(x)$ 是一次多项式，即 $f(x)=x-\alpha$ 时，有 $\alpha\in F$，故而扩张 $F(\alpha)=F$ 是平凡的；
   2. 其余情形，$f(x)$ 是高于一次的不可约多项式，且 $\alpha\in E\setminus F$，此时的像 $F[\alpha]$ 已经是包含 F 和 α 的域，因而，它就是 $F(\alpha)$，即 F 上由 α 生成的扩张，且 $F(\alpha)\supset F$ 不是平凡的.

这些讨论启发了如下的定义：

自 代数元与超越元

对于扩张 E/F，如果元素 $\alpha\in E$ 是 F 上某个非零多项式 $f(x)$ 的根，则称 α 是 F 上的 **代数元**（algebraic element）；否则，称元素 α 是 F 上的 **超越元**（transcendental element）.

自 极小多项式

对于域 F 上的代数元 α，以 α 为根且次数最小的首一多项式 $f(x)$ 称作它的 **极小多项式**（minimal polynomial）.

此处的极小多项式就是前文分析中的不可约多项式 $f(x)$. 当然，也可以直接证明极小多项式都是不可约的. 极小多项式 $f(x)$ 的极小性就意味着，只要域 F 上的多项式以 α 为根，就必然能够分解出因子 $f(x)$.

例子

1. $\sqrt{2}$ 是 $\mathbf{Q}$ 上的代数元，极小多项式是 x^2-2.
2. $\sqrt{2}$ 是 $\mathbf{R}$ 上的代数元，极小多项式是 $x-\sqrt{2}$.
3. π 是 $\mathbf{Q}$ 上的超越元.
4. 一般地，$\mathbf{Q}$ 上的代数元称为 **代数数**（algebraic number），而超越元称为 **超越数**（transcendental number）. 特别地，如果代数数的极小多项式是整系数多项式，它就称作 **代数整数**（algebraic integer）. 代数扩张中的全体代数整数构成环. 例如，二次域 $\mathbf{Q}(\sqrt{D})$ 中的代数整数就构成二次整数环 $\mathbf{Z}[\omega]$. 此处记号的含义见 二次整数环→9.16.3@p1830 页面.

自 代数扩张与超越扩张

对于扩张 E/F，如果域 E 的元素都是 F 中的代数元，则称域 E 是 F 上的 **代数扩张**（algebraic extension）；否则，称域 E 是 F 上的 **超越扩张**（transcendental extension）.

单扩张的结果，根据添加元素的性质不同，可以分为两类. 当添加的元素是超越元时，单扩张总是同构于有理分式域. 此时，没有任何可以进一步化简的可能性. 但是，当添加的元素是代数元时，单扩张实际上就是 $F[\alpha]$，即将 α 直接替换多项式环 $F[x]$ 中的不定元 x 得到的结果. 从初等的视角看，相较于超越元的情形，此时扩域中的元素可以没有分母；这意味着，类似于初等算术中「分母有理化」的过程，在代数元的单扩张中总是可行的. 因为算法竞赛中涉及到的扩域主要是单代数扩张，下一节要对它的计算做更为细致的讨论.

单代数扩张的重要性，也反映在如下的定理中：

定理

域扩张是有限扩张，当且仅当它是有限生成的代数扩张.

证明

设 F 为域，$E=F(\alpha_1,\cdots,\alpha_n)$ 为域上的有限生成代数扩张，即 α_i 都是 F 上的代数元. 设 $E_i=F(\alpha_1,\cdots,\alpha_i)$，则 $E_0=F$ 且 $E_n=E$. 注意到，α_i 必然是 E_{i-1} 上的代数元，因为 α_i 在域 F 上的极小多项式也是 E_{i-1} 上的多项式；而且 α_i 在 E_{i-1} 上的极小多项式次数必然不超过 α_i 在域 F 上的极小多项式次数. 故而，$[E_i:E_{i-1}]$ 必然是有限的，根据域扩张次数的乘法原理，$[E:F]=\prod_{i=1}^{n}[E_i:E_{i-1}]$ 也是有限的. 反过来，从 $E_0=F$ 开始，对于已经构造好的 E_i，可以每次都在 $E\setminus E_i$ 中选择元素 α_{i+1} 加入到 E_i 中，得到扩域 $E_{i+1}=E_i(\alpha_{i+1})$，直到 $E_n=E$ 为止. 因为扩张的次数在不断的降低，这个过程必然在有限步内终止. 因此，有限扩张必然是有限生成的代数扩张.

这意味着，要理解有限扩张的性质，只要理解单代数扩张即可. 因为有限扩张总是可以通过有限多个的单代数扩张得到.

单代数扩张的结构与计算

本节中，设 F 是域，E 是它的扩域，且 $\alpha\in E\setminus F$ 是域 F 上的代数元. 设 α 的极小多项式是 $f(x)$，且多项式 $f(x)$ 是 n 次首一多项式，亦即

$$f(x)=x^n+a_{n-1}x^{n-1}+\cdots+a_1x+a_0,$$

其中，$a_0,a_1,\cdots,a_{n-1}\in F$ 且 $f(x)$ 在 F 上不可约.

同构关系 $F(\alpha)\cong F[x]/(f(x))$ 指出，扩域 $F(\alpha)$ 中的运算就是模 $f(x)$ 的多项式的计算. 根据多项式的带余除法，只需要考虑所有次数小于 $n=\deg f(x)$ 的多项式的同余类就可以了. 对于这些多项式，自然的一组基就是 $\{1,\alpha,\cdots,\alpha^{n-1}\}$. 因此，有如下结论：

定理

在本节的假设下，扩域 $F(\alpha)$ 可以写作

$$F(\alpha)=\{\lambda(\alpha)=\lambda_0+\lambda_1\alpha+\cdots+\lambda_{n-1}\alpha^{n-1}:\lambda_0,\lambda_1,\cdots,\lambda_{n-1}\in F\}.$$

其中，$\lambda(x)$ 遍历全体次数小于 n 的多项式. 因此，扩张次数 $[F(\alpha):F]=n$，即 α 的极小多项式的次数. 扩域中，元素 $\lambda(\alpha)$ 和 $\mu(\alpha)$ 的加法，就是多项式的加法，即对应位置的系数相加；元素 $\lambda(\alpha)$ 和 $\mu(\alpha)$ 的乘法，结果可以写作 $\rho(\alpha)$，其中 $\rho(x)$ 是乘积 $\lambda(x)\mu(x)$ 除以 $f(x)$ 的余式.

当然，作为域，还可以计算 $F(\alpha)$ 中元素的除法. 根据定理中描述的乘法的过程，这相当于求解多项式环上的 线性同余方程→9.11.11@p1233. 类比整数的做法，要计算商 $\lambda(\alpha)/\mu(\alpha)$，可以先确定 $\mu(\alpha)$ 的乘法逆元，再乘以 $\lambda(\alpha)$ 即可. 要计算 $\mu(\alpha)$ 的乘法逆元，只要解同余方程 $\mu(x)\xi(x)\equiv 1\pmod{f(x)}$ 即可. 这可以通过扩展欧几里得算法实现.

下面，通过几个具体的例子理解计算的细节.

例子

考察扩域 $\mathbf{Q}(\alpha)$，其中的 α 是方程 $x^3-2x-2=0$ 的一个根. 要计算

$$\frac{1+\alpha}{1+\alpha+\alpha^2}$$

的值.

第一步是计算 $1+\alpha+\alpha^2$ 的逆元，也就是要计算同余方程

$$(x^2+x+1)\xi(x)+(x^3-2x-2)\nu(x)=1$$

的解．对此应用扩展欧几里得算法．先做辗转相除法，即有如下过程：

$$\begin{aligned}x^3-2x-2&=(x-1)(x^2+x+1)+(-2x-1),\\ x^2+x+1&=\left(-\frac{1}{2}x-\frac{1}{4}\right)(-2x-1)+\frac{3}{4},\\ -2x-1&=\left(-\frac{8}{3}x-\frac{4}{3}\right)\frac{3}{4}.\end{aligned}$$

再计算同余方程中的系数，即有

$$\begin{aligned}\frac{3}{4}&=(x^2+x+1)+\left(\frac{1}{2}x+\frac{1}{4}\right)(-2x-1)\\ &=(x^2+x+1)+\left(\frac{1}{2}x+\frac{1}{4}\right)((x^3-2x-2)-(x-1)(x^2+x+1))\\ &=\left(-\frac{1}{2}x^2+\frac{1}{4}x+\frac{5}{4}\right)(x^2+x+1)+\left(\frac{1}{2}x+\frac{1}{4}\right)(x^3-2x-2).\end{aligned}$$

因而，方程的解为

$$\xi(x)=-\frac{2}{3}x^2+\frac{1}{3}x+\frac{5}{3},\ \nu(x)=\frac{2}{3}x+\frac{1}{3}.$$

这说明 $1+\alpha+\alpha^2$ 的逆元是

$$-\frac{2}{3}\alpha^2+\frac{1}{3}\alpha+\frac{5}{3}.$$

第二步，就是计算逆元和 $1+\alpha$ 的乘积．对此，有

$$\begin{aligned}(1+\alpha)\left(-\frac{2}{3}\alpha^2+\frac{1}{3}\alpha+\frac{5}{3}\right)&=-\frac{2}{3}\alpha^3-\frac{1}{3}\alpha^2+2\alpha+\frac{5}{3}\\ &=-\frac{2}{3}(2\alpha+2)-\frac{1}{3}\alpha^2+2\alpha+\frac{5}{3}\\ &=-\frac{1}{3}\alpha^2+\frac{2}{3}\alpha+\frac{1}{3}.\end{aligned}$$

这就是最后的答案．

在例子中只用到了 α 是方程的一个根这个条件，却并没有指定它是任何一个具体的根．多项式 $x^3-2x-2=0$ 在复数域 $\mathbf{C}$ 有一个实根和一对共轭复根，将它们中的任何一个添加进有理数域 $\mathbf{Q}$ 中得到的扩域都是同构的．也就是说，这三个互异的根在代数的视角上是没有区别的．

一般地，对于域 F 上的不可约多项式 $f(x)$，在扩域中有不同的根 $\alpha\neq\beta$，这些根在分别对域 F 做单扩张时表现出相同的代数性质，这些根互相称为 **共轭**（conjugate）．复数域上的通常意义的共轭，就是这一概念在域扩张 $\mathbf{C}/\mathbf{R}$ 上的特例．

例子

考察扩域 $\mathbf{F}_2(\alpha)$，其中的 α 是方程 $x^2+x+1=0$ 的一个根．一般地，对于 $a+b\alpha$ 和 $c+d\alpha$，有运算规则

$$\begin{aligned}(a+b\alpha)+(c+d\alpha)&=(a+c)+(b+d)\alpha,\\ (a+b\alpha)(c+d\alpha)&=ac+(ad+bc)\alpha+bd\alpha^2\\ &=(ac+bd)+(ad+bc+bd)\alpha.\end{aligned}$$

这提供了类似于复数域上的运算法则. 大多数读者对于这样的根 α 都应当是陌生的, 但这并不妨碍对这样的域中的元素进行运算．实际上，有 $[\mathbf{F}_2(\alpha):\mathbf{F}_2]=2$，因而，作为线性空间 $|\ \mathbf{F}_2(\alpha)|=4$，即这样得到的是大小为 4 的有限域．稍后会看到，所有的有限域都是这样构造的．

在小规模运算时，对首一多项式取模 $f(x)$ 的运算通常可以通过代入

$$x^n = -a_{n-1}x^{n-1} - \cdots - a_1 x - a_0$$

对目标多项式降次来进行．而且，对于低次的扩张，往往可以直接计算出系数的运算规则，使用类似复数类的实现而不必每次都计算取模等过程．

作为单代数扩张的实例，可以参考下文中的有限域的 参考实现→9.16@p1805．

此处描述的算法在实践中都只能处理扩张次数比较低的情形，这对于绝大多数算法竞赛中的应用都是足够的．对于扩张次数高到成为复杂度瓶颈的情形，应当采取适当的多项式技术（快速傅里叶变换→9.12.3@p1527、快速数论变换→9.12.4@p1539、多项式快速取余→9.12.9@p1559、多项式欧几里得→9.12.1@p1519 等）加速运算．

分裂域

上文已经对单代数扩张的结构做了详尽的讨论．但是，这样的扩张往往并不充分：

例子

考察扩张 $\mathbf{Q}(\sqrt[3]{2})/\mathbf{Q}$．代数元 $\sqrt[3]{2}$ 在域 $\mathbf{Q}$ 上的极小多项式是 x^3-2．在复数域 $\mathbf{C}$ 中，多项式 x^3-2 有三个根，即 $\sqrt[3]{2}, \sqrt[3]{2}\omega, \sqrt[3]{2}\omega^2$，其中，$\omega = \mathrm{e}^{2\pi\mathrm{i}/3}$ 是 1 的三次本原单位根．尽管 $\mathbf{Q}(\sqrt[3]{2}) \cong \mathbf{Q}(\sqrt[3]{2}\omega) \cong \mathbf{Q}(\sqrt[3]{2}\omega^2)$，但是 $\mathbf{Q}(\sqrt[3]{2})$ 中并没有另外的两个根，这使得 $\sqrt[3]{2}+\sqrt[3]{2}\omega$ 这种运算就已经无法进行．如果要完整地考察这三个根，需要对域 $\mathbf{Q}(\sqrt[3]{2})$ 做进一步扩张，即扩张至 $\mathbf{Q}(\sqrt[3]{2}, \sqrt[3]{2}\omega, \sqrt[3]{2}\omega^2)$．

前文已经说明，要做这样的扩张，只要对元素逐个做单扩张即可．应当注意的是，$\sqrt[3]{2}\omega$ 在域 $\mathbf{Q}$ 中和在域 $\mathbf{Q}(\sqrt[3]{2})$ 中的极小多项式并不相同：前者是 $x^3-2 \in \mathbf{Q}[x]$，后者则是 $x^2+\sqrt[3]{2}x+\sqrt[3]{4} \in \mathbf{Q}(\sqrt[3]{2})[x]$，因为有

$$x^3-2 = (x-\sqrt[3]{2})(x^2+\sqrt[3]{2}x+\sqrt[3]{4}).$$

原来的极小多项式在域的扩张后分解出一次因子，因而剩余的根的极小多项式的次数低于原来的域上的极小多项式．域不断扩张的过程，就是多项式不断「分裂」的过程．因此，每次单扩张时，都需要重新确定极小多项式．

将多项式的全部根都添加到域中，得到的就是多项式的分裂域．

自 分裂

设 F 为域．如果多项式 $f(x)$ 在 $F[x]$ 中可以分解为一系列一次因子的乘积，就称多项式 $f(x)$ 在域 F 中 **分裂**（split）．

自 分裂域

对于域 F 上的多项式 $f(x)$，如果扩张 E/F 满足 $f(x)$ 在域 E 中分裂但不在任何 E 的真子域中分裂，就称域 E 是多项式 $f(x)$ 的 **分裂域**（splitting field）．

可以证明，如同单扩张一样，给定多项式的分裂域在同构意义下是唯一确定的，与具体的构造方法无关．分裂域总是有限扩张．

自 正规扩张

对于代数扩张 E/F，如果对所有 $\alpha \in E$ 都有 α 的极小多项式在 E 中分裂，则称域 E 是域 F 的 **正规扩张**（normal extension）．

正规扩张在 Galois 理论中起到基础的作用．

代数闭域

前文提及的多数扩张的概念原则上需要在比扩张更大的域内进行. 尽管对于单扩张的情形, 通过多项式环可以不依赖于更大的域构造出域的扩张, 但是对于一般的情形并没有这样的手段. 对于有理数域 $\mathbf{Q}$ 和实数域 $\mathbf{R}$, 总是可以假定代数扩张包含在复数域 $\mathbf{C}$ 内部. 对于有限域, 并没有类似的已知的域. 其实, 对于所有的域, 都存在代数闭包, 使得域上所有的代数扩张都可以假定在代数闭包内进行. 这就彻底解决了这一问题.

代数闭包

对于域 F, 如果域 $\overline{F}$ 是域 F 的代数扩张, 且所有的 $f(x)\in F[x]$ 都在 $\overline{F}$ 中分裂, 则称域 $\overline{F}$ 是域 F 的 **代数闭包** (algebraic closure).

代数闭包是域上的正规扩张. 它的构造方式也基本上就是将所有可能的多项式的根添加到域中. 而且和分裂域一样, 某个域的代数闭包在同构意义下也是唯一的.

定理

任何域 F 都有代数闭包.

证明

证明的困难来自于集合论. 此处引用 Artin 的一个证明.

证明的第一部分从 F 出发, 构造了扩张 K_1/F 使得所有 F 上的多项式在 K_1 中都有至少一个根. 对于域 F, 考察多元多项式环[19] $R=F[\cdots,x_f,\cdots]$, 其中的不定元 x_f 的下标取遍所有 F 上的首一多项式. 此时, 由所有 $f(x_f)$ 生成的理想记作 I. 首先, $I\neq R$, 故而极大理想 $M\supseteq I$ 存在. 否则, 如果 $1\in I$, 必然存在有限多个域 F 上的首一多项式 f_i 和相应的环 R 中的元素 g_i 使得 $g_1f_1(x_{f_1})+\cdots+g_kf_k(x_{f_k})=1$ 成立. 设 $F(\alpha_1,\cdots,\alpha_k)$ 为向 F 中添加 $f_i(x)$ 的根 α_i 后得到的代数扩张, 则在 $F(\alpha_1,\cdots,\alpha_k)$ 中令上面得到的恒等式中 x_{f_i} 都代入 α_i, 而在各个 g_i 中出现的其它不定元 x_f 都带入 0, 则得到 $F(\alpha_1,\cdots,\alpha_k)$ 上的等式 $0=1$, 这矛盾. 故而, $I\neq R$, 极大理想 M 的构造是合法的. 此时商环 R/M 是域, 记作 K_1, 且任何 F 上的首一多项式 $f(x)$ 都在 K_1 中有根 $\overline{x_f}$.

证明的第二部分则归纳地得到了包含 F 的一个代数闭域 K (定义见下文). 重复上述构造, 基于域 K_i 可以构造出域 K_{i+1}, 使得 K_i 的多项式在 K_{i+1} 中都至少有一个根. 而且, K_i 自然地嵌入到 K_{i+1} 中, 所以可以定义它们的并集 $K=\bigcup_{i=1}^{\infty}K_i$. 容易验证, 这也是域, 而且 K 上的任何多项式的系数必然全部包含在某个 K_i 中, 故而它的一个根必然出现 $K_{i+1}\subseteq K$ 中. 这说明 K 上的所有多项式都在 K 上至少有一个根, 所以, K 是代数闭域.

最后, 令 K 中所有 F 上的代数元组成的集合记作 $\overline{F}$. 它显然是域; 因为对于任何 $\alpha,\beta\in\overline{F}$, 都有 $\alpha\pm\beta,\alpha\beta,\alpha/\beta\in F(\alpha,\beta)\subseteq\overline{F}$. 它也是 F 的代数扩张, 因为它的元素都是 F 上的代数元. 对于 F 上的多项式 $f(x)$, 它的所有根都是 F 上的代数元, 故而也在 $\overline{F}$ 中, 故而必然可以分裂为一次因子的乘积. 这就说明 $\overline{F}$ 是 F 上的代数闭包.

例子

1. 实数域 $\mathbf{R}$ 的代数闭包是复数域 $\mathbf{C}$.

[19] 此处的多项式环有无限多个不定元. 要定义这样的多项式环, 首先要定义单项式. 设不定元的集合为 X, 则它上面的单项式是全体只在有限多个不定元处取值不为零的函数 $\alpha:X\to\mathbf{N}$, 可以记作 $x_{i_1}^{\alpha(i_1)}\cdots x_{i_k}^{\alpha(i_k)}$, 其中, $i_1,\cdots,i_k$ 是所有 α 取值不为零的不定元的指标. 多项式是所有有限多个单项式的线性组合. 它们在相应定义的加法和乘法运算下成为环. 对于有限多个不定元的情形, 可以证明这种定义与 多元多项式环→9.16.3@p1830 一节的递归定义得到的结果是一致的.

2. 有理数域 $\mathbf{Q}$ 的代数闭包是全体代数数（即域扩张 $\mathbf{C}/\mathbf{Q}$ 中的代数元）的集合，记作 $\overline{\mathbf{Q}}$.

所有的代数闭包的代数扩张都是平凡的. 这样的域称为代数闭域.

自 代数闭域

如果域 F 上任意非常数多项式 $f(x)$ 都至少有一个根 $\alpha \in F$，那么就称域 F 为一个 **代数闭域**（algebraically closed field）.

事实上，它有如下等价定义：

定理

对于域 F，以下性质都是等价的：
1. 域 F 是代数闭域；
2. 域 F 上的所有多项式 $f(x)$ 都分裂；
3. 域 F 上的不可约多项式只有一次多项式；
4. 域 F 没有非平凡的代数扩张；
5. 域 F 没有非平凡的有限扩张；
6. 域 F 是某个域的代数闭包.

证明

前五条性质的等价性显然，考察定义即可. 对于第六条，代数闭域显然是自身的代数闭包，因为它没有非平凡的代数扩张；反过来，要证明域 F 的代数闭包必然是代数闭域. 设 $\overline{F}$ 是域 F 的代数闭包，且 $f(x)$ 是 $\overline{F}$ 上的多项式. 设 α 是 $f(x)$ 的分裂域中 $f(x)$ 的一个根，且 $f(x)$ 的非零系数的集合是 $S \subseteq \overline{F}$. 那么，因为 $F(S)(\alpha) = F(S \cup \{\alpha\})$ 是有限扩张，α 必然也是 F 上的代数元，故而根据代数闭包的定义，α 在 F 上的极小多项式在域 $\overline{F}$ 中分裂，故而 $\alpha \in \overline{F}$. 这说明，$\overline{F}$ 上任意多项式都有至少一个根.

最后，代数基本定理→9.12.2@p1524[20]说明，$\mathbf{C}$ 是代数闭域. 实数域 $\mathbf{R}$ 上的不可约多项式至多是二次的，或者等价地，它上面的代数扩张至多是二次扩张，就是因为通过代数扩张能够得到的最大的域就是 $\mathbf{C}$.

可分扩张

分裂域的概念保证了对于任何域上的多项式，总有扩域能够包括它的所有根，且分裂域精确地给出了这样的最小的扩域. 多项式的性质和它的分裂域的性质紧密联系. 但是，如果希望通过多项式的分裂域来研究多项式的性质，那么首先要面临的一个问题就是，多项式的分裂域与多项式的根的重数无关. 因而，如果有可能，应当考虑多项式的某种意义上的「最简表示」. 受此启发，把在域的代数闭包中也没有重根的多项式称为可分多项式.

自 可分多项式

对于域 F 上的多项式 $f(x)$，如果 $f(x)$ 在 F 的代数闭包 $\overline{F}$ 中没有重根，即它分解成一次因子的乘积时没有重复因子，那么 $f(x)$ 称为 **可分的**（separable）.

因为总是要扩张到域 F 的代数闭包上讨论，可分多项式的判断其实与域 F 的选取无关. 但是，因为多项式的系数在 F 中，应当考虑给出一个判断方法，能够使得在域 F 上就能判断多项式是否可分而不必显式地构造出其扩域.

[20] 虽然名字是代数基本定理，这个结论并不是纯代数的，这是因为实数域的构造需要通过拓扑结构进行.

熟悉分析学的读者知道, 多项式函数的重根有无可以通过它的导数判断: 多项式函数的重根同样是它的导数的根. 虽然多项式和多项式函数并非一致的概念, 但是判断多项式函数重根有无的方法可以类比地迁移到多项式上. 多项式的导数可以形式地定义如下:

自 形式导数

域 F 上的多项式

$$f(x)=a_0+a_1x+a_2x^2+\cdots+a_{n-1}x^{n-1}+a_nx^n=\sum_{i=0}^{n}a_ix^i$$

的 **（形式）导数**（derivative），记作 $Df(x)$，定义为多项式

$$Df(x)=a_1+2a_2x+\cdots+(n-1)a_{n-1}x^{n-2}+na_nx^{n-1}=\sum_{i=1}^{n}ia_ix^{i-1}.$$

这个定义对于所有域上的多项式都适用，不依赖于任何拓扑结构，此处的导数算子 D 只是把一个多项式映射到了另一个多项式. 而且，可以通过对比系数验证，常见的导数运算法则，比如 $D(f(x)g(x))=(Df(x))g(x)+f(x)(Dg(x))$ 等，对于形式导数依然成立.

进而，要检查多项式 $f(x)$ 和它的导数 $Df(x)$ 在分裂域中是否有相同的根，可以不显式地构造出这个分裂域，而是通过它们的最大公因子来判断；这是因为多项式的根总是出现在它的极小多项式中，重复的根意味着相应的极小多项式因子也重复. 于是，有重根的判断法则如下:

定理

对于域 F 上的多项式 $f(x)$，如果 $f(x)$ 有重根 α，那么导数 $Df(x)$ 也有同样的根 α. 进而，多项式 $f(x)$ 可分的充分必要条件是 $f(x)$ 与它的导数 $Df(x)$ 互素，即 $\gcd(f(x),Df(x))=1$.

证明

首先, 因为带余除法在扩域中依然保持, 欧几里得算法的结果也和扩域的选取无关, 所以只要在分裂域中讨论它们的公因子就好了. 由此, 设 α 是 $f(x)$ 的 $k>1$ 重根, 则分裂域中有分解 $f(x)=(x-\alpha)^kg(x)$, 于是它的导数 $Df(x)=k(x-\alpha)^{k-1}g(x)+(x-\alpha)^kDg(x)$ 必然也有根 α. 反过来, 如果 $f(x)$ 和 $Df(x)$ 都有根 α，那么对于分裂域中的分解 $f(x)=(x-\alpha)g(x)$，就有 $Df(x)=(x-\alpha)Dg(x)+g(x)$, 故而 α 也是 $g(x)$ 的根, 因而 α 是 $f(x)$ 的重根. 这就说明定理的第一部分. 进而, 多项式 $f(x)$ 有重根 α, 等价于 $x-\alpha$ 是 $\gcd(f(x),Df(x))$ 的因子. 所以, 多项式 $f(x)$ 不可分, 就等价于 $\gcd(f(x),Df(x))$ 次数大于等于一.

域上的多项式总是可以分解为若干个不可约多项式的乘积. 因为（相伴意义下）不同的不可约多项式总是有着不同的根, 根的重复自然联系到相应的多项式因子的重复. 那么, 如果多项式的分解中没有重复的不可约因子, 是否就能判断多项式可分呢? 换句话说, 不可约的多项式是否都可分? 很遗憾, 在一般的情形下, 无法得到肯定的答案. 问题出现在有限特征的域.

对于域 F 上的不可约多项式 $f(x)$, 多项式 $\gcd(f(x),Df(x))$ 作为 $f(x)$ 的因子, 只能有两种情形, 即 1 或者 $f(x)$. 对于前一种情况, 多项式 $f(x)$ 自然是可分的; 问题出现在后一种情况. 但是, 由于导数的定义中已经保证 $Df(x)=0$ 或者 $\deg Df(x)<\deg f(x)$，多项式 $f(x)$ 成为 $Df(x)$ 的因子，只能说明 $Df(x)=0$. 这在有限特征的域中是有可能的.

对于特征为 p 的域 F，如果 $Df(x)=0$，则多项式的所有非零系数都只能出现在次数恰为 p 的倍数的项上，即多项式 $f(x)$ 可以写作

$$f(x)=a_0+a_px^p+a_{2p}x^{2p}+\cdots+a_{(k-1)p}x^{(k-1)p}+a_{kp}x^{kp}.$$

如果真的存在域 F 上的多项式 $f(x)$ 既不可约也不可分，则它只能有这种形式．但是，如果域 F 中的所有元素总有 p 次根，即对每个系数 a_{jp} 都存在 $b_j \in F$ 使得 a_{jp} 可以写作 b_j^p 的形式，那么，根据 Frobenius 自同态，总有

$$\begin{aligned} f(x) &= a_0 + a_p x^p + a_{2p} x^{2p} + \cdots + a_{(k-1)p} x^{(k-1)p} + a_{kp} x^{kp} \\ &= b_0^p + b_1^p x^p + b_2^p x^{2p} + \cdots + b_{k-1}^p x^{(k-1)p} + b_k^p x^{kp} \\ &= \left(b_0 + b_1 x + b_2 x^2 + \cdots + b_{k-1} x^{k-1} + b_k x^k\right)^p. \end{aligned}$$

因而，这样的域 F 上并不存在这种形式的不可约多项式．因而，这种域上，所有不可约多项式都是可分的．这种域称为完美域．

自 完美域

如果域 F 上的所有不可约多项式都是可分多项式，就称它为 **完美域**（perfect field）．

对于完美域，可分多项式的概念和唯一分解中没有平方因子的多项式的概念是等同的．

定理

设域 F 是完美域，则 F 上的多项式可分，当且仅当它可以写作若干个（相伴意义下）不同的不可约多项式的乘积．

本节的讨论其实已经足够给出移除多项式中的重复因子的方法，它是对多项式的因式分解算法中的关键步骤．但这超出了本文范畴，有兴趣的读者可以参考文末的相关资料．

这些讨论其实也给出了完美域的刻画：

定理

域 F 为完美域，当且仅当域 F 的特征是零，或者域 F 的特征是 p 且任何元素 $x \in F$ 都有 p 次根（即 Frobenius 自同态也是自同构）．

有理数域 $\mathbf{Q}$ 和下文会讨论的有限域 $\mathbf{F}_q$ 都是完美域．

对于域不是完美域的情形，的确存在不可分的不可约多项式．

例子

考虑 $\mathbf{F}_2$ 的有理分式域 $\mathbf{F}_2(t)$ 上的多项式 $x^2 - t$．因为 $\mathbf{F}_2(t)$ 是唯一分解整环 $\mathbf{F}_2[t]$ 的分式域，且 t 是 $\mathbf{F}_2[t]$ 中的素元，则对素元 t 应用 Eisenstein 判别法可知 $x^2 - t$ 在 $\mathbf{F}_2[t]$ 中不可约，故而在 $\mathbf{F}_2(t)$ 中也不可约．但是，它的导数为 0，因而 $x^2 - t$ 并不可分．事实上，在扩域 $\mathbf{F}_2(t)(\sqrt{t})$ 中，它有二重根 $\sqrt{t}$．

最后回到域的扩张的讨论．

自 可分扩张

对于代数扩张 E/F，如果对所有 $\alpha \in E$ 都有 α 的极小多项式是可分多项式，那么称域 E 是域 F 的 **可分扩张**（separable extension）．

完美域上的代数扩张都是可分扩张．这也可以作为完美域的等价定义．

如果一个代数扩张既是正规扩张，也是可分扩张，它也称作 Galois 扩张．Galois 扩张中，任何不可约多项式都没有重根，且根的数目都恰好等于多项式的次数，因而对根的置换可以充分地反映域扩张和多项式的性质．这样的扩张提供了建立 Galois 理论的基石．有兴趣的读者可以参考文末的相关资料．

分圆域

作为域扩张的简单例子，本节讨论分圆域．另一个域扩张的简单例子是 二次域→9.11.32@p1495．

单位根群

复数域 $\mathbf{C}$ 中，多项式 x^n-1 的根称为 n **次单位根**（n-th root of unity）．记 $\zeta_n=\mathrm{e}^{2\pi\mathrm{i}/n}$．那么，全体 n 次单位根就是集合 $C_n=\{\zeta_n^k:k\in\mathbf{Z}\}$．在乘法运算下，$C_n$ 构成 n 次循环群，可以记作 $\langle\zeta_n\rangle$，称为 n 次单位根群．群 C_n 的生成元，也就是那些阶恰好为 n 的元素，称为 n **次本原单位根**（primitive n-th root of unity）．n 次本原单位根的集合 $P_n=\{\zeta_n^k:k\in\mathbf{Z},k\perp n\}$，恰有 $\varphi(n)$ 个元素；其中，$\varphi(n)$ 是 欧拉函数→9.11.5@p1196．将单位根群 C_n 的元素按照它的阶分类，就得到如下分解：

$$C_n=\bigcup_{d\mid n}P_d.$$

对两边元素计数，就得到恒等式 $n=\sum_{d|n}\varphi(d)$．

分圆域

分圆域是将单位根添加到有理数域中得到的扩域．

自 分圆域

将 n 次复单位根 $\zeta_n=\mathrm{e}^{2\pi\mathrm{i}/n}$ 添加到有理数域 $\mathbf{Q}$ 中得到的扩域 $\mathbf{Q}(\zeta_n)$ 称为 n **次分圆域**（n-th cyclotomic field）．

因为全体 n 次单位根在乘法运算下构成循环群 $\langle\zeta_n\rangle$，分圆域 $\mathbf{Q}(\zeta_n)$ 也包括所有这些 n 次单位根．其实，$\mathbf{Q}(\zeta_n)$ 正是域 $\mathbf{Q}$ 上多项式 x^n-1 的分裂域．

定理

分圆域 $\mathbf{Q}(\zeta_n)$ 是有理数域 $\mathbf{Q}$ 上多项式 x^n-1 的分裂域．

证明

设 F 为有理数域 $\mathbf{Q}$ 上多项式 x^n-1 的分裂域．因为 $\mathbf{Q}(\zeta_n)$ 上有多项式 x^n-1 的全体复根，所以 $F\subseteq\mathbf{Q}(\zeta_n)$．反过来，因为 $\zeta_n\in F$，就必然有 $\mathbf{Q}(\zeta_n)=F$．这就说明 $F=\mathbf{Q}(\zeta_n)$．

这可以作为分圆域的等价定义．其实，将任何 n 次本原单位根添加到分圆域中都能够得到 $\mathbf{Q}(\zeta_n)$．

分圆多项式

分圆域 $\mathbf{Q}(\zeta_n)$ 是有理数域 $\mathbf{Q}$ 上的单代数扩张．根据上文的分析，这样的域总是同构于某个多项式环的商环．为了得到这样的同构，需要分析 ζ_n 的极小多项式 $f(x)$．因为 ζ_n 是 x^n-1 的根，所以 $f(x)$ 必然是 x^n-1 的某个因子．这说明，需要考察多项式 x^n-1 在 $\mathbf{Q}[x]$ 内的因式分解．根据 Gauss 引理，它必然可以在 $\mathbf{Z}[x]$ 中分解为若干个不可约的首一多项式的乘积．

因为 $\mathbf{Q}(\zeta_n)$ 是分裂域，多项式 x^n-1 有分解：

$$x^n-1=\prod_{\zeta\in C_n}(x-\zeta)=\prod_{d|n}\prod_{\zeta\in P_d}(x-\zeta).$$

因为不同阶的单位根的代数性质不同，它们必然不会是同一个不可约多项式的根．因此，要考察 ζ_n 的极小多项式，只要考虑上述分解中的因子

$$\Phi_n(x)=\prod_{\zeta\in P_n}(x-\zeta)$$

即可．单位根 ζ_n 的极小多项式，必然是 $\Phi_n(x)$ 的因子．而且，这样定义的 $\Phi_n(x)$ 有如下性质：

定理

$\Phi_n(x)$ 是整系数首一多项式，且在 $\mathbf{Z}[x]$ 中不可约.

证明

由定义，$\Phi_n(x)$ 显然是首一多项式. 首先，要证明 $\Phi_n(x) \in \mathbf{Z}[x]$. 根据 Gauss 引理，多项式 $x^n - 1$ 在 $\mathbf{Z}[x]$ 和 $\mathbf{Q}[x]$ 中有着相同的分解，且每个因子都是整系数首一多项式. 这个分解中，每个因子 $f(x)$ 都是 $\mathbf{Q}[x]$ 上不可约的，且在 $\mathbf{Q}(\zeta_n)$ 中分裂；它的所有根都是 n 次单位根，且必然有着相同的阶，所以这些根必然全部属于某一个 P_d 而不能分别存在于不同的 P_d. 这意味着，每个因子 $f(x)$ 都是某个 $\Phi_d(x)$ 的因子. 故而，$\Phi_n(x)$ 可以写成若干个的整系数首一多项式的乘积，必然也是整系数首一多项式.

接下来，要证明 $\Phi_n(x)$ 在 $\mathbf{Z}[x]$ 是不可约的. 设它有分解 $f(x)g(x)$，且 $f(x)$ 在 $\mathbf{Z}[x]$ 中不可约，那么只要证明 $f(x)$ 包含所有 n 次本原单位根即可. 也就是说，设 ζ 是 $f(x)$ 的一个根，要证明对于所有 $k \perp n$，都有 ζ^k 也是 $f(x)$ 的根；由于 k 总是可以分解为素数的乘积，所以只需要证明对于所有素数 $p \perp n$，ζ^p 是 $f(x)$ 的根就可以了. 假设不然，ζ^p 是 $g(x)$ 的根. 因而，ζ 是 $\mathbf{Z}[x]$ 中多项式 $f(x)$ 和 $g(x^p)$ 的共同的根. 因为 $f(x)$ 是 ζ 在 $\mathbf{Q}$ 上的极小多项式，必然有 $f(x)$ 整除 $g(x^p)$；亦即存在 $h(x) \in \mathbf{Z}[x]$ 使得 $g(x^p) = f(x)h(x)$. 等式两侧同时对 p 取模，得到 $\mathbf{F}_p[x]$ 上的等式 $\overline{g}(x^p) = \overline{f}(x)\overline{h}(x)$. 利用 Frobenius 自同态可知，$\overline{g}(x)^p = \overline{f}(x)\overline{h}(x)$. 因为 $\mathbf{F}_p[x]$ 也是唯一分解整环，$\overline{g}(x)$ 和 $\overline{f}(x)$ 必然有不平凡的公因子，所以，$x^n - \overline{1} = \overline{f}(x)\overline{g}(x)$ 在 $\mathbf{F}_p$ 上不可分. 但是，因为 $p \perp n$，它的形式导数 nx^{n-1} 与它自身互素，这与它不可分矛盾. 故而，可以证明 ζ^p 必然还是 $f(x)$ 的根，因而 $f(x)$ 包含所有 n 次本原单位根，它就是 $\Phi_n(x)$.

这就说明，它就是 ζ_n 的极小多项式，也称为 n **次分圆多项式**（n-th cyclotomic polynomial）. 上面的定义式指出，它有 $\varphi(n)$ 个复根，且这些复根正是全体 n 次本原单位根；其中，$\varphi(n)$ 是 欧拉函数→9.11.5@p1196. 这也说明，$\mathbf{Q}(\zeta_n)/\mathbf{Q}$ 是 $\varphi(n)$ 次扩张.

分圆域 $\mathbf{Q}(\zeta_n)$ 中的代数整数环是 $\mathbf{Z}[\zeta_n]$. 另外，当 $\varphi(n) = 2$ 时，分圆域是 二次扩张→9.11.32@p1495. 具体来说，$\mathbf{Q}(\zeta_4)$ 是二次域 $\mathbf{Q}(\sqrt{-1})$；$\mathbf{Q}(\zeta_3)$ 和 $\mathbf{Q}(\zeta_6)$ 相同，都是二次域 $\mathbf{Q}(\sqrt{-3})$.

利用分圆多项式，多项式 $x^n - 1$ 在 $\mathbf{Z}[x]$ 中有唯一分解

$$x^n - 1 = \prod_{d|n} \Phi_d(x).$$

因此，$(x^d - 1) \mid (x^n - 1)$ 当且仅当 $d \mid n$. 而且，对此式应用 Möbius 反演→9.11.23@p1331 可得

$$\Phi_n(x) = \prod_{d|n} (x^d - 1)^{\mu(n/d)}.$$

利用这个表达式，可以递归地计算出全部的分圆多项式. 此处给出前几个分圆多项式的例子，便于读者熟悉.

分圆多项式

前 10 个分圆多项式如下：

$$\begin{aligned}
\Phi_1(x) &= x-1,\\
\Phi_2(x) &= x+1,\\
\Phi_3(x) &= x^2+x+1,\\
\Phi_4(x) &= x^2+1,\\
\Phi_5(x) &= x^4+x^3+x^2+x+1,\\
\Phi_6(x) &= x^2-x+1,\\
\Phi_7(x) &= x^6+x^5+x^4+x^3+x^2+x+1,\\
\Phi_8(x) &= x^4+1,\\
\Phi_9(x) &= x^6+x^3+1,\\
\Phi_{10}(x) &= x^4-x^3+x^2-x+1.
\end{aligned}$$

一个有趣的事实是，虽然看起来这些分圆多项式的系数都只能是 0 和 ± 1，但是对于一般的 n，这个结论是不对的．第一个反例出现在 $\Phi_{105}(x)$，而且可以证明，随着 n 的增大，它的系数可以取到任意大的值．

利用上文的 Möbius 反演式，可以总结出如下性质来简化 $\Phi_n(x)$ 的计算：

性质

对于分圆多项式 $\Phi_n(x)$，有：

1. 如果素数 $p \mid n$，则 $\Phi_{pn}(x) = \Phi_n(x^p)$；
2. 如果素数 $p \perp n$，则 $\Phi_{pn}(x) = \dfrac{\Phi_n(x^p)}{\Phi_n(x)}$；
3. 特别地，如果 $n > 1$ 是奇数，则 $\Phi_{2n}(x) = \Phi_n(-x)$；
4. 对于素数 p，有 $\Phi_p(x) = 1 + x + \cdots + x^{p-1}$；
5. 特别地，$\Phi_{2^k}(x) = x^{2^{k-1}} + 1$．

这些性质说明，对分圆多项式的计算，重点在于那些次数是无平方因子的奇数的情形．而对于这种情形，可以用性质二逐个添加素因子；每个素因子的加入，只需要做一次多项式除法．

分圆多项式还有很多其它的性质．

定理

设 $\Phi_n(x)$ 为 $n > 1$ 次分圆多项式，且多项式的次数是 $\varphi(n)$．于是，有：

1. 多项式 $\Phi_n(x)$ 是回文多项式，它的 j 次项系数和 $\varphi(n) - j$ 次项系数相同，即 $\Phi_n(x) = x^{\varphi(n)}\Phi_n(1/x)$；
2. 多项式的 $\varphi(n) - 1$ 次项系数等于 Möbius 函数 $-\mu(n)$；
3. 如果 n 是素数幂 p^k，那么 $\Phi_n(1) = p$；否则，$\Phi_n(1) = 1$；
4. 设 $b > 1$ 且 p 为 $\Phi_n(b)$ 的素因子，则 $p \mid n$，或者 n 是乘法群 $(\mathbf{Z}/p\mathbf{Z})^\times$ 中的 b 的阶，且这两种情况不能同时发生．

证明

对于前三条性质，只需要利用 Möbius 反演即可．对于 1，直接考察 $\Phi_n(x)$ 的 Möbius 反演形式，即 $\Phi_n(x) = \prod_{d\mid n}(x^d-1)^{\mu(n/d)}$；对于 2，设 $\Phi_n(x)$ 的 $\varphi(n)-1$ 次项系数为 $f(n)$，则比较 $x^n - 1 = \prod_{d\mid n}\Phi_d(x)$ 等式两侧的 $n-1$ 次项系数可知，$\sum_{d\mid n} f(d) = -[n=1]$，再做 Möbius 反演；对于 3，在 $x^n -$

$1=\prod_{d|n}\Phi_d(x)$ 两侧同时除以 $\Phi_1(x)=x-1$，再代入 $x=1$，即有 $n=\prod_{d|n,d\neq 1}\Phi_d(1)$，再做 Möbius 反演.

下面证明第四条性质. 首先，如果 n 是乘法群 $(\mathbf{Z}/p\mathbf{Z})^\times$ 中的 b 的阶，那么 n 是满足 $p\mid b^n-1$ 的正整数中最小的，故而 $p\mid\Phi_n(b)$. 反过来，如果 $p\mid\Phi_n(b)$，则有 $b^n\equiv 1 \pmod p$；可如果 n 不是乘法群 $(\mathbf{Z}/p\mathbf{Z})^\times$ 中的 b 的阶，那么设它的阶为 k，必然有 $k\mid n$ 且 $p\mid\Phi_k(b)$. 此时，$\Phi_k(x)$ 和 $\Phi_n(x)$ 在域 $\mathbf{F}_p$ 中有公共根 b，这说明 x^n-1 有重根 b. 这说明 $p\mid n$；否则，x^n-1 与它的导数互素，所以在 $\mathbf{F}_p$ 上可分，不可能有重根. 因而，$\Phi_n(b)$ 的素因子 p 只有两种情况：$p\mid n$，或者 n 是乘法群 $(\mathbf{Z}/p\mathbf{Z})^\times$ 中的 b 的阶. 这两种情况是互斥的，因为后者意味着 $n\mid p-1$.

分圆多项式还可以用于解决一些数论和代数问题. 比如说分数在写成某个进制下的小数时的循环节长度，就和分圆多项式有密切的联系. 对于这些具体的应用，有兴趣的读者可以参考文末的资料.

有限域

有限域（finite field），也称作 **Galois 域**（Galois field），就是只有有限多个元素的域. 有限域的结构由其元素个数唯一确定，且它的元素个数必然是素数的幂.

定理

大小为 q 的域存在，当且仅当 q 具有素数幂 p^n 的形式. 而且，这样的域在同构意义下唯一，记作 $\mathbf{F}_q$. 素数 p 是域 $\mathbf{F}_q$ 的特征，正整数 n 为域扩张 $\mathbf{F}_q/\mathbf{F}_p$ 的次数. 最后，$\mathbf{F}_q$ 是 $\mathbf{F}_p$ 上多项式 x^q-x 的分裂域，且恰好包括 x^q-x 的 q 个互异的根.

证明

设域 F 是有限域. 域 F 的特征必然有限，记作 p；故而，域 F 有素子域 $\mathbf{F}_p$. 而且，域 F 必然是 $\mathbf{F}_p$ 上的有限扩张，扩张次数记作 n. 作为 $\mathbf{F}_p$ 上的 n 维向量空间，域 F 有 $q=p^n$ 个元素. 域 F 的全体非零元构成群 $F^\times$，它的阶为 $q-1$，所以有 $x^{q-1}=1$. 因此，$F=F^\times\cup\{0\}$ 的所有元素都满足 $x^q=x$，即它们是多项式 x^q-x 的 q 个互异的根. 因此，在域 F 中多项式 x^q-x 有因子 $\prod_{\alpha\in F}(x-\alpha)$，但是这个因子的次数已经是 q 且最高次项系数就等于 1，所以有 $x^q-x=\prod_{\alpha\in F}(x-\alpha)$. 这说明 x^q-x 在 F 中分裂. 对于任何能够使 x^q-x 分裂的域，由于 x^q-x 有 q 个相异的根，必然至少有 q 个元素. 这说明 F 是使 x^q-x 可以分裂的最小的域，即 x^q-x 的分裂域. 总而言之，大小为 q 的有限域必然是它的素子域上的多项式 x^q-x 的分裂域. 因为分裂域在同构意义下唯一，所以大小为 q 的域必然也唯一.

反过来，给定素数 p 和它的幂 $q=p^n$，要说明 $\mathbf{F}_p$ 上的多项式 x^q-x 的分裂域恰好有 q 个元素，才能说明所有素数幂 q 阶的域都存在. 因为 $\mathbf{F}_p$ 上的多项式 x^q-x 的分裂域总是存在，所以可以设该分裂域中多项式 x^q-x 的全部根组成的集合为 F. 现在要证明 F 是域，因而它就是多项式 x^q-x 的分裂域本身. 但是，迭代 n 次 Frobenius 自同态就可以知道 $x\mapsto x^q$ 也是自同态，因此对任意 $\alpha,\beta\in F$ 都有 $(\alpha\pm\beta)^q=\alpha^q\pm\beta^q$，$(\alpha\beta)^q=\alpha^q\beta^q$ 和 $(\alpha^{-1})^q=(\alpha^q)^{-1}$. 因此，集合 F 对加、减、乘、除都封闭，它是域. 这就说明 F 就是 $\mathbf{F}_p$ 上的多项式 x^q-x 的分裂域.

推论

有限域 $\mathbf{F}_q$（$q>2$）中，全体非零元的和是 0，积是 -1.

证明

有限域的全体非零元恰好是多项式 $x^{q-1}-1$ 的 $q-1$ 个根，应用 Vieta 定理即可.

在素域 $\mathbf{F}_p$ 中，这个推论关于积的结论正是数论中的 Wilson 定理→9.11.14@p1242（的一部分）.

乘法结构

有限域的乘法群 $\mathbf{F}_q^\times = \mathbf{F}_q \setminus \{0\}$ 一定是循环群.

定理

域 F 的乘法群的有限子群一定是循环群.

证明

设 G 为域 F 的乘法群的子群且 $|\ G\ | = n$. 因而，G 是有限 Abel 群. 根据有限 Abel 群基本定理，群 G 有不变因子分解 $C_{n_1} \times \cdots \times C_{n_s}$ 且 $n_1 \mid \cdots \mid n_s$. 所以，对于 G 中的所有元素 x，都有 $x^{n_s} = 1$. 也就是说，群 G 中的元素都是域 F 上多项式 $x^{n_s} - 1$ 的根. 但是，多项式 $x^{n_s} - 1$ 至多有 n_s 个相异的根，即 $n \le n_s$. 但是，$n_s \le n$，所以其实有 $n_s = n$. 这说明 $G \cong C_{n_s}$，即群 G 是循环群.

推论

有限域 $\mathbf{F}_q$ 的乘法群 $\mathbf{F}_q^\times \cong C_{q-1}$.

循环群 $\mathbf{F}_q^\times$ 中有 $\varphi(q-1)$ 个生成元，它们称为有限域的本原元；其中，$\varphi(n)$ 是 欧拉函数→9.11.5@p1196.

本原元

有限域 $\mathbf{F}_q$ 的乘法群的生成元，称为 $\mathbf{F}_q$ 的 **本原元**（primitive element）.

单扩张中的本原元和有限域中的本原元并不相同

尽管单扩张中的本原元和有限域中的本原元的名称一致，两者并不相同. 单扩张中的本原元是相应的单扩张的生成元，而有限域中的本原元是相应的乘法群（作为循环群）的生成元. 有限域作为它的素子域的单扩张的本原元，未必是有限域本身的本原元. 例如，$\mathbf{F}_{25} \cong \mathbf{F}_5[x]/(x^2+x+1)$ 中，$\overline{x}$ 是域扩张的本原元，但是并不是域 $\mathbf{F}_{25}$ 的本原元，因为它的阶数是 3.

$\mathbf{F}_q$ 中的本原元和模 q 的原根也不相同

对于奇数特征的有限域 $\mathbf{F}_q$，总是存在模 q 的 原根→9.16.3@p1830（primitive root）. 但是，不应将它与有限域 $\mathbf{F}_q$ 中的本原元（primitive element）混淆. 虽然它们都是相应的乘法结构作为循环群时的生成元，但是 $(\mathbf{Z}/q\mathbf{Z})^\times$ 和 $\mathbf{F}_q$ 在 q 本身不是素数的情况下并不相同. 比如，前者的阶是 $\varphi(q)$ 而后者的阶是 $q-1$，两个乘法群的大小就不相同.

设 α 是有限域 $\mathbf{F}_q$ 的一个本原元. 那么，对于所有非零元 $x \in \mathbf{F}_q^\times$ 都存在唯一的自然数 $k < q-1$ 使得 $x = \alpha^k$；这个 k 就称为 $\mathbf{F}_q$ 上元素 x 关于基 α 的 **离散对数**（discrete logarithm）. 和 $\mathbf{F}_p$ 上的情形一致，离散对数的算法→9.11.19@p1286 的复杂度都比较高.

通过乘法运算，本原元已经可以生成域的全体非零元素. 这说明，有限域作为它的子域的扩张，一定是单扩张.

定理

对于有限域 $\mathbf{F}_q$，设 F 为 $\mathbf{F}_q$ 的子域，则 $\mathbf{F}_q$ 是 F 上的单代数扩张；又设 α 为 $\mathbf{F}_q$ 的本原元，则 $\mathbf{F}_q = F(\alpha)$.

本原元的极小多项式是有限域的子域上的不可约多项式.

包含关系

有限域的子域也是有限域．有限域之间的包含关系，也完全由它们的阶决定．

定理

设 $\mathbf{F}_q$ 和 $\mathbf{F}_r$ 是有限域，则 $\mathbf{F}_r$ 是 $\mathbf{F}_q$ 的子域，当且仅当存在 k 使得 $q = r^k$．换句话说，$\mathbf{F}_{p^d}$ 是 $\mathbf{F}_{p^n}$ 的子域，当且仅当 $d \mid n$.

证明

如果 $\mathbf{F}_r$ 是 $\mathbf{F}_q$ 的子域，那么两者必然有相同的特征 p．域扩张 $\mathbf{F}_q/\mathbf{F}_r$、$\mathbf{F}_r/\mathbf{F}_p$ 和 $\mathbf{F}_q/\mathbf{F}_p$ 都是单代数扩张，分别记它们的扩张次数为 k, d, n，则扩张次数必然满足 $n = kd$．而且，$r = p^d$ 和 $q = p^n$，并成立 $q = p^n = p^{kd} = (p^d)^k = r^k$.

反过来，要证明对于所有 $d \mid n$，$\mathbf{F}_{p^d}$ 都是 $\mathbf{F}_{p^n}$ 的子域．记 $r = p^d$ 且 $q = p^n$．设 F 为有限域 $\mathbf{F}_q$ 中方程 $x^r - x = 0$ 的全体根的集合．通过 Frobenius 自同态可以证明，集合 F 必然构成域；关键是要证明，这样的根恰好有 r 个，所以才有 $F \cong \mathbf{F}_r$．因为 $d \mid n$，所以 $(p^d - 1) \mid (p^n - 1)$，所以 $(x^{p^d-1} - 1) \mid (x^{p^n-1} - 1)$，也就是 $(x^r - x) \mid (x^q - x)$．所以，$x^r - x$ 在 $\mathbf{F}_q$ 上分裂，故而在 $\mathbf{F}_q$ 内有 r 个不同的根．这就说明 $F \cong \mathbf{F}_r$ 是 $\mathbf{F}_q$ 的子域.

这一定理说明，有限域 $\mathbf{F}_{p^n}$ 的包含关系，对应着域的阶 p^n 中的指数 n 之间的整除关系．所有特征为 p 的有限域 $\mathbf{F}_{p^n}$ 之间形成的格，也就同构于整数 n 在整除关系下形成的格．当然，为了让有限域 $\mathbf{F}_{p^n}$ 之间的交集等运算有意义，需要将所有特征为 p 的域都嵌入到 $\mathbf{F}_p$ 的代数闭包中.

定理

设 F 为 $\mathbf{F}_p$ 的代数闭包，则域 F 中多项式 $x^{p^n} - x$ 的根的集合构成有限域 $\mathbf{F}_{p^n}$．那么，有：

1. $F = \bigcup_{n=1}^{\infty} \mathbf{F}_{p^n}$，即 $\mathbf{F}_p$ 的代数闭包就是所有特征为 p 的有限域的并集；
2. 全体特征为 p 的有限域 $\mathbf{F}_{p^n}$ 在包含关系下形成的格，同构于整数 n 在整除关系下形成的格．特别地，$\mathbf{F}_{p^n}$ 和 $\mathbf{F}_{p^m}$ 的交集 $\mathbf{F}_{p^n} \cap \mathbf{F}_{p^m} = \mathbf{F}_{p^{\gcd(n,m)}}$，而同时包含 $\mathbf{F}_{p^n}$ 和 $\mathbf{F}_{p^m}$ 的最小的域 $\mathbf{F}_{p^n}\mathbf{F}_{p^m} = \mathbf{F}_{p^{\operatorname{lcm}(n,m)}}$.

证明

关键在于证明第一部分，即 $\bigcup_{n=1}^{\infty} \mathbf{F}_{p^n}$ 是 $\mathbf{F}_p$ 的代数闭包．第二部分是前面关于有限域的子域的定理的简单推论.

注意到，任取 $\alpha \in \bigcup_{n=1}^{\infty} \mathbf{F}_{p^n}$，必然存在 $n \in \mathbf{N}_+$ 使得 $\alpha \in \mathbf{F}_{p^n}$ 成立，故而 α 是 $\mathbf{F}_p$ 上的代数元；因而，$\bigcup_{n=1}^{\infty} \mathbf{F}_{p^n}$ 是 $\mathbf{F}_p$ 的代数扩张．对于任何 $\mathbf{F}_p$ 上的 m 次多项式 $f(x)$，它在代数闭包 F 中有至多 m 个不同的根 $\{\alpha_i\}_{i=1}^{m}$．设根 α_i 的极小多项式次数为 n_i，则 α_i 必然包含在域 $\mathbf{F}_{p^{n_i}}$ 内；故而，$f(x)$ 的所有根都在 $\bigcup_{n=1}^{\infty} \mathbf{F}_{p^n}$ 中，亦即 $f(x)$ 在 $\bigcup_{n=1}^{\infty} \mathbf{F}_{p^n}$ 上分裂．根据代数闭包的定义，$\bigcup_{n=1}^{\infty} \mathbf{F}_{p^n}$ 就是 $\mathbf{F}_p$ 的代数闭包.

自同构群

有限域 $\mathbf{F}_q$ 上的子域都是形如 $x^r - x$ 的多项式的根的集合．换言之，它们都是某个映射 $x \mapsto x^r$ 的不动点集合．这其实揭示了有限域的子域和自同构群的子群之间的深刻对应关系.

特征为 p 的域上都有 Frobenius 自同态 $\sigma_p : x \mapsto x^p$．对于有限域 $\mathbf{F}_q$ 的情形，这也是自同构；这说明有限域 $\mathbf{F}_q$ 都是完美域．域 $\mathbf{F}_q$ 的自同构群就是 n 阶循环群 $\langle \sigma_p \rangle$，它的一个生成元就是 Frobenius 自同态 σ_p.

定理

有限域 $\mathbf{F}_q$ 的自同构群 $\mathrm{Aut}(\mathbf{F}_q) = \langle \sigma_p \rangle$ 是 n 阶循环群，且生成元 σ_p 是 Frobenius 自同态 $x \mapsto x^p$.

证明

首先，Frobenius 自同态 σ_p 在有限域 $\mathbf{F}_q$ 上是自同构，因为有限集合上的单射必然也是满射．因此，$\sigma_p \in \mathrm{Aut}(\mathbf{F}_q)$.

然后，σ_p 的阶是 n．这是因为对于所有 $x \in \mathbf{F}_q$ 都有 $\sigma_p^n(x) = x^{p^n} = x$，故而 σ_p^n 是恒等映射；而且对于任何 $k < n$ 都有 σ_p^k 不是恒等映射，否则 $\mathbf{F}_q$ 的元素都得是 $x^{p^k} - x$ 的根，这不可能.

最后，$\mathrm{Aut}(\mathbf{F}_q)$ 至多有 n 个元素．设 α 为 $\mathbf{F}_q$ 的一个本原元，则自同构 $\sigma \in \mathrm{Aut}(\mathbf{F}_q)$ 由它在 α 处的取值 $\sigma(\alpha)$ 唯一确定．但是，σ 必须将 α 映射到它的共轭元；否则，α 和 $\sigma(\alpha)$ 不再是同一个极小多项式的根．这样的共轭元只有 n 个，这就说明 $\mathrm{Aut}(\mathbf{F}_q)$ 也至多有 n 个元素.

因此，$\mathrm{Aut}(\mathbf{F}_q)$ 中的 n 个元素正是 $\langle \sigma_p \rangle$．定理得证.

自同构群 $\mathrm{Aut}(\mathbf{F}_q)$ 的子群和有限域 $\mathbf{F}_q$ 上的子域一一对应.

定理

设 $\mathbf{F}_q$ 为有限域，$\mathcal{F}$ 为它的全体子域，$\mathcal{G}$ 为它的自同构群 $\mathrm{Aut}(\mathbf{F}_q)$ 的全体子群．对此，有:

1. 对于 $F \in \mathcal{F}$，设 $\mathrm{Aut}(\mathbf{F}_q/F)$ 为 $\mathrm{Aut}(\mathbf{F}_q)$ 中保持 F 不变的自同构的集合，即 $\mathrm{Aut}(\mathbf{F}_q/F) = \{\sigma \in \mathrm{Aut}(\mathbf{F}_q) : \forall x \in F(\sigma(x) = x)\}$，则 $\mathrm{Aut}(\mathbf{F}_q/F) \leq \mathrm{Aut}(\mathbf{F}_q)$;
2. 对于 $G \in \mathcal{G}$，设 F^G 为 G 中的所有自同构的不动点集合的交集，即 $F^G = \{x \in \mathbf{F}_q : \forall \sigma \in G(\sigma(x) = x)\}$，则 F^G 为 $\mathbf{F}_q$ 的子域;
3. 映射 $F \to \mathrm{Aut}(\mathbf{F}_q/F)$ 和映射 $G \to F^G$ 互为逆映射，且是 $\mathcal{F}$ 和 $\mathcal{G}$ 之间的一一对应;
4. 这个一一对应，将子域之间的扩张关系映射为子群之间的包含关系，即对于任何 $F_1 \subseteq F_2$，都有 $\mathrm{Aut}(\mathbf{F}_q/F_2) \leq \mathrm{Aut}(\mathbf{F}_q/F_1)$.

这个结论是一般的 Galois 理论的基本定理的特殊情形，它将域扩张和群论的内容联系起来，从而可以通过群论的方法解决域扩张的问题.

不可约多项式

有限域 $\mathbf{F}_q$ 上的不可约多项式十分容易确定. 因为有限域 $\mathbf{F}_q$ 上的每个 n 次不可约多项式都对应着 n 次代数扩张, 而这样的扩张是唯一的, 故而所有 n 次不可约多项式的根都可以在 $\mathbf{F}_{q^n}$ 中找到. 这说明, $\mathbf{F}_q$ 上的 n 次不可约多项式必然是 $x^{q^n} - x$ 的因子．要确定有限域 $\mathbf{F}_q$ 上的所有 n 次不可约多项式，需要考察 $x^{q^n} - x$ 在 $\mathbf{F}_q$ 上的因式分解．这和分圆多项式的情形十分类似.

有限域 $\mathbf{F}_q$ 上的代数元可以根据其极小多项式的次数分类．设 P_n 是极小多项式次数恰为 n 的代数元集合，则

$$\mathbf{F}_{q^n} = \bigcup_{d|n} P_d.$$

这对应着因式分解

$$x^{q^n} - x = \prod_{d \mid n} \prod_{\zeta \in P_d} (x - \zeta).$$

因为 n 次不可约多项式有 n 个根，而且这些根的极小多项式的次数都是 n，所以 n 次不可约多项式必然是多项式

$$\prod_{\zeta \in P_n} (x - \zeta) = \prod_{d|n} \left(x^{q^d} - x\right)^{\mu(n/d)}$$

的因子；这个表达式是对前面的因式分解应用 Möbius 反演→9.11.23@p1331 得到的．因为这个多项式的次数是

$$\sum_{d|n}\mu(d)q^{n/d},$$

所以，$\mathbf{F}_q$ 上的 n 次不可约首一多项式共计

$$\frac{1}{n}\sum_{d|n}\mu(d)q^{n/d}$$

个．这恰为不计旋转意义下，q 个颜色的珠子能够串成的长度为 n 的非周期项链的种类个数（证明→9.13.14@p1680），所以又称为项链多项式（necklace polynomial）．

定理

有限域 $\mathbf{F}_q$ 上存在任意次数的不可约多项式．

因为有限域上的不可约多项式有着简单的结构，这使得有限域上的多项式的因式分解十分容易．比如，要确定给定多项式的全体 n 次不可约因子，只要求解给定多项式与 $x^{q^n}-x$ 的最大公因子即可[21]．类似地，只要 n 次多项式对所有的 $k<n$ 都与多项式 $x^{q^k}-x$ 互素，就可以断定该 n 次多项式在 $\mathbf{F}_q$ 上不可约．

前文已经指出，有限域上的不可约多项式的根未必是相应扩域作为有限域的本原元．有限域 $\mathbf{F}_q$ 的本原元在它的素子域 $\mathbf{F}_p$ 上的极小多项式也称为域 $\mathbf{F}_p$ 上的 **本原多项式**[22]（primitive polynomial）．用这样的多项式实现扩域，就可以保证 $\overline{x}$ 必然是扩域中的本原元．域 $\mathbf{F}_p$ 上的 n 次本原多项式可以通过在 $\mathbf{F}_p$ 上对分圆多项式 $\Phi_{p^n-1}(x)$ 进行因式分解得到．

定理

设 p 为素数，n 为正整数，且 $p\perp n$．又设 d 是乘法群 $(\mathbf{Z}/n\mathbf{Z})^\times$ 中元素 p 的阶．那么，分圆多项式 $\Phi_n(x)$ 在域 $\mathbf{F}_p$ 上可以分解为 $\dfrac{\varphi(n)}{d}$ 个 $\mathbf{F}_p$ 上的 d 次不可约多项式的乘积；而且，当且仅当 $n=p^d-1$ 时，这些因子是本原多项式．特别地，分圆多项式 $\Phi_n(x)$ 在域 $\mathbf{F}_p$ 上不可约，当且仅当 p 是模 n 的原根．

证明

如果注意到，n 次分圆多项式的根是所有 $\mathbf{F}_p$ 的 n 次本原单位根，而 n 次本原单位根的极小多项式的次数 d 就是它的共轭（包括自身）的数量，也就等于它在自同构群 $\langle\sigma_p\rangle$ 下的轨道长度，那么就可以知道 d 是 $\{\zeta^i:i\perp n\}$ 中映射 $\zeta^i\mapsto\zeta^{ip}$ 的循环子群的轨道长度，亦即乘法群 $(\mathbf{Z}/n\mathbf{Z})^\times$ 中元素 p 的阶．如果不想依赖于 Galois 理论，也可以通过说明 d 是最小的正整数使得 $(x^n-1)\mid(x^{p^d-1}-1)$ 成立来证明此事．其余结论显然．

虽然不可约多项式对于有限域的实现很重要，但是要找到有限域 $\mathbf{F}_q$ 上的一个 n 次不可约多项式却并没有较好的确定性的方法．在一般的情况下，可以使用随机方法生成这样的不可约多项式．因为所有 n 次首一多项式中，不可约多项式占的比例是 $\Theta(1/n)$ 的，所以可以先随机生成一个 n 次首一多项式再判断它是否可约．这样做可以在生成期望 $\Theta(n)$ 个首一多项式后找到一个不可约多项式．当然，这样生成的不可约多项式未必是本原多项式，系数也未必是简单的．在实际操作中，如果有限域的大小提前给定，往往可以通过查表[23]的方式找到系数简单的本原多项式，方便后续的计算．

[21] 这样的说法有失严谨，因为还会得到次数 $d\mid n$ 的不可约因子．但是，由于算法实现时通常会从较小的次数的因子开始分离，在分离 n 次不可约多项式因子时，较小的因子应该已经分离完了，所以这说法也是可以接受的．

[22] 不要把这里的名称和多项式理论中的本原多项式（即所有系数的最大公因子是一的多项式）混淆．

[23] 比如，Hansen, T., & Mullen, G. L. (1992). Primitive polynomials over finite fields. Mathematics of computation, 59(200), 639-643[3] 的附录就提供了这样的列表．

参考实现

本节提供一个朴素的有限域的实现，仅供参考．代码中实现了随机生成不可约多项式的方法．

参考实现

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

class FiniteField {
  int p, k;
  std::vector<int> mod;  // Monic.

  // Remove leadings zeros of a polynomial.
  static void trim(std::vector<int>& poly) {
    int m = poly.size();
    for (; m && !poly[m - 1]; --m);
    poly.resize(m);
  }

  // Binary exponentiation mod p.
  int pow(int a, int b) const {
    int res = 1, po = a;
    while (b) {
      if (b & 1) res = (long long)res * po % p;
      po = (long long)po * po % p;
      b >>= 1;
    }
    return res;
  }

  // Multiplicative inverse mod p.
  int inv(int a) const { return pow(a, p - 2); }

  // Polynomial GCD. Inputs are supposed to have no leading zeros.
  std::vector<int> poly_gcd(const std::vector<int>& lhs,
                            const std::vector<int>& rhs) const {
    if (lhs.size() < rhs.size()) {
      return poly_gcd(rhs, lhs);
    } else if (rhs.size()) {
      auto rem = lhs;  // remainder.
      long long v = inv(rhs.back());
      for (int i = rem.size() - rhs.size(); i >= 0; --i) {
        auto d = v * (p - rem[i + rhs.size() - 1]) % p;
        for (int j = 0; j < rhs.size(); ++j) {
          rem[i + j] = (rem[i + j] + d * rhs[j]) % p;
        }
      }
      trim(rem);  // Remove leading zeros.
      return poly_gcd(rhs, rem);
    } else {
      return lhs;
    }
```

```
  }

  // Polynomials Ex-GCD. Inputs are supposed to have no leading zeros.
  void poly_ex_gcd(const std::vector<int>& lhs, const std::vector<int>& rhs,
                   std::vector<int>& x, std::vector<int>& y) const {
    if (lhs.size() < rhs.size()) {
      poly_ex_gcd(rhs, lhs, y, x);
    } else if (rhs.size()) {
      std::vector<int> quo(lhs.size() - rhs.size() + 1);  // quotient.
      auto rem = lhs;                                     // remainder.
      long long v = inv(rhs.back());
      for (int i = rem.size() - rhs.size(); i >= 0; --i) {
        quo[i] = v * rem[i + rhs.size() - 1] % p;
        long long d = p - quo[i];  // d = -quo[i].
        for (int j = 0; j < rhs.size(); ++j) {
          rem[i + j] = (rem[i + j] + d * rhs[j]) % p;
        }
      }
      trim(rem);  // Remove leading zeros.
      // Recursively ex_gcd.
      poly_ex_gcd(rhs, rem, y, x);
      // y -= a/b*x.
      if (y.size() < quo.size() + x.size() - 1) {
        y.resize(quo.size() + x.size() - 1, 0);
      }
      for (int i = 0; i < quo.size(); ++i) {
        for (int j = 0; j < x.size(); ++j) {
          y[i + j] = (y[i + j] - (long long)quo[i] * x[j]) % p;
          if (y[i + j] < 0) y[i + j] += p;
        }
      }
      trim(y);  // Remove leading zeros.
    } else {
      // x = 1, y = 0.
      x.assign(1, inv(lhs.back()));
      y.clear();
    }
  }

public:
  // Class for Finite Field Elements.
  struct Element {
    const FiniteField* gf;
    std::vector<int> a;

    // Element initialization as zero.
    Element(const FiniteField* gf) : gf(gf), a(gf->k) {}

    // Element initialization from the numeric representation.
    Element(const FiniteField* gf, int id) : gf(gf), a(gf->k) {
      for (int i = 0; i < gf->k; ++i) {
        a[i] = id % gf->p;
```

```
        id /= gf->p;
      }
    }

    // Generate the numeric representation from an element.
    int idx() const {
      int id = 0;
      for (int i = gf->k - 1; i >= 0; --i) {
        id = id * gf->p + a[i];
      }
      return id;
    }

    // Access the i-th coefficient.
    int& operator[](int i) { return a[i]; }

    // Addition.
    Element& operator+=(const Element& rhs) {
      for (int i = 0; i < gf->k; ++i) {
        a[i] += rhs.a[i];
        if (a[i] >= gf->p) a[i] -= gf->p;
      }
      return *this;
    }

    // Addition.
    Element operator+(const Element& rhs) const {
      Element res(*this);
      res += rhs;
      return res;
    }

    // Subtraction.
    Element& operator-=(const Element& rhs) {
      for (int i = 0; i < gf->k; ++i) {
        a[i] -= rhs.a[i];
        if (a[i] < 0) a[i] += gf->p;
      }
      return *this;
    }

    // Subtraction.
    Element operator-(const Element& rhs) const {
      Element res(*this);
      res -= rhs;
      return res;
    }

    // Multiplication by a scalar.
    Element& operator*=(int x) {
      for (int i = 0; i < gf->k; ++i) {
        a[i] = (long long)a[i] * x % gf->p;
```

```
153       }
154       return *this;
155     }
156
157     // Multiplication by a scalar.
158     Element operator*(int x) const {
159       Element res(*this);
160       res *= x;
161       return res;
162     }
163
164     // Multiplication by x.
165     Element& shift() {
166       long long d = gf->p - a.back();  // d = -a[k-1].
167       for (int i = gf->k - 1; i >= 0; --i) {
168         a[i] = ((i ? a[i - 1] : 0) + d * gf->mod[i]) % gf->p;
169       }
170       return *this;
171     }
172
173     // Multiplication.
174     Element& operator*=(const Element& rhs) {
175       Element prod(*this);
176       *this *= rhs.a[0];
177       for (int j = 1; j < gf->k; ++j) {
178         prod.shift();
179         *this += prod * rhs.a[j];
180       }
181       return *this;
182     }
183
184     // Multiplication.
185     Element operator*(const Element& rhs) const {
186       Element res(*this);
187       res *= rhs;
188       return res;
189     }
190
191     // Binary exponentiation.
192     Element pow(int b) const {
193       Element res(gf, 1);
194       Element po(*this);
195       while (b) {
196         if (b & 1) res *= po;
197         po = po * po;
198         b >>= 1;
199       }
200       return res;
201     }
202
203     // Multiplicative inverse.
204     Element inv() const {
```

```
205       Element res(gf);
206       auto& x = res.a;
207       std::vector<int> y;
208       auto lhs = a;
209       trim(lhs);
210       auto rhs = gf->mod;
211       gf->poly_ex_gcd(lhs, rhs, x, y);
212       x.resize(gf->k);
213       return res;
214     }
215
216     // Division.
217     Element& operator/=(const Element& rhs) {
218       *this *= rhs.inv();
219       return *this;
220     }
221
222     // Division.
223     Element operator/(const Element& rhs) const {
224       Element res(*this);
225       res /= rhs;
226       return res;
227     }
228   };
229
230   private:
231    // Check whether the current MOD is irreducible.
232    bool checkIrreducible() const {
233     Element f(this, p);
234     for (int j = 1; j < k; ++j) {
235       // F = X^{p^j} mod MOD.
236       f = f.pow(p);
237       // G = X^{p^j}-X mod MOD.
238       auto g = f.a;
239       g[1] = g[1] ? g[1] - 1 : p - 1;
240       trim(g);
241       // H = MOD.
242       auto h = mod;
243       // Reducible if deg(GCD(G,H))>0.
244       if (poly_gcd(h, g).size() > 1) return false;
245     }
246     return true;
247   }
248
249   public:
250    FiniteField(int p, int k) : p(p), k(k), mod(k + 1, 1) {
251     do {  // Randomly generate a polynomial.
252       for (int i = 0; i < k; ++i) {
253         mod[i] = rand() % p;
254       }
255     } while (!checkIrreducible());
256   }
```

```
257 };
258
259 int main() {
260   int p = 13331, k = 50;
261   FiniteField gf(p, k);
262   FiniteField::Element e1(&gf, rand() + 1);
263   FiniteField::Element e2(&gf, rand() + 1);
264   FiniteField::Element e3(&gf, rand() + 1);
265   // Test Frobenius endomorphism.
266   std::cout
267       << ((e1 * e2 + e3).pow(p) - e1.pow(p) * e2.pow(p) - e3.pow(p)).idx();
268   // Test inverse.
269   std::cout << ((e1 * e2).inv() - e1.inv() / e2).idx();
270   return 0;
271 }
272
```

密码学上用的最多的是特征为 2 的有限域．对于这类有限域，可以将域中的元素存储为 01 串，用位运算的方式实现域中的操作．

应用

本节列举一些域扩张在算法竞赛中的应用．最主要的情形，就是在对域上的算术表达式进行计算时，需要在中间过程引入一些原本的域中并不存在的元素，从而使得直接的计算成为可能．读者应当熟悉利用复数解决实数问题的情形，这就是实数域上的扩张的例子；读者相对陌生的，可能是有限域上的扩张．所以，这里主要讨论有限域上的扩张，尤其是素域 $\mathbf{F}_p$ 上的扩张．

有些情形下，域扩张可以降低计算的复杂度，故而是必要的，例如实数域上的 快速傅里叶变换 →9.12.3@p1527；有些情形下，域扩张只是众多解决问题方法中的一种，且通常有类似复杂度的方法可以避免使用域扩张，例如马上要提到的对斐波那契数列的计算．读者在理解这些应用的同时，应当比较不同方法的优劣，从而能够在解决问题时选择合适的方法．

斐波那契数列

对于 斐波那契数列→9.13.4@p1629 的计算，常见方法有 $O(n)$ 的线性递推和 $O(\log n)$ 的矩阵快速幂．实际上，它还可以通过域扩张的方法加以解决，时间复杂度同样是 $O(\log n)$．斐波那契数列有通项公式：

$$f(n)=\frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n-\left(\frac{1-\sqrt{5}}{2}\right)^n\right).$$

现在要计算 $f(n)$ 在素数模 $p\neq 5$ 下的值[24]．将这个问题转化为有限域 $\mathbf{F}_p$ 上的计算，首先要解决的就是 $\sqrt{5}$ 在 $\mathbf{F}_p$ 中的意义．从代数角度看，它就是元素 5 的平方根．因而，如果 $\mathbf{F}_p$ 中存在 5 的平方根，即 5 是模 p 的二次剩余的时候，可以直接计算其二次剩余并带入计算；否则，就需要在扩域 $\mathbf{F}_p(\sqrt{5})\cong\mathbf{F}_p[x]/(x^2-5)$ 下进行计算．

当然，在扩域中进行计算的时候，没有必要一定加入 $\sqrt{5}$．比如说，对于斐波那契数列，也可以设 ϕ 是多项式 x^2-x-1 的根，从而 $f(n)$ 可以写作

$$f(n)=\frac{\phi^n-(-\phi)^{-n}}{2\phi-1}=\frac{\phi^n-(1-\phi)^n}{2\phi-1}.$$

如果 5 并非模 p 下的二次剩余，多项式 x^2-x-1 就是不可约的．此时，可以在扩域 $\mathbf{F}_p(\phi)\cong\mathbf{F}_p[x]/(x^2-x-1)$ 上进行计算，能够得到和前文一致的结果．

[24] 当 $p=5$ 时，Fibonacci 数列的特征方程 $x^2-x-1=0$ 有重根 $x=3$，因而在 $\mathbf{F}_5$ 中，Fibonacci 数列的通项公式是 $f(n)=n3^{n-1}$.

计算斐波那契数列的方法当然可以推广到别的情形. 但是, 有一点应当注意: 有限域上多项式的不可约性和有理数域并不一致. 譬如 x^4-10x^2+1, 它在 $\mathbf{Q}$ 上是不可约的, 相应的分裂域是 $\mathbf{Q}(\sqrt{2}+\sqrt{3})=\mathbf{Q}(\sqrt{2},\sqrt{3})$; 但是在 $\mathbf{F}_p$ 中, 如果 2 和 3 都不是模 p 的二次剩余, 那么它是两个不可约多项式的乘积, 亦即在扩域 $\mathbf{F}_p(\sqrt{2})$ 中就已经存在平方根 $\sqrt{3}$ 而不需要进一步扩张.

推广到环上的「扩张」

正如上一节所展示的那样, 域的扩张有着各种各样的限制. 对于斐波那契数列的计算, 只用扩域的方法只能解决模数 p 是素数且 5 不是 p 的二次剩余的情形. 但是应当注意, 在 代数扩张→9.16@p1805 一节中的论述表明, 如果不要求在扩张后的结构中做除法运算, 那么可以对环进行扩张[25]. 本节以任意模数 m 下斐波那契数列的计算为例, 简要讨论这种方法. 其他的不涉及过多除法运算的常见情景, 包括行列式的计算、快速傅里叶变换等, 有必要的时候都可以尝试应用这种方法.

设 m 为任意正整数, $f(n)$ 为斐波那契数列的第 n 项. 问题是要计算 $f(n)\bmod m$ 的值. 原则上, 需要在 $\mathbf{Z}/m\mathbf{Z}$ 上进行计算. 但是, 正如上节所表明的, 在不同模数下, 多项式 x^2-x-1 的可约性和有无重根的情形不一致, 所以斐波那契数列的通项可能相差甚远. 而且, 如果本身 $\mathbf{Z}/m\mathbf{Z}$ 并不是域, 扩张后的元素也往往没有合法的逆 (比如模 5 的时候, 分母 $\sqrt{5}$ 直接就是零). 虽然有着诸多问题, 但是其实在系数对 m 取模的条件下, 计算余数

$$(1-x)^n-x^n \bmod x^2-x-1$$

的常数项即可. 比对上文中的通项公式, 这个做法的合理性是显然的: 这似乎就是在「扩张」$(\mathbf{Z}/m\mathbf{Z})[x]/(x^2-x-1)$ 中计算

$$f(n)=\frac{(1-\phi)^n-\phi^n}{1-2\phi}$$

的值, 且 ϕ 是 x^2-x-1 的根.

虽然没有那么显然, 但是这个做法也是合法的. 注意到, 在有理数域的扩张 $\mathbf{Q}(\phi)\cong\mathbf{Q}[x]/(x^2-x-1)$ 中, 通项公式成立. 这就说明, 在 $\mathbf{Q}[x]$ 中,

$$(1-x)^n-x^n\equiv f(n)(1-2x)\pmod{x^2-x-1}$$

成立. 这只涉及到整系数多项式, 因而在 $\mathbf{Z}[x]$ 中也成立. 写成带余除法, 等式两边的系数都对 m 取模, 就得到 $(\mathbf{Z}/m\mathbf{Z})[x]$ 上的恒等式. 这个结论对于任意 m 都成立.

一般的情况下, 如果某个表达式可以在有理数域 $\mathbf{Q}$ 的扩域上进行计算, 那么就一定可以通过约去分母的方式得到 $\mathbf{Z}[x]$ 上的结论, 再对 m 取模就得到 $(\mathbf{Z}/m\mathbf{Z})[x]$ 上的结论. 这种思路能够行得通的关键在于, 约去分母这一步不应该造成「不可挽回」的后果. 比如, 对于斐波那契数列的计算, 如果不用常数项的系数, 而是用一次项的系数, 那么因为系数有因子 2, 那么 2 在模 m 是偶数的情况下就没有逆元, 没有办法恢复 $f(n)$ 的值; 再比如, 同样是斐波那契数列的计算, 如果使用带有 $(-\phi)^{-n}$ 项的通项公式, 那么约去分母的步骤会引入难以处理的因子, 从而无法从取余后的结果得到结论. 因而, 对于计算过程的选择, 是能够应用这一技巧的关键.

Cipolla 算法

这是利用有限域的扩域进行计算的典型例子. 对于模 $p\neq 2$ 下的二次剩余 a, 要找到它的平方根, 即使得 $x^2\equiv a\pmod p$ 成立的 x. 虽然这是 $\mathbf{F}_p$ 上的问题, 但是 Cipolla 算法→9.11.17@p1263 在有限域 $\mathbf{F}_{p^2}$ 上进行计算. 本节使用域论的语言对这个算法进行说明. 初等数论的证明可以参考所给链接.

具体来说, Cipolla 算法首先选择 r 使得 r^2-a 是模 p 的二次非剩余, 这意味着 $x^2-(r^2-a)$ 是不可约多项式. 因而, 令 $u=r^2-a$, 可以考虑扩域 $\mathbf{F}_p(\sqrt{u})$. 因为 Frobenius 自同态只能将元素映射到它的共轭, 而这样的共轭在二次扩张中是唯一的, 即有 $(r-\sqrt{u})^p=r+\sqrt{u}$. 故而, $(r-\sqrt{u})^{p+1}=(r+\sqrt{u})(r-$

[25] 所谓的环上的扩张, 通常有两种含义: 一种是 群的扩张的推广[4], 一种是 域的扩张的推广[5]. 本文指的是第二种含义. 更具体地说, 本节涉及的扩张都是交换幺环上的 整扩张[6], 它是域上的代数扩张的概念在交换幺环上的推广.

$\sqrt{u}) = r^2 - u = a$. 所以，要确定平方根，只要计算 $(r - \sqrt{u})^{(p+1)/2}$ 就好了．这个值必然位于 $\mathbf{F}_p$ 中，因为 $x^2 - a$ 的分裂域就是 $\mathbf{F}_p$ 本身．

习题

最后，列举一些直接应用本文内容的题目，以便加深理解．但应注意，很多内容并不是算法竞赛的常规考点．

- 分圆多项式：
  - Luogu P1520 因式分解[7]
  - Gym102114C Call It What You Want[8]
- 有限域：
  - Luogu P3923 大学数学题[9]
  - [COTS 2021] 菜 Jelo[10]
  - CF1310F. Bad Cryptography[11]
  - LOJ 178. 多项式求根[12]
- 域扩张：
  - [Oleksandr Kulkov Contest 2] Problem A. Square Root Partitioning[13]
  - CF1103E. Radix Sum[14]

参考资料与注释

- Dummit, D.S. and Foote, R.M. (2004) Abstract Algebra. 3rd Edition, John Wiley & Sons, Inc.
- Milne, J.S. Fields and Galois Theory.[15]
- Factorization of polynomials - Wikipedia[16]
- Factorization of polynomials over finite fields - Wikipedia[17]
- Cyclotomic Polynomial - Wikipedia[18]
- Brett Porter's Notes on Cyclotomic Polynomials[19]
- Jordan Bell's Notes on Cyclotomic Polynomials[20]
- Michel Waldschmidt. An introduction to the theory of finite fields[21]
- Finite Field Arithmetic - Wikipedia[22]

[1] 始对象

[2] 万有性质

[3] Hansen, T., & Mullen, G. L. (1992). Primitive polynomials over finite fields. Mathematics of computation, 59(200), 639-643

[4] 群的扩张的推广

[5] 域的扩张的推广

[6] 整扩张

[7] Luogu P1520 因式分解

[8] Gym102114C Call It What You Want

[9] Luogu P3923 大学数学题

[10] [COTS 2021] 菜 Jelo

[11] CF1310F. Bad Cryptography

[12] LOJ 178. 多项式求根

[13] [Oleksandr Kulkov Contest 2] Problem A. Square Root Partitioning

[14] CF1103E. Radix Sum

[15] Milne, J.S. Fields and Galois Theory.

[16] Factorization of polynomials - Wikipedia

[17] Factorization of polynomials over finite fields - Wikipedia

[18] Cyclotomic Polynomial - Wikipedia

[19] Brett Porter's Notes on Cyclotomic Polynomials

[20] Jordan Bell's Notes on Cyclotomic Polynomials

[21] Michel Waldschmidt. An introduction to the theory of finite fields

[22] Finite Field Arithmetic - Wikipedia

9.16.5 Schreier–Sims 算法

Authors: Enter-tainer, iamtwz, Ir1d, isdanni, ksyx, StudyingFather, Tiphereth-A, Xeonacid, c-forrest

前置知识：抽象代数基本概念→9.16.1@p1805、群论→9.16.2@p1812、置换与排列→9.8@p1138

引入

Schreier–Sims 算法 是计算群论（computational group theory）的一种算法，以数学家 Otto Schreier 和 Charles Sims 的名字命名．该算法能够在多项式时间内解决诸如找到有限置换群的阶数、查看给定置换是否包含在所给群中等许多问题．Schreier–Sims 算法最早由 Sims 在 1970 年基于 Schreier 引理引入．在 1981 年[26]，Donald Knuth 进一步改进了该算法的运行时间．后来，该算法又发展出来一种更快的随机化版本．计算机代数系统（例如 GAP 和 Magma）通常使用该算法的高度优化过的 Monte Carlo 版本[27]．

记号

本文依照计算群论文献的惯例，将群作用记作右作用，这意味着置换的复合由左向右进行．本文涉及的群作用都可以视为置换作用，尽管部分算法对于更广泛的群作用也成立．相应地，群作用的集合默认为 $X=\{1,2,\cdots,n\}$，其中的元素则称为点．置换 g 作用在点 x 上得到的结果记作 x^g，有时也称置换 g 将点 x 移动到点 x^g．最后，置换群 G 作用下，点 x 的轨道记作 $x^G=\{x^g:g\in G\}$，它的稳定化子则记作 $G_x=\{g\in G:x^g=x\}$．稳定化子的概念还可以推广到集合 $B\subseteq X$，它的稳定化子定义为 $G_B=\bigcap_{x\in B}G_x$．

概述

Schreier–Sims 算法主要试图解决这样一个问题：

- 给定大小为 n 的集合 X 上的一些置换组成的集合 S，如何在计算机中高效地存储由 S 生成的置换群 $G=\langle S\rangle$，并完成一系列对该群的查询任务？

显然，这样的群 G 的规模可能很大，且远远大于集合 X 和生成集 S 的规模．比如，n 次对称群 $S_n=\langle(123\cdots n),(12)\rangle$ 的大小为 $n!$，但是它可以仅由两个置换生成．存储群中的每一个元素是不现实的．

类似于利用 Gauss 消元法→9.19.3@p1944 构建出向量空间的一组 线性基→9.14.8@p1738，Schreier–Sims 算法的思路是找到有限置换群 G 的一组「基」：

1. 算法的输入是 G 的一个生成集 S，其中有若干个置换；
2. 如果群 G 不平凡，总能找到在群 G 作用下位置会发生变化的点 β，即 $|\beta^G|>1$；
3. 找到点 β 的轨道 $\Delta=\beta^G$，并对轨道中的每个点 $\delta\in\Delta$，都找到群 G 中一个置换 t_δ，它能够将点 β 移动到 δ；
4. 找到点 β 的稳定化子 G_β 的一个生成集 S'；
5. 递归地对 $G'=\langle S'\rangle$ 调用该算法，直到得到平凡的群 $\{e\}$．

这个思路的合理性在于，点 β 的稳定化子 G_β 是群 G 的子群，它的全体（右）陪集构成群 G 的分划，且这些陪集和点 β 的轨道 β^G 一一对应，步骤 3 中求得的那些置换 t_δ 就是这些陪集的代表元．这个陪集代表元的集合 T 称为稳定化子 G_β 的 **陪集代表系**（transversal）．换句话说，群 G 中的每个元素 g 都和唯一的一对元素 $(h,t)\in G_\beta\times T$ 对应，且 $g=ht$．因而，只要能想办法存储子群 G_β 和相应的陪集代表系 T，就可以存储整个群 G．然而，存储子群 G_β 的问题也已经解决了——只要递归调用算法即可．

当然，算法的实现还有很多细节需要梳理，这是本文的主要内容．但是在那之前，首先要考察调用该算法之后得到的结果，看看算法将群存储为怎样的结构，能够解决怎样的查询问题．为此，应当明晰一些概念．

[26] Knuth 的论文是在 1991 年发表的，但是他的改进在 1981 年就通过会议广泛地宣传．论文是基于他的会议讲稿写作的．

[27] 不要与 Monte Carlo 方法混淆．此处的 Monte Carlo 算法是指出错概率恒定且任意小的随机算法．

稳定化子链

假设对群 $G = \langle S \rangle$ 调用该算法，共进行了 k 次；且在第 i 次调用该算法时，输入是置换的集合 $S^{(i-1)}$，找到的点是 β_i，得到的陪集代表系是 T_i，得到的稳定化子的生成集是 $S^{(i)}$. 如果记 $G^{(i)} = \langle S^{(i)} \rangle$，那么，算法实际上得到了子群链

$$G = G^{(0)} > G^{(1)} > \cdots > G^{(k-1)} > G^{(k)} = \{e\}.$$

而且，每一个链中的子群都是一个稳定化子

$$G^{(i)} = G_{\beta_1,\cdots,\beta_i}.$$

因而，Schreier–Sims 算法可以看作就是在计算这样一个 **稳定化子链**（stabilizer chain）.

基和强生成集

如果集合 X 的子集 B 满足 $G_B = \{e\}$，就称 B 是置换群 G 的一组 **基**（base）. 显然，上述算法得到了一组基 $B = \{\beta_1, \cdots, \beta_k\}$. 这意味着，置换 $g \in G$ 对这些点作用的结果在群 G 中唯一地确定了这个置换. 而且，算法输出的这个基还满足条件

$$G^{(i-1)} = G_{\beta_1,\cdots,\beta_{i-1}} > G_{\beta_1,\cdots,\beta_{i-1},\beta_i} = G^{(i)}.$$

这就是说，基中的每个点都蕴含着关于群中的元素的有效信息. 这样的基称为 **无冗余的**（nonredundant）. 算法总是输出无冗余的基. 这样的基对应的稳定化子链是严格递降的.

同时，算法得到的生成集的并集

$$\bar{S} = \bigcup_{i=0}^{k} S^{(i)}$$

也是群 G 的生成集，且成立 $\langle \bar{S} \cap G^{(i)} \rangle = G^{(i)}$. 满足这个条件的生成集称为群 G 相对于基 B 的 **强生成集**（strong generating set）. 因而，Schreier–Sims 算法也可以看作是在计算群 G 的 **基和强生成集**（base and strong generating set, BSGS）. 这个说法和稳定化子链的说法是等价的，本文不加以区分.

当然，算法还得到了一系列轨道 $\Delta_i = \beta_i^{G^{(i-1)}}$ 和相应的陪集代表系 T_i. 这些轨道称为群 G 的 **基础轨道**（basic orbits）. 当本文提及群 G 的稳定化子链或者基和强生成集时，总是默认相应的基础轨道和陪集代表系都已经一并求出.

数据结构

本文会提供一系列伪代码. 伪代码中，群的稳定化子链（或基和强生成集）存储在数据结构 C 中：

$$C = (S, \Delta, T, C').$$

这个结构中的数据成员分别是当前群的生成集 S、基础轨道 Δ、相应的陪集代表系 T 和存储为嵌套子结构的稳定化子 C'. 当然，稳定化子 C' 也是这样的一个结构. 整个群实际存储在一个层状结构中，每层都描述了稳定化子链中的一个群. 最内层是空的结构体，表示 $G^{(k)} = \{e\}$.

在伪代码中，该数据结构中的成员可以分别由 $C.generators$、$C.orbit$、$C.transversal$ 和 $C.next$ 访问. 轨道的首个元素 $C.orbit[0]$ 默认是基中的点 β，而相应的陪集代表元 $C.transversal[\beta]$ 默认是恒等变换 e. 注意，虽然此处用数组下标访问了轨道和陪集代表系中的元素，但是它们未必存储为数组，而应当理解为它们提供了访问轨道首元素和根据轨道中的点查询相应的陪集代表元的方法. 下文会讨论具体的实现细节.

应用

在获得群的基和强生成集后，能够解决一系列关于群的查询问题. 其中，最基础的，也是算法竞赛中最常遇到的，是查询群的阶数和查询某个置换是否属于给定群的问题.

群的阶数

如果已知群 G 的基和强生成集，那么应用 Lagrange 定理和轨道稳定化子定理可知，群 G 的阶数可以计算为

$$\mid G \mid = \prod_{i=1}^{k} [G^{(i-1)} : G^{(i)}] = \prod_{i=1}^{k} [G^{(i-1)} : G_{\beta_i}^{(i-1)}] = \prod_{i=1}^{k} \mid T_i \mid .$$

所以，只要将所有陪集代表系 T_i 的大小（或者等价地，基础轨道 Δ_i 的长度）乘起来就可以得到群 G 的阶数.

成员判定

已知群 G 的基和强生成集，也可以判定某个置换 h 是否属于群 G. 这称为 **成员判定**（membership testing）问题.

这个问题可以递归地解决. 要判定 $h \in G^{(i-1)}$，首先要找到 $G^{(i)}$ 的包含 h 的陪集的代表元 $t \in T_i$. 如果能够找到，那么设 $h' = ht^{-1}$，就有 $h = h't$ 且 $h \in G^{(i-1)}$ 等价于 $h' \in G^{(i)}$；问题就转化为判定 $h' \in G^{(i)}$. 如果找不到这样的 t，或者已经递归到了 $G^{(k)} = \{e\}$ 但是 $h \neq e$，就可以得出结论，$h \notin G$. 其实，这个过程不仅判定了 $h \in G$，而且在 $h \in G$ 的情形下，还能够将 h 表示为一系列陪集代表元的乘积 $t_k \cdots t_2 t_1$，其中，$t_i \in T_i$. 对于群 G 中的元素，这样的表示存在且唯一. 这再次证明了上面关于群的阶数的公式是正确的.

现在将该过程写成如下伪代码：

Algorithm MembershipTest(C, h) :
Input. A stabilizer chain C for a group G and a permutation h.
Output. Whether $h \in G$.
Method.

```
1  while C is not empty
2      β ← C.orbit[0]
3      δ ← β^h
4      if δ ∈ C.orbit then
5          t ← C.transversal[δ]
6          h ← ht^{-1}
7      else
8          return false
9      end if
10     C ← C.next
11 end while
12 return h = e
```

下文会看到，成员判定问题也是本文所讨论的 Schreier–Sims 算法的实现中的一个重要组成部分.

轨道、陪集代表系和稳定化子的计算

要实现 Schreier–Sims 算法，首先要解决如下子问题：[28]

- 给定群 G 的生成集 S 和一个点 β，如何求出轨道 β^G、相应的陪集代表系 T 和稳定化子 G_β 的生成集?

这就是本节要解决的问题.

轨道和陪集代表系的存储

要求得轨道和陪集代表系，只要直接搜索就好了. 伪代码如下：

[28]这个问题以及本节的算法都并不需要假设所讨论的群作用是置换作用，因而可以应用于更广泛的场景. 比如，如果将这些算法应用于共轭作用，同样可以求得轨道（共轭类）、陪集代表系和稳定化子（中心化子）.

Algorithm OrbitTransversal(S, β) :
Input. A generating set S for a group G and a point β.
Output. The orbit $\Delta = \beta^G$ and the transversal T.
Method.

```
1  Δ ← [β]
2  T[β] ← e
3  for δ ∈ Δ
4      for s ∈ S
5          γ ← δ^s
6          if γ ∉ Δ then
7              append γ to Δ
8              T[γ] ← T[δ] · s
9          end if
10     end for
11 end for
12 return Δ, T
```

具体实现的时候，使用广度优先搜索和深度优先搜索都是可以的．搜索遍历到的状态数目是 $|S||T|$，只要能合理地存储轨道和陪集代表系，时间复杂度是完全可以接受的．

由于在置换群的语境下，轨道无非是至多 n 个点的集合，为了高效完成查找和添加操作，可以使用布尔值数组或是无序集合存储．这样两个操作的时间复杂度都是 $O(1)$ 的，整体的空间占用是 $O(n)$ 的．当然，取决于陪集代表系的实现，可能还需要额外标记首元素的位置．

问题在于使用什么样的数据结构存储相应的陪集代表系 T.

直接存储

最简单的方法，当然是直接存储陪集代表系 T 中的每一个元素 t．单个置换存储为 单行记号 →9.8@p1138，需要的空间恰为 n，所以存储这样的陪集代表系的空间复杂度是 $O(|T|n)$ 的．这样做的好处是访问单个陪集代表元的时间复杂度是 $O(1)$ 的，代价是初次计算这些陪集代表元的时间复杂度是 $O(|T|n)$ 的.

Schreier 树

另外一种常见的做法是实现一个树形结构用于存储陪集代表系．它称为 **Schreier 树**（Schreier tree）或 **Schreier 向量**（Schreier vector）[29]．它以 β 为根，以轨道 Δ 中的元素 δ 为顶点．每次在搜索过程中得到新的顶点 $\gamma = \delta^s$ 时，就从旧的顶点 δ 到新的顶点 γ 连一条边，边上记录生成集 S 中置换 s 的序号（或指针）．因为已经存储了生成集，存储陪集代表系的额外空间复杂度是 $O(|T|)$ 的．对于 n 的规模很大的情形，这样做可以有效地节约空间，而且初次计算的时候复杂度是 $O(|T|)$ 的．但是，副作用就是每次需要获得陪集代表元的时候，需要遍历顶点到根的路径上的边，重新计算陪集代表元，因而时间复杂度高度依赖于 Schreier 树的深度．对于一般的情形，树的深度可能达到 $O(n)$ 级别.

在具体实现的时候，需要根据实际情况权衡算法的时空复杂度．对于算法竞赛可能涉及的情形，n 通常都不大，空间充足，而时间复杂度常常成为瓶颈．稍后会看到，Schreier–Sims 算法最耗时的步骤恰好需要多次访问陪集代表元，因而使用直接存储的方式往往更优．但是对于某些应用场景，n 可能很大，存储空间可能更为紧张，就有可能需要使用 Schreier 树的方式存储陪集代表系．对于这种情况，为避免 Schreier 树深度过深，研究者提出了很多方法，可以在树的深度过深的时候重构出浅的 Schreier 树．有兴趣的读者可以参考文末的文献.

在伪代码中，本文不会区分具体的陪集代表系的实现，而只假设存储陪集代表系 T 的数据结构实现了根据轨道元素 $\delta \in \Delta$ 访问和修改对应陪集代表元 $T[\delta]$ 的操作.

Schreier 引理

在获得了轨道 β^G 和陪集代表系 T 后，Schreier 引理继而提供了获得稳定化子 G_β 的生成集的方法.

[29]因为这个树形结构可以通过一列指向父节点的指针来实现，所以也称作 Schreier 向量.

Schreier 引理

设群 $G=\langle S\rangle$ 有子群 $H\leq G$. 设 T 是子群 H 的（右）陪集代表系，且 $e\in T$[30]，并记 $g\in G$ 所在陪集的代表元 $t\in T$ 为 $\overline{g}$. 那么，集合

$$U=\{ts(\overline{ts})^{-1}:t\in T,s\in S\}$$

是子群 H 的一个生成集. 它的元素称为子群 H 的 **Schreier 生成元**（Schreier generator）.

证明

首先，根据陪集代表元的定义可知，$ts(\overline{ts})^{-1}\in H$ 对所有 $t\in T,s\in S$ 都成立，故而 $\langle U\rangle\subseteq H$.

反过来，对于任何 $h\in H$，因为 S 是 $G\geq H$ 的生成集，必然存在一列 $s_i\in S\cup S^{-1}$ 使得

$$h=s_1s_2\cdots s_r$$

成立. 令 $t_1=e$，并递归地定义 $t_{i+1}=\overline{t_is_i}\in T$，于是，有

$$h=(t_1s_1t_2^{-1})(t_2s_2t_3^{-1})\cdots(t_rs_rt_{r+1}^{-1})t_{r+1}.$$

而对于每个 $i=1,2,\cdots,r$ 都有 $t_is_it_{i+1}^{-1}=t_is_i(\overline{t_is_i})^{-1}\in U\cup U^{-1}\subseteq H$，故而有 $t_{r+1}\in H$. 但是，$H\cap T=\{e\}$，所以有 $t_{r+1}=e$. 这就说明，任意 $h\in H$ 都可以写作一列 $u_i=t_is_it_{i+1}^{-1}\in U\cup U^{-1}$ 的乘积，亦即 U 生成 H.

因为陪集代表系 T 对应的子群就是稳定化子 G_β，所以求出陪集代表系 T 后再结合群 G 的生成集 S 就能得到稳定化子 G_β 的生成集.

算法

只要对上面的伪代码稍作修改，就能在计算轨道和陪集代表系的同时得到相应的稳定化子的生成集：

Algorithm OrbitTransversalStabilizer(S,β) :
Input. A generating set S for a group G and a point β.
Output. The orbit $\Delta=\beta^G$, the transversal T, and a
generating set S' for the stabilizer G_β.
Method.

```
1  Δ ← [β]
2  T[β] ← e
3  S' ← [e]
4  for δ ∈ Δ
5      for s ∈ S
6          γ ← δ^s
7          if γ ∉ Δ then
8              append γ to Δ
9              T[γ] ← T[δ] · s
10         else
11             append T[δ] · s · T[γ]^{-1} to S'
12         end if
13     end for
14 end for
15 return Δ, T, S'
```

伪代码中，对于每对 (δ,s)，只有轨道中不产生新的元素时，$t_\delta st_{\delta^s}^{-1}$ 才是新的 Schreier 生成元；否则，它就是恒等变换 e. 因而，算法中实际生成的 Schreier 生成元（不计最初的恒等变换）最多只有

$$|S||T|-(|T|-1)=(|S|-1)|T|+1$$

个. 对于一般的情形，这个上界是紧的[31]. 但是，对于实际要处理的有限群，这个上界相当地宽松：这些新得到的 Schreier 生成元大多数都是之前得到的生成元的重复，或者可以由之前的生成元复合而成.

[30]这个 $e\in T$ 的条件对于 Schreier 引理的成立不是必要的.

[31]Nielsen–Schreier 定理[1] 说明，对于由 n 个生成元生成的 自由群[2]，它的指数为 k 的子群是由 $k(n-1)+1$ 个生成元生成的自由群.

因为 Schreier–Sims 算法的基本流程可以实现为递归地调用上述计算轨道和稳定化子的算法，所以其实此时就已经得到了 Schreier–Sims 算法的一种朴素实现．但是，如果不加以筛选，Schreier 生成元的规模的增长速度是指数级的：反复应用 $|S^{(i)}| = O(|S^{(i-1)}| \, |T_i|)$ 可知，最内层的稳定化子的生成集的规模将达到 $O(|S| \, |G|)$．这显然低效得荒诞，因为最内层的稳定化子是 $\{e\}$．

Sims 的工作提供了限制 Schreier 生成元的规模的增长速度的方法，它能够保证最终得到的强生成集 $\bar{S}$ 的大小是 $O(n^2)$ 的．这样就可以在多项式时间内计算基和强生成集．

Schreier–Sims 算法

为解决上述问题，本节讨论 Schreier–Sims 算法的一种增量实现，它得到的强生成集的大小是 $O(n^2)$ 的．

筛选

Schreier–Sims 算法对上述朴素算法的核心优化十分简明：它要求在向稳定化子的生成集添加任何 Schreier 生成元之前，都首先需要经过 **筛选**（sifting）．所谓筛选，就是首先判定新的 Schreier 生成元是否已经存在于已有的生成元生成的子群中，然后只添加那些尚不存在的生成元．为此，只需要使用前文的成员判定算法 $\mathrm{MembershipTest}(C, h)$ 即可．

但是，能够这样做的前提是，基于当前群的已经产生了的 Schreier 生成元，早就构建好了它们生成的群的稳定化子链（或基和强生成集）．这意味着，每次向稳定化子的生成集中添加一个新的 Schreier 生成元，都需要动态地维护内层的稳定化子链，以用于之后的筛选．但是，当前层每插入一个生成元，可能会产生多个 Schreier 生成元，也就可能会多次更新内层结构；而内层结构的每次更新，都可能会引发更内层结构的多次更新．

似乎之前提到的指数级爆炸的问题依然存在．其实不然．因为提前做好了筛选，只有待添加的生成元真的会引发某一层结构的扩大的时候，该层结构才会更新．这说明，单层结构更新的次数实际上等于该层结构存储的群严格增长的次数．但是，大小为 $|G|$ 的群的子群链长度至多为 $\log_2|G|$；因为 Lagrange 定理保证，子群链长度每增加一，群的大小至少要翻倍．这就说明，单层结构至多只会更新 $\log_2|G|$ 次，因而最后得到的强生成集 $\bar{S}$ 的大小就是 $O(|B| \log |G|)$ 的．

这个估计还可以进一步改进．因为此处出现的群 G 已知是 n 次对称群 S_n 的子群，所以 G 的子群链长度不会超过 S_n 的子群链长度．可以证明[32]，S_n 的严格递增子群链长度不会超过 $3n/2$．这说明，单层结构更新的次数其实是 $O(n)$ 的．显然，基的大小也不超过 n．故而，最后得到的强生成集 $\bar{S}$ 的大小就是 $O(n^2)$ 的．

此处提到的筛选方法是 Sims 提出的，也称为 Sims 筛（Sims filter）．还有一种更为复杂的筛选方法，是由 Jerrum 提出的，也称为 Jerrum 筛[4]（Jerrum filter），它能够保证得到的强生成集的大小是 $O(n)$ 的．有兴趣的读者可以自行学习．

对于筛选过程，有一个小优化是，在 $\mathrm{MembershipTest}(C, h)$ 的实现中，并不输出布尔值，而是输出最后得到的「筛渣」[33]h（即用 **return** h 代替伪代码中的第 8 和第 12 行）．如果「筛渣」$h \neq e$，就说明成员判定失败，此时可以直接将「筛渣」h 而不是原来的 h 添加到当前层．此处的「筛渣」h 已经除去了若干个陪集代表元的因子，因而移动了更少的元素，所以会减少局部的计算量．这个优化对于整体的复杂度没有任何影响．

过程

现在可以描述 Schreier–Sims 算法的具体过程：首先，初始化一个空结构 C，用于存储群的稳定化子链．然后，逐个向结构 C 中添加生成集 S 中的生成元，最后得到的结构 C 就是群 $\langle S \rangle$ 的稳定化子链．伪代码如下：

[32]参见 Cameron, P. J., Solomon, R., & Turull, A. (1989). Chains of subgroups in symmetric groups. Journal of algebra, 127(2), 340-352.[3]

[33]这并不是什么严格的术语，在不同的英文文献中可能称作 siftee 或者 sifted element.

Algorithm SchreierSims(S) :
Input. A generating set S for a group G.
Output. The stabilizer chain C for the group G.
Method.
1 $C \leftarrow []$
2 **for** $s \in S$
3 $\quad C \leftarrow \mathrm{Extend}(C, s)$
4 **end for**
5 **return** C

算法的核心在于向当前的 C 中添加新的生成元 s 这一步，即子程序 $\mathrm{Extend}(C, s)$．正如前文所述，添加置换 s 之前和之后，都需要保证 C 是稳定化子链．这样，在添加置换 s 之前，可以首先做筛选．如果发现 s 不在已有的群中，就 **增量地** 计算轨道、陪集代表元和 Schreier 生成元．此处的「增量」的含义是，已经计算过的，不要重复计算．这样才能保证正确的复杂度．

考虑如何将 $\mathrm{OrbitTransversalStabilizer}(S, \beta)$ 改造为增量版本．算法搜索的状态空间是 $\Delta \times S$．在添加新的生成元 s 之后，状态空间将变成 $\Delta' \times (S \cup \{s\})$．两者的差集就是

$$(\Delta \times \{s\}) \cup ((\Delta' \setminus \Delta) \times (S \cup \{s\})).$$

这意味着，当加入新的生成元 s 的时候，首先需要计算新的生成元 s 与旧的轨道和相应的陪集代表元的组合；如果在这个过程中还得到了新的轨道的元素，就再考虑这些元素与所有生成元（无论新旧）的组合；过程重复到轨道不再延长为止．

向结构 C 中添加置换 g 的伪代码如下：

Algorithm Extend(C, g) :
Input. A stabilizer chain C for the group generated by S and a permutation g.
Output. A stabilizer chain C for the group generated by $S \cup \{g\}$.
Method.

```
1  if MembershipTest(C, g) is passed then
2      return C
3  end if
4  if C is empty then
5      β ← an element moved by g
6      C.orbit[0] ← β
7      C.transversal[β] ← e
8  end if
9  append g to C.generators
10 Δ ← a copy of C.orbit
11 for δ ∈ Δ
12     γ ← δ^g
13     if γ ∉ C.orbit then
14         append γ to C.orbit
15         C.transversal[γ] ← C.transversal[δ] · g
16     else
17         s' ← C.transversal[δ] · g · C.transversal[γ]^{-1}
18         C.next ← Extend(C.next, s')
19     end if
20 end for
21 for δ ∈ C.orbit \ Δ
22     for s ∈ C.generators
23         γ ← δ^s
24         if γ ∉ C.orbit then
25             append γ to C.orbit
26             C.transversal[γ] ← C.transversal[δ] · s
27         else
28             s' ← C.transversal[δ] · s · C.transversal[γ]^{-1}
29             C.next ← Extend(C.next, s')
30         end if
31     end for
32 end for
33 return C
```

这样就得到了完整的 Schreier–Sims 算法.

另一种实现

上述的实现已经是正确的，但是 12 ~ 19 行和 23 ~ 30 行略显重复．基于此，Knuth 在论文中提出了一种递归实现，更为简明．他的做法是，将这个重复的部分视作是对陪集代表系（和轨道）的更新．每次更新陪集代表系都要和所有的生成元组合，根据是否产生了新的陪集代表元，决定是递归地调用自身还是添加生成元的程序．伪代码如下：

Algorithm Extend(C, g) :
Input. A stabilizer chain C for the group generated by S and a permutation g.
Output. A stabilizer chain C for the group generated by $S \cup \{g\}$.
Method.

```
1  if MembershipTest(C, g) is passed then
2      return C
3  end if
4  if C is empty then
5      β ← an element moved by g
6      C.orbit[0] ← β
7      C.transversal[β] ← e
8  end if
9  append g to C.generators
10 for t ∈ a copy of C.transversal
11     ExtendTransversal(C, t · g)
12 end for
13 return C
```

Sub-Algorithm ExtendTransversal(C, t) :
Method.

```
1  β ← C.orbit[0]
2  γ ← β^t
3  if γ ∉ C.orbit then
4      append γ to C.orbit
5      C.transversal[γ] ← t
6      for s ∈ C.generators
7          ExtendTransversal(C, t · s)
8      end for
9  else
10     s' ← t · C.transversal[γ]^{-1}
11     C.next ← Extend(C.next, s')
12 end if
```

将此处的伪代码和上节的相比，就可以知道它是正确的．而且，两者复杂度并无差异．注意，Extend 的第 10 行遍历的是进入循环时的陪集代表系：循环中新产生的陪集代表元已经在 ExtendTransversal 的递归中和所有生成元组合过，不应重复遍历．

复杂度

为了分析 Schreier–Sims 算法的复杂度，需要一些记号．设置换的长度为 n，生成集的大小 $|\ S\ |$ 为 m．得到的（无冗余）基的长度记为 $|\ B\ |$．而且，最后得到的自外向内第 i 层的稳定化子 $G^{(i-1)}$ 中，生成元的数目记为 $|\ S^{(i-1)}|$，陪集代表系的大小（或轨道的长度）记为 $|\ T_i|$．下面分析利用上述 Schreier–Sims 算法的增量实现所需要的时间复杂度．算法主要分为两部分：筛选，以及对轨道、陪集代表系和 Schreier 生成元的计算．

最初输入的生成元和算法中得到的 Schreier 生成元都需要进行筛选，因而筛选过程执行的总次数是 $O(\sum_{i=1}^{|\ B\ |}|\ S^{(i-1)}|\ |\ T_i| + |\ S\ |)$．单次筛选需要与 $O(|\ B\ |)$ 个陪集代表元计算置换乘积．设计算与单个陪集代表元的乘积的时间是 τ．直接存储陪集代表元时，$\tau \in O(n)$；而用 Schreier 树存储陪集代表元时，$\tau \in O(n^2)$．执行筛选过程的时间复杂度总共为 $O(\tau\ |\ B\ |\sum_{i=1}^{|\ B\ |}|\ S^{(i-1)}|\ |\ T_i| + \tau\ |\ B\ |\ |\ S\ |)$．

对于轨道等信息的计算，因为是增量实现，状态空间中的每对 $(\delta, s) \in \Delta_i \times S^{(i-1)}$ 都只计算了至多一次．对于轨道和陪集代表系的计算，根据存储方式不同，单次计算陪集代表元的时间复杂度可能是 $O(n)$ 的或是 $O(1)$ 的．但是，无论如何，这都不超过计算 Schreier 生成元的复杂度．直接存储时，它是 $O(n)$ 的；使用 Schreier 树时，它是 $O(n^2)$ 的．与筛选过程的总时间复杂度比较，会发现计算这些信息的时间复杂度都不会超过筛选需要的时间复杂度．所以，Schreier–Sims 算法的时间复杂度就是上一段得到的 $O(\tau\ |\ B\ |\sum_{i=1}^{|\ B\ |}|\ S^{(i-1)}|\ |\ T_i| + \tau\ |\ B\ |\ |\ S\ |)$．

至于空间复杂度，算法最后得到的数据结构中存储了 $O(\sum_{i=1}^{|B|}|S^{(i-1)}|)$ 个生成元和 $O(\sum_{i=1}^{|B|}|T_i|)$ 个陪集代表元．如果使用直接存储，生成元和陪集代表元都需要 $O(n)$ 的空间；如果使用 Schreier 树，生成元需要 $O(n)$ 的空间，而陪集代表元只需要 $O(1)$ 的空间．

前文已经说明，n 次对称群中严格递增子群链的长度是 $O(n)$ 的，这对 $|B|$ 和 $|S^{(i)}|$ 都适用．因而，使用直接存储陪集代表系的方式实现的 Schreier–Sims 算法，时间复杂度是 $O(n^5+mn^2)$ 的，空间复杂度是 $O(n^3)$ 的．当然对于 $\log|G|\in O(n)$ 的情形，一个更好的估计是时间复杂度[34] $O(n^2\log^3|G|+mn\log|G|)$ 和空间复杂度 $O(n^2\log|G|)$．对于随机的生成集的情形，实际测试发现算法的复杂度明显低于 $\Theta(n^5)$，而大致是 $\Theta(n^4)$ 的．

虽然相较于直接存储，用 Schreier 树会在时间复杂度中引入额外的 n 因子，但对于 n 很大，但是群本身远小于 n 次对称群的情形，它的空间复杂度是 $O(n\log^2|G|)$ 的，远小于直接存储的 $O(n^2\log|G|)$．但是在算法竞赛中，很难遇到这样使用 Schreier 树存储更优的情形．

参考实现

此处提供一个 Schreier–Sims 算法的参考实现．因为 n 规模较小，实现中直接指定基 $B=\{n,n-1,\cdots,1\}$ 而不是通过算法选择它．这样做的好处是，自内向外第 k 层（不计空结构体）的群中的置换只会改变前 k 个元素，方便后续计算．代码中的另一项优化是，在存储陪集代表元的时候，存储的实际上是它的逆置换，这简化了置换的运算．

参考实现

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

// A permutation.
class Permutation {
  std::vector<int> perm;

  void shrink() {
    int m = perm.size();
    for (; m && perm[m - 1] == m - 1; --m);
    perm.resize(m);
  }

 public:
  Permutation() {}

  Permutation(const std::vector<int>& vec) : perm(vec) { shrink(); }

  int operator[](int i) const { return i < perm.size() ? perm[i] : i; }

  bool empty() const { return perm.empty(); }

  // First LHS then RHS.
  Permutation operator*(const Permutation& rhs) const {
    Permutation res;
    res.perm.resize(std::max(perm.size(), rhs.perm.size()));
    for (int i = 0; i < res.perm.size(); ++i) {
      res.perm[i] = rhs[(*this)[i]];
    }
```

[34] Knuth 的论文给出的上界还要再少一个对数因子，这需要对群的稳定化子链的基础轨道长度做更仔细的估计．

```
    res.shrink();
    return res;
  }

  // First LHS^{-1} then RHS.
  Permutation operator/(const Permutation& rhs) const {
    Permutation res;
    res.perm.resize(std::max(perm.size(), rhs.perm.size()));
    for (int i = 0; i < res.perm.size(); ++i) {
      res.perm[(*this)[i]] = rhs[i];
    }
    res.shrink();
    return res;
  }

  // Inverse.
  Permutation inv() const {
    Permutation res;
    res.perm.resize(perm.size());
    for (int i = 0; i < res.perm.size(); ++i) {
      res.perm[(*this)[i]] = i;
    }
    return res;
  }
};

// A stabilizer chain (a.k.a., BSGS) for a group.
class PermutationGroup {
  size_t n, k;
  std::vector<bool> orbit;               // Orbit of the n-th point.
  std::vector<Permutation> generators;   // Generators.
  std::vector<Permutation> transversal;  // Inverse of coset
representatives.
  PermutationGroup* next;                // Stabilizer.

  // Sift a permutation.
  void sift(Permutation& h) const {
    if (!n) return;
    int i = h[n - 1];
    if (!orbit[i]) return;
    h = h * transversal[i];
    next->sift(h);
  }

  // Add one more element into the transversal.
  void extend_transversal(Permutation t) {
    int i = t[n - 1];
    if (!orbit[i]) {
      ++k;
      orbit[i] = true;
      transversal[i] = t.inv();
      for (const auto& s : generators) {
```

```
        extend_transversal(t * s);
      }
    } else {
      next->extend(t * transversal[i]);
    }
  }

 public:
  PermutationGroup(int n)
      : n(n), k(1), orbit(n), transversal(n), next(nullptr) {
    if (!n) return;
    // Initialize the current layer.
    orbit[n - 1] = true;
    next = new PermutationGroup(n - 1);
  }

  // Destructor.
  ~PermutationGroup() {
    if (next) delete next;
  }

  // Add one more permutation into the group.
  void extend(Permutation g) {
    sift(g);
    if (g.empty()) return;
    generators.emplace_back(g);
    for (int i = 0; i < n; ++i) {
      if (orbit[i]) {
        extend_transversal(transversal[i] / g);
      }
    }
  }

  // Check whether a permutation belongs to the group.
  bool membership_test(Permutation h) const {
    sift(h);
    return h.empty();
  }

  // Return the size of the group.
  long long size() const { return n ? next->size() * k : 1LL; }
};

int main() {
  int n, m;
  std::cin >> n >> m;
  PermutationGroup group(n);
  // Read permutations and insert them to the group.
  std::vector<int> vec(n);
  for (; m; --m) {
    for (int& x : vec) {
      std::cin >> x;
```

```
      --x;  // Index starting at 0.
    }
    group.extend(Permutation(vec));
  }
  // Output the size of the group.
  std::cout << group.size();
  return 0;
}

```

习题

- LOJ 177. 生成子群阶数[5]
- [WC2017] 棋盘[6]
- Permutations[7]
- [Grand Prix of Yekaterinburg 2015] Problem H. Heimdall[8]

参考资料与注释

- Schreier–Sims algorithm - Wikipedia[9]
- Sims, Charles C, Computational methods in the study of permutation groups, Computational Problems in Abstract Algebra, pp. 169–183, Pergamon, Oxford, 1970.[10]
- Knuth, Donald E. Efficient representation of perm groups, Combinatorica 11 (1991), no. 1, 33–43.[11]
- Ákos Seress, Permutation Group Algorithms, Cambridge University Press[12]
- Alexander Hulpke's Notes on Computational Group Theory[13]
- Derek Holt's Slides on The Schreier–Sims algorithm for finite permutation groups[14]
- Martin Jaggi, Implementations of 3 Types of the Schreier–Sims Algorithm, MAS334 - Mathematics Computing Project, 2005[15]
- Henrik Bäärnhielm. The Schreier–Sims algorithm for matrix groups[16]

[1] Nielsen–Schreier 定理

[2] 自由群

[3] Cameron, P. J., Solomon, R., & Turull, A. (1989). Chains of subgroups in symmetric groups. Journal of algebra, 127(2), 340-352.

[4] Jerrum 筛

[5] LOJ 177. 生成子群阶数

[6] [WC2017] 棋盘

[7] Permutations

[8] [Grand Prix of Yekaterinburg 2015] Problem H. Heimdall

[9] Schreier–Sims algorithm - Wikipedia

[10] Sims, Charles C, Computational methods in the study of permutation groups, Computational Problems in Abstract Algebra, pp. 169–183, Pergamon, Oxford, 1970.

[11] Knuth, Donald E. Efficient representation of perm groups, Combinatorica 11 (1991), no. 1, 33–43.

[12] Ákos Seress, Permutation Group Algorithms, Cambridge University Press

[13] Alexander Hulpke's Notes on Computational Group Theory

[14] Derek Holt's Slides on The Schreier–Sims algorithm for finite permutation groups

[15] Martin Jaggi, Implementations of 3 Types of the Schreier–Sims Algorithm, MAS334 - Mathematics Computing Project, 2005

[16] Henrik Bäärnhielm. The Schreier–Sims algorithm for matrix groups

9.17 概率论

9.17.1 基本概念

概述

在研究具体的随机现象时我们通常着重关注以下要素：

- 样本空间 Ω，指明随机现象所有可能出现的结果.
- 事件域 $\mathcal{F}$，表示我们所关心的所有事件.
- 概率 P，描述每一个事件发生的可能性大小.

样本空间、随机事件

定义

一个随机现象中可能发生的不能再细分的结果被称为 **样本点**. 所有样本点的集合称为 **样本空间**，通常用 Ω 来表示.

一个 **随机事件** 是样本空间 Ω 的子集，它由若干样本点构成，用大写字母 $A, B, C, \cdots$ 表示.

对于一个随机现象的结果 ω 和一个随机事件 A，我们称事件 A **发生了** 当且仅当 $\omega \in A$.

例如，掷一次骰子得到的点数是一个随机现象，其样本空间可以表示为 $\Omega = \{1, 2, 3, 4, 5, 6\}$. 设随机事件 A 为「获得的点数大于 4」，则 $A = \{5, 6\}$. 若某次掷骰子得到的点数 $\omega = 3$，由于 $\omega \notin A$，故事件 A 没有发生.

事件的运算

由于我们将随机事件定义为了样本空间 Ω 的子集，故我们可以将集合的运算（如交、并、补等）移植到随机事件上. 记号与集合运算保持一致.

特别的，事件的并 $A \cup B$ 也可记作 $A + B$，事件的交 $A \cap B$ 也可记作 AB，此时也可分别称作 **和事件** 和 **积事件**.

事件域

研究具体的随机现象时我们需要明确哪些事件是我们感兴趣的. 根据随机事件的定义，显然有 $\mathcal{F} \subset 2^\Omega$（记号 2^Ω 表示 Ω 的幂集），但 $\mathcal{F} = 2^\Omega$ 却不是必须的. 这在样本空间 Ω 有限时可能有些难以理解，毕竟 2^Ω 尽管更大了但仍然有限. 而当 Ω 为无穷集时，2^Ω 的势变得更大，其中也难免会出现一些「性质不太好」且我们不关心的事件，这时为了兼顾这些事件而放弃一些性质就显得得不偿失了.

尽管 $\mathcal{F} = 2^\Omega$ 不是必须的，这并不代表 2^Ω 的任一子集都能成为事件域. 我们通常会对一些事件进行运算得到的结果事件的概率感兴趣，因此我们希望事件域 $\mathcal{F}$ 满足下列条件：

- $\emptyset \in \mathcal{F}$；
- 若 $A \in \mathcal{F}$，则补事件 $\bar{A} \in \mathcal{F}$；
- 若有一列事件 $A_n \in \mathcal{F}, n = 1, 2, 3...$，则 $\bigcup A_n \in \mathcal{F}$.

简言之，就是事件域 $\mathcal{F}$ 对在补运算、和可数并下是封闭的，且包含元素 $\emptyset$.

可以证明满足上述三个条件的事件域 $\mathcal{F}$ 对可数交也是封闭的.

以掷骰子为例，当样本空间记为 $\Omega = \{1, 2, 3, 4, 5, 6\}$ 时，以下两个集合能够成为事件域：

- $\mathcal{F}_1 = \{\emptyset, \Omega\}$
- $\mathcal{F}_2 = \{\emptyset, \{1, 3, 5\}, \{2, 4, 6\}, \Omega\}$

但以下两个集合则不能

- $\mathcal{F}_3 = \{\emptyset, \{1\}, \Omega\}$（对补不封闭）
- $\mathcal{F}_4 = \{\{1, 3, 5\}, \{2, 4, 6\}\}$（不含有 $\emptyset$ 且对并不封闭）

概率

定义

古典定义

在概率论早期实践中，由于涉及到的随机现象都比较简单，具体表现为样本空间 Ω 是有限集，且直观上所有样本点是等可能出现的，因此人们便总结出了下述定义：

如果一个随机现象满足：

- 只有有限个基本结果；
- 每个基本结果出现的可能性是一样的；

那么对于每个事件 A，定义它的概率为

$$P(A)=\frac{\#(A)}{\#(\Omega)}$$

其中 $\#(\cdot)$ 表示对随机事件（一个集合）大小的度量.

后来人们发现这一定义可以直接推广到 Ω 无限的一部分情景中，于是就有了所谓 几何概型[1].

公理化定义

上述基于直观认识的定义在逻辑上有一个很大的漏洞：在定义「概率」这一概念时用到了「可能性」这一说法，产生了循环定义的问题. 同时「等可能」在样本空间无限时会产生歧义，由此产生了包括 Bertrand 悖论[2] 在内的一系列问题.

经过不断探索，苏联数学家柯尔莫哥洛夫于 1933 年在他的《概率论基础》一书中第一次给出了概率的公理化定义：

概率函数 P 是一个从事件域 $\mathcal{F}$ 到闭区间 $[0,1]$ 的映射，且满足：

- **规范性**：事件 Ω 的概率值为 1，即 $P(\Omega)=1$.
- **可数可加性**：若一列事件 $A_1,A_2,\cdots$ 两两不交，则 $P\left(\bigcup_{i\geq 1}A_i\right)=\sum_{i\geq 1}P(A_i)$.

概率函数的性质

对于任意随机事件 $A,B\in\mathcal{F}$，有

- **单调性**：若 $A\subset B$，则有 $P(A)\leq P(B)$.
- **容斥原理**：$P(A+B)=P(A)+P(B)-P(AB)$.
- $P(A-B)=P(A)-P(AB)$，这里 $A-B$ 表示差集.

概率空间

我们在一开始提到，研究具体的随机现象时我们通常关注样本空间 Ω、事件域 $\mathcal{F}$ 以及概率函数 P. 我们将三元组 $(\Omega,\mathcal{F},P)$ 称为一个概率空间.

概率只有在确定的概率空间下讨论才有意义. 我们前面提到的 Bertrand 悖论归根结底就是因对样本空间 Ω 的定义不明确而产生的.

参考资料与注释

- 概率论（数学分支）_百度百科[3]
- Probability - Wikipedia[4]

外部参考资料

[1] 几何概型 

[2] Bertrand 悖论 

[3] 概率论（数学分支）_百度百科 

[4] Probability - Wikipedia 

9.17.2 条件概率与独立性

概述

当某事件已经发生时，一些随机事件的概率会因为已知信息的增加发生变化. 例如在手游抽卡时，我们可能会认为单次抽卡出六星与不出六星是等概率的，但随着我们连抽 50 发一个六星都没有，再固执地认为「出六星与不出六星等概率」就显得不是那么明智.

总之，研究在某些已知条件下事件发生的概率是必要的.

条件概率

定义

若已知事件 A 发生，在此条件下事件 B 发生的概率称为 **条件概率**，记作 $P(B \mid A)$.

在概率空间 $(\Omega, \mathcal{F}, P)$ 中，若事件 $A \in \mathcal{F}$ 满足 $P(A) > 0$，则条件概率 $P(\cdot \mid A)$ 定义为

$$P(B \mid A) = \frac{P(AB)}{P(A)} \quad \forall B \in \mathcal{F}$$

可以验证根据上式定义出的 $P(\cdot \mid A)$ 是 $(\Omega, \mathcal{F})$ 上的概率函数.

根据条件概率的定义可以直接推出下面两个等式：

- **概率乘法公式**：在概率空间 $(\Omega, \mathcal{F}, P)$ 中，若 $P(A) > 0$，则对任意事件 B 都有
$$P(AB) = P(A)P(B \mid A)$$
- **全概率公式**：在概率空间 $(\Omega, \mathcal{F}, P)$ 中，若一组事件 $A_1, \cdots, A_n$ 两两不交且和为 Ω，则对任意事件 B 都有
$$P(B) = \sum_{i=1}^{n} P(A_i)P(B \mid A_i)$$

Bayes 公式

一般来说，设可能导致事件 B 发生的原因为 $A_1, A_2, \cdots, A_n$，则在 $P(A_i)$ 和 $P(B \mid A_i)$ 已知时可以通过全概率公式计算事件 B 发生的概率. 但在很多情况下，我们需要根据「事件 B 发生」这一结果反推其各个原因事件的发生概率. 于是有

$$P(A_i \mid B) = \frac{P(A_iB)}{P(B)} = \frac{P(A_i)P(B \mid A_i)}{\sum_{j=1}^{n} P(A_j)P(B \mid A_j)}$$

上式即 Bayes 公式.

事件的独立性

在研究条件概率的过程中，可能会出现 $P(B \mid A) = P(B)$ 的情况. 从直观上讲就是事件 B 是否发生并不会告诉我们关于事件 A 的任何信息，即事件 B 与事件 A「无关」. 于是我们就有了下面的定义

定义

若同一概率空间中的事件 A,B 满足

$$P(AB) = P(A)P(B)$$

则称 A,B **独立**. 对于多个事件 $A_1, A_2, \cdots, A_n$, 我们称其独立, 当且仅当对任意一组事件 $\{A_{i_k} : 1 \le i_1 < i_2 < \cdots < i_k \le n\}$ 都有

$$P(A_{i_1}A_{i_2}\cdots A_{i_r}) = \prod_{k=1}^{r} P(A_{i_k})$$

多个事件的独立性

对于多个事件，一般不能从两两独立推出这些事件独立. 考虑以下反例：

有一个正四面体骰子，其中三面被分别涂成红色、绿色、蓝色，另一面则三色皆有. 现在扔一次该骰子，令事件 A,B,C 分别表示与桌面接触的一面包含红色、绿色、蓝色.

不难计算 $P(A) = P(B) = P(C) = \frac{1}{2}$，而 $P(AB) = P(BC) = P(CA) = P(ABC) = \frac{1}{4}$.

显然 A, B, C 两两独立，但由于 $P(ABC) \neq P(A)P(B)P(C)$，故 A, B, C 不独立.

9.17.3 随机变量

相关概念

随机变量

给定概率空间 $(\Omega, \mathcal{F}, P)$，定义在样本空间 Ω 上的函数 $X : \Omega \to \mathbb{R}$ 若满足：对任意 $t \in \mathbb{R}$ 都有

$$\{\omega \in \Omega : X(\omega) \leq t\} \in \mathcal{F}$$

则称 X 为 **随机变量**.

示性函数

对于样本空间 Ω 上的事件 A，定义随机变量

$$I_A(\omega) = \begin{cases} 1, \omega \in A \\ 0, \omega \notin A \end{cases}$$

称 I_A 是事件 A 的 **示性函数**.

分布函数

对于随机变量 X，称函数

$$F(x) = P(X \leq x)$$

为随机变量 X 的 **分布函数**. 记作 $X \sim F(x)$.

分布函数具有以下性质：

- **右连续性**：$F(x) = F(x + 0)$
- **单调性**：在 $\mathbb{R}$ 上单调递增（非严格）
- $F(-\infty) = 0, F(+\infty) = 1$

同时我们可以证明，满足上述要求的函数都是某个随机变量的分布函数. 因此，分布函数与随机变量之间一一对应.

随机变量的分类

随机变量按其值域（根据定义，随机变量是一个函数）是否可数分为 **离散型** 和 **连续型** 两种.

离散型随机变量

设 X 为离散型随机变量，其所有可能的取值为 $x_1, x_2, \cdots$，则我们可以用一系列形如 $P\{X = x_i\} = p_i$ 的等式来描述 X. 这就是我们在高中课本中学过的 **分布列**.

连续型随机变量

设 X 为连续型随机变量，考察 $P\{X = x\}$ 往往是无意义的（因为这一概率很可能是 0）.

> **为什么说概率「很可能」是 0**
>
> 考虑这样的随机变量 X：它以 $\frac{1}{2}$ 的概率取 0，以 $\frac{1}{2}$ 的概率服从开区间 $(0, 1)$ 上的均匀分布. 显然 X 满足连续型随机变量的定义.
>
> 对任何实数 $r \in (0, 1)$，不难得到 $P\{X = r\} = 0$，但同时有 $P\{X = 0\} = \frac{1}{2}$.

另一方面，设 $X \sim F(x)$，则

$$P(l < x \leq l + \Delta x) = F(l + \Delta x) - F(l)$$

一个自然的想法是用极限 $\lim\limits_{\Delta x \to 0^+} \frac{F(l+\Delta x) - F(l)}{\Delta x}$ 来描述 X 取值为 l 的可能性.

这个式子就是我们熟知的导数，于是问题转化为寻找一个非负函数 $f(x)$ 使得

$$F(x) = \int_{-\infty}^{x} f(x) \mathrm{d}x$$

若这样的 $f(x)$ 存在，则称之为 X 的 **密度函数**.

随机变量的独立性

前面讨论了随机事件的独立性. 由于随机变量和随机事件紧密联系, 我们还可以类似地给出随机变量独立性的定义.

定义

若随机变量 X,Y 满足对任意的 $x,y\in\mathbb{R}$ 都有

$$P(X\leq x,Y\leq y)=P(X\leq x)P(Y\leq y)$$

则称随机变量 X,Y **独立**.

> **Note**
>
> 有些同学也许会注意到，中学课本中对随机变量独立性的定义是用形如 $P(X=\alpha)$ 的概率定义的，但由于连续性随机变量取特定值的概率通常是 0，故在更一般的情形下借助分布函数定义才是更加明智的选择.

性质

若随机变量 X,Y 相互独立，则对于任意函数 f,g，随机变量 $f(X)$ 与 $g(Y)$ 相互独立.

> **注意**
>
> 有时候我们会研究相互独立的随机变量 X,Y 的某一函数 $f(X,Y)$（如 XY^2）的分布.
>
> 尽管 X 与 Y 是独立的，但不能想当然地认为对 Y 的某一取值 y，$f(X,y)$ 与 $f(X,Y)$ 服从同样的分布.

9.17.4 随机变量的数字特征

本文将介绍随机变量的期望、方差等数字特征.

期望

定义

离散型随机变量

设离散型随机变量 X 的概率分布为 $p_i=P\{X=x_i\}$，若和式

$$\sum x_ip_i$$

绝对收敛，则称其值为 X 的 **期望**，记作 EX.

连续型随机变量

设连续型随机变量 X 的密度函数为 $f(x)$. 若积分

$$\int_{\mathbb{R}}xf(x)\mathrm{d}x$$

绝对收敛，则称其值为 X 的 **期望**，记作 EX.

统一定义

设随机变量 X 的分布函数为 $F(x)$，若 Stieltjes 积分[1]

$$\int_{\mathbb{R}}x\mathrm{d}F(x)$$

绝对收敛，则称其值为 X 的 **期望**，记作 EX.

期望不存在的例子

考虑有如下分布的离散型随机变量 X

$$P\left\{X=(-1)^k\frac{2^k}{k}\right\}=\frac{1}{2^k},\quad k=1,2,\cdots$$

尽管和式 $\sum x_ip_i$ 收敛于 $-\ln 2$，但由于其不是绝对收敛的，故 X 的期望不存在.

再考虑有如下密度函数的连续型随机变量 Y

$$f(y)=\frac{1}{\pi}\cdot\frac{1}{1+y^2},\quad y\in(-\infty,+\infty)$$

容易验证 Y 的期望也不存在.

期望的性质

线性性

若随机变量 X,Y 的期望存在，则

- 对任意实数 a,b，有 $E(aX+b)=a\cdot EX+b$.
- $E(X+Y)=EX+EY$.

随机变量乘积的期望

若随机变量 X,Y 的期望存在且 X,Y 相互独立，则有

$$E(XY)=EX\cdot EY$$

注意：上述性质中的独立性 **并非** 必要条件.

反例

考察随机变量 X 和 Y，其中 X 服从 $[-1,1]$ 上的均匀分布，$Y=X^2$.

期望与概率的转化

对于随机事件 A，考虑其示性函数 I_A：

$$I_A(\omega)=\begin{cases}1,\omega\in A\\0,\omega\notin A\end{cases}$$

根据定义可以求得其期望 $EI_A=P(A)$. 这一转化在实际应用中非常常见.

例子

假设对于一个长为 n 的序列 $\{a_i\}$，其中 a_k 以 p_k 的概率取 k，以 $1-p_k$ 的概率取 0. 考虑如何求 $S=\sum_{i=1}^n a_i$ 的期望.

如果使用定义直接求，需要求出 S 在每个可能取值处的概率，这个计算过程比较繁琐，这里不展开叙述.

另一方面，用 I_k 表示随机事件 $a_k=k$ 的示性函数，则有

$$S=\sum_{k=1}^n k\cdot I_k$$

进而不难求出

$$ES=E\left(\sum_{k=1}^n k\cdot I_k\right)=\sum_{k=1}^n k\cdot E[I_k]=\sum_{k=1}^n k\cdot p_k$$

条件分布与条件期望

我们之前研究过条件概率，类似的也可以提出所谓条件期望的概念.

定义

对于两个随机变量 X,Y，在已知 $Y=y$ 的条件下 X 的概率分布（密度函数）称之为 **条件概率分布（条件概率密度）**，分别记作

$$P(X=x_i \mid Y=y) \qquad f_{X\mid Y}(x\mid y)$$

在此条件下，X 的期望称为 **条件期望**，记作 $E[X\mid Y=y]$.

条件期望的性质

条件期望的诸多性质可由条件概率推知，在此不做赘述.

值得一提的是 $E[X\mid Y]$ 一般是随机变量 Y 的函数，且这个函数通常不是线性的. 但实际上有

$$E[E[X\mid Y]]=EX$$

上式称作 **全期望公式**.

应用

HDU 5984 Pocky[2]

有一根长为 L 的 Pocky，每次随机折成两段. 若右边一段的长度不大于 d 则停止，否则对右边一段重复上述过程. 求重复次数的期望.

题解

记 $f(x)$ 表示长度为 x 的期望次数. $x\le d$ 的情形平凡.

当 $x>d$ 时，不妨设折断的位置距右端的长度为 k，则显然 $k\sim U[0,x]$，此时期望的重复次数为

$$g(k)=\begin{cases}1, & k\le d\\ 1+f(k), & k>d\end{cases}$$

由全期望公式可知

$$f(x)=Eg(k)=1+\frac{1}{x}\cdot\int_d^x f(t)\mathrm{d}t$$

解上述积分方程并代入初值条件得

$$f(x)=1+\ln\frac{x}{d}$$

方差

定义

设随机变量 X 的期望 EX 存在且期望

$$E(X-EX)^2$$

也存在，则称上式的值为随机变量 X 的 **方差**，记作 DX 或 $Var(x)$. 方差的算术平方根称为 **标准差**，记作 $\sigma(X)=\sqrt{DX}$.

方差的性质

若随机变量 X 的方差存在，则

- 对任意常数 a,b 都有 $D(aX+b)=a^2\cdot DX$
- $DX=E(X^2)-(EX)^2$

协方差与相关系数

一般来说，等式 $D(X+Y)=DX+DY$ 并不成立，我们自然会提出两个问题：

- $D(X+Y)$ 与 $DX+DY$ 之间相差的部分到底是什么.
- $D(X+Y)$ 与 $DX+DY$ 在什么情况下相等.

对于第一个问题，我们引入协方差作为解答.

协方差的定义

对于随机变量 X,Y，称

$$E((X-EX)(Y-EY))$$

为 X 与 Y 的 **协方差**，记作 $\operatorname{Cov}(X,Y)$.

协方差的性质

对于随机变量 X,Y,Z，有

- $\operatorname{Cov}(X,Y)=\operatorname{Cov}(Y,X)$
- 对任意常数 a,b，有 $\operatorname{Cov}(aX+bY,Z)=a\cdot\operatorname{Cov}(X,Z)+b\cdot\operatorname{Cov}(Y,Z)$

同时协方差与方差也有如下联系：

- $DX=\operatorname{Cov}(X,X)$
- $D(X+Y)=DX+2\operatorname{Cov}(X,Y)+DY$

> **关于协方差**
>
> 你可能会发现协方差的性质与向量内积的运算性质在形式上高度一致.
>
> 在泛函分析的视角下，对于给定的概率空间，其上的全体随机变量构成一个线性空间，而协方差是这个空间上的一个内积，标准差则是由该内积导出的范数.

对于刚才提出的第二个问题，不难看出 $D(X+Y)=DX+DY$ 当且仅当 $\operatorname{Cov}(X,Y)=0$. 一个直观的必要条件是 X 与 Y 独立，因为此时有

$$\operatorname{Cov}(X,Y)=E((X-EX)(Y-EY))=E(X-EX)E(Y-EY)=0$$

但这个条件并不是充分的. 为了描述满足 $\operatorname{Cov}(X,Y)=0$ 的随机变量 X,Y 之间的关系，我们引入相关系数

相关系数

对于随机变量 X,Y，称

$$\frac{\operatorname{Cov}(X,Y)}{\sigma(X)\sigma(Y)}$$

为 X 与 Y 的 **Pearson 相关系数**，记作 $\rho_{X,Y}$.

Pearson 相关系数描述了两个随机变量之间线性关联的紧密程度. $|\rho_{X,Y}|$ 越大，则 X 与 Y 之间的线性关联程度越强. 不难证明 $|\rho_{X,Y}|\le 1$，且 $|\rho_{X,Y}|=1$ 仅可能出现在以下两种情况

- 当存在实数 a 和正实数 b 使得 $P(X=a+bY)=1$ 时，有 $\rho_{X,Y}=1$;
- 当存在实数 a 和负实数 b 使得 $P(X=a+bY)=1$ 时，有 $\rho_{X,Y}=-1$.

当 $\rho_{X,Y}=0$ 时我们称随机变量 X 与 Y **不相关**，此时 X 和 Y 之间不存在线性关系.

> **「不相关」与「独立」**
>
> 两随机变量不相关只是表明他们之间没有线性关联，并不代表没有其他形式的联系.
>
> 因此两随机变量 X,Y 不相关是他们相互独立的 **必要而不充分** 条件.

对于这一小节开头提到的第二个问题，我们给出结论：$\operatorname{Cov}(X,Y)=0$ 的充要条件就是 X,Y 中的某一个以概率 1 取常值，或 X,Y 不相关.

外部参考资料

[1] Stieltjes 积分  [2] HDU 5984 Pocky 

9.17.5 概率不等式

算法竞赛中有时会用到 随机化算法→13.6.2@p3262，这些算法的正确性与时空复杂度通常依赖于「某些随机事件发生的概率很小」这一前提. 例如，快速排序的复杂度依赖于「所选的 `pivot` 元素几乎是最小或最大元素」这一事件较少发生.

本文将简要介绍一些用于分析随机化算法的工具并给出几个简单应用的例子.

Union Bound

记 $A_1, \cdots, A_m$ 为随机事件，则

$$P\left\{\bigcup_{i=1}^{m} A_i\right\} \leq \sum_{i=1}^{m} P\{A_i\}$$

即：一组事件中至少一个发生的概率，不超过每一个的发生概率之和.

实际上，这一结论还可以稍作加强：

- 一组事件中至少一者发生的概率，**不小于** 每一个的发生概率之和，减掉每两个同时发生的概率之和.
- 一组事件中至少一者发生的概率，**不超过** 每一个的发生概率之和，减掉每两个同时发生的概率之和，加上每三个同时发生的概率之和.
- ……

随着层数越来越多，交替出现的上界和下界也越来越紧. 这一系列结论形式上类似容斥原理，证明过程也和容斥类似，这里略去.

Markov 不等式

设 X 是一个取值非负的随机变量，则对任意正实数 a 有

$$P\{X \geq a\} \leq \frac{EX}{a}$$

事实上，由于 Markov 不等式本身并没有用到随机变量除期望外的与分布有关的任何信息，因此直接应用这个不等式得到的约束通常很松.

证明

记 I 为事件 $X \geq a$ 的示性函数，则有

$$I \leq \frac{X}{a}$$

进而

$$P\{X \geq a\} = EI \leq E\left[\frac{X}{a}\right] = \frac{EX}{a}$$

Chebyshev 不等式

设 X 是一随机变量，则对任意的 $a > 0$ 都有

$$P\{|\ X - EX\ | \geq a\} \leq \frac{DX}{a^2}$$

特别地，当 a 取 $k\sigma$ 时有

$$P\{|\ X - EX\ | \geq k\sigma\} \leq \frac{1}{k^2}$$

其中 σ 是 X 的标准差.

证明

由已知，有

$$P\{|\ X-EX\ |\geq a\}=P\{(X-EX)^2\geq a^2\}$$

注意到 $(X-EX)^2$ 非负，故由 Markov 不等式可知

$$P\{(X-EX)^2\geq a^2\}\leq\frac{E(X-EX)^2}{a^2}=\frac{DX}{a^2}$$

Chernoff 不等式

一般的 Chernoff 不等式可以从直接对随机变量 e^{tX} 应用 Markov 不等式得出：

设 X 是一随机变量，则对任意的 $t>0$ 都有

$$P\{X\geq a\}=P\{\mathrm{e}^{tX}>\mathrm{e}^{ta}\}\leq\frac{E\mathrm{e}^{tX}}{\mathrm{e}^{ta}}$$

类似地，当 $t<0$ 时有

$$P\{X\leq a\}=P\{\mathrm{e}^{tX}>\mathrm{e}^{ta}\}\leq\frac{E\mathrm{e}^{tX}}{\mathrm{e}^{ta}}$$

Poisson 试验之和的 Chernoff 不等式

算法竞赛中涉及的随机变量通常没有那么「一般」，我们可以用概率论中的 Poisson 试验对其进行描述.

所谓 Poisson 试验，是指在只有两种可能结果的随机试验.

一次的 Poisson 试验的结果可以用一个取值为 0 或 1 的随机变量 X 进行刻画，其概率分布为

$$P\{X=i\}=\begin{cases}p_i, & i=1\\1-p_1, & i=0\end{cases}$$

对于 Poisson 试验，我们有如下结论：

对于 n 个独立的 Poisson 试验 $X_1,X_2,\cdots,X_n$，记 $X=\sum_{i=1}^n X_i$ 以及 $\mu=EX$，则对任意 $0<\epsilon<1$ 有

$$P\{|\ X-\mu\ |\geq\epsilon\mu\}\leq 2\exp\left(-\frac{1}{3}\mu\epsilon^2\right)$$

Hoeffding 不等式

若 $X_1,\cdots,X_n$ 为互相独立的实随机变量且 $X_i\in[a_i,b_i]$，记随机变量 $X=\sum\limits_{i=1}^n X_i$，则

$$P\{|\ X-EX\ |\geq\epsilon\}\leq 2\exp\left(\frac{-2\epsilon^2}{\sum\limits_{i=1}^n(b_i-a_i)^2}\right)$$

Chernoff 不等式和 Hoeffding 不等式都限制了随机变量偏离其期望值的程度. 这两个不等式的证明过程较为冗长，有兴趣的同学可以查阅 Probability and Computing 一书中的相关章节.

从经验上讲，如果 EX 不太接近 $a_1+\cdots+a_n$，则该不等式给出的界往往相对比较紧；如果非常接近的话（例如在 UOJ #72 全新做法[1] 中），给出的界则往往很松，此时更好的选择是使用 Chernoff 不等式.

应用举例

例：随机撒点估算圆周率

考虑下列估计圆周率 π 的精确值的算法：

在正方形区域 $[-1,1]^2$ 内随机生成 n 个点，记其中落入单位圆盘 $x^2+y^2\leq 1$ 的点数为 m，则可以取 $\dfrac{4m}{n}$ 为 π 的近似值.

问题：若要保证上述算法以至少 $(1-\delta)$ 的概率返回相对误差不超过 ϵ 的结果，n 应该如何取定？

解答

记 X_i 表示事件「随机生成的第 i 个点在单位圆内」，则圆内总点数 $X=\sum_{i=1}^{n}X_i$．我们需要找到一个合适的 n 使得

$$P\left\{\left|\frac{4X}{n}-\pi\right|\geq\epsilon\pi\right\}\leq\delta$$

上式等价于

$$P\left\{\left|X-\frac{\pi}{4}n\right|\geq\epsilon\cdot\frac{\pi}{4}n\right\}\leq\delta$$

根据 Chernoff 不等式，我们只需令

$$2\exp\left(-\frac{1}{3}\epsilon^2\cdot\frac{\pi}{4}n\right)\leq\delta$$

即可，由此可解得

$$n\geq\frac{12}{\pi}\epsilon^{-2}\ln\frac{2}{\delta}$$

即当 $n=\Omega(\epsilon^{-2}\ln\frac{1}{\delta})$ 时可以达到需要的准确率．

例：抽奖问题

一个箱子里有 n 个球，其中恰有 k 个球对应着大奖．你要进行若干次独立、等概率的随机抽取，每次抽完之后会把球放回箱子．请问抽多少次能保证以至少 $(1-\epsilon)$ 的概率，满足 **每一个** 奖球都被抽到至少一次？

解答

假如只有一个奖球，则抽取 $M=n\log\epsilon^{-1}$ 次即可保证，因为 M 次全不中的概率

$$\left(1-\frac{1}{n}\right)^{n\log\epsilon^{-1}}\leq e^{\log\epsilon}=\epsilon$$

现在有 $k>1$ 个奖球，那么根据 Union Bound，我们只需保证每个奖球被漏掉的概率都不超过 $\frac{\epsilon}{k}$ 即可．于是答案是 $n\log\frac{k}{\epsilon}$．

例：随机选取一半元素

给出一个算法，从 n 个元素中等概率随机选取一个大小为 $\frac{n}{2}$ 的子集，保证 n 是偶数．你能使用的唯一的随机源是一枚均匀硬币，同时请你尽量减少抛硬币的次数（不要求最少）．

解法

首先可以想到这样的算法：

- 通过抛 n 次硬币，可以从所有子集中等概率随机选一个．
- 不断重复这一过程，直到选出的子集大小恰好为 $\frac{n}{2}$．
  - 注意到大小为 $\frac{n}{2}$ 的子集至少占所有子集的 $\frac{1}{n}$，因此重复次数的期望值 $\leq n$．

这一算法期望需要抛 n^2 次硬币．

另一个算法：

- 我们可以通过抛期望 $2\lceil\log_2 n\rceil$ 次硬币来实现随机 n 选 1．
  - 具体方法：随机生成 $\lceil\log_2 n\rceil$ 位的二进制数，如果大于等于 n 则重新随机，否则选择对应编号（编号从 0 开始）的元素并结束过程．

- 然后我们从所有元素中选一个，再从剩下的元素中再选一个，以此类推，直到选出 $\frac{n}{2}$ 个元素为止.

这一算法期望需要抛 $n\lceil\log_2 n\rceil$ 次硬币.

将两个算法缝合起来：

- 先用第一个算法随机得到一个子集.
- 如果该子集大小不到 $\frac{n}{2}$，则利用第二个算法不断添加元素，直到将大小补到 $\frac{n}{2}$.
- 如果该子集大小超过 $\frac{n}{2}$，则利用第二个算法不断删除元素，直到将大小削到 $\frac{n}{2}$.

尝试分析第二、第三步所需的操作次数（即添加/删除元素的次数）：

- 记 01 随机变量 X_i 表示 i 是否被选入初始的子集，令 $X := X_1 + \cdots + X_n$ 表示子集大小，则第二、第三步所需的操作次数等于 $|X - \mathrm{E}[X]|$. 在 Hoeffding 不等式中取 $t = c \cdot \sqrt{n}$（其中 c 为任意常数），得到 $\Pr\left[|X - \mathrm{E}[X]| \geq t\right] \leq 2\mathrm{e}^{-c^2}$. 也就是说，我们可以通过允许 $\Theta(\sqrt{n})$ 级别的偏移，来得到任意小的常数级别的失败概率.

至此我们已经说明：该算法可以以很大概率保证抛硬币次数在 $n + \Theta(\sqrt{n}\log n)$ 以内.

- 其中 n 来自获得初始子集的抛硬币次数；$\Theta(\sqrt{n}\log n)$ 是 $\Theta(\sqrt{n})$ 次添加/删除元素的总开销.

计算期望复杂度

我们再从另一个角度分析，尝试计算该算法的期望抛硬币次数.

用 Hoeffding 不等式求第二、第三步中操作次数期望值的上界：

$$E\mid X - EX\mid = \int_0^\infty P\{|\ X - E[X]| \geq t\}\mathrm{d}t \leq 2\int_0^\infty \exp\left(-\frac{t^2}{n}\right)\mathrm{d}t = \sqrt{\pi n}$$

从而第二、第三步所需抛硬币次数的期望值是 $\sqrt{\pi n}\cdot 2\lceil\log_2 n\rceil$.

综上，该算法期望需要抛 $n + 2\sqrt{\pi n}\lceil\log_2 n\rceil$ 次硬币.

练习：Balls and Bins

n 个球独立随机地扔到 n 个盒子里，试证明：球最多的盒子中的球数以 $1 - \frac{1}{n}$ 的概率不少于 $\Omega\left(\frac{\log n}{\log\log n}\right)$.

外部参考资料

[1] UOJ #72 全新做法



9.18 博弈论

9.18.1 博弈论简介

Authors: cutekibry, woruo27, Backl1ght, c-forrest

博弈论（game theory）是经济学的一个分支，主要研究具有竞争或对抗性质的个体，在特定规则下所产生的各种行为. 博弈论关注博弈中个体的预期行为与实际行为，并研究其最优策略.

通俗地讲，博弈论主要研究的是：在一个游戏中，进行游戏的多位玩家如何选择策略.

基础概念

本节将简要介绍博弈论中的一些常见概念.

合作/非合作博弈

合作博弈（cooperative game）是指参与者可以结成联盟、相互合作的博弈．在这类博弈中，个体的不合作行为往往会受到某种外部机制的惩罚．与之相对，**非合作博弈**（noncooperative game）中并不存在这样的机制，因此，参与者要么无法结成联盟，要么只能依赖可信的威胁机制维持合作．

相比合作博弈，非合作博弈的研究更为系统和成熟．本文讨论的所有博弈均为非合作博弈．

对称/非对称博弈

在**对称博弈**（symmetric game）中，不同参与者在做出相同行为时获得的收益是相同的，也就是说，收益只取决于行为本身，而与行为者的身份无关．不满足这一条件的博弈称为**非对称博弈**（asymmetric game）．

零和/非零和博弈

主页面：零和博弈→9.18.3@p1924

零和博弈（zero-sum game）指的是无论各方采取何种行为，所有参与者的收益总和始终为零．通常讨论的零和博弈涉及两名参与者，此时，一方的收益必然是另一方的损失．相对地，**非零和博弈**（non-zero-sum game）允许多方共赢或共输，包括**正和博弈**（positive-sum game）和**负和博弈**（negative-sum game）等．

同时/序贯博弈

在**同时博弈**（simultaneous game）中，所有参与者在不知道他人选择的前提下同时做出决策．例如剪刀石头布就是一个典型的同时博弈．这类博弈常用收益矩阵表示，并通常不涉及时间的概念．

与此相对的是**序贯博弈**（sequential game），即参与者依次行动．需要注意的是，后行动者至少能够观察到部分先行动者的行为，否则先后顺序将毫无意义．序贯博弈通常借助博弈树来刻画．

完美/不完美信息博弈

完美信息（perfect information）指参与者在任意时刻做出决策时，完全了解此前所有事件的发生情况，包括游戏初始状态．例如象棋、围棋等属于完美信息博弈；而麻将、扑克则是不完美信息博弈，因为玩家无法获知他人的手牌．完美信息通常用于描述序贯博弈；由于在同时博弈中玩家彼此无法得知对方即将采取的行动，因此通常认为同时博弈不是完美信息博弈．

完全/不完全信息博弈

完全信息（complete information）是指所有参与者对博弈结构本身（包括各方可选决策和最终收益）有完全了解，且这些信息为公共知识（common knowledge）．与之相对的是不完全信息博弈，其中某些博弈要素（如对手的可选决策或收益函数）对参与者来说是未知的．

值得注意的是，「完全信息」和「完美信息」是两个独立概念，互不包含．例如，麻将是一种完全信息但不完美信息的博弈，因其规则和收益是公开的，但牌面信息并不透明；而某些具有隐藏目标、但行为全程公开的游戏，则属于完美信息但不完全信息的博弈．

组合博弈论

在算法竞赛中，最常见的博弈类型是**组合博弈**（combinatorial game）．该术语通常指那些因状态数量巨大而难以求解的博弈．正因为一般的组合博弈相当复杂，组合博弈论主要关注以下类型：两人轮流行动的、完美信息、无随机因素的博弈．象棋、围棋等都是典型的组合博弈．

公平组合博弈

主页面：公平组合博弈→9.18.2@p1903

公平博弈（impartial game）指满足如下条件的组合博弈：

- 在任意确定状态下，所有参与者可选择的行动完全相同，仅取决于当前状态，与身份无关；
- 博弈中的同一个状态不可能多次抵达，博弈以参与者无法行动为结束，且博弈一定会在有限步后以非平局结束．

公平博弈总是对称博弈.

非公平组合博弈

主页面：非公平组合博弈→9.18.4@p1930

与公平博弈相对的概念是 **非公平博弈**（partizan game），即参与者在某一状态下可采取的行动依赖其身份．大多数棋类游戏（如国际象棋、中国象棋、围棋、五子棋等）都是非公平博弈，因为参与者只能操作自己的棋子.

正常/反常博弈

组合博弈中，通常的胜利者是博弈结束前，最后一名采取行动的参与者．这称为 **正常博弈**（normal game）．与之相对应的是 **反常博弈**（misère game），即博弈结束前，最后一名采取行动的参与者是失败者.

公平和非公平组合博弈都可以是正常或反常博弈.

参考资料

- Game theory - Wikipedia[1]
- Combinatorial game theory - Wikipedia[2]
- Impartial game - Wikipedia[3]
- Misère - Wikipedia[4]

外部参考资料

[1] Game theory - Wikipedia

[2] Combinatorial game theory - Wikipedia

[3] Impartial game - Wikipedia

[4] Misère - Wikipedia

9.18.2 公平组合游戏

> **Authors:** cutekibry, woruo27, tinjyu, 2008verser, Backl1ght, billchenchina, Enter-tainer, FFjet, Ir1d, Molmin, orzAtalod, ouuan, SaMiiKaaaa, SamZhangQingChuan, Tiphereth-A, chu-yuehan

前置知识：博弈论简介→9.18.1@p1901

本文讨论 公平组合游戏→9.18.1@p1901.

公平组合游戏中，最基础也最重要的是正常 Nim 游戏．Sprague–Grundy 定理指出，所有正常规则的公平组合游戏都等价于一个单堆 Nim 游戏．由此，可以发展出 Sprague–Grundy 函数和 Nim 数的概念，它们完全地刻画了一个正常规则的公平组合游戏．因此，本文首先建立了正常 Nim 游戏的结论和 Sprague–Grundy 理论．随后，本文讨论了算法竞赛中常见的一些公平组合游戏.

最后，本文简单地讨论了反常 Nim 游戏．反常游戏相对于正常游戏来说要复杂得多，也很少在算法竞赛中出现．本文提到的游戏，如果没有特别说明，均默认为正常的公平组合游戏.

「状态」、「局面」与「游戏」

本文会交替地使用这三个词语．在博弈论中，游戏的状态（state）通常包括到游戏的某一时刻为止，所有可能与游戏有关的信息．在一般的情形下，游戏的状态通常包括双方玩家过往的行动、已经实现的随机变量值、双方已知信息的内容等．游戏的局面（position）相对来说并非博弈论的标准术语，通常指在游戏的某一时刻，双方玩家面对的局势，例如棋类游戏中各棋子的位置等．仅对于公平组合游戏（或更一般的零和、确定、完美信息游戏）而言，由于游戏不涉及随机性，且玩家未来的行动集合与收益函数均与到达当前局面的历史路径（即之前双方的行为）无关，所以，游戏的状态（state）

和局面（position）没有区别，且都可以看作博弈图上的一个结点（node）. 由于一个游戏（game）总是可以由它的初始局面描述，所以有时也会直接使用「局面」一词代指游戏本身.

Nim 游戏

Nim 游戏的规则很简单：

自 Nim 游戏

共有 n 堆石子，第 i 堆有 a_i 枚石子. 两名玩家轮流取走任意一堆中的任意多枚石子，但不能不取. 取走最后一枚石子的玩家获胜.

容易验证，Nim 游戏是正常规则的公平组合游戏.

例子

举个例子. 当前，有 3 堆石子，石子的数量分别为 $2, 5, 4$. 那么，可以取走第 1 堆中的 2 枚石子，局面就变成了 $0, 5, 4$；也可以取走第 2 堆的 4 枚石子，局面就变成了 $2, 1, 4$. 如果某一时刻的局面变为了 $0, 0, 5$，且轮到玩家甲行动，那么甲可以取走第 3 堆的 5 枚石子，也就是取走最后一枚石子，此时甲获胜.

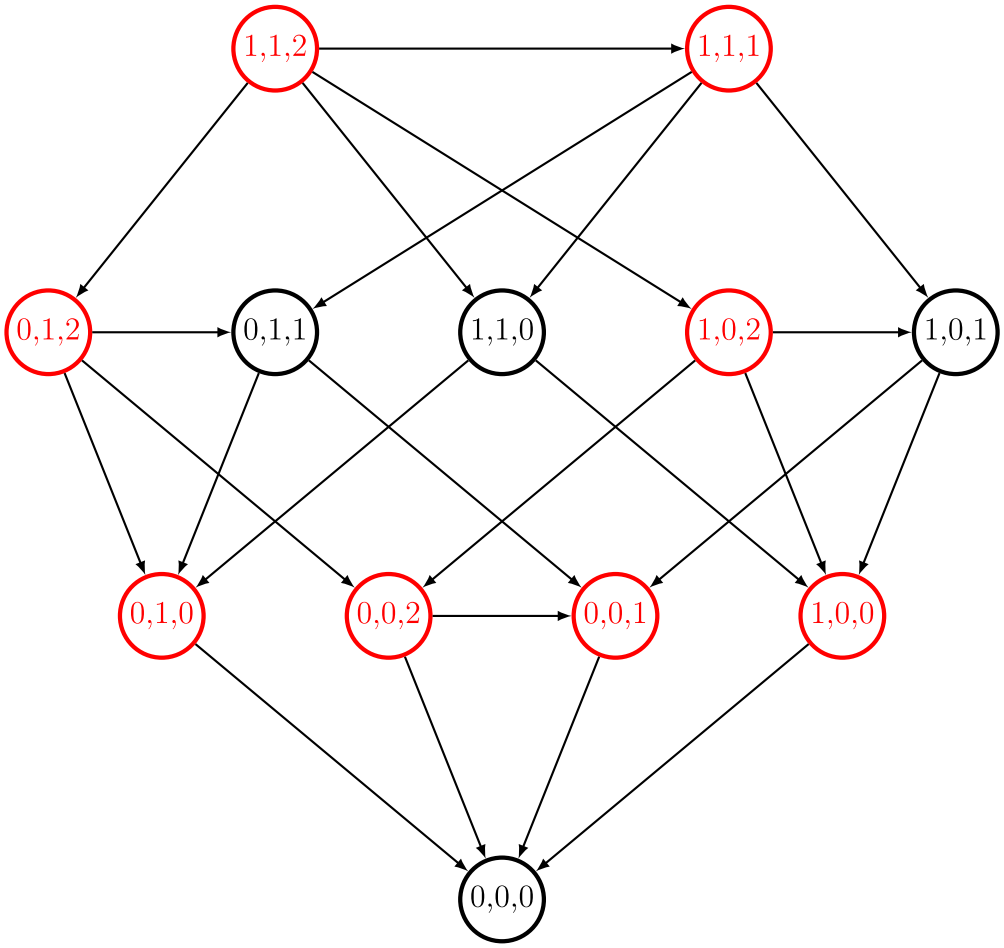
博弈图和状态

Nim 游戏中，局面可能的变化可以用博弈图来描述.

将每一个可能的状态都看作是图中的一个结点，并将状态向它的后继状态（即通过一次操作可以达到的状态）连边，就得到一个有向无环图，这就是博弈图. 图是无环的，因为 Nim 游戏中，每次操作，石子的总数量都是严格减少的.

例子

例如，对于初始局面有 3 堆石子，且每堆石子的数量分别为 $1, 1, 2$ 的 Nim 游戏，可以绘制如下的博弈图：



马上就会提到，图中的红色结点表示必胜状态，黑色结点表示必败状态.

由于 Nim 游戏是公平组合游戏，每个玩家是否有必胜策略，只取决于当前游戏所处的状态，而与玩家的身份无关. 因此，所有状态可以分为（先手）**必胜状态** 和（先手）**必败状态**，分别记为 $\mathcal{N}$ 态和 $\mathcal{P}$ 态[1]. 这个定义适用于所有公平组合游戏.

通过下述引理，可以归纳地将所有状态标记为必胜状态和必败状态：

引理

正常规则的公平组合游戏中，

1. 没有后继状态的状态是必败状态 $\mathcal{P}$，
2. 一个状态是必胜状态 $\mathcal{N}$ 当且仅当存在至少一个它的后继状态为必败状态 $\mathcal{P}$，
3. 一个状态是必败状态 $\mathcal{P}$ 当且仅当它的所有后继状态均为必胜状态 $\mathcal{N}$.

证明

对于第一条，如果玩家当前已经没有可选的行动，那么玩家已经输掉了游戏.

对于第二条，如果该状态至少有一个后继状态为必败状态，那么玩家可以操作到该必败状态；此时，对手面临了先手必败状态，玩家自己就获得了胜利.

[1]「$\mathcal{N}$ 态」和「$\mathcal{P}$ 态」这两个名称分别表示「下一名玩家胜利」（Next player wins）和「前一名玩家胜利」（Previous player wins）.

对于第三条，如果不存在一个后继状态为必败状态，那么无论如何，玩家只能操作到必胜状态；此时，对手面临了先手必胜状态，玩家自己就输掉了游戏.

所有公平组合游戏中，博弈图都是有向无环图. 所以，通过这三条性质，可以在绘制出博弈图后，在 $O(|V|+|E|)$ 的时间内，计算出每个状态是必胜状态还是必败状态. 其中，$|V|$ 为博弈图的状态数目，$|E|$ 为边数，即所有状态可以采取的行动的数量的总和.

这一引理可以推广到反常游戏和有向图可能有环的情形. 相关讨论详见 有向图游戏→9.18@p1901 一节.

Nim 和

继续考察 Nim 游戏.

通过绘制博弈图，可以求出某一局面是否是先手必胜. 但是，这样做需要 $\Omega(\prod_{i=1}^{n}(a_i+1))$ 的时间，复杂度过高，无法实际应用. 实际上，可以发现 Nim 游戏的状态是否先手必胜，只与当前局面的石子数目的 Nim 和有关.

自 Nim 和

自然数 $a_1, a_2, \cdots, a_n$ 的 **Nim 和**（Nim sum）定义为 $a_1 \oplus a_2 \oplus \cdots \oplus a_n$.

所谓 Nim 和，就是 异或运算→9.4@p1099.

定理

Nim 游戏中，状态 $(a_1, a_2, \cdots, a_n)$ 是必败状态 $\mathcal{P}$，当且仅当 Nim 和

$$a_1 \oplus a_2 \oplus \cdots \oplus a_n = 0.$$

证明

对所有可能的状态应用归纳法：

1. 如果 $a_i = 0$ 对所有 $i = 1, \cdots, n$ 都成立，该状态没有后继状态，且 Nim 和等于 0，命题成立.
2. 如果 $k = a_1 \oplus a_2 \oplus \cdots \oplus a_n \neq 0$，那么，需要证明该状态是必胜状态. 也就是说，需要构造一个合法移动，使得后继状态为必败状态；由归纳假设，只需要证明后继状态满足 $a'_1 \oplus a'_2 \oplus \cdots \oplus a'_n = 0$. 利用 Nim 和（即异或）的性质，这等价于说，存在第 i 堆石子，从中拿走若干枚石子后，数量可以变为 $a_i \oplus k$，亦即 $a_i > a_i \oplus k$.

   实际上，设 k 的二进制表示中，最高位的 1 是第 d 位. 那么，一定存在某个 a_i，使得它的二进制第 d 位是 1. 对于相应的石子堆，就一定有 $a_i > a_i \oplus k$，因为 $a_i \oplus k$ 中第 d 位为 0，更高位和 a_i 一样.
3. 如果 $a_1 \oplus a_2 \oplus \cdots \oplus a_n = 0$，那么，需要证明该状态是必败状态. 由归纳假设可知，只要证明它的所有后继状态的 Nim 和都不是 0. 这是必然的，任何合法移动将 a_i 变为 $a'_i \neq a_i$，就必然会使得 Nim 和变为 $a'_i \oplus a_i \neq 0$.

由此，可以在 $O(n)$ 时间内判断 Nim 游戏的一个状态是否为先手必胜状态.

Sprague–Grundy 理论

Sprague–Grundy 理论指出，所有正常规则的公平组合游戏都等价于单堆 Nim 游戏. 这一结论主要应用的场景，就是游戏由多个相互独立的子游戏组成的情形. 此时，游戏的状态判定可以通过计算子游戏的 SG 函数值的 Nim 和来完成. 如果游戏本身没有这样的结构，那么，判定必胜状态和必败状态只需要应用前文 博弈图→9.18@p1901 一节的引理.

游戏的记法

前文已经说明，所有公平组合游戏都可以通过绘制博弈图来描述. 由于博弈图中，每个状态的性质只由它的后继状态决定，所以，可以将博弈图中的一个状态 S 用它的后继状态的集合来表示.

例子（续）

以上文的博弈图为例，可以得到如下状态表示：

$$
\begin{aligned}
&S_{0,0,0} = \{\},\\
&S_{0,1,0} = \{S_{0,0,0}\} = \{\{\}\},\\
&S_{0,0,1} = \{S_{0,0,0}\} = \{\{\}\},\\
&S_{0,0,2} = \{S_{0,0,0}, S_{0,0,1}\} = \{\{\}, \{\{\}\}\},\\
&S_{0,1,1} = \{S_{0,1,0}, S_{0,0,1}\} = \{\{\{\}\}\},\\
&S_{0,1,2} = \{S_{0,0,2}, S_{0,1,0}, S_{0,1,1}\} = \{\{\{\}, \{\{\}\}\}, \{\{\}\}, \{\{\{\}\}\}\}.
\end{aligned}
$$

其中，$S_{0,1,0} = S_{0,0,1}$.

一个游戏可以用它的初始状态表示.

尽管公平游戏的表示可能相当复杂，单堆 Nim 游戏相对来说简单很多. 只有一堆石子，石子数量为 n 时，它可以表示为

$$*0 = \{\},\ *n = \{*m : m < n,\ m \in \mathbf{N}\} = \{*0, *1, \cdots, *(n-1)\}.$$

其中，记号 $*n$ 表示石子数量为 n 时的单堆 Nim 游戏（的初始状态）.

例子（续）

利用这一记号，上面的例子中的状态可以简单地表示为

$$S_{0,0,0} = *0,\ S_{0,1,0} = S_{0,0,1} = *1,\ S_{0,0,2} = *2,\ S_{0,1,1} = \{*1\},\ S_{0,1,2} = \{*1, *2, \{*1\}\}.$$

在随后的讨论中，记号 $T \in S$ 应当理解为状态 T 是状态 S 的后继状态.

游戏的和与等价

游戏的等价关系，依赖于游戏的和[2]的概念.

游戏的和

游戏 G 和 H 的 **和**（sum），或称 **游戏组合**（combined game），记作 $G + H$，是指游戏

$$G + H = \{g + H : g \in G\} \cup \{G + h : h \in H\}.$$

游戏的和，可以理解为由两个同时进行且互不干扰的子游戏组成的游戏，玩家在每一步能且只能选择其中一个子游戏移动一步，且游戏在两个子游戏都无法移动时结束. 游戏的和的概念，可以推广到任意多个游戏的情形，且满足结合律和交换律——也就是说，多个游戏组合的结果，和组合进行的次序以及游戏的顺序都无关. Nim 游戏就是多个单堆 Nim 游戏的和.

一个观察是，尽管单堆 Nim 游戏中，除了没有石子的情形，都是先手必胜状态，但是这些不同的单堆 Nim 游戏在和其他的单堆 Nim 游戏组合起来时，得到的游戏并不相同. 比如，游戏 $*n$ 只有在和另一个 $*n$ 组合时，才能得到一个必败游戏；和所有其他的单堆 Nim 游戏 $*n' \neq *n$ 组合，得到的游戏都是必胜游戏.

这个观察带来的启示是，可以通过考察与其他游戏的和来研究某个游戏的性质. 这就引出了游戏的等价的概念.

[2]本文讨论的「和」都是 **长规则**（long rule）下的 **析取和**（disjunctive sum）. 这也是最常见的一种游戏组合方式. 除此之外，还有其他可能的游戏组合方式. 关于它们的详细讨论，可以参考 Conway, John H. On numbers and games. AK Peters/CRC Press, 2000. 一书的第 14 章.

自 游戏的等价关系

如果对于所有游戏 H，游戏 $G_1 + H$ 和 $G_2 + H$ 都同处于必败状态或必胜状态，那么，称游戏 G_1 和 G_2 **等价**（equivalent），记作 $G_1 \approx G_2$.

容易验证，这样定义的 $\approx$ 确实是全体公平游戏上的 等价关系→9.20@p1951.

Sprague–Grundy 函数

对 Nim 游戏的分析说明，不同的单堆 Nim 游戏互不等价．但是，所有的公平游戏都等价于某个单堆 Nim 游戏．由此，可以给每个公平游戏都分配一个数字，这就是 Sprague–Grundy 函数.

为了证明这些结论，首先需要建立关于游戏等价关系的两个引理．第一，将必败游戏和任何游戏组合到一起，都和原来的游戏等价.

引理 1

对于游戏 G 和任何必败游戏 $A \in \mathcal{P}$，都有 $G \approx G + A$.

证明

按照定义，只需要证明对于任何游戏 H，游戏 $G + H$ 和 $G + A + H$ 都同处于必败状态或必胜状态.

如果游戏 $G + H$ 有必胜策略，那么，游戏 $G + A + H$ 也有必胜策略．如果对手在子游戏 A 中进行了移动，就在子游戏 A 中进行移动，将它恢复至必败状态；否则，按照游戏 $G + H$ 中的必胜策略移动．这样一定能保证最终的胜利.

如果游戏 $G + H$ 是必败游戏，那么，游戏 $G + A + H$ 也同样是必败游戏．因为无论这一回合进行的是子游戏 $G + H$ 还是子游戏 A 中的移动，对手都可以在下一回合将相应子游戏恢复至必败状态．最终，先手玩家一定无法获胜.

第二，两个游戏等价，当且仅当它们的和是必败游戏．这一引理提供了证明两个游戏等价的方法.

引理 2

游戏 G 和 G' 等价，当且仅当 $G + G' \in \mathcal{P}$ 是必败游戏.

证明

如果游戏 G 和 G' 等价，那么，$G + G'$ 与 $G + G$ 同时必胜或同时必败，而游戏 $G + G$ 是必败游戏．这是因为，对于先手玩家的任何操作，后手玩家都可以在另一个子游戏中采取相同的行动，最后一定是先手玩家无法移动.

反过来，如果 $G + G'$ 是必败游戏，那么，由引理 1 可知，$G \approx G + (G + G') = (G + G) + G' \approx G'$.

利用这些引理，可以得到如下定理：

定理（Sprague–Grundy）

对于任何一个（有限）公平游戏 G，都存在 $n \in \mathbf{N}$，使得 $G \approx {*}n$ 成立.

证明

要证明定理的结论，可以应用数学归纳法．设游戏 $G=\{G_1,G_2,\cdots,G_k\}$．根据归纳假设可知，存在 $n_1,n_2,\cdots,n_k$ 使得 $G_i\approx *n_i$，那么，可以考察游戏

$$G'=\{*n_1,*n_2,\cdots,*n_k\}.$$

将要证明的是，$G'\approx *m$，其中，$m=\operatorname{mex}\{n_1,n_2,\cdots,n_k\}$ 是没有出现在集合中的最小自然数．

第一步，需要说明 $G\approx G'$．根据引理 2，只需要证明游戏 $G+G'$ 是必败游戏．不妨假设 $G\neq *0$．如果先手玩家选择 G_i，那么后手玩家就可以选择 $*n_i$；反过来，如果先手玩家选择了 $*n_i$，后手玩家就可以选择 G_i．总之，在这两步操作后，游戏变为 G_i+*n_i，根据引理 2 和 $G_i\approx *n_i$，这是必败游戏．这就证明了 $G\approx G'$．

第二步，需要说明 $G'\approx *m$．根据引理 2，只需要证明 $G'+*m$ 是必败游戏．不妨假设 $G'\neq *0$．如果先手玩家选择了 $*n_i\in *m$，那么根据 m 的定义，后手玩家就可以选择 $*n_i\in G'$，将游戏局面变为 $*n_i+*n_i\in\mathcal{P}$，先手必败．如果先手玩家选择了 $*n_i\in G'$ 且 $n_i<m$，那么，后手玩家可以选择 $*n_i\in *m$，游戏局面同样变为 $*n_i+*n_i\in\mathcal{P}$，先手必败．最后，如果先手玩家选择了 $*n_i\in G'$ 且 $n_i>m$，那么，后手玩家可以选择 $*m\in *n_i$，游戏局面变为 $*m+*m\in\mathcal{P}$，先手必败．这就证明了 $G'\approx *m$．

由等价关系的传递性可知，$G\approx *m$．这就完成了归纳，证明所有游戏 G 都等价于一个单堆 Nim 游戏．

这一结论说明，可以为每一个公平游戏 G 都分配一个自然数 n，使得 $G\approx *n$．

自 Nim 数

一个公平游戏 G 对应的 **Nim 数**（nimber）就是使得 $G\approx *n$ 成立的唯一自然数 n．

这个将公平游戏映射到 Nim 数的函数称为 **Sprague–Grundy 函数**（Sprague–Grundy function），简称 **SG 函数**，记作 $\operatorname{SG}(\cdot)$．由于每个公平游戏的状态都是另一个公平游戏，所以，对于公平游戏的每一个状态都可以计算相应的 Nim 数，也称为相应的 SG 函数值．

根据本节定理的证明过程可知，Sprague–Grundy 函数可以递归地计算如下：

推论

公平游戏 G 中的一个状态 x 对应的 Sprague–Grundy 函数值 $\operatorname{SG}(x)$ 满足

$$\operatorname{SG}(x)=\operatorname{mex}\{\operatorname{SG}(x'):x'\in x\}.$$

其中，$\operatorname{mex}(A):=\min\{n\in\mathbf{N}:n\notin A\}$ 是没有出现在集合 A 中的最小自然数．

也就是说，一个状态的 SG 函数值，等于它的所有后继状态的 SG 函数值的 mex 值．

利用 SG 函数值（即 Nim 数），可以判断一个状态是否为先手必胜状态．

推论

公平游戏 G 中的一个状态 x 是先手必胜状态，当且仅当 $\operatorname{SG}(x)\neq 0$．

最后，游戏的和的 SG 函数值，就是子游戏的 SG 函数值的 Nim 和（即异或）．

定理（Sprague–Grundy）

对于公平游戏 $G_1,G_2,\cdots,G_n$，有

$$\operatorname{SG}(G_1+G_2+\cdots+G_n)=\operatorname{SG}(G_1)\oplus\operatorname{SG}(G_2)\oplus\cdots\oplus\operatorname{SG}(G_n).$$

证明

因为 $*a_1+*a_2+\cdots+*a_n$ 就是石子数量为 $(a_1,a_2,\cdots,a_n)$ 的 Nim 游戏，所以，根据 Nim 游戏的结论可知，游戏

$$*a_1+*a_2+\cdots+*a_n+*(a_1\oplus a_2\oplus\cdots\oplus a_n)$$

是先手必败的．根据引理 2，有

$$*a_1+*a_2+\cdots+*a_n\approx *(a_1\oplus a_2\oplus\cdots\oplus a_n).$$

所以，有

$$\mathrm{SG}(*a_1+*a_2+\cdots+*a_n)=a_1\oplus a_2\oplus\cdots\oplus a_n.$$

设 $a_i=\mathrm{SG}(G_i)$，就有 $G_i\approx *a_i$，那么，利用 $\approx$ 的代数性质，有

$$(G_1+G_2+\cdots+G_n)+(*a_1+*a_2+\cdots+*a_n)=\sum_{i=1}^{n}(G_i+*a_i)\in\mathcal{P}.$$

所以，就有

$$\begin{aligned}\mathrm{SG}(G_1+G_2+\cdots+G_n)&=\mathrm{SG}(*a_1+*a_2+\cdots+*a_n)\\&=a_1\oplus a_2\oplus\cdots\oplus a_n\\&=\mathrm{SG}(G_1)\oplus\mathrm{SG}(G_2)\oplus\cdots\oplus\mathrm{SG}(G_n).\end{aligned}$$

利用这一定理，在计算游戏的和的 SG 函数值时，可以大幅简化计算．

由此，可以总结出 SG 函数值的计算方法：

- 对于多个独立的游戏，可以分别计算它们的 SG 函数值，再求 Nim 和；
- 对于单个游戏，每个状态的 SG 函数值都是它的所有后继状态的 SG 函数值的 mex 值；
- 特别地，终止状态（即没有后继状态的状态）的 SG 函数值为 $\mathrm{mex}\,\emptyset=0$．

Nim 数

所有的公平游戏都唯一对应一个 Nim 数．（有限）Nim 数的集合就是自然数集 $\mathbf{N}$．但是，它的代数性质和自然数集不同．具体来说，Nim 数上可以定义 Nim 和 $\oplus$、Nim 积 $\otimes$ 两种运算：

自 Nim 数的运算

对于 Nim 数 a,b，可以定义：

- Nim 和 $a\oplus b=\mathrm{mex}(\{a'\oplus b:a'<a,\ a'\in\mathbf{N}\}\cup\{a\oplus b':b'<b,\ b'\in\mathbf{N}\})$，
- Nim 积 $a\otimes b=\mathrm{mex}(\{(a'\otimes b)\oplus(a\otimes b')\oplus(a'\otimes b'):a'<a,\ b'<b,\ a',b'\in\mathbf{N}\})$．

全体 Nim 数在运算 $\oplus$ 和 $\otimes$ 下构成一个特征为 2 的 域→9.16.1@p1805．而且，这些运算以及它们的逆运算，对于前 2^{2^n} 个 Nim 数是封闭的；这就得到一系列大小为 2^{2^n} 的 有限域→9.16.4@p1849 $\mathbf{F}_{2^{2^n}}$．

常见的公平游戏

尽管 Sprague–Grundy 理论完全解决了公平游戏的问题，但是，处理实际的公平游戏时，直接应用 Sprague–Grundy 定理计算效率仍然不高．比如，Nim 游戏中，暴力计算 Sprague–Grundy 值的复杂度是指数级的．因此，往往需要通过打表的方式猜测具体的公平游戏的结论．

本节列举了一些常见的公平游戏及其结论．叙述结论时，本节只给出了必胜和必败状态的判断法则．至于必胜策略，就是进行恰当的操作，使得留给对手的局面恰好为必败状态．由于算法竞赛中经常出现这些游戏的变体，所以，掌握每个游戏的结论的证明过程也很重要．

本节结论的证明方法

本节结论的证明都是验证性的．对于一个游戏，结论中会描述它的先手必败状态和先手必胜状态．证明中，只需要验证从一个先手必败状态出发，只能得到先手必胜状态；而从先手必胜状态出发，总能得到至少一个先手必败状态．要将这些证明改写为严格的证明，需要建立博弈图，然后对博弈图上的状态应用数学归纳法，而这些验证的步骤就是其中的归纳部分．

Bachet 游戏

相较于单堆 Nim 游戏，Bachet 游戏限制了每次可以取走的石子的数量．

Bachet 游戏

有一堆石子，共计 n 枚．两名玩家轮流取走至少 1 枚、至多 k 枚石子．取走最后一枚石子的玩家获胜．

对此，有如下结论：

定理

游戏先手必败，当且仅当 $n \equiv 0 \pmod{k+1}$．

证明一

当 $n \not\equiv 0 \pmod{k+1}$ 时，只要取走 $n \bmod (k+1) \in [1,k]$ 枚石子，就能保证对手处于必败状态．因此，此时是先手必胜状态．

反过来，当 $n \equiv 0 \pmod{k+1}$ 时，要么已经没有选择，要么自己取走 k' 枚石子后，对手紧接着可以取走 $k+1-k'$ 枚石子，让自己回到必败状态．

证明二

作为 Sprague–Grundy 定理的应用，可以计算 $f(n)$ 为只剩下 n 枚石子时，对应局面的 SG 函数值．

对于 $n \le k$，可以归纳地证明 $f(n)=n$．这与单堆 Nim 游戏相同，因为取走石子数目的限制没有发挥作用．对于 $n>k$，需要证明 $f(n)=n \bmod (k+1)$．由于每次可以取走 1 到 k 枚石子，有

$$f(n)=\operatorname{mex}\{f(n-k), f(n-k+1), \cdots, f(n-1)\}.$$

由归纳假设，这遍历了模 $k+1$ 的全体余数，除了 $n \bmod (k+1)$．因此，就有 $f(n)=n \bmod (k+1)$．

Moore's Nim-k 游戏

相较于 Nim 游戏，Moore's Nim-k 游戏允许一次性从至多 k 个石子堆中取石子．

Moore's Nim-k 游戏

共有 n 堆石子，第 i 堆有 a_i 枚石子．两名玩家轮流取走至少 1 堆、至多 k 堆中的任意多枚石子，但不能不取．取走最后一枚石子的玩家获胜．

对此，有如下结论：

定理

将每一堆石子的数目都表示为二进制数，并对每个数位 d，都统计有多少堆石子数目的第 d 位是 1，并计算这个数目对于 $(k+1)$ 的余数．如果对于每个数位，这个余数都等于 0，那么先手必败；否则，先手必胜．

证明

仿照 Nim 游戏的结论的证明，很容易证明本结论．设 d 为余数不为 0 的最高二进制位，且对应的余数为 $k' \le k$．那么，必胜策略为，在石子数目二进制第 d 位为 1 的石子堆中，选择 k' 堆，并选择移走的石子数目恰好使得对手局面中，每个数位的余数都是 0．唯一需要说明的是，最后取走石子数量的选择总是可行的．

实际上，只要将选定的 k' 堆石子的第 d 位都变为 0，就能使得结果中，第 d 位余数变为 0；由于更高位不变，这些堆的更低位可以任意设置．对于更低的数位，从高到低依次处理：设已选定 c 堆，其余堆在该位的余数为 r．如果 $c \ge k+1-r$，就将选定堆中恰好 $k+1-r$ 堆的该位设为 1；否则，有 $c+r \le k$，就再选定 r 个该位为 1 的堆，将它们的该位变为 0，并将原先选定堆的该位都设为 0．这样，选定的堆数始终不超过 k．

阶梯 Nim 游戏

阶梯 Nim 游戏稍微复杂一些，它允许石子在相邻的堆之间移动．

阶梯 Nim 游戏

共有 n 堆石子，第 i 堆有 a_i 枚石子．两名玩家轮流操作，每次操作中，要么取走第 1 堆石子中的任意多枚，要么将第 $i>1$ 堆石子中的任意多枚移动到第 $i-1$ 堆，但不能不做任何操作．取走最后一枚石子的玩家获胜．

对此，有如下结论：

定理

游戏先手必败，当且仅当奇数堆石子数量的 Nim 和 $a_1 \oplus a_3 \oplus \cdots \oplus a_{n-1+(n \bmod 2)} = 0$.

证明

任何玩家将偶数堆的石子移动到奇数堆时，对手都可以将这些石子继续移动到下一个偶数堆（或移走），因此，这样的移动不会影响奇数堆的局面．此时，每一个奇数堆向下移动到相邻的偶数堆（或移走）都可以看作独立的单堆 Nim 游戏．根据 Sprague–Grundy 定理关于游戏的和的结论，阶梯 Nim 游戏的 SG 函数值，是这些子游戏的 SG 函数值的 Nim 和．这就得到上述结论．

Fibonacci Nim 游戏

Fibonacci Nim 游戏类似 Bachet 游戏，只有一堆石子，且限制了每次取走的数量．与 Bachet 游戏不同，Fibonacci Nim 游戏中，每次取走的数量的限制是动态的．

Fibonacci Nim 游戏

有一堆石子，共计 n 枚．两名玩家轮流取石子．第一个行动的玩家不限制取走的石子数目，但是不能取完石子；随后，每次取走的石子数目不得超过上次（指对手回合）取走的石子数目的二倍．每次取走的石子的数目不得为 0．取走最后一枚石子的玩家获胜．

对此，有如下结论：

定理

游戏开始时，先手必败，当且仅当石子数目 n 是 Fibonacci 数→9.13.4@p1629.

证明

设 q 为当前局面可移走石子数量的限额（quota）．那么，第一回合中，$q=n-1$；而之后的回合中，q 是上次（对手）移走的石子数目的二倍．考察剩余石子数目 n 的 Fibonacci 编码→9.13.4@p1629，也就是将 n 唯一地分解为一系列不相邻的、正的 Fibonacci 数的和．需要证明的是，当前状态是必胜状态，当且仅当 q 大于等于 n 的分解中的最小 Fibonacci 数．

必胜策略是：如果可以，移走所有剩余石子；否则，移走分解中最小的 Fibonacci 数．由于分解中，次小的 Fibonacci 数一定严格大于最小的 Fibonacci 数的两倍，所以，只要处于必胜状态的当前回合取不走所有石子，对手在下一回合也取不走次小的 Fibonacci 数（也就是下一回合最小的 Fibonacci 数），对手一定处于必败状态．

反过来，如果当前处于必败状态，那么，设当前取走的数目为 k，它一定严格小于当前分解中的最小 Fibonacci 数 F．假设下一回合最小的 Fibonacci 数是 F'，它一定也是 $F-k$ 对应的分解中最小的 Fibonacci 数．如果 $F'\leq 2$，那么，由 $k\geq 1$ 可知下一回合的限额 $2k\geq 2\geq F'$，已是必胜状态．否则，设 $F'=F''+F'''$ 且 $F''>F'''$，也就是说，F''',F'',F' 是 Fibonacci 数列中相邻三项．如果 $k<F''$，那么，利用 Fibonacci 编码计算 $k+(F-k)$ 时，不需要进位，自然得不到 F．所以，一定有 $k\geq F''$．这就说明，下一回合的限额 $2k>F''+F'''=F'$，是必胜状态．

Wythoff 游戏

Wythoff 游戏允许同时从多堆石子中移除，但是要求每堆移除相同数量的石子．

自 Wythoff 游戏

有两堆石子，分别有 a_1 和 a_2 枚石子．两名玩家轮流从其中一堆或两堆中取石子，不能不取，但要求从两堆都取石子时，取走的石子数量必须相同．取走最后一枚石子的玩家获胜．

对此，有如下结论：

定理

不妨设 $a_1\leq a_2$，那么，先手必败，当且仅当 $a_1=\lfloor(a_2-a_1)\phi\rfloor$，其中，$\phi=(\sqrt{5}+1)/2$ 是黄金分割比．

为了证明这一结论，需要用到如下引理：

自 Beatty 序列

设 $r>1$ 为无理数．它生成的 Beatty 序列是 $\mathcal{B}_r=\{\lfloor kr\rfloor:k\in\mathbf{N}_+\}$.

Rayleigh 定理

设 $r,s>1$ 是两个无理数，且 $\dfrac{1}{r}+\dfrac{1}{s}=1$．那么，序列 $\mathcal{B}_r$ 和 $\mathcal{B}_s$ 构成正整数集 $\mathbf{N}_+$ 的一个分划．此时，它们也称为互补的 Beatty 序列.

证明

设 $\mathcal{A}_r = \{kr : k \in \mathbf{N}_+\}$. 考虑将集合 $\mathcal{A} = \mathcal{A}_r \cup \mathcal{A}_s$ 里的元素排序得到序列 $\{a_i\}_{i\in\mathbf{N}_+}$. 只需要证明 $i = \lfloor a_i \rfloor$ 对所有 $i \in \mathbf{N}_+$ 都成立，就能得到 $\mathcal{B}_r \cup \mathcal{B}_s$ 是正整数集 $\mathbf{N}_+$ 的一个分划.

首先，证明序列里没有重复的元素. 假设不然，存在 $k, \ell \in \mathbf{N}_+$ 使得 $kr = \ell s$ 成立. 那么，有

$$\frac{\ell}{k} = \frac{r}{s} = r - 1.$$

但是，等式左侧是有理数，等式右侧是无理数，矛盾. 因此，序列的数字各不相同.

然后，证明集合 $\mathcal{A}$ 中小于等于 a_i 的数恰有 $\lfloor a_i \rfloor$ 个. 不妨设 $a_i \in \mathcal{A}_r$，即 $a_i = kr$，那么，对集合 $\mathcal{A}_r$ 和 $\mathcal{A}_s$ 中的元素分别计数，就得到集合 $\mathcal{A}$ 中小于等于 a_i 的元素恰有

$$k + \left\lfloor \frac{kr}{s} \right\rfloor = k + \lfloor k(r-1) \rfloor = \lfloor kr \rfloor = \lfloor a_i \rfloor$$

个. 进而，由于序列 $\{a_i\}$ 是严格递增的，小于等于 a_i 的数恰有 i 个. 这就得到 $i = \lfloor a_i \rfloor$.

由此，可以得到前述结论的证明.

Wythoff 游戏结论的证明

对于定理所述的所有 $a_1 < a_2$ 且先手必败的状态 (a_1, a_2)，令 $k = a_2 - a_1 \in \mathbf{N}_+$，都有 $a_1 = \lfloor k\phi \rfloor$ 且 $a_2 = \lfloor k(\phi+1) \rfloor$. 由于 ϕ 是黄金分割比，所以 $\dfrac{1}{\phi} + \dfrac{1}{\phi+1} = 1$. 由 Rayleigh 定理可知，序列 $\{\lfloor k\phi \rfloor\}$ 和 $\{\lfloor k(\phi+1) \rfloor\}$ 构成正整数集 $\mathbf{N}_+$ 的一个分划. 这其实说明，所有 $a_1 < a_2$ 且先手必败的状态 (a_1, a_2) 中，分量 a_1 和 a_2 恰取遍全体正整数一次，且它们的差 $a_2 - a_1$ 也恰取遍全体正整数一次.

由于 Wythoff 游戏中，一次合法的操作要么保持分量之一不变，要么保持分量之差不变，所以，从一个先手必败状态开始，确实无法通过一次合法的操作得到另一个先手必败状态. 反过来，对于任何先手必胜状态 (a_1, a_2)，不妨设 $a_1 \le a_2$，并令 $k = a_2 - a_1$. 如果 $a_1 > \lfloor k\phi \rfloor$，那么，先手玩家可以从两堆石子各取 $(a_1 - \lfloor k\phi \rfloor)$ 枚，将局面变为必败状态. 否则，由于 (a_1, a_2) 是先手必胜状态，有 $a_1 < \lfloor k\phi \rfloor$，从而 $k \ge 1$. 由前一段的结论（以及必败状态 $(0,0)$），对于这个 a_1 必然存在唯一一个必败状态 $(a_1, a_{(2)'})$（这里不要求 $a_1 \le a_{(2)'}$）. 进而，如果 $a_1 \ge a_{(2)'}$，显然有 $a_{(2)'} \le a_1 < a_2$；否则，如果 $a_1 < a_{(2)'}$，那么，可以取 $k' = a_{(2)'} - a_1$ 使得 $a_1 = \lfloor k'\phi \rfloor$，又有 $a_1 < \lfloor k\phi \rfloor$，故而 $k' < k$，因此 $a_{(2)'} = a_1 + k' < a_1 + k = a_2$. 所以，只要 $a_1 < \lfloor k\phi \rfloor$，就一定有 $a_{(2)'} < a_2$，先手玩家只需要从第二堆石子中取走 $(a_2 - a'_2)$ 枚石子就可以使得局面变为必败状态.

翻硬币游戏

翻硬币游戏也是一类常见的公平组合游戏.

翻硬币游戏

设 $(S, \preceq)$ 是一个 良基偏序集→9.20@p1951，映射 $f : S \to \mathcal{P}\mathcal{P}S$ 满足对于所有 $s \in S$，集合 $f(s)$ 都非空，对于 $T \in f(s)$ 都有 $s \in T$，而且对于所有 $t \in T$，都有 $t \preceq s$. 集合 S 的每个元素处都有一枚硬币，可能正面朝上也可能背面朝上. 玩家轮流行动，选择一枚正面朝上的硬币 s 和集合 $T \in f(s)$，并将集合 T 中所有硬币翻转. 将所有硬币都翻转到背面朝上的玩家获胜.

翻硬币游戏其实是一大类游戏. 取决于具体的偏序集 S 和映射 f 的选择，翻硬币游戏的具体形式也有所不同. 游戏描述中，映射 f 需要满足的条件是在说，每次玩家选择翻转硬币的集合 T 中，一定存在一枚正面朝上的硬币 s，使得集合 T 中其余元素都排在 s 前面. 这保证了游戏可以在若干步后终止.

例子

1. 设 $S=\{1,2,\cdots,n\}$ 且 $f(s)=\{\{t,s\}:t\le s\}$. 这相当于说，有一排 n 枚硬币，每次翻转一枚正面朝上的硬币，并且可以选择一枚它左侧的硬币翻转.
2. 设 $S=\{1,2,\cdots,n\}$ 且 $f(s)=\{[t,s]:t\le s\}$. 这相当于说，有一排 n 枚硬币，每次翻转一段连续的硬币，但是必须保证这些硬币中最右侧的那枚硬币在翻转前是正面朝上的.
3. 设 $S=\{1,2,\cdots,n\}^2$ 且 $f(s)=\{\{s\}\}$. 这相当于说，有 n 行 n 列硬币，每次只能翻转一枚正面朝上的硬币.
4. 设 S 是一棵有根树的结点集合，且 $f(s)$ 是顶点 s 到树根的路径经过的结点集合的子集中，所有包含 s 自身的子集的集合. 这相当于说，有一棵有根树，每个结点处放置一枚硬币，每次翻转一枚正面朝上的硬币，并且可以选择它的若干个祖先结点处的硬币翻转.

尽管翻硬币游戏种类繁多，但是它们的求解思路是一致的. 对于翻硬币游戏 (S,f)，设 G_s 为只有元素 s 处的硬币正面朝上的局面. 这些局面称为基础局面. 那么，任意一个局面 G 都可以看作是这些基础局面对应的游戏的和. 也就是说，以下结论成立：

定理

对于翻硬币游戏 (S,f) 和局面 G，设其中正面朝上的硬币所处位置的集合为 $H(G)\subseteq S$. 那么，局面 G 的 SG 函数值就是

$$\mathrm{SG}(G)=\bigoplus_{s\in H(G)}\mathrm{SG}(G_s).$$

证明

考虑一个相关的游戏：一个局面 G' 中，集合 S 的每个元素处都放置有若干枚石子；玩家每次行动时，都可以取走 s 处的一枚石子，并选取集合 $T\in f(s)$，再在集合 $T\setminus\{s\}$ 中的各个元素处均放置一枚石子. 对于这类游戏，仍然可以定义基础局面 G'_s，即仅在位置 s 处放置有一枚石子的局面. 这类游戏中，每个局面均为其所有石子对应基础局面的和. 这是因为只要放置新石子时将它对应到取走的石子上，就可以将游戏过程中出现的每枚石子都对应到初始局面中的各个石子上，进而对应初始局面不同石子的子游戏进程互不干扰，整个游戏就可以看作是这些子游戏的和. 由于相同位置石子对应基础局面的 SG 值是一样的，所以利用异或值的特性可知，局面 G' 的 SG 值仅由各堆石子数量的奇偶性决定，而与具体数量无关. 因此，对于游戏局面 G'，如果记其石子数量为奇数的位置集合为 $H(G')$，那么，本段的分析可以总结为公式：

$$\mathrm{SG}(G')=\bigoplus_{s\in H(G')}\mathrm{SG}(G'_s).$$

由此，下文只需要建立游戏 G' 与游戏 G 的等价性就可以证明定理中的公式.

需要说明的是，对于新游戏的局面 G' 与翻硬币游戏的局面 G，只要 G' 中石子数量为奇数的位置与 G 中硬币正面朝上的位置相同，就有 G' 与 G 等价. 根据 Sprague–Grundy 函数→9.18@p1901 一节的引理 2，这等价于证明局面 $G+G'$ 是必败状态. 后手玩家的胜利策略很简单：如果先手玩家选择取走 s 处的石子且该处不止一枚石子，那么后手玩家直接模仿先手玩家的行为；否则，后手玩家选择和先手玩家同样的 s 和 $T\in f(s)$，但是选择和先手玩家不同的子游戏，即先手取石子后手就翻硬币，先手翻硬币后手就取石子. 无论先手玩家如何操作，后手玩家都可以继续操作，并保证残余局面中石子数量为奇数的位置与硬币正面朝上的位置相同. 这样，游戏必然结束在先手玩家没有合法操作时，因此，先手必败. 定理由此得证.

利用这一结论，判断某一局面是否必胜，只需要计算其中所有正面朝上的硬币对应的基础局面的 SG 函数值，再求 Nim 和即可．这些基础局面的 SG 函数值也不难计算，因为它们的后继局面已经由映射 f 给出，且后继局面的 SG 值可以归纳地计算：

$$\mathrm{SG}(G_s) = \operatorname*{mex}_{T\in f(s)} \bigoplus_{t\in T\setminus\{s\}} \mathrm{SG}(G_t).$$

这相当于提供了一个基础局面 SG 函数值的递推公式．

二分图博弈

前置知识：二分图最大匹配→11.24.2@p2959

本节的最后，讨论二分图博弈．尽管这个游戏常称作二分图博弈，但是它的描述和结论的证明都与二分图的结构无关，所以，它的结论实际上对于一般的无向图都成立．但是，一般图的最大匹配较为复杂，所以这一结论常出现在二分图的题目中．

自 二分图博弈

两个玩家轮流行动．每个玩家面临的局面都由一个无向图 $G=(V,E)$ 和它的一个顶点 $v\in V$ 构成．在一名玩家的回合中，若当前局面为 (G,v)，则该玩家必须选择一个与 v 相邻的顶点 u．随后，将顶点 v 及其所有关联边从图 G 中删除，得到残余图 G'．新的局面即为 (G',u)，交由下一位玩家．若某位玩家在其回合开始时，当前顶点 v 在图中没有相邻顶点（即不存在合法选择），则该玩家无法行动，并因此输掉游戏．

对此，有如下结论：

定理

游戏先手必胜，当且仅当顶点 v 是图 G 的最大匹配关键点，也就是说，在图 G 的所有最大匹配中，顶点 v 都是匹配点．

证明

首先，设顶点 v 是图 G 的最大匹配关键点．设 G 的一个最大匹配为 M．此时，先手可以将局面移动到在 M 中与顶点 v 匹配的顶点 u．由于顶点 v 出现在所有图 G 的最大匹配中，所以，残余图 G' 的最大匹配的大小至多是 $|M|-1$；而且将 M 去掉边 (u,v) 就能得到图 G' 的一个大小为 $|M|-1$ 的匹配 M'：结合这两点就知道，M' 是图 G' 的一个最大匹配．但是，后手玩家所处的局面中，顶点 u 并不是匹配 M' 的一个匹配点．因此，后手玩家必然处于一个必败状态．

反过来，假设存在最大匹配 M 使得 v 是未匹配点．由于 M 不含顶点 v，它也是残余图 G' 的一个最大匹配；因此，G' 的任何最大匹配 N 都是 G 的一个不含顶点 v 的最大匹配．与顶点 v 相邻的顶点 u 一定是 N 的匹配点；否则，就可以将边 (u,v) 添加到 N 中，得到 G 的一个更大的匹配．这说明 u 是 G' 的最大匹配关键点．因此，无论先手怎么选择，后手都处于一个必胜状态．

求出二分图最大匹配关键点的算法详见 二分图最大匹配页面→11.24.2@p2959．

另外，二分图博弈还有一个变体：

自 二分图博弈的变体

设 $G=(V,E)$ 是一个无向图，且图的每个顶点上都放置了一枚石子．两名玩家轮流行动取走石子．游戏开始时，先手玩家可以取走任何一枚石子；后续的回合中，每名玩家取走石子的顶点必须与上一回合中对方取走石子的顶点相邻．最先无法取走石子的玩家输掉游戏．

显然，这个变体相当于在前文所述二分图博弈中，让先手玩家选择初始局面，然后从后手玩家开始二分图博弈．因此，这个变体中，先手玩家必败，当且仅当每个顶点都是最大匹配关键点，亦即图 G 存在 完美匹配→11.24.1@p2949．

反常 Nim 游戏

本节讨论反常 Nim 游戏的求解．

自 反常 Nim 游戏

共有 n 堆石子，第 i 堆有 a_i 枚石子．两名玩家轮流取走任意一堆中的任意多枚石子，但不能不取．取走最后一枚石子的玩家失败．

对此，有如下结论：

定理

反常 Nim 游戏中，状态 $(a_1, a_2, \cdots, a_n)$ 是必败状态 $\mathcal{P}$，当且仅当

1. 存在 i 使得 $a_i > 1$，且 Nim 和 $a_1 \oplus a_2 \oplus \cdots \oplus a_n = 0$，或者
2. 对于所有 i 都有 $a_i \leq 1$，且剩余的非空石子堆数是奇数．

证明

由于无法操作是先手必胜态 $\mathcal{N}$，所以，可以归纳地证明，如果每堆石子都至多一枚，那么非空石子堆数是奇数就对应着先手必败态 $\mathcal{P}$，非空石子堆数是偶数就对应着先手必胜态 $\mathcal{N}$．

接下来，考察有些堆石子的数量严格大于 1 的情况．

情形 A：如果只有一堆石子的数量严格大于 1，那么，此时 Nim 和一定不为 0．而且，先手玩家可以选择转移到全部堆的石子数量均不超过 1 的局面，并且可以控制剩余的非空石子堆数的奇偶性．因此，此时为先手必胜态 $\mathcal{N}$．

情形 B：现在，有不止一堆石子的数量严格大于 1，那么，无论怎么操作，下一个局面中，都至少有一堆石子的数量严格大于 1．根据归纳假设，下一局面中，先手必败对应着 Nim 和为零，先手必胜对应着 Nim 和不为零．这与正常 Nim 游戏的归纳假设完全相同．因此，重复 Nim 游戏的论证，就能知道，当前局面同样符合 Nim 和为零对应先手必败状态的结论．

有向图游戏

本文讨论的公平组合游戏，要求同一局面不能出现两次，也不存在平局的可能性．因此，对应的博弈图总是有向无环图．本节放宽了这一限制，讨论如何在一般的有向图上判定各个状态是先手必胜、先手必败或平局．

有向图游戏的规则和其他的公平组合游戏大体一致：从起始状态出发，轮流沿着有向图的边移动一步，直到无路可走．根据游戏是正常规则还是反常规则，无法移动的玩家分别是败者和胜者．在这样的游戏里，每个状态的胜负情况共有三种可能性：先手必胜、先手必败、平局．平局中游戏永远不会终止．尽管稍微复杂一些，但是关于必败状态和必胜状态的 引理→9.18@p1901 依然成立，而剩下的状态就是平局状态：

- 如果一个状态有后继状态，那么它先手必胜，当且仅当后继状态之一是必败状态；
- 如果一个状态有后继状态，那么它先手必败，当且仅当所有后继状态都是必胜状态；
- 如果一个状态无法分类为必胜状态和必败状态，那么它就是平局状态．

要将所有状态分类为这三种状态，只需要采用类似 拓扑排序→11.8@p2729 的思路：

1. 初始化时，记录所有状态的出度，将所有出度为零的状态压入队列，并根据游戏是正常规则或是反常规则分别设为必败状态或必胜状态．

2. 弹出队首状态. 如果是必败状态, 则将所有尚未判断的前驱状态设为必胜状态; 否则, 当前状态是必胜状态, 将它的所有前驱状态的出度减一, 并将出度为零的前驱状态设为必败状态. 将可以判断是必胜或必败状态的前驱状态压入队列.
3. 算法在队列为空时终止. 尚未判断为必胜或必败状态的状态均为平局状态.

这一算法可以在 $O(|V|+|E|)$ 时间内将所有状态分类.

例题

本节讨论一些典型的例题.

Luogu P2148 [SDOI2009] E&D[1]

有 $2n$ 堆石子. 对于 $k=1,2,\cdots,n$, 石子堆 $2k-1$ 和 $2k$ 分为一组. 两名玩家轮流操作, 每次选择一组石子堆, 将其中一堆移走, 并将另一堆分为非空的两堆, 放到该组石子堆所在的两个位置. 如果所有石子堆都只有一枚石子, 当前玩家就没有合法操作, 输掉游戏. 给定每堆石子的数量 $\{a_i\}_{i=1}^{2n}$, 问是否为先手必胜状态.

解答

显然, 不同组石子堆的游戏相互独立, 所以, 只要计算每组游戏的 SG 函数值, 就能计算出整个游戏的 SG 值, 进而判断是否为必胜状态. 关键在于如何计算每组石子堆的 SG 函数值. 这并不容易. 解决这类博弈论问题的常见思路是打表. 设一组石子堆中石子数量分别为 (i,j) 时, SG 值为 $f(i,j)$. 那么, 写一个暴力打表的程序, 就得到如下结果:

```
0 1 0 2 0 1 0 3 0 1 0 2 0 1 0 4
1 1 2 2 1 1 3 3 1 1 2 2 1 1 4 4
0 2 0 2 0 3 0 3 0 2 0 2 0 4 0 4
2 2 2 2 3 3 3 3 2 2 2 2 4 4 4 4
0 1 0 3 0 1 0 3 0 1 0 4 0 1 0 4
1 1 3 3 1 1 3 3 1 1 4 4 1 1 4 4
0 3 0 3 0 3 0 3 0 4 0 4 0 4 0 4
3 3 3 3 3 3 3 3 4 4 4 4 4 4 4 4
0 1 0 2 0 1 0 4 0 1 0 2 0 1 0 4
1 1 2 2 1 1 4 4 1 1 2 2 1 1 4 4
0 2 0 2 0 4 0 4 0 2 0 2 0 4 0 4
2 2 2 2 4 4 4 4 2 2 2 2 4 4 4 4
0 1 0 4 0 1 0 4 0 1 0 4 0 1 0 4
1 1 4 4 1 1 4 4 1 1 4 4 1 1 4 4
0 4 0 4 0 4 0 4 0 4 0 4 0 4 0 4
4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
```

这个表很具有规律性. 有一个简单的观察: 表格分成若干个 2×2 的矩阵, 且左上角处总是 0, 而其余三个值总是一样的. 于是, 不妨将这个表格压缩, 将每个 2×2 的矩阵都压缩为除了左上角之外那个共同的数值:

```
1 2 1 3 1 2 1 4
2 2 3 3 2 2 4 4
1 3 1 3 1 4 1 4
3 3 3 3 4 4 4 4
1 2 1 4 1 2 1 4
2 2 4 4 2 2 4 4
1 4 1 4 1 4 1 4
4 4 4 4 4 4 4 4
```

可以发现，这个压缩的表格是前面完整表格相同位置的值加一．其实问题已经解决了．设下标从 0 开始，那么表格中 (i,j) 处的值 $g(i,j)$ 可以由如下递推公式给出：

$$g(i,j)=\begin{cases}0, & \text{if } 2\mid i \text{ and } 2\mid j,\\ g(\lfloor i/2\rfloor,\lfloor j/2\rfloor)+1, & \text{otherwise.}\end{cases}$$

要求的 SG 函数 $f(i,j)=g(i-1,j-1)$．利用这一递推公式，算法可以在 $O(\log\max\{i,j\})$ 时间内求出 $f(i,j)$ 的取值．

当然，可以通过简单的归纳法得到 $g(i,j)$ 其实就是将 i 和 j 反复同时除以 2 得到两个偶数的最少次数．换句话说，它就是 i 与 j 的按位或中末尾连续 1 的个数．由此，也可以直接利用 `__builtin_ctz(~(i | j))` 算出该值．

这类题目中，只要通过打表观察的方法得到 SG 函数表达式，它都很容易通过归纳法证明，因而解题的关键在于以某种形式获得这些结论而非推导．例如，已知结论后，本题中的递推关系可以归纳证明如下．设 S_k 为将 k 枚石子分成非空的两堆能得到的局面的 SG 值集合，那么，$f(i,j)=\operatorname{mex}(S_i\cup S_j)$．故而，$S_k$ 有递推关系：

$$S_k=\{\operatorname{mex}(S_i\cup S_j):i+j=k,\ i,j\in\mathbf{N}_+\}.$$

需要证明的是，$d\in S_k$ 当且仅当 $(k-1)$ 的二进制表示中第 d 位（最低位是第 0 位）是 1．

利用数学归纳法．归纳起点 $S_1=\emptyset$ 显然成立．假设命题对小于 k 的正整数都成立．那么，$d\in S_k$，当且仅当存在 $i,j\in\mathbf{N}_+$ 使得 $i+j=k$，$(i-1)$ 和 $(j-1)$ 两个数的第 d 位均为 0，且对于每个 $d'<d$，两者的第 d' 位至少有一个为 1．显然，存在这样一种拆分，当且仅当只考虑第 $0\sim d$ 位的部分，即模 2^{d+1} 时，$(k-1)=(i-1)+(j-1)+1$ 的取值范围为 $[2^d,2^{d+1})$．这一条件就等价于 $(k-1)$ 的第 d 位是 1．由此，归纳步骤成立．原命题得证．

参考代码

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

int g(int i, int j) {
  return i % 2 == 0 && j % 2 == 0 ? 0 : g(i / 2, j / 2) + 1;
}

int f(int i, int j) { return g(i - 1, j - 1); }

int main() {
  int t;
  std::cin >> t;
  for (; t; --t) {
    int n;
    std::cin >> n;
    int v = 0;
    for (int i = 0; i < n / 2; ++i) {
      int x, y;
      std::cin >> x >> y;
      v ^= f(x, y);
    }
    std::cout << (v ? "YES" : "NO") << '\n';
  }
  return 0;
}
```

```
}

```

Luogu P5675 [GZOI2017] 取石子游戏[2]

有 n 堆石子，第 i 堆有 a_i 枚．两人玩 Nim 游戏．现在，可以任意指定若干堆石子作为初始局面，并指定其中一堆石子要求先手玩家首轮必须从中取走石子，但不能指定取走石子的数目．问有多少种指定方式，使得先手无法获得胜利．数据满足 $n, a_i \leq 200$．

解答

对于这类问题，需要利用常见游戏的结论，并结合其他部分知识来进行解答．假设指定先手必须取走第 i 堆石子，且指定的所有石子堆数量 Nim 和为 v，那么，先手无法获得胜利，当且仅当 $a_i \leq a_i \oplus v$，也就是说，第 i 堆石子数量 a_i 不超过除第 i 堆外剩余石子堆数量 Nim 和 $a_i \oplus v$．由于数据范围很小，直接枚举指定首轮取石子的堆；枚举到第 i 堆时，剩余每个堆选或不选，得到不同 Nim 和的方案数可以通过 DP 计算出来，将最后得到的方案数中大于等于 a_i 的部分加总起来即可．

参考代码

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

int main() {
  constexpr int M = 1e9 + 7;
  constexpr int L = 256;
  int n;
  std::cin >> n;
  std::vector<int> a(n);
  for (auto& x : a) std::cin >> x;
  int res = 0;
  for (int i = 0; i < n; ++i) {
    std::array<int, L> dp = {};
    dp[0] = 1;
    for (int j = 0; j < n; ++j) {
      if (j == i) continue;
      std::array<int, L> _dp = {};
      for (int v = 0; v < L; ++v) {
        _dp[v] = (dp[v] + dp[v ^ a[j]]) % M;
      }
      dp = std::move(_dp);
    }
    for (int v = a[i]; v < L; ++v) {
      res = (res + dp[v]) % M;
    }
  }
  std::cout << res << std::endl;
  return 0;
}
```

```
31
```

Luogu P2599 [ZJOI2009] 取石子游戏[3]

有 n 堆石子，第 i 堆有 a_i 枚．两人轮流取走石子，每次都只能从最左或最右的两堆中选择一堆取走任意枚石子，但不能不取．取走最后一枚石子的玩家胜利．问先手是否必胜．

解答

由于本题中并不存在相互独立的子游戏，所以这道题目原则上只用到 判断必败和必胜状态的引理→9.18@p1901．从最简单的情形开始分析．当 $n \leq 2$ 时，就是 Nim 游戏．当 $n \geq 3$ 时，问题变得复杂．但是，由于可操作的石子堆只能是两端的石子堆，不妨设它们中石子数量分别为 x 和 y．进一步地，设 $f(x,y)$ 为先手必胜状态的指示函数，即先手必胜时 $f(x,y)=1$，否则 $f(x,y)=0$．容易发现，$f(x,y)$ 的取值满足递推关系：$f(x,y)=0$，当且仅当对于所有 $s<x$ 和 $t<y$ 都有 $f(x,t)=f(s,y)=1$．递推起点在 $x=0$ 或 $y=0$ 时，此时，游戏已经不足 n 堆石子，需要进一步考虑中间石子堆的数量．因此，不妨暂时假设 $f(x,0)$ 和 $f(0,y)$ 是已知的，考虑如何从它们的取值推出所有 $f(x,y)$ 的取值．这并不困难．考虑下标集合为 $\mathbf{N}\times\mathbf{N}$ 的无穷大矩阵，求 $f(x,y)$ 相当于向里面填 0 和 1，需要满足的条件是，每行和每列都至多一个 0，且如果同行和同列中之前的位置都没有出现过 0，该位置一定是 0．每行中 0 的位置实际上定义了一个从行号 x 到列号 y 的函数．简单尝试几个例子（即打表）之后就可以发现，如果设使得 $f(x,0)=0$ 的唯一的 x 是 x_0，使得 $f(0,y)=0$ 的唯一的 y 是 y_0，那么，对于任何正整数 x，使得 $f(x,y)=0$ 成立的

$$y=\begin{cases}0, & x=x_0,\\ x-1, & x_0<x\leq y_0,\\ x+1, & y_0\leq x<x_0,\\ x, & \text{otherwise.}\end{cases}$$

也就是说，只要知道 x_0 和 y_0，就可以在 $O(1)$ 时间内计算出任意 $f(x,y)$ 的值，判断当前状态是否为先手必胜状态．而 x_0 和 y_0 可以递归计算．例如，x_0 是使得 $f(x,0)=0$ 的唯一解，但同时，$f(x,0)$ 的取值可以通过移除最右侧石子堆后，只考虑剩下的 $n-1$ 堆石子来计算；也就是说，只考虑前 $n-1$ 堆石子，同样可以计算一个 $f_{1,n-1}(x,y)$，那么，显然有 $f(x,0)=f_{1,n-1}(x,a_{n-1})$；类似地，移除最左侧石子堆并计算得出 $f_{2,n}(x,y)$ 后，就得到 $f(0,y)=f_{2,n}(a_2,y)$．当然，内层的函数 $f_{1,n-1}(x,y)$ 和 $f_{2,n}(x,y)$ 的计算依赖于更内层的函数．这是典型的 区间 DP→7.5@p734．每层只需要维护相应函数的 x_0 和 y_0 即可．

参考代码

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

int main() {
  // Transition helper as explained above.
  auto calc = [](int x0, int y0, int x) -> int {
    return x == x0 ? 0
                   : ((x < x0 && x < y0) || (x > x0 && x > y0)
                          ? x
                          : (x0 < y0 ? x - 1 : x + 1));
```

```
  };

  int t;
  std::cin >> t;
  for (; t; --t) {
    int n;
    std::cin >> n;
    std::vector<int> a(n);
    for (auto& x : a) std::cin >> x;

    // dp[i][j][0] = the unique value x such that
    //     f_{i-1,j}(x, a_j) = 0
    // i.e. the interval game [i, j] is in a P-position
    // when a pile of size x is attached to the LEFT.
    //
    // dp[i][j][1] = the unique value y such that
    //     f_{i,j+1}(a_i, y) = 0
    // i.e. the interval game [i, j] is in a P-position
    // when a pile of size y is attached to the RIGHT.
    std::vector<std::vector<std::array<int, 2>>> dp(
        n, std::vector<std::array<int, 2>>(n));

    // Base case: single-element interval [i, i].
    // The "left" and "right" pile values both equal a[i].
    for (int i = 0; i < n; ++i) dp[i][i][0] = dp[i][i][1] = a[i];

    // Build up intervals of increasing length d.
    for (int d = 1; d < n; ++d) {
      for (int i = 0; i + d < n; ++i) {
        dp[i][i + d][0] =
            calc(dp[i][i + d - 1][1], dp[i][i + d - 1][0], a[i + d]);
        dp[i][i + d][1] = calc(dp[i + 1][i + d][0], dp[i + 1][i + d]
[1], a[i]);
      }
    }

    // The original game corresponds to attaching nothing on the left.
    // It is a P-position if and only if the unique value x satisfying
    //     f_{-1, n-1}(x, a_{n-1}) = 0
    // is x = 0.
    std::cout << (dp[0][n - 1][0] != 0) << '\n';
  }
  return 0;
}

```

习题

首先是一些模板题．它们是对本页面的结论的简单应用：

- Luogu P2197【模板】Nim 游戏[4]
- Luogu P2252 [SHOI2002] 取石子游戏[5]
- Luogu P2594 [ZJOI2009] 染色游戏[6]

- Luogu P3185 [HNOI2007] 分裂游戏[7]
- Luogu P3480 [POI 2009] KAM-Pebbles[8]
- Luogu P4101 [HEOI2014] 人人尽说江南好[9]
- Luogu P4279 [SHOI2008] 小约翰的游戏[10]
- Luogu P6487 [COCI 2010/2011 #4] HRPA[11]
- Luogu P6560 [SBCOI2020] 时光的流逝[12]
- Luogu P7589 黑白棋（2021 CoE-II B）[13]
- AtCoder Regular Contest 168 B - Arbitrary Nim[14]

然后是一些思维性更强或更为综合的题目：

- Luogu P2490 [SDOI2011] 黑白棋[15]
- Luogu P3179 [HAOI2015] 数组游戏[16]
- Luogu P5363 [SDOI2019] 移动金币[17]
- Luogu P5970 [POI 2016] Nim z utrudnieniem[18]
- Luogu P6791 [SNOI2020] 取石子[19]
- Luogu P7864「EVOI-RD1」摘叶子[20]
- Luogu P8347「Wdoi-6」另一侧的月[21]
- AtCoder Grand Contest 002 E - Candy Piles[22]
- AtCoder Grand Contest 010 F - Tree Game[23]
- AtCoder Grand Contest 017 D - Game on Tree[24]
- AtCoder Beginner Contest 278 G - Generalized Subtraction Game[25]
- SPOJ COT3 - Combat on a tree[26]
- Codeforces 494 E. Sharti[27]
- Codeforces 1149 E. Election Promises[28]
- Codeforces 1451 F. Nullify The Matrix[29]
- Codeforces 1704 F. Colouring Game[30]

最后是一些二分图博弈的题目．由于需要用到一些二分图匹配的算法，故将它们单独列出：

- Luogu P4136 谁能赢呢？[31]
- Luogu P4617 [COCI 2017/2018 #5] Planinarenje[32]
- Luogu P4055 [JSOI2009] 游戏[33]
- Luogu P1971 [NOI2011] 兔兔与蛋蛋游戏[34]
- Codeforces 1147 F. Zigzag Game[35]

参考资料与注释

- （转载）Nim 游戏博弈（收集完全版）by exponent - 博客园[36]
- [组合游戏与博弈论]【学习笔记】by Candy? - 博客园[37]
- Nim - Wikipedia[38]
- Sprague–Grundy theorem - Wikipedia[39]
- Nimber - Wikipedia[40]
- Beatty Sequence - Wikipedia[41]
- Games on arbitrary graphs - CP Algorithms[42]
- 算法学习笔记（74): 二分图博弈 by Pecco - 知乎[43]
- Conway, John H. On numbers and games. AK Peters/CRC Press, 2000.
- Berlekamp, Elwyn R., John H. Conway, and Richard K. Guy. Winning ways for your mathematical plays, volume 1-4. AK Peters/CRC Press, 2001-2004.

[1] Luogu P2148 [SDOI2009] E&D 

[2] Luogu P5675 [GZOI2017] 取石子游戏 

[3] Luogu P2599 [ZJOI2009] 取石子游戏 

[4] Luogu P2197【模板】Nim 游戏 

[5] Luogu P2252 [SHOI2002] 取石子游戏 

[6] Luogu P2594 [ZJOI2009] 染色游戏 

[7] Luogu P3185 [HNOI2007] 分裂游戏 

[8] Luogu P3480 [POI 2009] KAM-Pebbles 

[9] Luogu P4101 [HEOI2014] 人人尽说江南好 

[10] Luogu P4279 [SHOI2008] 小约翰的游戏 

[11] Luogu P6487 [COCI 2010/2011 #4] HRPA 

[12] Luogu P6560 [SBCOI2020] 时光的流逝 

[13] Luogu P7589 黑白棋（2021 CoE-II B） 

[14] AtCoder Regular Contest 168 B - Arbitrary Nim 

[15] Luogu P2490 [SDOI2011] 黑白棋 

[16] Luogu P3179 [HAOI2015] 数组游戏 

[17] Luogu P5363 [SDOI2019] 移动金币 

[18] Luogu P5970 [POI 2016] Nim z utrudnieniem 

[19] Luogu P6791 [SNOI2020] 取石子 

[20] Luogu P7864「EVOI-RD1」摘叶子 

[21] Luogu P8347「Wdoi-6」另一侧的月 

[22] AtCoder Grand Contest 002 E - Candy Piles 

[23] AtCoder Grand Contest 010 F - Tree Game 

[24] AtCoder Grand Contest 017 D - Game on Tree 

[25] AtCoder Beginner Contest 278 G - Generalized Subtraction Game 

[26] SPOJ COT3 - Combat on a tree 

[27] Codeforces 494 E. Sharti 

[28] Codeforces 1149 E. Election Promises 

[29] Codeforces 1451 F. Nullify The Matrix 

[30] Codeforces 1704 F. Colouring Game 

[31] Luogu P4136 谁能赢呢? 

[32] Luogu P4617 [COCI 2017/2018 #5] Planinarenje 

[33] Luogu P4055 [JSOI2009] 游戏 

[34] Luogu P1971 [NOI2011] 兔兔与蛋蛋游戏 

[35] Codeforces 1147 F. Zigzag Game 

[36]（转载）Nim 游戏博弈（收集完全版） by exponent - 博客园 

[37] [组合游戏与博弈论]【学习笔记】by Candy? - 博客园 

[38] Nim - Wikipedia 

[39] Sprague–Grundy theorem - Wikipedia 

[40] Nimber - Wikipedia 

[41] Beatty Sequence - Wikipedia 

[42] Games on arbitrary graphs - CP Algorithms 

[43] 算法学习笔记（74): 二分图博弈 by Pecco - 知乎 

9.18.3 零和游戏

前置知识：博弈论简介→9.18.1@p1901

本文讨论（二人）零和游戏→9.18.1@p1901.

在零和游戏中，两名玩家的收益之和恒为零，一方的收益必然意味着另一方的损失．零和游戏可以视为常和游戏的特殊情形．反过来，任何常和游戏都可以通过对某一方的收益整体加上或减去一个常数，等价地转化为零和游戏，所以仅需要讨论零和游戏．

在算法竞赛中常见的零和游戏大致可分为两类：序贯零和游戏与同时零和游戏．

序贯零和游戏

序贯零和游戏中，两名玩家轮流行动，直到游戏终止．

序贯零和游戏中，玩家的收益函数呈现递归结构．游戏局面 S 可以分为三类，即终止局面 S_0、玩家 1 行动的局面 S_1 和玩家 2 行动的局面 S_2．假设终止局面 $s \in S_0$ 处，玩家 1 的收益为 $v(s)$，相应地，玩家

2 的收益为 $-v(s)$．因此，轮到玩家 2 行动时，最大化它的收益就相当于最小化玩家 1 的收益．由此，假设双方都采取最优策略，玩家 1 在局面 $s \in S$ 处能够获得的最大收益 $V(s)$ 满足如下递推关系：

$$V(s)=\begin{cases} v(s), & s\in S_0,\\ \max_{t\in s} V(t), & s\in S_1,\\ \min_{t\in s} V(t), & s\in S_2. \end{cases}$$

其中，$t \in s$ 表示 t 是 s 的后继局面．这就是 极小化极大思想→6.11@p700．

将这一算法应用于实际问题中，通常有如下具体方法：

- 如果游戏中涉及的局面数量较少，直接暴力实现这一算法即可．
- 如果游戏中涉及的局面数量较为庞大且没有特殊结构，可以考虑 Alpha–Beta 剪枝→6.11@p700 并结合其他搜索剪枝算法使用．
- 如果游戏中单个局面经常是多个局面的后继局面，为避免重复搜索，可以考虑记忆化搜索或其他动态规划算法．
- 如果游戏中玩家的最终收益是终局前所有行动的收益和，可以适当优化建模方式．具体地，假设到达终局 $s \in S_0$ 时，玩家 $i=1,2$ 的行动序列分别为 $\{a_j^{(i)}\}_{j=1}^{k_i}$，行动 a 对应的收益为 $w(a)$，玩家 1 的收益函数为

  $$v(s)=\sum_{j=1}^{k_1} w(a_j^{(1)})-\sum_{j=1}^{k_2} w(a_j^{(2)}).$$

  那么，可以设 $\tilde{V}(s)$ 为在局面 $s \in S$ 之后的游戏中，当前行动的玩家所得收益与对手所得收益之差的最大值．对于由玩家 1 首先行动的初始状态 s_0，有 $V(s_0)=\tilde{V}(s_0)$，因此求出 $\tilde{V}(\cdot)$ 足以求解原问题．对于 $\tilde{V}(\cdot)$，有如下递推关系：

  $$\tilde{V}(s)=\begin{cases} 0, & s\in S_0,\\ \max_{t\in s}\{w(a_{s\to t})-\tilde{V}(t)\}, & s\in S_1\cup S_2. \end{cases}$$

  其中，$a_{s\to t}$ 表示可以使得状态从 s 转移到 t 的行动，如果有多个这样的行动，取收益 $w(a)$ 最高的那个．
- 公平组合游戏都是序贯零和游戏，只需要设游戏中胜利方和失败方的收益分别为 $+1$ 和 -1．此时，收益函数 $V(\cdot)$ 的递推关系其实就是判定必胜状态和必败状态的 引理→9.18.2@p1903．

  这类问题还有一种常见的变形，即求胜利方最少需要的回合数和失败方最多可以坚持的回合数．为此，只需要注意到从终止状态开始做 BFS 并按照引理判定必胜状态和必败状态时，记录判定必胜状态和必败状态时 BFS 进行到的轮次数，就是所求的回合数．这是因为判定为必胜状态只需要一个后继状态是必败状态即可，它总是由后继状态中轮次数最小的必败状态转移而来；而判定为必败状态需要所有后继状态都是必胜状态，它总是由后继状态中轮次数最大的必胜状态转移而来．

  这一方法同样可以推广到一般的 有向图游戏→9.18.2@p1903．

例题

Codeforces 794 E. Choosing Carrot[1]

设有一个长度为 n 的数列 $\{a_i\}$．两名玩家 1 和 2 轮流从数列的两端取走一个数，直到数列中仅剩下最后一个数字为止．玩家 1 的目标是最大化这个最后剩下的数字，玩家 2 的目标是最小化它．在游戏正式开始前，玩家 1 还可以先进行 k 次行动．假设两名玩家在整个过程中都采取最优策略．对于每一个 $k=0,1,2,\cdots,n-1$，求出游戏结束时最后剩下的数字．其中，$1\le n\le 3\times 10^5$．

解答

因为无论双方怎样取走数字，数列剩余部分都是一段完整的区间，所以，游戏中的局面可以仅由区间 $[l,r]$ 和当前行动的玩家 $i=1,2$ 描述，可以使用动态规划算法求解．设 $f(l,r,i)$ 为局面由 (l,r,i) 描述时，游戏最后剩下的数字．由前文分析可知，当 $l<r$ 时，这一函数满足状态转移方程：

$$f(l,r,1)=\max\{f(l+1,r,2),f(l,r-1,2)\},$$

$$f(l,r,2)=\min\{f(l+1,r,1),f(l,r-1,1)\}.$$

终值条件为 $f(l,l,1)=f(l,l,2)=a_l$．据此，可以在 $\Theta(n^2)$ 时间内求出所有可能局面的函数值．对于每个 k，答案就是

$$g(k)=\max f(l,r,1)\text{ subject to }r-l+1=n-k.$$

这一算法无法通过原题所设的数据范围，因此需要考虑优化转移．此处有很多种处理方法，本文只提供其中一种．

将状态转移方程看作是对数列整体的操作．两个转移方程分别表示将相邻数字取最大值和最小值得到新数列，将它们分别称为「最大化操作」和「最小化操作」．每次操作都会使得数列长度减一．所有长度为 d 的区间对应结果共计 $(n-d+1)$ 个，这就相当于对序列进行 $(d-1)$ 次操作得到的序列．另外，要得到 $f(l,r,1)$ 的结果，就需要保证最后一次操作是最大化操作．因此，这些操作序列中两种操作交替进行，且结尾总是最大化操作．

考虑连续两次操作给数列带来的变化．不妨考虑首先做最小化操作，再做最大化操作．此时，数列 a_1,a_2,a_3 将变为

$$\max\{\min\{a_1,a_2\},\min\{a_2,a_3\}\}.$$

枚举 a_1,a_2,a_3 三个数字之间所有可能的大小关系可知，除了 $a_1<a_2$ 且 $a_2>a_3$（即 a_2 是严格极大值）这种情形外，这一表达式总是等于 a_2．也就是说，如果一个数列不存在任何严格极大值点，那么，连续两次操作对它的唯一影响就是删去了数列首尾各一个数字．这显然大幅简化了转移．剩下唯一的问题就是：如何保证数列不存在任何严格极大值点？事实上，只要对序列做一次最大化操作，就能保证不存在严格极大值点．故而，所有偶数次操作的结果，可以通过对初始数列进行两次操作得到的序列，逐对删去首尾数字得到；所有奇数次操作的结果，可以通过对初始数列进行一次操作得到的序列，逐对删去首尾数字得到．（零次操作的情形，即 $k=n-1$ 时，答案就是 $\max_i a_i$，需要单独处理．）

由于对序列的完整操作至多只需要进行 3 次，而后续统计答案只需要 2 次遍历，所以该算法的总时间复杂度为 $\Theta(n)$．

参考代码

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

int main() {
  int n;
  std::cin >> n;
  std::vector<int> a(n);
  for (int& x : a) std::cin >> x;
  std::vector<int> ans(n), tmp;
  tmp = a;
  for (int i = 0; i < n - 1; ++i) {
```

```
    tmp[i] = std::max(tmp[i], tmp[i + 1]);
  }
  for (int l = n / 2 - 1, r = (n - 1) / 2, ma = 0; l >= 0; --l, ++r) {
    ma = std::max({ma, tmp[l], tmp[r]});
    ans[r - l] = ma;
  }
  tmp = a;
  for (int i = 0; i < n - 1; ++i) {
    tmp[i] = std::min(tmp[i], tmp[i + 1]);
  }
  for (int i = 0; i < n - 2; ++i) {
    tmp[i] = std::max(tmp[i], tmp[i + 1]);
  }
  for (int l = (n - 1) / 2 - 1, r = n / 2 - 1, ma = 0; l >= 0; --l, ++r) {
    ma = std::max({ma, tmp[l], tmp[r]});
    ans[r - l] = ma;
  }
  ans[n - 1] = *std::max_element(a.begin(), a.end());
  for (auto x : ans) std::cout << x << ' ';
  std::cout << std::endl;
  return 0;
}

```

习题

- Luogu P2734 [USACO3.3] 游戏 A Game[2]
- Luogu P4576 [CQOI2013] 棋盘游戏[3]
- Luogu P7097 [yLOI2020] 牵丝戏[4]
- Codeforces 388 C. Fox and Card Game[5]
- Codeforces 794 E. Choosing Carrot[6]
- Codeforces 1628 D2. Game on Sum (Hard Version)[7]
- Luogu P3210 [HNOI2010] 取石头游戏[8]

同时零和游戏

同时零和游戏中，两名玩家同时行动.

同时零和游戏通常采用收益矩阵表示. 假设玩家 $i=1,2$ 的行动集合为 A_i，且当玩家 $i=1,2$ 分别采取行动 $a_i \in A_i$ 时，两人的收益分别是 $v(a_1,a_2)$ 和 $-v(a_1,a_2)$.

例子

考虑石头剪刀布游戏. 假定胜利得 1 分，失败得 -1 分，平局得 0 分. 那么，游戏中两人的收益可以表示为

$$\begin{pmatrix} 0,0 & 1,-1 & -1,1 \\ -1,1 & 0,0 & 1,-1 \\ 1,-1 & -1,1 & 0,0 \end{pmatrix}.$$

一般的二人同时游戏也可以表示为类似形式，故而也称为 双矩阵游戏[9]（bimatrix game）. 对于零和游戏，由于玩家 1 的收益矩阵和玩家 2 的收益矩阵互为相反数，所以可以只考虑玩家 1 的收益矩阵：

$$V=(v(a_1,a_2))_{(a_1,a_2)\in A_1\times A_2}=\begin{pmatrix}0&1&-1\\-1&0&1\\1&-1&0\end{pmatrix}.$$

需要解决的问题是：给定收益矩阵 $V=(v(a_1,a_2))_{(a_1,a_2)\in A_1\times A_2}$，如何求出两名玩家的最优策略和最大收益？

混合策略

相较于序贯零和游戏，同时游戏中两名玩家的角色是对称的．但是，既然已经解决了序贯零和游戏，那么不妨考虑同时游戏的序贯版本．例如，如果假定玩家 1 首先做出行动，玩家 2 再做出行动，那么，根据前文讨论，游戏结束时玩家 1 的收益将由

$$w_-=\max_{a_1\in A_1}\min_{a_2\in A_2}v(a_1,a_2)$$

给出．由于玩家 1 的行动对于玩家 2 单向透明，这应该是玩家 1 所能获得的最差结果．对称地，如果假定玩家 2 首先行动，那么，玩家 1 的收益将由

$$w_+=\min_{a_2\in A_2}\max_{a_1\in A_1}v(a_1,a_2)$$

给出．由于玩家 2 的行动对于玩家 1 单向透明，这应该是玩家 1 所能获得的最好结果．玩家 1 应该期待实际进行游戏时，所能获得的收益 $w\in[w_-,w_+]$．尽管不等式 $w_-\le w_+$ 总是成立（证明参见 弱对偶定理→9.15.1@p1774），但是由于等号未必成立，所以，仅采用序贯游戏的分析手段，一般情况下没有办法唯一确定游戏结果．

例子（续）

石头剪刀布游戏中，如果出手有先后，那么先手必输，后手必赢．转换为数学语言，这就是下列不等式：

$$w_-=-1\le+1=w_+.$$

此时，$w_-=w_+$ 并不成立．

上述分析过程遗漏了同时游戏的一个关键因素，就是玩家无法准确预测对手的行动．形式上，这意味着双方可以采取某种随机策略．这一想法在序贯游戏的语境下并不成立，因为无论先手玩家如何随机选择行动，后手玩家总能准确地观测到这一行动，并有针对性地回应．但是，对于同时游戏，随机策略引入的战略模糊将使得对手无法有效地针对己方的行动．

例子（续）

石头剪刀布游戏中，如果玩家 1 均匀随机地选择石头、剪刀、布三个行动之一，那么，根据玩家 2 的行动不同，玩家 1 可能获得的收益是

$$\frac{1}{3}(0,1,-1)^T+\frac{1}{3}(-1,0,1)^T+\frac{1}{3}(1,-1,0)^T=(0,0,0)^T.$$

此时，无论玩家 2 如何选择行动，玩家 1 的期望收益总是 0．这显然好于确定性地选择单个行动．

由此，就引入了混合策略的概念．

混合策略

同时游戏中，玩家 i 的 **混合策略**（mixed strategy），简称 **策略**，是指函数 $s_i:A_i\to[0,1]$，且它满足 $\sum_{a_i\in A_i}s_i(a_i)=1$．也就是说，策略 s_i 就是玩家 i 的行动集合 A_i 上的一个概率分布．玩家 i 全体混合策略的集合记作 $S_i=\Delta(A_i)$，其中，$\Delta(A_i)$ 表示 A_i 上的全体概率分布的集合．如果 s_i 是退化的概率分布，即存在 $a\in A_i$ 使得 $s_i(a)=1$，那么，也称策略 s_i 为 **纯策略**（pure strategy）．

混合策略的收益就是单个行动收益的期望：

$$v(s_1,s_2)=\sum_{a_1\in A_1}\sum_{a_2\in A_2}s_1(a_1)s_2(a_2)v(a_1,a_2).$$

将单个行动看作对应的纯策略，那么，就可以将行动集合 A_i 嵌入（混合）策略集合 S_i 中，且上式定义的 $v(s_1,s_2)$ 就可以看作是将 $v(a_1,a_2)$ 从 $A_1\times A_2$ 延拓到 $S_1\times S_2$ 上.

von Neumann 定理

引入混合策略后，极大化极小思想和极小化极大思想得到的结果是一致的，由此，同时零和游戏的结果也是唯一确定的.

定理（von Neumann）

允许混合策略的同时零和游戏中，如果双方都采取最优策略，那么，玩家 1 的最大收益为

$$w=\max_{s_1\in S_1}\min_{s_2\in S_2}v(s_1,s_2)=\min_{s_2\in S_2}\max_{s_1\in S_1}v(s_1,s_2),$$

玩家 2 的最大收益为 $-w$.

证明

设 $w=\max_{s_1\in S_1}\min_{s_2\in S_2}v(s_1,s_2)$. 考虑内层最小化问题，因为 $v(s_1,s_2)=\sum_{a_2\in A_2}s_2(a_2)v(s_1,a_2)$，所以，$\min_{s_2\in S_2}v(s_1,s_2)=\min_{a_2\in A_2}v(s_1,a_2)$，后者的最优解对应的纯策略就是前者的一个最优解. 因此，有 $w=\max_{s_1\in S_1}\min_{a_2\in A_2}v(s_1,a_2)$. 进而，引入辅助变量 u，问题就可以改写为

$$w=\max_{u,s_1}u\ \text{subject to}\ u\le\min_{a_2\in A_2}v(s_1,a_2).$$

这个约束等价于 $u\le v(s_1,a_2)$ 对于所有 $a_2\in A_2$ 都成立. 最后，引入混合策略 s_1 的定义和收益函数 $v(s_1,a_2)$ 的表达式，原问题就等价于 线性规划问题→9.15.1@p1774

$$(P)\quad\begin{aligned}w=&\max_{u,s_1}u\\ &\text{subject to}\sum_{a_1\in A_1}s_1(a_1)v(a_1,a_2)\ge u,\ \forall a_2\in A_2,\\ &\sum_{a_1\in A_1}s_1(a_1)=1,\\ &s_1(a_1)\ge 0,\ \forall a_1\in A_1.\end{aligned}$$

这个问题显然是可行的，且目标函数有界，所以存在最优解. 根据 对偶原理→9.15.1@p1774 可知，它的最优值就等于对偶问题的最优值：

$$(D)\quad\begin{aligned}w=&\min_{t,s_2}t\\ &\text{subject to}\sum_{a_2\in A_2}s_2(a_2)v(a_1,a_2)\le t,\ \forall a_1\in A_1,\\ &\sum_{a_2\in A_2}s_2(a_2)=1,\\ &s_2(a_2)\ge 0,\ \forall a_2\in A_2.\end{aligned}$$

重复前文的步骤，这一问题就等价于 $\min_{s_2\in S_2}\max_{s_1\in S_1}v(s_1,s_2)$. 定理得证.

双方的最优策略构成了这一游戏的 Nash 均衡[10]. 也就是说，假定双方都选择均衡中的最优策略，那么，没有任何玩家能够从单方面偏离均衡策略中严格获益.

转化为线性规划问题

von Neumann 定理的证明同时也指出了同时零和游戏的求解方法．设 n 和 m 分别是玩家 1 和 2 可采取的行动数目．给定玩家 1 的收益矩阵 $V \in \mathbf{R}^{n\times m}$，可以求解如下线性规划问题：

$$w = \max_{(u,s)\in\mathbf{R}\times\mathbf{R}^n} u$$

$$\text{subject to } V^T s \geq u\mathbf{1},$$

$$\mathbf{1}^T s = 1,$$

$$s \geq 0.$$

这是一个有 $n+1$ 个变量、$m+1$ 个约束（不计非负约束）的线性规划问题，可以用 单纯形法 →9.15.2@p1787 高效求解．由于 u 可能为负，可以先将 V 的所有元素都加上同一个常数 c 使其均为正，从而可以要求 $u \geq 0$；这不改变最优策略，求得的 w 减去 c 即为原游戏的值．算法得到的最优解 s 就是玩家 1 的最优（混合）策略．要求得玩家 2 的最优策略，只需要从单纯形表中获得该问题最优解的对偶变量（即影子价格）即可．

习题

- Luogu P4232 无意识之外的捉迷藏[11]

参考资料与注释

- Zero-sum game - Wikipedia[12]
- Minimax theorem - Wikipedia[13]

外部参考资料

[1] Codeforces 794 E. Choosing Carrot

[2] Luogu P2734 [USACO3.3] 游戏 A Game

[3] Luogu P4576 [CQOI2013] 棋盘游戏

[4] Luogu P7097 [yLOI2020] 牵丝戏

[5] Codeforces 388 C. Fox and Card Game

[6] Codeforces 794 E. Choosing Carrot

[7] Codeforces 1628 D2. Game on Sum (Hard Version)

[8] Luogu P3210 [HNOI2010] 取石头游戏

[9] 双矩阵游戏

[10] Nash 均衡

[11] Luogu P4232 无意识之外的捉迷藏

[12] Zero-sum game - Wikipedia

[13] Minimax theorem - Wikipedia

9.18.4 非公平组合游戏

9.19 数值算法

9.19.1 插值

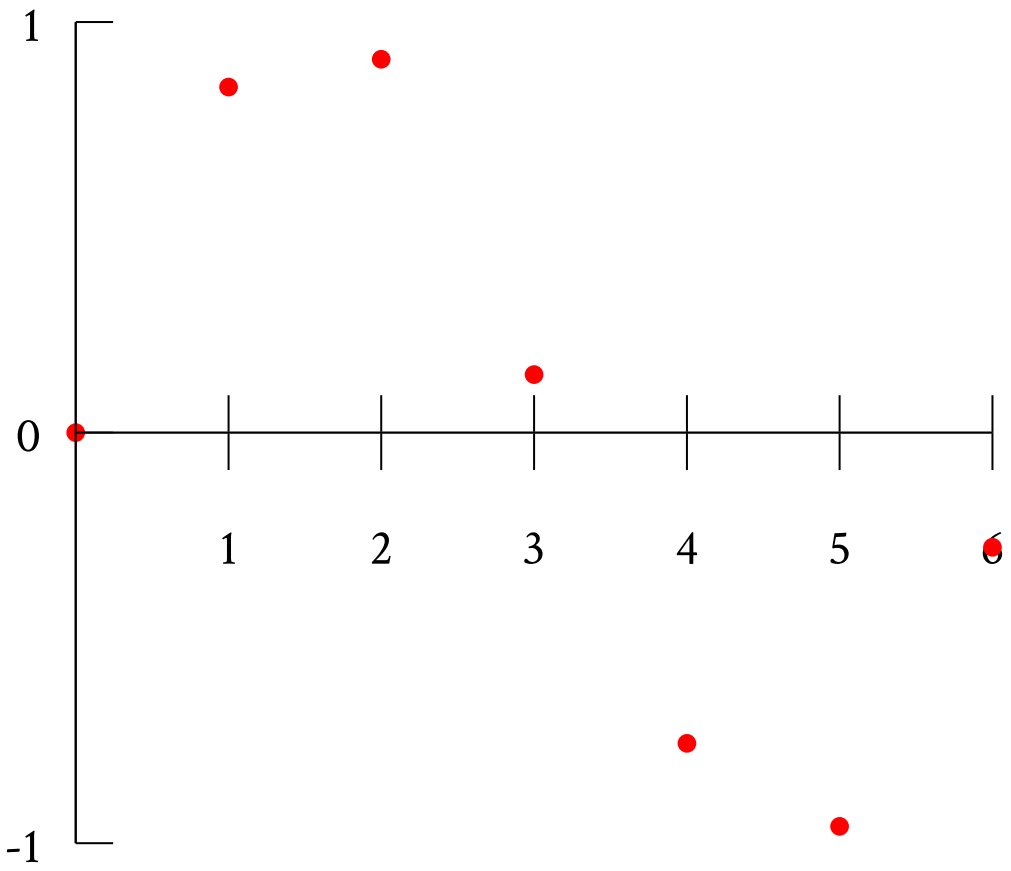
> **Authors:** AtomAlpaca, billchenchina, caibyte, Chrogeek, Early0v0, EndlessCheng, Enter-tainer, Henry-ZHR, hly1204, hsfzLZH1, Ir1d, Ghastlcon, kenlig, Marcythm, megakite, Peanut-Tang, qwqAutomaton, qz-cqy, StudyingFather, swift-zym, swiftqwq, Tiphereth-A, TrisolarisHD, Watersail2005, x4Cx58x54, Xeonacid, xiaopangfeiyu, YanWQ-monad

引入

插值是一种通过已知的、离散的数据点推算一定范围内的新数据点的方法．插值法常用于函数拟合中．

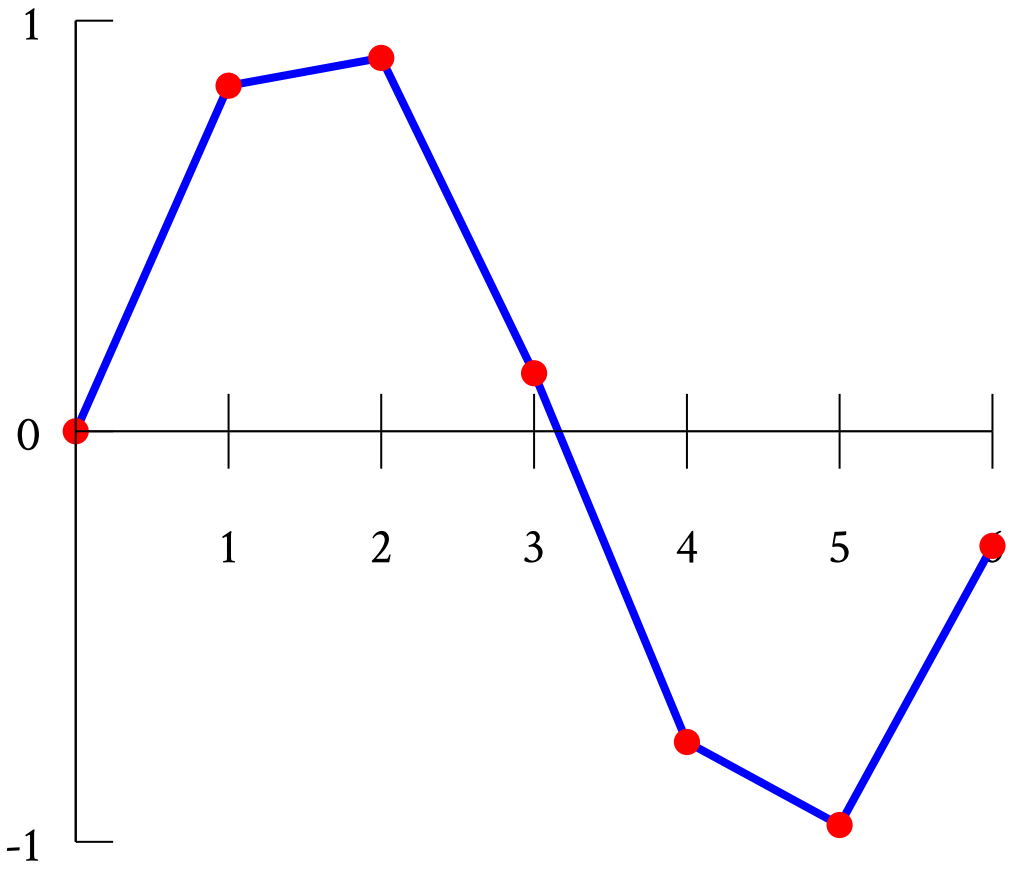
例如对数据点：

x	0	1	2	3	4	5	6
$f(x)$	0	0.8415	0.9093	0.1411	−0.7568	−0.9589	−0.2794



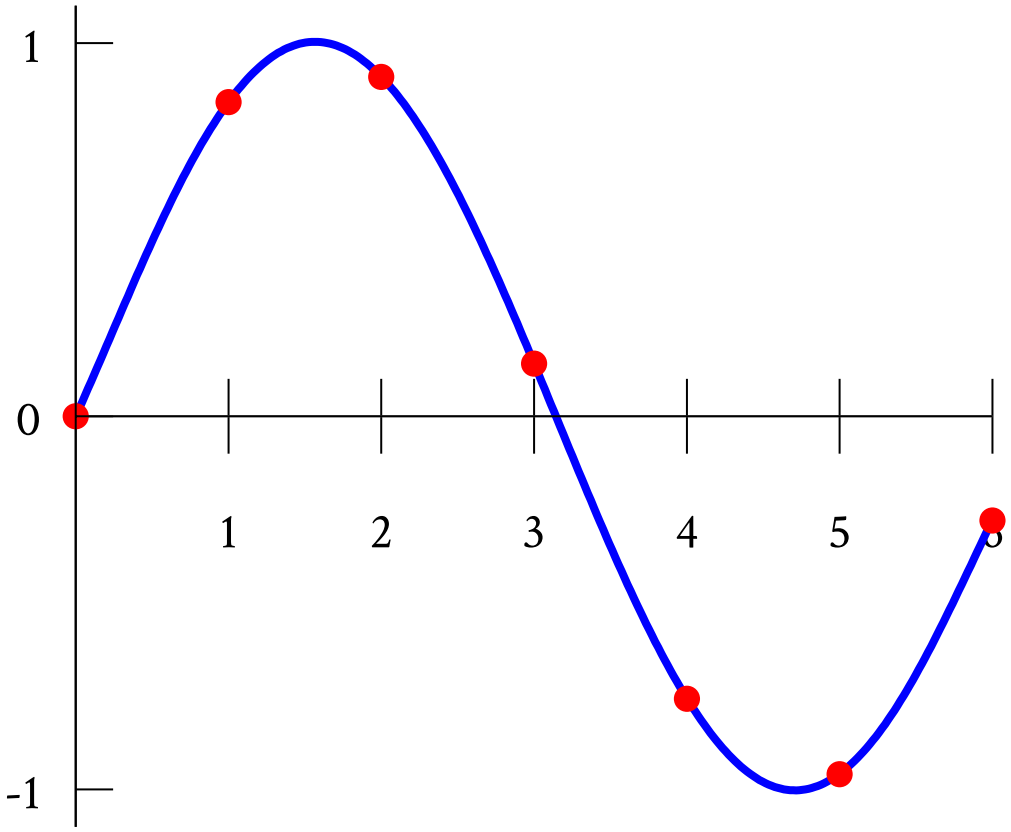
其中 $f(x)$ 未知，插值法可以通过按一定形式拟合 $f(x)$ 的方式估算未知的数据点.

例如，我们可以用分段线性函数拟合 $f(x)$：



这种插值方式叫做 线性插值[1].

我们也可以用多项式拟合 $f(x)$:



这种插值方式叫做 多项式插值[2].

多项式插值的一般形式如下:

多项式插值

对已知的 $n+1$ 的点 $(x_0,y_0),(x_1,y_1),...,(x_n,y_n)$，求形如 $f(x)=\sum_{i=0}^{n}a_ix^i$ 且满足

$$f(x_i)=y_i,\qquad \forall i=0,1,...,n$$

的多项式 $f(x)$.

下面介绍多项式插值中的两种方式：Lagrange 插值法与 Newton 插值法. 不难证明这两种方法得到的结果是相等的.

Lagrange 插值法

由于要求构造一个函数 $f(x)$ 过点 $P_1(x_1,y_1),P_2(x_2,y_2),\cdots,P_n(x_n,y_n)$. 首先设第 i 个点在 x 轴上的投影为 $P_i'(x_i,0)$.

考虑构造 n 个函数 $f_1(x),f_2(x),\cdots,f_n(x)$，使得对于第 i 个函数 $f_i(x)$，其图像过 $\begin{cases}P_j'(x_j,0),(j\neq i)\\P_i(x_i,y_i)\end{cases}$，则可知题目所求的函数 $f(x)=\sum\limits_{i=1}^{n}f_i(x)$.

那么可以设 $f_i(x)=a\cdot\prod_{j\neq i}(x-x_j)$，将点 $P_i(x_i,y_i)$ 代入可以知道 $a=\dfrac{y_i}{\prod_{j\neq i}(x_i-x_j)}$，所以

$$f_i(x)=y_i\cdot\frac{\prod_{j\neq i}(x-x_j)}{\prod_{j\neq i}(x_i-x_j)}=y_i\cdot\prod_{j\neq i}\frac{x-x_j}{x_i-x_j}$$

那么我们就可以得出 Lagrange 插值的形式为:

$$f(x)=\sum_{i=1}^{n}y_i\cdot\prod_{j\neq i}\frac{x-x_j}{x_i-x_j}$$

朴素实现的时间复杂度为 $O(n^2)$，可以优化到 $O(n\log^2 n)$，参见 多项式快速插值→9.12.8@p1558.

Luogu P4781【模板】拉格朗日插值[3]

给出 n 个点对 (x_i,y_i) 和 k，且 $\forall i,j$ 有 $i\neq j\iff x_i\neq x_j$ 且 $f(x_i)\equiv y_i\pmod{998244353}$ 和 $\deg(f(x))<n$（定义 $\deg(0)=-\infty$），求 $f(k)\bmod 998244353$.

题解

本题中只用求出 $f(k)$ 的值，所以在计算上式的过程中直接将 k 代入即可；有时候则需要进行多次求值等等更为复杂的操作，这时候需要求出 f 的各项系数．代码给出了一种求出系数的实现．

$$f(k)=\sum_{i=1}^{n}y_i\prod_{j\neq i}\frac{k-x_j}{x_i-x_j}$$

本题中，还需要求解逆元．如果先分别计算出分子和分母，再将分子乘进分母的逆元，累加进最后的答案，时间复杂度的瓶颈就不会在求逆元上，时间复杂度为 $O(n^2)$.

因为在固定模 998244353 意义下运算，计算乘法逆元的时间复杂度我们在这里暂且认为是常数时间.

代码实现

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

constexpr int MOD = 998244353;
using LL = long long;

int inv(int k) {
  int res = 1;
  for (int e = MOD - 2; e; e /= 2) {
    if (e & 1) res = (LL)res * k % MOD;
    k = (LL)k * k % MOD;
  }
  return res;
}

// 返回 f 满足 f(x_i) = y_i
// 不考虑乘法逆元的时间，显然为 O(n^2)
std::vector<int> lagrange_interpolation(const std::vector<int> &x,
                                        const std::vector<int> &y) {
  const int n = x.size();
  std::vector<int> M(n + 1), px(n, 1), f(n);
  M[0] = 1;
  // 求出 M(x) = prod_(i=0..n-1)(x - x_i)
  for (int i = 0; i < n; ++i) {
    for (int j = i; j >= 0; --j) {
      M[j + 1] = (M[j] + M[j + 1]) % MOD;
      M[j] = (LL)M[j] * (MOD - x[i]) % MOD;
```

```
    }
  }
  // 求出 px_i = prod_(j=0..n-1, j!=i) (x_i - x_j)
  for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j)
      if (i != j) {
        px[i] = (LL)px[i] * (x[i] - x[j] + MOD) % MOD;
      }
  }
  // 组合出 f(x) = sum_(i=0..n-1)(y_i / px_i)(M(x) / (x - x_i))
  for (int i = 0; i < n; ++i) {
    LL t = (LL)y[i] * inv(px[i]) % MOD, k = M[n];
    for (int j = n - 1; j >= 0; --j) {
      f[j] = (f[j] + k * t) % MOD;
      k = (M[j] + k * x[i]) % MOD;
    }
  }
  return f;
}

int main() {
  std::ios::sync_with_stdio(false);
  std::cin.tie(nullptr);
  int n, k;
  std::cin >> n >> k;
  std::vector<int> x(n), y(n);
  for (int i = 0; i < n; ++i) std::cin >> x[i] >> y[i];
  const auto f = lagrange_interpolation(x, y);
  int v = 0;
  for (int i = n - 1; i >= 0; --i) v = ((LL)v * k + f[i]) % MOD;
  std::cout << v << '\n';
  return 0;
}

```

横坐标是连续整数的 Lagrange 插值

如果已知点的横坐标是连续整数，我们可以做到 $O(n)$ 插值.

设要求的多项式为 $f(x)$，我们已知 $f(1),\cdots,f(n+1)$（$1\le i\le n+1$），考虑代入上面的插值公式：

$$f(x)=\sum_{i=1}^{n+1}y_i\prod_{j\neq i}\frac{x-x_j}{x_i-x_j}$$

$$=\sum_{i=1}^{n+1}y_i\prod_{j\neq i}\frac{x-j}{i-j}$$

后面的累乘可以分子分母分别考虑，不难得到分子为：

$$\frac{\prod_{j=1}^{n+1}(x-j)}{x-i}$$

分母的 $i-j$ 累乘可以拆成两段阶乘来算：

$$(-1)^{n+1-i}\cdot(i-1)!\cdot(n+1-i)!$$

于是横坐标为 $1,\cdots,n+1$ 的插值公式：

$$f(x)=\sum_{i=1}^{n+1}(-1)^{n+1-i}y_i\cdot\frac{\prod\limits_{j=1}^{n+1}(x-j)}{(i-1)!(n+1-i)!(x-i)}$$

预处理 $(x-i)$ 前后缀积、阶乘、阶乘逆，然后代入这个式子，复杂度为 $O(n)$.

例题 CF622F The Sum of the k-th Powers[4]

给出 n,k，求 $\sum\limits_{i=1}^{n}i^k$ 对 10^9+7 取模的值.

题解

本题中，答案是一个 $k+1$ 次多项式，因此我们可以线性筛出 $1^i,\cdots,(k+2)^i$ 的值然后进行 $O(n)$ 插值.

也可以通过组合数学相关知识由差分法的公式推得下式：

$$f(x)=\sum_{i=1}^{n+1}\binom{x-1}{i-1}\sum_{j=1}^{i}(-1)^{i+j}\binom{i-1}{j-1}y_j=\sum_{i=1}^{n+1}y_i\cdot\frac{\prod\limits_{j=1}^{n+1}(x-j)}{(x-i)\cdot(-1)^{n+1-i}\cdot(i-1)!\cdot(n+1-i)!}$$

代码实现

```cpp
// By: Luogu@rui_er(122461)
#include <iostream>
using namespace std;
constexpr int N = 1e6 + 5, mod = 1e9 + 7;

int n, k, tab[N], p[N], pcnt, f[N], pre[N], suf[N], fac[N], inv[N], ans;

int qpow(int x, int y) {
  int ans = 1;
  for (; y; y >>= 1, x = 1LL * x * x % mod)
    if (y & 1) ans = 1LL * ans * x % mod;
  return ans;
}

void sieve(int lim) {
  f[1] = 1;
  for (int i = 2; i <= lim; i++) {
    if (!tab[i]) {
      p[++pcnt] = i;
      f[i] = qpow(i, k);
    }
    for (int j = 1; j <= pcnt && 1LL * i * p[j] <= lim; j++) {
      tab[i * p[j]] = 1;
      f[i * p[j]] = 1LL * f[i] * f[p[j]] % mod;
      if (!(i % p[j])) break;
    }
  }
  for (int i = 2; i <= lim; i++) f[i] = (f[i - 1] + f[i]) % mod;
```

```cpp
}

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  cin >> n >> k;
  sieve(k + 2);
  if (n <= k + 2) return cout << f[n], 0;
  pre[0] = suf[k + 3] = 1;
  for (int i = 1; i <= k + 2; i++) pre[i] = 1LL * pre[i - 1] * (n - i) % mod;
  for (int i = k + 2; i >= 1; i--) suf[i] = 1LL * suf[i + 1] * (n - i) % mod;
  fac[0] = inv[0] = fac[1] = inv[1] = 1;
  for (int i = 2; i <= k + 2; i++) {
    fac[i] = 1LL * fac[i - 1] * i % mod;
    inv[i] = 1LL * (mod - mod / i) * inv[mod % i] % mod;
  }
  for (int i = 2; i <= k + 2; i++) inv[i] = 1LL * inv[i - 1] * inv[i] % mod;
  for (int i = 1; i <= k + 2; i++) {
    int P = 1LL * pre[i - 1] * suf[i + 1] % mod;
    int Q = 1LL * inv[i - 1] * inv[k + 2 - i] % mod;
    int mul = ((k + 2 - i) & 1) ? -1 : 1;
    ans = (ans + 1LL * (Q * mul + mod) % mod * P % mod * f[i] % mod) % mod;
  }
  cout << ans << '\n';
  return 0;
}

```

Newton 插值法

Newton 插值法是基于高阶差分来插值的方法，优点是支持 $O(n)$ 插入新数据点.

为了实现 $O(n)$ 插入新数据点，我们令：

$$f(x)=\sum_{j=0}^{n}a_jn_j(x)$$

其中 $n_j(x):=\prod_{i=0}^{j-1}(x-x_i)$ 称为 **Newton 基**（Newton basis）.

若解出 a_j，则可得到 $f(x)$ 的插值多项式. 我们按如下方式定义 **前向差商**（forward divided differences）：

$$[y_k]:=y_k,\qquad k=0,...,n,$$

$$[y_k,...,y_{k+j}]:=\frac{[y_{k+1},...,y_{k+j}]-[y_k,...,y_{k+j-1}]}{x_{k+j}-x_k},k=0,...,n-j,\ j=1,...,n.$$

则：

$$\begin{aligned}f(x)&=[y_0]+[y_0,y_1](x-x_0)+...+[y_0,...,y_n](x-x_0)...(x-x_{n-1})\\&=\sum_{j=0}^{n}[y_0,...,y_j]n_j(x)\end{aligned}$$

此即 Newton 插值的形式. 朴素实现的时间复杂度为 $O(n^2)$.

若样本点是等距的（即 $x_i = x_0 + ih$，$i = 1, \ldots, n$），我们可以推出

$$[y_k, \ldots, y_{k+j}] = \frac{1}{j!h^j}\Delta^{(j)}y_k,$$

其中 $\Delta^{(j)}y_k$ 为 **前向差分**（forward differences），定义如下：

$$\Delta^{(0)}y_k := y_k, \qquad k = 0, \ldots, n,$$

$$\Delta^{(j)}y_k := \Delta^{(j-1)}y_{k+1} - \Delta^{(j-1)}y_k, k = 0, \ldots, n-j,\ j = 1, \ldots, n.$$

令 $x = x_0 + sh$，则 Newton 插值的公式可化为

$$f(x) = \sum_{j=0}^{n}\binom{s}{j}j!h^j[y_0, \ldots, y_j] = \sum_{j=0}^{n}\binom{s}{j}\Delta^{(j)}y_0.$$

代码实现（Luogu P4781【模板】拉格朗日插值[5]）

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

constexpr uint32_t MOD = 998244353;

struct mint {
  uint32_t v_;

  mint() : v_(0) {}

  mint(int64_t v) {
    int64_t x = v % (int64_t)MOD;
    v_ = (uint32_t)(x + (x < 0 ? MOD : 0));
  }

  friend mint inv(mint const &x) {
    int64_t a = x.v_, b = MOD;
    if ((a %= b) == 0) return 0;
    int64_t s = b, m0 = 0;
    for (int64_t q = 0, _ = 0, m1 = 1; a;) {
      _ = s - a * (q = s / a);
      s = a;
      a = _;
      _ = m0 - m1 * q;
      m0 = m1;
      m1 = _;
    }
    return m0;
  }

  mint &operator+=(mint const &r) {
    if ((v_ += r.v_) >= MOD) v_ -= MOD;
    return *this;
  }

  mint &operator-=(mint const &r) {
    if ((v_ -= r.v_) >= MOD) v_ += MOD;
```

```
    return *this;
  }

  mint &operator*=(mint const &r) {
    v_ = (uint32_t)((uint64_t)v_ * r.v_ % MOD);
    return *this;
  }

  mint &operator/=(mint const &r) { return *this = *this * inv(r); }

  friend mint operator+(mint l, mint const &r) { return l += r; }

  friend mint operator-(mint l, mint const &r) { return l -= r; }

  friend mint operator*(mint l, mint const &r) { return l *= r; }

  friend mint operator/(mint l, mint const &r) { return l /= r; }
};

template <class T>
struct NewtonInterp {
  // {(x_0,y_0),...,(x_{n-1},y_{n-1})}
  vector<pair<T, T>> p;
  // dy[r][l] = [y_l,...,y_r]
  vector<vector<T>> dy;
  // (x-x_0)...(x-x_{n-1})
  vector<T> base;
  // [y_0]+...+[y_0,y_1,...,y_n](x-x_0)...(x-x_{n-1})
  vector<T> poly;

  void insert(T const &x, T const &y) {
    p.emplace_back(x, y);
    size_t n = p.size();
    if (n == 1) {
      base.push_back(1);
    } else {
      size_t m = base.size();
      base.push_back(0);
      for (size_t i = m; i; --i) base[i] = base[i - 1];
      base[0] = 0;
      for (size_t i = 0; i < m; ++i)
        base[i] = base[i] - p[n - 2].first * base[i + 1];
    }
    dy.emplace_back(p.size());
    dy[n - 1][n - 1] = y;
    if (n > 1) {
      for (size_t i = n - 2; ~i; --i)
        dy[n - 1][i] =
            (dy[n - 2][i] - dy[n - 1][i + 1]) / (p[i].first - p[n
- 1].first);
    }
    poly.push_back(0);
```

```cpp
    for (size_t i = 0; i < n; ++i) poly[i] = poly[i] + dy[n - 1][0] * base[i];
  }

  T eval(T const &x) {
    T ans{};
    for (auto it = poly.rbegin(); it != poly.rend(); ++it) ans = ans * x + *it;
    return ans;
  }
};

int main() {
  NewtonInterp<mint> ip;
  int n, k;
  cin >> n >> k;
  for (int i = 1, x, y; i <= n; ++i) {
    cin >> x >> y;
    ip.insert(x, y);
  }
  cout << ip.eval(k).v_;
  return 0;
}
```

横坐标是连续整数的 Newton 插值

例如：求多项式 $f(x)=\sum_{i=0}^{3}a_ix^i$ 的系数，已知 $f(1)$ 至 $f(6)$ 的值分别为 $1,5,14,30,55,91$.

$$\begin{array}{ccccccccccc} 1 & & 5 & & 14 & & 30 & & 55 & & 91 \\ & 4 & & 9 & & 16 & & 25 & & 36 & \\ & & 5 & & 7 & & 9 & & 11 & & \\ & & & 2 & & 2 & & 2 & & & \end{array}$$

第一行为 $f(x)$ 的连续的前 n 项；之后的每一行为之前一行中对应的相邻两项之差．观察到，如果这样操作的次数足够多（前提是 $f(x)$ 为多项式），最终总会返回一个定值.

计算出第 $i-1$ 阶差分的首项为 $\sum_{j=1}^{i}(-1)^{i+j}\binom{i-1}{j-1}f(j)$，第 $i-1$ 阶差分的首项对 $f(k)$ 的贡献为 $\binom{k-1}{i-1}$ 次.

$$f(k)=\sum_{i=1}^{n}\binom{k-1}{i-1}\sum_{j=1}^{i}(-1)^{i+j}\binom{i-1}{j-1}f(j)$$

时间复杂度为 $O(n^2)$.

C++ 中的实现

自 C++ 20 起，标准库添加了 `std::midpoint`[6] 和 `std::lerp`[7] 函数，分别用于求中点和线性插值.

习题

- 「NOIP2020」微信步数[8]
- 「联合省选 2022」填树[9]
- 「NOI2019」机器人[10]

参考资料

1. Interpolation - Wikipedia[11]

2. Newton polynomial - Wikipedia[12]

外部参考资料

[1] 线性插值 [2] 多项式插值 [3] Luogu P4781【模板】拉格朗日插值

[4] CF622F The Sum of the k-th Powers [5] Luogu P4781【模板】拉格朗日插值 [6] `std::midpoint`

[7] `std::lerp` [8] 「NOIP2020」微信步数 [9] 「联合省选 2022」填树

[10] 「NOI2019」机器人 [11] Interpolation - Wikipedia [12] Newton polynomial - Wikipedia

9.19.2 数值积分

> **Authors:** H-J-Granger, Chrogeek, countercurrent-time, Enter-tainer, Great-designer, iamtwz, Ir1d, ksyx, mao1t, Menci, NachtgeistW, Nanarikom, ShaoChenHeng, StudyingFather, SukkaW, Tiphereth-A, zyj-111

定积分的定义

简单来说，函数 $f(x)$ 在区间 $[l,r]$ 上的定积分 $\int_l^r f(x)\mathrm{d}x$ 指的是 $f(x)$ 在区间 $[l,r]$ 中与 x 轴围成的区域的面积（其中 x 轴上方的部分为正值，x 轴下方的部分为负值）.

很多情况下，我们需要高效，准确地求出一个积分的近似值. 下面介绍的 **辛普森法**，就是这样一种求数值积分的方法.

辛普森法

这个方法的思想是将被积区间分为若干小段，每段套用二次函数的积分公式进行计算.

二次函数积分公式（辛普森公式）

对于一个二次函数 $f(x)=ax^2+bx+c$，有：

$$\int_l^r f(x)\mathrm{d}x=\frac{(r-l)(f(l)+f(r)+4f(\frac{l+r}{2}))}{6}$$

推导过程： 对于一个二次函数 $f(x)=ax^2+bx+c$； 求积分可得 $F(x)=\int_0^x f(x)\mathrm{d}x=\frac{a}{3}x^3+\frac{b}{2}x^2+cx+D$ 在这里 D 是一个常数，那么

$$\begin{aligned}\int_l^r f(x)\mathrm{d}x&=F(r)-F(l)\\&=\frac{a}{3}(r^3-l^3)+\frac{b}{2}(r^2-l^2)+c(r-l)\\&=(r-l)(\frac{a}{3}(l^2+r^2+lr)+\frac{b}{2}(l+r)+c)\\&=\frac{r-l}{6}(2al^2+2ar^2+2alr+3bl+3br+6c)\\&=\frac{r-l}{6}((al^2+bl+c)+(ar^2+br+c)+4(a(\frac{l+r}{2})^2+b(\frac{l+r}{2})+c))\\&=\frac{r-l}{6}(f(l)+f(r)+4f(\frac{l+r}{2}))\end{aligned}$$

根据这个辛普森公式，我们先介绍一种普通的辛普森积分法.

普通辛普森法

1743 年，这种方法发表于托马斯·辛普森的一篇论文中.

描述

给定一个自然数 n，将区间 $[l,r]$ 分成 $2n$ 个等长的区间 x.

$x_i = l + ih,\ \ i = 0...2n, h = \frac{r-l}{2n}$.

我们就可以计算每个小区间 $[x_{2i-2}, x_{2i}]$，$i = 1...n$ 的积分值，将所有区间的积分值相加即为总积分.

对于 $[x_{2i-2}, x_{2i}]$，$i = 1...n$ 的一个区间，选其中的三个点 $(x_{2i-2}, x_{2i-1}, x_{2i})$ 就可以构成一条抛物线从而得到一个函数 $P(x)$，这个函数存在且唯一. 计算原函数在该区间的积分值就变成了计算新的二次函数 $P(x)$ 在该段区间的积分值. 这样我们就可以利用辛普森公式来近似计算它.

$\int_{x_{2i-2}}^{x_{2i}} f(x)\ dx \approx \int_{x_{2i-2}}^{x_{2i}} P(x)\ dx = (f(x_{2i-2}) + 4f(x_{2i-1}) + (f(x_{2i}))\frac{h}{3}$

将其分段求和即可得到如下结论:

$\int_l^r f(x)dx \approx (f(x_0) + 4f(x_1) + 2f(x_2) + 4f(x_3) + 2f(x_4) + ... + 4f(x_{2N-1}) + f(x_{2N}))\frac{h}{3}$

误差

我们直接给出结论，普通辛普森法的误差为:

$$-\frac{1}{90}\left(\frac{r-l}{2}\right)^5 f^{(4)}(\xi)$$

其中 ξ 是位于区间 $[l,r]$ 的某个值.

实现

- C++

```cpp
constexpr int N = 1000 * 1000;

double simpson_integration(double a, double b) {
  double h = (b - a) / N;
  double s = f(a) + f(b);
  for (int i = 1; i <= N - 1; ++i) {
    double x = a + h * i;
    s += f(x) * ((i & 1) ? 4 : 2);
  }
  s *= h / 3;
  return s;
}
```

- Python

```python
N = 1000 * 1000


def simpson_integration(a, b):
    h = (b - a) / N
    s = f(a) + f(b)
    for i in range(1, N):
        x = a + h * i
        if i & 1:
            s = s + f(x) * 4
        else:
            s = s + f(x) * 2
    s = s * (h / 3)
    return s
```

自适应辛普森法

普通的方法为保证精度在时间方面无疑会受到 n 的限制，我们应该找一种更加合适的方法.

现在唯一的问题就是如何进行分段. 如果段数少了计算误差就大，段数多了时间效率又会低. 我们需要找到一个准确度和效率的平衡点.

我们这样考虑：假如有一段图像已经很接近二次函数的话，直接带入公式求积分，得到的值精度就很高了，不需要再继续分割这一段了.

于是我们有了这样一种分割方法：每次判断当前段和二次函数的相似程度，如果足够相似的话就直接代入公式计算，否则将当前段分割成左右两段递归求解.

现在就剩下一个问题了：如果判断每一段和二次函数是否相似?

我们把当前段直接代入公式求积分，再将当前段从中点分割成两段，把这两段再直接代入公式求积分. 如果当前段的积分和分割成两段后的积分之和相差很小的话，就可以认为当前段和二次函数很相似了，不用再递归分割了.

上面就是自适应辛普森法的思想. 在分治判断的时候，除了判断精度是否正确，一般还要强制执行最少的迭代次数.

参考代码如下：

- C++

```cpp
double simpson(double l, double r) {
  double mid = (l + r) / 2;
  return (r - l) * (f(l) + 4 * f(mid) + f(r)) / 6;  // 辛普森公式
}

double asr(double l, double r, double eps, double ans, int step) {
  double mid = (l + r) / 2;
  double fl = simpson(l, mid), fr = simpson(mid, r);
  if (abs(fl + fr - ans) <= 15 * eps && step < 0)
    return fl + fr + (fl + fr - ans) / 15;  // 足够相似的话就直接返回
  return asr(l, mid, eps / 2, fl, step - 1) +
         asr(mid, r, eps / 2, fr, step - 1);  // 否则分割成两段递归求解
}

double calc(double l, double r, double eps) {
  return asr(l, r, eps, simpson(l, r), 12);
}
```

- Python

```python
def simpson(l, r):
    mid = (l + r) / 2
    return (r - l) * (f(l) + 4 * f(mid) + f(r)) / 6  # 辛普森公式


def asr(l, r, eps, ans, step):
    mid = (l + r) / 2
    fl = simpson(l, mid)
    fr = simpson(mid, r)
    if abs(fl + fr - ans) <= 15 * eps and step < 0:
        return fl + fr + (fl + fr - ans) / 15  # 足够相似的话就直接返回
    return asr(l, mid, eps / 2, fl, step - 1) + asr(
        mid, r, eps / 2, fr, step - 1
```

```
    )  # 否则分割成两段递归求解


def calc(l, r, eps):
    return asr(l, r, eps, simpson(l, r), 12)
```

习题

- Luogu4525【模板】自适应辛普森法 1[1]
- HDU1724 Ellipse[2]
- NOI2005 月下柠檬树[3]

参考资料

https://doi.org/10.1145/321526.321537[4]：该文章讨论了自适应 Simpson 法的改进方案，其中详细论述了上文代码中的常数 `15` 的由来与优势.

外部参考资料

[1] Luogu4525【模板】自适应辛普森法 1 [2] HDU1724 Ellipse [3] NOI2005 月下柠檬树

[4] https://doi.org/10.1145/321526.321537

9.19.3 高斯消元

> **Authors:** StudyingFather, CCXXXI, Chrogeek, ChungZH, countercurrent-time, Early0v0, Enter-tainer, GavinZhengOI, Great-designer, H-J-Granger, henrytbtrue, HeRaNO, huayucaiji, iamtwz, Ir1d, ksyx, MegaOwIer, NachtgeistW, P-Y-Y, qwqAutomaton, shuzhouliu, shuzhouliu-bot, Siger Young, sshwy, SukkaW, Tiphereth-A, tsentau, WhenMelancholy, Xeonacid, Yukimaikoriya, Zhoier, zyj-111, qute-firefly-26710-zjyjoe-lg-592080

引入

高斯消元法（Gauss–Jordan elimination）是求解线性方程组的经典算法，它在当代数学中有着重要的地位和价值，是线性代数课程教学的重要组成部分.

高斯消元法除了用于线性方程组求解外，还可以用于行列式计算、求矩阵的逆，以及其他计算机和工程方面.

消元法及高斯消元法思想

定义

消元法是将方程组中的一方程的未知数用含有另一未知数的代数式表示，并将其带入到另一方程中，这就消去了一未知数，得到一解；或将方程组中的一方程倍乘某个常数加到另外一方程中去，也可达到消去一未知数的目的. 消元法主要用于二元一次方程组的求解.

解释

例一：利用消元法求解二元一次线性方程组：

$$\begin{cases} 4x + y = 100 \\ x - y \ \ = 100 \end{cases}$$

解：将方程组中两方程相加，消元 y 可得：

$$5x = 200$$

解得：

$$x = 40$$

将 $x = 40$ 代入方程组中第二个方程可得：

$$y = -60$$

消元法理论的核心

消元法理论的核心主要如下：

- 两方程互换，解不变；
- 一方程乘以非零数 k，解不变；
- 一方程乘以数 k 加上另一方程，解不变.

高斯消元法思想概念

德国数学家高斯对消元法进行了思考分析，得出了如下结论：

- 在消元法中，参与计算和发生改变的是方程中各变量的系数；
- 各变量并未参与计算，且没有发生改变；
- 可以利用系数的位置表示变量，从而省略变量；
- 在计算中将变量简化省略，方程的解不变.

高斯在这些结论的基础上，提出了高斯消元法，首先将方程的增广矩阵利用行初等变换化为行最简形，然后以线性无关为准则对自由未知量赋值，最后列出表达方程组通解.

高斯消元五步骤法

解释

高斯消元法在将增广矩阵化为最简形后对于自由未知量的赋值, 需要掌握线性相关知识, 且赋值存在人工经验的因素, 使得在学习过程中有一定的困难，将高斯消元法划分为五步骤，从而提出五步骤法，内容如下：

1. 增广矩阵行初等行变换为行最简形；
2. 还原线性方程组；
3. 求解第一个变量；
4. 补充自由未知量；
5. 列表示方程组通解.

利用实例进一步说明该算法的运作情况.

过程

例二：利用高斯消元法五步骤法求解线性方程组：

$$\begin{cases} 2x_1 + 5x_3 + 6x_4 = 9 \\ x_3 + x_4 = -4 \\ 2x_3 + 2x_4 = -8 \end{cases}$$

增广矩阵行（初等）变换为行最简形

所谓增广矩阵，即为方程组系数矩阵 A 与常数列 b 的并生成的新矩阵，即 $(A \mid b)$，增广矩阵行初等变换化为行最简形, 即是利用了高斯消元法的思想理念, 省略了变量而用变量的系数位置表示变量，增广矩阵中用竖线隔开了系数矩阵和常数列，代表了等于符号.

$$\left(\begin{array}{cccc|c} 2 & 0 & 5 & 6 & 9 \\ 0 & 0 & 1 & 1 & -4 \\ 0 & 0 & 2 & 2 & -8 \end{array}\right)$$

$$\xrightarrow{r_3 - 2r_2} \left(\begin{array}{cccc|c} 2 & 0 & 5 & 6 & 9 \\ 0 & 0 & 1 & 1 & -4 \\ 0 & 0 & 0 & 0 & 0 \end{array}\right)$$

化为行阶梯形

$$\xrightarrow{\frac{r_1}{2}} \left(\begin{array}{cccc|c} 1 & 0 & 2.5 & 3 & 4.5 \\ 0 & 0 & 1 & 1 & -4 \\ 0 & 0 & 0 & 0 & 0 \end{array}\right)$$

$$\xrightarrow{r_1-r_2\times 2.5} \left(\begin{array}{cccc|c} 1 & 0 & 0 & 0.5 & 14.5 \\ 0 & 0 & 1 & 1 & -4 \\ 0 & 0 & 0 & 0 & 0 \end{array}\right)$$

化为最简形

还原线性方程组

$$\begin{cases} x_1 + 0.5x_4 = 14.5 \\ x_3 + x_4 \quad\ = -4 \end{cases}$$

解释

所谓的还原线性方程组，即是在行最简形的基础上，将之重新书写为线性方程组的形式，即将行最简形中各位置的系数重新赋予变量，中间的竖线还原为等号.

求解第一个变量

$$\begin{cases} x_1 = -0.5x_4 + 14.5 \\ x_3 = -x_4 - 4 \end{cases}$$

解释

即是对于所还原的线性方程组而言，将方程组中每个方程的第一个变量，用其他量表达出来. 如方程组两方程中的第一个变量 x_1 和 x_3.

补充自由未知量

$$\begin{cases} x_1 = -0.5x_4 + 14.5 \\ x_2 = x_2 \\ x_3 = -x_4 - 4 \\ x_4 = x_4 \end{cases}$$

解释

第 3 步中，求解出变量 x_1 和 x_3，从而说明了方程剩余的变量 x_2 和 x_4 不受方程组的约束，是自由未知量，可以取任意值，所以需要在第 3 步骤解得基础上进行解得补充，补充的方法为 $x_2 = x_2, x_4 = x_4$，这种解得补充方式符合自由未知量定义，并易于理解，因为是自由未知量而不受约束，所以只能自己等于自己.

列表示方程组的通解

$$\begin{aligned} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} &= \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} x_2 + \begin{pmatrix} -0.5 \\ 0 \\ -1 \\ 1 \end{pmatrix} x_4 + \begin{pmatrix} 14.5 \\ 0 \\ -4 \\ 0 \end{pmatrix} \\ &= \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} C_1 + \begin{pmatrix} -0.5 \\ 0 \\ -1 \\ 1 \end{pmatrix} C_2 + \begin{pmatrix} 14.5 \\ 0 \\ -4 \\ 0 \end{pmatrix} \end{aligned}$$

其中 C_1 和 C_2 为任意常数.

解释

即在第 4 步的基础上，将解表达为列向量组合的表示形式，同时由于 x_2 和 x_4 是自由未知量，可以取任意值，所以在解得右边，令二者分别为任意常数 C_1 和 C_2，即实现了对方程组的求解.

行列式计算

解释

$N \times N$ 方阵行列式（Determinant）可以理解为所有列向量所夹的几何体的有向体积.

例如:

$$\begin{vmatrix} 1 & 0 \\ 0 & 1 \end{vmatrix} = 1$$

$$\begin{vmatrix} 1 & 2 \\ 2 & 1 \end{vmatrix} = -3$$

行列式有公式

$$\det(A) = \sum_{\sigma \in S_n} \operatorname{sgn}(\sigma) \prod_{i=1}^{n} a_{i,\sigma(i)}$$

其中 S_n 是指长度为 n 的全排列的集合，σ 就是一个全排列，如果 σ 的逆序对对数为偶数，则 $\operatorname{sgn}(\sigma) = 1$，否则 $\operatorname{sgn}(\sigma) = -1$.

通过体积概念理解行列式不变性是一个非常简单的办法:

- 矩阵转置，行列式不变;
- 矩阵行（列）交换，行列式取反;
- 矩阵行（列）相加或相减，行列式不变;
- 矩阵行（列）所有元素同时乘以数 k，行列式等比例变大.

由此，对矩阵应用高斯消元之后，我们可以得到一个对角线矩阵，此矩阵的行列式由对角线元素之积所决定. 其符号可由交换行的数量来确定（如果为奇数，则行列式的符号应颠倒）. 因此，我们可以在 $O(n^3)$ 的复杂度下使用高斯算法计算矩阵.

注意，如果在某个时候，我们在当前列中找不到非零单元，则算法应停止并返回 0.

实现

```
constexpr double EPS = 1E-9;
int n;
vector<vector<double>> a(n, vector<double>(n));

double det = 1;
for (int i = 0; i < n; ++i) {
  int k = i;
  for (int j = i + 1; j < n; ++j)
    if (abs(a[j][i]) > abs(a[k][i])) k = j;
  if (abs(a[k][i]) < EPS) {
    det = 0;
    break;
  }
  swap(a[i], a[k]);
  if (i != k) det = -det;
  det *= a[i][i];
```

```
  for (int j = i + 1; j < n; ++j) a[i][j] /= a[i][i];
  for (int j = 0; j < n; ++j)
    if (j != i && abs(a[j][i]) > EPS)
      for (int k = i + 1; k < n; ++k) a[j][k] -= a[i][k] * a[j][i];
}

cout << det;
```

矩阵求逆

对于方阵 A，若存在方阵 A^{-1}，使得 $A \times A^{-1} = A^{-1} \times A = I$，则称矩阵 A 可逆，A^{-1} 被称为它的逆矩阵.

给出 n 阶方阵 A，求解其逆矩阵的方法如下：

1. 构造 $n \times 2n$ 的矩阵 (A, I_n)；
2. 用高斯消元法将其化简为最简形 (I_n, A^{-1})，即可得到 A 的逆矩阵 A^{-1}. 如果最终最简形的左半部分不是单位矩阵 I_n，则矩阵 A 不可逆.

该方法的正确性证明需要用到较多线性代数的知识，限于篇幅这里不再给出. 感兴趣的读者可以自行查阅相关资料.

高斯消元法解异或方程组

异或方程组是指形如

$$\begin{cases} a_{1,1}x_1 \oplus a_{1,2}x_2 \oplus \cdots \oplus a_{1,n}x_n & = b_1 \\ a_{2,1}x_1 \oplus a_{2,2}x_2 \oplus \cdots \oplus a_{2,n}x_n & = b_2 \\ \cdots & \cdots \\ a_{m,1}x_1 \oplus a_{m,2}x_2 \oplus \cdots \oplus a_{m,n}x_n & = b_m \end{cases}$$

的方程组，其中 $\oplus$ 表示「按位异或」（即 `xor` 或 C++ 中的 `^`），且式中所有系数/常数（即 $a_{i,j}$ 与 b_i）均为 0 或 1.

由于「异或」符合交换律与结合律，故可以按照高斯消元法逐步消元求解. 值得注意的是，我们在消元的时候应使用「异或消元」而非「加减消元」，且不需要进行乘除改变系数（因为系数均为 0 和 1）.

注意到异或方程组的增广矩阵是 01 矩阵（矩阵中仅含有 0 与 1），所以我们可以使用 C++ 中的 `std::bitset` 进行优化，将时间复杂度降为 $O(\frac{n^2m}{\omega})$，其中 n 为元的个数，m 为方程条数，ω 一般为 32（与机器有关）.

参考实现：

```
std::bitset<1010> matrix[2010];  // matrix[1~n]: 增广矩阵，0 位置为常数

std::vector<bool> GaussElimination(
    int n, int m)  // n 为未知数个数，m 为方程个数，返回方程组的解
                   //  （多解 / 无解返回一个空的 vector）
{
  for (int i = 1; i <= n; i++) {
    int cur = i;
    while (cur <= m && !matrix[cur].test(i)) cur++;
    if (cur > m) return std::vector<bool>(0);
    if (cur != i) swap(matrix[cur], matrix[i]);
    for (int j = 1; j <= m; j++)
      if (i != j && matrix[j].test(i)) matrix[j] ^= matrix[i];
  }
  std::vector<bool> ans(n + 1);
```

```
16    for (int i = 1; i <= n; i++) ans[i] = matrix[i].test(0);
17    return ans;
18  }
```

练习题

- Codeforces - 巫师和赌注[1]
- luogu - SDOI2010 外星千足虫[2]

外部参考资料

[1] Codeforces - 巫师和赌注  [2] luogu - SDOI2010 外星千足虫 

9.19.4 牛顿迭代法

> **Authors:** Marcythm, iamtwz, nutshellfool, sshwy, allenanswerzq, countercurrent-time, Enter-tainer, H-J-Granger, hly1204, Ir1d, Menci, NachtgeistW, SukkaW, Tiphereth-A, Xeonacid

引入

本文介绍如何用牛顿迭代法（Newton's method for finding roots）求方程的近似解，该方法于 17 世纪由牛顿提出.

具体的任务是，对于在 $[a,b]$ 上连续且单调的函数 $f(x)$，求方程 $f(x)=0$ 的近似解.

解释

初始时我们从给定的 $f(x)$ 和一个近似解 x_0 开始（初值的问题与 Newton 分形有关，可参考 3Blue1Brown 的 牛顿分形[1]）.

假设我们目前的近似解是 x_i，我们画出与 $f(x)$ 切于点 $(x_i,f(x_i))$ 的直线 l，将 l 与 x 轴的交点横坐标记为 x_{i+1}，那么这就是一个更优的近似解. 重复这个迭代的过程. 根据导数的几何意义，可以得到如下关系：

$$f'(x_i)=\frac{f(x_i)}{x_i-x_{i+1}}$$

整理后得到如下递推式：

$$x_{i+1}=x_i-\frac{f(x_i)}{f'(x_i)}$$

直观地说，如果 $f(x)$ 比较平滑，那么随着迭代次数的增加，x_i 会越来越逼近方程的解.

牛顿迭代法的收敛率是平方级别的，这意味着每次迭代后近似解的精确数位会翻倍. 关于牛顿迭代法的收敛性证明可参考 citizendium - Newton method Convergence analysis[2]

当然牛顿迭代法也同样存在着缺陷，详情参考 Xiaolin Wu - Roots of Equations 第 18 - 20 页分析[3]

求解平方根

我们尝试用牛顿迭代法求解平方根. 设 $f(x)=x^2-n$，这个方程的近似解就是 $\sqrt{n}$ 的近似值. 于是我们得到

$$x_{i+1}=x_i-\frac{x_i^2-n}{2x_i}=\frac{x_i+\frac{n}{x_i}}{2}$$

在实现的时候注意设置合适的精度. 代码如下

实现

- C++

```cpp
double sqrt_newton(double n) {
  constexpr static double eps = 1E-15;
  double x = 1;
  while (true) {
    double nx = (x + n / x) / 2;
    if (abs(x - nx) < eps) break;
    x = nx;
  }
  return x;
}
```

- Python

```python
def sqrt_newton(n):
    eps = 1e-15
    x = 1
    while True:
        nx = (x + n / x) / 2
        if abs(x - nx) < eps:
            break
        x = nx
    return x
```

求解整数平方根

尽管我们可以调用 `sqrt()` 函数来获取平方根的值，但这里还是讲一下牛顿迭代法的变种算法，用于求不等式 $x^2 \leq n$ 的最大整数解．我们仍然考虑一个类似于牛顿迭代的过程，但需要在边界条件上稍作修改．如果 x 在迭代的过程中上一次迭代值得近似解变小，而这一次迭代使得近似解变大，那么我们就不进行这次迭代，退出循环．

实现

- C++

```cpp
int isqrt_newton(int n) {
  int x = 1;
  bool decreased = false;
  for (;;) {
    int nx = (x + n / x) >> 1;
    if (x == nx || (nx > x && decreased)) break;
    decreased = nx < x;
    x = nx;
  }
  return x;
}
```

- Python

```python
def isqrt_newton(n):
    x = 1
    decreased = False
    while True:
        nx = (x + n // x) // 2
        if x == nx or (nx > x and decreased):
```

```
            break
        decreased = nx < x
        x = nx
    return x
```

高精度平方根

最后考虑高精度的牛顿迭代法. 迭代的方法是不变的，但这次我们需要关注初始时近似解的设置，即 x_0 的值. 由于需要应用高精度的数一般都非常大，因此不同的初始值对于算法效率的影响也很大. 一个自然的想法就是考虑 $x_0 = 2^{\lfloor \frac{1}{2} \log_2 n \rfloor}$，这样既可以快速计算出 x_0，又可以较为接近平方根的近似解.

实现

给出 Java 代码的实现：

```
public static BigInteger isqrtNewton(BigInteger n) {
  BigInteger a = BigInteger.ONE.shiftLeft(n.bitLength() / 2);
  boolean p_dec = false;
  for (;;) {
    BigInteger b = n.divide(a).add(a).shiftRight(1);
    if (a.compareTo(b) == 0 || a.compareTo(b) < 0 && p_dec)
      break;
    p_dec = a.compareTo(b) > 0;
    a = b;
  }
  return a;
}
```

实践效果：在 $n = 10^{1000}$ 的时候该算法的运行时间是 60 ms，如果我们不优化 x_0 的值，直接从 $x_0 = 1$ 开始迭代，那么运行时间将增加到 120 ms.

习题

- UVa 10428 - The Roots[4]
- LeetCode 69. x 的平方根[5]

本页面主要译自博文 Метод Ньютона (касательных) для поиска корней[6] 与其英文翻译版 Newton's method for finding roots[7]. 其中俄文版版权协议为 Public Domain + Leave a Link；英文版版权协议为 CC-BY-SA 4.0.

外部参考资料

[1] 牛顿分形

[2] citizendium - Newton method Convergence analysis

[3] Xiaolin Wu - Roots of Equations 第 18 - 20 页分析

[4] UVa 10428 - The Roots

[5] LeetCode 69. x 的平方根

[6] Метод Ньютона (касательных) для поиска корней

[7] Newton's method for finding roots

9.20 序理论

引入

序理论是利用二元关系来将「次序」这一概念严格化的数学分支，下面将介绍这一分支的基本定义.

定义

二元关系

定义

集合 X 和集合 Y 上的一个 **二元关系**（binary relation）R 定义为元组 $(X, Y, G(R))$，其中 X 称为定义域（domain），Y 称为陪域（codomain），$G(R) \subseteq X \times Y = \{(x, y) : x \in X, y \in Y\}$ 称为二元关系 R 的图（graph）. xRy 成立当且仅当 $(x, y) \in G(R)$.

若 $X = Y$，则称该二元关系为齐次二元关系（homogeneous relation）或内关系（endorelation）.

若没有特别说明，下文中的二元关系均为齐次二元关系.

例如 $\mathbf{N}_+$ 上的整除 $|$ 和小于等于 $\leq$ 均为二元关系.

我们研究二元关系时，往往会关注其是否具有一些特别的性质. 对集合 S 上的二元关系 R，我们定义如下特殊性质：

1. 自反性（reflexive）：$(\forall\ a \in S)\ \ aRa$,
2. 反自反性（irreflexive，anti-reflexive）：$(\forall\ a \in S)\ \ \neg(aRa)$,
3. 对称性（symmetric）：$(\forall\ a, b \in S)\ \ aRb \iff bRa$,
4. 反对称性（antisymmetric）：$(\forall\ a, b \in S)\ \ (aRb \land bRa) \implies a = b$,
5. 非对称性（asymmetric）：$(\forall\ a, b \in S)\ \ aRb \implies \neg(bRa)$,
6. 传递性（transitive）：$(\forall\ a, b, c \in S)\ \ (aRb \land bRc) \implies aRc$,
7. 连接性（connected）：$(\forall\ a, b \in S)\ \ a \neq b \implies (aRb \lor bRa)$,
8. 良基性（well-founded）：$(\exists\ m \in S \neq \emptyset)\ \ (\forall\ a \in S \setminus \{m\})\ \ \neg(aRm)$（即非空集合 S 中有极小元 m）,
9. 不可比的传递性（transitive of incomparability）：$(\forall\ a, b, c \in S)\ \ (\neg(aRb \lor bRa) \land \neg(bRc \lor cRb)) \implies \neg(aRc \lor cRa)$（若 $\neg(aRb \lor bRa)$，则称 a 和 b 是不可比的）.

同时我们定义一些特殊的二元关系：

二元关系	自反性	反自反性	对称性	反对称性	非对称性	传递性	连接性	良基性	不可比的传递性
等价关系（equivalence relation）	有		有			有			
预序（preorder, quasiorder）	有					有			
偏序（partial order）	有			有		有			
全序（total order）	有			有		有	有		
良序（well-order）	有			有		有	有	有	
严格预序（strict preorder）		有				有			
严格偏序（strict partial order）		有			有	有			

严格弱序（strict weak order）		有			有	有			有
严格全序（strict total order）		有			有	有	有		

关系间的运算

对集合 X 和集合 Y 上的二元关系 R 和 S，我们可以定义如下运算：

1. R 和 S 的并 $R \cup S$ 满足 $G(R \cup S) := \{(x, y) : xRy \vee xSy\}$（如 $\leq$ 是 $<$ 和 $=$ 的并），
2. R 和 S 的交 $R \cap S$ 满足 $G(R \cap S) := \{(x, y) : xRy \wedge xSy\}$，
3. R 的补 $\bar{R}$ 满足 $G(\bar{R}) := \{(x, y) : \neg(xRy)\}$，
4. R 的对偶 R^T 满足 $G(R^T) := \{(y, x) : xRy\}$.

对集合 X 和集合 Y 上的二元关系 R 以及集合 Y 和集合 Z 上的二元关系 S，我们可以定义其复合 $S \circ R$ 满足 $G(S \circ R) := \{(x, z) : (\exists\ y \in Y)\ \ xRy \wedge ySz\}$.

偏序集

定义

若集合 S 上的一个二元关系 $\preceq$ 具有 **自反性**、**反对称性**、**传递性**，则称 S 是 **偏序集**（partially ordered set，poset），$\preceq$ 为其上一 **偏序**（partial order）.

若偏序 $\preceq$ 还具有 **连接性**，则称其为 **全序**（total order），对应的集合称为 **全序集**（totally ordered set）、**线性序集**（linearly ordered set，loset）、**简单序集**（simply ordered set）.

不难发现 $\mathbf{N}$，$\mathbf{Z}$，$\mathbf{Q}$、$\mathbf{R}$ 均关于 $\leq$ 构成全序集.

偏序集的可视化表示：Hasse 图

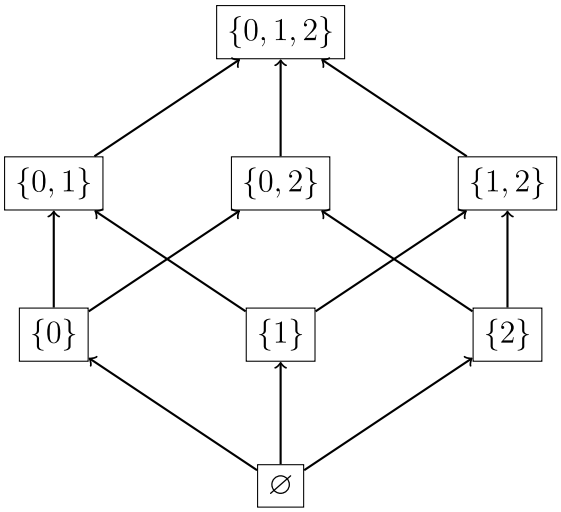
对于有限偏序集，我们可以用 Hasse 图直观地表示其上的偏序关系.

定义

对有限偏序集 S 和其上的偏序 $\preceq$，定义 $x \prec y \iff (x \preceq y \wedge x \neq y)$ 其对应的 **Hasse 图** 为满足如下条件的图 $G = \langle V, E \rangle$：

- $V = S$,
- $E = \{(x, y) \in S \times S : x \prec y \wedge ((\nexists\ z \in S)\ \ x \prec z \prec y)\}$

如对于集合 $\{0, 1, 2\}$ 的幂集 S 和集合的包含关系 $\subseteq$，其对应的 Hasse 图为：



由于偏序具有反对称性，所以 Hasse 图一定是 有向无环图→11.7@p2727，进而我们可以根据 拓扑排序→11.8@p2729 对任意有限偏序集构造全序.

链与反链

定义

对偏序集 S 和其上的偏序 $\preceq$，称 S 的全序子集为 **链**（chain）. 若 S 的子集 T 中任意两个不同元素均不可比（即 $(\forall\ a, b \in T)\ \ a \neq b \implies (a \not\preceq b \wedge b \not\preceq a)$），则称 T 为 **反链**（antichain）.

对偏序集 S 和其上的偏序 $\preceq$，我们将偏序集 S 的最长反链长度称为 **宽度**（partial order width）.

如对于集合 $\{0, 1, 2\}$ 的幂集 S 和集合的包含关系 $\subseteq$，$\{\emptyset, \{1\}, \{1, 2\}\}$ 为一条链，$\{\{1\}, \{0, 2\}\}$ 为一条反链，S 的宽度为 3.

预序集中的特殊元素

在预序集中，我们可以定义极大（小）元、上（下）界、上（下）确界等概念，这些概念可以推广到其他序关系中.

定义

对预序集 S 和其上的预序 $\preceq$，取 S 中的元素 m：

1. 若 $(\forall\ a \in S \setminus \{m\})\ \ \neg(m \preceq a)$，则称 m 为 **极大元**（maximal element），
2. 若对 $T \subseteq S$ 满足 $(\forall\ t \in T)\ \ t \preceq m$，则称 m 为 T 的 **上界**（upper bound），
3. 若对 $T \subseteq S$ 满足 m 是 T 的上界且对 T 的任意上界 n 均有 $m \preceq n$，则称 m 为 T 的 **上确界**（supremum）.

类似可定义 **极小元**（minimal element）、**下界**（lower bound）和 **下确界**（infimum）.

如 1 是 $\mathbf{N}_+$ 的极小元和下界.

可以证明：

- 预序集中，极大（小）元、上（下）界、上（下）确界都是不一定存在的，即使存在也不一定唯一.
- 若偏序集 S 的子集 T 存在上（下）确界，则一定唯一.

  我们可将 T 的上确界、下确界分别记为 $\sup T$，$\inf T$. 若偏序集 S 既有上界又有下界，则称 S 是有界的.

在无限偏序集中，极大元不一定存在. 可用 **Zorn 引理**（Zorn's Lemma）来判断无限偏序集中是否存在极大元.

Zorn 引理[1]

Zorn 引理 也被称为 **Kuratowski–Zorn 引理**，其内容为：若非空偏序集的每条链都有上界，则该偏序集存在极大元.

Zorn 引理与 **选择公理**[2]、**良序定理**[3] 等价.

有向集与格

我们知道若偏序集的子集存在上（下）确界，则一定唯一. 但是这一点并不适用于极大（小）元. 例如：考虑偏序集 $S=\{\{0\},\{1\},\{2\},\{0,1\},\{0,2\},\{1,2\}\}$ 和其上的偏序 $\subseteq$，不难发现其有 3 个极大元和 3 个极小元.

我们希望通过向偏序集添加一定的条件来使得若极大（小）元存在则一定唯一，这样我们就可以定义最大（小）元的概念了.

有向集

对预序集 S 和其上的预序 $\preceq$，若 $(\forall\ a,b\in S)\ (\exists\ c\in S)\ \ a\preceq c\wedge b\preceq c$，则称 $\preceq$ 为 S 的一个 **方向**（direction），S 称为 **有向集**（directed set）或 **过滤集**（filtered set）.

有时也将满足上述定义的集合 S 称为 **上有向集**（upward directed set），类似地可定义 **下有向集**（downward directed set）.

有向集也可用如下方式定义：

有向集的等价定义

对预序集 S 和其上的预序 $\preceq$，若 S 的任意有限子集 T 均有上界，则称 $\preceq$ 为 S 的一个方向，S 称为有向集.

不难发现：

- 若上有向集存在极大元，则一定唯一. 我们将上有向集的极大元称为 **最大元**（greatest element）.
- 若下有向集存在极小元，则一定唯一. 我们将下有向集的极小元称为 **最小元**（least element）.

有方向的偏序集中，对任意元素 a,b，$\{a,b\}$ 都有上界，若将上界修改为上确界，则得到了并半格的定义.

对偏序集 S 和其上的偏序 $\preceq$:

并半格

若对 S 中的任意元素 a,b，$\{a,b\}$ 均有上确界 c，则称 S 为 **并半格**（join-semilattice，upper semilattice），并且我们称 c 为 a 和 b 的 **并**（join），记为 $a\vee b$.

交半格

若对 S 中的任意元素 a,b，$\{a,b\}$ 均有下确界 c，则称 S 为 **交半格**（meet-semilattice，lower semilattice），并且我们称 c 为 a 和 b 的 **交**（meet），记为 $a\wedge b$.

格

若 S 既是并半格也是交半格，则称 S 为 **格**（lattice）.

例如 60 的正因子构成的集合 $S=\{1,2,3,4,5,6,10,12,15,20,30,60\}$ 关于整除构成偏序集，其上的任意正整数 a,b，$\operatorname{lcm}(a,b)$ 为 a 和 b 的并，$\gcd(a,b)$ 为 a 和 b 的交，从而 S 是格.

对偶

在序理论中，对偶是非常常见的概念，如上文提到的极大元与极小元对偶、上界与下界对偶、上确界与下确界对偶.

对偏序集 P 和其上的偏序 $\preceq$，定义其 **对偶**（dual，opposite）偏序集 P^d 满足：$x \preceq y$ 在 P 中成立当且仅当 $y \preceq x$ 在 P^d 中成立. 将 P 的 Hasse 图的边反转即可得到 P^d 的 Hasse 图.

Dilworth 定理与 Mirsky 定理

对有限偏序集 S 和其上的偏序 $\preceq$，我们有如下的一对对偶的定理：

Dilworth 定理

S 的宽度（最长反链长度）等于最小的链覆盖数.

证明

考虑数学归纳法. 当 $|S| \leq 3$ 时，命题显然成立.

假设命题对所有元素个数小于 $|S|$ 的偏序集都成立，令 S 的宽度为 d. 若 $|S|$ 中所有元素均不可比，则命题显然成立，否则在 S 中取一条长度大于 1 的链，令其中的最小元为 m，最大元为 M.

令 $T = S \setminus \{m, M\}$，若 T 中的宽度不超过 $d-1$，则由归纳假设知 T 可被至多 $d-1$ 条链覆盖，进而 S 可被这些链再加上链 $\{m, M\}$ 覆盖，命题成立，否则说明 T 中的宽度也为 d，令 T 中最长的一条反链为 A.

我们考虑如下两个集合：

$$S^+ := \{x \in S : (\exists\ a \in A)\ \ a \preceq x\}$$

$$S^- := \{x \in S : (\exists\ a \in A)\ \ x \preceq a\}$$

我们不难发现如下性质：

- $S^+ \cup S^- = S$,
- $S^+ \cap S^- = A$,
- $|S^+| < |S|, |S^-| < |S|$（因为 $m \notin S^+$ 且 $M \notin S^-$）.

对 S^+ 和 S^- 都应用归纳假设，则这两个集合的最小链覆盖数为 d，且这些链中恰好包含一个 A 中的元素 a，设这些链分别为 C_a^+，C_a^-，则 $\{C_a^- \cup \{a\} \cup C_a^+\}_{a \in A}$ 是 S 的一个最小链覆盖，命题得证.

Mirsky 定理

S 的最长链长度等于最小的反链覆盖数.

证明

设 S 的最长链长度为 d，则由定义，最小反链覆盖数至少为 d.

令 $f(s)$ 为以 s 为最小元的最长链长度，注意到若 $f(s) = f(t)$，则 s 与 t 不可比，进而 $(\forall\ n \in \mathbf{N})$ $f^{-1}(\{n\})$ 均为反链，其中 $f^{-1}(\{n\}) := \{a \in S : f(a) = n\}$ 称为 水平集（level set）[4].

因此不难得出 $\{f^{-1}(\{i\}) : 1 \leq i \leq d\}$ 是一个反链覆盖，从而最小反链覆盖数至多为 d.

Dilworth 定理与 Hall 婚配定理→11.24.1@p2949 等价.

我们可以用 Dilworth 定理证明如下定理：

Erdős–Szekeres 定理

含至少 $rs+1$ 个元素的实数序列 $\{a_i\}$ 要么有一个长为 $r+1$ 的不下降子序列，要么有一个长为 $s+1$ 的不上升子序列.

证明

设序列长度为 $n \geq rs+1$，定义偏序集 $\{(i,a_i)\}_{i=1}^{n}$，其上的偏序 $\preceq$ 定义为：

$$(i,a_i) \preceq (j,a_j) \iff (i \leq j \wedge a_i \leq a_j)$$

假设该偏序集的宽度不超过 s，则由 Dilworth 定理可知该偏序集可以被至多 s 条链覆盖，若这些链的长度都不超过 r，则序列所含元素数至多为 rs，与条件矛盾.

例题

Luogu P1020 [NOIP1999 提高组] 导弹拦截[5]

某国为了防御敌国的导弹袭击，发展出一种导弹拦截系统. 但是这种导弹拦截系统有一个缺陷：虽然它的第一发炮弹能够到达任意的高度，但是以后每一发炮弹都不能高于前一发的高度. 某天，雷达捕捉到敌国的导弹来袭. 由于该系统还在试用阶段，所以只有一套系统，因此有可能不能拦截所有的导弹.

输入导弹依次飞来的高度，计算这套系统最多能拦截多少导弹，如果要拦截所有导弹最少要配备多少套这种导弹拦截系统.

对于全部数据，满足导弹的高度为正整数，且不超过 5×10^4.

题解

令一共有 n 个导弹，第 i 个导弹的高度为 h_i，则集合 $\{(i,h_i)\}_{i=1}^{n}$ 为偏序集，其上的偏序 $\preceq$ 定义为：

$$(i,h_i) \preceq (j,h_j) \iff (i \leq j \wedge h_i \geq h_j)$$

进而根据 Dilworth 定理有：**序列的不上升子序列的最少覆盖数等于最长上升子序列长度**. 从而可以通过 最长不下降子序列的 $O(n \log n)$ 做法→7.2@p714 解决本题.

参考代码

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

int main() {
  vector<int> a;
  int x;
  while (cin >> x) a.push_back(x);
  vector<int> f, g;
  for (int i : a) {
    if (f.empty() || -i >= f.back())
      f.push_back(-i);
```

```
    else
      *upper_bound(f.begin(), f.end(), -i) = -i;
    if (g.empty() || i > g.back())
      g.push_back(i);
    else
      *lower_bound(g.begin(), g.end(), i) = i;
  }
  cout << f.size() << '\n' << g.size() << '\n';
  return 0;
}
```

[TJOI2015] 组合数学[6]

给一个 n 行 m 列的网格图，其中每个格子中均有若干块财宝．每次从左上角出发，只能往右或下走，每次经过一个格子至多只能捡走一块财宝．问至少要走几次才可能把财宝全捡完．

$1 \leq n \leq 1000$，$1 \leq m \leq 1000$，每个格子中的财宝不超过 10^6 块．

题解

不考虑网格图的点权，不难发现按给定的规则下在网格图上行走等价于在 DAG 上行走，从而我们可以将其视作 Hasse 图来构造偏序集，进而根据 Dilworth 定理有：**DAG 的最小链覆盖数等于最大的点独立集大小**.

因此本题所求即为给定网格图最大点权独立集的点权和.

令 a_{ij} 为网格图在点 (i,j) 处的权值，$f(i,j)$ 为从 (i,j) 到 $(1,m)$ 这个子网格中的答案，注意到每个点都和其右上角的点不相邻，则状态转移方程为：

$$f(i,j) = \max\{f(i-1,j), f(i,j+1), f(i-1,j+1) + a_{ij}\}$$

答案即为 $f(n,1)$.

参考代码

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

int main() {
  int t = 0;
  cin >> t;
  while (t--) {
    int n, m;
    cin >> n >> m;
    vector<vector<int64_t>> a(n, vector<int64_t>(m));
    for (auto &i : a)
      for (auto &j : i) cin >> j;
    vector<vector<int64_t>> f(n, vector<int64_t>(m));
```

```
    for (int i = 0; i < n; ++i)
      for (int j = m - 1; j >= 0; --j)
        f[i][j] =
            max({(i == 0 ? 0 : f[i - 1][j]), (j == m - 1 ? 0 : f[i][j
+ 1]),
                 (i == 0 || j == m - 1 ? 0 : f[i - 1][j + 1]) +
a[i][j]});
    cout << f[n - 1][0] << '\n';
  }
  return 0;
}
```

习题

- [CTSC2008] 祭祀[7]
- CodeForces 590E Birthday[8]

C++ 中的应用

另请参阅：排序相关 STL - 算法基础→5.7.14@p613.

C++ STL 中需要使用比较的算法和数据结构[9]中有序理论的应用．我们经常需要在 C++ 中自定义比较器，STL 要求[10]其必须为 **严格弱序**．令 $<$ 为自定义比较器，则可以定义：

- $x > y$ 为 $y < x$；
- $x \leq y$ 为 $y \not< x$；
- $x \geq y$ 为 $x \not< y$；
- $x = y$ 为 $x \not< y \wedge y \not< x$.

参考资料与拓展阅读

1. Order theory - From Academic Kids[11]
2. Binary Relation - Wikipedia[12]
3. Order Theory - Wikipedia[13]
4. Hasse diagram - Wikipedia[14]
5. Directed set - Wikipedia[15]
6. Order Theory, Lecture Notes by Mark Dean for Decision Theory[16]
7. 卢开澄，卢华明，《组合数学》（第 3 版）[17], 2006
8. List of Order Theory Topics - Wikipedia[18]
9. 浅谈邻项交换排序的应用以及需要注意的问题 by ouuan[19]
10. One thing you should know about comparators—Strict Weak Ordering[20]
11. Dilworth's theorem - Wikipedia[21]
12. Dilworth's Theorem | Brilliant Math & Science Wiki[22]
13. Hall's marriage theorem - Wikipedia[23]
14. Hall's Marriage Theorem | Brilliant Math & Science Wiki[24]
15. Dilworth 学习笔记 - Selfish[25]

外部参考资料

[1] Zorn 引理

[2] 选择公理

[3] 良序定理

[4] 水平集（level set）

[5] Luogu P1020 [NOIP1999 提高组] 导弹拦截

[6] [TJOI2015] 组合数学

[7] [CTSC2008] 祭祀

[8] CodeForces 590E Birthday

[9] 需要使用比较的算法和数据结构

[10] 要求

[11] Order theory - From Academic Kids

[12] Binary Relation - Wikipedia

[13] Order Theory - Wikipedia

[14] Hasse diagram - Wikipedia

[15] Directed set - Wikipedia

[16] Order Theory, Lecture Notes by Mark Dean for Decision Theory

[17] 《组合数学》（第 3 版）

[18] List of Order Theory Topics - Wikipedia

[19] 浅谈邻项交换排序的应用以及需要注意的问题 by ouuan

[20] One thing you should know about comparators—Strict Weak Ordering

[21] Dilworth's theorem - Wikipedia

[22] Dilworth's Theorem | Brilliant Math & Science Wiki

[23] Hall's marriage theorem - Wikipedia

[24] Hall's Marriage Theorem | Brilliant Math & Science Wiki

[25] Dilworth 学习笔记 - Selfish

9.21 杨氏矩阵

引入

杨氏矩阵(Young tableau)，又名杨表，是一种常用于表示论和舒伯特演算中的组合对象.

杨表是一种特殊的矩阵. 它便于对称群和一般线性群的群表示和性质研究. 杨表由剑桥大学数学家阿尔弗雷德·杨（Alfred Young）于 1900 年首次提出，于 1903 年被德国数学家弗罗贝尼乌斯（Ferdinand Georg Frobenius）应用于对称群的研究.

> **注释**
>
> **表示论**（Representation theory）是数学的一个分支. 它通过将元素表示为向量空间的线性变换来研究抽象代数结构. **舒伯特演算**（Schubert calculus）是代数几何的一个分支，于 19 世纪由赫尔曼·舒伯特为了解决射影几何的计数问题而引入.

定义

杨图

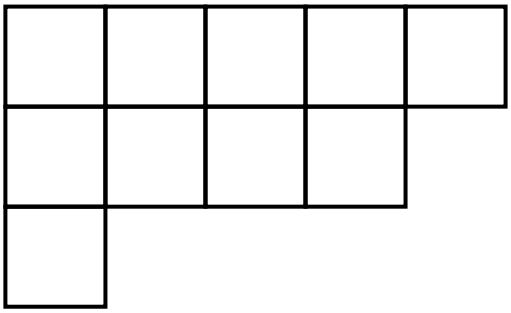
杨图（Young diagram，使用点表示时又称 Ferrers 图[1]，在 分拆数→9.13.12@p1672 一节中有相关介绍）是一个有限的框或单元格集合，左对齐排列，行长按非递增顺序排列. 如果把杨图每行的方格数列出，我们得到了一个非负整数 n（总方格数）的 **整数分拆**(integer partition)λ. 因此，我们可以将杨图的形状看作 λ，因为它携带与其整数拆分相同的信息.

杨图之间的包含关系定义了整数分拆上的一个 偏序→9.20@p1951 关系，此关系拥有 格→9.20@p1951 的结构，被称为 **杨格**(Young's lattice). 如果把杨图各列的方格数列出，则会得到整数分拆 λ 的「共轭分拆」，或「转置分拆」，它所对应到的杨图可由原本的杨图沿主对角线作镜射对称而得.

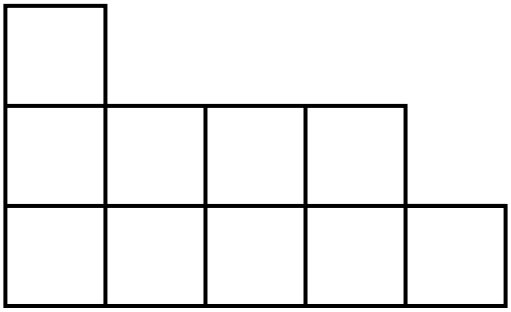
杨图每个方格的位置由分别代表 **行数** 与 **列数** 的两个座标点决定. 列的顺序由左向右, 行的顺序则按方格数的由多向少的方向. 此处需要注意, 根据习惯不同存在着两种不同的杨图画法: 第一个将方格数较少的行排在方格数较多的行的下方, 第二种画法将各行由大到小一层一层往上叠. 由于前一种画法主要由英语国家使用, 而后者通常被法语国家使用, 习惯上我们分别称它们为英式画法和法式画法.

以下表格中分别为整数分拆 $(5,4,1)$ 对应的杨图不同画法:

- 英式画法:



- 法式画法:



杨表

定义

杨表（Young tableau）是通过用取自某个字母表的符号填充杨氏图的框来获得的, 这通常需要是一个全序集和. 填入的元素写作 $x_1,x_2,x_3,\ldots$. 但为了方便起见, 都直接填入正整数.

杨表最初应用于对称群的表示理论时, 允许在杨图的 n 的方格中任意填入 1 到 n 中相异的正整数. 但现在的研究大多使用「标准」的杨表, 即上述条件中各行与各列中方格的数字皆为严格递增的. 由 n 个方格的相异杨表数个数形成 对和数[2]:

> **注释**
>
> **对和数**（involution number/telephone number）是在数学中是一个整数序列, 用来计算 n 条电话线中每条线路最多可以连接到另一条线路时可以相互连接的方法个数. 它还可以用来描述完全图 n 个顶点上的匹配数, n 个对合元素的排列数, Hermite 多项式系数的绝对值之和, 含有 n 个格子的标准杨表的个数, 以及不可约对称群的度数之和.

$1,1,2,4,10,26,76,232,764,2620,9496,\ldots$（OEIS[3] 中的数列 A000085[4]）

在其他应用中, 杨图也可以被填入相同的数字. 若填法的同列数字严格递增, 且同行数字单调递增, 则该杨表被称为是 **半标准的**（Semistandard Young Tableaux, 有时称为列严格）. 杨表中个数字出现的次数记录下来得到的序列被视为杨表的 **权重**. 因此, 标准杨表的权重必然是 $(1,1,\ldots,1)$. 因为在标准杨表中, 1 到 n 的每个正整数恰好各出现一次.

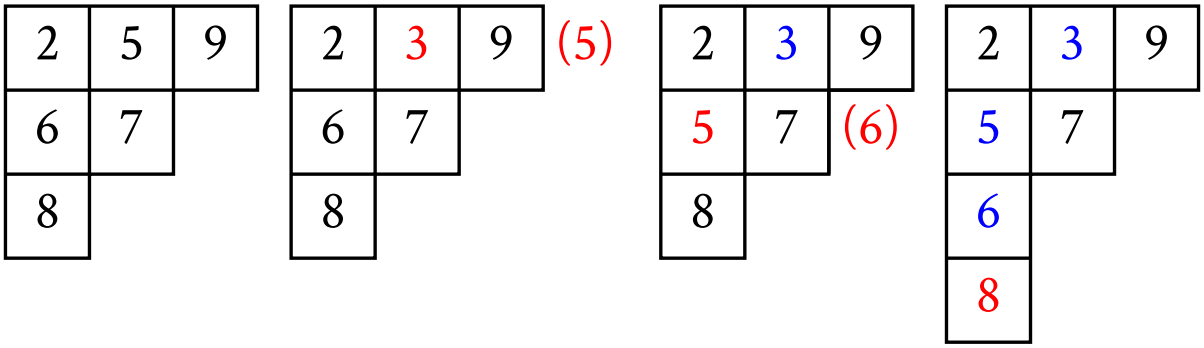
标准杨表的插入算法

排列的性质可以由杨表直观地表现出来. **RSK 插入算法** 就提供了一个将杨表和排列联系起来的途径. 它由 Robinson, Schensted 和 Knuth 提出.

令 S 是一个杨表，定义 $S \leftarrow x$ 表示将 x 从第一行插入杨表中，具体如下:

1. 在当前行中找到最小的比 x 大的数 y.
2. 如果找到了，用 x 去替换 y，移到下一行，令 $x \leftarrow y$ 重复操作 1.
3. 如果找不到，就把 x 放在该行末尾并退出. 记 x 在第 s 行第 t 列，(s,t) 必定是一个边角. 一个格子 (s,t) 是边角当且仅当 $(s+1,t)$ 和 $(s,t+1)$ 都不存在格子.

例如，将 3 插入杨表 $(2,5,9)(6,7)(8)$ 的步骤为:



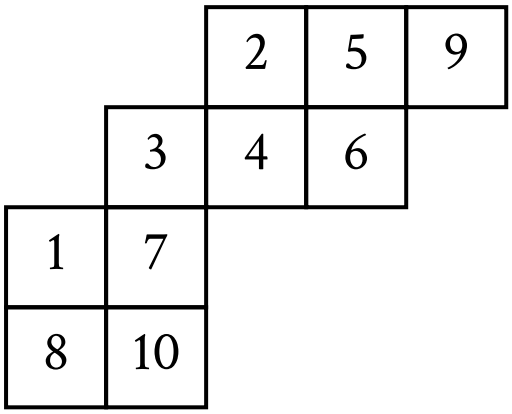
变体

非完全严格标准的杨表有许多变体（Variations）. 例如行严格杨表要求同行数字严格递增，且同列数字单调递增，即列严格杨表的共轭. 此外，在平面分拆（plane partitions）理论中，习惯上会将上述的定义中的递增改为递减. 其他变体例如带状杨表，会先将一些方块打包成群，然后要求各群的方块必须填入相同数字.

斜杨表

给定两个杨图 $\lambda = (\lambda 1, \lambda 2 ...)$、$\mu = (\mu 1, \mu 2, ...)$，满足 λ 包含 μ，即 $\mu i \leq \mu i$ 对所有 i. 定义「斜杨图」λ/μ 为 λ 中所有方格减去 μ 中的所有方格 即 λ 差集 μ. 在斜杨图的各方格中填入元素就形成了 **斜杨表** (Skew tableaux).

例如，下图为整数分拆 $(5,4,1)$ 对应的一个标准斜杨表:



同理，若满足同一列中的数字严格递增，且同一行中的数字单调递增，则该斜杨表被称作 **半标准斜杨表**；若半标准斜杨表满足各方格不重复的填入数字 1 到 n（方格总数），则该斜杨表被称作 **标准斜杨表**. 注意，由不同的 λ 和 μ 可得到相同的 λ/μ. 虽然大部分斜杨表的性质都只依赖于取完差集的方格，但是仍然部分运算依赖于 λ 和 μ 的选取. 因此，λ/μ 必须被视为包含两个元素信息：λ 和 μ. 当 μ 是空分拆（0 的唯一一种分拆）时，斜杨表 λ/μ 就变成杨表 λ.

应用

杨表常用于在组合学、表示理论和代数几何中，用各种不同计算杨表个数的方法得到舒尔函数的定义及相关的恒等式. 在信息学竞赛中，常有考察杨表勾长公式的题目.

勾长

给定一个共有 n 个方格的杨表 π_λ，把 1 到 n 的 n 个数字填入杨表中，使得每行从左到右，每列从下到上都是递增的. 用 dim_{π_λ} 表示可以这样填的方法个数.

对于杨表中的一个方格 v，定义其 **勾长** $\mathrm{hook}(v)$ 等于同行右边的方格数加上同列上面的方格数，再加 1（即方格本身）.

勾长公式

如果用 dim_λ 表示这样的方法个数，**勾长公式** 就是方法个数等于 $n!$ 除以所有方格的勾长的乘积.

$$\dim \pi_\lambda = \frac{n!}{\prod_{x \in Y(\lambda)} \mathrm{hook}(x)}.$$

7	5	4	3	1
5	3	2	1	
1				

所以对于整数分拆 $10 = 5 + 4 + 1$ 的杨表，如上图所示. 有

$$\dim \pi_\lambda = \frac{10!}{7 \cdot 5 \cdot 4 \cdot 3 \cdot 1 \cdot 5 \cdot 3 \cdot 2 \cdot 1 \cdot 1} = 288.$$

种方法.

例题

子序列问题

对于杨表 P, 定义对于一个从 1 到 n 的排列 $X = x_1, ..., x_n$.

1. P_X 中第一行的长度即为排列 X 的 **最长上升子序列（LIS）** 长度. 注意，P 的第一行并不一定是 LIS 本身，所以不能直接利用杨表性质解决「LIS 划分」之类的问题.
2. 对于一个排列 X 和它产生的杨表 P_X，若 X^R 是 X 的翻转，那么 X^R 产生的杨表 P_{X^R} 即为 P_X 交换行列得到.

   例如，对于排列 $X = 1, 5, 7, 2, 8, 6, 3, 4$ 和 $X^R = 4, 3, 6, 8, 2, 7, 5, 1$, 我们可得到如下杨表 P_X:

$P_X =$

1	2	3	4
5	6	8	
7			

$P_{X^R} =$

1	5	7
2	6	
3	8	
4		

3. 杨表 P_X 中的第一列长度即为排列 X 的 **最长下降子序列（LDS）** 长度.

定义长度不超过 k 的 LIS/LDS 长度为 $k-LIS$ 和 $k-LDS$, 此类问题我们同样可以用杨表来解决. 对于 $1-LIS$，显而易见最长的 $1-LIS$ 子序列就是该序列的 LDS，这也正是杨表的第一列；同样可得，杨表前 k 列的长度就是最长的 $k-LIS$ 子序列的长度. 证明如下：

对于一个排列 X 和它的 m 行杨表 P，令排列 X^* 为 $(P_{m,1}...,P_{m,\lambda_m},P_{m-1,1}...,P_{1,1}...P_{1,\lambda_1})$（即将杨表从下往上每行依次写在后面）. 那么 X 一定可以通过交换操作转化成 X^*.

所以，最长 $k-LIS$ 子序列长度可以表示成 $F(k)=\sum_{i=1}^{m}\min(k,\lambda_i)$，即前 k 列的长度和.

CTSC2017 最长上升子序列[5]

有一个长为 n 的数列 b. 对于序列 $B_m=(b_1,b_2,...,b_m)$，设 C 是 B_m 的子序列，且 C 的最长上升子序列的长度不超过 k，询问 C 的长度最大值.

解题思路

多个询问考虑使用扫描线的方法. 这样我们就需要维护每个前缀的杨表. 如果使用以上结论，可以发现问题变成了如何快速维护杨表前 k 列的长度之和. 如果直接维护，复杂度是 $O(n^2\log n)$ 的不能接受. 考虑维护前 $\sqrt{n}$ 列和前 $\sqrt{n}$ 行.

可以发现, 杨表一定不会完全覆盖这个 $W\times H$ 的矩形. 如果 $K\le W$, 那么可以直接得答案; 如果 $K>W$，那么大于 W 的部分一定在 H 行内. 所以可以考虑如何同时维护前 $\sqrt{n}$ 列和前 $\sqrt{n}$ 行. 将这个排列翻转一下就可以得到杨表的翻转，所以只需要再同时维护 $-A_i$ 即可，复杂度为 $O(n\sqrt{n}\log n)$.

BJWC2018 最长上升子序列[6]

现在有一个长度为 n 的随机排列，求它的最长上升子序列长度的期望.

CF1268B 杨氏多米诺骨牌[7]

给定一个具有 n 列长度 $a_1,a_2,...,a_n$ $(a_1\ge a_2\ge...\ge a_n\ge 1)$ 的直方图. $a=[3,2,2,2,1]$ 的杨图. 找到可以在此直方图中绘制的最大数量的非重叠多米诺骨牌（1×2 或 2×1 矩形）.

参考资料与拓展阅读

1. Young Tableau - from Wolfram MathWorld[8]

2. Young tableau - Wikipedia[9]
3. Hook length formula - Wikipedia[10]
4. 袁方舟，《浅谈杨氏矩阵在信息学竞赛中的应用》IOI2019[11], 中国国家候选队论文集，202-229

外部参考资料

[1] Ferrers 图 
[2] 对和数 
[3] OEIS 
[4] A000085 
[5] CTSC2017 最长上升子序列 
[6] BJWC2018 最长上升子序列 
[7] CF1268B 杨氏多米诺骨牌 
[8] Young Tableau - from Wolfram MathWorld 
[9] Young tableau - Wikipedia 
[10] Hook length formula - Wikipedia 
[11] 《浅谈杨氏矩阵在信息学竞赛中的应用》IOI2019 

9.22 拟阵

引言

拟阵（Matroid） 是哈斯勒·惠特尼（Hassler Whitney）于 1935 年提出的一种抽象代数结构，旨在统一和推广关于独立性的概念，例如线性代数中的线性无关性和图论中的无环性.

拟阵为处理与独立性相关的优化问题提供了强大的理论工具，广泛应用于组合数学、图论、算法设计等领域，尤其在为贪心算法等优化方法提供数学理论支持方面发挥了重要作用.

定义

拟阵

一个 **拟阵（Matroid）** 可以表示为 $M = (E, \mathcal{I})$，其中：

- E 是一个有限集，称为 **基础集（Ground Set）**.
- $\mathcal{I}$ 是 E 的子集族，称为 **独立集族（Family of Independent Sets）**，其中的集合称为 **独立集（Independent Set）**. 有以下三个性质：
  - **非空性**：空集是独立的，即 $\emptyset \in \mathcal{I}$.
  - **遗传性**：独立集的任意子集也是独立集. 若 $I \in \mathcal{I}$，则对于任意 $I' \subseteq I$，都有 $I' \in \mathcal{I}$.
  - **扩张性**：若 $I, J \in \mathcal{I}$ 且 $|I| < |J|$，则存在 $j \in J \setminus I$，使得 $I \cup \{j\} \in \mathcal{I}$.

如果一个形如 $(E, \mathcal{I})$ 的结构满足上述三个性质，则称其为一个拟阵.

基

基（Basis） 是拟阵中极大的独立集，即无法再添加元素而保持独立性的独立集. 所有基的集合称为 **基集族**，记为 $\mathcal{B}$.

性质：

1. **等基数性**：所有基的大小都相同，称为拟阵的 **秩（Rank）**.
2. **扩张性**：任何独立集通过添加基中的元素都可以扩张为一个基.

圈

圈（Circuit） 是拟阵中最小的依赖集，即其所有真子集都是独立的，但自身不是独立集，任意两个圈之间不存在包含关系.

秩

秩函数（Rank Function） $r:2^E\to\mathbb{Z}_{\geq 0}$ 将基础集 E 的子集映射到非负整数．对于任意 $S\subseteq E$，$r(S)$ 定义为 S 中最大独立集的大小，即

$$r(S)=\max\{\mid I\mid \mid I\subseteq S\wedge I\in\mathcal{J}\}.$$

性质：

1. **非负性**：对于任意 $S\subseteq E$，有 $0\leq r(S)\leq\mid S\mid$.
2. **单调性**：若 $A\subseteq B\subseteq E$，则 $r(A)\leq r(B)$.
3. **次模性**：对于任意 $A,B\subseteq E$，有 $r(A\cup B)+r(A\cap B)\leq r(A)+r(B)$.

典型示例

1. 均匀拟阵（Uniform Matroid）

定义：给定基础集 E 和非负整数 k，均匀拟阵 $U_{k,E}$ 的独立集族是所有大小不超过 k 的子集，表示为：

$$\mathcal{J}=\{I\subseteq E\mid \mid I\mid\leq k\}.$$

- **基（Bases）**：所有大小为 k 的子集.
- **圈（Circuits）**：所有大小为 $k+1$ 的子集.
- **秩（Rank）**：$r(E)=\min(k,\mid E\mid)$，即独立集中最多能有 k 个元素.

2. 图拟阵（Graphical Matroid）

定义：给定一个无向图 $G=(V,E)$，图拟阵 $M(G)$ 的基础集是边集 E，其独立集族是所有不包含环的边集，即所有的森林.

- **基**：图中的生成树（在连通图的情况下）．生成树是极大的独立集，无法再增加边而不形成环.
- **圈**：图中的简单环，去掉环中的任意一条边，剩余部分都为独立集.
- **秩**：$r(E)=\mid V\mid-c$，其中 c 是图的连通分支数．对于一个连通的无向图，其秩等于顶点数减一，即 $\mid V\mid-1$.

3. 线性拟阵（Linear Matroid）

定义：线性拟阵基于向量空间．给定向量空间 V，基础集 E 是 V 中的一组有限向量，其独立集族是 E 中所有线性无关的向量子集.

- **基**：极大的线性无关向量集，其大小等于向量空间的维数.
- **圈**：最小的线性相关向量集合，其任意真子集都是独立的，而自身是线性相关的.
- **秩**：线性拟阵的秩 $r(E)=\dim(V)$，即向量空间的维数．独立集的大小不能超过向量空间的维数.

4. 划分拟阵（Partition Matroid）

定义：将基础集 E 划分为不相交的子集 $E_1,E_2,...,E_m$，并为每个子集 E_i 指定一个非负整数 k_i．划分拟阵的独立集族由满足每个部分选取元素数量不超过 k_i 的子集组成，表示为：

$$\mathcal{J}=\{I\subseteq E\mid \forall i,\mid I\cap E_i\mid\leq k_i\}.$$

- **基**：满足 $\mid I\cap E_i\mid=k_i$ 的独立集是划分拟阵的基．每个基在每个子集中选取了恰好 k_i 个元素.
- **圈**：划分拟阵的圈是最小的依赖集，即包含至少一个元素数量超过 k_i 的子集.
- **秩**：划分拟阵的秩为 $r(E)=\sum_{i=1}^m k_i$，即最大独立集的大小等于每个子集中允许选取的最大元素数的总和.

5. 有色拟阵（Colored Matroid）

定义：有色拟阵是划分拟阵的一种特殊形式，其中每个元素都赋予了颜色．给定基础集 E 和颜色集 C，每个元素 $e\in E$ 都与某个颜色 $c\in C$ 相关联．有色拟阵的独立集不仅需要满足普通拟阵的独立性条件，还必须遵守颜色上指定的限制，例如同一种颜色的元素在独立集中最多选取一定数量.

- **基**：有色拟阵的基是符合颜色限制和独立性条件的极大独立集.

- **圈**：圈是最小的依赖集，包含至少一个违反独立性或颜色限制的元素集合.
- **秩**：有色拟阵的秩是满足颜色限制条件下的最大独立集大小. 它既依赖于拟阵的结构，也依赖于颜色限制的具体规定.

构造和运算

对偶

给定拟阵 $M = (E, \mathcal{I})$，其 **对偶拟阵** $M^* = (E, \mathcal{I}^*)$ 定义为:

$$\mathcal{I}^* = \{I^* \subseteq E \mid \exists B \in \mathcal{I}, \mid B \mid = r(E), B \subseteq E \setminus I^*\}.$$

性质:

- **基**：对偶拟阵 M^* 的基是 M 的基在基础集 E 中的补集. 换句话说，如果 B 是 M 的基，那么 $E \setminus B$ 就是 M^* 的基.
- **秩函数**：对偶拟阵的秩函数为 $r^*(S) = \mid S \mid - r(E) + r(E \setminus S)$，其中 S 是 E 的子集. 这意味着对偶拟阵的秩可以通过基础集的大小、原拟阵的秩以及从基础集中移除 S 后的秩来计算.
- **自反性**：对偶拟阵的对偶仍是原拟阵，即 $(M^*)^* = M$.

示例:

对于一个无向图 $G = (V, E)$，图拟阵 $M(G)$ 的对偶 $M(G)^*$ 是由图的割集组成的拟阵. 图拟阵 $M(G)$ 的基是图中的生成树，而其对偶 $M(G)^*$ 的基是这些生成树的补集，对偶 $M(G)^*$ 的圈则是图的最小割集，即将图分成两个不连通部分的最小边集.

例如，考虑一个简单的三角形图 G，其边集为 $E = \{e_1, e_2, e_3\}$. 图拟阵 $M(G)$ 的基是两条边的集合（如 $\{e_1, e_2\}$），而对偶拟阵 $M(G)^*$ 的基是单条边的集合（如 $\{e_3\}$），$M(G)^*$ 的圈是两条边的集合（即最小割集，如 $\{e_2, e_3\}$），因为移除其中的一条边就会将图分割为两个连通分支.

删除和收缩

删除（Deletion）:

对于 $A \subseteq E$，拟阵 M 删除 A 后得到新的拟阵 $M \setminus A$，其独立集族 $\mathcal{I}'$ 定义为:

$$\mathcal{I}' = \{I \subseteq E \setminus A \mid I \in \mathcal{I}\}.$$

可以看出，删除操作就是从拟阵中移除某些元素，并保留剩余元素形成的独立集，其保持原独立集不变，只是移除了元素.

收缩（Contraction）:

对于 $A \subseteq E$，拟阵 M 收缩 A 后得到拟阵 M/A，其独立集族 $\mathcal{I}''$ 定义为:

$$\mathcal{I}'' = \left\{ I \subseteq E \setminus A \,\middle|\, \exists B \subseteq A, B \in \mathcal{I}, r(B) = r(A), I \cup B \in \mathcal{I} \right\}$$

收缩操作可以理解为将集合 A 中的元素缩约，并考虑剩下的元素与 A 的基一起形成的独立集. 收缩的结果依赖于集合 A 的基，缩约后的独立集实际上是对原拟阵中更高秩的子集进行约简后得到的独立集.

示例 - 图拟阵:

- **删除**：在图拟阵中，删除操作即从图中删除一些边. 一个图 G 删除某条边后，考虑的是剩余边所形成的独立集，即那些不包含环的边集. 例如，如果从一个三角形图中删除一条边，剩下的两个边仍然是一个森林.
- **收缩**：收缩操作则是将某条边收缩为一个顶点. 对于图拟阵，收缩一条边相当于将这条边的两个顶点合并成一个顶点，并删除该边，合并顶点后，图中的其他边仍然可以形成独立集. 例如，在一个三角形图中，收缩任意一条边将把两个顶点合并成一个，剩下的两条边将构成一个新的拟阵.

拟阵和贪心

问题描述：

拟阵的应用之一是解决贪心算法中的最优化问题．具体而言，给定一个拟阵 $M=(S,\mathcal{I})$，其中 S 是基础集，$\mathcal{I}$ 是独立集族．对于每个元素 $x\in S$，赋予一个正整数权值 $w(x)$，目标是找到权值最大的独立集，形式化为：

$$\max_{A\in\mathcal{I}} w(A)=\max_{A\in\mathcal{I}}\sum_{x\in A} w(x)$$

显然，权值最大独立集必须是极大独立集．如果一个独立集 A 不是极大独立集，则存在一个可以加入 A 的元素 x，且由于 $w(x)>0$，加入该元素后权值会增加，说明 A 不是权值最大的独立集．

步骤

贪心算法求解权值最大独立集的步骤如下：

1. **元素排序**：将基础集 S 按照权值从大到小排序，记为序列 $e_1,e_2,...,e_n$．
2. **初始化**：设独立集 $A=\emptyset$．
3. **构建独立集**：依次考虑排序后的元素 e_i，如果 $A\cup\{e_i\}\in\mathcal{I}$，则更新 $A=A\cup\{e_i\}$．
4. **输出结果**：最终的集合 A 即为权值最大的独立集．

复杂度分析：

设 $n=|\ S\ |$ 为基础集的大小，$f(n)$ 表示判断一个集合是否为独立集的复杂度．贪心算法的时间复杂度为：

$$O(n\log n+nf(n))$$

其中，$O(n\log n)$ 是排序的复杂度，$O(nf(n))$ 是逐一判断独立性的复杂度．

> **备注**
>
> - 在图拟阵中，可以使用 并查集→10.6.1@p1999 来高效检测是否形成环，从而使 $f(n)$ 接近常数时间．
> - 在线性拟阵中，独立性检测通常涉及矩阵运算，其复杂度依赖于具体实现方式．

正确性证明：

设 $M=(S,\mathcal{I})$ 是一个拟阵，$A\in\mathcal{I}$ 是一个独立集，且 A 是某个权值最大独立集 T 的子集．定义集合 $P=\{x\in S\setminus A\mid A\cup\{x\}\in\mathcal{I}\}$，即所有加入 A 后，仍然使 A 保持独立性的元素所构成的集合．

设 y 为 P 中权值最大的元素，则 $A'=A\cup\{y\}$ 也是某个权值最大独立集的子集，证明如下：

假设 $A'=A\cup\{y\}$ 不是任何权值最大独立集的子集，则存在一个权值最大的独立集 T，且 $|\ A'|<|\ T\ |$．

由于 $|\ A'|<|\ T\ |$，根据拟阵的 **扩张性**，存在 $x\in T\setminus A'$ 使得 $A'\cup\{x\}\in\mathcal{I}$．

利用 **扩张性**，不断将 x 加入 A'，最终构造出一个新的独立集 A''，使得 $|\ A''|=|\ T\ |$．

设 $K=A''\cap T$，此时有 $x=T\setminus K$，$y=A''\setminus K$．由于 y 为 P 中权值最大的元素，有 $w(x)\le w(y)$．

因此，$w(A'')=w(K)+w(y)\ge w(K)+w(x)=w(T)$，此时：

- 若 $w(A'')>w(T)$，则 T 不是权值最大独立集，与假设矛盾．
- 若 $w(A'')=w(T)$，则 A'' 为权值最大独立集，且 A' 为其子集，与假设 A' 不是任何权值最大独立集的子集矛盾．

综上，假设不成立，即 $A'=A\cup\{y\}$ 必须是某个权值最大独立集的子集，因此通过不断使用贪心策略，最终可以找到权值最大的独立集．

示例

最小生成树：

给定一个连通无向图 $G = (V, E)$，每条边 $e \in E$ 都具有权值 $w(e)$．目标为找到一棵生成树，使其包含所有顶点且总权值最小．

拟阵的构建：

为了将最小生成树问题形式化为拟阵问题，可以构建图拟阵 $M(G)$：

- **基础集**：$S = E$，即图中的所有边．
- **独立集族**：$\mathcal{I}$ 为所有不包含环的边集（即所有森林）．

贪心算法：

在图拟阵的框架下，Kruskal 算法→11.10.1@p2764 是一个典型的基于拟阵理论的贪心算法，可以用于构建最小生成树．虽然 Prim 算法→11.10.1@p2764 也是一种有效的贪心算法，同样能够找到最小生成树，但它并不严格依赖于拟阵的贪心．因此，在拟阵理论的讨论中，Kruskal 算法是主要的贪心算法实例．

- **Kruskal 算法**：
  1. **边排序**：将所有边按权值从小到大排序．
  2. **逐步选择**：依次选择权值最小的边，若加入后不形成环，则将其加入生成树．
  3. **终止条件**：重复上述过程，直到生成树包含 $|V| - 1$ 条边．
- **Prim 算法**：
  - **原理**：Prim 算法通过从一个起始顶点开始，逐步扩展生成树，每次选择连接树内与树外的最小权值边．
  - 虽然 Prim 算法也是贪心的，但其选择策略不同于其他基于拟阵扩张性质的贪心算法．因此，在拟阵理论的严格意义下，Prim 算法不被视为典型的拟阵贪心算法．

拟阵交

对于定义在同一基础集 S 上的两个拟阵 $M_1 = (S, \mathcal{I}_1)$ 和 $M_2 = (S, \mathcal{I}_2)$，若 $\mathcal{I} = \mathcal{I}_1 \cap \mathcal{I}_2$ 满足拟阵独立集族的三条性质，则称 $M = (S, \mathcal{I})$ 为 M_1 和 M_2 的 **交**．

注意：并非任意两个拟阵的交都是一个拟阵，只有当其独立集族的交集满足拟阵独立集族定义中的三条性质时，其交才构成一个拟阵．

问题描述

1. **最大独立集**：在 $\mathcal{I}_1 \cap \mathcal{I}_2$ 中找到最大的独立集（即具有最大基数的独立集）．
2. **加权最大独立集**：给定权值函数 $w : S \to \mathbb{R}$，在 $\mathcal{I}_1 \cap \mathcal{I}_2$ 中找到权值和最大的独立集．

算法

无权版本：

1. **初始化**：选择一个初始独立集 $I \in \mathcal{I}_1 \cap \mathcal{I}_2$，通常设定 $I = \emptyset$．
2. **迭代**：
   - **构建交换图**：根据当前独立集 I 构建交换图 $D_{M_1, M_2}(I)$．
   - **路径选择**：在交换图中，寻找从源点 s 到汇点 t 的增广路径 P．
   - **增广**：沿路径 P 从 s 到 t 遍历每一个节点：
     - 如果节点属于左部顶点（即 I 中的元素），则将该元素从 I 中移除．
     - 如果节点属于右部顶点（即 $S \setminus I$ 中的元素），则将该元素加入 I 中．
   - **重复**：更新独立集 I 后，重复上述步骤，直到无法找到新的增广路径为止．
3. **结果**：最终得到的独立集 I 即为拟阵交 $M = M_1 \cap M_2$ 中的一个最大独立集．

加权版本：

为了找到权值和最大的独立集，算法需要在增广路径的选择上进行优化.

1. **权值设置**：对于每个元素 $e \in S$，定义其在交换图中的权值 $w'(e)$：
   - **左部顶点**（I 中的元素）：$w'(e) = -w(e)$.
   - **右部顶点**（$S \setminus I$ 中的元素）：$w'(e) = w(e)$.
2. **路径选择**：在交换图 $D_{M_1,M_2}(I)$ 中，寻找一条从源点 s 到汇点 t 的 **增广路径** P，使得沿路径进行增广操作后，独立集 I 的总权值增加最大.
   - **增广条件**：路径 P 上加入的元素的权值总和大于移除的元素的权值总和，即 $\sum_{y\in \text{加入的元素}} w(y) > \sum_{x\in \text{移除的元素}} w(x)$
3. **增广操作**：沿路径 P 从 s 到 t 遍历每一个节点：
   - 如果节点属于左部顶点（即 I 中的元素），则将该元素从 I 中移除.
   - 如果节点属于右部顶点（即 $S \setminus I$ 中的元素），则将该元素加入 I 中.
4. **迭代**：重复步骤 1 至 3，不断构建交换图并寻找增广路径，逐步优化独立集 I 的总权值.
5. **终止条件**：当无法在交换图中找到满足增广条件的路径时，算法终止.
6. **结果**：最终得到的独立集 I 即为拟阵交 $M = M_1 \cap M_2$ 中的一个 **权值最大独立集**.

复杂度：

- **增广次数**：设两个拟阵的最大秩分别为 r_1 和 r_2，则最大增广次数为 $\min(r_1, r_2)$.
- **每次增广的复杂度**：
  - 构建交换图的复杂度为 $O(n^2)$，其中 $n = |\ S\ |$.
  - 寻找增广路径的复杂度取决于路径搜索策略，通常为 $O(n^2)$，例如使用广度优先搜索.
- **总时间复杂度**：总体的时间复杂度为 $O(r \cdot n^2)$，其中 $r = \min(r_1, r_2)$.

例题

最小生成树：

给定一个无向图 $G = (V, E)$，每条边 $e \in E$ 都有一个权值 $w(e)$. 寻找一棵生成树，使其包含所有顶点且总权值最小.

- 详细介绍：最小生成树→11.10.1@p2764.
- 题目模板：洛谷 P3366【模板】最小生成树[1].

解题思路

使用 Kruskal 算法，将所有边按权值从小到大排序，然后逐步选择边，若加入后不形成环，则将其加入生成树，最终得到的生成树即为最小生成树.

Colorful Graph：

给定一张带有多种颜色的无向图 $G = (V, E)$，每条边有一个颜色属性. 寻找一个最大的边集，使得：

1. 所选边不形成任何环.
2. 每种颜色的边数不超过 k 条（k 为给定的正整数）.

解题思路

1. **拟阵建模**：
   - **图拟阵** (M_1): 定义为所有不形成环的边集, 即独立集族 $\mathcal{I}_1$ 包含所有不构成环的边集合.
   - **颜色拟阵** (M_2): 定义为每种颜色的边数不超过 k 的边集，即独立集族 $\mathcal{I}_2$ 包含所有满足每种颜色边数 $\leq k$ 的边集合.
2. **求解拟阵交**：通过求解 $M = M_1 \cap M_2$，找到既不形成环又满足每种颜色边数不超过 k 的最大边集.

约束的资源分配问题:

在一个资源分配问题中，有一组资源 $R = \{r_1, r_2, ..., r_n\}$ 和一组项目 $P = \{p_1, p_2, ..., p_m\}$. 每个项目 p_i 需要分配一定数量的资源，且每种资源的总分配量不能超过其供应量.

目标：寻找一个资源分配方案，使其满足所有项目需求且不超过资源供应量.

解题思路

1. **拟阵建模**：
   - **需求拟阵** (M_1)：定义为满足各项目资源需求的分配方案，即独立集族 $\mathcal{I}_1$ 包含所有满足项目需求的资源分配集合.
   - **供应拟阵** (M_2)：定义为不超过每种资源供应量的分配方案，即独立集族 $\mathcal{I}_2$ 包含所有满足资源供应限制的资源分配集合.
2. **求解拟阵交**：通过求解 $M = M_1 \cap M_2$，找到既满足所有项目需求又不超过资源供应量的资源分配方案.

参考资料与注释

1. Wikipedia - Matroid[2]
2. 百度百科 - 拟阵[3]
3. 洛谷 - 拟阵与最优化问题[4]
4. 洛谷 - 从拟阵基础到 Shannon 开关游戏[5]

外部参考资料

[1] 洛谷 P3366【模板】最小生成树

[2] Wikipedia - Matroid

[3] 百度百科 - 拟阵

[4] 洛谷 - 拟阵与最优化问题

[5] 洛谷 - 从拟阵基础到 Shannon 开关游戏

9.23 Berlekamp–Massey 算法

Authors: AntiLeaf

Berlekamp–Massey 算法是一种用于求数列的最短递推式的算法. 给定一个长为 n 的数列，如果它的最短递推式的阶数为 m, 则 Berlekamp–Massey 算法能够在 $O(nm)$ 时间内求出数列的每个前缀的最短递推式. 最坏情况下 $m = O(n)$，因此算法的最坏复杂度为 $O(n^2)$.

定义

定义一个数列 $\{a_0 ... a_{n-1}\}$ 的递推式为满足下式的序列 $\{r_0 ... r_m\}$:

$\sum_{j=0}^{m} r_j a_{i-j} = 0, \forall i \geq m$

其中 $r_0 = 1$. m 称为该递推式的 **阶数**.

数列 $\{a_i\}$ 的最短递推式即为阶数最小的递推式.

做法

与上面定义的稍有不同，这里定义一个新的递推系数 $\{f_0 ... f_{m-1}\}$，满足:

$a_i = \sum_{j=0}^{m-1} f_j a_{i-j-1}, \forall i \geq m$

容易看出 $f_i = -r_{i+1}$，并且阶数 m 与之前的定义是相同的.

我们可以增量地求递推式，按顺序考虑 $\{a_i\}$ 的每一位，并在递推结果出现错误时对递推系数 $\{f_i\}$ 进行调整. 方便起见，以下将前 i 位的最短递推式记为 $F_i = \{f_{i,j}\}$.

显然初始时有 $F_0 = \{\}$. 假设递推系数 F_{i-1} 对数列 $\{a_i\}$ 的前 $i-1$ 项均成立，这时对第 i 项就有两种情况：

1. 递推系数对 a_i 也成立，这时不需要进行任何调整，直接令 $F_i = F_{i-1}$ 即可.
2. 递推系数对 a_i 不成立，这时需要对 F_{i-1} 进行调整，得到新的 F_i.

设 $\Delta_i = a_i - \sum_{j=0}^{m} f_{i-1,j} a_{i-j-1}$，即 a_i 与 F_{i-1} 的递推结果的差值.

如果这是第一次对递推系数进行修改，则说明 a_i 是序列中的第一个非零项. 这时直接令 F_i 为 i 个 0 即可，显然这是一个合法的最短递推式.

否则设上一次对递推系数进行修改时，已考虑的 $\{a_i\}$ 的项数为 k. 如果存在一个序列 $G = \{g_0 \cdots g_{m'-1}\}$，满足：

$\sum_{j=0}^{m'-1} g_j a_{i'-j-1} = 0, \forall i' \in [m', i)$

并且 $\sum_{j=0}^{m'-1} g_j a_{i-j-1} = \Delta_i$，那么不难发现将 F_k 与 G 按位分别相加之后即可得到一个合法的递推系数 F_i.

考虑如何构造 G. 一种可行的构造方案是令

$G = \{0, 0, ..., 0, \frac{\Delta_i}{\Delta_k}, -\frac{\Delta_i}{\Delta_k} F_{k-1}\}$

其中前面一共有 $i-k-1$ 个 0，且最后的 $-\frac{\Delta_i}{\Delta_k} F_{k-1}$ 表示将 F_{k-1} 每项乘以 $-\frac{\Delta_i}{\Delta_k}$ 后接在序列后面.

不难验证此时 $\sum_{j=0}^{m'-1} g_j a_{i-j-1} = \Delta_k \frac{\Delta_i}{\Delta_k} = \Delta_i$，因此这样构造出的是一个合法的 G. 将 F_i 赋值为 F_k 与 G 逐项相加后的结果即可.

如果要求的是符合最开始定义的递推式 $\{r_i\}$，则将 $\{f_j\}$ 全部取相反数后在最开始插入 $r_0 = 1$ 即可.

从上述算法流程中可以看出，如果数列的最短递推式的阶数为 m，则算法的复杂度为 $O(nm)$. 最坏情况下 $m = O(n)$，因此算法的最坏复杂度为 $O(n^2)$.

在实现算法时，由于每次调整递推系数时都只需要用到上次调整时的递推系数 F_k，因此如果只需要求整个数列的最短递推式，可以只存储当前递推系数和上次调整时的递推系数，空间复杂度为 $O(n)$.

参考实现

```cpp
vector<int> berlekamp_massey(const vector<int> &a) {
  vector<int> v, last;  // v is the answer, 0-based, p is the module
  int k = -1, delta = 0;

  for (int i = 0; i < (int)a.size(); i++) {
    int tmp = 0;
    for (int j = 0; j < (int)v.size(); j++)
      tmp = (tmp + (long long)a[i - j - 1] * v[j]) % p;

    if (a[i] == tmp) continue;

    if (k < 0) {
      k = i;
      delta = (a[i] - tmp + p) % p;
      v = vector<int>(i + 1);

      continue;
    }

    vector<int> u = v;
    int val = (long long)(a[i] - tmp + p) * power(delta, p - 2) % p;
```

```
22
23      if (v.size() < last.size() + i - k) v.resize(last.size() + i - k);
24
25      (v[i - k - 1] += val) %= p;
26
27      for (int j = 0; j < (int)last.size(); j++) {
28        v[i - k + j] = (v[i - k + j] - (long long)val * last[j]) % p;
29        if (v[i - k + j] < 0) v[i - k + j] += p;
30      }
31
32      if ((int)u.size() - i < (int)last.size() - k) {
33        last = u;
34        k = i;
35        delta = a[i] - tmp;
36        if (delta < 0) delta += p;
37      }
38    }
39
40    for (auto &x : v) x = (p - x) % p;
41    v.insert(v.begin(), 1);
42
43    return v;  // $\forall i, \sum_{j = 0} ^ m a_{i - j} v_j = 0$
44  }
```

朴素的 Berlekamp–Massey 算法求解的是有限项数列的最短递推式．如果待求递推式的序列有无限项，但已知最短递推式的阶数上界，则只需取出序列的前 $2m$ 项即可求出整个序列的最短递推式．（证明略）

应用

由于 Berlekamp–Massey 算法的数值稳定性比较差，在处理实数问题时一般很少使用．为了叙述方便，以下均假定在某个质数 p 的剩余系下进行运算．

求向量列或矩阵列的最短递推式

如果要求向量列 $\boldsymbol{v}_i$ 的最短递推式，设向量的维数为 n，我们可以随机一个 n 维行向量 $\mathbf{u}^T$，并计算标量序列 $\{\boldsymbol{u}^T\boldsymbol{v}_i\}$ 的最短递推式．由 Schwartz–Zippel 引理，二者的最短递推式有至少 $1-\frac{n}{p}$ 的概率相同．

求矩阵列 $\{A_i\}$ 的最短递推式也是类似的，设矩阵的大小为 $n\times m$，则只需随机一个 $1\times n$ 的行向量 $\mathbf{u}^T$ 和一个 $m\times 1$ 的列向量 $\boldsymbol{v}$，并计算标量序列 $\{\boldsymbol{u}^TA_i\boldsymbol{v}\}$ 的最短递推式即可．由 Schwartz–Zippel 引理可以类似地得到二者相同的概率至少为 $1-\frac{n+m}{p}$．

优化矩阵快速幂

设 $\boldsymbol{f}_i$ 是一个 n 维列向量，并且转移满足 $\boldsymbol{f}_i=A\boldsymbol{f}_{i-1}$，则可以发现 $\{\boldsymbol{f}_i\}$ 是一个不超过 n 阶的线性递推向量列．（证明略）

我们可以直接暴力求出 $\boldsymbol{f}_0\cdots\boldsymbol{f}_{2n-1}$，然后用前面提到的做法求出 $\{\boldsymbol{f}_i\}$ 的最短递推式，再调用 常系数齐次线性递推→9.12.10@p1572 即可．

如果要求的向量是 $\boldsymbol{f}_m$，则算法的复杂度是 $O(n^3+n\log n\log m)$．如果 A 是一个只有 k 个非零项的稀疏矩阵，则复杂度可以降为 $O(nk+n\log n\log m)$．但由于算法至少需要 $O(nk)$ 的时间预处理，因此在压力不大的情况下也可以使用 $O(n^2\log m)$ 的线性递推算法，复杂度同样是可以接受的．

求矩阵的最小多项式

方阵 A 的最小多项式是次数最小的并且满足 $f(A)=0$ 的多项式 f．

实际上最小多项式就是 $\{A^i\}$ 的最小递推式，所以直接调用 Berlekamp–Massey 算法就可以了．如果 A 是一个 n 阶方阵，则显然最小多项式的次数不超过 n．

瓶颈在于求出 A^i，因为如果直接每次做矩阵乘法的话复杂度会达到 $O(n^4)$．但考虑到求矩阵列的最短递推式时实际上求的是 $\{\boldsymbol{u}^T A^i \boldsymbol{v}\}$ 的最短递推式，因此我们只要求出 $A^i\boldsymbol{v}$ 就行了．

假设 A 有 k 个非零项，则复杂度为 $O(kn+n^2)$．

求稀疏矩阵行列式

如果能求出方阵 A 的特征多项式，则常数项乘上 $(-1)^n$ 就是行列式．但是最小多项式不一定就是特征多项式．

实际上如果把 A 乘上一个随机对角阵 B，则 AB 的最小多项式有至少 $1-\frac{2n^2-n}{p}$ 的概率就是特征多项式．最后再除掉 $\det B$ 就行了．

设 A 为 n 阶方阵，且有 k 个非零项，则复杂度为 $O(kn+n^2)$．

求稀疏矩阵的秩

设 A 是一个 $n\times m$ 的矩阵，首先随机一个 $n\times n$ 的对角阵 P 和一个 $m\times m$ 的对角阵 Q，然后计算 $QAPA^TQ$ 的最小多项式即可．

实际上不用调用矩阵乘法，因为求最小多项式时要用 $QAPA^TQ$ 乘一个向量，所以我们依次把这几个矩阵乘到向量里就行了．答案就是最小多项式除掉所有 x 因子后剩下的次数．

设 A 有 k 个非零项，且 $n\le m$，则复杂度为 $O(kn+n^2)$．

解稀疏方程组

问题：已知 $A\mathbf{x}=\mathbf{b}$，其中 A 是一个 $n\times n$ 的 **满秩** 稀疏矩阵，$\mathbf{b}$ 和 $\mathbf{x}$ 是 $1\times n$ 的列向量．$A,\mathbf{b}$ 已知，需要在低于 n^ω 的复杂度内解出 x．

做法：显然 $\mathbf{x}=A^{-1}\mathbf{b}$．如果我们能求出 $\{A^i\mathbf{b}\}(i\ge 0)$ 的最小递推式 $\{r_0 ... r_{m-1}\}(m\le n)$，那么就有结论

$A^{-1}\mathbf{b}=-\frac{1}{r_{m-1}}\sum_{i=0}^{m-2}A^i\mathbf{b}r_{m-2-i}$

（证明略）

因为 A 是稀疏矩阵，直接按定义递推出 $\mathbf{b}...A^{2n-1}\mathbf{b}$ 即可．

同样地，设 A 中有 k 个非零项，则复杂度为 $O(kn+n^2)$．

参考实现

```cpp
vector<int> solve_sparse_equations(const vector<tuple<int, int, int>> &A,
                                   const vector<int> &b) {
  int n = (int)b.size();  // 0-based

  vector<vector<int>> f({b});

  for (int i = 1; i < 2 * n; i++) {
    vector<int> v(n);
    auto &u = f.back();

    for (auto [x, y, z] : A)  // [x, y, value]
      v[x] = (v[x] + (long long)u[y] * z) % p;

    f.push_back(v);
  }
```

```
16
17    vector<int> w(n);
18    mt19937 gen;
19    for (auto &x : w) x = uniform_int_distribution<int>(1, p - 1)(gen);
20
21    vector<int> a(2 * n);
22    for (int i = 0; i < 2 * n; i++)
23      for (int j = 0; j < n; j++) a[i] = (a[i] + (long long)f[i][j] * w[j])
  % p;
24
25    auto c = berlekamp_massey(a);
26    int m = (int)c.size();
27
28    vector<int> ans(n);
29
30    for (int i = 0; i < m - 1; i++)
31      for (int j = 0; j < n; j++)
32        ans[j] = (ans[j] + (long long)c[m - 2 - i] * f[i][j]) % p;
33
34    int inv = power(p - c[m - 1], p - 2);
35
36    for (int i = 0; i < n; i++) ans[i] = (long long)ans[i] * inv % p;
37
38    return ans;
39  }
```

例题

1. LibreOJ #163. 高斯消元 2[1]
2. ICPC2021 台北 Gym103443E. Composition with Large Red Plane, Yellow, Black, Gray, and Blue[2]

外部参考资料

[1] LibreOJ #163. 高斯消元 2

[2] ICPC2021 台北 Gym103443E. Composition with Large Red Plane, Yellow, Black, Gray, and Blue

第十章

数据结构

10.1 数据结构部分简介

Authors: HeRaNO, Zhoier, hsfzLZH1

数据结构是在计算机中存储、组织数据的方式. 小到变量、数组, 大到线段树、平衡树, 都是数据结构.

程序运行离不开数据结构, 不同的数据结构又各有优劣, 能够处理的问题各不相同, 而根据具体问题选取合适的数据结构, 可以大大提升程序的效率. 所以, 学习各种各样的数据结构是很有必要的.

10.2 栈

引入



栈是 OI 中常用的一种线性数据结构. 请注意, 本文主要讲的是栈这种数据结构, 而非程序运行时的系统栈/栈空间.

栈的修改与访问是按照后进先出的原则进行的, 因此栈通常被称为是后进先出（last in first out）表, 简称 LIFO 表.

Warning

LIFO 表达的是 **当前在容器** 内最后进来的最先出去.

我们考虑这样一个栈

```
push(1)
pop(1)
push(2)
pop(2)
```

如果从整体考虑, 1 最先入栈, 最先出栈, 2 最后入栈, 最后出栈, 这样就成了一个先进先出表, 显然是错误的.

所以, 在考虑数据结构是 LIFO 还是 FIFO 的时候, 应当考虑在当前容器内的情况.

使用数组模拟栈

我们可以方便的使用数组来模拟一个栈, 如下:

实现

- C++

```cpp
int st[N];
// 这里使用 st[0] (即 *st) 代表栈中元素数量, 同时也是栈顶下标

```

```cpp
// 压栈 :
st[++*st] = var1;
// 取栈顶 :
int u = st[*st];
// 弹栈 : 注意越界问题, *st == 0 时不能继续弹出
if (*st) --*st;
// 清空栈
*st = 0;
```

- Python

```python
st = [0] * N
# 这里使用 st[0] 代表栈中元素数量，同时也是栈顶下标

# 压栈 :
st[st[0] + 1] = var1
st[0] = st[0] + 1
# 取栈顶:
u = st[st[0]]
# 弹栈: 注意越界问题, *st == 0 时不能继续弹出
if st[0]:
    st[0] = st[0] - 1
# 清空栈
st[0] = 0
```

C++ STL 中的栈

C++ 中的 STL 也提供了一个容器 `std::stack`，使用前需要引入 `stack` 头文件.

STL 中对 stack 的定义

```cpp
// clang-format off
template<
    class T,
    class Container = std::deque<T>
> class stack;
```

`T` 为 stack 中要存储的数据类型.

`Container` 为用于存储元素的底层容器类型. 这个容器必须提供通常语义的下列函数:

- `back()`
- `push_back()`
- `pop_back()`

STL 容器 `std::vector`、`std::deque` 和 `std::list` 满足这些要求. 如果不指定，则默认使用 `std::deque` 作为底层容器.

STL 中的 `stack` 容器提供了一众成员函数以供调用，其中较为常用的有:

- 元素访问
  - `st.top()` 返回栈顶
- 修改
  - `st.push()` 插入传入的参数到栈顶
  - `st.pop()` 弹出栈顶

- 容量
  - `st.empty()` 返回是否为空
  - `st.size()` 返回元素数量

此外，`std::stack` 还提供了一些运算符．较为常用的是使用赋值运算符 `=` 为 `stack` 赋值，示例：

```cpp
// 新建两个栈 st1 和 st2
std::stack<int> st1, st2;

// 为 st1 装入 1
st1.push(1);

// 将 st1 赋值给 st2
st2 = st1;

// 输出 st2 的栈顶元素
cout << st2.top() << endl;
// 输出: 1
```

使用 Python 中的 list 模拟栈

在 Python 中，你可以使用列表来模拟一个栈：

实现

```python
st = [5, 1, 4]

# 使用 append() 向栈顶添加元素
st.append(2)
st.append(3)
# >>> st
# [5, 1, 4, 2, 3]

# 使用 pop 取出栈顶元素
st.pop()
# >>> st
# [5, 1, 4, 2]

# 使用 clear 清空栈
st.clear()
```

参考资料

1. std::stack - zh.cppreference.com[1]

外部参考资料

[1] std::stack - zh.cppreference.com



10.3 队列

本页面介绍和队列有关的数据结构及其应用．



引入

队列（queue）是一种具有「先进入队列的元素一定先出队列」性质的表．由于该性质，队列通常也被称为先进先出（first in first out）表，简称 FIFO 表．

实现

数组模拟队列

通常用一个数组模拟一个队列，用两个变量标记队列的首尾．

```
int q[SIZE], ql = 1, qr;
```

队列操作对应的代码如下：

- 插入元素：`q[++qr] = x;`
- 删除元素：`ql++;`
- 访问队首：`q[ql]`
- 访问队尾：`q[qr]`
- 清空队列：`ql = 1; qr = 0;`

Luogu B3616【模板】队列[1] 数组模拟参考实现

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

const int SIZE = 10000 + 5;

struct Queue {
  int q[SIZE], ql, qr;

  Queue() : ql(1), qr(0) {}

  bool empty() { return ql > qr; }

  void push(int x) { q[++qr] = x; }

  void pop() { ++ql; }

  int front() { return q[ql]; }

  int back() { return q[qr]; }

  int size() { return qr - ql + 1; }

  int clear() {
    ql = 1;
    qr = 0;
  }
};

```

```cpp
int main() {
  Queue q;
  int n;
  scanf("%d", &n);
  while (n--) {
    int opt;
    scanf("%d", &opt);
    if (opt == 1) {
      int x;
      scanf("%d", &x);
      q.push(x);
    } else if (opt == 2) {
      if (q.empty())
        printf("ERR_CANNOT_POP\n");
      else
        q.pop();
    } else if (opt == 3) {
      if (q.empty())
        printf("ERR_CANNOT_QUERY\n");
      else
        printf("%d\n", q.front());
    } else
      printf("%d\n", q.size());
  }
  return 0;
}

```

双栈模拟队列

还有一种冷门的方法是使用两个 栈→10.2@p1978 来模拟一个队列.

这种方法使用两个栈 F 和 S 模拟一个队列，其中 F 是队尾的栈，S 代表队首的栈，支持 push（在队尾插入），pop（在队首弹出）操作：

- push：插入到栈 F 中.
- pop：如果 S 非空，让 S 弹栈；否则把 F 的元素倒过来压到 S 中（其实就是一个一个弹出插入，做完后是首尾颠倒的），然后再让 S 弹栈.

容易证明，每个元素只会进入/转移/弹出一次，均摊复杂度 $O(1)$.

Luogu B3616【模板】队列[2] 双栈模拟参考实现

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

struct Queue {
  stack<int> f, s;

  bool empty() { return f.empty() && s.empty(); }

  void push(int x) { f.push(x); }
```

```
11
12   void pop() {
13     if (s.empty())
14       for (; !f.empty(); f.pop()) s.push(f.top());
15     s.pop();
16   }
17
18   int front() {
19     if (s.empty())
20       for (; !f.empty(); f.pop()) s.push(f.top());
21     return s.top();
22   }
23
24   int size() { return f.size() + s.size(); }
25 };
26
27 int main() {
28   Queue q;
29   int n;
30   scanf("%d", &n);
31   while (n--) {
32     int opt;
33     scanf("%d", &opt);
34     if (opt == 1) {
35       int x;
36       scanf("%d", &x);
37       q.push(x);
38     } else if (opt == 2) {
39       if (q.empty())
40         printf("ERR_CANNOT_POP\n");
41       else
42         q.pop();
43     } else if (opt == 3) {
44       if (q.empty())
45         printf("ERR_CANNOT_QUERY\n");
46       else
47         printf("%d\n", q.front());
48     } else
49       printf("%d\n", q.size());
50   }
51   return 0;
52 }
53
```

C++ STL 中的队列

C++ 在 STL 中提供了一个容器 `std::queue`，使用前需要先引入 `<queue>` 头文件.

STL 中对 queue 的定义

```
1 // clang-format off
```

```
template<
    class T,
    class Container = std::deque<T>
> class queue;
```

`T` 为 queue 中要存储的数据类型.

`Container` 为用于存储元素的底层容器类型. 这个容器必须提供通常语义的下列函数:

- `back()`
- `front()`
- `push_back()`
- `pop_front()`

STL 容器 `std::deque` 和 `std::list` 满足这些要求. 如果不指定, 则默认使用 `std::deque` 作为底层容器.

STL 中的 `queue` 容器提供了一众成员函数以供调用. 其中较为常用的有:

- 元素访问
  - `q.front()` 返回队首元素
  - `q.back()` 返回队尾元素
- 修改
  - `q.push()` 在队尾插入元素
  - `q.pop()` 弹出队首元素
- 容量
  - `q.empty()` 队列是否为空
  - `q.size()` 返回队列中元素的数量

此外, `queue` 还提供了一些运算符. 较为常用的是使用赋值运算符 `=` 为 `queue` 赋值, 示例:

```
std::queue<int> q1, q2;

// 向 q1 的队尾插入 1
q1.push(1);

// 将 q1 赋值给 q2
q2 = q1;

// 输出 q2 的队首元素
std::cout << q2.front() << std::endl;
// 输出: 1
```

特殊队列

双端队列

双端队列是指一个可以在队首/队尾插入或删除元素的队列. 相当于是栈与队列功能的结合. 具体地, 双端队列支持的操作有 4 个:

- 在队首插入一个元素
- 在队尾插入一个元素
- 在队首删除一个元素
- 在队尾删除一个元素

数组模拟双端队列的方式与普通队列相同.

同样地，也可以使用双栈模拟队列的思想来维护双端队列，但需注意当某个栈为空时，交替查询队首和队尾将导致均摊分析失效. 考虑在移动时，只将非空栈的一半元素移动到空栈中，并保持队首与队尾栈的性质，这样处理后仍可以做到均摊常数时间的插入和删除.

简要证明

由于插入操作只贡献常数复杂度，现在考虑弹出操作. 假设初始时队列中有 m 个元素，下面我们计算将所有元素全部弹出（无论首尾）的时间复杂度. 则第一次平衡的复杂度是 $O(m)$ 的. 然后两个栈就各有 $\frac{m}{2}$ 个元素. 这时就需要 $O(\frac{m}{2})$ 的时间清空其中一个栈，然后就又可以触发一次复杂度为 $O(\frac{m}{2})$ 的平衡操作，以此类推，直到所有元素被弹出. 因此，这样做的总复杂度是

$$T(m) = T\left(\frac{m}{2}\right) + O(m)$$

根据主定理，解得 $T(m) = O(m)$. 于是，这种维护方式的总复杂度仍是均摊常数的.

Luogu B3656【模板】双端队列 1[3] 参考实现

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

const int M = 1000000 + 5;

struct Deque {
  // 将 stack 的底层容器从 deque 换为 vector 以减少空间常数
  stack<int, vector<int>> f, s;

  bool empty() { return f.empty() && s.empty(); }

  void push_back(int x) { f.push(x); }

  void push_front(int x) { s.push(x); }

  void balance() {
    // 平衡中需要辅助栈实现栈内元素倒置
    stack<int, vector<int>> t;
    if (s.empty()) {
      int n = f.size() / 2;
      for (; f.size() > n; f.pop()) t.push(f.top());
      for (; !t.empty(); t.pop()) s.push(t.top());
      for (; !f.empty(); f.pop()) t.push(f.top());
      f.swap(t);
      if (!f.empty()) s.swap(f);
    } else if (f.empty()) {
      int n = s.size() / 2;
      for (; s.size() > n; s.pop()) t.push(s.top());
      for (; !t.empty(); t.pop()) f.push(t.top());
      for (; !s.empty(); s.pop()) t.push(s.top());
      s.swap(t);
      if (!s.empty()) f.swap(s);
    }
  }
```

```
37
38   void pop_front() {
39     if (s.empty()) balance();
40     s.pop();
41   }
42
43   void pop_back() {
44     if (f.empty()) balance();
45     f.pop();
46   }
47
48   int front() {
49     if (s.empty()) balance();
50     return s.top();
51   }
52
53   int back() {
54     if (f.empty()) balance();
55     return f.top();
56   }
57
58   int size() { return f.size() + s.size(); }
59 };
60
61 int main() {
62   ios::sync_with_stdio(false);
63   cin.tie(nullptr);
64   vector<Deque> q(M);
65   int n;
66   cin >> n;
67   while (n--) {
68     string opt;
69     int a;
70     cin >> opt >> a;
71     if (opt == "push_back") {
72       int x;
73       cin >> x;
74       q[a].push_back(x);
75     } else if (opt == "pop_back") {
76       if (!q[a].empty()) q[a].pop_back();
77     } else if (opt == "push_front") {
78       int x;
79       cin >> x;
80       q[a].push_front(x);
81     } else if (opt == "pop_front") {
82       if (!q[a].empty()) q[a].pop_front();
83     } else if (opt == "size")
84       cout << q[a].size() << "\n";
85     else if (opt == "front") {
86       if (!q[a].empty()) cout << q[a].front() << "\n";
87     } else {
88       if (!q[a].empty()) cout << q[a].back() << "\n";
```

```
89      }
90    }
91    return 0;
92 }
93
```

C++ STL 中的双端队列

C++ 在 STL 中也提供了一个容器 `std::deque`，使用前需要先引入 `<deque>` 头文件.

STL 中对 deque 的定义

```
1 // clang-format off
2 template<
3     class T,
4     class Allocator = std::allocator<T>
5 > class deque;
```

`T` 为 deque 中要存储的数据类型.

`Allocator` 为分配器，此处不做过多说明，一般保持默认即可.

STL 中的 `deque` 容器提供了一众成员函数以供调用. 其中较为常用的有:

- 元素访问
  - `q.front()` 返回队首元素
  - `q.back()` 返回队尾元素
- 修改
  - `q.push_back()` 在队尾插入元素
  - `q.pop_back()` 弹出队尾元素
  - `q.push_front()` 在队首插入元素
  - `q.pop_front()` 弹出队首元素
  - `q.insert()` 在指定位置前插入元素（传入迭代器和元素）
  - `q.erase()` 删除指定位置的元素（传入迭代器）
- 容量
  - `q.empty()` 队列是否为空
  - `q.size()` 返回队列中元素的数量

此外，`deque` 还提供了一些运算符. 其中较为常用的有:

- 使用赋值运算符 `=` 为 `deque` 赋值，类似 `queue`.
- 使用 `[]` 访问元素，类似 `vector`.

`<queue>` 头文件中还提供了优先队列 `std::priority_queue`，因其与 堆→10.7.1@p2025 更为相似，在此不作过多介绍.

Python 中的双端队列

在 Python 中，双端队列的容器由 `collections.deque` 提供.

示例如下:

实现

```python
from collections import deque

# 新建一个 deque，并初始化内容为 [1, 2, 3]
queue = deque([1, 2, 3])

# 在队尾插入元素 4
queue.append(4)

# 在队首插入元素 0
queue.appendleft(0)

# 访问队列
# >>> queue
# deque([0, 1, 2, 3, 4])
```

循环队列

使用数组模拟队列会导致一个问题：随着时间的推移，整个队列会向数组的尾部移动，一旦到达数组的最末端，即使数组的前端还有空闲位置，再进行入队操作也会导致溢出（这种数组里实际有空闲位置而发生了上溢的现象被称为「假溢出」）.

解决假溢出的办法是采用循环的方式来组织存放队列元素的数组，即将数组下标为 0 的位置看做是最后一个位置的后继.（数组下标为 `x` 的元素，它的后继为 `(x + 1) % SIZE`）. 这样就形成了循环队列.

参考资料

1. std::queue - zh.cppreference.com[4]
2. std::deque - zh.cppreference.com[5]

外部参考资料

[1] Luogu B3616【模板】队列

[2] Luogu B3616【模板】队列

[3] Luogu B3656【模板】双端队列 1

[4] std::queue - zh.cppreference.com

[5] std::deque - zh.cppreference.com

10.4 链表

本页面将简要介绍链表.

引入

链表是一种用于存储数据的数据结构，通过如链条一般的指针来连接元素. 它的特点是插入与删除数据十分方便，但寻找与读取数据的表现欠佳.

与数组的区别

链表和数组都可用于存储数据. 与链表不同，数组将所有元素按次序依次存储. 不同的存储结构令它们有了不同的优势：

链表因其链状的结构，能方便地删除、插入数据，操作次数是 $O(1)$. 但也因为这样，寻找、读取数据的效率不如数组高，在随机访问数据中的操作次数是 $O(n)$.

数组可以方便地寻找并读取数据，在随机访问中操作次数是 $O(1)$．但删除、插入的操作次数是 $O(n)$ 次．

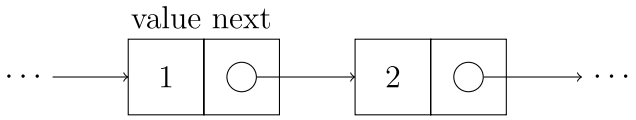
构建链表

Tip

构建链表时，使用指针的部分比较抽象，光靠文字描述和代码可能难以理解，建议配合作图来理解．

单向链表

单向链表中包含数据域和指针域，其中数据域用于存放数据，指针域用来连接当前结点和下一节点．



实现

- C++

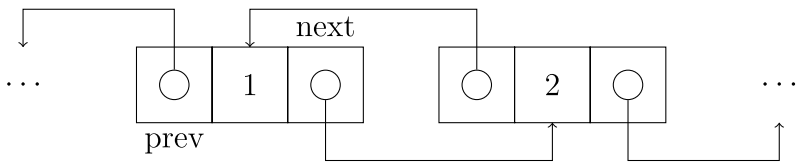
```cpp
struct Node {
  int value;
  Node *next;
};
```

- Python

```python
class Node:
    def __init__(self, value=None, next=None):
        self.value = value
        self.next = next
```

双向链表

双向链表中同样有数据域和指针域．不同之处在于，指针域有左右（或上一个、下一个）之分，用来连接上一个结点、当前结点、下一个结点．



实现

- C++

```cpp
struct Node {
  int value;
  Node *left;
  Node *right;
```

```
};
```

- Python

```python
class Node:
    def __init__(self, value=None, left=None, right=None):
        self.value = value
        self.left = left
        self.right = right
```

向链表中插入（写入）数据

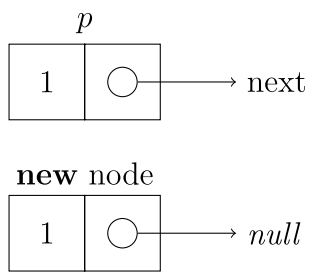
单向链表

流程大致如下：

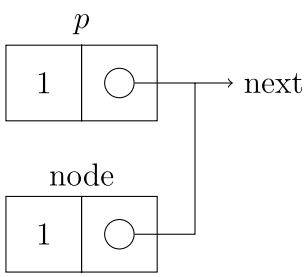
1. 初始化待插入的数据 `node`；
2. 将 `node` 的 `next` 指针指向 `p` 的下一个结点；
3. 将 `p` 的 `next` 指针指向 `node`.

具体过程可参考下图：

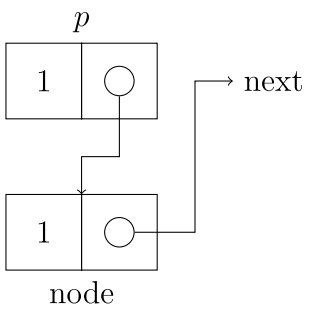
1.



2.



3.



代码实现如下：

实现

- C++

```cpp
void insertNode(int i, Node *p) {
```

```
    Node *node = new Node;
    node->value = i;
    node->next = p->next;
    p->next = node;
}
```

- Python

```
def insertNode(i, p):
    node = Node()
    node.value = i
    node.next = p.next
    p.next = node
```

单向循环链表

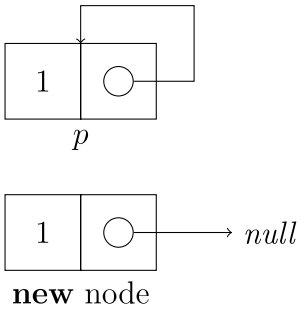
将链表的头尾连接起来，链表就变成了循环链表．由于链表首尾相连，在插入数据时需要判断原链表是否为空：为空则自身循环，不为空则正常插入数据．

大致流程如下：

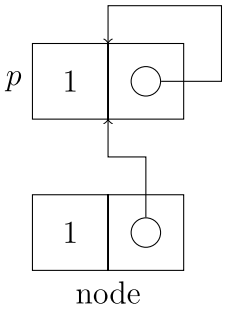
1. 初始化待插入的数据 `node`；
2. 判断给定链表 `p` 是否为空；
3. 若为空，则将 `node` 的 `next` 指针和 `p` 都指向自己；
4. 否则，将 `node` 的 `next` 指针指向 `p` 的下一个结点；
5. 将 `p` 的 `next` 指针指向 `node`.

具体过程可参考下图：

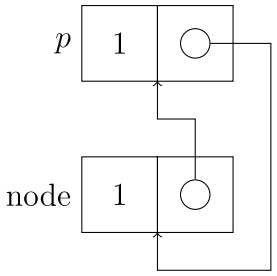
1.



2.



3.



代码实现如下：

实现

- C++

```cpp
void insertNode(int i, Node *p) {
  Node *node = new Node;
  node->value = i;
  node->next = NULL;
  if (p == NULL) {
    p = node;
    node->next = node;
  } else {
    node->next = p->next;
    p->next = node;
  }
}
```

- Python

```python
def insertNode(i, p):
    node = Node()
    node.value = i
    node.next = None
    if p == None:
        p = node
        node.next = node
    else:
        node.next = p.next
        p.next = node
```

双向循环链表

在向双向循环链表插入数据时，除了要判断给定链表是否为空外，还要同时修改左、右两个指针.

大致流程如下：

1. 初始化待插入的数据 `node`；
2. 判断给定链表 `p` 是否为空；
3. 若为空，则将 `node` 的 `left` 和 `right` 指针，以及 `p` 都指向自己；
4. 否则，将 `node` 的 `left` 指针指向 `p`
5. 将 `node` 的 `right` 指针指向 `p` 的右结点；
6. 将 `p` 右结点的 `left` 指针指向 `node`；
7. 将 `p` 的 `right` 指针指向 `node`.

代码实现如下：

实现

- C++

```cpp
void insertNode(int i, Node *p) {
  Node *node = new Node;
  node->value = i;
  if (p == NULL) {
    p = node;
    node->left = node;
    node->right = node;
  } else {
    node->left = p;
    node->right = p->right;
    p->right->left = node;
    p->right = node;
  }
}
```

- Python

```python
def insertNode(i, p):
    node = Node()
    node.value = i
    if p == None:
        p = node
        node.left = node
        node.right = node
    else:
        node.left = p
        node.right = p.right
        p.right.left = node
        p.right = node
```

从链表中删除数据

单向（循环）链表

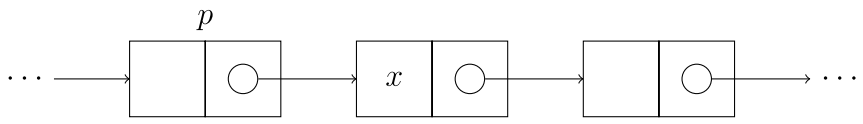
设待删除结点为 `p`，从链表中删除它时，将 `p` 的下一个结点 `p->next` 的值覆盖给 `p` 即可，与此同时更新 `p` 的下下个结点.

流程大致如下：

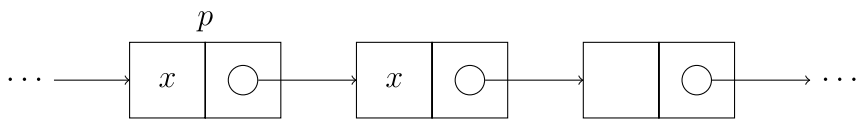
1. 将 `p` 下一个结点的值赋给 `p`，以抹掉 `p->value`；
2. 新建一个临时结点 `t` 存放 `p->next` 的地址；
3. 将 `p` 的 `next` 指针指向 `p` 的下下个结点，以抹掉 `p->next`；
4. 删除 `t`. 此时虽然原结点 `p` 的地址还在使用，删除的是原结点 `p->next` 的地址，但 `p` 的数据被 `p->next` 覆盖，`p` 名存实亡.

具体过程可参考下图：

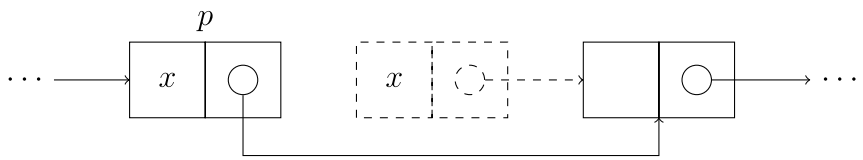
1.



2.



3.



代码实现如下：

实现

- C++

```cpp
void deleteNode(Node *p) {
  p->value = p->next->value;
  Node *t = p->next;
  p->next = p->next->next;
  delete t;
}
```

- Python

```python
def deleteNode(p):
    p.value = p.next.value
    p.next = p.next.next
```

双向循环链表

流程大致如下：

1. 将 p 左结点的右指针指向 p 的右节点；
2. 将 p 右结点的左指针指向 p 的左节点；
3. 新建一个临时结点 t 存放 p 的地址；
4. 将 p 的右节点地址赋给 p，以避免 p 变成悬垂指针；
5. 删除 t.

代码实现如下：

实现

- C++

```cpp
void deleteNode(Node *&p) {
  p->left->right = p->right;
  p->right->left = p->left;
  Node *t = p;
  p = p->right;
  delete t;
}
```

- Python

```python
def deleteNode(p):
    p.left.right = p.right
```

```
    p.right.left = p.left
    p = p.right
```

技巧

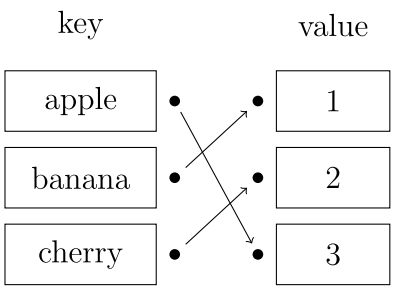
异或链表

异或链表（XOR Linked List）本质上还是 **双向链表**，但它利用按位异或的值，仅使用一个指针的内存大小便可以实现双向链表的功能.

我们在结构 `Node` 中定义 `lr = left ^ right`，即前后两个元素地址的 **按位异或值**. 正向遍历时用前一个元素的地址异或当前节点的 `lr` 可得到后一个元素的地址，反向遍历时用后一个元素的地址异或当前节点的 `lr` 又可得到前一个的元素地址. 这样一来，便可以用一半的内存实现双向链表同样的功能.

10.5 哈希表

引入



哈希表又称散列表，一种以「key-value」形式存储数据的数据结构. 所谓以「key-value」形式存储数据，是指任意的键值 key 都唯一对应到内存中的某个位置. 只需要输入查找的键值，就可以快速地找到其对应的 value. 可以把哈希表理解为一种高级的数组，这种数组的下标可以是很大的整数，浮点数，字符串甚至结构体.

哈希函数

要让键值对应到内存中的位置，就要为键值计算索引，也就是计算这个数据应该放到哪里. 这个根据键值计算索引的函数就叫做哈希函数，也称散列函数. 举个例子，如果键值是一个人的身份证号码，哈希函数就可以是号码的后四位，当然也可以是号码的前四位. 生活中常用的「手机尾号」也是一种哈希函数. 在实际的应用中，键值可能是更复杂的东西，比如浮点数、字符串、结构体等，这时候就要根据具体情况设计合适的哈希函数. 哈希函数应当易于计算，并且尽量使计算出来的索引均匀分布.

能为 key 计算索引之后，我们就可以知道每个键值对应的值 value 应该放在哪里了. 假设我们用数组 a 存放数据，哈希函数是 f，那键值对 `(key, value)` 就应该放在 `a[f(key)]` 上. 不论键值是什么类型，范围有多大，`f(key)` 都是在可接受范围内的整数，可以作为数组的下标.

在 OI 中，最常见的情况应该是键值为整数的情况. 当键值的范围比较小的时候，可以直接把键值作为数组的下标，但当键值的范围比较大，比如以 10^9 范围内的整数作为键值的时候，就需要用到哈希表. 一般把键值模一个较大的质数作为索引，也就是取 $f(x) = x \bmod M$ 作为哈希函数.

另一种比较常见的情况是 key 为字符串的情况，由于不支持以字符串作为数组下标，并且将字符串转化成数字存储也可以避免多次进行字符串比较. 所以在 OI 中，一般不直接把字符串作为键值，而是先算

出字符串的哈希值，再把其哈希值作为键值插入到哈希表里．关于字符串的哈希值，我们一般采用进制的思想，将字符串想象成一个 127 进制的数．那么，对于每一个长度为 n 的字符串 s，就有：

$$x = s_0 \cdot 127^0 + s_1 \cdot 127^1 + s_2 \cdot 127^2 + ... + s_n \cdot 127^n$$

我们可以将得到的 x 对 2^{64}（即 `unsigned long long` 的最大值）取模．这样 `unsigned long long` 的自然溢出就等价于取模操作了．可以使操作更加方便．

这种方法虽然简单，但并不是完美的．可以构造数据使这种方法发生冲突（即两个字符串的 x 对 2^{64} 取模后的结果相同）．

我们可以使用双哈希的方法：选取两个大质数 a, b．当且仅当两个字符串的哈希值对 a 和对 b 取模都相等时，我们才认为这两个字符串相等．这样可以大大降低哈希冲突的概率．

冲突

如果对于任意的键值，哈希函数计算出来的索引都不相同，那只用根据索引把 `(key, value)` 放到对应的位置就行了．但实际上，常常会出现两个不同的键值，他们用哈希函数计算出来的索引是相同的．这时候就需要一些方法来处理冲突．在 OI 中，最常用的方法是拉链法．

拉链法

拉链法也称开散列法（open hashing）．

拉链法是在每个存放数据的地方开一个链表，如果有多个键值索引到同一个地方，只用把他们都放到那个位置的链表里就行了．查询的时候需要把对应位置的链表整个扫一遍，对其中的每个数据比较其键值与查询的键值是否一致．如果索引的范围是 $1...M$，哈希表的大小为 N，那么一次插入/查询需要进行期望 $O(\frac{N}{M})$ 次比较．

实现

- C++

```cpp
constexpr int SIZE = 1000000;
constexpr int M = 999997;

struct HashTable {
  struct Node {
    int next, value, key;
  } data[SIZE];

  int head[M], size;

  int f(int key) { return (key % M + M) % M; }

  int get(int key) {
    for (int p = head[f(key)]; p; p = data[p].next)
      if (data[p].key == key) return data[p].value;
    return -1;
  }

  int modify(int key, int value) {
    for (int p = head[f(key)]; p; p = data[p].next)
      if (data[p].key == key) return data[p].value = value;
  }

  int add(int key, int value) {
    if (get(key) != -1) return -1;
```

```
    data[++size] = Node{head[f(key)], value, key};
    head[f(key)] = size;
    return value;
  }
};
```

- Python

```
M = 999997
SIZE = 1000000


class Node:
    def __init__(self, next=None, value=None, key=None):
        self.next = next
        self.value = value
        self.key = key


data = [Node() for _ in range(SIZE)]
head = [0] * M
size = 0


def f(key):
    return key % M


def get(key):
    p = head[f(key)]
    while p:
        if data[p].key == key:
            return data[p].value
        p = data[p].next
    return -1


def modify(key, value):
    p = head[f(key)]
    while p:
        if data[p].key == key:
            data[p].value = value
            return data[p].value
        p = data[p].next


def add(key, value):
    if get(key) != -1:
        return -1
    size = size + 1
    data[size] = Node(head[f(key)], value, key)
    head[f(key)] = size
    return value
```

这里再提供一个封装过的模板，可以像 map 一样用，并且较短

```
struct hash_map {  // 哈希表模板

  struct data {
    long long u;
    int v, nex;
  };  // 前向星结构

  data e[SZ << 1];  // SZ 是 const int 表示大小
  int h[SZ], cnt;

  int hash(long long u) { return (u % SZ + SZ) % SZ; }

  // 这里使用 (u % SZ + SZ) % SZ 而非 u % SZ 的原因是
  // C++ 中的 % 运算无法将负数转为正数

  int& operator[](long long u) {
    int hu = hash(u);  // 获取头指针
    for (int i = h[hu]; i; i = e[i].nex)
      if (e[i].u == u) return e[i].v;
    return e[++cnt] = data{u, -1, h[hu]}, h[hu] = cnt, e[cnt].v;
  }

  hash_map() {
    cnt = 0;
    memset(h, 0, sizeof(h));
  }
};
```

在这里，hash 函数是针对键值的类型设计的，并且返回一个链表头指针用于查询．在这个模板中我们写了一个键值对类型为 `(long long, int)` 的 hash 表，并且在查询不存在的键值时返回 -1．函数 `hash_map()` 用于在定义时初始化．

闭散列法

闭散列方法把所有记录直接存储在散列表中，如果发生冲突则根据某种方式继续进行探查．

比如线性探查法：如果在 d 处发生冲突，就依次检查 `d + 1`，`d + 2`……

实现

```
constexpr int N = 360007;  // N 是最大可以存储的元素数量

class Hash {
 private:
  int keys[N];
  int values[N];

 public:
  Hash() { memset(values, 0, sizeof(values)); }

  int& operator[](int n) {
    // 返回一个指向对应 Hash[Key] 的引用
    // 修改成不为 0 的值 0 时候视为空
    int idx = (n % N + N) % N, cnt = 1;
    while (keys[idx] != n && values[idx] != 0) {
```

```
      idx = (idx + cnt * cnt) % N;
      cnt += 1;
    }
    keys[idx] = n;
    return values[idx];
  }
};
```

例题

「JLOI2011」不重复数字[1]

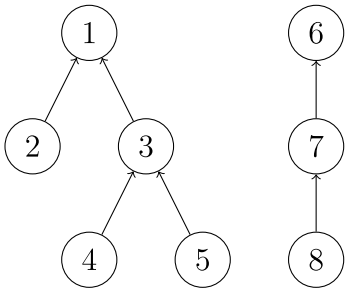
外部参考资料

[1] 「JLOI2011」不重复数字

10.6 并查集

10.6.1 并查集

Authors: HeRaNO, JuicyMio, Xeonacid, sailordiary, ouuan, Pig-Eat-Earth



引入

并查集是一种用于管理元素所属集合的数据结构，实现为一个森林，其中每棵树表示一个集合，树中的节点表示对应集合中的元素.

顾名思义，并查集支持两种操作：

- 合并（Unite）：合并两个元素所属集合（合并对应的树）.
- 查询（Find）：查询某个元素所属集合（查询对应的树的根节点），这可以用于判断两个元素是否属于同一集合.

并查集在经过修改后可以支持单个元素的删除、移动或维护树上的边权. 使用动态开点线段树还可以实现 可持久化并查集→10.17.2@p2424.

> **Warning**
>
> 并查集无法以较低复杂度实现集合的分离.

初始化

初始时，每个元素都位于一个单独的集合，表示为一棵只有根节点的树. 方便起见，我们将根节点的父亲设为自己.

实现

- C++

```cpp
struct dsu {
  vector<size_t> pa;

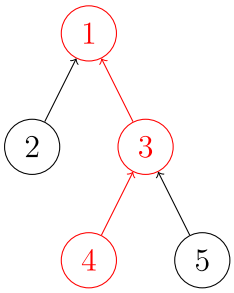
  explicit dsu(size_t size) : pa(size) { iota(pa.begin(), pa.end(), 0); }
};
```

- Python

```python
class Dsu:
    def __init__(self, size):
        self.pa = list(range(size))
```

查询

我们需要沿着树向上移动，直至找到根节点.



实现

- C++

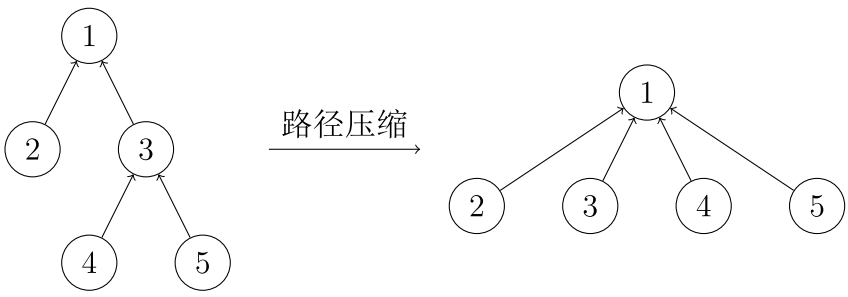
```cpp
size_t dsu::find(size_t x) { return pa[x] == x ? x : find(pa[x]); }
```

- Python

```python
def find(self, x):
    return x if self.pa[x] == x else self.find(self.pa[x])
```

路径压缩

查询过程中经过的每个元素都属于该集合，我们可以将其直接连到根节点以加快后续查询.



实现

- C++

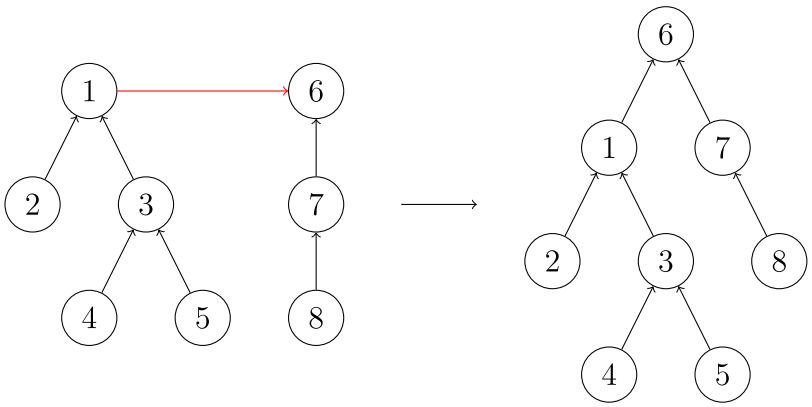
```cpp
size_t dsu::find(size_t x) { return pa[x] == x ? x : pa[x] =
find(pa[x]); }
```

- Python

```python
def find(self, x):
    if self.pa[x] != x:
        self.pa[x] = self.find(self.pa[x])
    return self.pa[x]
```

合并

要合并两棵树，我们只需要将一棵树的根节点连到另一棵树的根节点.



实现

- C++

```cpp
void dsu::unite(size_t x, size_t y) { pa[find(x)] = find(y); }
```

- Python

```python
def unite(self, x, y):
    self.pa[self.find(x)] = self.find(y)
```

启发式合并

合并时，选择哪棵树的根节点作为新树的根节点会影响未来操作的复杂度. 我们可以将节点较少或深度较小的树连到另一棵，以免发生退化.

具体复杂度讨论

由于需要我们支持的只有集合的合并、查询操作，当我们需要将两个集合合二为一时，无论将哪一个集合连接到另一个集合的下面，都能得到正确的结果. 但不同的连接方法存在时间复杂度的差异. 具体来说，如果我们将一棵点数与深度都较小的集合树连接到一棵更大的集合树下，显然相比于另一种连接方案，接下来执行查找操作的用时更小（也会带来更优的最坏时间复杂度）.

当然，我们不总能遇到恰好如上所述的集合——点数与深度都更小. 鉴于点数与深度这两个特征都很容易维护，我们常常从中择一，作为估价函数. 而无论选择哪一个，时间复杂度都为 $O(m\alpha(m,n))$，具体的证明可参见 References 中引用的论文.

在算法竞赛的实际代码中，即便不使用启发式合并，代码也往往能够在规定时间内完成任务. 在 Tarjan 的论文[1]中，证明了不使用启发式合并、只使用路径压缩的最坏时间复杂度是 $O(m\log n)$. 在姚期智的论文[2]中，证明了不使用启发式合并、只使用路径压缩，在平均情况下，时间复杂度依然是 $O(m\alpha(m,n))$.

如果只使用启发式合并，而不使用路径压缩，时间复杂度为 $O(m\log n)$. 由于路径压缩单次合并可能造成大量修改，有时路径压缩并不适合使用. 例如，在可持久化并查集、线段树分治 + 并查集中，一般使用只启发式合并的并查集.

按节点数合并的参考实现：（注意需要调整初始化方法）

实现

- C++

```cpp
struct dsu {
  vector<size_t> pa, size;

  explicit dsu(size_t size_) : pa(size_), size(size_, 1) {
    iota(pa.begin(), pa.end(), 0);
  }

  void unite(size_t x, size_t y) {
    x = find(x), y = find(y);
    if (x == y) return;
    if (size[x] < size[y]) swap(x, y);
    pa[y] = x;
    size[x] += size[y];
  }
};
```

- Python

```python
class Dsu:
    def __init__(self, size):
        self.pa = list(range(size))
        self.size = [1] * size

    def unite(self, x, y):
        x, y = self.find(x), self.find(y)
        if x == y:
            return
        if self.size[x] < self.size[y]:
            x, y = y, x
        self.pa[y] = x
        self.size[x] += self.size[y]
```

[1]Tarjan, R. E., & Van Leeuwen, J. (1984). Worst-case analysis of set union algorithms. Journal of the ACM (JACM), 31(2), 245-281.ResearchGate PDF[1]

[2]Yao, A. C. (1985). On the expected performance of path compression algorithms.SIAM Journal on Computing, 14(1), 129-133.[2]

参考实现

带有路径压缩、按节点数合并的并查集的完整实现如下所示：

模板题 Luogu P3367【模板】并查集[3] 参考实现

- C++

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

struct DSU {
  std::vector<size_t> pa, size;

  explicit DSU(size_t size_) : pa(size_), size(size_, 1) {
    std::iota(pa.begin(), pa.end(), 0);
  }

  size_t find(size_t x) { return pa[x] == x ? x : pa[x] = find(pa[x]); }

  void unite(size_t x, size_t y) {
    x = find(x), y = find(y);
    if (x == y) return;
    if (size[x] < size[y]) std::swap(x, y);
    pa[y] = x;
    size[x] += size[y];
  }
};

int main() {
  int n, m;
  std::cin >> n >> m;
  DSU dsu(n + 1);
  for (; m; --m) {
    int z, x, y;
    std::cin >> z >> x >> y;
    if (z == 1) {
      dsu.unite(x, y);
    } else {
      std::cout << (dsu.find(x) == dsu.find(y) ? 'Y' : 'N') << '\n';
    }
  }
  return 0;
}

```

- Python

```python
class Dsu:
    def __init__(self, size):
        self.pa = list(range(size))
        self.size = [1] * size

```

```python
    def find(self, x):
        if self.pa[x] != x:
            self.pa[x] = self.find(self.pa[x])
        return self.pa[x]

    def unite(self, x, y):
        x, y = self.find(x), self.find(y)
        if x == y:
            return
        if self.size[x] < self.size[y]:
            x, y = y, x
        self.pa[y] = x
        self.size[x] += self.size[y]


if __name__ == "__main__":
    n, m = map(int, input().split())
    dsu = Dsu(n + 1)
    for _ in range(m):
        z, x, y = map(int, input().split())
        if z == 1:
            dsu.unite(x, y)
        else:
            print("Y" if dsu.find(x) == dsu.find(y) else "N")

```

复杂度

同时使用路径压缩和启发式合并之后，并查集的每个操作平均时间仅为 $O(\alpha(n))$. 其中，α 为阿克曼函数的反函数，增长极其缓慢. 也就是说，并查集单次操作的平均运行时间可以认为是一个很小的常数. 时间复杂度的证明在 这个页面→10.6.2@p2021 中.

反 Ackermann 函数

Ackermann 函数[4] $A(m,n)$ 的定义是这样的:

$$A(m,n)=\begin{cases} n+1 & \text{if } m=0 \\ A(m-1,1) & \text{if } m>0 \text{ and } n=0 \\ A(m-1,A(m,n-1)) & \text{otherwise} \end{cases}$$

而反 Ackermann 函数 $\alpha(n)$ 的定义是 Ackermann 函数的反函数，即为最大的整数 m 使得 $A(m,m)\leqslant n$.

并查集的空间复杂度显然为 $O(n)$.

拓展操作

在普通的并查集的基础上，还可以做一系列修改使之支持更多的操作或维护更复杂的信息.

带删除并查集

普通的并查集无法支持删除操作，是因为删除一个节点的时候，不可避免地会将以它为根的子树上所有节点都删除. 为了解决这一问题，在带删除操作的并查集中，可以通过建立虚点的方法保证所有实际存储数据的节点总是叶子节点. 为此，需要在初始化时，就为每个数据节点都建立一个虚点，并将数据节点的父节点设置为该虚点. 由于每次合并两个集合时，都只会将两个集合的树根连接，所以，从始至终只有虚点会有子节点. 这就保证了删除一个节点时，不会误删其他节点.

注意，删除单个节点后，需要重新为该节点建立一个虚点作为其父节点；否则，无法正确执行后续的合并和删除操作.

模板题 SPOJ JMFILTER - Junk-Mail Filter[5] 参考实现

- C++

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

struct DSU {
  size_t id;
  std::vector<size_t> pa, size;

  explicit DSU(size_t size_, size_t m)
      : id(size_ * 2), pa(size_ * 2 + m), size(size_ * 2 + m, 1) {
    // size 的前半段其实没有使用，只是为了让下标计算更简单
    std::iota(pa.begin(), pa.begin() + size_,
              size_);  // 令 i 指向虚点 i + size_
    std::iota(pa.begin() + size_, pa.end(), size_);  // 所有虚点指向它自身
  }

  size_t find(size_t x) { return pa[x] == x ? x : pa[x] = find(pa[x]); }

  void unite(size_t x, size_t y) {
    x = find(x), y = find(y);
    if (x == y) return;
    if (size[x] < size[y]) std::swap(x, y);
    pa[y] = x;
    size[x] += size[y];
  }

  void erase(size_t x) {
    size_t y = find(x);
    --size[y];
    pa[x] = id++;
  }
};

int main() {
  int n, m, case_id = 0;
  while ((std::cin >> n >> m), n) {
    DSU dsu(n, m);
    for (; m; --m) {
      char ch;
      std::cin >> ch;
      if (ch == 'M') {
        int x, y;
        std::cin >> x >> y;
        dsu.unite(x, y);
      } else if (ch == 'S') {
```

```
        int x;
        std::cin >> x;
        dsu.erase(x);
      }
    }
    int res = 0;
    for (int i = n; i < dsu.id; ++i) {
      if (dsu.size[i] && i == dsu.find(i)) {
        ++res;
      }
    }
    std::cout << "Case #" << (++case_id) << ": " << res << '\n';
  }
  return 0;
}

```

- Python

```
# 此代码仅作示意，因超时无法通过原题
class Dsu:
    def __init__(self, size, m):
        self.id = size * 2
        # 令 i 指向虚点 i + size_，所有虚点指向它自身
        self.pa = list(range(size, size * 2)) + list(range(size, size * 2 + m))
        # size 的前半段其实没有使用，只是为了让下标计算更简单
        self.size = [1] * (size * 2 + m)

    def find(self, x):
        if self.pa[x] != x:
            self.pa[x] = self.find(self.pa[x])
        return self.pa[x]

    def unite(self, x, y):
        x, y = self.find(x), self.find(y)
        if x == y:
            return
        if self.size[x] < self.size[y]:
            x, y = y, x
        self.pa[y] = x
        self.size[x] += self.size[y]

    def erase(self, x):
        y = self.find(x)
        self.size[y] -= 1
        self.pa[x] = self.id
        self.id += 1


if __name__ == "__main__":
    case_id = 0
    while True:
```

```
        n, m = map(int, input().split())
        if not n:
            break
        dsu = Dsu(n, m)
        for _ in range(m):
            op = input().split()
            if op[0] == "M":
                x = int(op[1])
                y = int(op[2])
                dsu.unite(x, y)
            elif op[0] == "S":
                x = int(op[1])
                dsu.erase(x)
        res = 0
        for i in range(n, dsu.id):
            if dsu.size[i] and i == dsu.find(i):
                res += 1
        case_id += 1
        print(f"Case #{case_id}: {res}")
        input()

```

类似的方法还可以用于实现在集合间移动单个元素. 实现细节详见例题.

带权并查集

我们还可以在并查集的边上定义某种权值和这种权值在路径压缩时产生的运算, 从而解决更多的问题. 比如对于经典的「NOI2001」食物链, 我们可以在边权上维护模 3 意义下的加法群. 对于这类维护模意义下边权且模数很小的问题, 还可以通过将并查集的单个点拆分为多个状态的方式来解决. 这种特殊情形下的技巧, 也称为「种类并查集」或「拓展域并查集」. 后文会通过例题来说明这些做法.

为了维护并查集中的边权, 需要将边权下放到子节点中存储. 因此, 每个节点存储的都是它到它的父节点之间的边权. 只有当一个节点的父节点发生变化时, 才需要相应地调整边权. 一般情形中, 这可能发生在路径压缩和合并两个节点时. 例如, 如果边权是当前节点与父节点之间的距离, 那么, 在路径压缩时, 每次将当前节点的父节点替换为根节点, 都需要将父节点到根节点的距离加到当前节点存储的边权上; 类似地, 在合并两个节点所在集合时, 需要计算两个根节点之间新连接的边的权值.

模板题 Library Checker - Unionfind with Potential[6] 参考实现

- C++

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

constexpr int M = 998244353;

struct DSU {
  std::vector<size_t> pa, size, dist;

  explicit DSU(size_t size_) : pa(size_), size(size_, 1), dist(size_) {
    std::iota(pa.begin(), pa.end(), 0);
```

```cpp
  }

  size_t find(size_t x) {
    if (pa[x] == x) return x;
    size_t y = find(pa[x]);
    (dist[x] += dist[pa[x]]) %= M;
    return pa[x] = y;
  }

  bool unite(size_t x, size_t y, int d) {
    find(x), find(y);
    (d += M - dist[y]) %= M;
    (d += dist[x]) %= M;
    x = pa[x], y = pa[y];
    if (x == y) return d == 0;
    if (size[x] < size[y]) {
      std::swap(x, y);
      d = (M - d) % M;
    }
    pa[y] = x;
    size[x] += size[y];
    dist[y] = d;
    return true;
  }

  int check(size_t x, size_t y) {
    find(x), find(y);
    if (pa[x] != pa[y]) return -1;
    return (dist[y] - dist[x] + M) % M;
  }
};

int main() {
  int n, m;
  std::cin >> n >> m;
  DSU dsu(n);
  for (; m; --m) {
    int op;
    std::cin >> op;
    if (op) {
      int u, v;
      std::cin >> u >> v;
      std::cout << dsu.check(u, v) << '\n';
    } else {
      int u, v, x;
      std::cin >> u >> v >> x;
      std::cout << dsu.unite(u, v, x) << '\n';
    }
  }
  return 0;
}

```

- Python

```python
M = 998244353


class DSU:
    def __init__(self, size: int):
        self.pa = list(range(size))
        self.size = [1] * size
        self.dist = [0] * size

    def find(self, x: int) -> int:
        if self.pa[x] == x:
            return x
        y = self.find(self.pa[x])
        self.dist[x] = (self.dist[x] + self.dist[self.pa[x]]) % M
        self.pa[x] = y
        return y

    def unite(self, x: int, y: int, d: int) -> bool:
        self.find(x)
        self.find(y)
        d = (d + M - self.dist[y]) % M
        d = (d + self.dist[x]) % M
        x, y = self.pa[x], self.pa[y]
        if x == y:
            return d == 0
        if self.size[x] < self.size[y]:
            x, y = y, x
            d = (M - d) % M
        self.pa[y] = x
        self.size[x] += self.size[y]
        self.dist[y] = d
        return True

    def check(self, x: int, y: int) -> int:
        self.find(x)
        self.find(y)
        if self.pa[x] != self.pa[y]:
            return -1
        return (self.dist[y] - self.dist[x] + M) % M


if __name__ == "__main__":
    n, m = map(int, input().split())
    dsu = DSU(n)
    for _ in range(m):
        op, *rest = map(int, input().split())
        if op:
            u, v = rest
            print(dsu.check(u, v))
        else:
            u, v, x = rest
```

```
            print(int(dsu.unite(u, v, x)))

```

例题

算法竞赛中，直接考察并查集的题目大多都需要针对题目设计特殊的结构.

UVa11987 Almost Union-Find[7]

实现类似并查集的数据结构，支持以下操作：

1. 合并两个元素所属集合.
2. 将单个元素移动到另一个元素所在的集合.
3. 查询某个元素所属集合的大小及元素和.

解答

这道题目中，操作 1 和操作 3 都容易处理，难点在于操作 2. 假定要将元素 x 移动到元素 y 所在的集合. 在普通的并查集中，直接将元素 x 的父亲设为元素 y 所在集合的根节点是不行的，因为这样会将元素 x 所在子树的元素都一起移动. 针对这个问题，解决方法就是保证元素 x 没有子节点. 为此，在建立并查集时为每个元素 x 都建立一个虚点 $\tilde{x}$，并将元素 x 的父亲指向对应的虚点 $\tilde{x}$. 这样，在合并两个集合的时候，因为总是将一个树根连接到另一个树根，而树根又全部是虚点，所以，只有虚点会有子节点，而所有实际存储元素的点都没有子节点. 此时，要移动元素，就容易实现得多.

参考实现

- C++

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

using namespace std;

struct dsu {
  vector<size_t> pa, size, sum;

  explicit dsu(size_t size_)
      : pa(size_ * 2), size(size_ * 2, 1), sum(size_ * 2) {
    // size 与 sum 的前半段其实没有使用，只是为了让下标计算更简单
    iota(pa.begin(), pa.begin() + size_, size_);
    iota(pa.begin() + size_, pa.end(), size_);
    iota(sum.begin() + size_, sum.end(), 0);
  }

  void unite(size_t x, size_t y) {
    x = find(x), y = find(y);
    if (x == y) return;
    if (size[x] < size[y]) swap(x, y);
    pa[y] = x;
```

```
    size[x] += size[y];
    sum[x] += sum[y];
  }

  void move(size_t x, size_t y) {
    auto fx = find(x), fy = find(y);
    if (fx == fy) return;
    pa[x] = fy;
    --size[fx], ++size[fy];
    sum[fx] -= x, sum[fy] += x;
  }

  size_t find(size_t x) { return pa[x] == x ? x : pa[x] =
find(pa[x]); }
};

int main() {
  size_t n, m, op, x, y;
  while (cin >> n >> m) {
    dsu dsu(n + 1);  // 元素范围是 1..n
    while (m--) {
      cin >> op;
      switch (op) {
        case 1:
          cin >> x >> y;
          dsu.unite(x, y);
          break;
        case 2:
          cin >> x >> y;
          dsu.move(x, y);
          break;
        case 3:
          cin >> x;
          x = dsu.find(x);
          cout << dsu.size[x] << ' ' << dsu.sum[x] << '\n';
          break;
        default:
          assert(false);  // not reachable
      }
    }
  }
  return 0;
}

```

- Python

```
class Dsu:
    def __init__(self, size):
        # size 与 sum 的前半段其实没有使用，只是为了让下标计算更简单
        self.pa = list(range(size, size * 2)) * 2
        self.size = [1] * size * 2
        self.sum = list(range(size)) * 2
```

```python

    def unite(self, x, y):
        x, y = self.find(x), self.find(y)
        if x == y:
            return
        if self.size[x] < self.size[y]:
            x, y = y, x
        self.pa[y] = x
        self.size[x] += self.size[y]
        self.sum[x] += self.sum[y]

    def move(self, x, y):
        fx, fy = self.find(x), self.find(y)
        if fx == fy:
            return
        self.pa[x] = fy
        self.size[fx] -= 1
        self.size[fy] += 1
        self.sum[fx] -= x
        self.sum[fy] += x

    def find(self, x):
        if self.pa[x] != x:
            self.pa[x] = self.find(self.pa[x])
        return self.pa[x]


if __name__ == "__main__":
    while True:
        try:
            n, m = map(int, input().split())
            dsu = Dsu(n + 1)  # 元素范围是 1..n
            for _ in range(m):
                op_x_y = list(map(int, input().split()))
                op = op_x_y[0]
                if op == 1:
                    dsu.unite(op_x_y[1], op_x_y[2])
                elif op == 2:
                    dsu.move(op_x_y[1], op_x_y[2])
                elif op == 3:
                    x = dsu.find(op_x_y[1])
                    print(dsu.size[x], dsu.sum[x])
        except EOFError:
            break

```

Luogu P2024「NOI2011」食物链[8]

动物王国中有三类动物 A, B, C，这三类动物的食物链构成了有趣的环形．A 吃 B，B 吃 C，C 吃 A.

现有 N 个动物，以 $1 \sim N$ 编号．每个动物都是 A, B, C 中的一种，但是我们并不知道它到底是哪一种．

有人用两种说法对这 N 个动物所构成的食物链关系进行描述：

- 第一种说法是 `1 X Y`，表示 X 和 Y 是同类．
- 第二种说法是 `2 X Y`，表示 X 吃 Y．

此人对 N 个动物，用上述两种说法，一句接一句地说出 K 句话，这 K 句话有的是真的，有的是假的．当一句话满足下列三条之一时，这句话就是假话，否则就是真话．

- 当前的话与前面的某些真的话冲突，就是假话；
- 当前的话中 X 或 Y 比 N 大，就是假话；
- 当前的话表示 X 吃 X，就是假话．

你的任务是根据给定的 N 和 K 句话，输出假话的总数．

解答一

考虑用带权并查集维护食物链信息．如果 x 和 y 是同类，那么 $x \equiv y \pmod 3$；如果 x 吃 y，那么 $x - y \equiv 1 \pmod 3$．这样就将本题转化为前文的模板题．

具体地，对于每一句话，除去那些 $x > n$ 或 $y > n$ 的显然的假话外，需要判断 x 和 y 是否已经连接：如果已经连接，计算两者的模意义下的距离，并与这句话声称的信息进行比较；否则，将两者按照这句话提供的信息连接．除了显然的情形外，一句话是假话，当且仅当提到的两个节点已经连接，且对应的距离与这句话声称的信息矛盾．

参考实现一

- C++

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

constexpr int M = 3;

struct DSU {
  std::vector<size_t> pa, size, dist;

  explicit DSU(size_t size_) : pa(size_), size(size_, 1), dist(size_) {
    std::iota(pa.begin(), pa.end(), 0);
  }

  size_t find(size_t x) {
    if (pa[x] == x) return x;
    size_t y = find(pa[x]);
    (dist[x] += dist[pa[x]]) %= M;
    return pa[x] = y;
  }

  bool unite(size_t x, size_t y, int d) {
    find(x), find(y);
    (d += M - dist[y]) %= M;
```

```
    (d += dist[x]) %= M;
    x = pa[x], y = pa[y];
    if (x == y) return d == 0;
    if (size[x] < size[y]) {
      std::swap(x, y);
      d = (M - d) % M;
    }
    pa[y] = x;
    size[x] += size[y];
    dist[y] = d;
    return true;
  }
};

int main() {
  int n, m;
  std::cin >> n >> m;
  DSU dsu(n + 1);
  int res = 0;
  for (; m; --m) {
    int op, x, y;
    std::cin >> op >> x >> y;
    if (x > n || y > n)
      ++res;
    else
      res += !dsu.unite(x, y, op == 1 ? 0 : 1);
  }
  std::cout << res << std::endl;
  return 0;
}

```

- Python

```
M = 3


class DSU:
    def __init__(self, size: int):
        self.pa = list(range(size))
        self.size = [1] * size
        self.dist = [0] * size

    def find(self, x: int) -> int:
        if self.pa[x] == x:
            return x
        y = self.find(self.pa[x])
        self.dist[x] = (self.dist[x] + self.dist[self.pa[x]]) % M
        self.pa[x] = y
        return y

    def unite(self, x: int, y: int, d: int) -> bool:
        self.find(x)
```

```
        self.find(y)
        d = (d + M - self.dist[y]) % M
        d = (d + self.dist[x]) % M
        x, y = self.pa[x], self.pa[y]
        if x == y:
            return d == 0
        if self.size[x] < self.size[y]:
            x, y = y, x
            d = (M - d) % M
        self.pa[y] = x
        self.size[x] += self.size[y]
        self.dist[y] = d
        return True


if __name__ == "__main__":
    n, m = map(int, input().split())
    dsu = DSU(n + 1)
    res = 0
    for _ in range(m):
        op, x, y = map(int, input().split())
        if x > n or y > n:
            res += 1
        else:
            res += not dsu.unite(x, y, 0 if op == 1 else 1)
    print(res)

```

解答二

将一种生物 x 拆分为三种状态. 在具体实现中, 我们可以直接将不同的状态当作不同的元素:

- 与 x 处于同一集合的状态与 x 属于同一物种;
- 与 $x+n$ 处于同一集合的状态能被 x 吃;
- 与 $x+2n$ 处于同一集合的能吃 x.

于是, 对于一句话:

- `1 x y` 为假话当且仅当:
    1. $x>N$ 或 $y>N$;
    2. y 与 $x+n$ 或 $x+2n$ 中的一个处于同一集合内.
- `2 x y` 为假话当且仅当:
    1. $x>N$ 或 $y>N$;
    2. y 与 x 或 $x+2n$ 中的一个处于同一集合内.
- 若为真话, 合并对应状态.

参考实现二

- C++

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

```

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

struct DSU {
  std::vector<size_t> pa, size;

  explicit DSU(size_t size_) : pa(size_), size(size_, 1) {
    std::iota(pa.begin(), pa.end(), 0);
  }

  size_t find(size_t x) { return pa[x] == x ? x : pa[x] = find(pa[x]); }

  void unite(size_t x, size_t y) {
    x = find(x), y = find(y);
    if (x == y) return;
    if (size[x] < size[y]) std::swap(x, y);
    pa[y] = x;
    size[x] += size[y];
  }
};

int main() {
  int n, m;
  std::cin >> n >> m;
  DSU dsu(n * 3 + 1);
  int res = 0;
  for (; m; --m) {
    int op, x, y;
    std::cin >> op >> x >> y;
    if (x > n || y > n)
      ++res;
    else if (op == 1) {
      if (dsu.find(x) == dsu.find(y + n) ||
          dsu.find(x) == dsu.find(y + (n << 1))) {
        ++res;
      } else {
        dsu.unite(x, y);
        dsu.unite(x + n, y + n);
        dsu.unite(x + n * 2, y + n * 2);
      }
    } else {
      if (dsu.find(x) == dsu.find(y) || dsu.find(x) == dsu.find(y + n)) {
        ++res;
      } else {
        dsu.unite(x, y + n * 2);
        dsu.unite(x + n, y);
        dsu.unite(x + n * 2, y + n);
      }
    }
  }
```

```
  std::cout << res << std::endl;
  return 0;
}

```

- Python

```python
class Dsu:
    def __init__(self, size):
        self.pa = list(range(size))
        self.size = [1] * size

    def find(self, x):
        if self.pa[x] != x:
            self.pa[x] = self.find(self.pa[x])
        return self.pa[x]

    def unite(self, x, y):
        x, y = self.find(x), self.find(y)
        if x == y:
            return
        if self.size[x] < self.size[y]:
            x, y = y, x
        self.pa[y] = x
        self.size[x] += self.size[y]


if __name__ == "__main__":
    n, m = map(int, input().split())
    dsu = Dsu(n * 3 + 1)
    res = 0
    for _ in range(m):
        op, x, y = map(int, input().split())
        if x > n or y > n:
            res += 1
        elif op == 1:
            if dsu.find(x) == dsu.find(y + n) or dsu.find(x) == dsu.find(y + (n << 1)):
                res += 1
            else:
                dsu.unite(x, y)
                dsu.unite(x + n, y + n)
                dsu.unite(x + n * 2, y + n * 2)
        else:
            if dsu.find(x) == dsu.find(y) or dsu.find(x) == dsu.find(y + n):
                res += 1
            else:
                dsu.unite(x, y + n * 2)
                dsu.unite(x + n, y)
                dsu.unite(x + n * 2, y + n)
    print(res)

```

ABC396E Min of Restricted Sum[9]

给定整数 N, M 和长度为 M 的整数序列 $X = (X_1, X_2, ..., X_M)$、$Y = (Y_1, Y_2, ..., Y_M)$、$Z = (Z_1, Z_2, ..., Z_M)$. 其中，保证 X 和 Y 的所有元素均在 1 至 N 的范围内.

定义长度为 N 的非负整数序列 $A = (A_1, A_2, ..., A_N)$ 为 **好的整数序列**，当且仅当满足以下条件：

- 对于所有满足 $1 \leq i \leq M$ 的整数 i，有 $A_{X_i} \oplus A_{Y_i} = Z_i$，其中 $\oplus$ 表示异或运算.

请判断是否存在这样的好的整数序列. 若存在，请找出使得元素总和 $\sum_{i=1}^{N} A_i$ 最小的好的整数序列，并输出该序列.

解答

异或就是单个二进制位上的「相同」或「不同」关系. 那么，将 A_i 的所有二进制位拆开，异或关系就能用带权并查集（或种类并查集）维护了. 同一个连通块内的元素一定对应着 A 中不同数字的同一个数位. 统计答案时，同一连通块的元素通常分为两组，两组之间取值应当不同，只需要取其中较大的一组赋值为 0，另一组赋值为 1 即可保证总权值最小.

参考实现

- C++

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

constexpr int M = 2;

struct DSU {
  std::vector<size_t> pa, size, dist;

  explicit DSU(size_t size_) : pa(size_), size(size_, 1), dist(size_) {
    std::iota(pa.begin(), pa.end(), 0);
  }

  size_t find(size_t x) {
    if (pa[x] == x) return x;
    size_t y = find(pa[x]);
    (dist[x] += dist[pa[x]]) %= M;
    return pa[x] = y;
  }

  bool unite(size_t x, size_t y, int d) {
    find(x), find(y);
    (d += M - dist[y]) %= M;
    (d += dist[x]) %= M;
    x = pa[x], y = pa[y];
    if (x == y) return d == 0;
    if (size[x] < size[y]) {
      std::swap(x, y);
      d = (M - d) % M;
```

```
    }
    pa[y] = x;
    size[x] += size[y];
    dist[y] = d;
    return true;
  }
};

int main() {
  int n, m;
  std::cin >> n >> m;
  DSU dsu((n + 1) << 5);
  for (; m; --m) {
    int x, y, z;
    std::cin >> x >> y >> z;
    for (int i = 0; i < 31; ++i) {
      if (!dsu.unite((x << 5) | i, (y << 5) | i, (z >> i) & 1)) {
        std::cout << -1 << std::endl;
        return 0;
      }
    }
  }
  std::vector<int> a(n + 1), cnt((n + 1) << 5);
  for (int i = 1; i < ((n + 1) << 5); ++i) {
    dsu.find(i);
    if (dsu.dist[i]) ++cnt[dsu.pa[i]];
  }
  for (int i = 1; i <= n; ++i) {
    for (int j = 0; j < 31; ++j) {
      int x = (i << 5) | j, y = dsu.pa[x];
      if ((cnt[y] > dsu.size[y] / 2) ^ dsu.dist[x]) {
        a[i] |= 1 << j;
      }
    }
  }
  for (int i = 1; i <= n; ++i) std::cout << a[i] << ' ';
  std::cout << std::endl;
  return 0;
}

```

- Python

```
M = 2


class DSU:
    def __init__(self, size: int):
        self.pa = list(range(size))
        self.size = [1] * size
        self.dist = [0] * size

    def find(self, x: int) -> int:
```

```
        if self.pa[x] == x:
            return x
        y = self.find(self.pa[x])
        self.dist[x] = (self.dist[x] + self.dist[self.pa[x]]) % M
        self.pa[x] = y
        return y

    def unite(self, x: int, y: int, d: int) -> bool:
        self.find(x)
        self.find(y)
        d = (d + M - self.dist[y]) % M
        d = (d + self.dist[x]) % M
        x, y = self.pa[x], self.pa[y]
        if x == y:
            return d == 0
        if self.size[x] < self.size[y]:
            x, y = y, x
            d = (M - d) % M
        self.pa[y] = x
        self.size[x] += self.size[y]
        self.dist[y] = d
        return True


if __name__ == "__main__":
    n, m = map(int, input().split())
    dsu = DSU((n + 1) << 5)
    for _ in range(m):
        x, y, z = map(int, input().split())
        for i in range(31):
            if not dsu.unite((x << 5) | i, (y << 5) | i, (z >> i) & 1):
                print(-1)
                exit()

    a = [0] * (n + 1)
    cnt = [0] * ((n + 1) << 5)

    for i in range(1, (n + 1) << 5):
        dsu.find(i)
        if dsu.dist[i]:
            cnt[dsu.pa[i]] += 1

    for i in range(1, n + 1):
        for j in range(31):
            x = (i << 5) | j
            y = dsu.pa[x]
            if (cnt[y] > dsu.size[y] // 2) ^ dsu.dist[x]:
                a[i] |= 1 << j

    print(" ".join(map(str, a[1:])))

```

习题

- 「NOI2015」程序自动分析[10]
- 「JSOI2008」星球大战[11]
- 「NOIP2023」三值逻辑[12]
- 「NOI2002」银河英雄传说[13]

其他应用

最小生成树算法→11.10.1@p2764 中的 Kruskal 和 最近公共祖先→11.6.5@p2646 中的 Tarjan 算法是基于并查集的算法.

相关专题见 并查集应用→14.2@p3346.

参考资料与拓展阅读

1. 知乎回答：是否在并查集中真的有二分路径压缩优化？ [14]
2. Gabow, H. N., & Tarjan, R. E. (1985). A Linear-Time Algorithm for a Special Case of Disjoint Set Union. JOURNAL OF COMPUTER AND SYSTEM SCIENCES, 30, 209-221.PDF[15]
3. CSDN：扩展域并查集 & 带权并查集[16]

[1] ResearchGate PDF

[2] SIAM Journal on Computing, 14(1), 129-133.

[3] Luogu P3367【模板】并查集

[4] Ackermann 函数

[5] SPOJ JMFILTER - Junk-Mail Filter

[6] Library Checker - Unionfind with Potential

[7] UVa11987 Almost Union-Find

[8] Luogu P2024「NOI2011」食物链

[9] ABC396E Min of Restricted Sum

[10]「NOI2015」程序自动分析

[11] 「JSOI2008」星球大战

[12] 「NOIP2023」三值逻辑

[13]「NOI2002」银河英雄传说

[14] 知乎回答：是否在并查集中真的有二分路径压缩优化?

[15] PDF

[16] CSDN：扩展域并查集 & 带权并查集

10.6.2 并查集复杂度

Authors: orzAtalod

本部分内容转载并修改自 时间复杂度 - 势能分析浅谈[1]，已取得原作者授权同意.

定义

阿克曼函数

这里，先给出 $\alpha(n)$ 的定义．为了给出这个定义，先给出 $A_k(j)$ 的定义.

定义 $A_k(j)$ 为:

$$A_k(j)=\begin{cases}j+1 & k=0\\ A_{k-1}^{(j+1)}(j) & k\geq 1\end{cases}$$

即阿克曼函数.

这里，$f^i(x)$ 表示将 f 连续应用在 x 上 i 次，即 $f^0(x)=x$，$f^i(x)=f(f^{i-1}(x))$.

再定义 $\alpha(n)$ 为使得 $A_{\alpha(n)}(1)\geq n$ 的最小整数值．注意，我们之前将它描述为 $A_{\alpha(n)}(\alpha(n))\geq n$，反正他们的增长速度都很慢，值都不超过 4.

基础定义

每个节点都有一个 rank．这里的 rank 不是节点个数，而是深度．节点的初始 rank 为 0，在合并的时候，如果两个节点的 rank 不同，则将 rank 小的节点合并到 rank 大的节点上，并且不更新大节点的 rank 值．否则，随机将某个节点合并到另外一个节点上，将根节点的 rank 值 +1．这里根节点的 rank 给出了该树的高度．记 x 的 rank 为 $rnk(x)$，类似的，记 x 的父节点为 $fa(x)$．我们总有 $rnk(x)+1 \le rnk(fa(x))$．

为了定义势函数，需要预先定义一个辅助函数 $level(x)$．其中，$level(x) = \max(k : rnk(fa(x)) \ge A_k(rnk(x)))$．当 $rnk(x) \ge 1$ 的时候，再定义一个辅助函数 $iter(x) = \max(i : rnk(fa(x)) \ge A^i_{level(x)}(rnk(x))$．这些函数定义的 x 都满足 $rnk(x) > 0$ 且 x 不是某个树的根．

上面那些定义可能让你有点头晕．再理一下，对于一个 x 和 $fa(x)$，如果 $rnk(x) > 0$，总是可以找到一对 i, k 令 $rnk(fa(x)) \ge A^i_k(rnk(x))$，而 $level(x) = \max(k)$，在这个前提下，$iter(x) = \max(i)$．$level$ 描述了 A 的最大迭代级数，而 $iter$ 描述了在最大迭代级数时的最大迭代次数．

对于这两个函数，$level(x)$ 总是随着操作的进行而增加或不变，如果 $level(x)$ 不增加，$iter(x)$ 也只会增加或不变．并且，它们总是满足以下两个不等式：

$$0 \le level(x) < \alpha(n)$$

$$1 \le iter(x) \le rnk(x)$$

考虑 $level(x)$、$iter(x)$ 和 A^j_k 的定义，这些很容易被证明出来，就留给读者用于熟悉定义了．

定义势能函数 $\Phi(S) = \sum_{x \in S} \Phi(x)$，其中 S 表示一整个并查集，而 x 为并查集中的一个节点．定义 $\Phi(x)$ 为：

$$\Phi(x) = \begin{cases} \alpha(n) \times rnk(x) & rnk(x) = 0 \text{ 或 } x \text{ 为某棵树的根节点} \\ (\alpha(n) - level(x)) \times rnk(x) - iter(x) & \text{otherwise} \end{cases}$$

然后就是通过操作引起的势能变化来证明摊还时间复杂度为 $\Theta(\alpha(n))$ 啦．注意，这里我们讨论的 $union(x, y)$ 操作保证了 x 和 y 都是某个树的根，因此不需要额外执行 $find(x)$ 和 $find(y)$．

可以发现，势能总是个非负数．另，在开始的时候，并查集的势能为 0．

证明

union(x,y) 操作

其花费的时间为 $\Theta(1)$，因此我们考虑其引起的势能的变化．

这里，我们假设 $rnk(x) \le rnk(y)$，即 x 被接到 y 上．这样，势能增加的节点仅有 x（从树根变成非树根），y（秩可能增加）和操作前 y 的子节点（父节点的秩可能增加）．我们先证明操作前 y 的子节点 c 的势能不可能增加，并且如果减少了，至少减少 1．

设操作前 c 的势能为 $\Phi(c)$，操作后为 $\Phi(c')$，这里 c 可以是任意一个 $rnk(c) > 0$ 的非根节点，操作可以是任意操作，包括下面的 find 操作．我们分三种情况讨论．

1. $iter(c)$ 和 $level(c)$ 并未增加．显然有 $\Phi(c) = \Phi(c')$．
2. $iter(c)$ 增加了，$level(c)$ 并未增加．这里 $iter(c)$ 至少增加一，即 $\Phi(c') \le \Phi(c) - 1$，势能函数减少了，并且至少减少 1．
3. $level(c)$ 增加了，$iter(c)$ 可能减少．但是由于 $0 < iter(c) \le rnk(c)$，$iter(c)$ 最多减少 $rnk(c) - 1$，而 $level(c)$ 至少增加 1．由定义 $\Phi(c) = (\alpha(n) - level(c)) \times rnk(c) - iter(c)$，可得 $\Phi(c') \le \Phi(c) - 1$．
4. 其他情况．由于 $rnk(c)$ 不变，$rnk(fa(c))$ 不减，所以不存在．

所以，势能增加的节点仅可能是 x 或 y．而 x 从树根变成了非树根，如果 $rnk(x) = 0$，则一直有 $\Phi(x) = \Phi(x') = 0$．否则，一定有 $\alpha(x) \times rnk(x) \ge (\alpha(n) - level(x)) \times rnk(x) - iter(x)$．即，$\Phi(x') \le \Phi(x)$．

因此，唯一势能可能增加的点就是 y．而 y 的势能最多增加 $\alpha(n)$．因此，可得 $union$ 操作均摊后的时间复杂度为 $\Theta(\alpha(n))$．

find(a) 操作

如果查找路径包含 $\Theta(s)$ 个节点, 显然其查找的时间复杂度是 $\Theta(s)$. 如果由于查找操作, 没有节点的势能增加, 且至少有 $s-\alpha(n)$ 个节点的势能至少减少 1, 就可以证明 $find(a)$ 操作的时间复杂度为 $\Theta(\alpha(n))$. 为了避免混淆, 这里用 a 作为参数, 而出现的 x 都是泛指某一个并查集内的结点.

首先证明没有节点的势能增加. 很显然, 我们在上面证明过所有非根节点的势能不增, 而根节点的 rnk 没有改变, 所以没有节点的势能增加.

接下来证明至少有 $s-\alpha(n)$ 个节点的势能至少减少 1. 我们上面证明过了, 如果 $level(x)$ 或者 $iter(x)$ 有改变的话, 它们的势能至少减少 1. 所以, 只需要证明至少有 $s-\alpha(n)$ 个节点的 $level(x)$ 或者 $iter(x)$ 有改变即可.

回忆一下非根节点势能的定义, $\Phi(x)=(\alpha(n)-level(x))\times rnk(x)-iter(x)$, 而 $level(x)$ 和 $iter(x)$ 是使 $rnk(fa(x))\geq A^{iter(x)}_{level(x)}(rnk(x))$ 的最大数.

所以, 如果 $root_x$ 代表 x 所处的树的根节点, 只需要证明 $rnk(root_x)\geq A^{iter(x)+1}_{level(x)}(rnk(x))$ 就好了. 根据 A^i_k 的定义, $A^{iter(x)+1}_{level(x)}(rnk(x))=A_{level(x)}(A^{iter(x)}_{level(x)}(rnk(x)))$.

注意, 我们可能会用 $k(x)$ 代表 $level(x)$, $i(x)$ 代表 $iter(x)$ 以避免式子过于冗长. 这里, 就是 $rnk(root_x)\geq A_{k(x)}(A^{i(x)}_{k(x)}(x))$.

当你看到这的时候, 可能会有一种「这啥玩意」的感觉. 这意味着你可能需要多看几遍, 或者跳过一些内容以后再看.

这里, 我们需要一个外接的 $A_{k(x)}$, 意味着我们可能需要再找一个点 y. 令 y 是搜索路径上在 x 之后的满足 $k(y)=k(x)$ 的点, 这里「搜索路径之后」相当于「是 x 的祖先」. 显然, 不是每一个 x 都有这样一个 y. 很容易证明, 没有这样的 y 的 x 不超过 $\alpha(n)+2$ 个. 因为只有每个 k 的最后一个 x 和 a 以及 $root_a$ 没有这样的 y.

我们再强调一遍 $fa(x)$ 指的是路径压缩 **之前** x 的父节点, 路径压缩 **之后** x 的父节点一律用 $root_x$ 表示. 对于每个存在 y 的 x, 总是有 $rnk(y)\geq rnk(fa(x))$. 同时, 我们有 $rnk(fa(x))\geq A^{i(x)}_{k(x)}(rnk(x))$. 由于 $k(x)=k(y)$, 我们用 k 来统称, 即, $rnk(fa(x))\geq A^{i(x)}_k(rnk(x))$. 我们需要造一个 A_k 出来, 所以我们可以不关注 $iter(y)$ 的值, 直接使用弱化版的 $rnk(fa(y))\geq A_k(rnk(y))$.

如果我们将不等式组合起来, 神奇的事情就发生了. 我们发现, $rnk(fa(y))\geq A^{i(x)+1}_k(rnk(x))$. 也就是说, 为了从 $rnk(x)$ 迭代到 $rnk(fa(y))$, 至少可以迭代 A_k 不少于 $i(x)+1$ 次而不超过 $rnk(fa(y))$.

显然, 有 $rnk(root_y)\geq rnk(fa(y))$, 且 $rnk(x)$ 在路径压缩时不变. 因此, 我们可以得到 $rnk(root_x)\geq A^{i(x)+1}_k(rnk(x))$, 也就是说 $iter(x)$ 的值至少增加 1, 如果 $rnk(x)$ 没有增加, 一定是 $level(x)$ 增加了.

所以, $\Phi(x)$ 至少减少了 1. 由于这样的 x 节点至少有 $s-\alpha(n)-2$ 个, 所以最后 $\Phi(S)$ 至少减少了 $s-\alpha(n)-2$, 均摊后的时间复杂度即为 $\Theta(\alpha(n)+2)=\Theta(\alpha(n))$.

为何并查集会被卡

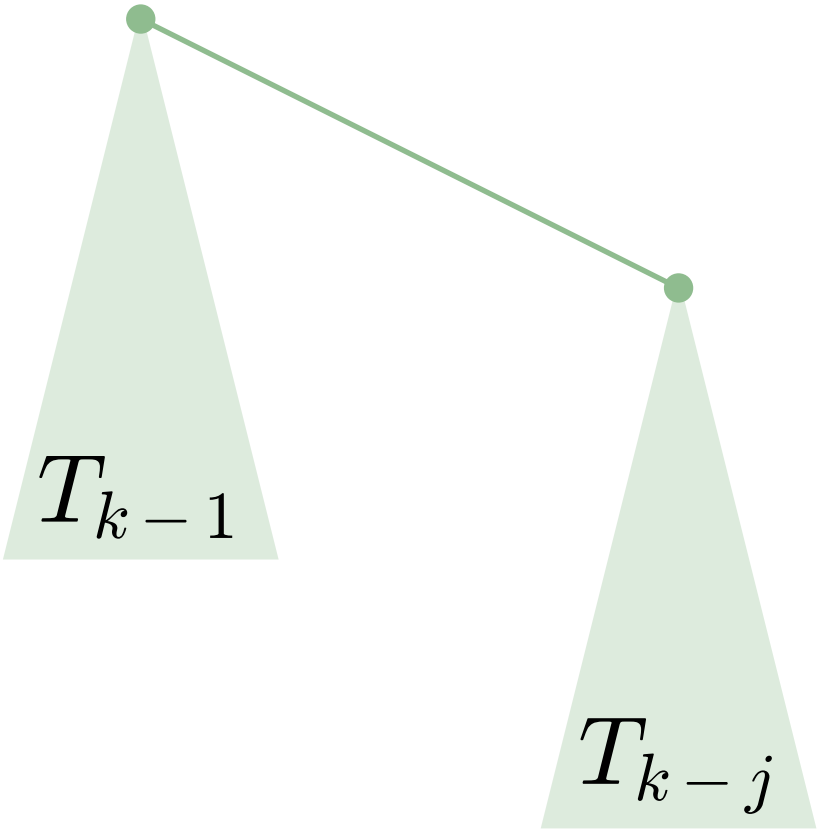
这个问题也就是问, 如果我们不按秩合并, 会有哪些性质被破坏, 导致并查集的时间复杂度不能保证为 $\Theta(m\alpha(n))$.

如果我们在合并的时候, rnk 较大的合并到了 rnk 较小的节点上面, 我们就将那个 rnk 较小的节点的 rnk 值设为另一个节点的 rnk 值加一. 这样, 我们就能保证 $rnk(fa(x))\geq rnk(x)+1$, 从而不会出现类似于满地 compile error 一样的性质不符合.

显然, 如果这样子的话, 我们破坏的就是 $union(x,y)$ 函数「y 的势能最多增加 $\alpha(n)$」这一句.

存在一个能使路径压缩并查集时间复杂度降至 $\Omega(m\log_{1+\frac{m}{n}}n)$ 的结构, 定义如下:

二项树（实际上和一般的二项树不太一样), 其中 j 是常数, T_k 为一个 T_{k-1} 加上一个 T_{k-j} 作为根节点的儿子.



边界条件，T_1 到 T_j 都是一个单独的点.

令 $rnk(T_k)=r_k$，这里我们有 $r_k=(k-1)/j$（证明略）. 每轮操作，我们将它接到一个单节点上，然后查询底部的 j 个节点. 也就是说，我们接到单节点上的时候，单节点的势能提高了 $(k-1)/j+1$. 在 $j=\lfloor\frac{m}{n}\rfloor$，$i=\lfloor\log_{j+1}\frac{n}{2}\rfloor$，$k=ij$ 的时候，势能增加量为：

$$\alpha(n)\times((ij-1)/j+1)=\alpha(n)\times((\left\lfloor\log_{\lfloor\frac{m}{n}\rfloor+1}\frac{n}{2}\right\rfloor\times\left\lfloor\frac{m}{n}\right\rfloor-1)/\left\lfloor\frac{m}{n}\right\rfloor+1)$$

变换一下，去掉所有的取整符号，就可以得出，势能增加量 $\geq\alpha(n)\times(\log_{1+\frac{m}{n}}n-\frac{n}{m})$，m 次操作就是 $\Omega(m\log_{1+\frac{m}{n}}n-n)=\Omega(m\log_{1+\frac{m}{n}}n)$.

关于启发式合并

由于按秩合并比启发式合并难写，所以很多 dalao 会选择使用启发式合并来写并查集. 具体来说，则是对每个根都维护一个 $size(x)$，每次将 $size$ 小的合并到大的上面.

所以，启发式合并会不会被卡?

首先，可以从秩参与证明的性质来说明. 如果 $size$ 可以代替 rnk 的地位，则可以使用启发式合并. 快速总结一下，秩参与证明的性质有以下三条：

1. 每次合并，最多有一个节点的秩上升，而且最多上升 1.

2. 总有 $rnk(fa(x)) \geq rnk(x)+1$.
3. 节点的秩不减.

关于第二条和第三条，siz 显然满足，然而第一条不满足，如果将 x 合并到 y 上面，则 $siz(y)$ 会增大 $siz(x)$ 那么多.

所以，可以考虑使用 $\log_2 siz(x)$ 代替 $rnk(x)$.

关于第一条性质，由于节点的 siz 最多翻倍，所以 $\log_2 siz(x)$ 最多上升 1. 关于第二三条性质，结论较为显然，这里略去证明.

所以说，如果不想写按秩合并，就写启发式合并好了，时间复杂度仍旧是 $\Theta(m\alpha(n))$.

外部参考资料

[1] 时间复杂度 - 势能分析浅谈 

10.7 堆

10.7.1 堆简介

Authors: ouuan, HeRaNO

堆是一棵树，其每个节点都有一个键值，且每个节点的键值都大于等于/小于等于其父亲的键值.

每个节点的键值都大于等于其父亲键值的堆叫做小根堆，否则叫做大根堆. STL 中的 `priority_queue`→4.3.2.6@p384 其实就是一个大根堆.

（小根）堆主要支持的操作有：插入一个数、查询最小值、删除最小值、合并两个堆、减小一个元素的值.

一些功能强大的堆（可并堆）还能（高效地）支持 merge 等操作.

一些功能更强大的堆还支持可持久化，也就是对任意历史版本进行查询或者操作，产生新的版本.

堆的分类

操作 \ 数据结构[1]	配对堆	二叉堆	左偏树	二项堆	斐波那契堆
插入（insert）	$O(1)$	$O(\log n)$	$O(\log n)$	$O(\log n)^{2}$	$O(1)$
查询最小值（find-min）	$O(1)$	$O(1)$	$O(1)$	$O(1)^{3,4}$	$O(1)$
删除最小值（delete-min）	$O(\log n)^{5}$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)^{6}$
合并 (merge)	$O(1)$	$O(n)$	$O(\log n)$	$O(\log n)$	$O(1)$
减小一个元素的值 (decrease-key)	$o(\log n)$（下界 $\Omega(\log\log n)$，上界 $O(2^{2\sqrt{\log\log n}})$）[7]	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(1)^{8}$
是否支持可持久化	×	✓	✓	✓	×

习惯上，不加限定提到「堆」时往往都指二叉堆.

[1] 表格来自于 Wikipedia[1]
[2] 单次插入的复杂度为 $O(\log n)$，但有 k 次连续插入时，可创建一个只包含要插入元素的二项堆，再将此堆与原先的二项堆进行合并，均摊复杂度为 $O(1)$
[3] 可以保存一个指向最小元素的指针，在执行其他操作时修改该指针，即可在 $O(1)$ 的复杂度下进行查询了
[4] 复杂度为均摊复杂度
[5] 复杂度为均摊复杂度
[6] 复杂度为均摊复杂度
[7] 复杂度为均摊复杂度
[8] 复杂度为均摊复杂度

[1] Wikipedia

10.7.2 二叉堆

Authors: HeRaNO, Xeonacid, AzurIce

结构

从二叉堆的结构说起，它是一棵二叉树，并且是完全二叉树，每个结点中存有一个元素（或者说，有个权值）.

堆性质: 父亲的权值不小于儿子的权值（大根堆）. 同样的, 我们可以定义小根堆. 本文以大根堆为例.

由堆性质，树根存的是最大值（getmax 操作就解决了）.

过程

插入操作

插入操作是指向二叉堆中插入一个元素，要保证插入后也是一棵完全二叉树.

最简单的方法就是，最下一层最右边的叶子之后插入.

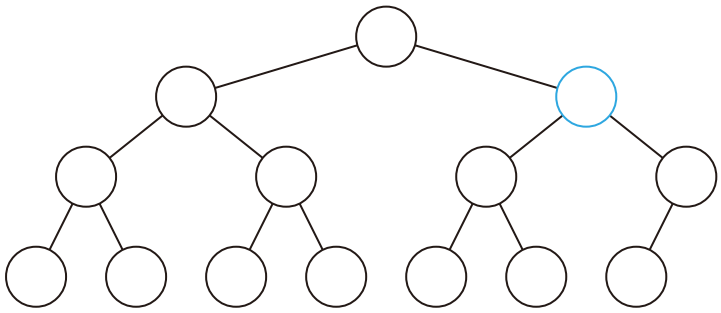
如果最下一层已满，就新增一层.

插入之后可能会不满足堆性质?

向上调整：如果这个结点的权值大于它父亲的权值，就交换，重复此过程直到不满足或者到根.

可以证明，插入之后向上调整后，没有其他结点会不满足堆性质.

向上调整的时间复杂度是 $O(\log n)$ 的.



删除操作

删除操作指删除堆中最大的元素，即删除根结点.

但是如果直接删除，则变成了两个堆，难以处理.

所以不妨考虑插入操作的逆过程，设法将根结点移到最后一个结点，然后直接删掉.

然而实际上不好做，我们通常采用的方法是，把根结点和最后一个结点直接交换.

于是直接删掉（在最后一个结点处的）根结点，但是新的根结点可能不满足堆性质……

向下调整：在该结点的儿子中，找一个最大的，与该结点交换，重复此过程直到底层.

可以证明，删除并向下调整后，没有其他结点不满足堆性质.

时间复杂度 $O(\log n)$.

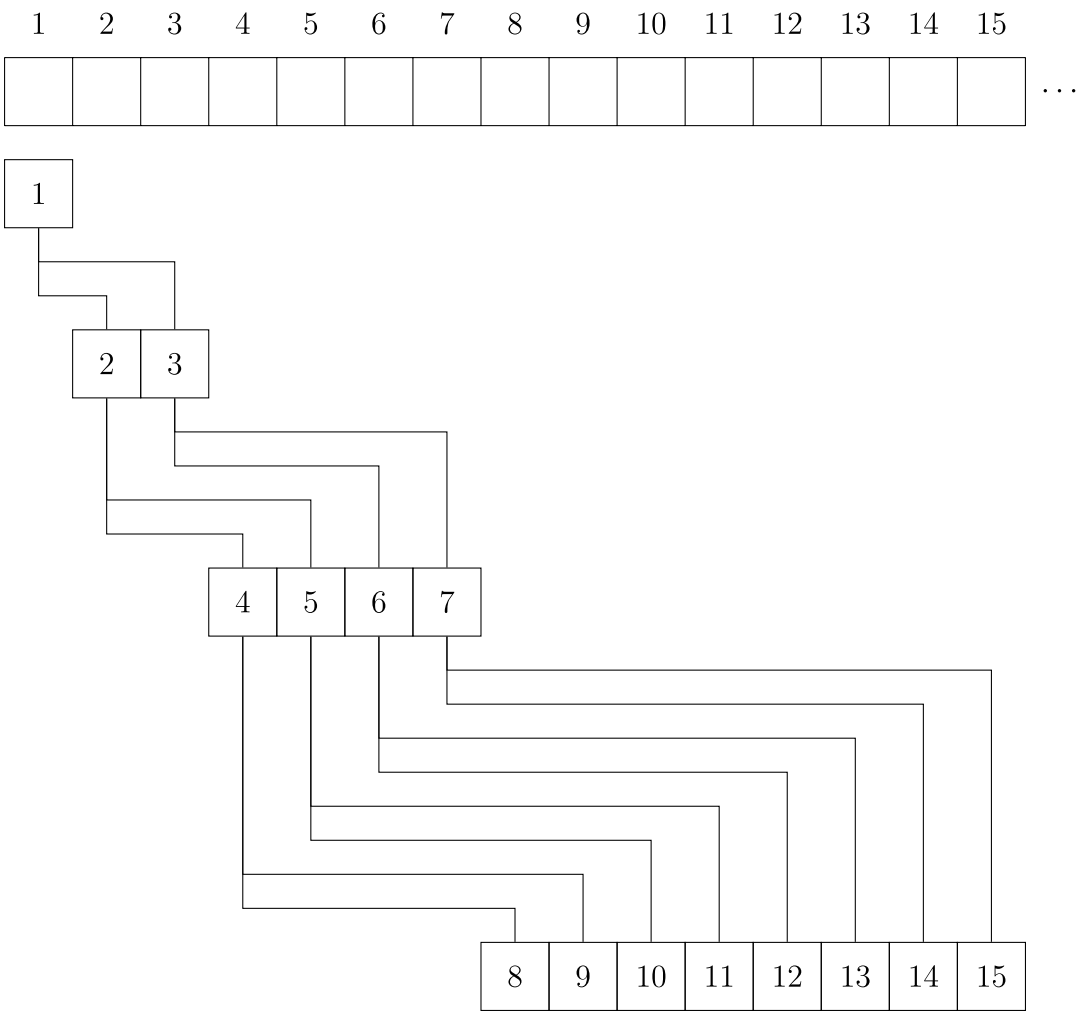
增加某个点的权值

很显然，直接修改后，向上调整一次即可，时间复杂度为 $O(\log n)$.

实现

我们发现，上面介绍的几种操作主要依赖于两个核心：向上调整和向下调整.

考虑使用一个序列 h 来表示堆. h_i 的两个儿子分别是 h_{2i} 和 h_{2i+1}，1 是根结点：



参考代码：

```
void up(int x) {
  while (x > 1 && h[x] > h[x / 2]) {
    std::swap(h[x], h[x / 2]);
    x /= 2;
  }
}

void down(int x) {
  while (x * 2 <= n) {
    t = x * 2;
    if (t + 1 <= n && h[t + 1] > h[t]) t++;
    if (h[t] <= h[x]) break;
    std::swap(h[x], h[t]);
    x = t;
```

```
15   }
16 }
```

建堆

考虑这么一个问题，从一个空的堆开始，插入 n 个元素，不在乎顺序.

直接一个一个插入需要 $O(n\log n)$ 的时间，有没有更好的方法?

方法一：向上调整

从根开始，按 BFS 序进行.

```
1 void build_heap_1() {
2   for (i = 1; i <= n; i++) up(i);
3 }
```

这个做法仍然相当于一个一个插入, 只是把元素提前放在了数组里, 可以改善常数. 故而最坏情况下, 算法时间复杂度满足的递推式为 $T(n)=T(n-1)+\Theta(\log n)$，累加得 $T(n)=\Theta(n\log n)$.

方法二：向下调整

这时换一种思路，从叶子开始，逐个向下调整.

```
1 void build_heap_2() {
2   for (i = n; i >= 1; i--) down(i);
3 }
```

换一种理解方法，每次「合并」两个已经调整好的堆，这说明了正确性.

注意到叶节点无需调整, 因此可从序列约 $n/2$ 的位置开始调整, 可以改善常数. 根据每次合并两个堆这一理解，可以写出算法时间复杂度的递推式 $T(n)=2T(\frac{n}{2})+O(\log n)$，由主定理可得 $T(n)=\Theta(n)$.

之所以能 $\Theta(n)$ 建堆，是因为堆性质很弱，二叉堆并不是唯一的.

要是像排序那样的强条件就难说了.

应用

对顶堆

> **SPOJ RMID2 - Running Median Again**[1]
>
> 维护一个序列，支持两种操作：
> 1. 向序列中插入一个元素
> 2. 输出并删除当前序列的中位数（若序列长度为偶数，则输出较小的中位数）

这个问题可以被进一步抽象成：动态维护一个序列上第 k 大的数，k 值可能会发生变化.

对于此类问题, 我们可以使用 **对顶堆** 这一技巧予以解决(可以避免写权值线段树或 BST 带来的繁琐).

对顶堆由一个大根堆与一个小根堆组成，小根堆维护大值即前 k 大的值（包含第 k 个），大根堆维护小值即比第 k 大数小的其他数.

这两个堆构成的数据结构支持以下操作：

- 维护：当小根堆的大小小于 k 时，不断将大根堆堆顶元素取出并插入小根堆，直到小根堆的大小等于 k；当小根堆的大小大于 k 时，不断将小根堆堆顶元素取出并插入大根堆，直到小根堆的大小等于 k；
- 插入元素：若插入的元素大于等于小根堆堆顶元素，则将其插入小根堆，否则将其插入大根堆，然后维护对顶堆；

- 查询第 k 大元素：小根堆堆顶元素即为所求；
- 删除第 k 大元素：删除小根堆堆顶元素，然后维护对顶堆；
- k 值 $+1/-1$：根据新的 k 值直接维护对顶堆.

显然，查询第 k 大元素的时间复杂度是 $O(1)$ 的. 由于插入、删除或调整 k 值后，小根堆的大小与期望的 k 值最多相差 1，故每次维护最多只需对大根堆与小根堆中的元素进行一次调整，因此，这些操作的时间复杂度都是 $O(\log n)$ 的.

参考代码

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

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  int t, x;
  cin >> t;
  while (t--) {
    // 大根堆，维护前一半元素（存小值）
    priority_queue<int, vector<int>, less<int>> a;
    // 小根堆，维护后一半元素（存大值）
    priority_queue<int, vector<int>, greater<int>> b;
    while (cin >> x, x) {
      // 若为查询并删除操作，输出并删除大根堆堆顶元素
      // 因为这题要求输出中位数中较小者（偶数个数字会存在两个中位数候选）
      // 这个和上面的第k大讲解有稍许出入，但如果理解了上面的，这个稍微变通下便可理清
      if (x == -1) {
        cout << a.top() << '\n';
        a.pop();
      }
      // 若为插入操作，根据大根堆堆顶的元素值，选择合适的堆进行插入
      else {
        if (a.empty() || x <= a.top())
          a.push(x);
        else
          b.push(x);
      }
      // 对对顶堆进行调整
      if (a.size() > (a.size() + b.size() + 1) / 2) {
        b.push(a.top());
        a.pop();
      } else if (a.size() < (a.size() + b.size() + 1) / 2) {
        a.push(b.top());
        b.pop();
      }
    }
  }
  return 0;
}

```

习题

- SPOJ RMID - Running Median[2]

- 洛谷 P1801 黑匣子[3]

外部参考资料

[1] SPOJ RMID2 - Running Median Again

[2] SPOJ RMID - Running Median

[3] 洛谷 P1801 黑匣子

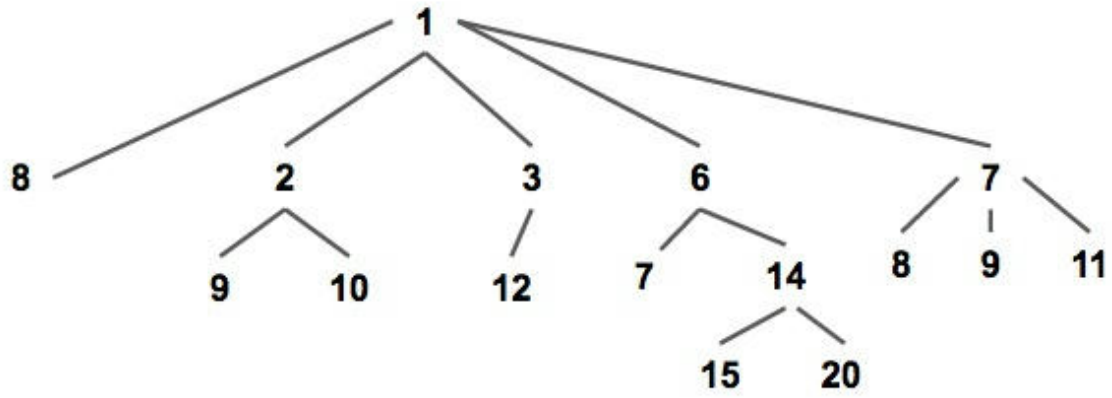
10.7.3 配对堆

引入

配对堆是一个支持插入，查询/删除最小值，合并，修改元素等操作的数据结构，是一种可并堆．有速度快和结构简单的优势，但由于其为基于势能分析的均摊复杂度，无法可持久化．

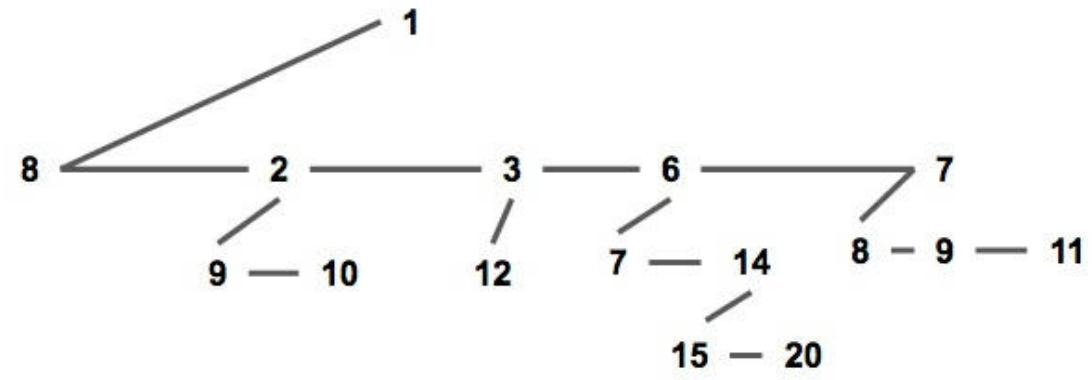
定义

配对堆是一棵满足堆性质的带权多叉树（如下图），即每个节点的权值都小于或等于他的所有儿子（以小根堆为例，下同）．



通常我们使用儿子 - 兄弟表示法储存一个配对堆（如下图），一个节点的所有儿子节点形成一个单向链表．每个节点储存第一个儿子的指针，即链表的头节点；和他的右兄弟的指针．

这种方式便于实现配对堆，也将方便复杂度分析．




```
1 struct Node {
2   T v;  // T为权值类型
```

```
  Node *child, *sibling;
  // child 指向该节点第一个儿子，sibling 指向该节点的下一个兄弟.
  // 若该节点没有儿子/下个兄弟则指针指向 nullptr.
};
```

从定义可以发现，和其他常见的堆结构相比，配对堆不维护任何额外的树大小，深度，排名等信息（二叉堆也不维护额外信息，但它是通过维持一个严格的完全二叉树结构来保证操作的复杂度），且任何一个满足堆性质的树都是一个合法的配对堆，这样简单又高度灵活的数据结构奠定了配对堆在实践中优秀效率的基础；作为对比，斐波那契堆糟糕的常数就是因为它需要维护很多额外的信息.

配对堆通过一套精心设计的操作顺序来保证它的总复杂度，原论文[9]将其称为「一种自调整的堆（Self Adjusting Heap）」. 在这方面和 Splay 树（在原论文中被称作「Self Adjusting Binary Tree」）颇有相似之处.

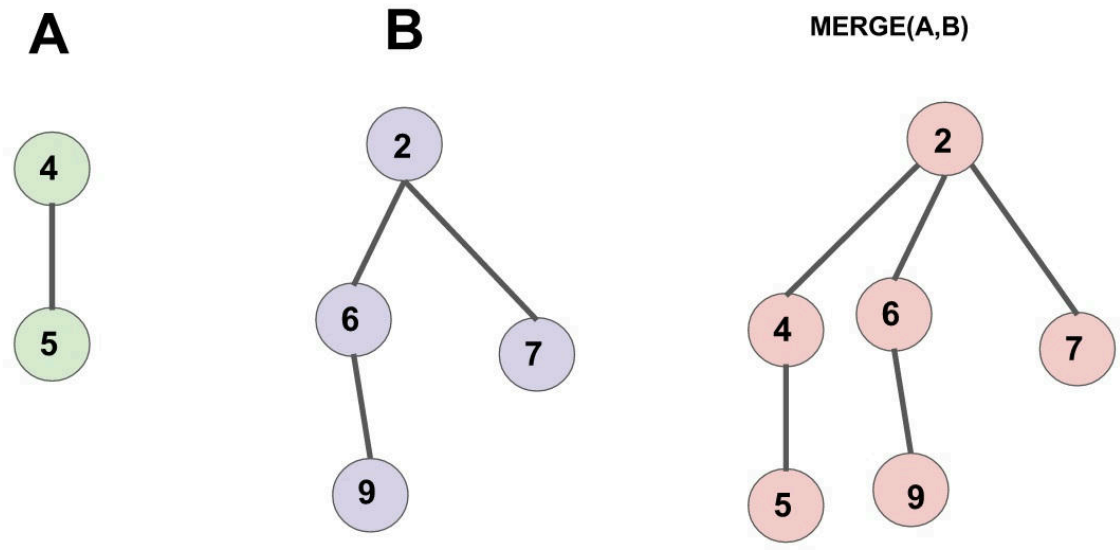
过程

查询最小值

从配对堆的定义可看出，配对堆的根节点的权值一定最小，直接返回根节点即可.

合并

合并两个配对堆的操作很简单，首先令两个根节点较小的一个为新的根节点，然后将较大的根节点作为它的儿子插入进去.（见下图）



需要注意的是，一个节点的儿子链表是按插入时间排序的，即最右边的节点最早成为父节点的儿子，最左边的节点最近成为父节点的儿子.

实现

```
Node* meld(Node* x, Node* y) {
  // 若有一个为空则直接返回另一个
  if (x == nullptr) return y;
  if (y == nullptr) return x;
  if (x->v > y->v) std::swap(x, y);  // swap后x为权值小的堆，y为权值大的堆
  // 将y设为x的儿子
```

[9] The pairing heap: a new form of self-adjusting heap[1]

```
  y->sibling = x->child;
  x->child = y;
  return x;  // 新的根节点为 x
}
```

插入

合并都有了，插入就直接把新元素视为一个新的配对堆和原堆合并就行了.

删除最小值

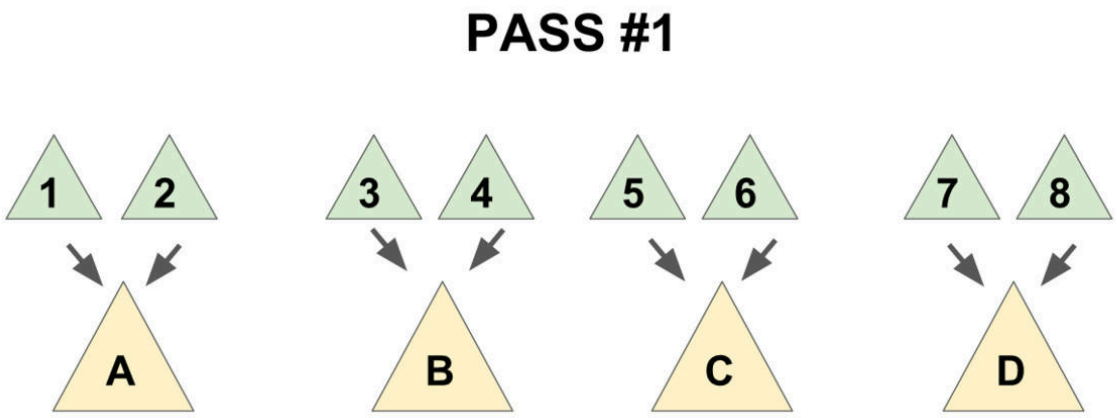
首先要提及的一点是，上文的几个操作都十分偷懒，完全没有对数据结构进行维护，所以我们需要小心设计删除最小值的操作，来保证总复杂度不出问题.

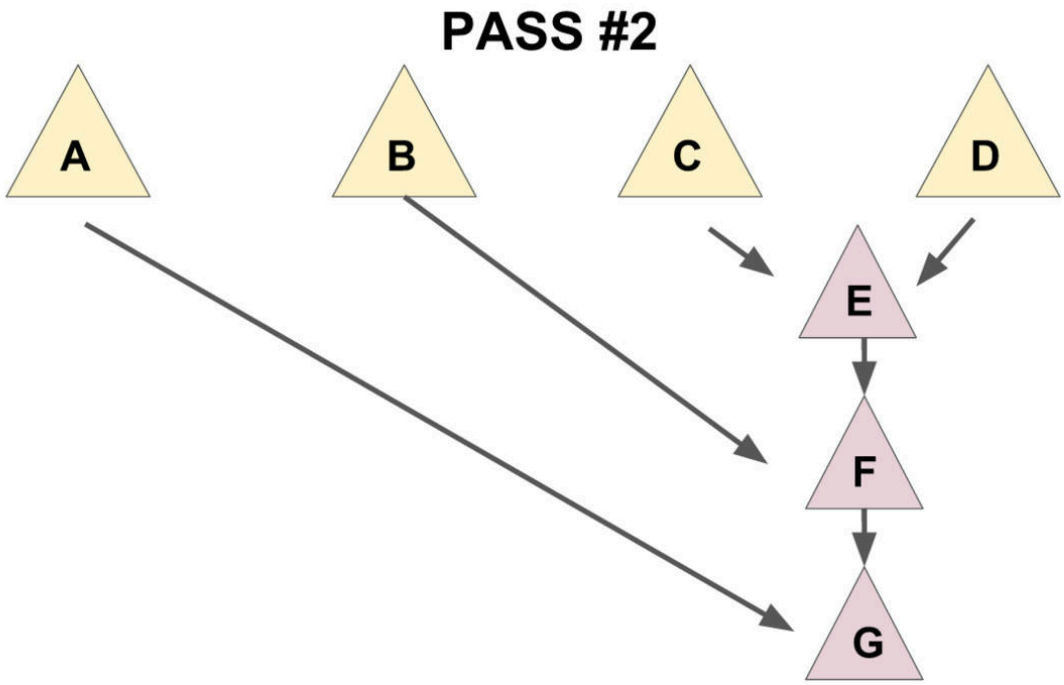
根节点即为最小值，所以要删除的是根节点. 考虑拿掉根节点之后会发生什么：根节点原来的所有儿子构成了一片森林；而配对堆应当是一棵树，所以我们需要通过某种顺序把这些儿子全部合并起来.

一个很自然的想法是使用 `meld` 函数把儿子们从左到右挨个并在一起，这样做的话正确性是显然的，但是会导致单次操作复杂度退化到 $O(n)$.

为了保证总的均摊复杂度，需要使用一个「两步走」的合并方法：

1. 把儿子们两两配成一对，用 `meld` 操作把被配成同一对的两个儿子合并到一起（见下图 1），
2. 将新产生的堆 **从右往左**（即老的儿子到新的儿子的方向）挨个合并在一起（见下图 2）.





先实现一个辅助函数 `merges`，作用是合并一个节点的所有兄弟.

实现

```
Node* merges(Node* x) {
  if (x == nullptr || x->sibling == nullptr)
    return x;  // 如果该树为空或他没有下一个兄弟，就不需要合并了，return.
  Node* y = x->sibling;                  // y 为 x 的下一个兄弟
  Node* c = y->sibling;                  // c 是再下一个兄弟
  x->sibling = y->sibling = nullptr;     // 拆散
  return meld(merges(c), meld(x, y));  // 核心部分
}
```

最后一句话是该函数的核心，这句话分三部分:

1. `meld(x,y)`「配对」了 x 和 y.
2. `merges(c)` 递归合并 c 和他的兄弟们.
3. 将上面 2 个操作产生的 2 个新树合并.

需要注意到的是, 上文提到了第二步时的合并方向是有要求的(从右往左合并), 该递归函数的实现已保证了这个顺序，如果读者需要自行实现迭代版本的话请务必注意保证该顺序，否则复杂度将失去保证.

有了 `merges` 函数，`delete-min` 操作就显然了.

实现

```
Node* delete_min(Node* x) {
  Node* t = merges(x->child);
  delete x;  // 如果需要内存回收
  return t;
}
```

减小一个元素的值

要实现这个操作，需要给节点添加一个「父」指针，当节点有左兄弟时，其指向左兄弟而非实际的父节点；否则，指向其父节点.

首先节点的定义修改为：

实现

```
struct Node {
  LL v;
  int id;
  Node *child, *sibling;
  Node *father;  // 新增：父指针，若该节点为根节点则指向空节点 nullptr
};
```

`meld` 操作修改为：

实现

```
Node* meld(Node* x, Node* y) {
  if (x == nullptr) return y;
  if (y == nullptr) return x;
  if (x->v > y->v) std::swap(x, y);
  if (x->child != nullptr) {  // 新增：维护父指针
    x->child->father = y;
  }
  y->sibling = x->child;
  y->father = x;  // 新增：维护父指针
  x->child = y;
  return x;
}
```

`merges` 操作修改为：

实现

```
Node *merges(Node *x) {
  if (x == nullptr) return nullptr;
  x->father = nullptr;  // 新增：维护父指针
  if (x->sibling == nullptr) return x;
  Node *y = x->sibling, *c = y->sibling;
  y->father = nullptr;  // 新增：维护父指针
  x->sibling = y->sibling = nullptr;
  return meld(merges(c), meld(x, y));
}
```

现在我们来考虑如何实现 `decrease-key` 操作.

首先我们发现，当我们减少节点 x 的权值之后，以 x 为根的子树仍然满足配对堆性质，但 x 的父亲和 x 之间可能不再满足堆性质.

因此我们把整棵以 x 为根的子树剖出来，现在两棵树都符合配对堆性质了，然后把他们合并起来，就完成了全部操作.

实现

```
// root为堆的根，x为要操作的节点，v为新的权值，调用时需保证 v <= x->v
// 返回值为新的根节点
Node *decrease_key(Node *root, Node *x, LL v) {
  x->v = v;                  // 更新权值
  if (x == root) return x;  // 如果 x 为根，则直接返回
  // 把x从fa的子节点中剖出去，这里要分x的位置讨论一下.
  if (x->father->child == x) {
    x->father->child = x->sibling;
  } else {
    x->father->sibling = x->sibling;
  }
  if (x->sibling != nullptr) {
    x->sibling->father = x->father;
  }
  x->sibling = nullptr;
  x->father = nullptr;
  return meld(root, x);  // 重新合并 x 和根节点
}
```

复杂度分析

配对堆结构与实现简单，但时间复杂度分析并不容易.

原论文[10]仅将复杂度分析到 `meld` 和 `delete-min` 操作均为均摊 $O(\log n)$，但提出猜想认为其各操作都有和斐波那契堆相同的复杂度.

遗憾的是，后续发现，不维护额外信息的配对堆，在特定的操作序列下，`decrease-key` 操作的均摊复杂度下界至少为 $\Omega(\log\log n)$[11].

目前对复杂度上界比较好的估计有，Iacono 的 $O(1)$ `meld`，$O(\log n)$ `decrease-key`[12]；Pettie 的 $O(2^{2\sqrt{\log\log n}})$ `meld` 和 `decrease-key`[13]. 需要注意的是，前述复杂度均为均摊复杂度，因此不能对各结果分别取最小值.

参考文献

- https://en.wikipedia.org/wiki/Pairing_heap[6]
- https://brilliant.org/wiki/pairing-heap/[7]

[1] The pairing heap: a new form of self-adjusting heap

[2] The pairing heap: a new form of self-adjusting heap

[3] On the efficiency of pairing heaps and related data structures

[4] Improved upper bounds for pairing heaps

[5] Towards a Final Analysis of Pairing Heaps

[6] https://en.wikipedia.org/wiki/Pairing_heap

[7] https://brilliant.org/wiki/pairing-heap/

10.7.4 左偏树

Authors: JiZiQian, llleixx, firefly-zjyjoe

[10] The pairing heap: a new form of self-adjusting heap[2]
[11] On the efficiency of pairing heaps and related data structures[3]
[12] Improved upper bounds for pairing heaps[4]
[13] Towards a Final Analysis of Pairing Heaps[5]

什么是左偏树?

左偏树 与 **配对堆**→10.7.3@p2030 一样，是一种 **可并堆**，具有堆的性质，并且可以快速合并.

左偏树的定义和性质

对于一棵二叉树，我们定义 **外节点** 为子节点数小于两个的节点，定义一个节点的 dist 为其到子树中最近的外节点所经过的边的数量. 空节点的 dist 为 0.

注意

有些资料中对 dist 的定义是本文中的 dist 减 1，这样定义是因为代码编写时可以省略一些判空流程，但需要注意应预先置空节点的 dist 为 -1. 本文中所有代码对 dist 的定义 **均为空节点 dist 为 -1 的定义**，请注意与行文间 dist 定义的差别.

左偏树是一棵二叉树，它不仅具有堆的性质，并且是「左偏」的：每个节点左儿子的 dist 都大于等于右儿子的 dist.

因此，左偏树每个节点的 dist 都等于其右儿子的 dist 加一.

需要注意的是，dist 不是深度，**左偏树的深度没有保证**，一条向左的链也符合左偏树的定义.

核心操作：合并（merge）

合并两个堆时，由于要满足堆性质，先取值较小（为了方便，本文讨论小根堆）的那个根作为合并后堆的根节点，然后将这个根的左儿子作为合并后堆的左儿子，递归地合并其右儿子与另一个堆，作为合并后的堆的右儿子. 为了满足左偏性质，合并后若左儿子的 dist 小于右儿子的 dist，就交换两个儿子.

参考代码：

实现

```
int merge(int x, int y) {
  if (!x || !y) return x | y;  // 若一个堆为空则返回另一个堆
  if (t[x].val > t[y].val) swap(x, y);  // 取值较小的作为根
  t[x].rs = merge(t[x].rs, y);          // 递归合并右儿子与另一个堆
  if (t[t[x].rs].d > t[t[x].ls].d)
    swap(t[x].ls, t[x].rs);   // 若不满足左偏性质则交换左右儿子
  t[x].d = t[t[x].rs].d + 1;  // 更新dist
  return x;
}
```

由于左偏性质，每递归一层，其中一个堆根节点的 dist 就会减小 1，而一棵有 n 个节点的二叉树，根的 dist 不超过 $\lceil \log(n+1) \rceil$，所以合并两个大小分别为 n 和 m 的堆复杂度是 $O(\log n + \log m)$.

关于 dist 性质的证明

一棵根的 dist 为 x 的二叉树至少有 $x-1$ 层是满二叉树，那么就至少有 $2^x - 1$ 个节点. 注意这个性质是所有二叉树都具有的，并不是左偏树所特有的.

左偏树还有一种无需交换左右儿子的写法：将 dist 较大的儿子视作左儿子，dist 较小的儿子视作右儿子：

实现

```cpp
int& rs(int x) { return t[x].ch[t[t[x].ch[1]].d < t[t[x].ch[0]].d]; }

int merge(int x, int y) {
  if (!x || !y) return x | y;
  if (t[x].val < t[y].val) swap(x, y);
  int& rs_ref = rs(x);
  rs_ref = merge(rs_ref, y);
  t[x].d = t[rs(x)].d + 1;
  return x;
}
```

左偏树的其它操作

插入节点

单个节点也可以视为一个堆，合并即可.

删除根

合并根的左右儿子即可.

删除任意节点

做法

先将左右儿子合并，然后自底向上更新 dist、不满足左偏性质时交换左右儿子，当 dist 无需更新时结束递归:

实现

```cpp
int& rs(int x) { return t[x].ch[t[t[x].ch[1]].d < t[t[x].ch[0]].d]; }

// 有了 pushup，直接 merge 左右儿子就实现了删除节点并保持左偏性质
int merge(int x, int y) {
  if (!x || !y) return x | y;
  if (t[x].val < t[y].val) swap(x, y);
  int& rs_ref = rs(x);
  rs_ref = merge(rs_ref, y);
  t[rs_ref].fa = x;
  t[x].d = t[rs(x)].d + 1;
  return x;
}

void pushup(int x) {
  if (!x) return;
  if (t[x].d != t[rs(x)].d + 1) {
    t[x].d = t[rs(x)].d + 1;
    pushup(t[x].fa);
  }
}

void erase(int x) {
  int y = merge(t[x].ch[0], t[x].ch[1]);
  t[y].fa = t[x].fa;
```

```
  if (t[t[x].fa].ch[0] == x)
    t[t[x].fa].ch[0] = y;
  else if (t[t[x].fa].ch[1] == x)
    t[t[x].fa].ch[1] = y;
  pushup(t[y].fa);
}
```

复杂度证明

先考虑 `merge` 的过程，每次都会使 x 或 y 向下一层，也就是说最极端的情况，就是一直选择左偏树的右节点（dist 最小的节点）向下一层，此时 dist 减少了 1.

再考虑 `pushup` 的过程，我们令当前 `pushup` 的这个节点为 x，其父亲为 y，一个节点的「初始 dist」为它在 `pushup` 前的 dist. 从被删除节点的父亲开始递归，有两种情况：

1. x 是 y 的右儿子，此时 y 的初始 dist 为 x 的初始 dist 加一.
2. x 是 y 的左儿子，由于节点的 dist 最多减一，因此只有 y 的左右儿子初始 dist 相等时（此时左儿子 dist 减一会导致左右儿子互换）才会继续递归下去，因此 y 的初始 dist 仍然是 x 的初始 dist 加一.

所以，我们得到，每递归一层 x 的初始 dist 就会加一，因此最多递归 $O(\log n)$ 层.

整个堆加上/减去一个值、乘上一个正数

其实可以打标记且不改变相对大小的操作都可以.

在根打上标记，删除根/合并堆（访问儿子）时下传标记即可：

实现

```
int merge(int x, int y) {
  if (!x || !y) return x | y;
  if (t[x].val > t[y].val) swap(x, y);
  pushdown(x);
  t[x].rs = merge(t[x].rs, y);
  if (t[t[x].rs].d > t[t[x].ls].d) swap(t[x].ls, t[x].rs);
  t[x].d = t[t[x].rs].d + 1;
  return x;
}

int pop(int x) {
  pushdown(x);
  return merge(t[x].ls, t[x].rs);
}
```

其他可并堆

随机堆

实现

```
int merge(int x, int y) {
  if (!x || !y) return x | y;
  if (t[y].val < t[x].val) swap(x, y);
```

```
  if (rand() & 1)  // 随机选择是否交换左右子节点
    swap(t[x].ls, t[x].rs);
  t[x].ls = merge(t[x].ls, y);
  return x;
}
```

可以看到该实现方法唯一不同之处便是采用了随机数来实现合并，这样一来便可以省去 dist 的相关计算．且平均时间复杂度亦为 $O(\log n)$，详细证明可参考 Randomized Heap[1].

斜堆

斜堆是左偏树的自适应形式．当合并两个堆时，它无条件交换合并路径上的所有节点，以此试图维护平衡．根据均摊分析，自顶向下斜堆（top-down skew heap）插入，合并，删除最小值的复杂度为 $O(\log n)$[14].

例题

模板题

luogu P3377【模板】左偏树（可并堆）[3]

Monkey King[4]

罗马游戏[5]

需要注意的是：

1. 合并前要检查是否已经在同一堆中.
2. 左偏树的深度可能达到 $O(n)$，因此找一个点所在的堆顶要用并查集维护，不能直接暴力跳父亲.（虽然很多题数据水，暴力跳父亲可以过.....）（用并查集维护根时要保证原根指向新根，新根指向自己.）

罗马游戏参考代码

```
#include <iostream>

using namespace std;

constexpr int N = 1000010;

struct Node {
  int val, ls, rs, d;

  Node() {
    val = ls = rs = 0;
    d = -1;
  }

  Node(int v) {
    val = v;
    ls = rs = d = 0;
  }
} t[N];

```

[14] Self-Adjusting Heaps[2]

```cpp
int merge(int x, int y) {
  if (!x || !y) return x | y;
  if (t[x].val > t[y].val) swap(x, y);
  t[x].rs = merge(t[x].rs, y);
  if (t[t[x].rs].d > t[t[x].ls].d) swap(t[x].ls, t[x].rs);
  t[x].d = t[t[x].rs].d + 1;
  return x;
}

int f[N];

// 查找
int find(int x) { return x == f[x] ? x : f[x] = find(f[x]); }

bool kill[N];

int main() {
  ios::sync_with_stdio(false);
  cin.tie(nullptr);

  int n;
  cin >> n;
  for (int i = 1; i <= n; ++i) {
    int v;
    cin >> v;
    t[i] = Node(v);
    f[i] = i;
  }

  int m;
  cin >> m;
  int x, y;
  char op;
  while (m--) {
    cin >> op;
    if (op == 'M') {
      cin >> x >> y;
      if (kill[x] || kill[y]) continue;
      x = find(x);
      y = find(y);
      if (x != y) f[x] = f[y] = merge(x, y);
    } else {
      cin >> x;
      if (!kill[x]) {
        x = find(x);
        kill[x] = true;
        f[x] = f[t[x].ls] = f[t[x].rs] = merge(t[x].ls, t[x].rs);
        // 由于堆中的点会 find 到 x，所以 f[x] 也要修改
        cout << t[x].val << '\n';
      } else
        cout << "0\n";
    }
```

```
  }

  return 0;
}

```

树上问题

「APIO2012」派遣[6]

「JLOI2015」城池攻占[7]

这类题目往往是每个节点维护一个堆，与儿子合并，依题意弹出、修改、计算答案，有点像线段树合并的类似题目.

城池攻占参考代码

```cpp
#include <iostream>

using namespace std;

using ll = long long;

constexpr int N = 300010;

struct Node {
  int ls, rs, d;
  ll val, add, mul;

  Node() {
    ls = rs = 0;
    d = -1;
    val = add = 0;
    mul = 1;
  }

  Node(int v) {
    ls = rs = d = 0;
    val = v;
    add = 0;
    mul = 1;
  }
} t[N];

int head[N], nxt[N], to[N], cnt;
int n, m, p[N], f[N], a[N], dep[N], c[N], ans1[N],
    ans2[N];  // p 是树上每个点对应的堆顶
ll h[N], b[N];

void add(int u, int v) {
  nxt[++cnt] = head[u];
  head[u] = cnt;
  to[cnt] = v;
```

```
}

void madd(int u, ll x) {
  t[u].val += x;
  t[u].add += x;
}

void mmul(int u, ll x) {
  t[u].val *= x;
  t[u].add *= x;
  t[u].mul *= x;
}

void pushdown(int x) {  // 类似线段树下传标记
  mmul(t[x].ls, t[x].mul);
  madd(t[x].ls, t[x].add);
  mmul(t[x].rs, t[x].mul);
  madd(t[x].rs, t[x].add);
  t[x].add = 0;
  t[x].mul = 1;
}

int merge(int x, int y) {
  if (!x || !y) return x | y;
  if (t[x].val > t[y].val) swap(x, y);
  pushdown(x);
  t[x].rs = merge(t[x].rs, y);
  if (t[t[x].rs].d > t[t[x].ls].d) swap(t[x].ls, t[x].rs);
  t[x].d = t[t[x].rs].d + 1;
  return x;
}

int pop(int x) {
  pushdown(x);
  return merge(t[x].ls, t[x].rs);
}

void dfs(int u) {
  for (int i = head[u]; i; i = nxt[i]) {
    int v = to[i];
    dep[v] = dep[u] + 1;
    dfs(v);
  }
  while (p[u] && t[p[u]].val < h[u]) {
    ++ans1[u];
    ans2[p[u]] = dep[c[p[u]]] - dep[u];
    p[u] = pop(p[u]);
  }
  if (a[u])
    mmul(p[u], b[u]);
  else
    madd(p[u], b[u]);
```

```
  if (u > 1)
    p[f[u]] = merge(p[u], p[f[u]]);
  else
    while (p[u]) {
      ans2[p[u]] = dep[c[p[u]]] + 1;
      p[u] = pop(p[u]);
    }
}

int main() {
  ios::sync_with_stdio(false);
  cin.tie(nullptr);

  cin >> n >> m;

  for (int i = 1; i <= n; ++i) cin >> h[i];

  for (int i = 2; i <= n; ++i) {
    cin >> f[i] >> a[i] >> b[i];
    add(f[i], i);
  }

  for (int i = 1; i <= m; ++i) {
    int v;
    cin >> v >> c[i];
    t[i] = Node(v);
    p[c[i]] = merge(i, p[c[i]]);
  }

  dfs(1);

  for (int i = 1; i <= n; ++i) cout << ans1[i] << '\n';
  for (int i = 1; i <= m; ++i) cout << ans2[i] << '\n';

  return 0;
}

```

「SCOI2011」棘手的操作[8]

首先，找一个节点所在堆的堆顶要用并查集，而不能暴力向上跳.

再考虑单点查询，若用普通的方法打标记，就得查询点到根路径上的标记之和，最坏情况下可以达到 $O(n)$ 的复杂度. 如果只有堆顶有标记，就可以快速地查询了，但如何做到呢?

可以用类似启发式合并的方式，每次合并的时候把较小的那个堆标记暴力下传到每个节点，然后把较大的堆的标记作为合并后的堆的标记. 由于合并后有另一个堆的标记，所以较小的堆下传标记时要下传其标记减去另一个堆的标记. 由于每个节点每被合并一次所在堆的大小至少乘二，所以每个节点最多被下放 $O(\log n)$ 次标记，暴力下放标记的总复杂度就是 $O(n \log n)$.

再考虑单点加，先删除，再更新，最后插入即可.

然后是全局最大值，可以用一个平衡树/支持删除任意节点的堆（如左偏树）/multiset 来维护每个堆的堆顶.

所以，每个操作分别如下：

1. 暴力下传点数较小的堆的标记，合并两个堆，更新 size、tag，在 multiset 中删去合并后不在堆顶的那个原堆顶.
2. 删除节点，更新值，插入回来，更新 multiset. 需要分删除节点是否为根来讨论一下.
3. 堆顶打标记，更新 multiset.
4. 打全局标记.
5. 查询值 + 堆顶标记 + 全局标记.
6. 查询根的值 + 堆顶标记 + 全局标记.
7. 查询 multiset 最大值 + 全局标记.

棘手的操作参考代码

```cpp
#include <iostream>
#include <set>

using namespace std;

constexpr int N = 300010;

struct Node {
  int val, ch[2], d, fa;

  Node() {
    val = ch[0] = ch[1] = 0;
    d = -1;
    fa = 0;
  }

  Node(int v) {
    val = v;
    ch[0] = ch[1] = d = fa = 0;
  }
} t[N];

int n, m, f[N], tag[N], siz[N], delta;
char op[10];
multiset<int> s;

int& rs(int x) { return t[x].ch[t[t[x].ch[1]].d < t[t[x].ch[0]].d]; }

int merge(int x, int y) {
  if (!x || !y) return x | y;
  if (t[x].val < t[y].val) swap(x, y);
  int& rs_ref = rs(x);
  rs_ref = merge(rs_ref, y);
  t[rs_ref].fa = x;
  t[x].d = t[rs(x)].d + 1;
  return x;
}

void pushdown(int x, int y) {
  if (!x) return;
```

```
  t[x].val += y;
  pushdown(t[x].ch[0], y);
  pushdown(t[x].ch[1], y);
}

int find(int x) { return x == f[x] ? x : f[x] = find(f[x]); }

int main() {
  ios::sync_with_stdio(false);
  cin.tie(nullptr);

  cin >> n;

  for (int i = 1; i <= n; ++i) {
    int v;
    cin >> v;
    t[i] = Node(v);
    f[i] = i;
    siz[i] = 1;
    s.insert(v);
  }

  cin >> m;

  while (m--) {
    cin >> op;
    if (op[0] == 'U') {
      int x, y;
      cin >> x >> y;
      x = find(x);
      y = find(y);
      if (x != y) {
        if (siz[x] > siz[y]) swap(x, y);
        pushdown(x, tag[x] - tag[y]);
        f[x] = f[y] = merge(x, y);
        if (f[x] == x) {
          s.erase(s.find(t[y].val + tag[y]));
          tag[x] = tag[y];
          siz[x] += siz[y];
          tag[y] = siz[y] = 0;
        } else {
          s.erase(s.find(t[x].val + tag[y]));
          siz[y] += siz[x];
          tag[x] = siz[x] = 0;
        }
      }
    } else if (op[0] == 'A') {
      if (op[1] == '1') {
        int x, v;
        cin >> x >> v;
        if (x == find(x)) {
          t[t[x].ch[0]].fa = t[t[x].ch[1]].fa = 0;
```

```
93            int y = merge(t[x].ch[0], t[x].ch[1]);
94            s.erase(s.find(t[x].val + tag[x]));
95            t[x].val += v;
96            t[x].fa = t[x].ch[0] = t[x].ch[1] = 0;
97            t[x].d = 1;
98            f[x] = f[y] = merge(x, y);
99            s.insert(t[f[x]].val + tag[x]);
100           if (f[x] == y) {
101             tag[y] = tag[x];
102             siz[y] = siz[x];
103             tag[x] = siz[x] = 0;
104           }
105         } else {
106           t[t[x].ch[0]].fa = t[t[x].ch[1]].fa = t[x].fa;
107           t[t[x].fa].ch[x == t[t[x].fa].ch[1]] = merge(t[x].ch[0],
    t[x].ch[1]);
108           t[x].val += v;
109           t[x].fa = t[x].ch[0] = t[x].ch[1] = 0;
110           t[x].d = 1;
111           int y = find(x);
112           f[x] = f[y] = merge(x, y);
113           if (f[x] == x) {
114             s.erase(s.find(t[y].val + tag[y]));
115             s.insert(t[x].val + tag[y]);
116             tag[x] = tag[y];
117             siz[x] = siz[y];
118             tag[y] = siz[y] = 0;
119           }
120         }
121       } else if (op[1] == '2') {
122         int x, v;
123         cin >> x >> v;
124         x = find(x);
125         s.erase(s.find(t[x].val + tag[x]));
126         tag[x] += v;
127         s.insert(t[x].val + tag[x]);
128       } else {
129         int v;
130         cin >> v;
131         delta += v;
132       }
133     } else {
134       if (op[1] == '1') {
135         int x;
136         cin >> x;
137         cout << t[x].val + tag[find(x)] + delta << '\n';
138       } else if (op[1] == '2') {
139         int x;
140         cin >> x;
141         x = find(x);
142         cout << t[x].val + tag[x] + delta << '\n';
143       } else {
```

```
144             cout << *s.rbegin() + delta << '\n';
145           }
146         }
147       }
148
149       return 0;
150     }
151
```

「BOI2004」Sequence 数字序列[9]

这是一道论文题，详见《黄源河 -- 左偏树的特点及其应用》[10].

参考资料

[1] Randomized Heap 

[2] Self-Adjusting Heaps 

[3] luogu P3377【模板】左偏树（可并堆） 

[4] Monkey King 

[5] 罗马游戏 

[6] 「APIO2012」派遣 

[7] 「JLOI2015」城池攻占 

[8] 「SCOI2011」棘手的操作 

[9] 「BOI2004」Sequence 数字序列 

[10]《黄源河 -- 左偏树的特点及其应用》 

10.8 块状数据结构

10.8.1 分块思想

Authors: Ir1d, HeRaNO, Xeonacid

简介

其实，分块是一种思想，而不是一种数据结构.

从 NOIP 到 NOI 到 IOI，各种难度的分块思想都有出现.

分块的基本思想是，通过对原数据的适当划分，并在划分后的每一个块上预处理部分信息，从而较一般的暴力算法取得更优的时间复杂度.

分块的时间复杂度主要取决于分块的块长，一般可以通过均值不等式求出某个问题下的最优块长，以及相应的时间复杂度.

分块是一种很灵活的思想，相较于树状数组和线段树，分块的优点是通用性更好，可以维护很多树状数组和线段树无法维护的信息.

当然，分块的缺点是渐近意义的复杂度，相较于线段树和树状数组不够好.

不过在大多数问题上，分块仍然是解决这些问题的一个不错选择.

下面是几个例子.

区间和

例题 LibreOJ 6280 数列分块入门 4[1]

给定一个长度为 n 的序列 $\{a_i\}$，需要执行 n 次操作．操作分为两种：

1. 给 $a_l \sim a_r$ 之间的所有数加上 x；
2. 求 $\sum_{i=l}^{r} a_i$．

$1 \le n \le 5 \times 10^4$

我们将序列按每 s 个元素一块进行分块，并记录每块的区间和 b_i．

$$\underbrace{a_1, a_2, ..., a_s}_{b_1}, \underbrace{a_{s+1}, ..., a_{2s}}_{b_2}, ..., \underbrace{a_{(s-1)\times s+1}, ..., a_n}_{b_{\frac{n}{s}}}$$

最后一个块可能是不完整的（因为 n 很可能不是 s 的倍数），但是这对于我们的讨论来说并没有太大影响．

首先看查询操作：

- 若 l 和 r 在同一个块内，直接暴力求和即可，因为块长为 s，因此最坏复杂度为 $O(s)$．
- 若 l 和 r 不在同一个块内，则答案由三部分组成：以 l 开头的不完整块，中间几个完整块，以 r 结尾的不完整块．对于不完整的块，仍然采用上面暴力计算的方法，对于完整块，则直接利用已经求出的 b_i 求和即可．这种情况下，最坏复杂度为 $O(\frac{n}{s}+s)$．

接下来是修改操作：

- 若 l 和 r 在同一个块内，直接暴力修改即可，因为块长为 s，因此最坏复杂度为 $O(s)$．
- 若 l 和 r 不在同一个块内，则需要修改三部分：以 l 开头的不完整块，中间几个完整块，以 r 结尾的不完整块．对于不完整的块，仍然是暴力修改每个元素的值（别忘了更新区间和 b_i），对于完整块，则直接修改 b_i 即可．这种情况下，最坏复杂度和仍然为 $O(\frac{n}{s}+s)$．

利用均值不等式可知，当 $\frac{n}{s}=s$，即 $s=\sqrt{n}$ 时，单次操作的时间复杂度最优，为 $O(\sqrt{n})$．

参考代码

```cpp
#include <cmath>
#include <iostream>
using namespace std;
int id[50005], len;
// id 表示块的编号, len=sqrt(n) , 即上述题解中的s, sqrt的时候时间复杂度最优
long long a[50005], b[50005], s[50005];

// a 数组表示数据数组, b 数组记录每个块的整体赋值情况, 类似于 lazy_tag, s
// 表示块内元素总和
void add(int l, int r, long long x) {  // 区间加法
  int sid = id[l], eid = id[r];
  if (sid == eid) {  // 在一个块中
    for (int i = l; i <= r; i++) a[i] += x, s[sid] += x;
    return;
  }
  for (int i = l; id[i] == sid; i++) a[i] += x, s[sid] += x;
  for (int i = sid + 1; i < eid; i++)
    b[i] += x, s[i] += len * x;  // 更新区间和数组(完整的块)
  for (int i = r; id[i] == eid; i--) a[i] += x, s[eid] += x;
  // 以上两行不完整的块直接简单求和,就OK
}
```

```
22
23 long long query(int l, int r, long long p) {  // 区间查询
24   int sid = id[l], eid = id[r];
25   long long ans = 0;
26   if (sid == eid) {  // 在一个块里直接暴力求和
27     for (int i = l; i <= r; i++) ans = (ans + a[i] + b[sid]) % p;
28     return ans;
29   }
30   for (int i = l; id[i] == sid; i++) ans = (ans + a[i] + b[sid]) % p;
31   for (int i = sid + 1; i < eid; i++) ans = (ans + s[i]) % p;
32   for (int i = r; id[i] == eid; i--) ans = (ans + a[i] + b[eid]) % p;
33   // 和上面的区间修改是一个道理
34   return ans;
35 }
36
37 int main() {
38   int n;
39   cin >> n;
40   len = sqrt(n);  // 均值不等式可知复杂度最优为根号n
41   for (int i = 1; i <= n; i++) {  // 题面要求
42     cin >> a[i];
43     id[i] = (i - 1) / len + 1;
44     s[id[i]] += a[i];
45   }
46   for (int i = 1; i <= n; i++) {
47     int op, l, r, c;
48     cin >> op >> l >> r >> c;
49     if (op == 0)
50       add(l, r, c);
51     else
52       cout << query(l, r, c + 1) << endl;
53   }
54   return 0;
55 }
56
57 /*
58 https://loj.ac/s/1151495
59  */
60
```

区间和 2

上一个做法的复杂度是 $\Omega(1), O(\sqrt{n})$.

我们在这里介绍一种 $O(\sqrt{n}) - O(1)$ 的算法.

为了 $O(1)$ 询问，我们可以维护各种前缀和.

然而在有修改的情况下，不方便维护，只能维护单个块内的前缀和.

以及整块作为一个单位的前缀和.

每次修改 $O(T + \frac{n}{T})$.

询问：涉及三部分，每部分都可以直接通过前缀和得到，时间复杂度 $O(1)$.

对询问分块

同样的问题，现在序列长度为 n，有 m 个操作.

如果操作数量比较少，我们可以把操作记下来，在询问的时候加上这些操作的影响.

假设最多记录 T 个操作，则修改 $O(1)$，询问 $O(T)$.

T 个操作之后，重新计算前缀和，$O(n)$.

总复杂度：$O(mT + n\frac{m}{T})$.

$T = \sqrt{n}$ 时，总复杂度 $O(m\sqrt{n})$.

其他问题

分块思想也可以应用于其他整数相关问题：寻找零元素的数量、寻找第一个非零元素、计算满足某个性质的元素个数等等.

还有一些问题可以通过分块来解决，例如维护一组允许添加或删除数字的集合，检查一个数是否属于这个集合，以及查找第 k 大的数. 要解决这个问题，必须将数字按递增顺序存储，并分割成多个块，每个块中包含 $\sqrt{n}$ 个数字. 每次添加或删除一个数字时，必须通过在相邻块的边界移动数字来重新分块.

一种很有名的离线算法 莫队算法→13.4.4.2@p3209，也是基于分块思想实现的.

练习题

- UVa - 12003 - Array Transformer[2]
- UVa - 11990 Dynamic Inversion[3]
- SPOJ - Give Away[4]
- Codeforces - Till I Collapse[5]
- Codeforces - Destiny[6]
- Codeforces - Holes[7]
- Codeforces - XOR and Favorite Number[8]
- Codeforces - Powerful array[9]
- SPOJ - DQUERY[10]

本页面主要译自博文 Sqrt-декомпозиция[11] 与其英文翻译版 Sqrt Decomposition[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] LibreOJ 6280 数列分块入门 4

[2] UVa - 12003 - Array Transformer

[3] UVa - 11990 Dynamic Inversion

[4] SPOJ - Give Away

[5] Codeforces - Till I Collapse

[6] Codeforces - Destiny

[7] Codeforces - Holes

[8] Codeforces - XOR and Favorite Number

[9] Codeforces - Powerful array

[10] SPOJ - DQUERY

[11] Sqrt-декомпозиция

[12] Sqrt Decomposition

10.8.2 块状数组

建立块状数组

块状数组，即把一个数组分为几个块，块内信息整体保存，若查询时遇到两边不完整的块直接暴力查询. 一般情况下，块的长度为 $O(\sqrt{n})$. 详细分析可以阅读 2017 年国家集训队论文中徐明宽的《非常规大小分块算法初探》.

下面直接给出一种建立块状数组的代码.

实现

```
num = sqrt(n);
for (int i = 1; i <= num; i++)
  st[i] = n / num * (i - 1) + 1, ed[i] = n / num * i;
ed[num] = n;
for (int i = 1; i <= num; i++) {
  for (int j = st[i]; j <= ed[i]; j++) {
    belong[j] = i;
  }
  size[i] = ed[i] - st[i] + 1;
}
```

其中 `st[i]` 和 `ed[i]` 为块的起点和终点，`size[i]` 为块的大小.

保存与修改块内信息

例题 1：教主的魔法[1]

两种操作：

1. 区间 $[x, y]$ 每个数都加上 z；
2. 查询区间 $[x, y]$ 内大于等于 z 的数的个数.

我们要询问一个块内大于等于一个数的数的个数，所以需要一个 `t` 数组对块内排序，`a` 为原来的（未被排序的）数组. 对于整块的修改，使用类似于标记永久化的方式，用 `delta` 数组记录现在块内整体加上的值. 设 q 为查询和修改的操作次数总和，则时间复杂度 $O(q\sqrt{n}\log n)$.

用 `delta` 数组记录每个块的整体赋值情况.

实现

```
void Sort(int k) {
  for (int i = st[k]; i <= ed[k]; i++) t[i] = a[i];
  sort(t + st[k], t + ed[k] + 1);
}

void Modify(int l, int r, int c) {
  int x = belong[l], y = belong[r];
  if (x == y)  // 区间在一个块内就直接修改
  {
    for (int i = l; i <= r; i++) a[i] += c;
    Sort(x);
    return;
  }
  for (int i = l; i <= ed[x]; i++) a[i] += c;      // 直接修改起始段
  for (int i = st[y]; i <= r; i++) a[i] += c;      // 直接修改结束段
  for (int i = x + 1; i < y; i++) delta[i] += c;  // 中间的块整体打上标记
  Sort(x);
  Sort(y);
}

int Answer(int l, int r, int c) {
```

```
  int ans = 0, x = belong[l], y = belong[r];
  if (x == y) {
    for (int i = l; i <= r; i++)
      if (a[i] + delta[x] >= c) ans++;
    return ans;
  }
  for (int i = l; i <= ed[x]; i++)
    if (a[i] + delta[x] >= c) ans++;
  for (int i = st[y]; i <= r; i++)
    if (a[i] + delta[y] >= c) ans++;
  for (int i = x + 1; i <= y - 1; i++)
    ans +=
        ed[i] - (lower_bound(t + st[i], t + ed[i] + 1, c - delta[i]) - t)
+ 1;
  // 用 lower_bound 找出中间每一个整块中第一个大于等于 c 的数的位置
  return ans;
}
```

例题 2：寒夜方舟

两种操作：

1. 区间 $[x,y]$ 每个数都变成 z；
2. 查询区间 $[x,y]$ 内小于等于 z 的数的个数.

用 `delta` 数组记录现在块内被整体赋值为何值. 当该块未被整体赋值时，用一个特殊值（如 `0x3f3f3f3f3f3f3f3fll`）加以表示. 对于边角块，查询前要 `pushdown`，把块内存的信息下放到每一个数上. 赋值之后记得重新 `sort` 一遍. 其他方面同上题.

实现

```
void Sort(int k) {
  for (int i = st[k]; i <= ed[k]; i++) t[i] = a[i];
  sort(t + st[k], t + ed[k] + 1);
}

void PushDown(int x) {
  if (delta[x] != 0x3f3f3f3f3f3f3f3fll)  // 用该值标记块内没有被整体赋值
    for (int i = st[x]; i <= ed[x]; i++) a[i] = t[i] = delta[x];
  delta[x] = 0x3f3f3f3f3f3f3f3fll;
}

void Modify(int l, int r, int c) {
  int x = belong[l], y = belong[r];
  PushDown(x);
  if (x == y) {
    for (int i = l; i <= r; i++) a[i] = c;
    Sort(x);
    return;
  }
  PushDown(y);
  for (int i = l; i <= ed[x]; i++) a[i] = c;
  for (int i = st[y]; i <= r; i++) a[i] = c;
```

```
  Sort(x);
  Sort(y);
  for (int i = x + 1; i < y; i++) delta[i] = c;
}

int Binary_Search(int l, int r, int c) {
  int ans = l - 1, mid;
  while (l <= r) {
    mid = (l + r) / 2;
    if (t[mid] <= c)
      ans = mid, l = mid + 1;
    else
      r = mid - 1;
  }
  return ans;
}

int Answer(int l, int r, int c) {
  int ans = 0, x = belong[l], y = belong[r];
  PushDown(x);
  if (x == y) {
    for (int i = l; i <= r; i++)
      if (a[i] <= c) ans++;
    return ans;
  }
  PushDown(y);
  for (int i = l; i <= ed[x]; i++)
    if (a[i] <= c) ans++;
  for (int i = st[y]; i <= r; i++)
    if (a[i] <= c) ans++;
  for (int i = x + 1; i <= y - 1; i++) {
    if (0x3f3f3f3f3f3f3f3fll == delta[i])
      ans += Binary_Search(st[i], ed[i], c) - st[i] + 1;
    else if (delta[i] <= c)
      ans += size[i];
  }
  return ans;
}
```

练习

1. 单点修改，区间查询[2]
2. 区间修改，区间查询[3]
3. 【模板】线段树 2[4]
4. 「Ynoi2019 模拟赛」Yuno loves sqrt technology III[5]
5. 「Violet」蒲公英[6]
6. 作诗[7]

外部参考资料

[1] 教主的魔法

[2] 单点修改，区间查询

[3] 区间修改，区间查询

[4] 【模板】线段树 2

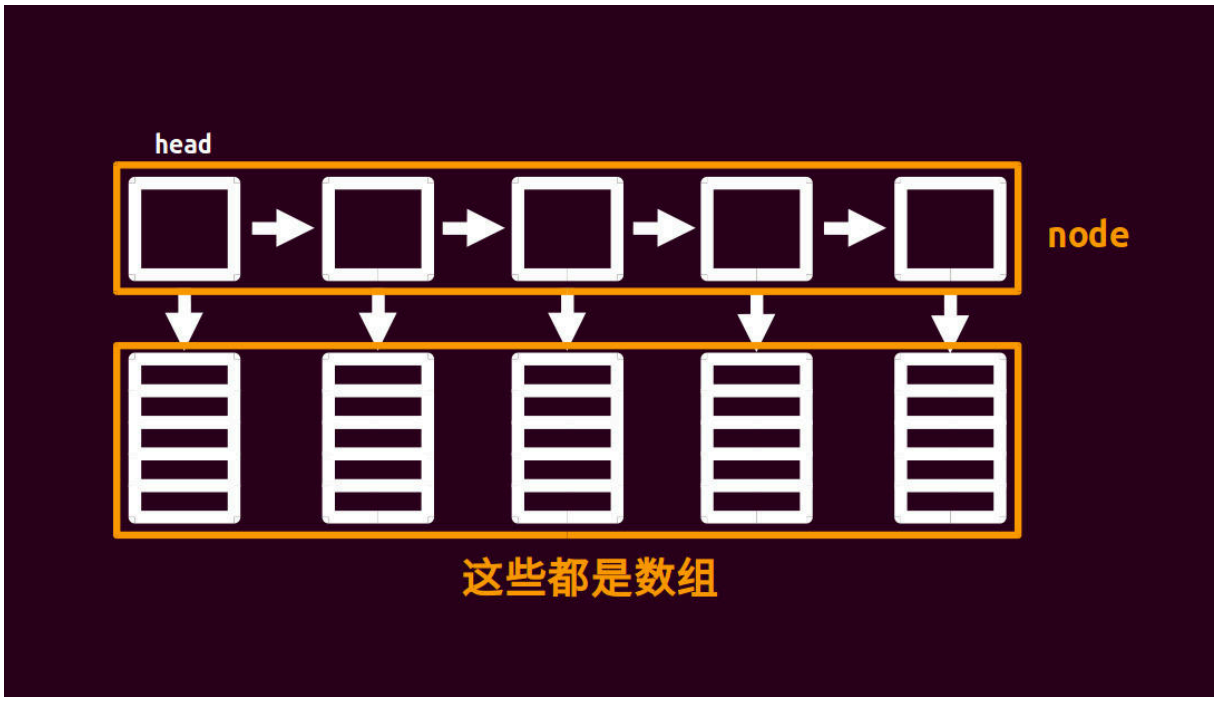
[5] 「Ynoi2019 模拟赛」Yuno loves sqrt technology III

[6] 「Violet」蒲公英

[7] 作诗

10.8.3 块状链表

Authors: HeRaNO, konnyakuxzy, littlefrog



块状链表大概就长这样……

不难发现块状链表就是一个链表，每个节点指向一个数组． 我们把原来长度为 n 的数组分为 $\sqrt{n}$ 个节点，每个节点对应的数组大小为 $\sqrt{n}$．所以我们这么定义结构体，代码见下．其中 `sqn` 表示 `sqrt(n)` 即 $\sqrt{n}$，`pb` 表示 `push_back`，即在这个 `node` 中加入一个元素．

实现

```
struct node {
  node* nxt;
  int size;
  char d[(sqn << 1) + 5];

  node() { size = 0, nxt = NULL, memset(d, 0, sizeof(d)); }

  void pb(char c) { d[size++] = c; }
};
```

块状链表应该至少支持：分裂、插入、查找． 什么是分裂？分裂就是分裂一个 `node`，变成两个小的 `node`，以保证每个 `node` 的大小都接近 $\sqrt{n}$（否则可能退化成普通数组）．当一个 `node` 的大小超过 $2\times\sqrt{n}$ 时执行分裂操作．

分裂操作怎么做呢？先新建一个节点，再把被分裂的节点的后 $\sqrt{n}$ 个值 `copy` 到新节点，然后把被分裂的节点的后 $\sqrt{n}$ 个值删掉（`size--`），最后把新节点插入到被分裂节点的后面即可.

块状链表的所有操作的复杂度都是 $\sqrt{n}$ 的.

还有一个要说的. 随着元素的插入（或删除），n 会变，$\sqrt{n}$ 也会变. 这样块的大小就会变化，我们难道还要每次维护块的大小?

其实不然，把 $\sqrt{n}$ 设置为一个定值即可. 比如题目给的范围是 10^6，那么 $\sqrt{n}$ 就设置为大小为 10^3 的常量，不用更改它.

```
1 list<vector<char>> orz_list;
```

libstdc++ 中的 `rope`

导入

libstdc++ 中的 `rope` 也起到块状链表的作用，它采用可持久化平衡树实现，可完成随机访问和插入、删除元素的操作.

由于 `rope` 并不是真正的用块状链表来实现，所以它的时间复杂度并不等同于块状链表，而是相当于可持久化平衡树的复杂度（即 $O(\log n)$）.

可以使用如下方法来引入：

```
1 #include <ext/rope>
2 using namespace __gnu_cxx;
```

> ⚠ **关于双下划线开头的库函数**
>
> OI 中，关于能否使用双下划线开头的库函数曾经一直不确定，2021 年 CCF 发布的 关于 NOI 系列活动中编程语言使用限制的补充说明[1] 中提到「允许使用以下划线开头的库函数或宏，但具有明确禁止操作的库函数和宏除外」. 故 `rope` 目前可以在 OI 中正常使用.

基本操作

操作	作用
`rope<int> a`	初始化 `rope`（与 `vector` 等容器很相似）
`a.push_back(x)`	在 `a` 的末尾添加元素 `x`
`a.insert(pos, x)`	在 `a` 的 `pos` 个位置添加元素 `x`
`a.erase(pos, x)`	在 `a` 的 `pos` 个位置删除 `x` 个元素
`a.at(x)` 或 `a[x]`	访问 `a` 的第 `x` 个元素
`a.length()` 或 `a.size()`	获取 `a` 的大小

例题

POJ2887 Big String[2]

题解： 很简单的模板题. 代码如下：

```
1 #include <cctype>
2 #include <cstring>
3 #include <iostream>
4 using namespace std;
5 constexpr int sqn = 1e3;
6
```

```
struct node {  // 定义块状链表
  node* nxt;
  int size;
  char d[(sqn << 1) + 5];

  node() { size = 0, nxt = NULL; }

  void pb(char c) { d[size++] = c; }
}* head = NULL;

char inits[(int)1e6 + 5];
int llen, q;

void readch(char& ch) {  // 读入字符
  do cin >> ch;
  while (!isalpha(ch));
}

void check(node* p) {  // 判断，记得要分裂
  if (p->size >= (sqn << 1)) {
    node* q = new node;
    for (int i = sqn; i < p->size; i++) q->pb(p->d[i]);
    p->size = sqn, q->nxt = p->nxt, p->nxt = q;
  }
}

void insert(char c, int pos) {  // 元素插入，借助链表来理解
  node* p = head;
  int tot, cnt;
  if (pos > llen++) {
    while (p->nxt != NULL) p = p->nxt;
    p->pb(c), check(p);
    return;
  }
  for (tot = head->size; p != NULL && tot < pos; p = p->nxt, tot += p->size);
  tot -= p->size, cnt = pos - tot - 1;
  for (int i = p->size - 1; i >= cnt; i--) p->d[i + 1] = p->d[i];
  p->d[cnt] = c, p->size++;
  check(p);
}

char query(int pos) {  // 查询
  node* p;
  int tot;
  for (p = head, tot = head->size; p != NULL && tot < pos;
       p = p->nxt, tot += p->size);
  tot -= p->size;
  return p->d[pos - tot - 1];
}

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  cin >> inits >> q;
  llen = strlen(inits);
```

```
  node* p = new node;
  head = p;
  for (int i = 0; i < llen; i++) {
    if (i % sqn == 0 && i) p->nxt = new node, p = p->nxt;
    p->pb(inits[i]);
  }
  char a;
  int k;
  while (q--) {
    readch(a);
    if (a == 'Q')
      cin >> k, cout << query(k) << '\n';
    else
      readch(a), cin >> k, insert(a, k);
  }
  return 0;
}

```

外部参考资料

[1] 关于 NOI 系列活动中编程语言使用限制的补充说明

[2] POJ2887 Big String

10.8.4 树分块

Authors: ouuan, Ir1d, Marcythm, Xeonacid

树分块的方式

可以参考 真 - 树上莫队→13.4.4.4@p3219.

也可以参考 ouuan 的博客/莫队、带修莫队、树上莫队详解/树上莫队[1].

树上莫队同样可以参考以上两篇文章.

树分块的应用

树分块除了应用于莫队，还可以灵活地运用到某些树上问题中. 但可以用树分块解决的题目往往都有更优秀的做法，所以相关的题目较少.

顺带提一句，「gty 的妹子树」的树分块做法可以被菊花图卡掉.

BZOJ4763 雪辉[2]

先进行树分块，然后对每个块的关键点，预处理出它到祖先中每个关键点的路径上颜色的 bitset，以及每个关键点的最近关键点祖先，复杂度是 $O(n\sqrt{n}+\frac{nc}{32})$，其中 $n\sqrt{n}$ 是暴力从每个关键点向上跳的复杂度，$\frac{nc}{32}$ 是把 $O(n)$ 个 `bitset` 存下来的复杂度.

回答询问的时候，先从路径的端点暴力跳到所在块的关键点，再从所在块的关键点一块一块地向上跳，直到 lca 所在块，然后再暴力跳到 lca. 关键点之间的 `bitset` 已经预处理了，剩下的在暴力跳的过程中计算. 单次询问复杂度是 $O(\sqrt{n}+\frac{c}{32})$，其中 $\sqrt{n}$ 是块内暴力跳以及块直接向上跳的复杂度，$O(\frac{c}{32})$ 是将预处理的结果与暴力跳的结果合并的复杂度. 数颜色个数可以用 `bitset` 的 `count()`，求 mex 可以用 `bitset` 的 `_Find_first()`.

所以，总复杂度为 $O((n+m)(\sqrt{n}+\frac{c}{32}))$.

参考代码

```cpp
#include <bitset>
#include <cctype>
#include <iostream>

#if defined(_MSC_VER) && !defined(__clang__)
#include <immintrin.h>
#endif

using namespace std;

constexpr int N = 100010;
constexpr int B = 666;
constexpr int C = 30000;

void add(int u, int v);
void dfs(int u);

int head[N], nxt[N << 1], to[N << 1], cnt;
int n, m, type, c[N], fa[N], dep[N], sta[N], top, tot, bl[N], key[N / B + 5],
    p[N], keyid[N];
bool vis[N];
bitset<C> bs[N / B + 5][N / B + 5], temp;

template <size_t N>
size_t find_first(std::bitset<N> b) {
#if defined(__GNUC__) && !defined(__clang__)
  return b._Find_first();
#elif defined(_MSC_VER) && !defined(__clang__)
  using word_t = decltype(b._Getword(0));
  constexpr ptrdiff_t word_len = CHAR_BIT * sizeof(word_t);
  constexpr ptrdiff_t words = N == 0 ? 0 : (N - 1) / word_len;
  size_t ans = 0;
  for (size_t i = 0; i <= words; ++i)
    if (b._Getword(i) != 0) {
      if (sizeof(word_t) == sizeof(unsigned int))
        return i * word_len + _tzcnt_u32(b._Getword(i));
      else
        return i * word_len + _tzcnt_u64(b._Getword(i));
    }
  return N;
#else
  auto s = b.to_string();
  for (size_t i = s.size() - 1; ~i; --i)
    if (s[i] & 1) return s.size() - 1 - i;
  return N;
#endif
}

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

```
  int i, u, v, x, y, k, lastans = 0;
  cin >> n >> m >> type;

  for (i = 1; i <= n; ++i) cin >> c[i];

  for (i = 1; i < n; ++i) {
    cin >> u >> v;
    add(u, v);
    add(v, u);
  }

  dfs(1);

  if (!tot) ++tot;
  if (keyid[key[tot]] == tot) keyid[key[tot]] = 0;
  key[tot] = 1;
  keyid[1] = tot;
  while (top) bl[sta[top--]] = tot;

  for (i = 1; i <= tot; ++i) {  // 预处理
    if (vis[key[i]]) continue;
    vis[key[i]] = true;
    temp.reset();
    for (u = key[i]; u; u = fa[u]) {
      temp[c[u]] = 1;
      if (keyid[u]) {
        if (!p[key[i]] && u != key[i]) p[key[i]] = u;
        bs[keyid[key[i]]][keyid[u]] = temp;
      }
    }
  }

  while (m--) {
    cin >> k;
    temp.reset();
    while (k--) {
      cin >> x >> y;
      u = x ^= lastans;
      v = y ^= lastans;

      while (key[bl[x]] != key[bl[y]]) {
        if (dep[key[bl[x]]] > dep[key[bl[y]]]) {
          if (x == u) {  // 若是第一次跳先暴力跳到关键点
            while (x != key[bl[u]]) {
              temp[c[x]] = 1;
              x = fa[x];
            }
          } else
            x = p[x];  // 否则跳一整块
        } else {
          if (y == v) {
            while (y != key[bl[v]]) {
```

```
103               temp[c[y]] = 1;
104               y = fa[y];
105             }
106           } else
107             y = p[y];
108         }
109       }
110
111       if (keyid[x]) temp |= bs[keyid[key[bl[u]]]][keyid[x]];
112       if (keyid[y]) temp |= bs[keyid[key[bl[v]]]][keyid[y]];
113
114       while (x != y) {
115         if (dep[x] > dep[y]) {
116           temp[c[x]] = 1;
117           x = fa[x];
118         } else {
119           temp[c[y]] = 1;
120           y = fa[y];
121         }
122       }
123       temp[c[x]] = true;
124     }
125     int ans1 = temp.count(), ans2 = find_first(~temp);
126     cout << ans1 << ' ' << ans2 << '\n';
127     lastans = (ans1 + ans2) * type;
128   }
129
130   return 0;
131 }
132
133 void dfs(int u) {  // 根据题意找点
134   int i, v, t = top;
135   for (i = head[u]; i; i = nxt[i]) {
136     v = to[i];
137     if (v == fa[u]) continue;
138     fa[v] = u;
139     dep[v] = dep[u] + 1;
140     dfs(v);
141     if (top - t >= B) {
142       key[++tot] = u;
143       if (!keyid[u]) keyid[u] = tot;
144       while (top > t) bl[sta[top--]] = tot;
145     }
146   }
147   sta[++top] = u;
148 }
149
150 void add(int u, int v) {
151   nxt[++cnt] = head[u];
152   head[u] = cnt;
153   to[cnt] = v;
154 }
```

BZOJ4812 由乃打扑克[3]

这题和上一题基本一样，唯一的区别是得到 `bitset` 后如何计算答案.

~~由于 BZOJ 是计算所有测试点总时限，不好卡，所以可以用 `_Find_next()` 水过去.~~

正解是每 16 位一起算，先预处理出 2^{16} 种可能的情况高位连续 1 的个数、低位连续 1 的个数以及中间的贡献. 只不过这样要手写 `bitset`，因为标准库的 `bitset` 不能取某 16 位……

代码可以参考 这篇博客[4].

外部参考资料

[1] ouuan 的博客/莫队、带修莫队、树上莫队详解/树上莫队

[2] BZOJ4763 雪辉

[3] BZOJ4812 由乃打扑克

[4] 这篇博客

10.8.5 Sqrt Tree

引入

给你一个长度为 n 的序列 $\langle a_i \rangle_{i=1}^{n}$，再给你一个满足结合律的运算 $\circ$（比如 $\gcd, \min, \max, +, \text{and}, \text{or}, \text{xor}$ 均满足结合律），然后对于每一次区间询问 $[l, r]$，我们需要计算 $a_l \circ a_{l+1} \circ \cdots \circ a_r$.

Sqrt Tree 可以在 $O(n \log \log n)$ 的时间内预处理，并在 $O(1)$ 的时间内回答询问.

解释

序列分块

首先我们把整个序列分成 $O(\sqrt{n})$ 个块，每一块的大小为 $O(\sqrt{n})$. 对于每个块，我们计算：

1. P_i 块内的前缀区间询问
2. S_i 块内的后缀区间询问
3. 维护一个额外的数组 $\langle B_{i,j} \rangle$ 表示第 i 个块到第 j 个块的区间答案.

举个例子，假设 $\circ$ 代表加法运算 $+$，序列为 $\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$.

首先我们将序列分成三块，变成了 $\{1, 2, 3\}, \{4, 5, 6\}, \{7, 8, 9\}$.

那么每一块的前缀区间答案和后缀区间答案分别为

$$P_1 = \{1, 3, 6\}, S_1 = \{6, 5, 3\}$$

$$P_2 = \{4, 9, 15\}, S_2 = \{15, 11, 6\}$$

$$P_3 = \{7, 15, 24\}, S_3 = \{24, 17, 9\}$$

B 数组为：

$$B = \begin{bmatrix} 6 & 21 & 45 \\ 0 & 15 & 39 \\ 0 & 0 & 24 \end{bmatrix}$$

（对于 $i > j$ 的不合法的情况我们假设答案为 0）

显然我们可以在 $O(n)$ 的时间内预处理这些值，空间复杂度同样是 $O(n)$ 的. 处理好之后，我们可以利用它们在 $O(1)$ 的时间内回答一些跨块的询问. 但对于那些整个区间都在一个块内的询问我们仍不能处理，因此我们还需要处理一些东西.

构建一棵树

容易想到我们在每个块内递归地构造上述结构以支持块内的查询. 对于大小为 1 的块我们可以 $O(1)$ 地回答询问. 这样我们就建出了一棵树，每一个结点代表序列的一个区间. 叶子结点的区间长度为 1 或 2.

一个大小为 k 的结点有 $O(\sqrt{k})$ 个子节点，于是整棵树的高度是 $O(\log\log n)$ 的，每一层的区间总长是 $O(n)$ 的，因此我们构建这棵树的复杂度是 $O(n\log\log n)$ 的.

树高度的证明

根据定义，设「控制」n 个元素的结点的子树高度为 $T(n)$，可以写出递归式：

$$T(n)=T(\sqrt{n})+1$$

作换元 $n=2^m$ 得

$$T(2^m)=T(2^{\frac{m}{2}})+1$$

再定义 $S(m)=T(2^m)$，代入有

$$S(m)=S(\frac{m}{2})+1$$

根据主定理，可知 $S(m)=O(\log m)$，因此 $T(n)=S(\log n)=O(\log\log n)$.

现在我们可以在 $O(\log\log n)$ 的时间内回答询问. 对于询问 $[l,r]$，我们只需要快速找到一个区间长度最小的结点 u 使得 u 能包含 $[l,r]$，这样 $[l,r]$ 在 u 的分块区间中一定是跨块的，就可以 $O(1)$ 地计算答案了. 查询一次的总体复杂度是 $O(\log\log n)$，因为树高是 $O(\log\log n)$ 的. 不过我们仍可以优化这个过程.

优化询问复杂度

容易想到二分高度，然后可以 $O(1)$ 判断是否合法. 这样复杂度就变成了 $O(\log\log\log n)$. 不过我们仍可以进一步加速这一过程.

我们假设

1. 每一块的大小都是 2 的整数幂次；
2. 每一层上的块大小是相同的.

为此我们需要在序列的末位补充一些 0 元素，使得它的长度变成 2 的整数次幂. 尽管有些块可能会变成原来的两倍大小，但这样仍是 $O(\sqrt{k})$ 的，于是预处理分块的复杂度仍是 $O(n)$ 的.

现在我们可以轻松地确定一个询问区间是否被整个地包含在一个块中. 对于区间 $[l,r]$（以 0 为起点），我们把端点写为二进制形式. 举一个例子，对于 $k=4,l=39,r=46$，二进制表示为

$$l=39_{10}=100111_2,r=46_{10}=101110_2$$

我们知道每一层的区间长度是相同的，而分块的大小也是相同的（在上述示例中 $2^k=2^4=16$）. 这些块完全覆盖了整个序列，因此第一块代表的元素为 $[0,15]$（二进制表示为 $[000000_2,001111_2]$），第二个块代表的元素区间为 $[16,31]$（二进制表示为 $[010000_2,011111_2]$），以此类推. 我们发现这些在同一个块内的元素的位置在二进制上只有后 k 位不同（上述示例中 $k=4$）. 而示例的 l,r 也只有后 k 位不同，因此他们在同一个块中.

因此我们需要检查区间两个端点是否只有后 k 位不同，即 $l\oplus r\le 2^k-1$. 因此我们可以快速找到答案区间所在的层：

1. 对于每个 $i\in[1,n]$，我们找到 i 最高位上的 1；
2. 现在对于一个询问 $[l,r]$，我们计算 $l\oplus r$ 的最高位，这样就可以快速确定答案区间所在的层.

这样我们就可以在 $O(1)$ 的时间内回答询问啦.

更新元素的过程

我们可以在 Sqrt Tree 上更新元素，单点修改和区间修改都是支持的.

单点修改

考虑一次单点赋值操作 $a_x=val$，我们希望高效更新这个操作的信息.

朴素实现

首先我们来看看在做了一次单点修改后 Sqrt Tree 会变成什么样子.

考虑一个长度为 l 的结点以及对应的序列：$\langle P_i\rangle, \langle S_i\rangle, \langle B_{i,j}\rangle$．容易发现在 $\langle P_i\rangle$ 和 $\langle S_i\rangle$ 中都只有 $O(\sqrt{l})$ 个元素改变．而在 $\langle B_{i,j}\rangle$ 中则有 $O(l)$ 个元素被改变．因此有 $O(l)$ 个元素在树上被更新．因此在 Sqrt Tree 上单点修改的复杂度是 $O(n+\sqrt{n}+\sqrt{\sqrt{n}}+\cdots)=O(n)$．

使用 Sqrt Tree 替代 B 数组

注意到单点更新的瓶颈在于更新根结点的 $\langle B_{i,j}\rangle$．因此我们尝试用另一个 Sqrt Tree 代替根结点的 $\langle B_{i,j}\rangle$，称其为 $index$．它的作用和原来的二维数组一样，维护整段询问的答案．其他非根结点仍然使用 $\langle B_{i,j}\rangle$ 维护．注意，如果一个 Sqrt Tree 根结点有 $index$ 结构，称其 Sqrt Tree 是 **含有索引** 的；如果一个 Sqrt Tree 的根结点有 $\langle B_{i,j}\rangle$ 结构，称其是 **没有索引** 的．而 $index$ 这棵树本身是没有索引的．

因此我们可以这样更新 $index$ 树：

1. 在 $O(\sqrt{n})$ 的时间内更新 $\langle P_i\rangle$ 和 $\langle S_i\rangle$．
2. 更新 $index$，它的长度是 $O(n)$ 的，但我们只需要更新其中的一个元素（这个元素代表了被改变的块），这一步的时间复杂度是 $O(\sqrt{n})$ 的（使用朴素实现的算法）．
3. 进入产生变化的子节点并使用朴素实现的算法在 $O(\sqrt{n})$ 的时间内更新信息．

注意，查询的复杂度仍是 $O(1)$ 的，因为我们最多使用 $index$ 树一次．于是单点修改的复杂度就是 $O(\sqrt{n})$ 的．

更新一个区间

Sqrt Tree 也支持区间覆盖操作 $\mathrm{Update}(l,r,x)$，即把区间 $[l,r]$ 的数全部变成 x．对此我们有两种实现方式，其中一种会花费 $O(\sqrt{n}\log\log n)$ 的复杂度更新信息，$O(1)$ 的时间查询；另一种则是 $O(\sqrt{n})$ 更新信息，但查询的时间会增加到 $O(\log\log n)$．

我们可以像线段树一样在 Sqrt Tree 上打懒标记．但是在 Sqrt Tree 上有一点不同．因为下传一个结点的懒标记，复杂度可能达到 $O(\sqrt{n})$，因此我们不是在询问的时候下传标记，而是看父节点是否有标记，如果有标记就把它下传．

第一种实现

在第一种实现中，我们只会给第 1 层的结点（结点区间长度为 $O(\sqrt{n})$）打懒标记，在下传标记的时候直接更新整个子树，复杂度为 $O(\sqrt{n}\log\log n)$．操作过程如下：

1. 考虑第 1 层上的结点，对于那些被修改区间完全包含的结点，给他们打一个懒标记；
2. 有两个块只有部分区间被覆盖，我们直接在 $O(\sqrt{n}\log\log n)$ 的时间内 **重建** 这两个块．如果它本身带有之前修改的懒标记，就在重建的时候顺便下传标记；
3. 更新根结点的 $\langle P_i\rangle$ 和 $\langle S_i\rangle$，时间复杂度 $O(\sqrt{n})$；
4. 重建 $index$ 树，时间复杂度 $O(\sqrt{n}\log\log n)$．

现在我们可以高效完成区间修改了．那么如何利用懒标记回答询问？操作如下：

1. 如果我们的询问被包含在一个有懒标记的块内，可以利用懒标记计算答案；
2. 如果我们的询问包含多个块，那么我们只需要关心最左边和最右边不完整块的答案．中间的块的答案可以在 $index$ 树中查询（因为 $index$ 树在每次修改完后会重建），复杂度是 $O(1)$．

因此询问的复杂度仍为 $O(1)$．

第二种实现

在这种实现中，每一个结点都可以被打上懒标记．因此在处理一个询问的时候，我们需要考虑祖先中的懒标记，那么查询的复杂度将变成 $O(\log\log n)$．不过更新信息的复杂度就会变得更快．操作如下：

1. 被修改区间完全包含的块，我们把懒标记添加到这些块上，复杂度 $O(\sqrt{n})$；
2. 被修改区间部分覆盖的块，更新 $\langle P_i\rangle$ 和 $\langle S_i\rangle$，复杂度 $O(\sqrt{n})$（因为只有两个被修改的块）；
3. 更新 $index$ 树，复杂度 $O(\sqrt{n})$（使用同样的更新算法）；
4. 对于没有索引的子树更新他们的 $\langle B_{i,j}\rangle$；
5. 递归地更新两个没有被完全覆盖的区间．

时间复杂度是 $O(\sqrt{n}+\sqrt{\sqrt{n}}+\cdots)=O(\sqrt{n})$．

实现

下面的实现在 $O(n \log \log n)$ 的时间内建树，在 $O(1)$ 的时间内回答询问，在 $O(\sqrt{n})$ 的时间内单点修改.

```
SqrtTreeItem op(const SqrtTreeItem &a, const SqrtTreeItem &b);

int log2Up(int n) {
  int res = 0;
  while ((1 << res) < n) {
    res++;
  }
  return res;
}

class SqrtTree {
 private:
  int n, lg, indexSz;
  vector<SqrtTreeItem> v;
  vector<int> clz, layers, onLayer;
  vector<vector<SqrtTreeItem>> pref, suf, between;

  void buildBlock(int layer, int l, int r) {
    pref[layer][l] = v[l];
    for (int i = l + 1; i < r; i++) {
      pref[layer][i] = op(pref[layer][i - 1], v[i]);
    }
    suf[layer][r - 1] = v[r - 1];
    for (int i = r - 2; i >= l; i--) {
      suf[layer][i] = op(v[i], suf[layer][i + 1]);
    }
  }

  void buildBetween(int layer, int lBound, int rBound, int betweenOffs) {
    int bSzLog = (layers[layer] + 1) >> 1;
    int bCntLog = layers[layer] >> 1;
    int bSz = 1 << bSzLog;
    int bCnt = (rBound - lBound + bSz - 1) >> bSzLog;
    for (int i = 0; i < bCnt; i++) {
      SqrtTreeItem ans;
      for (int j = i; j < bCnt; j++) {
        SqrtTreeItem add = suf[layer][lBound + (j << bSzLog)];
        ans = (i == j) ? add : op(ans, add);
        between[layer - 1][betweenOffs + lBound + (i << bCntLog) + j] = ans;
      }
    }
  }

  void buildBetweenZero() {
    int bSzLog = (lg + 1) >> 1;
    for (int i = 0; i < indexSz; i++) {
      v[n + i] = suf[0][i << bSzLog];
    }
    build(1, n, n + indexSz, (1 << lg) - n);
  }

  void updateBetweenZero(int bid) {
```

```
53      int bSzLog = (lg + 1) >> 1;
54      v[n + bid] = suf[0][bid << bSzLog];
55      update(1, n, n + indexSz, (1 << lg) - n, n + bid);
56    }
57
58    void build(int layer, int lBound, int rBound, int betweenOffs) {
59      if (layer >= (int)layers.size()) {
60        return;
61      }
62      int bSz = 1 << ((layers[layer] + 1) >> 1);
63      for (int l = lBound; l < rBound; l += bSz) {
64        int r = min(l + bSz, rBound);
65        buildBlock(layer, l, r);
66        build(layer + 1, l, r, betweenOffs);
67      }
68      if (layer == 0) {
69        buildBetweenZero();
70      } else {
71        buildBetween(layer, lBound, rBound, betweenOffs);
72      }
73    }
74
75    void update(int layer, int lBound, int rBound, int betweenOffs, int x) {
76      if (layer >= (int)layers.size()) {
77        return;
78      }
79      int bSzLog = (layers[layer] + 1) >> 1;
80      int bSz = 1 << bSzLog;
81      int blockIdx = (x - lBound) >> bSzLog;
82      int l = lBound + (blockIdx << bSzLog);
83      int r = min(l + bSz, rBound);
84      buildBlock(layer, l, r);
85      if (layer == 0) {
86        updateBetweenZero(blockIdx);
87      } else {
88        buildBetween(layer, lBound, rBound, betweenOffs);
89      }
90      update(layer + 1, l, r, betweenOffs, x);
91    }
92
93    SqrtTreeItem query(int l, int r, int betweenOffs, int base) {
94      if (l == r) {
95        return v[l];
96      }
97      if (l + 1 == r) {
98        return op(v[l], v[r]);
99      }
100     int layer = onLayer[clz[(l - base) ^ (r - base)]];
101     int bSzLog = (layers[layer] + 1) >> 1;
102     int bCntLog = layers[layer] >> 1;
103     int lBound = (((l - base) >> layers[layer]) << layers[layer]) + base;
104     int lBlock = ((l - lBound) >> bSzLog) + 1;
105     int rBlock = ((r - lBound) >> bSzLog) - 1;
106     SqrtTreeItem ans = suf[layer][l];
```

```
    if (lBlock <= rBlock) {
      SqrtTreeItem add =
          (layer == 0) ? (query(n + lBlock, n + rBlock, (1 << lg) - n, n))
                       : (between[layer - 1][betweenOffs + lBound +
                                             (lBlock << bCntLog) + rBlock]);
      ans = op(ans, add);
    }
    ans = op(ans, pref[layer][r]);
    return ans;
  }

 public:
  SqrtTreeItem query(int l, int r) { return query(l, r, 0, 0); }

  void update(int x, const SqrtTreeItem &item) {
    v[x] = item;
    update(0, 0, n, 0, x);
  }

  SqrtTree(const vector<SqrtTreeItem> &a)
      : n((int)a.size()), lg(log2Up(n)), v(a), clz(1 << lg), onLayer(lg + 1) {
    clz[0] = 0;
    for (int i = 1; i < (int)clz.size(); i++) {
      clz[i] = clz[i >> 1] + 1;
    }
    int tlg = lg;
    while (tlg > 1) {
      onLayer[tlg] = (int)layers.size();
      layers.push_back(tlg);
      tlg = (tlg + 1) >> 1;
    }
    for (int i = lg - 1; i >= 0; i--) {
      onLayer[i] = max(onLayer[i], onLayer[i + 1]);
    }
    int betweenLayers = max(0, (int)layers.size() - 1);
    int bSzLog = (lg + 1) >> 1;
    int bSz = 1 << bSzLog;
    indexSz = (n + bSz - 1) >> bSzLog;
    v.resize(n + indexSz);
    pref.assign(layers.size(), vector<SqrtTreeItem>(n + indexSz));
    suf.assign(layers.size(), vector<SqrtTreeItem>(n + indexSz));
    between.assign(betweenLayers, vector<SqrtTreeItem>((1 << lg) + bSz));
    build(0, 0, n, 0);
  }
};
```

习题

CodeChef - SEGPROD[1]

本页面主要译自 Sqrt Tree - Algorithms for Competitive Programming[2]，版权协议为 CC-BY-SA 4.0。

外部参考资料

[1] CodeChef - SEGPROD

[2] Sqrt Tree - Algorithms for Competitive Programming

10.9 单调栈

引入

何为单调栈？顾名思义，单调栈即满足单调性的栈结构．与单调队列相比，其只在一端进行进出．

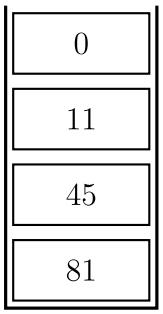
为了描述方便，以下举例及伪代码以维护一个整数的单调递增栈为例．

过程

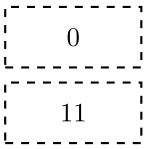
插入

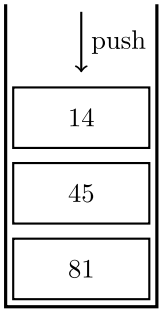
将一个元素插入单调栈时，为了维护栈的单调性，需要在保证将该元素插入到栈顶后整个栈满足单调性的前提下弹出最少的元素．

例如，栈中自顶向下的元素为 $\{0, 11, 45, 81\}$．



插入元素 14 时为了保证单调性需要依次弹出元素 $0, 11$，操作后栈变为 $\{14, 45, 81\}$．





用伪代码描述如下：

实现

```
insert x
```

```
while !sta.empty() && sta.top()<x
    sta.pop()
sta.push(x)
```

使用

自然就是从栈顶读出来一个元素，该元素满足单调性的某一端.

例如举例中取出的即栈中的最小值.

应用

POJ3250 Bad Hair Day[1]

有 N 头牛从左到右排成一排，每头牛有一个高度 h_i，设左数第 i 头牛与「它右边第一头高度 $\geq h_i$」的牛之间有 c_i 头牛，试求 $\sum_{i=1}^{N} c_i$.

比较基础的应用有这一题，就是个单调栈的简单应用，记录每头牛被弹出的位置，如果没有被弹出过则为最远端，稍微处理一下即可计算出题目所需结果.

另外，单调栈也可以用于离线解决 RMQ 问题.

我们可以把所有询问按右端点排序，然后每次在序列上从左往右扫描到当前询问的右端点处，并把扫描到的元素插入到单调栈中. 这样，每次回答询问时，单调栈中存储的值都是位置 $\leq r$ 的、可能成为答案的决策点，并且这些元素满足单调性质. 此时，单调栈上第一个位置 $\geq l$ 的元素就是当前询问的答案，这个过程可以用二分查找实现. 使用单调栈解决 RMQ 问题的时间复杂度为 $O(q \log q + q \log n)$，空间复杂度为 $O(n)$.

习题

- 洛谷 P5788【模板】单调栈[2]
- 洛谷 P1901 发射站[3]

外部参考资料

[1] POJ3250 Bad Hair Day

[2] 洛谷 P5788【模板】单调栈

[3] 洛谷 P1901 发射站

10.10 单调队列

Authors: Link-cute, Xeonacid, ouuan, Alphnia, Lyccrius

引入

在学习单调队列前，让我们先来看一道例题.

例题

Sliding Window[1]

本题大意是给出一个长度为 n 的数组，编程输出每 k 个连续的数中的最大值和最小值.

最暴力的想法很简单，对于每一段 $i \sim i+k-1$ 的序列，逐个比较来找出最大值（和最小值），时间复杂度约为 $O(n \times k)$.

很显然，这其中进行了大量重复工作，除了开头 $k-1$ 个和结尾 $k-1$ 个数之外，每个数都进行了 k 次比较，而题中 100% 的数据为 $n \leq 1000000$，当 k 稍大的情况下，显然会 TLE.

这时所用到的就是单调队列了.

定义

顾名思义，单调队列的重点分为「单调」和「队列」.

「单调」指的是元素的「规律」——递增（或递减）.

「队列」指的是元素只能从队头和队尾进行操作.

Ps. 单调队列中的 "队列" 与正常的队列有一定的区别，稍后会提到

例题分析

解释

有了上面「单调队列」的概念，很容易想到用单调队列进行优化.

要求的是每连续的 k 个数中的最大（最小）值，很明显，当一个数进入所要 "寻找" 最大值的范围中时，若这个数比其前面（先进队）的数要大，显然，前面的数会比这个数先出队且不再可能是最大值.

也就是说——当满足以上条件时，可将前面的数 "弹出"，再将该数真正 push 进队尾.

这就相当于维护了一个递减的队列，符合单调队列的定义，减少了重复的比较次数，不仅如此，由于维护出的队伍是查询范围内的且是递减的，队头必定是该查询区域内的最大值，因此输出时只需输出队头即可.

显而易见的是，在这样的算法中，每个数只要进队与出队各一次，因此时间复杂度被降到了 $O(n)$.

而由于查询区间长度是固定的，超出查询空间的值再大也不能输出，因此还需要 site 数组记录第 i 个队中的数在原数组中的位置，以弹出越界的队头.

过程

例如我们构造一个单调递增的队列会如下：

原序列为：

```
1 3 -1 -3 5 3 6 7
```

因为我们始终要维护队列保证其 **递增** 的特点，所以会有如下的事情发生：（假设 $k=3$）

操作	队列状态
1 入队	{1}
3 比 1 大，3 入队	{1 3}
-1 比队列中所有元素小，所以清空队列 -1 入队	{-1}
-3 比队列中所有元素小，所以清空队列 -3 入队	{-3}
5 比 -3 大，直接入队	{-3 5}
3 比 5 小，5 出队，3 入队	{-3 3}
-3 已经在窗体外，所以 -3 出队；6 比 3 大，6 入队	{3 6}
7 比 6 大，7 入队	{3 6 7}

例题参考代码

```
#include <cstdlib>
```

```cpp
#include <cstring>
#include <iostream>
constexpr int MAXN = 1000100;
using namespace std;
int q[MAXN], a[MAXN];
int n, k;

void getmin() {  // 得到这个队列里的最小值，直接找到最后的就行了
  int head = 0, tail = -1;
  for (int i = 1; i < k; i++) {
    while (head <= tail && a[q[tail]] >= a[i]) tail--;
    q[++tail] = i;
  }
  for (int i = k; i <= n; i++) {
    while (head <= tail && a[q[tail]] >= a[i]) tail--;
    q[++tail] = i;
    while (q[head] <= i - k) head++;
    cout << a[q[head]] << ' ';
  }
}

void getmax() {  // 和上面同理
  int head = 0, tail = -1;
  for (int i = 1; i < k; i++) {
    while (head <= tail && a[q[tail]] <= a[i]) tail--;
    q[++tail] = i;
  }
  for (int i = k; i <= n; i++) {
    while (head <= tail && a[q[tail]] <= a[i]) tail--;
    q[++tail] = i;
    while (q[head] <= i - k) head++;
    cout << a[q[head]] << ' ';
  }
}

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  cin >> n >> k;
  for (int i = 1; i <= n; i++) cin >> a[i];
  getmin();
  cout << '\n';
  getmax();
  cout << '\n';
  return 0;
}

```

Ps. 此处的 "队列" 跟普通队列的一大不同就在于可以从队尾进行操作，STL 中有类似的数据结构 deque.

例题 2 Luogu P2698 Flowerpot S[2]

给出 N 滴水的坐标，y 表示水滴的高度，x 表示它下落到 x 轴的位置．每滴水以每秒 1 个单位长度的速度下落．你需要把花盆放在 x 轴上的某个位置，使得从被花盆接着的第 1 滴水开始，到被花盆接着的最后 1 滴水结束，之间的时间差至少为 D．我们认为，只要水滴落到 x 轴上，与花盆的边沿对齐，就认为被接住．给出 N 滴水的坐标和 D 的大小，请算出最小的花盆的宽度 W．$1 \le N \le 100000, 1 \le D \le 1000000, 0 \le x, y \le 10^6$

将所有水滴按照 x 坐标排序之后，题意可以转化为求一个 x 坐标差最小的区间使得这个区间内 y 坐标的最大值和最小值之差至少为 D．我们发现这道题和上一道例题有相似之处，就是都与一个区间内的最大值最小值有关，但是这道题区间的大小不确定，而且区间大小本身还是我们要求的答案．

我们依然可以使用一个递增，一个递减两个单调队列在 R 不断后移时维护 $[L, R]$ 内的最大值和最小值，不过此时我们发现，如果 L 固定，那么 $[L, R]$ 内的最大值只会越来越大，最小值只会越来越小，所以设 $f(R) = \max[L, R] - \min[L, R]$，则 $f(R)$ 是个关于 R 的递增函数，故 $f(R) \ge D \Longrightarrow f(r) \ge D, R < r \le N$．这说明对于每个固定的 L，向右第一个满足条件的 R 就是最优答案．所以我们整体求解的过程就是，先固定 L，从前往后移动 R，使用两个单调队列维护 $[L, R]$ 的最值．当找到了第一个满足条件的 R，就更新答案并将 L 也向后移动．随着 L 向后移动，两个单调队列都需及时弹出队头．这样，直到 R 移到最后，每个元素依然是各进出队列一次，保证了 $O(n)$ 的时间复杂度．

参考代码

```cpp
#include <algorithm>
#include <iostream>
using namespace std;
constexpr int N = 100005;
using ll = long long;
int mxq[N], mnq[N];
int D, ans, n, hx, rx, hn, rn;

struct la {
  int x, y;

  bool operator<(const la &y) const { return x < y.x; }
} a[N];

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  cin >> n >> D;
  for (int i = 1; i <= n; ++i) cin >> a[i].x >> a[i].y;
  sort(a + 1, a + n + 1);
  hx = hn = 1;
  ans = 2e9;
  int L = 1;
  for (int i = 1; i <= n; ++i) {
    while (hx <= rx && a[mxq[rx]].y < a[i].y) rx--;
    mxq[++rx] = i;
    while (hn <= rn && a[mnq[rn]].y > a[i].y) rn--;
    mnq[++rn] = i;
    while (L <= i && a[mxq[hx]].y - a[mnq[hn]].y >= D) {
      ans = min(ans, a[i].x - a[L].x);
```

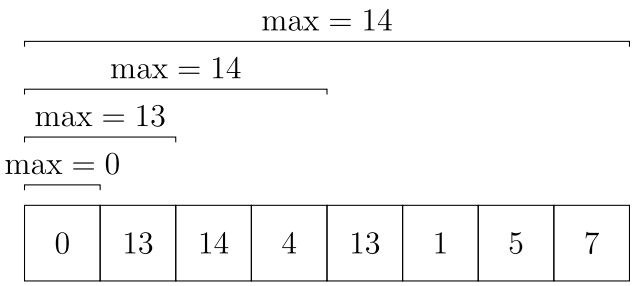
```
      L++;
      while (hx <= rx && mxq[hx] < L) hx++;
      while (hn <= rn && mnq[hn] < L) hn++;
    }
  }
  if (ans < 2e9)
    cout << ans << '\n';
  else
    cout << "-1\n";
  return 0;
}
```

外部参考资料

[1] Sliding Window  [2] Luogu P2698 Flowerpot S 

10.11 ST 表

定义



ST 表（Sparse Table，稀疏表）是用于解决 **可重复贡献问题** 的数据结构.

什么是可重复贡献问题?

可重复贡献问题 是指对于运算 opt，满足 $x \operatorname{opt} x = x$，则对应的区间询问就是一个可重复贡献问题. 例如，最大值有 $\max(x, x) = x$，gcd 有 $\gcd(x, x) = x$，所以 RMQ 和区间 GCD 就是一个可重复贡献问题. 像区间和就不具有这个性质，如果求区间和的时候采用的预处理区间重叠了，则会导致重叠部分被计算两次，这是我们所不愿意看到的. 另外，opt 还必须满足结合律才能使用 ST 表求解.

什么是 RMQ?

RMQ 是英文 Range Maximum/Minimum Query 的缩写，表示区间最大（最小）值. 解决 RMQ 问题有很多种方法，可以参考 RMQ 专题→14.1@p3342.

引入

> **Luogu P3865【模板】ST 表 & RMQ 问题**[1]
>
> 给定 n（$1 \le n \le 10^5$）个整数，有 m（$1 \le m \le 2 \times 10^6$）个询问，对于每个询问，你需要回答区间 $[l, r]$ 中的最大值.

考虑暴力做法. 每次都对区间 $[l, r]$ 扫描一遍，求出最大值.

显然，这个算法会超时.

ST 表

ST 表基于 倍增→5.10@p645 思想，可以做到 $\Theta(n \log n)$ 预处理，$\Theta(1)$ 回答每个询问. 但是不支持修改操作.

基于倍增思想，我们考虑如何求出区间最大值. 可以发现，如果按照一般的倍增流程，每次跳 2^i 步的话，询问时的复杂度仍旧是 $\Theta(\log n)$，并没有比线段树更优，反而预处理一步还比线段树慢.

我们发现 $\max(x, x) = x$，也就是说，区间最大值是一个具有「可重复贡献」性质的问题. 即使用来求解的预处理区间有重叠部分，只要这些区间的并是所求的区间，最终计算出的答案就是正确的.

如果手动模拟一下, 可以发现我们能使用至多两个预处理过的区间来覆盖询问区间, 也就是说询问时的时间复杂度可以被降至 $\Theta(1)$，在处理有大量询问的题目时十分有效.

具体实现如下:

令 $f(i, j)$ 表示区间 $[i, i + 2^j - 1]$ 的最大值.

显然 $f(i, 0) = a_i$.

根据定义式，第二维就相当于倍增的时候「跳了 $2^j - 1$ 步」，依据倍增的思路，写出状态转移方程: $f(i, j) = \max(f(i, j - 1), f(i + 2^{j-1}, j - 1))$.

$$\max\left(\boxed{\max(\{a_i, \ldots, a_{i+2^{j-1}-1}\})} \; \boxed{\max(\{a_{i+2^{j-1}}, \ldots, a_{i+2^j-1}\})}\right)$$

$$\| $$

$$\boxed{\max(\{a_i, \ldots, a_{i+2^j-1}\})}$$

以上就是预处理部分. 而对于查询，可以简单实现如下:

对于每个询问 $[l, r]$, 我们把它分成两部分: $[l, l + 2^s - 1]$ 与 $[r - 2^s + 1, r]$, 其中 $s = \lfloor \log_2(r - l + 1) \rfloor$. 两部分的结果的最大值就是回答.

$$\overbrace{\max = \max(\max_l, \max_r) = 14}$$

$$\overbrace{\max_r = f(4, 2) = 13}$$

$$\overbrace{\max_l = f(2, 2) = 14}$$

0	13	14	4	13	1	5	7

根据上面对于「可重复贡献问题」的论证，由于最大值是「可重复贡献问题」，重叠并不会对区间最大值产生影响. 又因为这两个区间完全覆盖了 $[l, r]$，可以保证答案的正确性.

Luogu P3865【模板】ST 表 & RMQ 问题[2] 参考实现

- C 风格

```c
#include <algorithm>
#include <iostream>
using namespace std;
constexpr int N = 100000 + 5;
constexpr int logN = 16;  // ⌊ log_2 N ⌋
int f[logN + 1][N], Logn[N];

// 初始化对数值
void pre() {
  Logn[2] = 1;
  for (int i = 3; i < N; i++) {
    Logn[i] = Logn[i / 2] + 1;
  }
}

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  pre();
  int n, m;
  cin >> n >> m;
  for (int i = 1; i <= n; i++) cin >> f[0][i];
  for (int j = 1; j <= logN; j++)
    for (int i = 1; i + (1 << j) - 1 <= n; i++)
      f[j][i] = max(f[j - 1][i], f[j - 1][i + (1 << (j - 1))]);  // ST 表具体实现
  for (int i = 1; i <= m; i++) {
    int x, y;
    cin >> x >> y;
    int s = Logn[y - x + 1];
    cout << max(f[s][x], f[s][y - (1 << s) + 1]) << '\n';
  }
  return 0;
}

```

- C++ 风格

```cpp
#include <algorithm>
#include <functional>
#include <iostream>
#include <vector>
#if defined(_MSC_VER) && !defined(__clang__)
#include <immintrin.h>
#define __builtin_clz _lzcnt_u32
#endif
using namespace std;

// 使用内建函数计算 ⌊ log_2 x ⌋
int lg2(int x) { return 31 - __builtin_clz(x); }

```

```cpp
template <typename T>
class SparseTable {
  using VT = vector<T>;
  using VVT = vector<VT>;
  using func_type = function<T(const T &, const T &)>;

  VVT ST;

  static T default_func(const T &t1, const T &t2) { return max(t1, t2); }

  func_type op;

 public:
  SparseTable(const vector<T> &v, func_type _func = default_func) {
    op = _func;
    int n = v.size(), l = lg2(n);
    ST.assign(l + 1, VT(n, 0));
    for (int i = 0; i < n; ++i) {
      ST[0][i] = v[i];
    }
    for (int j = 1; j <= l; ++j) {
      for (int i = 0; i + (1 << j) <= n; ++i) {
        ST[j][i] = op(ST[j - 1][i], ST[j - 1][i + (1 << (j - 1))]);
      }
    }
  }

  T query(int l, int r) {
    int q = lg2(r - l + 1);
    return op(ST[q][l], ST[q][r - (1 << q) + 1]);
  }
};

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  int n, m;
  cin >> n >> m;
  vector<int> a(n);
  for (int &i : a) cin >> i;
  SparseTable<int> st(a);
  for (int i = 1; i <= m; ++i) {
    int x, y;
    cin >> x >> y;
    cout << st.query(x - 1, y - 1) << '\n';
  }
  return 0;
}

```

- Python

```python
import sys
```

```python

input = sys.stdin.readline


class SparseTable:
    def __init__(self, arr: list, func=min):
        self.func = func
        self.n = len(arr)
        self.log = [0] * (self.n + 1)

        for i in range(2, self.n + 1):
            self.log[i] = self.log[i // 2] + 1

        self.k = self.log[self.n]
        self.st = [[0] * (self.n) for _ in range(self.k + 1)]
        self.st[0] = arr

        for j in range(1, self.k + 1):
            i = 0
            while i + (1 << j) <= self.n:
                self.st[j][i] = self.func(
                    self.st[j - 1][i], self.st[j - 1][i + (1 << (j - 1))]
                )
                i += 1

    def query(self, left: int, right: int):
        j = self.log[right - left + 1]
        return self.func(self.st[j][left], self.st[j][right - (1 << j) + 1])


n, m = map(int, input().split())
a = list(map(int, input().split()))
st = SparseTable(a, max)
for _ in range(m):
    left, right = map(int, input().split())
    print(st.query(left - 1, right - 1))

```

注意点

1. 输入输出数据一般很多，建议开启输入输出优化.
2. 在预处理 ST 表时通常需要建立一个一维大小为 $\log n$，另一维大小为 n 的数组，此时应优先让大小为 $\log n$ 的维度作为第一维，以提升缓存局部性.
3. 每次用 std::log[3] 重新计算对数函数数值并不值得，建议利用 `__builtin_clz` 或 `__lg` 等内建函数进行计算. 如无法利用这些内建函数，也可以预处理对数函数数值. 预处理方式如下所示:

$$\begin{cases} \operatorname{Logn}[1] \leftarrow 0, \\ \operatorname{Logn}[i] \leftarrow \operatorname{Logn}\left[\frac{i}{2}\right] + 1. \end{cases}$$

ST 表维护其他信息

除 RMQ 以外，还有其它的「可重复贡献问题」. 例如「区间按位与」、「区间按位或」、「区间 GCD」，ST 表都能高效地解决.

需要注意的是，对于「区间 GCD」，ST 表的查询复杂度并没有比线段树更优（令值域为 w，ST 表的查询复杂度为 $\Theta(\log w)$，而线段树为 $\Theta(\log n + \log w)$，且值域一般是大于 n 的），但是 ST 表的预处理复杂度也没有比线段树更劣，而编程复杂度方面 ST 表比线段树简单很多.

如果分析一下，「可重复贡献问题」一般都带有某种类似 RMQ 的成分. 例如「区间按位与」就是每一位取最小值，而「区间 GCD」则是每一个质因数的指数取最小值.

总结

ST 表能较好的维护「可重复贡献」的区间信息（同时也应满足结合律），时间复杂度较低，代码量相对其他算法很小. 但是，ST 表能维护的信息非常有限，不能较好地扩展，并且不支持修改操作.

习题

- 「SCOI2007」降雨量[4]
- [USACO07JAN] 平衡的阵容 Balanced Lineup[5]

附录：ST 表求区间 GCD 的时间复杂度分析

在算法运行的时候，可能要经过 $\Theta(\log n)$ 次迭代. 每一次迭代都可能会使用 GCD 函数进行递归，令值域为 w，GCD 函数的时间复杂度最高是 $\Omega(\log w)$ 的，所以总时间复杂度看似有 $O(n \log n \log w)$.

但是，在 GCD 的过程中，每一次递归（除最后一次递归之外）都会使数列中的某个数至少减半，而数列中的数最多减半的次数为 $\log_2(w^n) = \Theta(n \log w)$，所以，GCD 的递归部分最多只会运行 $O(n \log w)$ 次. 再加上循环部分（以及最后一层递归）的 $\Theta(n \log n)$，最终时间复杂度则是 $O(n(\log w + \log n))$，由于可以构造数据使得时间复杂度为 $\Omega(n(\log w + \log n))$，所以最终的时间复杂度即为 $\Theta(n(\log w + \log n))$.

而查询部分的时间复杂度很好分析，考虑最劣情况，即每次询问都询问最劣的一对数，时间复杂度为 $\Theta(\log w)$. 因此，ST 表维护「区间 GCD」的时间复杂度为预处理 $\Theta(n(\log n + \log w))$，单次查询 $\Theta(\log w)$.

线段树的相应操作是预处理 $\Theta(n \log w)$，单次查询 $\Theta(\log n + \log w)$.

这并不是一个严谨的数学论证，更为严谨的附在下方：

更严谨的证明

理解本段，可能需要具备 时间复杂度→5.2.1@p544 的关于「势能分析法」的知识.

先分析预处理部分的时间复杂度：

设「待考虑数列」为在预处理 ST 表的时候当前层循环的数列. 例如，第零层的数列就是原数列，第一层的数列就是第零层的数列经过一次迭代之后的数列，即 `st[1..n][1]`，我们将其记为 A.

而势能函数就定义为「待考虑数列」中所有数的累乘的以二为底的对数. 即：$\Phi(A) = \log_2\left(\prod_{i=1}^{n} A_i\right)$.

在一次迭代中，所花费的时间相当于迭代循环所花费的时间与 GCD 所花费的时间之和. 其中，GCD 花费的时间有长有短. 最短可能只有两次甚至一次递归，而最长可能有 $O(\log w)$ 次递归. 但是，GCD 过程中，除最开头一层与最末一层以外，每次递归都会使「待考虑数列」中的某个结果至少减半. 即，$\Phi(A)$ 会减少至少 1，该层递归所用的时间可以被势能函数均摊.

同时，我们可以看到，$\Phi(A)$ 的初值最大为 $\log_2(w^n) = \Theta(n \log w)$，而 $\Phi(A)$ 不增．所以，ST 表预处理部分的时间复杂度为 $O(n(\log w + \log n))$．

外部参考资料

[1] Luogu P3865【模板】ST 表 & RMQ 问题

[2] Luogu P3865【模板】ST 表 & RMQ 问题

[3] std::log

[4] 「SCOI2007」降雨量

[5] [USACO07JAN] 平衡的阵容 Balanced Lineup

10.12 树状数组

Authors: HeRaNO, Zhoier, Ir1d, Xeonacid, wangdehu, ouuan, ranwen, ananbaobeichicun, Ycrpro, dbxxx-oi, HowieHz, y-kx-b

引入

树状数组是一种支持 **单点修改** 和 **区间查询** 的，代码量小的数据结构．

什么是「单点修改」和「区间查询」?

假设有这样一道题：

已知一个数列 a，你需要进行下面两种操作：

- 给定 x, y，将 $a[x]$ 自增 y．
- 给定 l, r，求解 $a[l...r]$ 的和．

其中第一种操作就是「单点修改」，第二种操作就是「区间查询」．

类似地，还有：「区间修改」、「单点查询」．它们分别的一个例子如下：

- 区间修改：给定 l, r, x，将 $a[l...r]$ 中的每个数都分别自增 x；
- 单点查询：给定 x，求解 $a[x]$ 的值．

注意到，区间问题一般严格强于单点问题，因为对单点的操作相当于对一个长度为 1 的区间操作．

普通树状数组维护的信息及运算要满足 **结合律** 且 **可差分**，如加法（和）、乘法（积）、异或等．

- 结合律：$(x \circ y) \circ z = x \circ (y \circ z)$，其中 $\circ$ 是一个二元运算符．
- 可差分：具有逆运算的运算，即已知 $x \circ y$ 和 x 可以求出 y．

需要注意的是：

- 模意义下的乘法若要可差分，需保证每个数都存在逆元（模数为质数时一定存在）；
- 例如 gcd，max 这些信息不可差分，所以不能用普通树状数组处理，但是：
  - 使用两个树状数组可以用于处理区间最值，见 Efficient Range Minimum Queries using Binary Indexed Trees[1]．
  - 本页面也会介绍一种支持不可差分信息查询的，$\Theta(\log^2 n)$ 时间复杂度的拓展树状数组．

事实上，树状数组能解决的问题是线段树能解决的问题的子集：树状数组能做的，线段树一定能做；线段树能做的，树状数组不一定可以．然而，树状数组的代码要远比线段树短，时间效率常数也更小，因此仍有学习价值．

有时，在差分数组和辅助数组的帮助下，树状数组还可解决更强的 **区间加单点值** 和 **区间加区间和** 问题．

树状数组

初步感受

先来举个例子：我们想知道 $a[1...7]$ 的前缀和，怎么做?

一种做法是：$a_1+a_2+a_3+a_4+a_5+a_6+a_7$，需要求 7 个数的和.

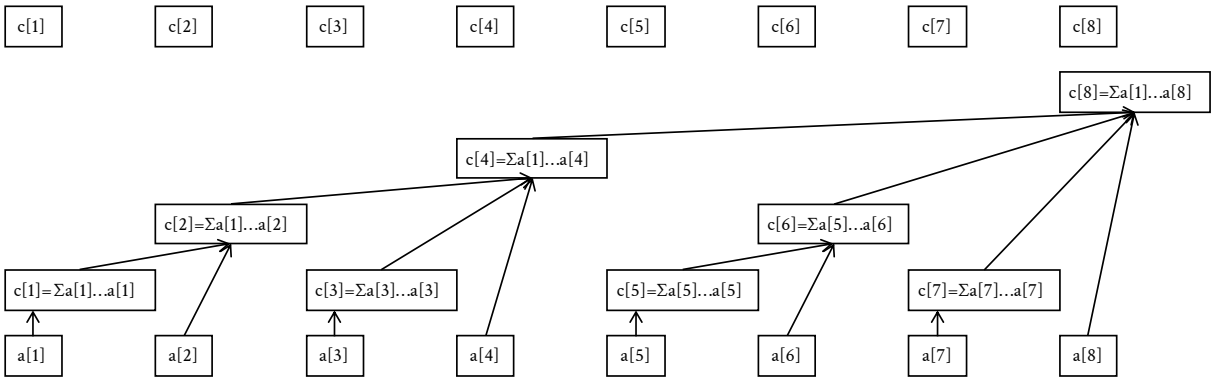
但是如果已知三个数 A，B，C，$A=a[1...4]$ 的和，$B=a[5...6]$ 的总和，$C=a[7...7]$ 的总和（其实就是 $a[7]$ 自己）. 你会怎么算？你一定会回答：$A+B+C$，只需要求 3 个数的和.

这就是树状数组能快速求解信息的原因：我们总能将一段前缀 $[1,n]$ 拆成 **不多于 $\log n$ 段区间**，使得这 $\log n$ 段区间的信息是 **已知的**.

于是，我们只需合并这 $\log n$ 段区间的信息，就可以得到答案. 相比于原来直接合并 n 个信息，效率有了很大的提高.

不难发现信息必须满足结合律，否则就不能像上面这样合并了.

下面这张图展示了树状数组的工作原理：



最下面的八个方块代表原始数据数组 a. 上面参差不齐的方块（与最上面的八个方块是同一个数组）代表数组 a 的上级——c 数组.

c 数组就是用来储存原始数组 a 某段区间的和的，也就是说，这些区间的信息是已知的，我们的目标就是把查询前缀拆成这些小区间.

例如，从图中可以看出：

- c_2 管辖的是 $a[1...2]$；
- c_4 管辖的是 $a[1...4]$；
- c_6 管辖的是 $a[5...6]$；
- c_8 管辖的是 $a[1...8]$；
- 剩下的 $c[x]$ 管辖的都是 $a[x]$ 自己（可以看做 $a[x...x]$ 的长度为 1 的小区间）.

不难发现，$c[x]$ 管辖的一定是一段右边界是 x 的区间总信息. 我们先不关心左边界，先来感受一下树状数组是如何查询的.

举例：计算 $a[1...7]$ 的和.

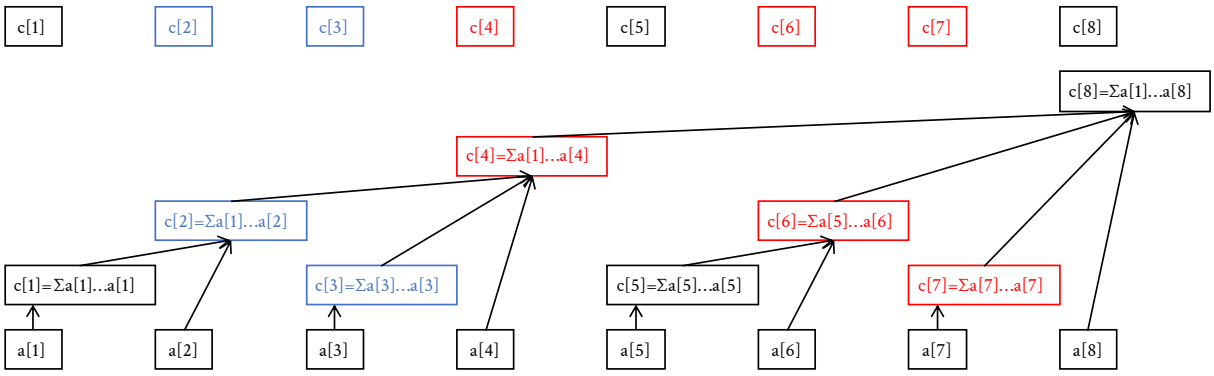
过程：从 c_7 开始往前跳，发现 c_7 只管辖 a_7 这个元素；然后找 c_6，发现 c_6 管辖的是 $a[5...6]$，然后跳到 c_4，发现 c_4 管辖的是 $a[1...4]$ 这些元素，然后再试图跳到 c_0，但事实上 c_0 不存在，不跳了.

我们刚刚找到的 c 是 c_7,c_6,c_4，事实上这就是 $a[1...7]$ 拆分出的三个小区间，合并得到答案是 $c_7+c_6+c_4$.

举例：计算 $a[4...7]$ 的和.

我们还是从 c_7 开始跳，跳到 c_6 再跳到 c_4. 此时我们发现它管理了 $a[1...4]$ 的和，但是我们不想要 $a[1...3]$ 这一部分，怎么办呢？很简单，减去 $a[1...3]$ 的和就行了.

那不妨考虑最开始，就将查询 $a[4...7]$ 的和转化为查询 $a[1...7]$ 的和，以及查询 $a[1...3]$ 的和，最终将两个结果作差.



查询 Σa[4]... a[7] = (Σa[1]...a[7]) - (Σa[1]...a[3])

管辖区间

那么问题来了，$c[x](x \geq 1)$ 管辖的区间到底往左延伸多少？也就是说，区间长度是多少？

树状数组中，规定 $c[x]$ 管辖的区间长度为 2^k，其中：

- 设二进制最低位为第 0 位，则 k 恰好为 x 二进制表示中，最低位的 `1` 所在的二进制位数；
- 2^k（$c[x]$ 的管辖区间长度）恰好为 x 二进制表示中，最低位的 `1` 以及后面所有 `0` 组成的数.

举个例子，c_{88} 管辖的是哪个区间?

因为 $88_{(10)} = 01011000_{(2)}$，其二进制最低位的 `1` 以及后面的 `0` 组成的二进制是 `1000`，即 8，所以 c_{88} 管辖 8 个 a 数组中的元素.

因此，c_{88} 代表 $a[81...88]$ 的区间信息.

我们记 x 二进制最低位 `1` 以及后面的 `0` 组成的数为 $\text{lowbit}(x)$，那么 $c[x]$ 管辖的区间就是 $[x - \text{lowbit}(x) + 1, x]$.

⚠ 注意

lowbit 指的不是最低位 `1` 所在的位数 k，而是这个 `1` 和后面所有 `0` 组成的 2^k.

怎么计算 `lowbit`？根据位运算知识，可以得到 `lowbit(x) = x & -x`.

lowbit 的原理

将 x 的二进制所有位全部取反，再加 1，就可以得到 `-x` 的二进制编码. 例如，6 的二进制编码是 `110`，全部取反后得到 `001`，加 1 得到 `010`.

设原先 `x` 的二进制编码是 `(...)10...00`，全部取反后得到 `[...]01...11`，加 `1` 后得到 `[...]10...00`，也就是 `-x` 的二进制编码了. 这里 `x` 二进制表示中第一个 `1` 是 `x` 最低位的 `1`.

`(...)` 和 `[...]` 中省略号的每一位分别相反，所以 `x & -x = (...)10...00 & [...]10...00 = 10...00`，得到的结果就是 `lowbit`.

实现

- C++

```cpp
int lowbit(int x) {
  // x 的二进制中，最低位的 1 以及后面所有 0 组成的数.
  // lowbit(0b01011000) == 0b00001000
  //           ~~~~^~~~
  // lowbit(0b01110010) == 0b00000010
  //           ~~~~~~~^~
  return x & -x;
}
```

- Python

```python
def lowbit(x):
    """
    x 的二进制中，最低位的 1 以及后面所有 0 组成的数.
    lowbit(0b01011000) == 0b00001000
              ~~~~~^~~
    lowbit(0b01110010) == 0b00000010
              ~~~~~~~^~
    """
    return x & -x
```

区间查询

接下来我们来看树状数组具体的操作实现，先来看区间查询.

回顾查询 $a[4...7]$ 的过程，我们是将它转化为两个子过程：查询 $a[1...7]$ 和查询 $a[1...3]$ 的和，最终作差.

其实任何一个区间查询都可以这么做：查询 $a[l...r]$ 的和，就是 $a[1...r]$ 的和减去 $a[1...l-1]$ 的和，从而把区间问题转化为前缀问题，更方便处理.

事实上，将有关 $l...r$ 的区间询问转化为 $1...r$ 和 $1...l-1$ 的前缀询问再差分，在竞赛中是一个非常常用的技巧.

那前缀查询怎么做呢？回顾下查询 $a[1...7]$ 的过程：

> 从 c_7 往前跳，发现 c_7 只管辖 a_7 这个元素；然后找 c_6，发现 c_6 管辖的是 $a[5...6]$，然后跳到 c_4，发现 c_4 管辖的是 $a[1...4]$ 这些元素，然后再试图跳到 c_0，但事实上 c_0 不存在，不跳了.
>
> 我们刚刚找到的 c 是 c_7, c_6, c_4，事实上这就是 $a[1...7]$ 拆分出的三个小区间，合并一下，答案是 $c_7 + c_6 + c_4$.

观察上面的过程，每次往前跳，一定是跳到现区间的左端点的左一位，作为新区间的右端点，这样才能将前缀不重不漏地拆分. 比如现在 c_6 管的是 $a[5...6]$，下一次就跳到 $5-1=4$，即访问 c_4.

我们可以写出查询 $a[1...x]$ 的过程：

- 从 $c[x]$ 开始往前跳，有 $c[x]$ 管辖 $a[x-\text{lowbit}(x)+1...x]$；
- 令 $x \leftarrow x-\text{lowbit}(x)$，如果 $x=0$ 说明已经跳到尽头了，终止循环；否则回到第一步.
- 将跳到的 c 合并.

实现时，我们不一定要先把 c 都跳出来然后一起合并，可以边跳边合并.

比如我们要维护的信息是和，直接令初始 $\text{ans}=0$，然后每跳到一个 $c[x]$ 就 $\text{ans} \leftarrow \text{ans}+c[x]$，最终 ans 就是所有合并的结果.

实现

- C++

```cpp
int getsum(int x) {  // a[1]..a[x]的和
  int ans = 0;
  while (x > 0) {
    ans = ans + c[x];
    x = x - lowbit(x);
  }
  return ans;
}
```

- Python

```python
def getsum(x):  # a[1]..a[x]的和
    ans = 0
    while x > 0:
        ans = ans + c[x]
        x = x - lowbit(x)
    return ans
```

树状数组与其树形态的性质

在讲解单点修改之前，先讲解树状数组的一些基本性质，以及其树形态来源，这有助于更好理解树状数组的单点修改.

我们约定：

- $l(x)=x-\text{lowbit}(x)+1$. 即，$l(x)$ 是 $c[x]$ 管辖范围的左端点.
- 对于任意正整数 x，总能将 x 表示成 $s\times 2^{k+1}+2^k$ 的形式，其中 $\text{lowbit}(x)=2^k$.
- 下面「$c[x]$ 和 $c[y]$ 不交」指 $c[x]$ 的管辖范围和 $c[y]$ 的管辖范围不相交，即 $[l(x),x]$ 和 $[l(y),y]$ 不相交.「$c[x]$ 包含于 $c[y]$」等表述同理.

性质 1：对于 $x\le y$，要么有 $c[x]$ 和 $c[y]$ 不交，要么有 $c[x]$ 包含于 $c[y]$.

证明

证明：假设 $c[x]$ 和 $c[y]$ 相交，即 $[l(x),x]$ 和 $[l(y),y]$ 相交，则一定有 $l(y)\le x\le y$.

将 y 表示为 $s\times 2^{k+1}+2^k$，则 $l(y)=s\times 2^{k+1}+1$. 所以，x 可以表示为 $s\times 2^{k+1}+b$，其中 $1\le b\le 2^k$.

不难发现 $\text{lowbit}(x)=\text{lowbit}(b)$. 又因为 $b-\text{lowbit}(b)\ge 0$,

所以 $l(x)=x-\text{lowbit}(x)+1=s\times 2^{k+1}+b-\text{lowbit}(b)+1\ge s\times 2^{k+1}+1=l(y)$，即 $l(y)\le l(x)\le x\le y$.

所以，如果 $c[x]$ 和 $c[y]$ 相交，那么 $c[x]$ 的管辖范围一定完全包含于 $c[y]$.

性质 2：$c[x]$ 真包含于 $c[x+\text{lowbit}(x)]$.

证明

证明：设 $y=x+\text{lowbit}(x)$，$x=s\times 2^{k+1}+2^k$，则 $y=(s+1)\times 2^{k+1}$，$l(x)=s\times 2^{k+1}+1$.

不难发现 $\text{lowbit}(y)\ge 2^{k+1}$，所以 $l(y)=(s+1)\times 2^{k+1}-\text{lowbit}(y)+1\le s\times 2^{k+1}+1=l(x)$，即 $l(y)\le l(x)\le x<y$.

所以，$c[x]$ 真包含于 $c[x+\text{lowbit}(x)]$.

性质 3：对于任意 $x<y<x+\text{lowbit}(x)$，有 $c[x]$ 和 $c[y]$ 不交。

证明

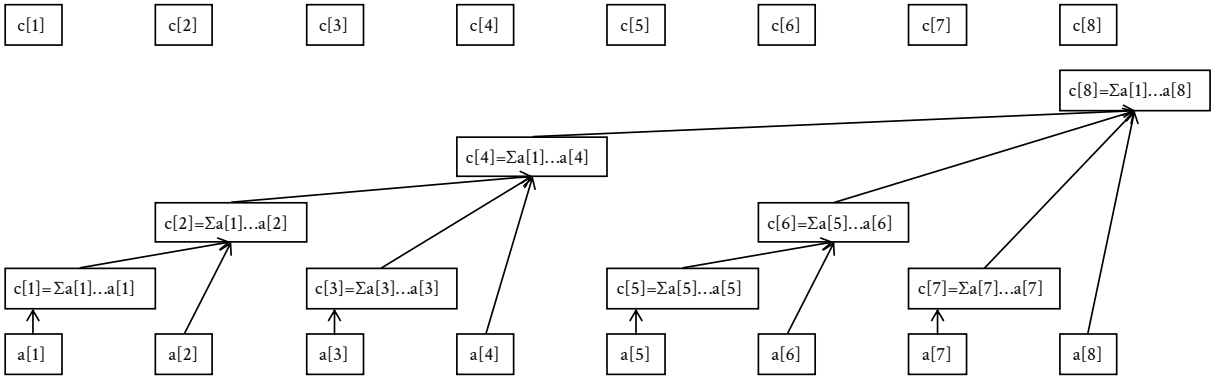
证明：设 $x=s\times 2^{k+1}+2^k$，则 $y=x+b=s\times 2^{k+1}+2^k+b$，其中 $1\le b<2^k$.

不难发现 $\text{lowbit}(y)=\text{lowbit}(b)$. 又因为 $b-\text{lowbit}(b)\ge 0$,

因此 $l(y)=y-\text{lowbit}(y)+1=x+b-\text{lowbit}(b)+1>x$，即 $l(x)\le x<l(y)\le y$.

所以，$c[x]$ 和 $c[y]$ 不交.

有了这三条性质的铺垫，我们接下来看树状数组的树形态（请忽略 a 向 c 的连边）.



事实上，树状数组的树形态是 x 向 $x+\text{lowbit}(x)$ 连边得到的图，其中 $x+\text{lowbit}(x)$ 是 x 的父亲.

注意，在考虑树状数组的树形态时，我们不考虑树状数组大小的影响，即我们认为这是一棵无限大的树，方便分析. 实际实现时，我们只需用到 $x\le n$ 的 $c[x]$，其中 n 是原数组长度.

这棵树天然满足了很多美好性质，下面列举若干（设 $fa[u]$ 表示 u 的直系父亲）：

- $u<fa[u]$.
- u 大于任何一个 u 的后代，小于任何一个 u 的祖先.
- 点 u 的 lowbit 严格小于 $fa[u]$ 的 lowbit.

证明

设 $y=x+\text{lowbit}(x)$，$x=s\times 2^{k+1}+2^k$，则 $y=(s+1)\times 2^{k+1}$，不难发现 $\text{lowbit}(y)\ge 2^{k+1}>\text{lowbit}(x)$，证毕.

- 点 x 的高度是 $\log_2\text{lowbit}(x)$，即 x 二进制最低位 1 的位数.

高度的定义

点 x 的高度 $h(x)$ 满足：如果 $x\bmod 2=1$，则 $h(x)=0$，否则 $h(x)=\max(h(y))+1$，其中 y 代表 x 的所有儿子（此时 x 至少存在一个儿子 $x-1$）.

也就是说，一个点的高度恰好比它最高的那个儿子再高 1. 如果一个点没有儿子，它的高度是 0.

这里引出高度这一概念，是为后面解释复杂度更方便.

- $c[u]$ 真包含于 $c[fa[u]]$（性质 2）.
- $c[u]$ 真包含于 $c[v]$，其中 v 是 u 的任一祖先（在上一条性质上归纳）.
- $c[u]$ 真包含 $c[v]$，其中 v 是 u 的任一后代（上面那条性质 u，v 颠倒）.

- 对于任意 $v' > u$，若 v' 不是 u 的祖先，则 $c[u]$ 和 $c[v']$ 不交.

证明

u 和 u 的祖先中，一定存在一个点 v 使得 $v < v' < fa[v]$，根据性质 3 得 $c[v']$ 不相交于 $c[v]$，而 $c[v]$ 包含 $c[u]$，因此 $c[v']$ 不交于 $c[u]$.

- 对于任意 $v < u$，如果 v 不在 u 的子树上，则 $c[u]$ 和 $c[v]$ 不交（上面那条性质 u，v' 颠倒）.
- 对于任意 $v > u$，当且仅当 v 是 u 的祖先，$c[u]$ 真包含于 $c[v]$（上面几条性质的总结）. 这就是树状数组单点修改的核心原理.
- 设 $u = s \times 2^{k+1} + 2^k$，则其儿子数量为 $k = \log_2 \operatorname{lowbit}(u)$，编号分别为 $u - 2^t (0 \le t < k)$.
  - 举例：假设 $k = 3$，u 的二进制编号为 `...1000`，则 u 有三个儿子，二进制编号分别为 `...0111`、`...0110`、`...0100`.

证明

在一个数 x 的基础上减去 2^t，x 二进制第 t 位会反转，而更低的位保持不变.

考虑 u 的儿子 v，有 $v + \operatorname{lowbit}(v) = u$，即 $v = u - 2^t$ 且 $\operatorname{lowbit}(v) = 2^t$. 设 $u = s \times 2^{k+1} + 2^k$.

考虑 $0 \le t < k$，u 的第 t 位及后方均为 0，所以 $v = u - 2^t$ 的第 t 位变为 1，后面仍为 0，**满足** $\operatorname{lowbit}(v) = 2^t$.

考虑 $t = k$，则 $v = u - 2^k$，v 的第 k 位变为 0，**不满足** $\operatorname{lowbit}(v) = 2^t$.

考虑 $t > k$，则 $v = u - 2^t$，v 的第 k 位是 1，所以 $\operatorname{lowbit}(v) = 2^k$，**不满足** $\operatorname{lowbit}(v) = 2^t$.

- u 的所有儿子对应 c 的管辖区间恰好拼接成 $[l(u), u - 1]$.
  - 举例：假设 $k = 3$，u 的二进制编号为 `...1000`，则 u 有三个儿子，二进制编号分别为 `...0111`、`...0110`、`...0100`.
  - `c[...0100]` 表示 `a[...0001 ~ ...0100]`.
  - `c[...0110]` 表示 `a[...0101 ~ ...0110]`.
  - `c[...0111]` 表示 `a[...0111 ~ ...0111]`.
  - 不难发现上面是三个管辖区间的并集恰好是 `a[...0001 ~ ...0111]`，即 $[l(u), u - 1]$.

证明

u 的儿子总能表示成 $u - 2^t (0 \le t < k)$，不难发现，t 越小，$u - 2^t$ 越大，代表的区间越靠右. 我们设 $f(t) = u - 2^t$，则 $f(k-1), f(k-2), ..., f(0)$ 分别构成 u 从左到右的儿子.

不难发现 $\operatorname{lowbit}(f(t)) = 2^t$，所以 $l(f(t)) = u - 2^t - 2^t + 1 = u - 2^{t+1} + 1$.

考虑相邻的两个儿子 $f(t+1)$ 和 $f(t)$. 前者管辖区间的右端点是 $f(t+1) = u - 2^{t+1}$，后者管辖区间的左端点是 $l(f(t)) = u - 2^{t+1} + 1$，恰好相接.

考虑最左面的儿子 $f(k-1)$，其管辖左边界 $l(f(k-1)) = u - 2^k + 1$ 恰为 $l(u)$.

考虑最右面的儿子 $f(0)$，其管辖右边界就是 $u - 1$.

因此，这些儿子的管辖区间可以恰好拼成 $[l(u), u - 1]$.

单点修改

现在来考虑如何单点修改 $a[x]$.

我们的目标是快速正确地维护 c 数组. 为保证效率，我们只需遍历并修改管辖了 $a[x]$ 的所有 $c[y]$，因为其他的 c 显然没有发生变化.

管辖 $a[x]$ 的 $c[y]$ 一定包含 $c[x]$（根据性质 1），所以 y 在树状数组树形态上是 x 的祖先．因此我们从 x 开始不断跳父亲，直到跳得超过了原数组长度为止．

设 n 表示 a 的大小，不难写出单点修改 $a[x]$ 的过程：

- 初始令 $x' = x$．
- 修改 $c[x']$．
- 令 $x' \leftarrow x' + \text{lowbit}(x')$，如果 $x' > n$ 说明已经跳到尽头了，终止循环；否则回到第二步．

区间信息和单点修改的种类，共同决定 $c[x']$ 的修改方式．下面给几个例子：

- 若 $c[x']$ 维护区间和，修改种类是将 $a[x]$ 加上 p，则修改方式则是将所有 $c[x']$ 也加上 p．
- 若 $c[x']$ 维护区间积，修改种类是将 $a[x]$ 乘上 p，则修改方式则是将所有 $c[x']$ 也乘上 p．

然而，单点修改的自由性使得修改的种类和维护的信息不一定是同种运算，比如，若 $c[x']$ 维护区间和，修改种类是将 $a[x]$ 赋值为 p，可以考虑转化为将 $a[x]$ 加上 $p - a[x]$．如果是将 $a[x]$ 乘上 p，就考虑转化为 $a[x]$ 加上 $a[x] \times p - a[x]$．

下面以维护区间和，单点加为例给出实现．

实现

- C++

```cpp
void add(int x, int k) {
  while (x <= n) {  // 不能越界
    c[x] = c[x] + k;
    x = x + lowbit(x);
  }
}
```

- Python

```python
def add(x, k):
    while x <= n:  # 不能越界
        c[x] = c[x] + k
        x = x + lowbit(x)
```

建树

也就是根据最开始给出的序列，将树状数组建出来（c 全部预处理好）．

一般可以直接转化为 n 次单点修改，时间复杂度 $\Theta(n \log n)$（复杂度分析在后面）．

比如给定序列 $a = (5, 1, 4)$ 要求建树，直接看作对 $a[1]$ 单点加 5，对 $a[2]$ 单点加 1，对 $a[3]$ 单点加 4 即可．

也有 $\Theta(n)$ 的建树方法，见本页面 $\Theta(n)$ 建树→10@p1977 一节．

复杂度分析

空间复杂度显然 $\Theta(n)$．

时间复杂度：

- 对于区间查询操作：整个 $x \leftarrow x - \text{lowbit}(x)$ 的迭代过程，可看做将 x 二进制中的所有 1，从低位到高位逐渐改成 0 的过程，拆分出的区间数等于 x 二进制中 1 的数量（即 $\text{popcount}(x)$）．因此，单次查询时间复杂度是 $\Theta(\log n)$；
- 对于单点修改操作：跳父亲时，访问到的高度一直严格增加，且始终有 $x \le n$．由于点 x 的高度是 $\log_2 \text{lowbit}(x)$，所以跳到的高度不会超过 $\log_2 n$，所以访问到的 c 的数量是 $\log n$ 级别．因此，单次单点修改复杂度是 $\Theta(\log n)$．

区间加区间和

前置知识：前缀和 & 差分→5.8@p617.

该问题可以使用两个树状数组维护差分数组解决.

考虑序列 a 的差分数组 d，其中 $d[i]=a[i]-a[i-1]$. 由于差分数组的前缀和就是原数组，所以 $a_i=\sum_{j=1}^{i} d_j$.

一样地，我们考虑将查询区间和通过差分转化为查询前缀和. 那么考虑查询 $a[1...r]$ 的和，即 $\sum_{i=1}^{r} a_i$，进行推导：

$$\sum_{i=1}^{r} a_i$$

$$=\sum_{i=1}^{r}\sum_{j=1}^{i} d_j$$

观察这个式子，不难发现每个 d_j 总共被加了 $r-j+1$ 次. 接着推导：

$$\sum_{i=1}^{r}\sum_{j=1}^{i} d_j$$

$$=\sum_{i=1}^{r} d_i\times(r-i+1)$$

$$=\sum_{i=1}^{r} d_i\times(r+1)-\sum_{i=1}^{r} d_i\times i$$

$\sum_{i=1}^{r} d_i$ 并不能推出 $\sum_{i=1}^{r} d_i\times i$ 的值，所以要用两个树状数组分别维护 d_i 和 $d_i\times i$ 的和信息.

那么怎么做区间加呢？考虑给原数组 $a[l...r]$ 区间加 x 给 d 带来的影响.

因为差分是 $d[i]=a[i]-a[i-1]$,

- $a[l]$ 多了 v 而 $a[l-1]$ 不变，所以 $d[l]$ 的值多了 v.
- $a[r+1]$ 不变而 $a[r]$ 多了 v，所以 $d[r+1]$ 的值少了 v.
- 对于不等于 l 且不等于 $r+1$ 的任意 i，$a[i]$ 和 $a[i-1]$ 要么都没发生变化，要么都加了 v，$a[i]+v-(a[i-1]+v)$ 还是 $a[i]-a[i-1]$，所以其它的 $d[i]$ 均不变.

那就不难想到维护方式了：对于维护 d_i 的树状数组，对 l 单点加 v，$r+1$ 单点加 $-v$；对于维护 $d_i\times i$ 的树状数组，对 l 单点加 $v\times l$，$r+1$ 单点加 $-v\times(r+1)$.

而更弱的问题，「区间加求单点值」，只需用树状数组维护一个差分数组 d_i. 询问 $a[x]$ 的单点值，直接求 $d[1...x]$ 的和即可.

这里直接给出「区间加区间和」的代码：

实现

- C++

```cpp
int t1[MAXN], t2[MAXN], n;

int lowbit(int x) { return x & (-x); }

void add(int k, int v) {
  int v1 = k * v;
  while (k <= n) {
    t1[k] += v, t2[k] += v1;
    // 注意不能写成 t2[k] += k * v，因为 k 的值已经不是原数组的下标了
```

```
    k += lowbit(k);
  }
}

int getsum(int *t, int k) {
  int ret = 0;
  while (k) {
    ret += t[k];
    k -= lowbit(k);
  }
  return ret;
}

void add1(int l, int r, int v) {
  add(l, v), add(r + 1, -v);  // 将区间加差分为两个前缀加
}

long long getsum1(int l, int r) {
  return (r + 1ll) * getsum(t1, r) - 1ll * l * getsum(t1, l - 1) -
         (getsum(t2, r) - getsum(t2, l - 1));
}
```

- Python

```
t1 = [0] * MAXN
t2 = [0] * MAXN
n = 0


def lowbit(x):
    return x & (-x)


def add(k, v):
    v1 = k * v
    while k <= n:
        t1[k] = t1[k] + v
        t2[k] = t2[k] + v1
        k = k + lowbit(k)


def getsum(t, k):
    ret = 0
    while k:
        ret = ret + t[k]
        k = k - lowbit(k)
    return ret


def add1(l, r, v):
    add(l, v)
    add(r + 1, -v)

```

```
30
31 def getsum1(l, r):
32     return (
33         (r) * getsum(t1, r)
34         - l * getsum(t1, l - 1)
35         - (getsum(t2, r) - getsum(t2, l - 1))
36     )
```

根据这个原理，应该可以实现「区间乘区间积」，「区间异或一个数，求区间异或值」等，只要满足维护的信息和区间操作是同种运算即可，感兴趣的读者可以自己尝试.

二维树状数组

单点修改，子矩阵查询

二维树状数组，也被称作树状数组套树状数组，用来维护二维数组上的单点修改和前缀信息问题.

与一维树状数组类似，我们用 $c(x,y)$ 表示 $a(x-\text{lowbit}(x)+1,y-\text{lowbit}(y)+1)...a(x,y)$ 的矩阵总信息，即一个以 $a(x,y)$ 为右下角，高 $\text{lowbit}(x)$，宽 $\text{lowbit}(y)$ 的矩阵的总信息.

对于单点修改，设:

$$f(x,i)=\begin{cases}x & i=0\\ f(x,i-1)+\text{lowbit}(f(x,i-1)) & i>0\end{cases}$$

即 $f(x,i)$ 为 x 在树状数组树形态上的第 i 级祖先（第 0 级祖先是自己）.

则只有 $c(f(x,i),f(y,j))$ 中的元素管辖 $a(x,y)$，修改 $a(x,y)$ 时只需修改所有 $c(f(x,i),f(y,j))$，其中 $f(x,i)\le n$，$f(y,j)\le m$.

正确性证明

$c(p,q)$ 管辖 $a(x,y)$，求 p 和 q 的取值范围.

考虑一个大小为 n 的一维树状数组 c_1（对应原数组 a_1）和一个大小为 m 的一维树状数组 c_2（对应原数组 a_2）.

则命题等价为：$c_1(p)$ 管辖 $a_1[x]$ 且 $c_2(q)$ 管辖 $a_2[y]$ 的条件.

也就是说，在树状数组树形态上，p 是 x 及其祖先中的一个点，q 是 y 及其祖先中的一个点.

所以 $p=f(x,i)$，$q=f(y,j)$.

对于查询，我们设:

$$g(x,i)=\begin{cases}x & i=0\\ g(x,i-1)-\text{lowbit}(g(x,i-1)) & i,g(x,i-1)>0\\ 0 & \text{otherwise.}\end{cases}$$

则合并所有 $c(g(x,i),g(y,j))$，其中 $g(x,i),g(y,j)>0$.

正确性证明

设 $\circ$ 表示合并两个信息的运算符（比如，如果信息是区间和，则 $\circ=+$）.

考虑一个一维树状数组 c_1，$c_1[g(x,0)]\circ c_1[g(x,1)]\circ c_1[g(x,2)]\circ\cdots$ 恰好表示原数组上 $[1...x]$ 这段区间信息.

类似地，设 $t(x)=c(x,g(y,0))\circ c(x,g(y,1))\circ c(x,g(y,2))\circ\cdots$，则 $t(x)$ 恰好表示 $a(x-\text{lowbit}(x)+1,1)...a(x,y)$ 这个矩阵信息.

> 又类似地，就有 $t(g(x,0))\circ t(g(x,1))\circ t(g(x,2))\circ\cdots$ 表示 $a(1,1)...a(x,y)$ 这个矩阵信息.
>
> 其实这里 $t(x)$ 这个函数如果看成一个树状数组，相当于一个树状数组套了一个树状数组，这也就是「树状数组套树状数组」这个名字的来源.

下面给出单点加、查询子矩阵和的代码.

实现

- 单点加

```
void add(int x, int y, int v) {
  for (int i = x; i <= n; i += lowbit(i)) {
    for (int j = y; j <= m; j += lowbit(j)) {
      // 注意这里必须得建循环变量，不能像一维数组一样直接 while (x <= n) 了
      c[i][j] += v;
    }
  }
}
```

- 查询子矩阵和

```
int sum(int x, int y) {
  int res = 0;
  for (int i = x; i > 0; i -= lowbit(i)) {
    for (int j = y; j > 0; j -= lowbit(j)) {
      res += c[i][j];
    }
  }
  return res;
}

int ask(int x1, int y1, int x2, int y2) {
  // 查询子矩阵和
  return sum(x2, y2) - sum(x2, y1 - 1) - sum(x1 - 1, y2) + sum(x1 - 1, y1 - 1);
}
```

子矩阵加，求子矩阵和

前置知识：前缀和 & 差分→5.8@p617 和本页面 区间加区间和→10@p1977 一节.

和一维树状数组的「区间加区间和」问题类似，考虑维护差分数组.

二维数组上的差分数组是这样的：

$$d(i,j)=a(i,j)-a(i-1,j)-a(i,j-1)+a(i-1,j-1).$$

为什么这么定义?

这是因为, 理想规定状态下, 在差分矩阵上做二维前缀和应该得到原矩阵, 因为这是一对逆运算.

二维前缀和的公式是这样的：

$s(i,j)=s(i-1,j)+s(i,j-1)-s(i-1,j-1)+a(i,j)$.

所以，设 a 是原数组，d 是差分数组，有：

$a(i,j)=a(i-1,j)+a(i,j-1)-a(i-1,j-1)+d(i,j)$

移项就得到二维差分的公式了.

$d(i,j)=a(i,j)-a(i-1,j)-a(i,j-1)+a(i-1,j-1)$.

这样以来，对左上角 (x_1,y_1)，右下角 (x_2,y_2) 的子矩阵区间加 v，相当于在差分数组上，对 $d(x_1,y_1)$ 和 $d(x_2+1,y_2+1)$ 分别单点加 v，对 $d(x_2+1,y_1)$ 和 $d(x_1,y_2+1)$ 分别单点加 $-v$.

至于原因，把这四个 d 分别用定义式表示出来，分析一下每项的变化即可.

举个例子吧，初始差分数组为 0，给 $a(2,2)...a(3,4)$ 子矩阵加 v 后差分数组会变为:

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & v & 0 & 0 & -v \\ 0 & 0 & 0 & 0 & 0 \\ 0 & -v & 0 & 0 & v \end{pmatrix}$$

（其中 $a(2,2)...a(3,4)$ 这个子矩阵恰好是上面位于中心的 2×3 大小的矩阵.）

因此，子矩阵加的做法是：转化为差分数组上的四个单点加操作.

现在考虑查询子矩阵和:

对于点 (x,y)，它的二维前缀和可以表示为:

$$\sum_{i=1}^{x}\sum_{j=1}^{y}\sum_{h=1}^{i}\sum_{k=1}^{j}d(h,k)$$

原因就是差分的前缀和的前缀和就是原本的前缀和.

和一维树状数组的「区间加区间和」问题类似, 统计 $d(h,k)$ 的出现次数, 为 $(x-h+1)\times(y-k+1)$.

然后接着推导:

$$\sum_{i=1}^{x}\sum_{j=1}^{y}\sum_{h=1}^{i}\sum_{k=1}^{j}d(h,k)$$

$$=\sum_{i=1}^{x}\sum_{j=1}^{y}d(i,j)\times(x-i+1)\times(y-j+1)$$

$$=\sum_{i=1}^{x}\sum_{j=1}^{y}d(i,j)\times(xy+x+y+1)-d(i,j)\times i\times(y+1)-d(i,j)\times j\times(x+1)+d(i,j)\times i\times j$$

所以我们需维护四个树状数组，分别维护 $d(i,j)$，$d(i,j)\times i$，$d(i,j)\times j$，$d(i,j)\times i\times j$ 的和信息.

当然了，和一维同理，如果只需要子矩阵加求单点值，维护一个差分数组然后询问前缀和就足够了.

下面给出代码:

实现

```
using ll = long long;
ll t1[N][N], t2[N][N], t3[N][N], t4[N][N];

void add(ll x, ll y, ll z) {
  for (int X = x; X <= n; X += lowbit(X))
    for (int Y = y; Y <= m; Y += lowbit(Y)) {
      t1[X][Y] += z;
      t2[X][Y] += z * x;  // 注意是 z * x 而不是 z * X，后面同理
      t3[X][Y] += z * y;
```

```
      t4[X][Y] += z * x * y;
    }
}

void range_add(ll xa, ll ya, ll xb, ll yb,
               ll z) {  //(xa, ya) 到 (xb, yb) 子矩阵
  add(xa, ya, z);
  add(xa, yb + 1, -z);
  add(xb + 1, ya, -z);
  add(xb + 1, yb + 1, z);
}

ll ask(ll x, ll y) {
  ll res = 0;
  for (int i = x; i; i -= lowbit(i))
    for (int j = y; j; j -= lowbit(j))
      res += (x + 1) * (y + 1) * t1[i][j] - (y + 1) * t2[i][j] -
             (x + 1) * t3[i][j] + t4[i][j];
  return res;
}

ll range_ask(ll xa, ll ya, ll xb, ll yb) {
  return ask(xb, yb) - ask(xb, ya - 1) - ask(xa - 1, yb) + ask(xa - 1, ya
- 1);
}
```

权值树状数组及应用

我们知道，普通树状数组直接在原序列的基础上构建，c_6 表示的就是 $a[5...6]$ 的区间信息.

然而事实上，我们还可以在原序列的权值数组上构建树状数组，这就是权值树状数组.

什么是权值数组?

一个序列 a 的权值数组 b，满足 $b[x]$ 的值为 x 在 a 中的出现次数.

例如：$a=(1,3,4,3,4)$ 的权值数组为 $b=(1,0,2,2)$.

很明显，b 的大小和 a 的值域有关.

若原数列值域过大，且重要的不是具体值而是值与值之间的相对大小关系，常 离散化→13.2@p3170 原数组后再建立权值数组.

另外，权值数组是原数组无序性的一种表示：它重点描述数组的元素内容，忽略了数组的顺序，若两数组只是顺序不同，所含内容一致，则它们的权值数组相同.

因此，对于给定数组的顺序不影响答案的问题，在权值数组的基础上思考一般更直观，比如 [NOIP2021] 数列[2].

运用权值树状数组，我们可以解决一些经典问题.

单点修改，查询全局第 k 小

在此处只讨论第 k 小，第 k 大问题可以通过简单计算转化为第 k 小问题.

该问题可离散化，如果原序列 a 值域过大，离散化后再建立权值数组 b. 注意，还要把单点修改中的涉及到的值也一起离散化，不能只离散化原数组 a 中的元素.

对于单点修改, 只需将对原数列的单点修改转化为对权值数组的单点修改即可. 具体来说, 原数组 $a[x]$ 从 y 修改为 z，转化为对权值数组 b 的单点修改就是 $b[y]$ 单点减 1，$b[z]$ 单点加 1.

对于查询第 k 小，考虑二分 x，查询权值数组中 $[1,x]$ 的前缀和，找到 x_0 使得 $[1,x_0]$ 的前缀和 $< k$ 而 $[1,x_0+1]$ 的前缀和 $\geq k$，则第 k 大的数是 x_0+1（注：这里认为 $[1,0]$ 的前缀和是 0）.

这样做时间复杂度是 $\Theta(\log^2 n)$ 的.

考虑用倍增替代二分.

设 $x=0$，$\text{sum}=0$，枚举 i 从 $\log_2 n$ 降为 0:

- 查询权值数组中 $[x+1...x+2^i]$ 的区间和 t.
- 如果 $\text{sum}+t<k$，扩展成功，$x\leftarrow x+2^i$，$\text{sum}\leftarrow \text{sum}+t$；否则扩展失败，不操作.

这样得到的 x 是满足 $[1...x]$ 前缀和 $< k$ 的最大值，所以最终 $x+1$ 就是答案.

看起来这种方法时间效率没有任何改善，但事实上，查询 $[x+1...x+2^i]$ 的区间和只需访问 $c[x+2^i]$ 的值即可.

原因很简单，考虑 $\text{lowbit}(x+2^i)$，它一定是 2^i，因为 x 之前只累加过 2^j 满足 $j>i$. 因此 $c[x+2^i]$ 表示的区间就是 $[x+1...x+2^i]$.

如此一来，时间复杂度降低为 $\Theta(\log n)$.

实现

- C++

```cpp
// 权值树状数组查询第 k 小
int kth(int k) {
  int sum = 0, x = 0;
  for (int i = log2(n); ~i; --i) {
    x += 1 << i;                    // 尝试扩展
    if (x > n || sum + t[x] >= k)  // 如果扩展失败
      x -= 1 << i;
    else
      sum += t[x];
  }
  return x + 1;  // 找不到就返回 n + 1
}
```

- Python

```python
# 权值树状数组查询第 k 小
def kth(k):
    sum = 0
    x = 0
    i = int(log2(n))
    while ~i:
        x = x + (1 << i)  # 尝试扩展
        if x > n or sum + t[x] >= k:  # 如果扩展失败
            x = x - (1 << i)
        else:
            sum = sum + t[x]
        i = i - 1
    return x + 1  # 找不到就返回 n + 1
```

全局逆序对（全局二维偏序）

相关阅读和参考实现：逆序对→9.8@p1138

全局逆序对也可以用权值树状数组巧妙解决. 问题是这样的: 给定长度为 n 的序列 a, 求 a 中满足 $i < j$ 且 $a[i] > a[j]$ 的数对 (i, j) 的数量.

该问题可离散化，如果原序列 a 值域过大，离散化后再建立权值数组 b.

我们考虑从 n 到 1 倒序枚举 i，作为逆序对中第一个元素的索引，然后计算有多少个 $j > i$ 满足 $a[j] < a[i]$，最后累计答案即可.

事实上，我们只需要这样做（设当前 $a[i] = x$）:

- 查询 $b[1...x-1]$ 的前缀和，即为左端点为 $a[i]$ 的逆序对数量.
- $b[x]$ 自增 1;

原因十分自然：出现在 $b[1...x-1]$ 中的元素一定比当前的 $x = a[i]$ 小，而 i 的倒序枚举，自然使得这些已在权值数组中的元素，在原数组上的索引 j 大于当前遍历到的索引 i.

用例子说明，$a = (4, 3, 1, 2, 1)$.

i 按照 $5 \to 1$ 扫:

- $a[5] = 1$，查询 $b[1...0]$ 前缀和，为 0，$b[1]$ 自增 1，$b = (1, 0, 0, 0)$.
- $a[4] = 2$，查询 $b[1...1]$ 前缀和，为 1，$b[2]$ 自增 1，$b = (1, 1, 0, 0)$.
- $a[3] = 1$，查询 $b[1...0]$ 前缀和，为 0，$b[1]$ 自增 1，$b = (2, 1, 0, 0)$.
- $a[2] = 3$，查询 $b[1...2]$ 前缀和，为 3，$b[3]$ 自增 1，$b = (2, 1, 1, 0)$.
- $a[1] = 4$，查询 $b[1...3]$ 前缀和，为 4，$b[4]$ 自增 1，$b = (2, 1, 1, 1)$.

所以最终答案为 $0 + 1 + 0 + 3 + 4 = 8$.

注意到, 遍历 i 后的查询 $b[1...x-1]$ 和自增 $b[x]$ 的两个步骤可以颠倒, 变成先自增 $b[x]$ 再查询 $b[1...x-1]$，不影响答案. 两个角度来解释:

- 对 $b[x]$ 的修改不影响对 $b[1...x-1]$ 的查询.
- 颠倒后，实质是在查询 $i \le j$ 且 $a[i] > a[j]$ 的数对数量，而 $i = j$ 时不存在 $a[i] > a[j]$，所以 $i \le j$ 相当于 $i < j$，所以这与原来的逆序对问题是等价的.

如果查询非严格逆序对（$i < j$ 且 $a[i] \ge a[j]$）的数量，那就要改为查询 $b[1...x]$ 的和，这时就不能颠倒两步了，还是两个角度来解释:

- 对 $b[x]$ 的修改 **影响** 对 $b[1...x]$ 的查询.
- 颠倒后，实质是在查询 $i \le j$ 且 $a[i] \ge a[j]$ 的数对数量，而 $i = j$ 时恒有 $a[i] \ge a[j]$，所以 $i \le j$ **不相当于** $i < j$，与原问题 **不等价**.

如果查询 $i \le j$ 且 $a[i] \ge a[j]$ 的数对数量，那这两步就需要颠倒了.

另外，对于原逆序对问题，还有一种做法是正着枚举 j，查询有多少 $i < j$ 满足 $a[i] > a[j]$. 做法如下（设 $x = a[j]$）:

- 查询 $b[x+1...V]$（V 是 b 的大小，即 a 的值域（或离散化后的值域））的区间和.
- 将 $b[x]$ 自增 1.

原因：出现在 $b[x+1...V]$ 中的元素一定比当前的 $x = a[j]$ 大，而 j 的正序枚举，自然使得这些已在权值数组中的元素，在原数组上的索引 i 小于当前遍历到的索引 j.

此外，逆序对的计数还可以通过 归并排序→5.7.8@p592 解决. 这一方法可以避免离散化. 时间复杂度同样为 $O(n \log n)$. 两种算法的参考实现都在 逆序对→9.8@p1138 章节.

树状数组维护不可差分信息

比如维护区间最值等.

注意，这种方法虽然码量小，但单点修改和区间查询的时间复杂度均为 $\Theta(\log^2 n)$，比使用线段树的时间复杂度 $\Theta(\log n)$ 劣.

区间查询

我们还是基于之前的思路，从 r 沿着 lowbit 一直向前跳，但是我们不能跳到 l 的左边.

因此，如果我们跳到了 $c[x]$，先判断下一次要跳到的 $x-\text{lowbit}(x)$ 是否小于 l:

- 如果小于 l，我们直接把 $\boldsymbol{a[x]}$ **单点** 合并到总信息里，然后跳到 $c[x-1]$.
- 如果大于等于 l，说明没越界，正常合并 $c[x]$，然后跳到 $c[x-\text{lowbit}(x)]$ 即可.

下面以查询区间最大值为例，给出代码:

实现

```
int getmax(int l, int r) {
  int ans = 0;
  while (r >= l) {
    ans = max(ans, a[r]);
    --r;
    for (; r - lowbit(r) >= l; r -= lowbit(r)) {
      // 注意，循环条件不要写成 r - lowbit(r) + 1 >= l
      // 否则 l = 1 时，r 跳到 0 会死循环
      ans = max(ans, C[r]);
    }
  }
  return ans;
}
```

可以证明，上述算法的时间复杂度是 $\Theta(\log^2 n)$.

时间复杂度证明

考虑 r 和 l 不同的最高位，一定有 r 在这一位上为 1，l 在这一位上为 0（因为 $r \geq l$）.

如果 r 在这一位的后面仍然有 1，一定有 $r-\text{lowbit}(r) \geq l$，所以下一步一定是把 r 的最低位 1 填为 0;

如果 r 的这一位 1 就是 r 的最低位 1，无论是 $r \leftarrow r-\text{lowbit}(r)$ 还是 $r \leftarrow r-1$，r 的这一位 1 一定会变为 0.

因此，r 经过至多 $\log n$ 次变换后，r 和 l 不同的最高位一定可以下降一位. 所以，总时间复杂度是 $\Theta(\log^2 n)$.

单点更新

注

请先理解树状数组树形态的以下两条性质，再学习本节.

- 设 $u=s\times 2^{k+1}+2^k$，则其儿子数量为 $k=\log_2 \text{lowbit}(u)$，编号分别为 $u-2^t(0\leq t<k)$.
- u 的所有儿子对应 c 的管辖区间恰好拼接成 $[l(u),u-1]$.

关于这两条性质的含义及证明，都可以在本页面的 树状数组与其树形态的性质→10@p1977 一节找到.

更新 $a[x]$ 后，我们只需要更新满足在树状数组树形态上，满足 y 是 x 的祖先的 $c[y]$.

对于最值（以最大值为例），一种常见的错误想法是，如果 $a[x]$ 修改成 p，则将所有 $c[y]$ 更新为 $\max(c[y], p)$．下面是一个反例：$(1, 2, 3, 4, 5)$ 中将 5 修改成 4，最大值是 4，但按照上面的修改这样会得到 5．将 $c[y]$ 直接修改为 p 也是错误的，一个反例是，将上面例子中的 3 修改为 4．

事实上，对于不可差分信息，不存在通过 p 直接修改 $c[y]$ 的方式．这是因为修改本身就相当于是把旧数从原区间「移除」，然后加入一个新数．「移除」时对区间信息的影响，相当于做「逆运算」，而不可差分信息不存在「逆运算」，所以无法直接修改 $c[y]$．

换句话说，对每个受影响的 $c[y]$，这个区间的信息我们必定要重构了．

考虑 $c[y]$ 的儿子们，它们的信息一定是正确的（因为我们先更新儿子再更新父亲），而这些儿子又恰好组成了 $[l(y), y-1]$ 这一段管辖区间，那再合并一个单点 $a[y]$ 就可以合并出 $[l(y), y]$，也就是 $c[y]$ 了．这样，我们能用至多 $\log n$ 个区间重构合并出每个需要修改的 c．

实现

```
void update(int x, int v) {
  a[x] = v;
  for (int i = x; i <= n; i += lowbit(i)) {
    // 枚举受影响的区间
    C[i] = a[i];
    for (int j = 1; j < lowbit(i); j *= 2) {
      C[i] = max(C[i], C[i - j]);
    }
  }
}
```

容易看出上述算法时间复杂度为 $\Theta(\log^2 n)$．

建树

可以考虑拆成 n 个单点修改，$\Theta(n \log^2 n)$ 建树．

也有 $\Theta(n)$ 的建树方法，见本页面 Θ(n) 建树→10@p1977 一节的方法一．

Tricks

$\Theta(n)$ 建树

以维护区间和为例．

方法一：

每一个节点的值是由所有与自己直接相连的儿子的值求和得到的．因此可以倒着考虑贡献，即每次确定完儿子的值后，用自己的值更新自己的直接父亲．

实现

- C++

```
// Θ(n) 建树
void init() {
  for (int i = 1; i <= n; ++i) {
    t[i] += a[i];
    int j = i + lowbit(i);
    if (j <= n) t[j] += t[i];
  }
```

```
}
```

- Python

```python
# O(n) 建树
def init():
    for i in range(1, n + 1):
        t[i] = t[i] + a[i]
        j = i + lowbit(i)
        if j <= n:
            t[j] = t[j] + t[i]
```

方法二：

前面讲到 $c[i]$ 表示的区间是 $[i-\text{lowbit}(i)+1, i]$，那么我们可以先预处理一个 sum 前缀和数组，再计算 c 数组.

实现

- C++

```cpp
// O(n) 建树
void init() {
  for (int i = 1; i <= n; ++i) {
    t[i] = sum[i] - sum[i - lowbit(i)];
  }
}
```

- Python

```python
# O(n) 建树
def init():
    for i in range(1, n + 1):
        t[i] = sum[i] - sum[i - lowbit(i)]
```

时间戳优化

对付多组数据很常见的技巧. 若每次输入新数据都暴力清空树状数组，就可能会造成超时. 因此使用 tag 标记，存储当前节点上次使用时间（即最近一次是被第几组数据使用）. 每次操作时判断这个位置 tag 中的时间和当前时间是否相同，就可以判断这个位置应该是 0 还是数组内的值.

实现

- C++

```cpp
// 时间戳优化
int tag[MAXN], t[MAXN], Tag;

void reset() { ++Tag; }

void add(int k, int v) {
  while (k <= n) {
    if (tag[k] != Tag) t[k] = 0;
    t[k] += v, tag[k] = Tag;
```

```
    k += lowbit(k);
  }
}

int getsum(int k) {
  int ret = 0;
  while (k) {
    if (tag[k] == Tag) ret += t[k];
    k -= lowbit(k);
  }
  return ret;
}
```

- Python

```
# 时间戳优化
tag = [0] * MAXN
t = [0] * MAXN
Tag = 0


def reset():
    Tag = Tag + 1


def add(k, v):
    while k <= n:
        if tag[k] != Tag:
            t[k] = 0
        t[k] = t[k] + v
        tag[k] = Tag
        k = k + lowbit(k)


def getsum(k):
    ret = 0
    while k:
        if tag[k] == Tag:
            ret = ret + t[k]
        k = k - lowbit(k)
    return ret
```

例题

- 树状数组 1：单点修改，区间查询[3]
- 树状数组 2：区间修改，单点查询[4]
- 树状数组 3：区间修改，区间查询[5]
- 二维树状数组 1：单点修改，区间查询[6]
- 二维树状数组 2：区间修改，单点查询[7]
- 二维树状数组 3：区间修改，区间查询[8]

外部参考资料

[1] Efficient Range Minimum Queries using Binary Indexed Trees

[2] [NOIP2021] 数列

[3] 树状数组 1：单点修改，区间查询

[4] 树状数组 2：区间修改，单点查询

[5] 树状数组 3：区间修改，区间查询

[6] 二维树状数组 1：单点修改，区间查询

[7] 二维树状数组 2：区间修改，单点查询

[8] 二维树状数组 3：区间修改，区间查询

10.13 线段树

10.13.1 线段树基础

> **Authors:** Ir1d, hsfzLZH1, Tiphereth-A, ksyx, Marcythm, StudyingFather, Xeonacid, ChungZH, Early0v0, chenryang, Enter-tainer, CCXXXI, cjsoft, HeRaNO, konnyakuxzy, NachtgeistW, orzAtalod, wy-luke, amlhdsan, billchenchina, c-forrest, Chrogeek, countercurrent-time, ethan-enhe, GavinZhengOI, H-J-Granger, iamtwz, luoguojie, AKindMouse, AngelKitty, chenzheAya, CJSoft, DawnMagnet, diauweb, ezoixx130, GekkaSaori, Gesrua, Haohu Shen, Henry-ZHR, hjsjhn, hly1204, jaxvanyang, Jebearssica, kenlig, Konano, kxccc, LovelyBuggies, lychees, Makkiy, megakite, Menci, mgt, minghu6, moon-dim, onelittlechildawa, ouuan, P-Y-Y, Peanut-Tang, PotassiumWings, SamZhangQingChuan, shadowice1984, shawlleyw, shuzhouliu, sshwy, SukkaW, Suyun514, weiyong1024, x2e6, Ycrpro, yifan0305, zeningc

引入

线段树是算法竞赛中常用的用来维护 **区间信息** 的数据结构.

线段树可以在 $O(\log N)$ 的时间复杂度内实现单点修改、单点查询、区间修改、区间查询等操作.

基本结构与操作

线段树是一棵二叉树. 它的每个结点都存储一段区间的信息:

- 叶结点处存储的是单个元素 x 处的信息，也可以看作是长度为 1 的区间 $[x, x]$ 上的信息.
- 以非叶结点为根的子树的叶结点构成一段连续的区间 $[l, r]$. 那么，该结点处就存储区间 $[l, r]$ 上的信息.

非叶结点总有两个子结点，它对应的区间就恰为这两个子结点对应区间的不交并. 设某个非叶结点存储区间 $[l, r]$ $(l < r)$ 上的信息，那么，它的两个子结点存储的就分别是 $[l, m]$ 和 $[m + 1, r]$ 上的信息，其中，$l \leq m < r$.

线段树的根结点处就存储着整段区间 $[L, R]$ 上的信息. 线段树的规模用该区间的长度 $N = R - L + 1$ 表示.

区间信息

线段树可以维护区间信息. 本节介绍区间信息需要满足的常见性质以及相应的实现方法.

一般地，**区间信息** 是以区间为自变量的函数 $\varphi : \mathcal{I} \to M$，其中，$\mathcal{I}$ 表示 $[L, R]$ 的所有子区间（包括空区间），M 表示区间信息可以取值的空间——本文称 **信息空间**. 线段树存储的区间信息 φ 需要满足如下性质:

1. 将一段区间划分成两段子区间，相应的信息也可以从子区间信息合并得到. 也就是说，如果设 I 是 I_1 和 I_2 的不交并，且 I_1 位于 I_2 左侧，那么 $\varphi(I) = \varphi(I_1) \circ \varphi(I_2)$，其中，运算 $\circ$ 表示 **信息合并** 操作.
2. 将一段区间划分为多段子区间的不交并，这些子区间信息合并的结果与划分和结合的方式都无关，都等于最开始一段区间的信息. 也就是说，运算 $\circ$ 满足结合律.

3. 空区间 $\emptyset$ 也有良定义的信息 $e = \varphi(\emptyset)$. 而且，因为任何区间 I 都可以看作是它自身与空区间 $\emptyset$ 的不交并，所以任何区间信息与空区间信息的合并都保持不变. 也就是说，$\varphi(I) \circ e = e \circ \varphi(I) = \varphi(I)$. 这说明 e 是运算 $\circ$ 的单位元.

这些性质保证，要查询区间 I 对应的信息时，只要能够找到一系列结点，使得它们对应的区间构成 I 的划分，就能从这些结点存储的信息中合并出区间 I 的信息.

区间信息可以合并，合并操作满足结合律，且存在单位元. 这些性质意味着信息空间 M 在信息合并操作 $\circ$ 下构成一个 幺半群→9.16.1@p1805. [1]

约定

在给出线段树操作的参考实现时，本文假设信息空间 M 的元素存储在 `Info` 结构体中，它的默认构造得到单位元，且它重载了加法运算符用于实现信息合并操作.

例子

满足幺半群性质的区间信息有很多. 区间长度、区间和、区间乘积、区间最大值都是一些简单的例子. 它们对应的幺半群分别是 $(\mathbf{N}, +)$, $(\mathbf{R}, +)$, $(\mathbf{R}, \times)$, $(\mathbf{R} \cup \{-\infty\}, \max)$, 对应的单位元则分别是 $0, 0, 1, -\infty$. 这些例子可以拓展到其他常见的符合幺半群性质的操作，例如按位取异或、矩阵乘法、函数复合等.

还有一些更为复杂的例子. 例如，区间最大（非空）子段和同样可以通过幺半群来维护. 当然，如果只维护最大子段和这一个变量，是无法进行合并操作的. 对于划分 $I = I_1 \cup I_2$，区间 I 的最大子段和可能在区间 I_1 的子区间处取得，可能在区间 I_2 的子区间处取得，也可能在某个横跨 I_1 和 I_2 的子区间处取得. 为了计算最后一种情况对应的最大值，需要分别记录区间和、区间最大（非空）前缀和、区间最大（非空）后缀和.

- 区间和

```cpp
// Sum.
struct Info {
  long long sum;

  Info(long long x = 0) : sum(x) {}

  Info operator+(const Info& o) const { return Info(sum + o.sum); }

  Info& operator+=(const Info& o) { return *this = *this + o; }
};

```

- 区间最大值（值域非负）

```cpp
// Range max; -1 marks the empty interval.
struct Info {
  int ma;

  Info(int _ma = -1) : ma(_ma) {}

```

[1] 严格地说，线段树维护的信息只需构成半群，不必要求单位元存在. 递归实现中用到单位元，通常是因为区间查询以它作为累计信息的初值；改为按相交情况分类讨论即可避开. 不过，非递归实现中的多余叶结点和动态开点中的空结点都绕不开单位元. 退一步说，即使确实需要，也总可以向半群 S 中添加一个元素 e，规定它与任意元素运算的结果都是该元素本身，从而得到幺半群 $S \cup \{e\}$. 本文为表述方便，总是假定维护的是幺半群信息.

```
  Info operator+(const Info& o) const { return Info(std::max(ma,
o.ma)); }

  Info& operator+=(const Info& o) { return *this = *this + o; }
};

```

- 一次函数

```
// Affine function: f(x) = ax + b.
// Composition: (f1 + f2)(x) = f2(f1(x)).
// Identity: e(x) = x.
struct Info {
  int a, b;

  Info(int _a = 1, int _b = 0) : a(_a), b(_b) {}

  Info operator+(const Info& o) const {
    return Info((long long)o.a * a % M, ((long long)o.a * b + o.b)
% M);
  }

  Info& operator+=(const Info& o) { return *this = *this + o; }
};

```

- 区间最大非空子段和

```
// Maximum (non-empty) subarray sum
struct Info {
  int sum;  // sum of the whole interval
  int lma;  // maximum sum of a non-empty prefix
  int rma;  // maximum sum of a non-empty suffix
  int ma;   // maximum sum of a non-empty subarray

  Info() : sum(0), lma(-1e9), rma(-1e9), ma(-1e9) {}

  Info(int x) : sum(x), lma(x), rma(x), ma(x) {}

  Info(int _s, int _l, int _r, int _m) : sum(_s), lma(_l), rma(_r),
ma(_m) {}

  Info operator+(const Info& o) const {
    return Info(sum + o.sum, std::max(lma, sum + o.lma),
                std::max(rma + o.sum, o.rma),
                std::max(std::max(ma, o.ma), rma + o.lma));
  }

  Info& operator+=(const Info& o) { return *this = *this + o; }
};

```

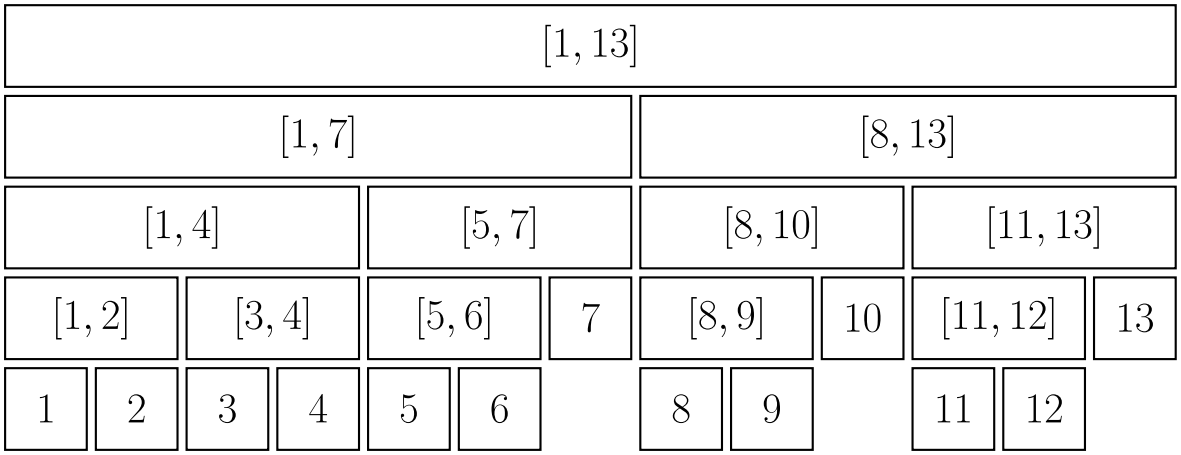
需要注意的是，区间信息的合并未必满足交换律．当执行合并操作时，需要严格按照子区间的顺序从左向右进行．

递归建树与存储方式

为了尽可能地减小树高，需要将当前结点的区间 $[l,r]$ 尽可能均匀地分成两段．为此，常选取

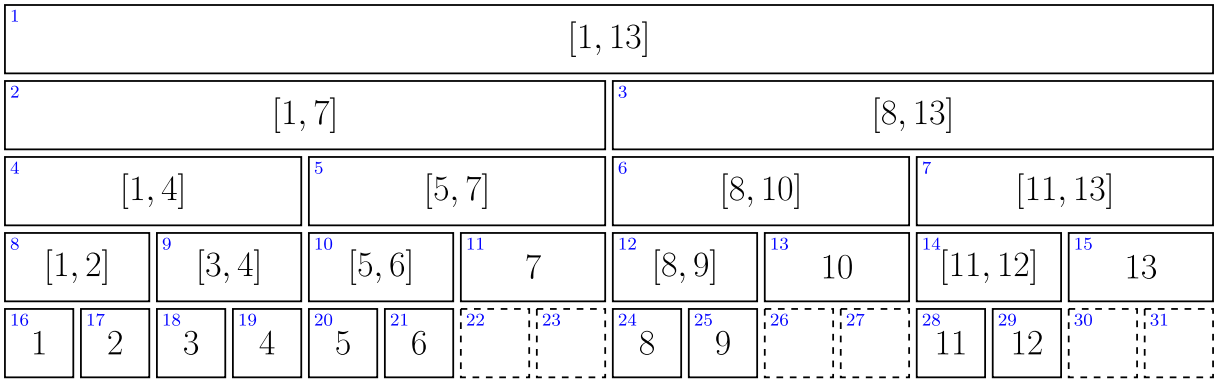
$$m=\left\lfloor\frac{l+r}{2}\right\rfloor.$$

这样，如果当前结点区间长度为 $n>1$，它的两个子结点 $[l,m]$ 和 $[m+1,r]$ 就分别对应长度为 $\lceil n/2\rceil$ 和 $\lfloor n/2\rfloor$ 的区间．由此得到的线段树高度是 $\lceil\log_2 N\rceil$．这就保证了线段树是平衡二叉树，单点操作的复杂度总是 $O(\log N)$ 的．



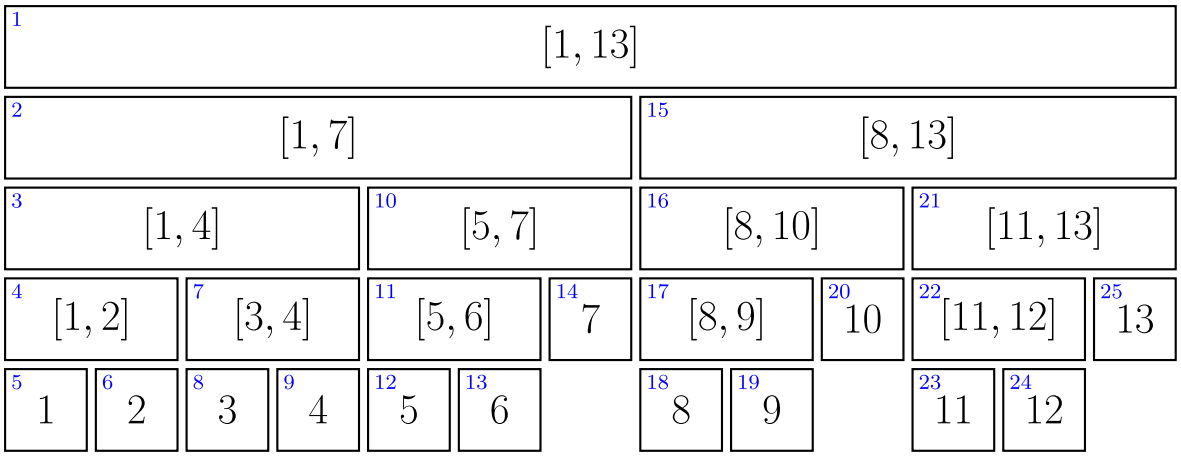
存储长度为 N 的区间上的信息，需要建立有 N 个叶结点的线段树．作为完整二叉树（即每个结点都有 0 或 2 个子结点），线段树恰有 $2N-1$ 个存储信息的结点．也就是说，这样构造的线段树空间复杂度是 $\Theta(N)$ 的；递归建树时每个结点恰好访问一次，时间复杂度也是 $\Theta(N)$ 的．

虽然线段树的结构相对固定，但是它的存储方式却并不唯一．



第一种常见的方式是堆式存储（如上图所示），即将线段树嵌入完美二叉树中．此时，根结点编号为 1；如果当前结点编号是 i，那么它的左右子结点编号分别是 $2i$ 和 $2i+1$．因为树高为 $\lceil\log_2 N\rceil$，所以这样存储线段树需要开长度为 $2^{\lceil\log_2 N\rceil+1}$ 的数组．为计算简便，通常会直接开长度为 $4N$ 的数组[2]；或者将 N 补齐为 2 的幂次 N' 后，开长度为 $2N'$ 的数组．这样存储的长处是无需额外存储子结点编号，短处则是存在部分无用结点，空间利用效率不高．

[2]考察二叉树占用的空间 $2^{\lceil\log_2 N\rceil+1}$ 与线段树规模 N 的比值．前者只与二叉树的高度有关．当线段树高度为 k 时，线段树的最小规模为 $N=2^{k-1}+1$．相应地，二者比值为 $(4N-4)/N=4-4/N$，上确界为 4．这就是为什么存储规模为 N 的线段树时，需要为相应的完美二叉树开辟大小为 $4N$ 的数组．



第二种常见的方式是利用内存池，动态地指定结点编号．因为线段树形态固定，只要在建树的时候分配一次结点编号即可．这样做保证了没有无用结点，只需要开出长度为 $2N$ 的数组．此时，结点编号是线段树前序遍历的序号．但是，由于（右）子结点的编号无法仅由当前结点编号计算[3]，通常会额外分配总长度为 $4N$ 的数组存储左右子结点编号．如果设存储单个编号需要的空间为 1，存储单个结点需要的空间为 K，那么，只要 $2NK + 4N \leq 4NK$，即 $K \geq 2$ 时，动态分配的空间占用就不会劣于堆式存储．

这两种存储和建立线段树的方式实现如下：

参考实现

- 堆式存储

```cpp
// Recursive structure, stored in heap.
#define lc(x) ((x) << 1)
#define rc(x) (((x) << 1) | 1)

int rt, L, R;
std::vector<Info> val;

void push_up(int cr) { val[cr] = val[lc(cr)] + val[rc(cr)]; }

// Build the tree based on info stored in vec (0-indexed).
void build(int cr, int ll, int rr, const std::vector<Info>& vec) {
  if (ll == rr) return (void)(val[cr] = vec[ll - L]);
  int mm = ll + ((rr - ll) >> 1);
  build(lc(cr), ll, mm, vec);
  build(rc(cr), mm + 1, rr, vec);
  push_up(cr);
}

void build(int l, int r, const std::vector<Info>& vec) {
  rt = 1, L = l, R = r;
  val.resize((R - L + 1) << 2);
  build(rt, L, R, vec);
}

```

- 动态分配结点

[3]实际上，当如图所示按照前序遍历编号时，右子结点编号可以由当前结点编号和左子树大小计算得到，而左子树大小可以通过区间长度计算．由于递归访问线段树结点时，通常都会维护区间左右端点，所以右子结点编号也是可以直接计算的．

```cpp
// Recursive structure, stored in a full binary tree.
#define lc(x) ch[(x)][0]
#define rc(x) ch[(x)][1]

int rt, id, L, R;
std::vector<std::array<int, 2>> ch;
std::vector<Info> val;
std::vector<Transform> lazy;

void push_up(int cr) { val[cr] = val[lc(cr)] + val[rc(cr)]; }

// Build the tree based on info stored in vec (0-indexed).
void build(int cr, int ll, int rr, const std::vector<Info>& vec) {
  if (ll == rr) return (void)(val[cr] = vec[ll - L]);
  int mm = ll + ((rr - ll) >> 1);
  build(lc(cr) = ++id, ll, mm, vec);
  build(rc(cr) = ++id, mm + 1, rr, vec);
  push_up(cr);
}

void build(int l, int r, const std::vector<Info>& vec) {
  rt = 1, id = 0, L = l, R = r;
  int n = R - L + 1;
  ch.resize(n << 1), val.resize(n << 1), lazy.resize(n << 1);
  build(rt = ++id, L, R, vec);
}

```

提示

1. 计算区间 $[l, r]$ 中点 $m = \lfloor (l + r)/2 \rfloor$ 时，使用 `m = l + (r - l) / 2` 而不是 `m = (l + r) / 2` 可以避免整型溢出和负数除法向零取整等问题.
2. 通常可以实现一个 `push_up` 函数用于将子结点信息合并到当前结点.

除了访问子结点的方法有所差异外，存储方式的选取不会影响建树后线段树操作的实现. 但是，堆式存储中结点编号由它在完美二叉树中的位置决定，拓展性较差，无法支持动态开点和可持久化等拓展.

单点修改与查询

线段树操作中，最简单的就是单点操作.

首先考虑单点查询操作. 线段树中, 叶结点存储的元素是有序排列的. 所以, 当递归到非叶结点时, 只需要比较要查询的元素和当前区间中点，就可以决定继续访问左子结点还是右子结点. 当找到叶结点时, 返回该叶结点处存储的信息即可. 这就完成了单点查询操作.

参考实现

```cpp
// Query info at x.
Info query(int cr, int ll, int rr, int x) {
  if (ll == rr) return val[cr];
  int mm = ll + ((rr - ll) >> 1);
  if (x <= mm)
```

```
    return query(lc(cr), ll, mm, x);
  else
    return query(rc(cr), mm + 1, rr, x);
}

Info query(int x) { return query(rt, L, R, x); }

```

单点修改操作类似. 同样是自上而下地找到要访问的叶结点. 然后, 对该叶结点进行修改操作. 最后, 还需要在回溯时, 相应地更新该叶结点的所有祖先结点的信息. 这就是单点修改操作. 由于所有对单点的修改操作都可以看作是将它的值进行替换, 所以这里仅提供了更改值的参考实现. 区间修改操作→10@p1977 一节讨论了更一般的修改操作.

参考实现

```
// Modify info at x to v.
void modify(int cr, int ll, int rr, int x, const Info& v) {
  if (ll == rr) return (void)(val[cr] = v);
  int mm = ll + ((rr - ll) >> 1);
  if (x <= mm)
    modify(lc(cr), ll, mm, x, v);
  else
    modify(rc(cr), mm + 1, rr, x, v);
  push_up(cr);
}

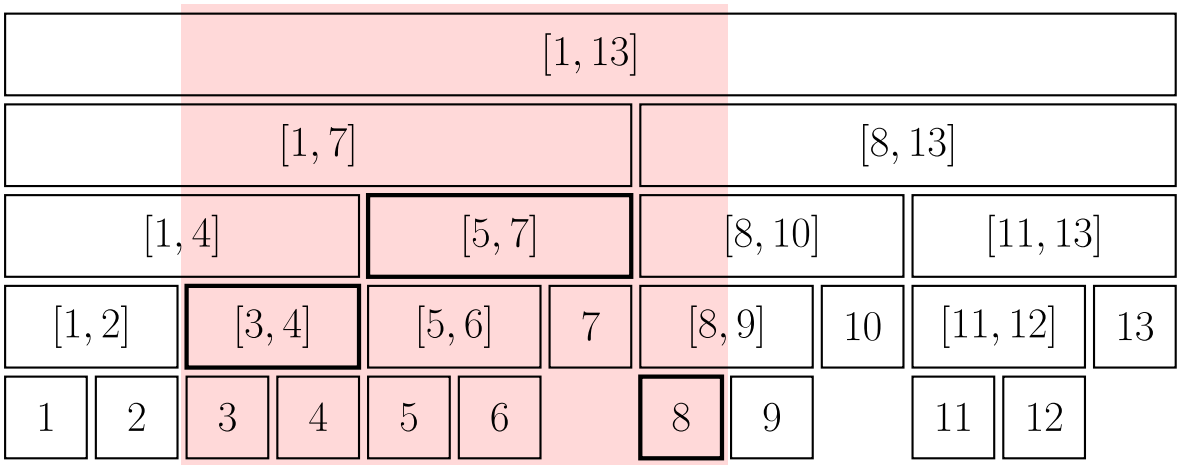
void modify(int x, const Info& v) { modify(rt, L, R, x, v); }

```

由于树高是 $\Theta(\log N)$ 的，线段树的单点操作都是 $O(\log N)$ 的.

区间查询

接下来，考虑区间查询操作.



前文已经说明, 只要将要查询的区间拆分成若干个线段树结点对应的区间的不交并, 然后将这些结点处的信息依次合并到一起即可. 当然, 拆分出的结点越少越好. 这就要求拆分出的结点对应的是包含在查询区间内的 **极大区间**. 上图中, 浅色区域标出操作区间, 粗边结点对应极大区间. 要找到这些极大区间对

应结点也很容易：从根结点开始，向下搜索所有对应区间与查询区间相交的结点；当搜索到一个完全包含于查询区间内的结点时，就找到了一个极大区间所对应的结点，因而无需继续搜索该结点的子孙结点.

参考实现

```
// Query info in [l, r].
Info query(int cr, int ll, int rr, int tl, int tr) {
  if (tl <= ll && rr <= tr) return val[cr];
  int mm = ll + ((rr - ll) >> 1);
  Info res;
  if (tl <= mm) res = query(lc(cr), ll, mm, tl, tr);
  if (mm < tr) res += query(rc(cr), mm + 1, rr, tl, tr);
  return res;
}

Info query(int l, int r) { return query(rt, L, R, l, r); }

```

提示

1. 除了上述实现方式外，还可以在向下搜索到与查询区间不交的区间结点时终止搜索.
2. 由于结果初始化为信息空间 M 中的单位元，左子结点返回结果可以直接覆盖，无需进行信息合并.

可以说明，在区间查询过程中访问的所有结点——所有极大区间结点及其祖先结点——的总数是 $O(\log N)$ 的. 因此，区间查询的复杂度也是 $O(\log N)$ 的.

证明

进行区间查询时，访问到的每个非根结点，其父结点对应区间必然包含查询区间左端点或右端点. 如果不然，父结点对应区间要么完全包含在查询区间内，要么与查询区间不交，都不应该继续访问其子结点（即当前结点）. 而且，相同深度的结点对应区间两两不交，包含左、右端点的各至多 1 个，而每个结点至多有 2 个子结点. 所以，区间查询时访问到的相同深度结点，至多只有 4 个. 线段树高度是 $\Theta(\log N)$ 的，区间查询所访问到的结点数目就也是 $O(\log N)$ 的.

区间修改与懒惰标记

最后，考虑区间修改操作.

与区间查询操作不同的是，区间修改操作的影响并不局限于极大区间结点及其祖先结点，还会波及其子孙结点. 这是因为区间修改操作会影响这些极大区间的子区间，也就是相应结点的子孙结点. 如果要修改区间波及的所有结点，复杂度显然过高，并不现实. 因此，实现区间修改操作时，利用了 **懒惰标记**（lazy tag）的思想. 具体地，每次区间修改操作仅实际修改极大区间结点及其祖先结点，而对极大区间结点子孙结点的修改则推迟到有必要的时候再进行. 为了记录后续需要进行的修改操作，还需要在极大区间结点处留下一个标记. 这个标记记录了对其子孙结点（不包括自身）需要进行但是尚未进行的操作. 它就是懒惰标记.

约定

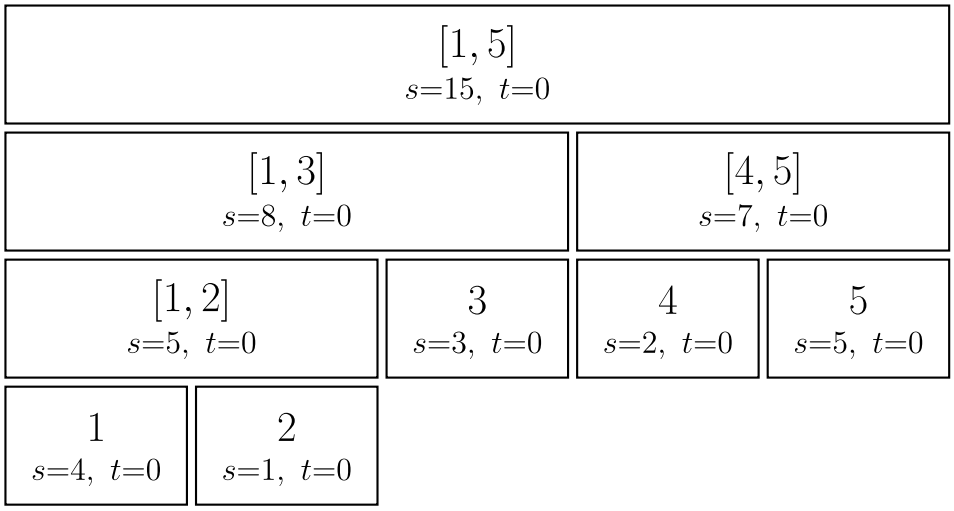
本文假定，带有懒惰标记的结点自身的修改已经完成. 这并非强制要求，可能会因实现而异.

在支持区间修改操作的线段树中，其他操作的实现也需要相应调整. 进行任何操作时，每当需要访问某结点的子结点时，都要检查当前结点是否存在尚未清空的懒惰标记. 如果存在，需要先下传该懒惰标记，

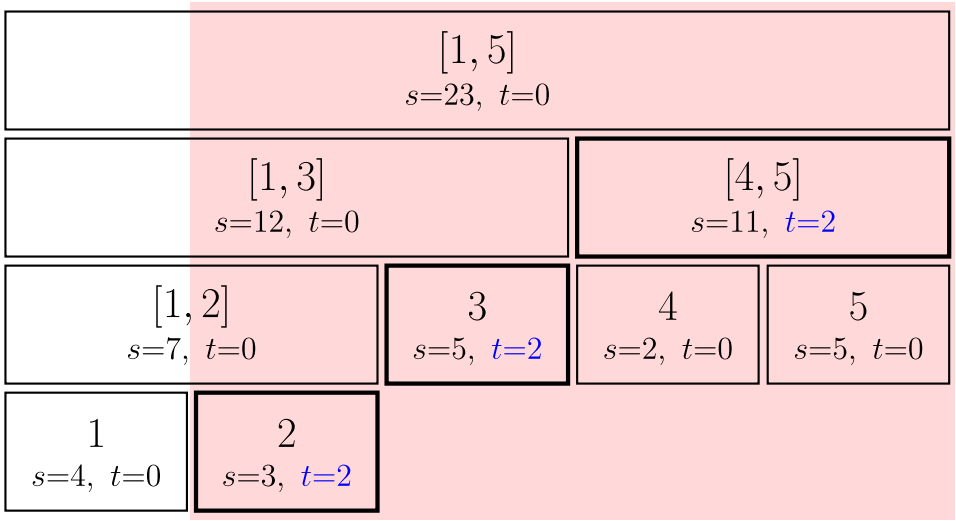
再继续访问子结点或进行其他操作. 所谓下传懒惰标记, 就是指对该结点的子结点进行相应修改操作, 同时给相应的子结点打上懒惰标记, 最后清空当前结点处的标记.

示例

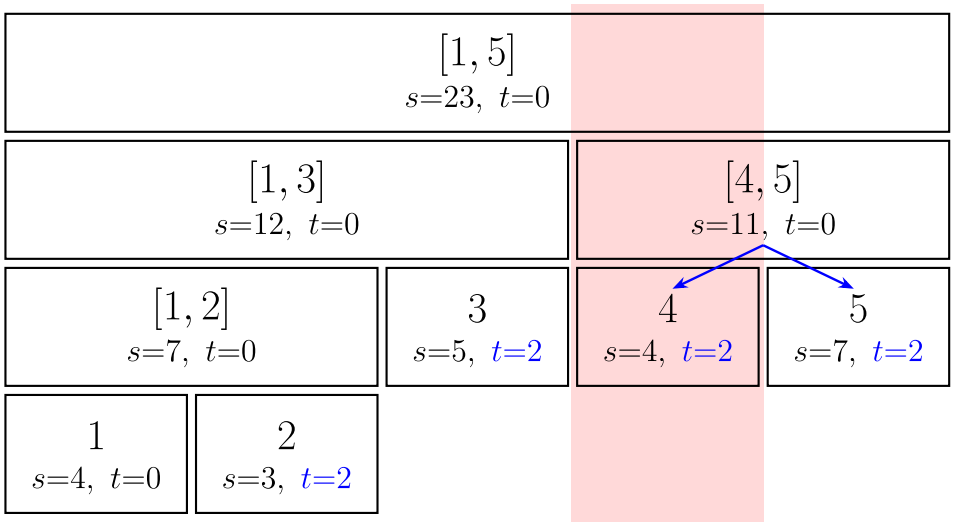
下图展示了由数组 $[4,1,3,2,5]$ 建立的线段树结构.



每个结点处, s 表示当前区间的元素和, t 是懒惰标记, 用于实现区间加操作. 初始时, 所有区间元素和都是正确的, 且懒惰标记均为空. 现在, 将区间 $[2,5]$ 中的所有元素都加 2, 所得结果如下图所示.



图中, 浅色区域标出操作区间, 粗边结点对应极大区间. 在这些极大区间结点处, 元素和 s 增加了区间长度的 2 倍, 正确反映了区间加操作带来的变化; 同时懒惰标记增加了 2, 表示其子孙结点尚未更新. 极大区间结点的祖先结点则从其子结点处获得了更新后的元素和. 此时若再查询区间 $[4,4]$ 的元素和, 所得结果如下图所示.



在访问区间 $[4,4]$ 的过程中，会先访问区间 $[4,5]$，并注意到它带有懒惰标记 $t=2$．在访问其子结点之前，需要先下传懒惰标记：对其子结点各执行一次区间加 2，子结点的懒惰标记也增加 2，表示其子孙结点（尽管图中并不存在）尚未更新，最后清空 $[4,5]$ 处的标记．下传之后，再继续访问子结点区间 $[4,4]$，就能读取到正确的元素和 4 了．

在执行区间修改操作或下传懒惰标记时，要标记的结点可能已经存在未清空的懒惰标记．此时，不能够直接覆盖标记，而需要将标记更新为两次操作复合的结果．由于操作的复合未必满足交换律，复合时需要注意顺序：新的操作总是复合在原有标记之后．下传标记时，总是将父结点的标记复合在子结点原有标记之后．这是因为访问子结点前必然先下传标记，子结点原有标记被打上时，父结点必定不带标记；父结点现有的标记只可能来自之后的操作．

为了正确记录区间修改操作，需要理解这类操作需要满足的性质．仍然设线段树记录的区间信息可以总结为函数 $\varphi:\mathcal{I}\to M$．进一步地，设修改操作是 $\pi:M\to M$，它将旧信息依某种规则变化为新信息．那么，区间修改操作 π 需要满足如下性质：

1. 它是良定义的．也就是说，区间修改的结果只与信息 $m\in M$ 有关，而与所处区间 $I\in\mathcal{I}$ 无关．对于区间修改结果涉及所处区间特征的情形，可以将区间特征（例如区间长度、左右端点等）视为区间信息的一部分．
2. 修改操作必须要与区间信息的合并相兼容．也就是说，如果某区间信息可以由子区间信息合并得到，那么对该区间进行修改操作得到的结果，仍然可以由对子区间信息进行修改操作得到的结果合并得到，亦即对于 $m_1,m_2\in M$，总是有 $\pi(m_1\circ m_2)=\pi(m_1)\circ\pi(m_2)$．
3. 特别地，规定空区间信息修改后仍然是空区间信息，即 $\pi(e)=e$．[4]

这说明，修改操作是幺半群 M 上的 自同态映射→9.16.2@p1812．而且，考虑这些修改操作形成的集合 Π，只要它对映射复合封闭，由于映射复合满足结合律且恒等映射是单位元，Π 也是一个幺半群．

约定

在给出线段树操作的参考实现时，本文假设修改操作存储在 `Transform` 结构体中，默认构造得到恒等操作，它重载了加法运算符用于实现操作的复合，且它重载了括号运算符用于修改区间信息．为了方便判断懒惰标记是否为空（即不修改），还重载了到布尔值的显式类型转换．

[4] 类似区间信息处的讨论，很多实现中单位元并不会作为修改操作的输入，此时就不必要求这条性质，只需要修改操作是半群上的自同态即可．但是，在后文标记永久化的非递归实现中，累计信息的初值就是单位元，且会逐层施加沿途的标记，这条性质不可忽略．本文为表述方便，总是假定修改操作是幺半群上的自同态．

例子

线段树能支持的修改操作很多，区间加、区间乘、区间赋值、区间仿射变换都是常见的例子．实现的难点往往不在操作本身，而在于如何把它作用到区间信息上，也就是如何让它满足前文的三条性质．例如，维护区间和而要支持区间加时，增量对区间和的贡献是所加的数乘以区间长度，而区间长度并不在区间信息之中．此时，按第一条性质的做法，把区间长度也记入区间信息即可．

- 区间最小值信息 & 区间加操作

```
// Minimum.
struct Info {
  long long mi;

  Info(long long _mi = LLONG_MAX) : mi(_mi) {}

  Info operator+(const Info& o) const { return Info(std::min(mi,
o.mi)); }

  Info& operator+=(const Info& o) { return *this = *this + o; }
};

// Addition.
struct Transform {
  long long x;

  Transform(long long _x = 0) : x(_x) {}

  explicit operator bool() const { return x != 0; }

  Transform operator+(const Transform& o) const { return Transform(x
+ o.x); }

  Transform& operator+=(const Transform& o) { return *this = *this +
o; }

  Info operator()(const Info& v) const {
    return Info(v.mi == LLONG_MAX ? LLONG_MAX : v.mi + x);
  }
};

```

- 区间和信息 & 区间加操作

```
// Interval length and sum of elements.
struct Info {
  long long len, sum;

  Info(long long _len = 0, long long _sum = 0) : len(_len), sum(_sum)
{}

  Info operator+(const Info& o) const { return Info(len + o.len, sum
+ o.sum); }

  Info& operator+=(const Info& o) { return *this = *this + o; }
```

```
};

// Range add.
struct Transform {
  long long v;

  Transform(long long _v = 0) : v(_v) {}

  explicit operator bool() const { return v != 0; }

  Transform operator+(const Transform& o) const { return Transform(v + o.v); }

  Transform& operator+=(const Transform& o) { return *this = *this + o; }

  Info operator()(const Info& x) const {
    return Info(x.len, x.sum + v * x.len);
  }
};
```

- 区间最大值信息 & 区间赋值操作

```
// Range max; -1 marks the empty interval.
struct Info {
  int ma;

  Info(int _ma = -1) : ma(_ma) {}

  Info operator+(const Info& o) const { return Info(std::max(ma, o.ma)); }

  Info& operator+=(const Info& o) { return *this = *this + o; }
};

// Assignment; -1 marks no assignment.
struct Transform {
  int x;

  Transform(int _x = -1) : x(_x) {}

  explicit operator bool() const { return x != -1; }

  Transform operator+(const Transform& o) const { return o ? o : *this; }

  Transform& operator+=(const Transform& o) { return *this = *this + o; }

  Info operator()(const Info& v) const {
    return *this && v.ma != -1 ? Info(x) : v;
  }
```

```
27 };
28
```

在实现好用于存储区间修改操作的结构后，懒惰更新和下传懒惰标记的操作就可以实现如下：

参考实现

```
// Lazy update.
void lazy_update(int cr, const Transform& f) {
  val[cr] = f(val[cr]);
  lazy[cr] += f;
}

// Push down lazy tag.
void push_down(int cr) {
  if (!lazy[cr]) return;
  lazy_update(lc(cr), lazy[cr]);
  lazy_update(rc(cr), lazy[cr]);
  lazy[cr] = Transform();
}

```

提示

1. 不要将懒惰标记打到空结点上.
2. 如果实现得仔细，也可以不将懒惰标记打到叶结点上.

单次下传操作的复杂度通常是 $O(1)$ 的.

进而，区间修改操作就可以实现如下：

参考实现

```
// Apply transformation f to the range [l, r].
void modify(int cr, int ll, int rr, int tl, int tr, const Transform& f) {
  if (tl <= ll && rr <= tr) return lazy_update(cr, f);
  push_down(cr);
  int mm = ll + ((rr - ll) >> 1);
  if (tl <= mm) modify(lc(cr), ll, mm, tl, tr, f);
  if (mm < tr) modify(rc(cr), mm + 1, rr, tl, tr, f);
  push_up(cr);
}

void modify(int l, int r, const Transform& f) { modify(rt, L, R, l, r, f); }

```

和区间查询操作一样，带懒惰标记的线段树区间修改操作同样是 $O(\log N)$ 的. 带懒惰标记的线段树的其他操作和不带懒惰标记的情形实现基本相同，只需要在访问子结点之前添加下传标记操作即可. 由于单次操作至多下传 $O(\log N)$ 次标记，所以这些操作时间复杂度仍然是 $O(\log N)$，只是常数变大了. 如果不涉及区间修改操作，通常选择实现不带懒惰标记的线段树，降低算法常数.

参考实现

下面给出不带懒惰标记和带懒惰标记的线段树的完整参考实现.

参考实现

- 不带懒惰标记

```cpp
// Segment Tree Implementation. Recursive. No lazy tag.
// Recursive structure, stored in heap.
#define lc(x) ((x) << 1)
#define rc(x) (((x) << 1) | 1)

int rt, L, R;
std::vector<Info> val;

void push_up(int cr) { val[cr] = val[lc(cr)] + val[rc(cr)]; }

// Build the tree based on info stored in vec (0-indexed).
void build(int cr, int ll, int rr, const std::vector<Info>& vec) {
  if (ll == rr) return (void)(val[cr] = vec[ll - L]);
  int mm = ll + ((rr - ll) >> 1);
  build(lc(cr), ll, mm, vec);
  build(rc(cr), mm + 1, rr, vec);
  push_up(cr);
}

void build(int l, int r, const std::vector<Info>& vec) {
  rt = 1, L = l, R = r;
  val.resize((R - L + 1) << 2);
  build(rt, L, R, vec);
}

// Query info at x.
Info query(int cr, int ll, int rr, int x) {
  if (ll == rr) return val[cr];
  int mm = ll + ((rr - ll) >> 1);
  if (x <= mm)
    return query(lc(cr), ll, mm, x);
  else
    return query(rc(cr), mm + 1, rr, x);
}

Info query(int x) { return query(rt, L, R, x); }

// Modify info at x to v.
void modify(int cr, int ll, int rr, int x, const Info& v) {
  if (ll == rr) return (void)(val[cr] = v);
  int mm = ll + ((rr - ll) >> 1);
  if (x <= mm)
    modify(lc(cr), ll, mm, x, v);
  else
    modify(rc(cr), mm + 1, rr, x, v);
  push_up(cr);
}
```

```
48
49 void modify(int x, const Info& v) { modify(rt, L, R, x, v); }
50
51 // Query info in [l, r].
52 Info query(int cr, int ll, int rr, int tl, int tr) {
53   if (tl <= ll && rr <= tr) return val[cr];
54   int mm = ll + ((rr - ll) >> 1);
55   Info res;
56   if (tl <= mm) res = query(lc(cr), ll, mm, tl, tr);
57   if (mm < tr) res += query(rc(cr), mm + 1, rr, tl, tr);
58   return res;
59 }
60
61 Info query(int l, int r) { return query(rt, L, R, l, r); }
62
```

- 带懒惰标记

```
1  // Segment Tree Implementation. Recursive. With lazy tag.
2  // Recursive structure, stored in a full binary tree.
3  #define lc(x) ch[(x)][0]
4  #define rc(x) ch[(x)][1]
5
6  int rt, id, L, R;
7  std::vector<std::array<int, 2>> ch;
8  std::vector<Info> val;
9  std::vector<Transform> lazy;
10
11 void push_up(int cr) { val[cr] = val[lc(cr)] + val[rc(cr)]; }
12
13 // Build the tree based on info stored in vec (0-indexed).
14 void build(int cr, int ll, int rr, const std::vector<Info>& vec) {
15   if (ll == rr) return (void)(val[cr] = vec[ll - L]);
16   int mm = ll + ((rr - ll) >> 1);
17   build(lc(cr) = ++id, ll, mm, vec);
18   build(rc(cr) = ++id, mm + 1, rr, vec);
19   push_up(cr);
20 }
21
22 void build(int l, int r, const std::vector<Info>& vec) {
23   rt = 1, id = 0, L = l, R = r;
24   int n = R - L + 1;
25   ch.resize(n << 1), val.resize(n << 1), lazy.resize(n << 1);
26   build(rt = ++id, L, R, vec);
27 }
28
29 // Lazy update.
30 void lazy_update(int cr, const Transform& f) {
31   val[cr] = f(val[cr]);
32   lazy[cr] += f;
33 }
34
35 // Push down lazy tag.
```

```
void push_down(int cr) {
  if (!lazy[cr]) return;
  lazy_update(lc(cr), lazy[cr]);
  lazy_update(rc(cr), lazy[cr]);
  lazy[cr] = Transform();
}

// Query info at x.
Info query(int cr, int ll, int rr, int x) {
  if (ll == rr) return val[cr];
  push_down(cr);
  int mm = ll + ((rr - ll) >> 1);
  if (x <= mm)
    return query(lc(cr), ll, mm, x);
  else
    return query(rc(cr), mm + 1, rr, x);
}

Info query(int x) { return query(rt, L, R, x); }

// Apply transformation f to the value at x.
void modify(int cr, int ll, int rr, int x, const Transform& f) {
  if (ll == rr) return (void)(val[cr] = f(val[cr]));
  push_down(cr);
  int mm = ll + ((rr - ll) >> 1);
  if (x <= mm)
    modify(lc(cr), ll, mm, x, f);
  else
    modify(rc(cr), mm + 1, rr, x, f);
  push_up(cr);
}

void modify(int x, const Transform& f) { modify(rt, L, R, x, f); }

// Query info in [l, r].
Info query(int cr, int ll, int rr, int tl, int tr) {
  if (tl <= ll && rr <= tr) return val[cr];
  push_down(cr);
  int mm = ll + ((rr - ll) >> 1);
  Info res;
  if (tl <= mm) res = query(lc(cr), ll, mm, tl, tr);
  if (mm < tr) res += query(rc(cr), mm + 1, rr, tl, tr);
  return res;
}

Info query(int l, int r) { return query(rt, L, R, l, r); }

// Apply transformation f to the range [l, r].
void modify(int cr, int ll, int rr, int tl, int tr, const Transform&
f) {
  if (tl <= ll && rr <= tr) return lazy_update(cr, f);
  push_down(cr);
```

```
  int mm = ll + ((rr - ll) >> 1);
  if (tl <= mm) modify(lc(cr), ll, mm, tl, tr, f);
  if (mm < tr) modify(rc(cr), mm + 1, rr, tl, tr, f);
  push_up(cr);
}

void modify(int l, int r, const Transform& f) { modify(rt, L, R, l,
r, f); }

```

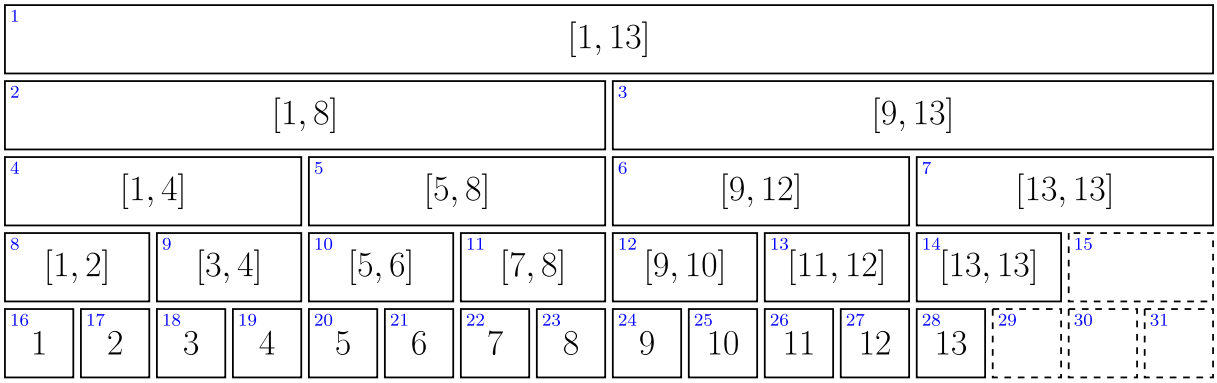
具体应用时，可能需要判断修改和查询操作是否合法.

常用技巧

本节介绍用线段树解决具体问题时常用的几类技巧. 它们或优化了算法常数，或丰富了基本功能.

非递归实现

前文介绍的递归实现的线段树，都只能自上而下地递归访问，需要支付相应的递归开销. 另一种方法是自下而上地维护线段树. 在国内竞赛资料中，这种实现因张昆玮介绍而广为流传，所以它也常称为 **zkw 线段树**.



上图展示了非递归实现的存储方式. 它仍然将线段树嵌入完美二叉树中，但改为自下而上建树. 首先，将所有叶结点存储在深度为 $\lceil \log_2 N \rceil$ 的同一层，多余的叶结点存储单位元 e，可以看作是空区间对应的信息. 然后，自下而上遍历非叶结点，将子结点的信息合并到当前结点. 由于所有叶结点都排在同一层，这样建立的线段树与前两种方式结构略有差异，但同样可以维护区间信息. 空间上与堆式存储相同，需要长度为 $2^{\lceil \log_2 N \rceil + 1}$ 的数组.

这种存储方式的最大好处是，叶结点编号连续，容易定位，且父子结点的编号都可以直接计算. 如上图所示，若设 $n = 2^{\lceil \log_2 N \rceil}$，即把 N 向上补齐到 2 的幂，则元素 $x \in [L, R]$ 处的值就存储在编号为 $n + x - L$ 的结点处；对于编号为 i 的非根结点，父结点编号是 $\lfloor i/2 \rfloor$，更一般地，d 级祖先结点的编号是 $\lfloor i/2^d \rfloor$；对于编号为 i 的非叶结点，左右子结点编号分别是 $2i$ 和 $2i + 1$.

根据这些特性，建树操作和单点操作都非常容易实现. 建树操作只需要将元素对应信息拷贝到对应叶结点上，然后倒序遍历结点 $[1, n - 1]$ 合并子结点信息即可. 单点查询可以直接返回对应叶结点存储的信息. 单点修改同样可以直接修改对应叶结点，同时还需要更新其祖先结点的信息.

参考实现

```
1 // Nonrecursive structure, embedded in a perfect binary tree.
```

```
int n, L, R;
std::vector<Info> val;

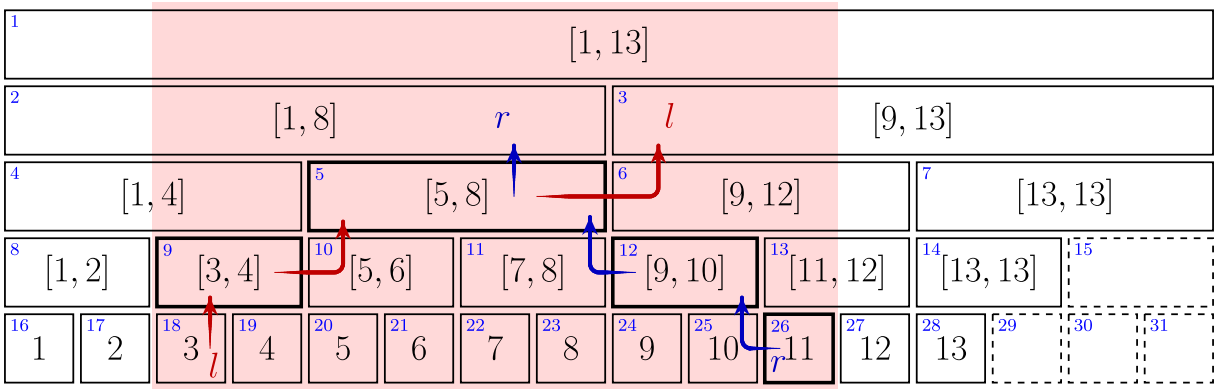
void push_up(int cr) { val[cr] = val[cr << 1] + val[(cr << 1) | 1]; }

// Build the tree based on info stored in vec (0-indexed).
void build(int l, int r, const std::vector<Info>& vec) {
  L = l, R = r;
  for (n = 1; n < r - l + 1; n <<= 1);
  val.resize(n << 1);
  std::copy(vec.begin(), vec.end(), val.begin() + n);
  for (int i = n - 1; i; --i) push_up(i);
}

// Query info at x.
Info query(int x) { return val[n + x - L]; }

// Modify info at x to v.
void modify(int x, const Info& v) {
  x = x - L + n;
  val[x] = v;
  for (x >>= 1; x; x >>= 1) push_up(x);
}

```



区间查询操作稍微复杂一些，需要找到区间 $[l, r]$ 在线段树上的所有极大区间结点．不妨暂且忽略查询区间右端点，先考虑左端点 l 所在的极大区间．包含它的所有区间结点就是 l 对应叶结点及其祖先．于是，从 l 对应叶结点出发不断上跳，直到当前结点是其父结点的右子结点为止：此时父结点还包含当前结点左侧的元素，而它们并不属于尚未统计的部分，所以当前结点对应的区间就是 l 所在的极大区间．累计它的信息后，在同一层向右移动一个结点，再跳到父结点，就得到尚未统计的最左元素所在的结点，重复上述操作即可．对于右端点的处理是对称的．实际实现时需要左右同时进行，直到两侧相遇，此时查询区间恰好统计完毕．需要注意的是，累计信息时需要左右分别统计，最后再将两侧的结果合并，以保证信息合并的顺序正确．

参考实现

```
// Query info in [l, r].
Info query(int l, int r) {
```

```
  Info la, ra;
  for (l = l - L + n, r = r - L + n; l <= r; l >>= 1, r >>= 1) {
    if (l & 1) la += val[l++];
    if (~r & 1) ra = val[r--] + ra;
  }
  return la + ra;
}

```

提示

对于区间查询，不同实现存在细微差别．此处实现的是闭区间 $[l,r]$ 查询，AtCoder Library[1] 实现的是 $[l,r)$ 查询，张昆玮原始资料实现的是 (l,r) 查询．三者没有本质区别，只是边界的处理方式不同．

最后，考虑区间修改操作，关键在于懒惰标记的处理．结点的访问仍然可以自下而上进行，但标记的下传必须自上而下，否则无法清空自根到叶路径上的全部标记．因此，只要找出极大区间结点的所有祖先结点，自上而下依次下传标记即可．为此，注意到极大区间结点的父结点必然包含左右端点之一，而包含某个端点的结点都位于自根到该端点对应叶结点的路径上．不过，这条路径上并非每个结点都是极大区间结点的祖先：自包含该端点的极大区间结点起，往下直到叶结点的一段就不是，处理时应当跳过．所以，只要沿着左右端点对应叶结点的两条路径，自根向叶下传一遍标记，跳过这一段即可．修改完成后，再沿同样的两条路径自下而上地更新祖先结点的信息．

参考实现

```
// Apply transformation f to the range [l, r].
void modify(int l, int r, const Transform& f) {
  l = l - L + n, r = r - L + n;
  for (int i = h, m = (1 << h) - 1; i; --i, m >>= 1) {
    if (l & m) push_down(l >> i);
    if (~r & m) push_down(r >> i);
  }
  for (int ll = l, rr = r; ll <= rr; ll >>= 1, rr >>= 1) {
    if (ll & 1) lazy_update(ll++, f);
    if (~rr & 1) lazy_update(rr--, f);
  }
  for (int i = 1, m = 1; i <= h; ++i, m = (m << 1) | 1) {
    if (l & m) push_up(l >> i);
    if (~r & m) push_up(r >> i);
  }
}

```

提示

1. 此处实现的仍是闭区间 $[l,r]$ 修改，半闭半开区间 $[l,r)$ 版本可以参考 AtCoder Library[2]．两个版本都应用了位操作，核心想法也一致：应当跳过的那一段结点，其对应区间的左（右）端点与修改区间的左（右）端点对齐．
2. 这一段结点，下传时跳过与否只影响效率，回溯时却必须跳过：对刚打上懒惰标记的结点调用 `push_up`，会用尚未更新的子结点信息覆盖它．

带懒惰标记的非递归实现中，其余操作也只需类似地在访问前增加自根向叶的下传.

非递归实现线段树的完整参考实现如下，同样分为不带懒惰标记和带懒惰标记两个版本：

参考实现

- 不带懒惰标记

```cpp
// Segment Tree Implementation. Nonrecursive. No lazy tag.
// Nonrecursive structure, embedded in a perfect binary tree.
int n, L, R;
std::vector<Info> val;

void push_up(int cr) { val[cr] = val[cr << 1] + val[(cr << 1) | 1]; }

// Build the tree based on info stored in vec (0-indexed).
void build(int l, int r, const std::vector<Info>& vec) {
  L = l, R = r;
  for (n = 1; n < r - l + 1; n <<= 1);
  val.resize(n << 1);
  std::copy(vec.begin(), vec.end(), val.begin() + n);
  for (int i = n - 1; i; --i) push_up(i);
}

// Query info at x.
Info query(int x) { return val[n + x - L]; }

// Modify info at x to v.
void modify(int x, const Info& v) {
  x = x - L + n;
  val[x] = v;
  for (x >>= 1; x; x >>= 1) push_up(x);
}

// Query info in [l, r].
Info query(int l, int r) {
  Info la, ra;
  for (l = l - L + n, r = r - L + n; l <= r; l >>= 1, r >>= 1) {
    if (l & 1) la += val[l++];
    if (~r & 1) ra = val[r--] + ra;
  }
  return la + ra;
}

```

- 带懒惰标记

```cpp
// Segment Tree Implementation. Nonrecursive. With lazy tag.
// Nonrecursive structure, embedded in a perfect binary tree.
int n, h, L, R;
std::vector<Info> val;
std::vector<Transform> lazy;

void push_up(int cr) { val[cr] = val[cr << 1] + val[(cr << 1) | 1]; }

// Build the tree based on info stored in vec (0-indexed).
```

```cpp
void build(int l, int r, const std::vector<Info>& vec) {
  L = l, R = r, n = r - l + 1;
  for (h = 0; (1 << h) < n; ++h);
  n = 1 << h;
  val.resize(n << 1), lazy.resize(n);
  std::copy(vec.begin(), vec.end(), val.begin() + n);
  for (int i = n - 1; i; --i) push_up(i);
}

// Lazy update.
void lazy_update(int cr, const Transform& f) {
  val[cr] = f(val[cr]);
  if (cr < n) lazy[cr] += f;
}

// Push down lazy tag.
void push_down(int cr) {
  if (!lazy[cr]) return;
  lazy_update(cr << 1, lazy[cr]);
  lazy_update((cr << 1) | 1, lazy[cr]);
  lazy[cr] = Transform();
}

// Query info at x.
Info query(int x) {
  x = x - L + n;
  for (int i = h; i; --i) push_down(x >> i);
  return val[x];
}

// Apply transformation f to the value at x.
void modify(int x, const Transform& f) {
  x = x - L + n;
  for (int i = h; i; --i) push_down(x >> i);
  val[x] = f(val[x]);
  for (int i = 1; i <= h; ++i) push_up(x >> i);
}

// Query info in [l, r].
Info query(int l, int r) {
  l = l - L + n, r = r - L + n;
  for (int i = h, m = (1 << h) - 1; i; --i, m >>= 1) {
    if (l & m) push_down(l >> i);
    if (~r & m) push_down(r >> i);
  }
  Info la, ra;
  for (; l <= r; l >>= 1, r >>= 1) {
    if (l & 1) la += val[l++];
    if (~r & 1) ra = val[r--] + ra;
  }
  return la + ra;
}
```

```
62
63 // Apply transformation f to the range [l, r].
64 void modify(int l, int r, const Transform& f) {
65   l = l - L + n, r = r - L + n;
66   for (int i = h, m = (1 << h) - 1; i; --i, m >>= 1) {
67     if (l & m) push_down(l >> i);
68     if (~r & m) push_down(r >> i);
69   }
70   for (int ll = l, rr = r; ll <= rr; ll >>= 1, rr >>= 1) {
71     if (ll & 1) lazy_update(ll++, f);
72     if (~rr & 1) lazy_update(rr--, f);
73   }
74   for (int i = 1, m = 1; i <= h; ++i, m = (m << 1) | 1) {
75     if (l & m) push_up(l >> i);
76     if (~r & m) push_up(r >> i);
77   }
78 }
79
```

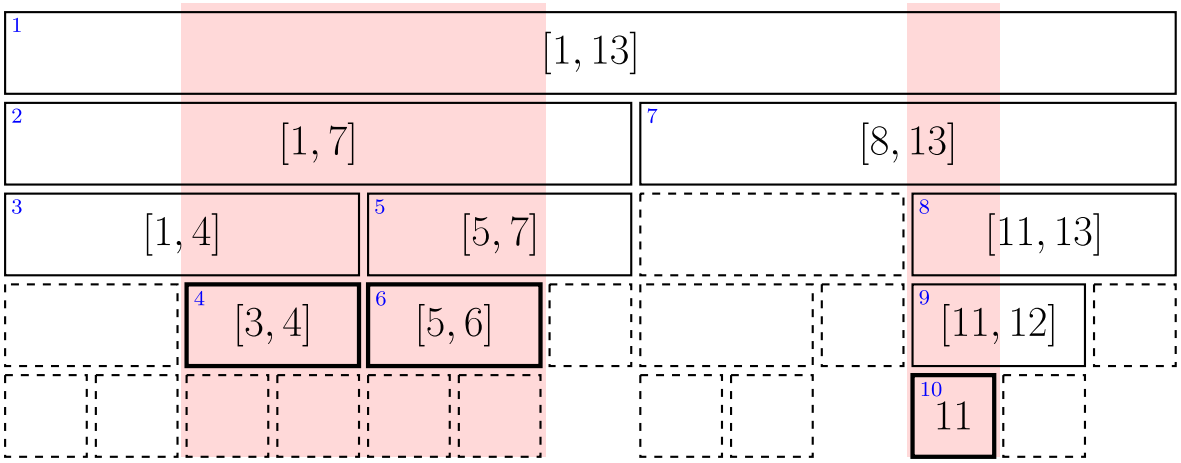
除了不带懒惰标记线段树的单点查询复杂度降低到 $O(1)$ 外，其余操作复杂度和递归实现版本相同，只是常数较小.

综上，非递归实现同样可以支持线段树的各种基本操作，包括区间修改和懒惰标记. 但它和堆式存储一样，结点编号由位置决定，要求整棵完美二叉树的结点预先存在，因而无法动态开点，也不便可持久化.

动态开点

前文介绍的所有存储方式，都需要在建树时一次性地分配大小为 $\Theta(N)$ 的存储空间. 这样做在区间长度 N 较大（例如 $\sim 10^9$）时并不可行. 为了节省空间，可以不一次性建好树，而是在最初只建立一个根结点代表整段区间. 当需要访问某个子区间时，才建立代表这段区间的子结点. 由于结点编号不固定，动态开点线段树只能采用内存池动态分配的方式实现，需要用数组记录子结点编号.

这样做的前提是，尚未建立的结点对应的信息是已知的. 最简单的情形是，未建立的结点对应区间的信息恰为信息空间 M 中的单位元 e（即空区间对应的信息）. 此时，查询遇到空结点直接返回 e 即可，合并时也无需特殊处理；只有修改操作需要写入信息时，才沿途建立经过的结点. 更一般的情形是，未建立的结点的信息与其对应区间有关. 对于这类问题，查询空结点、创建新结点以及合并子结点信息时，都需要根据当前区间的端点计算出相应的信息. 但无论如何，动态开点线段树通常无法从任意给定的初始序列建树. 此外，由于结点编号不再蕴含区间信息，划分区间时需要沿用递归实现的做法，将当前结点对应的区间作为参数逐层传递；格外需要注意的是，下传懒惰标记前，也需要先建立子结点.



作为参考，此处给出不带懒惰标记和带懒惰标记的动态开点线段树实现. 为对照上文两种情形，前者假定未建立结点的信息恰为单位元 e，后者则假定它与所处区间有关.

参考实现

- 不带懒惰标记

```cpp
// Segment Tree Implementation. Dynamic node allocation.
// Dynamic node allocation.
#define lc(x) ch[(x)][0]
#define rc(x) ch[(x)][1]

int rt, id, L, R;
std::vector<std::array<int, 2>> ch;
std::vector<Info> val;

void push_up(int cr) { val[cr] = val[lc(cr)] + val[rc(cr)]; }

// Build an empty tree.
void build(int l, int r, int n) {
  rt = 0, id = 0, L = l, R = r;
  ch.resize(n), val.resize(n);
}

// Query info at x.
Info query(int cr, int ll, int rr, int x) {
  if (!cr) return {};
  if (ll == rr) return val[cr];
  int mm = ll + ((rr - ll) >> 1);
  if (x <= mm)
    return query(lc(cr), ll, mm, x);
  else
    return query(rc(cr), mm + 1, rr, x);
}

Info query(int x) { return query(rt, L, R, x); }

// Modify info at x to v.
void modify(int& cr, int ll, int rr, int x, const Info& v) {
  if (!cr) cr = ++id;
```

```
  if (ll == rr) return (void)(val[cr] = v);
  int mm = ll + ((rr - ll) >> 1);
  if (x <= mm)
    modify(lc(cr), ll, mm, x, v);
  else
    modify(rc(cr), mm + 1, rr, x, v);
  push_up(cr);
}

void modify(int x, const Info& v) { modify(rt, L, R, x, v); }

// Query info in [l, r].
Info query(int cr, int ll, int rr, int tl, int tr) {
  if (!cr) return {};
  if (tl <= ll && rr <= tr) return val[cr];
  int mm = ll + ((rr - ll) >> 1);
  Info res;
  if (tl <= mm) res = query(lc(cr), ll, mm, tl, tr);
  if (mm < tr) res += query(rc(cr), mm + 1, rr, tl, tr);
  return res;
}

Info query(int l, int r) { return query(rt, L, R, l, r); }

```

- 带懒惰标记

```
// Segment Tree Implementation. Dynamic node allocation.
// Recursive structure, stored in a full binary tree.
#define lc(x) ch[(x)][0]
#define rc(x) ch[(x)][1]

int rt, id, L, R;
std::vector<std::array<int, 2>> ch;
std::vector<Info> val;
std::vector<Transform> lazy;
Info blank(int ll, int rr);  // Info of untouched node; problem-specific.

Info info_of(int cr, int ll, int rr) { return cr ? val[cr] : blank(ll, rr); }

void push_up(int cr, int ll, int rr) {
  int mm = ll + ((rr - ll) >> 1);
  val[cr] = info_of(lc(cr), ll, mm) + info_of(rc(cr), mm + 1, rr);
}

// Build an empty tree.
void build(int l, int r, int n) {
  rt = 0, id = 0, L = l, R = r;
  ch.resize(n), val.resize(n), lazy.resize(n);
}

```

```cpp
// Lazy update.
void lazy_update(int& cr, int ll, int rr, const Transform& f) {
  if (!cr) val[cr = ++id] = blank(ll, rr);
  val[cr] = f(val[cr]);
  lazy[cr] += f;
}

// Push down lazy tag.
void push_down(int cr, int ll, int rr) {
  if (!lazy[cr]) return;
  int mm = ll + ((rr - ll) >> 1);
  lazy_update(lc(cr), ll, mm, lazy[cr]);
  lazy_update(rc(cr), mm + 1, rr, lazy[cr]);
  lazy[cr] = Transform();
}

// Query info at x.
Info query(int cr, int ll, int rr, int x) {
  if (!cr) return blank(ll, rr);
  if (ll == rr) return val[cr];
  push_down(cr, ll, rr);
  int mm = ll + ((rr - ll) >> 1);
  if (x <= mm)
    return query(lc(cr), ll, mm, x);
  else
    return query(rc(cr), mm + 1, rr, x);
}

Info query(int x) { return query(rt, L, R, x); }

// Apply transformation f to the value at x.
void modify(int& cr, int ll, int rr, int x, const Transform& f) {
  if (!cr) val[cr = ++id] = blank(ll, rr);
  if (ll == rr) return (void)(val[cr] = f(val[cr]));
  push_down(cr, ll, rr);
  int mm = ll + ((rr - ll) >> 1);
  if (x <= mm)
    modify(lc(cr), ll, mm, x, f);
  else
    modify(rc(cr), mm + 1, rr, x, f);
  push_up(cr, ll, rr);
}

void modify(int x, const Transform& f) { modify(rt, L, R, x, f); }

// Query info in [l, r].
Info query(int cr, int ll, int rr, int tl, int tr) {
  if (!cr) return blank(std::max(ll, tl), std::min(rr, tr));
  if (tl <= ll && rr <= tr) return val[cr];
  push_down(cr, ll, rr);
  int mm = ll + ((rr - ll) >> 1);
  Info res;
```

```
  if (tl <= mm) res = query(lc(cr), ll, mm, tl, tr);
  if (mm < tr) res += query(rc(cr), mm + 1, rr, tl, tr);
  return res;
}

Info query(int l, int r) { return query(rt, L, R, l, r); }

// Apply transformation f to the range [l, r].
void modify(int& cr, int ll, int rr, int tl, int tr, const Transform&
f) {
  if (!cr) val[cr = ++id] = blank(ll, rr);
  if (tl <= ll && rr <= tr) return lazy_update(cr, ll, rr, f);
  push_down(cr, ll, rr);
  int mm = ll + ((rr - ll) >> 1);
  if (tl <= mm) modify(lc(cr), ll, mm, tl, tr, f);
  if (mm < tr) modify(rc(cr), mm + 1, rr, tl, tr, f);
  push_up(cr, ll, rr);
}

void modify(int l, int r, const Transform& f) { modify(rt, L, R, l,
r, f); }

```

提示

下传标记时需要先建立子结点，会增加空间常数因子，请分配好足量空间.

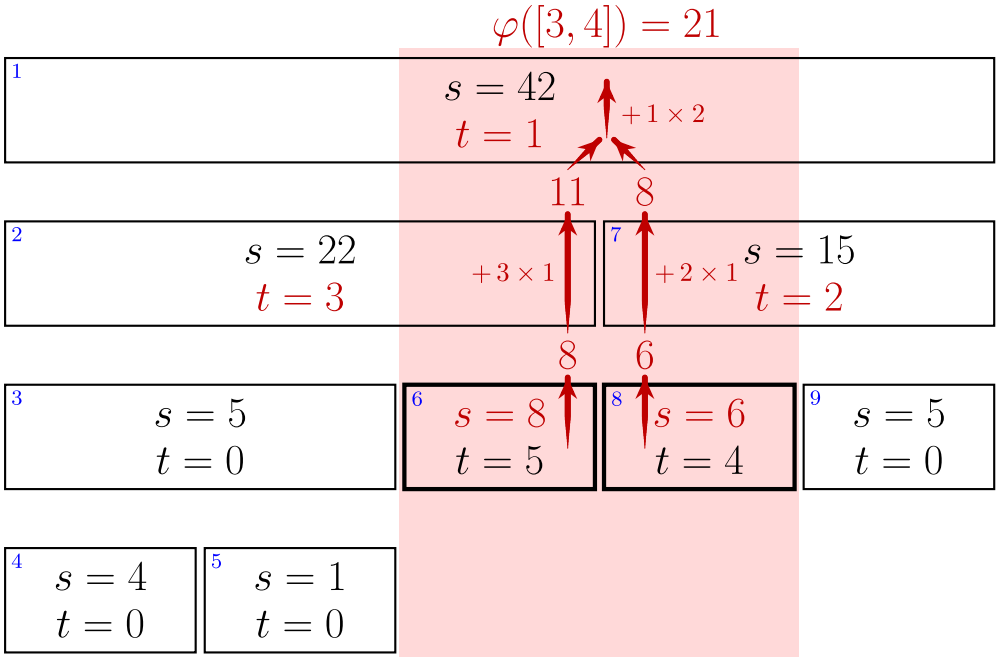
由于每次修改只沿着自根向叶的至多两条路径建立结点，单次操作至多新建 $O(\log N)$ 个结点. 因此，q 次操作后结点总数是 $O(q \log N)$ 的，这正是动态开点能处理大区间的原因. 时间复杂度则与前文相同，单次操作仍是 $O(\log N)$. 此处的 $O(q \log N)$ 是很宽松的上界. 实际使用时，若操作区间大量重叠，新建的结点会少得多；反之，若每次操作都涉及互不相交的区间，就会接近这个上界. 在初始化线段树时，应根据这一上界分配充足的空间.

动态开点只是处理这类问题的一种方法. 如果已知所有操作涉及的端点，可以先将它们 离散化 →13.2@p3170，再在离散化后的区间上建立普通线段树，效果相同且实现更简单；只有在操作无法预先得知（例如强制在线）时，才需要考虑使用动态开点.

此外，可持久化线段树→10.17.2@p2424 也建立在动态开点的基础上：每次修改只新建自根向叶一条路径上的结点，其余部分与旧版本共享，从而每次修改只需要 $O(\log N)$ 的额外空间.

标记永久化

前文介绍的懒惰标记都需要下传. 但是，每次下传都需要读写子结点，常数不小；而且，某些情形下难以下传标记. 为了避免下传懒惰标记，可以采用标记永久化的方法：标记一旦打上就永远留在原处，不再下传；查询时，再将自根到该结点沿途经过的标记复合起来，作用于该结点存储的信息.



和前文约定一致，设结点 x 处的标记只对其子孙结点生效，而 x 自身存储的信息已经计入了该标记的作用．换言之，x 存储的信息是不考虑其祖先结点处标记时，x 对应区间的信息．以递归实现为例，线段树区间操作的实现分别是：

- 区间修改：自上而下递归．递归到极大区间结点时，在该处打上标记，并更新其信息，随即返回，不再向下．回溯时，用子结点的信息合并出当前结点的信息，再作用上当前结点的标记．
- 区间查询：自上而下递归．递归到极大区间结点时，直接返回它存储的信息．回溯时，将各子结点返回的结果合并，再作用上当前结点的标记；逐层向上，沿途结点的标记便都作用到了结果上．

两种操作时间复杂度仍然是 $O(\log N)$，但省去了全部下传，常数更小．非递归实现类似，只是将递归换成了迭代．它们的参考实现如下：

参考实现

- 递归实现

```cpp
// Segment Tree Implementation. Recursive. With lazy tag.
// Recursive structure, stored in a full binary tree.
#define lc(x) ch[(x)][0]
#define rc(x) ch[(x)][1]

int rt, id, L, R;
std::vector<std::array<int, 2>> ch;
std::vector<Info> val;
std::vector<Transform> lazy;

void push_up(int cr) { val[cr] = lazy[cr](val[lc(cr)] + val[rc(cr)]); }

// Build the tree based on info stored in vec (0-indexed).
void build(int cr, int ll, int rr, const std::vector<Info>& vec) {
  if (ll == rr) return (void)(val[cr] = vec[ll - L]);
  int mm = ll + ((rr - ll) >> 1);
  build(lc(cr) = ++id, ll, mm, vec);
  build(rc(cr) = ++id, mm + 1, rr, vec);
  push_up(cr);
```

```
}

void build(int l, int r, const std::vector<Info>& vec) {
  rt = 1, id = 0, L = l, R = r;
  int n = R - L + 1;
  ch.resize(n << 1), val.resize(n << 1), lazy.resize(n << 1);
  build(rt = ++id, L, R, vec);
}

// Lazy update.
void lazy_update(int cr, const Transform& f) {
  val[cr] = f(val[cr]);
  lazy[cr] += f;
}

// Query info at x.
Info query(int cr, int ll, int rr, int x) {
  if (ll == rr) return val[cr];
  int mm = ll + ((rr - ll) >> 1);
  return lazy[cr](x <= mm ? query(lc(cr), ll, mm, x)
                          : query(rc(cr), mm + 1, rr, x));
}

Info query(int x) { return query(rt, L, R, x); }

// Apply transformation f to the value at x.
void modify(int cr, int ll, int rr, int x, const Transform& f) {
  if (ll == rr) return (void)(val[cr] = f(val[cr]));
  int mm = ll + ((rr - ll) >> 1);
  if (x <= mm)
    modify(lc(cr), ll, mm, x, f);
  else
    modify(rc(cr), mm + 1, rr, x, f);
  push_up(cr);
}

void modify(int x, const Transform& f) { modify(rt, L, R, x, f); }

// Query info in [l, r].
Info query(int cr, int ll, int rr, int tl, int tr) {
  if (tl <= ll && rr <= tr) return val[cr];
  int mm = ll + ((rr - ll) >> 1);
  Info res;
  if (tl <= mm) res = query(lc(cr), ll, mm, tl, tr);
  if (mm < tr) res += query(rc(cr), mm + 1, rr, tl, tr);
  return lazy[cr](res);
}

Info query(int l, int r) { return query(rt, L, R, l, r); }

// Apply transformation f to the range [l, r].
```

```
void modify(int cr, int ll, int rr, int tl, int tr, const Transform&
f) {
  if (tl <= ll && rr <= tr) return lazy_update(cr, f);
  int mm = ll + ((rr - ll) >> 1);
  if (tl <= mm) modify(lc(cr), ll, mm, tl, tr, f);
  if (mm < tr) modify(rc(cr), mm + 1, rr, tl, tr, f);
  push_up(cr);
}

void modify(int l, int r, const Transform& f) { modify(rt, L, R, l,
r, f); }

```

- 非递归实现

```
// Segment Tree Implementation. Nonrecursive. With lazy tag.
// Nonrecursive structure, embedded in a perfect binary tree.
int n, h, L, R;
std::vector<Info> val;
std::vector<Transform> lazy;

void push_up(int cr) { val[cr] = lazy[cr](val[cr << 1] + val[(cr << 1)
| 1]); }

// Build the tree based on info stored in vec (0-indexed).
void build(int l, int r, const std::vector<Info>& vec) {
  L = l, R = r, n = r - l + 1;
  for (h = 0; (1 << h) < n; ++h);
  n = 1 << h;
  val.resize(n << 1), lazy.resize(n);
  std::copy(vec.begin(), vec.end(), val.begin() + n);
  for (int i = n - 1; i; --i) push_up(i);
}

// Lazy update.
void lazy_update(int cr, const Transform& f) {
  val[cr] = f(val[cr]);
  if (cr < n) lazy[cr] += f;
}

// Query info at x.
Info query(int x) {
  x = x - L + n;
  auto res = val[x];
  for (int i = 1; i <= h; ++i) res = lazy[x >> i](res);
  return res;
}

// Apply transformation f to the value at x.
void modify(int x, const Transform& f) {
  x = x - L + n;
  val[x] = f(val[x]);
  for (int i = 1; i <= h; ++i) push_up(x >> i);
```

```
}

// Query info in [l, r].
Info query(int l, int r) {
  l = l - L + n, r = r - L + n;
  Info la, ra;
  for (int i = 0, ll = l, rr = r; i <= h; ++i) {
    if (i) la = lazy[l >> i](la), ra = lazy[r >> i](ra);
    if (ll <= rr) {
      if (ll & 1) la += val[ll++];
      if (~rr & 1) ra = val[rr--] + ra;
      ll >>= 1, rr >>= 1;
    }
  }
  return la + ra;
}

// Apply transformation f to the range [l, r].
void modify(int l, int r, const Transform& f) {
  l = l - L + n, r = r - L + n;
  for (int ll = l, rr = r; ll <= rr; ll >>= 1, rr >>= 1) {
    if (ll & 1) lazy_update(ll++, f);
    if (~rr & 1) lazy_update(rr--, f);
  }
  for (int i = 1, m = 1; i <= h; ++i, m = (m << 1) | 1) {
    if (l & m) push_up(l >> i);
    if (~r & m) push_up(r >> i);
  }
}

```

但是，标记永久化并非总是可行，它对修改操作有额外要求．修改操作的复合必须满足 **交换律**．前文说明过，懒惰标记的复合未必可交换，而下传之所以能保证顺序正确，是因为每次访问子结点前都会先下传，从而根到叶的路径上，越靠近根的标记对应的操作总是越晚发生．标记永久化恰恰放弃了这一性质：标记始终停留在原处，位置由修改区间决定，与操作发生的先后无关．于是，祖先结点上的标记既可能早于、也可能晚于子孙结点上的标记．查询时按自根向叶的顺序复合标记，只有在复合可交换时才能保证结果正确．

例子

区间加、区间乘等操作都是可交换的，可以标记永久化．区间仿射变换（即同时允许区间加和区间乘）、区间赋值[5]则不可交换，无法标记永久化．

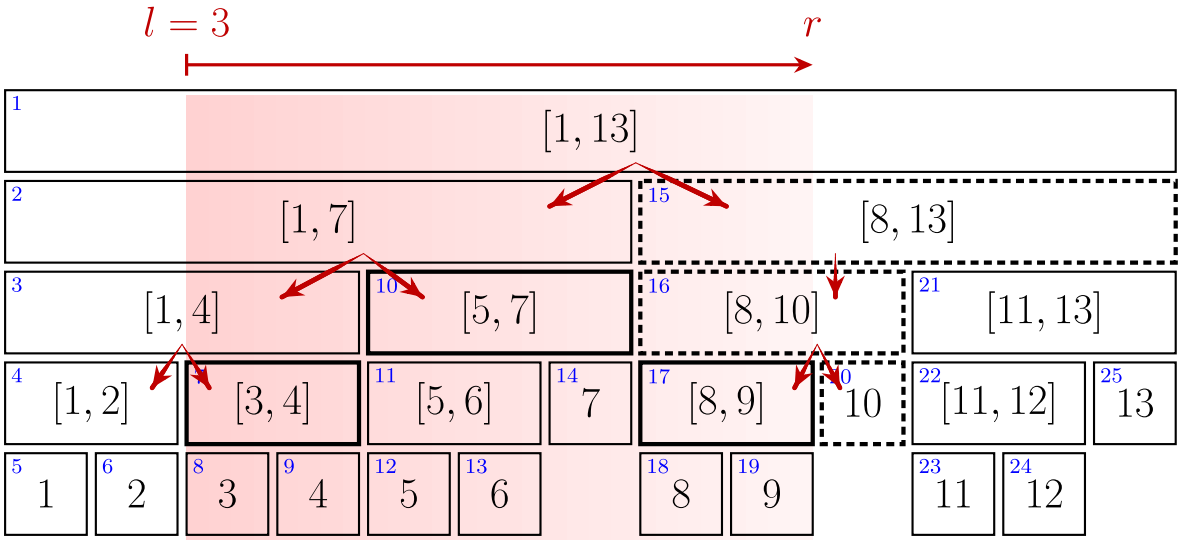
标记永久化经常应用于 可持久化线段树→10.17.2@p2424 和各类树套树中．这两种情形下，下传代价高昂；标记永久化即使不是必需的，也往往是更合适的选择．还有一些问题中，标记根本无从下传（见后文「矩形面积并」例题），此时标记永久化就是唯一的办法．

[5]区间赋值操作本身不可交换．如果同时记录赋值操作的时间戳 t，并规定标记 (v_1,t_1) 与 (v_2,t_2) 的复合取其中时间戳较大者，就得到了可交换的标记．利用这一方法，可以实现支持区间赋值和单点查询的标记永久化线段树：查询时沿途取时间戳最大的标记，作用于叶结点即可．但是，它无法支持区间查询，因为 (v,t) 并非信息空间上的自同态：一段区间中，哪些位置已有更晚的赋值覆盖，无法从这段区间的信息中得知．

线段树二分

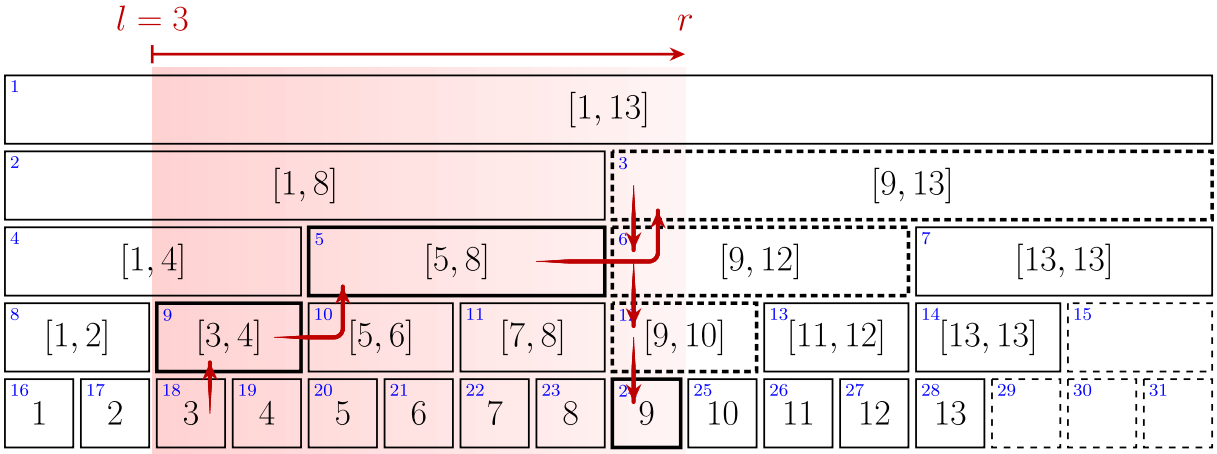
有些问题需要在序列上二分．例如，给定 l，求最大的 r，使得区间 $[l,r]$ 的信息满足某个条件．直接的做法是二分 r，每次用线段树查询一遍区间信息，复杂度是 $O(\log^2 N)$ 的．但是，线段树本身就是一棵二分的结构，把二分的过程放到线段树上进行，就可以做到 $O(\log N)$．

仍设线段树存储的区间信息为 $\varphi : \mathcal{I} \to M$．设条件是关于信息空间 M 的谓词 $g : M \to \{\text{True}, \text{False}\}$．要让二分有意义，$g$ 必须满足（向右合并的）**单调性**：若 $g(m)$ 为假，则对任意 m'，$g(m \circ m')$ 也为假．也就是说，一旦区间向右延伸到条件不再成立，就不会再重新成立．此外，还需要 $g(e)$ 为真，即空区间总是满足条件．于是，问题变为：给定 l，求最大的 r，使得 $g(\varphi([l,r]))$ 为真；若 $g(\varphi([l,l]))$ 已经为假，则约定答案为 $l-1$．



递归实现线段树中，二分的具体做法是，自上而下递归，同时维护一个累计信息 m，表示已经确定纳入答案区间的那一部分的信息．递归到某个结点时，若它对应的区间 I 完全落在 $[l,R]$ 内，就先试着把整个结点并入：计算 $m \circ \varphi(I)$，若 g 仍为真，说明该结点可以整个纳入，更新 m 并返回该结点的右端点；否则，答案的分界就在该结点内部，需要继续向下递归．递归到叶结点时，若仍不满足条件，说明分界就在此处．

这样做的复杂度仍然是 $O(\log N)$ 的．这是因为，递归过程中只有两类结点会继续向下：一类是对应区间越出 $[l,R]$ 的结点，它们都在自根到 l 的路径上；另一类是整个并入会使 g 变假的结点，其中只有分界所在的那条路径上的结点会真正向下，每层至多一个．两类结点都只有 $O(\log N)$ 个，其余结点要么整个纳入，要么整个跳过．



非递归实现把上述过程拆成两个阶段. 先自下而上: 从 l 对应叶结点出发, 只要当前结点是左子结点就上跳, 直到它是右子结点为止, 此时它对应区间的左端点恰与尚未纳入的部分对齐, 于是尝试整个并入; 成功则向右移动一个结点, 继续上跳. 一旦某个结点并入后 g 变假, 就转为自上而下: 在该结点内部下行, 每层先尝试并入左子结点, 成功则转向右子结点, 失败则进入左子结点, 直到叶结点, 即得分界. 两个阶段各只走一趟, 复杂度同样是 $O(\log N)$.

对称地, 也可以给定 r, 求最小的 l, 使得 $g(\varphi([l,r]))$ 为真. 此时累计信息应当从右向左合并, 即计算 $\varphi(I)\circ m$, 以保证合并顺序正确; 无解时约定答案为 $r+1$. 而且, g 需要满足向左合并的单调性: 若 $g(m)$ 为假, 则对任意 m', $g(m'\circ m)$ 也为假.

参考实现

- 递归实现

```cpp
// Find max r in [l, R] such that g(info([l, r])) is true.
// Return l - 1 if no such r exists.
template <typename G>
int max_right(int cr, int ll, int rr, int tl, const G& g, Info& acc) {
  if (tl <= ll) {
    auto nxt = acc + val[cr];
    if (g(nxt)) return acc = nxt, rr;
    if (ll == rr) return ll - 1;
  }
  push_down(cr);
  int mm = ll + ((rr - ll) >> 1);
  if (tl <= mm) {
    auto res = max_right(lc(cr), ll, mm, tl, g, acc);
    if (res < mm) return res;
  }
  return max_right(rc(cr), mm + 1, rr, tl, g, acc);
}

template <typename G>
int max_right(int l, const G& g) {
  Info acc;
  return max_right(rt, L, R, l, g, acc);
}

// Find min l in [L, r] such that g(info([l, r])) is true.
// Return r + 1 if no such l exists.
template <typename G>
int min_left(int cr, int ll, int rr, int tr, const G& g, Info& acc) {
  if (tr >= rr) {
    auto nxt = val[cr] + acc;
    if (g(nxt)) return acc = nxt, ll;
    if (ll == rr) return rr + 1;
  }
  push_down(cr);
  int mm = ll + ((rr - ll) >> 1);
  if (tr > mm) {
    auto res = min_left(rc(cr), mm + 1, rr, tr, g, acc);
    if (res > mm + 1) return res;
  }
  return min_left(lc(cr), ll, mm, tr, g, acc);
```

```
}

template <typename G>
int min_left(int r, const G& g) {
  Info acc;
  return min_left(rt, L, R, r, g, acc);
}

```

- 非递归实现

```
// Find max r in [l, R] such that g(info([l, r])) is true.
// Return l - 1 if no such r exists.
template <typename G>
int max_right(int l, const G& g) {
  l = l - L + n;
  for (int i = h; i; --i) push_down(l >> i);
  Info acc;
  for (;;) {
    while (~l & 1) l >>= 1;
    auto nxt = acc + val[l];
    if (!g(nxt)) {
      while (l < n) {
        push_down(l);
        l <<= 1;
        nxt = acc + val[l];
        if (g(nxt)) acc = nxt, ++l;
      }
      return l - n + L - 1;
    }
    acc = nxt, ++l;
    if ((l & -l) == l) break;
  }
  return R;
}

// Find min l in [L, r] such that g(info([l, r])) is true.
// Return r + 1 if no such l exists.
template <typename G>
int min_left(int r, const G& g) {
  r = r - L + n;
  for (int i = h; i; --i) push_down(r >> i);
  Info acc;
  for (;;) {
    while ((r & 1) && (r ^ 1)) r >>= 1;
    auto nxt = val[r] + acc;
    if (!g(nxt)) {
      while (r < n) {
        push_down(r);
        r = (r << 1) | 1;
        nxt = val[r] + acc;
        if (g(nxt)) acc = nxt, --r;
      }
```

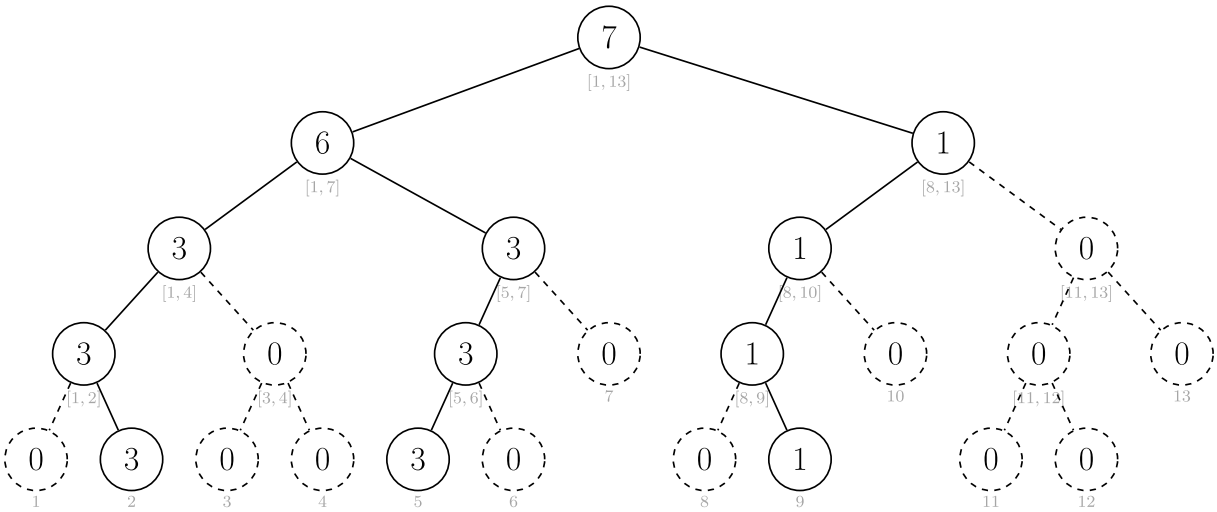
```
43        return r - n + L + 1;
44      }
45      if ((r & -r) == r) break;
46      acc = nxt, --r;
47    }
48    return L;
49  }
50
```

提示

1. 带懒惰标记时，向下访问子结点前同样需要下传标记. 非递归实现中，上跳阶段开始前需要先沿自根到 l 的路径下传一遍.
2. 若判定只与当前结点自身的信息有关（例如「区间内是否存在大于 x 的元素」），则无需维护累计信息，实现更为简单.
3. 非递归实现中，多余的叶结点存储单位元，g 在其上恒为真，因此上跳阶段可能一路越过它们. 此时应直接返回 R，而不是由结点编号换算位置.

权值线段树

前文的线段树都建立在序列的下标上. 但是，线段树并不关心下标的含义. 如果把它建立在值域上，即让元素 v 对应的叶结点记录值 v 在可重集中出现的次数，就得到 **权值线段树**.



这样的线段树可以当作一个支持 平衡树→10.15.1@p2213 常见操作的集合来使用，而这些操作都可以由前文的操作直接得到：

- 插入元素 v，即在位置 v 处单点加一.
- 删除元素 v，即在位置 v 处单点减一；注意判断元素是否存在.
- 查询 v 的排名，即查询值域区间 $[L, v-1]$ 上的区间和再加一.
- 查询第 k 小的元素，即利用线段树二分，求最大的 r，使得值域区间 $[L, r]$ 上的元素个数小于 k；此时 $r+1$ 就是所求. 由于计数可以相减，下行时只需比较左子结点的计数与 k，无需维护累计信息.
- 查询前驱与后继，可以由查询排名和查询第 k 小元素两个操作实现.

与常见平衡树相比，权值线段树的实现要简单得多，常数也更小；代价是只能处理值域上的元素，且不支持区间翻转等依赖树形结构本身的操作.

由于值域通常远大于元素个数，直接建树往往不可行．若所有元素可以预先得知，将值域 离散化 →13.2@p3170 后建立普通线段树即可；否则，就需要动态开点．后一种情形下，未出现的值对应的结点信息恰为单位元，正是前文最简单的那种情形．

参考实现

```cpp
// Segment tree on values, supporting the ordered-set operations of a BST.
int rt, id, L, R;
std::vector<int> lc, rc;
std::vector<int> cnt;

void build(int l, int r, int n) {
  rt = 0, id = 0, L = l, R = r;
  lc.resize(n), rc.resize(n), cnt.resize(n);
}

// Insert v.
void insert(int& cr, int ll, int rr, int v) {
  if (!cr) cr = ++id;
  ++cnt[cr];
  if (ll == rr) return;
  int mm = ll + ((rr - ll) >> 1);
  if (v <= mm)
    insert(lc[cr], ll, mm, v);
  else
    insert(rc[cr], mm + 1, rr, v);
}

void insert(int v) { insert(rt, L, R, v); }

// Remove v.
// If there are multiple, remove once.
// Return true if there is any, return false if there is none.
bool remove(int cr, int ll, int rr, int v) {
  if (!cnt[cr]) return false;
  if (ll == rr) return cnt[cr]--;
  int mm = ll + ((rr - ll) >> 1);
  auto succ = false;
  if (v <= mm)
    succ = remove(lc[cr], ll, mm, v);
  else
    succ = remove(rc[cr], mm + 1, rr, v);
  return succ && cnt[cr]--;
}

bool remove(int v) { return remove(rt, L, R, v); }

// Count values within range [l, r].
int count(int cr, int ll, int rr, int tl, int tr) {
  if (!cr) return 0;
  if (tl <= ll && rr <= tr) return cnt[cr];
  int mm = ll + ((rr - ll) >> 1);
  int res = 0;
```

```
  if (tl <= mm) res = count(lc[cr], ll, mm, tl, tr);
  if (mm < tr) res += count(rc[cr], mm + 1, rr, tl, tr);
  return res;
}

int count(int l, int r) { return count(rt, L, R, l, r); }

// Find the rank of v, i.e., one plus the count of numbers less than v.
int find_rank(int v) { return v > L ? count(L, v - 1) + 1 : 1; }

// Find the k-th element.
int find_kth(int cr, int ll, int rr, int k) {
  if (ll == rr) return ll;
  int mm = ll + ((rr - ll) >> 1);
  if (k <= cnt[lc[cr]])
    return find_kth(lc[cr], ll, mm, k);
  else
    return find_kth(rc[cr], mm + 1, rr, k - cnt[lc[cr]]);
}

int find_kth(int k) {
  return k > cnt[rt] || k <= 0 ? -1 : find_kth(rt, L, R, k);
}

// Find the predecessor of v.
int find_prev(int x) { return find_kth(find_rank(x) - 1); }

// Find the successor of v.
int find_next(int x) { return find_kth(find_rank(x + 1)); }

```

权值线段树也是 可持久化线段树→10.17.2@p2424 和线段树合并等拓展最常见的载体．前者可以查询序列某个区间内的第 k 小，后者可以高效地合并两个集合．

拓展

线段树应用十分广泛，常见的拓展和变体如下：

- 可持久化线段树→10.17.2@p2424
- 各类树套树：
  - 线段树套线段树→10.18.1@p2440
  - 树状数组套线段树→10.18.4@p2443
  - 线段树套平衡树→10.18.3@p2441
  - 平衡树套线段树→10.18.2@p2441
- 李超线段树→10.13.3@p2180
- 猫树→10.13.4@p2187
- 吉司机线段树→10.13.5@p2188

详细内容请参阅相关页面．

例题

前文介绍了线段树的基本原理，并给出了几份模板实现．但线段树的实现相当灵活．本节的参考实现将不拘泥于模板．

Library Checker - Point Add Range Sum[3]

已知一个数列. 需要进行如下操作:

- 将第 p 个数增加 x.
- 求区间 $[l, r)$ 中元素的和.

解答

只要实现支持单点修改、区间查询的线段树即可.

- 实现 1

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

// --8<-- [start:info]
// Sum.
struct Info {
  long long sum;

  Info(long long x = 0) : sum(x) {}

  Info operator+(const Info& o) const { return Info(sum + o.sum); }

  Info& operator+=(const Info& o) { return *this = *this + o; }
};

// --8<-- [end:info]
// --8<-- [start:seg-tree]
// Segment tree.
class SegmentTree {
  int rt, id, L, R;
  std::vector<int> lc, rc;
  std::vector<Info> val;

  void push_up(int cr) { val[cr] = val[lc[cr]] + val[rc[cr]]; }

  void build(int cr, int ll, int rr, const std::vector<int>& vec) {
    if (ll == rr) return (void)(val[cr] = Info(vec[ll - L]));
    int mm = ll + ((rr - ll) >> 1);
    build(lc[cr] = ++id, ll, mm, vec);
    build(rc[cr] = ++id, mm + 1, rr, vec);
    push_up(cr);
  }

  void modify(int cr, int ll, int rr, int x, int v) {
    if (ll == rr) return (void)(val[cr] += Info(v));
    int mm = ll + ((rr - ll) >> 1);
    if (x <= mm)
      modify(lc[cr], ll, mm, x, v);
    else
      modify(rc[cr], mm + 1, rr, x, v);
    push_up(cr);
```

```
  }

  Info query(int cr, int ll, int rr, int tl, int tr) {
    if (tl <= ll && rr <= tr) return val[cr];
    int mm = ll + ((rr - ll) >> 1);
    Info res;
    if (tl <= mm) res = query(lc[cr], ll, mm, tl, tr);
    if (mm < tr) res += query(rc[cr], mm + 1, rr, tl, tr);
    return res;
  }

 public:
  SegmentTree(int n, const std::vector<int>& vec)
      : rt(0), id(0), L(0), R(n - 1), lc(n << 1), rc(n << 1), val(n << 1) {
    build(rt = ++id, L, R, vec);
  }

  void modify(int x, int v) { modify(rt, L, R, x, v); }

  Info query(int l, int r) { return query(rt, L, R, l, r); }
};

// --8<-- [end:seg-tree]
int main() {
  std::ios::sync_with_stdio(false), std::cin.tie(nullptr);
  int n, q;
  std::cin >> n >> q;
  std::vector<int> vec(n);
  for (auto& x : vec) std::cin >> x;
  SegmentTree seg(n, vec);
  for (; q; --q) {
    int op;
    std::cin >> op;
    if (op == 0) {
      int p, x;
      std::cin >> p >> x;
      seg.modify(p, x);
    } else if (op == 1) {
      int l, r;
      std::cin >> l >> r;
      std::cout << seg.query(l, r - 1).sum << '\n';
    }
  }
  return 0;
}

```

- 实现 2

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

```cpp

// --8<-- [start:seg-tree]
// Segment tree.
class SegmentTree {
  int n;
  std::vector<long long> sum;

 public:
  SegmentTree(int _n, const std::vector<int>& vec) {
    for (n = 1; n < _n; n <<= 1);
    sum.resize(n << 1);
    std::copy(vec.begin(), vec.end(), sum.begin() + n);
    for (int i = n - 1; i; --i) sum[i] = sum[i << 1] + sum[(i << 1) | 1];
  }

  void modify(int x, int v) {
    for (x += n; x; x >>= 1) sum[x] += v;
  }

  long long query(int l, int r) {
    l += n, r += n;
    long long res = 0;
    for (int ll = l, rr = r; ll <= rr; ll >>= 1, rr >>= 1) {
      if (ll & 1) res += sum[ll++];
      if (~rr & 1) res += sum[rr--];
    }
    return res;
  }
};

// --8<-- [end:seg-tree]
int main() {
  std::ios::sync_with_stdio(false), std::cin.tie(nullptr);
  int n, q;
  std::cin >> n >> q;
  std::vector<int> vec(n);
  for (auto& x : vec) std::cin >> x;
  SegmentTree seg(n, vec);
  for (; q; --q) {
    int op;
    std::cin >> op;
    if (op == 0) {
      int p, x;
      std::cin >> p >> x;
      seg.modify(p, x);
    } else if (op == 1) {
      int l, r;
      std::cin >> l >> r;
      std::cout << seg.query(l, r - 1) << '\n';
    }
  }
```

```
  return 0;
}

```

Luogu P3372【模板】线段树 1[4]

已知一个数列. 需要进行下面两种操作:

- 将某区间每一个数加上 k.
- 求出某区间每一个数的和.

解答

只要实现支持区间修改、区间查询的线段树即可. 注意, 为了实现区间加操作, 需要维护当前区间长度作为区间信息的一部分, 或者在区间修改时利用区间端点计算.

- 实现 1

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

// --8<-- [start:info]
// Interval length and sum of elements.
struct Info {
  long long len, sum;

  Info(long long _len = 0, long long _sum = 0) : len(_len), sum(_sum)
{}

  Info operator+(const Info& o) const { return Info(len + o.len, sum
+ o.sum); }

  Info& operator+=(const Info& o) { return *this = *this + o; }
};

// --8<-- [end:info]
// --8<-- [start:transform]
// Range add.
struct Transform {
  long long v;

  Transform(long long _v = 0) : v(_v) {}

  explicit operator bool() const { return v != 0; }

  Transform operator+(const Transform& o) const { return Transform(v
+ o.v); }

  Transform& operator+=(const Transform& o) { return *this = *this +
o; }
```

```
31
32   Info operator()(const Info& x) const {
33     return Info(x.len, x.sum + v * x.len);
34   }
35 };
36
37 // --8<-- [end:transform]
38 // --8<-- [start:seg-tree]
39 // Segment Tree.
40 // --8<-- [start:build]
41 // Recursive structure, stored in a full binary tree.
42 #define lc(x) ch[(x)][0]
43 #define rc(x) ch[(x)][1]
44
45 int rt, id, L, R;
46 std::vector<std::array<int, 2>> ch;
47 std::vector<Info> val;
48 std::vector<Transform> lazy;
49
50 void push_up(int cr) { val[cr] = val[lc(cr)] + val[rc(cr)]; }
51
52 // Build the tree based on info stored in vec (0-indexed).
53 void build(int cr, int ll, int rr, const std::vector<Info>& vec) {
54   if (ll == rr) return (void)(val[cr] = vec[ll - L]);
55   int mm = ll + ((rr - ll) >> 1);
56   build(lc(cr) = ++id, ll, mm, vec);
57   build(rc(cr) = ++id, mm + 1, rr, vec);
58   push_up(cr);
59 }
60
61 void build(int l, int r, const std::vector<Info>& vec) {
62   rt = 1, id = 0, L = l, R = r;
63   int n = R - L + 1;
64   ch.resize(n << 1), val.resize(n << 1), lazy.resize(n << 1);
65   build(rt = ++id, L, R, vec);
66 }
67
68 // --8<-- [end:build]
69 // --8<-- [start:tag]
70 // Lazy update.
71 void lazy_update(int cr, const Transform& f) {
72   val[cr] = f(val[cr]);
73   lazy[cr] += f;
74 }
75
76 // Push down lazy tag.
77 void push_down(int cr) {
78   if (!lazy[cr]) return;
79   lazy_update(lc(cr), lazy[cr]);
80   lazy_update(rc(cr), lazy[cr]);
81   lazy[cr] = Transform();
82 }
```

```
83
84 // --8<-- [end:tag]
85 // --8<-- [start:range-get]
86 // Query info in [l, r].
87 Info query(int cr, int ll, int rr, int tl, int tr) {
88   if (tl <= ll && rr <= tr) return val[cr];
89   push_down(cr);
90   int mm = ll + ((rr - ll) >> 1);
91   Info res;
92   if (tl <= mm) res = query(lc(cr), ll, mm, tl, tr);
93   if (mm < tr) res += query(rc(cr), mm + 1, rr, tl, tr);
94   return res;
95 }
96
97 Info query(int l, int r) { return query(rt, L, R, l, r); }
98
99 // --8<-- [end:range-get]
100 // --8<-- [start:range-set]
101 // Apply transformation f to the range [l, r].
102 void modify(int cr, int ll, int rr, int tl, int tr, const Transform&
    f) {
103   if (tl <= ll && rr <= tr) return lazy_update(cr, f);
104   push_down(cr);
105   int mm = ll + ((rr - ll) >> 1);
106   if (tl <= mm) modify(lc(cr), ll, mm, tl, tr, f);
107   if (mm < tr) modify(rc(cr), mm + 1, rr, tl, tr, f);
108   push_up(cr);
109 }
110
111 void modify(int l, int r, const Transform& f) { modify(rt, L, R, l,
    r, f); }
112
113 // --8<-- [end:range-set]
114 // --8<-- [end:seg-tree]
115 int main() {
116   std::ios::sync_with_stdio(false), std::cin.tie(nullptr);
117   int n, m;
118   std::cin >> n >> m;
119   std::vector<Info> vec(n);
120   for (auto& v : vec) {
121     long long x;
122     std::cin >> x;
123     v = Info(1, x);
124   }
125   build(1, n, vec);
126   for (; m; --m) {
127     int op;
128     std::cin >> op;
129     if (op == 1) {
130       int x, y;
131       long long k;
132       std::cin >> x >> y >> k;
```

```
      modify(x, y, Transform(k));
    } else if (op == 2) {
      int x, y;
      std::cin >> x >> y;
      std::cout << query(x, y).sum << '\n';
    }
  }
  return 0;
}

```

- 实现 2

```
#include <iostream>
using LL = long long;
LL n, a[100005], d[270000], b[270000];

void build(LL l, LL r, LL p) {  // l:区间左端点 r:区间右端点 p:节点标号
  if (l == r) {
    d[p] = a[l];  // 将节点赋值
    return;
  }
  LL m = l + ((r - l) >> 1);
  build(l, m, p << 1), build(m + 1, r, (p << 1) | 1);  // 分别建立子树
  d[p] = d[p << 1] + d[(p << 1) | 1];
}

void update(LL l, LL r, LL c, LL s, LL t, LL p) {
  if (l <= s && t <= r) {
    d[p] += (t - s + 1) * c, b[p] += c;  // 如果区间被包含了，直接得出答案
    return;
  }
  LL m = s + ((t - s) >> 1);
  if (b[p])
    d[p << 1] += b[p] * (m - s + 1), d[(p << 1) | 1] += b[p] * (t - m),
        b[p << 1] += b[p], b[(p << 1) | 1] += b[p];
  b[p] = 0;
  if (l <= m)
    update(l, r, c, s, m, p << 1);  // 本行和下面的一行用来更新p*2和p*2+1的节点
  if (r > m) update(l, r, c, m + 1, t, (p << 1) | 1);
  d[p] = d[p << 1] + d[(p << 1) | 1];  // 计算该节点区间和
}

LL getsum(LL l, LL r, LL s, LL t, LL p) {
  if (l <= s && t <= r) return d[p];
  LL m = s + ((t - s) >> 1);
  if (b[p])
    d[p << 1] += b[p] * (m - s + 1), d[(p << 1) | 1] += b[p] * (t - m),
        b[p << 1] += b[p], b[(p << 1) | 1] += b[p];
  b[p] = 0;
  LL sum = 0;
  if (l <= m)
```

```
    sum =
        getsum(l, r, s, m, p << 1);  // 本行和下面的一行用来更新p*2和p*2+1的答案
  if (r > m) sum += getsum(l, r, m + 1, t, (p << 1) | 1);
  return sum;
}

int main() {
  std::ios::sync_with_stdio(false);
  LL q, i1, i2, i3, i4;
  std::cin >> n >> q;
  for (LL i = 1; i <= n; i++) std::cin >> a[i];
  build(1, n, 1);
  while (q--) {
    std::cin >> i1 >> i2 >> i3;
    if (i1 == 2)
      std::cout << getsum(i2, i3, 1, n, 1) << std::endl;  // 直接调用操作函数
    else
      std::cin >> i4, update(i2, i3, i4, 1, n, 1);
  }
  return 0;
}

```

Luogu P3373【模板】线段树 2[5]

已知一个数列. 需要进行下面三种操作:

- 将某区间每一个数乘上 x.
- 将某区间每一个数加上 x.
- 求出某区间每一个数的和.

解答

只要实现支持区间修改、区间查询的线段树即可. 可以直接实现区间仿射变换来同时支持区间加、区间乘操作. 如果要使用两个懒惰标记分别记录区间加和区间乘, 需要注意两标记下传的顺序.

- 实现 1

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

int M;

// --8<-- [start:info]
// Interval length and sum of elements.
struct Info {
  int len, sum;

```

```
  Info(int _len = 0, int _sum = 0) : len(_len), sum(_sum) {}

  Info operator+(const Info& o) const {
    return Info(len + o.len, (sum + o.sum) % M);
  }

  Info& operator+=(const Info& o) { return *this = *this + o; }
};

// --8<-- [end:info]
// --8<-- [start:transform]
// Affine transformation.
struct Transform {
  int a, b;

  Transform(int _a = 1, int _b = 0) : a(_a), b(_b) {}

  explicit operator bool() const { return a != 1 || b != 0; }

  Transform operator+(const Transform& o) const {
    return Transform((long long)o.a * a % M, ((long long)o.a * b + o.b)
% M);
  }

  Transform& operator+=(const Transform& o) { return *this = *this +
o; }

  Info operator()(const Info& x) const {
    return Info(x.len, ((long long)a * x.sum + (long long)b * x.len)
% M);
  }
};

// --8<-- [end:transform]
// --8<-- [start:seg-tree]
// Segment Tree.
// --8<-- [start:build]
// Recursive structure, stored in a full binary tree.
#define lc(x) ch[(x)][0]
#define rc(x) ch[(x)][1]

int rt, id, L, R;
std::vector<std::array<int, 2>> ch;
std::vector<Info> val;
std::vector<Transform> lazy;

void push_up(int cr) { val[cr] = val[lc(cr)] + val[rc(cr)]; }

// Build the tree based on info stored in vec (0-indexed).
void build(int cr, int ll, int rr, const std::vector<Info>& vec) {
  if (ll == rr) return (void)(val[cr] = vec[ll - L]);
  int mm = ll + ((rr - ll) >> 1);
```

```
  build(lc(cr) = ++id, ll, mm, vec);
  build(rc(cr) = ++id, mm + 1, rr, vec);
  push_up(cr);
}

void build(int l, int r, const std::vector<Info>& vec) {
  rt = 1, id = 0, L = l, R = r;
  int n = R - L + 1;
  ch.resize(n << 1), val.resize(n << 1), lazy.resize(n << 1);
  build(rt = ++id, L, R, vec);
}

// --8<-- [end:build]
// --8<-- [start:tag]
// Lazy update.
void lazy_update(int cr, const Transform& f) {
  val[cr] = f(val[cr]);
  lazy[cr] += f;
}

// Push down lazy tag.
void push_down(int cr) {
  if (!lazy[cr]) return;
  lazy_update(lc(cr), lazy[cr]);
  lazy_update(rc(cr), lazy[cr]);
  lazy[cr] = Transform();
}

// --8<-- [end:tag]
// --8<-- [start:range-get]
// Query info in [l, r].
Info query(int cr, int ll, int rr, int tl, int tr) {
  if (tl <= ll && rr <= tr) return val[cr];
  push_down(cr);
  int mm = ll + ((rr - ll) >> 1);
  Info res;
  if (tl <= mm) res = query(lc(cr), ll, mm, tl, tr);
  if (mm < tr) res += query(rc(cr), mm + 1, rr, tl, tr);
  return res;
}

Info query(int l, int r) { return query(rt, L, R, l, r); }

// --8<-- [end:range-get]
// --8<-- [start:range-set]
// Apply transformation f to the range [l, r].
void modify(int cr, int ll, int rr, int tl, int tr, const Transform&
f) {
  if (tl <= ll && rr <= tr) return lazy_update(cr, f);
  push_down(cr);
  int mm = ll + ((rr - ll) >> 1);
  if (tl <= mm) modify(lc(cr), ll, mm, tl, tr, f);
```

```
  if (mm < tr) modify(rc(cr), mm + 1, rr, tl, tr, f);
  push_up(cr);
}

void modify(int l, int r, const Transform& f) { modify(rt, L, R, l,
r, f); }

// --8<-- [end:range-set]
// --8<-- [end:seg-tree]
int main() {
  std::ios::sync_with_stdio(false), std::cin.tie(nullptr);
  int n, q;
  std::cin >> n >> q >> M;
  std::vector<Info> vec(n);
  for (auto& v : vec) {
    long long x;
    std::cin >> x;
    v = Info(1, x);
  }
  build(1, n, vec);
  for (; q; --q) {
    int op;
    std::cin >> op;
    if (op == 1) {
      int x, y, k;
      std::cin >> x >> y >> k;
      modify(x, y, Transform(k, 0));
    } else if (op == 2) {
      int x, y, k;
      std::cin >> x >> y >> k;
      modify(x, y, Transform(1, k));
    } else if (op == 3) {
      int x, y;
      std::cin >> x >> y;
      std::cout << query(x, y).sum << '\n';
    }
  }
  return 0;
}

```

- 实现 2

```
#include <iostream>
using ll = long long;

int n, m;
ll mod;
ll a[100005], sum[400005], mul[400005], laz[400005];

void up(int i) { sum[i] = (sum[(i << 1)] + sum[(i << 1) | 1]) % mod; }

void pd(int i, int s, int t) {
```

```
  int l = (i << 1), r = (i << 1) | 1, mid = (s + t) >> 1;
  if (mul[i] != 1) {  // 懒标记传递，两个懒标记
    mul[l] *= mul[i];
    mul[l] %= mod;
    mul[r] *= mul[i];
    mul[r] %= mod;
    laz[l] *= mul[i];
    laz[l] %= mod;
    laz[r] *= mul[i];
    laz[r] %= mod;
    sum[l] *= mul[i];
    sum[l] %= mod;
    sum[r] *= mul[i];
    sum[r] %= mod;
    mul[i] = 1;
  }
  if (laz[i]) {  // 懒标记传递
    sum[l] += laz[i] * (mid - s + 1);
    sum[l] %= mod;
    sum[r] += laz[i] * (t - mid);
    sum[r] %= mod;
    laz[l] += laz[i];
    laz[l] %= mod;
    laz[r] += laz[i];
    laz[r] %= mod;
    laz[i] = 0;
  }
  return;
}

void build(int s, int t, int i) {
  mul[i] = 1;
  if (s == t) {
    sum[i] = a[s];
    return;
  }
  int mid = s + ((t - s) >> 1);
  build(s, mid, i << 1);
  build(mid + 1, t, (i << 1) | 1);
  up(i);
}

void chen(int l, int r, int s, int t, int i, ll z) {
  int mid = s + ((t - s) >> 1);
  if (l <= s && t <= r) {
    mul[i] *= z;
    mul[i] %= mod;
    laz[i] *= z;
    laz[i] %= mod;
    sum[i] *= z;
    sum[i] %= mod;
    return;
```

```
  }
  pd(i, s, t);
  if (mid >= l) chen(l, r, s, mid, (i << 1), z);
  if (mid + 1 <= r) chen(l, r, mid + 1, t, (i << 1) | 1, z);
  up(i);
}

void add(int l, int r, int s, int t, int i, ll z) {
  int mid = s + ((t - s) >> 1);
  if (l <= s && t <= r) {
    sum[i] += z * (t - s + 1);
    sum[i] %= mod;
    laz[i] += z;
    laz[i] %= mod;
    return;
  }
  pd(i, s, t);
  if (mid >= l) add(l, r, s, mid, (i << 1), z);
  if (mid + 1 <= r) add(l, r, mid + 1, t, (i << 1) | 1, z);
  up(i);
}

ll getans(int l, int r, int s, int t,
          int i) {  // 得到答案，可以看下上面懒标记助于理解
  int mid = s + ((t - s) >> 1);
  ll tot = 0;
  if (l <= s && t <= r) return sum[i];
  pd(i, s, t);
  if (mid >= l) tot += getans(l, r, s, mid, (i << 1));
  tot %= mod;
  if (mid + 1 <= r) tot += getans(l, r, mid + 1, t, (i << 1) | 1);
  return tot % mod;
}

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

int main() {  // 读入
  cin.tie(nullptr)->sync_with_stdio(false);
  int i, j, x, y, bh;
  ll z;
  cin >> n >> m >> mod;
  for (i = 1; i <= n; i++) cin >> a[i];
  build(1, n, 1);  // 建树
  for (i = 1; i <= m; i++) {
    cin >> bh;
    if (bh == 1) {
      cin >> x >> y >> z;
      chen(x, y, 1, n, 1, z);
    } else if (bh == 2) {
      cin >> x >> y >> z;
      add(x, y, 1, n, 1, z);
```

```
    } else if (bh == 3) {
      cin >> x >> y;
      cout << getans(x, y, 1, n, 1) << '\n';
    }
  }
  return 0;
}

```

SPOJ GSS3 - Can you answer these queries III[6]

已知一个数列．需要进行如下操作：

- 将第 x 个数设置为 y.
- 求区间 $[x, y]$ 的最大非空子段和.

解答

只要实现支持单点修改、区间查询的线段树即可．前文→10@p1977 已经分析过维护最大非空子段和的方法．这道题目中，信息合并不再满足交换律，需要注意实现时的合并顺序.

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

// --8<-- [start:info]
// Maximum (non-empty) subarray sum
struct Info {
  int sum;  // sum of the whole interval
  int lma;  // maximum sum of a non-empty prefix
  int rma;  // maximum sum of a non-empty suffix
  int ma;   // maximum sum of a non-empty subarray

  Info() : sum(0), lma(-1e9), rma(-1e9), ma(-1e9) {}

  Info(int x) : sum(x), lma(x), rma(x), ma(x) {}

  Info(int _s, int _l, int _r, int _m) : sum(_s), lma(_l), rma(_r), ma(_m)
{}

  Info operator+(const Info& o) const {
    return Info(sum + o.sum, std::max(lma, sum + o.lma),
                std::max(rma + o.sum, o.rma),
                std::max(std::max(ma, o.ma), rma + o.lma));
  }

  Info& operator+=(const Info& o) { return *this = *this + o; }
};

// --8<-- [end:info]
// --8<-- [start:seg-tree]
// Segment tree.
```

```cpp
class SegmentTree {
  int rt, id, L, R;
  std::vector<int> lc, rc;
  std::vector<Info> val;

  void push_up(int cr) { val[cr] = val[lc[cr]] + val[rc[cr]]; }

  void build(int cr, int ll, int rr, const std::vector<int>& vec) {
    if (ll == rr) return (void)(val[cr] = Info(vec[ll - L]));
    int mm = ll + ((rr - ll) >> 1);
    build(lc[cr] = ++id, ll, mm, vec);
    build(rc[cr] = ++id, mm + 1, rr, vec);
    push_up(cr);
  }

  void modify(int cr, int ll, int rr, int x, int v) {
    if (ll == rr) return (void)(val[cr] = Info(v));
    int mm = ll + ((rr - ll) >> 1);
    if (x <= mm)
      modify(lc[cr], ll, mm, x, v);
    else
      modify(rc[cr], mm + 1, rr, x, v);
    push_up(cr);
  }

  Info query(int cr, int ll, int rr, int tl, int tr) {
    if (tl <= ll && rr <= tr) return val[cr];
    int mm = ll + ((rr - ll) >> 1);
    Info res;
    if (tl <= mm) res = query(lc[cr], ll, mm, tl, tr);
    if (mm < tr) res += query(rc[cr], mm + 1, rr, tl, tr);
    return res;
  }

 public:
  SegmentTree(int n, const std::vector<int>& vec)
      : rt(0), id(0), L(1), R(n), lc(n << 1), rc(n << 1), val(n << 1) {
    build(rt = ++id, L, R, vec);
  }

  void modify(int x, int v) { modify(rt, L, R, x, v); }

  Info query(int l, int r) { return query(rt, L, R, l, r); }
};

// --8<-- [end:seg-tree]
int main() {
  std::ios::sync_with_stdio(false), std::cin.tie(nullptr);
  int n;
  std::cin >> n;
  std::vector<int> vec(n);
  for (auto& x : vec) std::cin >> x;
```

```
  SegmentTree seg(n, vec);
  int m;
  std::cin >> m;
  for (; m; --m) {
    int op;
    std::cin >> op;
    if (op == 0) {
      int x, y;
      std::cin >> x >> y;
      seg.modify(x, y);
    } else if (op == 1) {
      int l, r;
      std::cin >> l >> r;
      std::cout << seg.query(l, r).ma << '\n';
    }
  }
  return 0;
}

```

Luogu P13825【模板】线段树 1.5[7]

设长度为 $n \le 10^9$ 的序列 $\{a_i\}$ 初始为 $a_i = i$. 需要进行如下操作:

- 将某区间每一个数加上 k.
- 求出某区间每一个数的和.

解答

由于 n 过大，不能直接建树，需要动态开点. 又因为尚未修改过的区间信息并非单位元，空结点的信息需要由其区间端点算出，创建新结点时也要据此初始化. 这也意味着下传标记、合并子结点信息时，都需要传入当前区间的端点.

就本题而言，其实还可以把初始值的贡献单独算出，线段树只维护各元素的增量. 此时空结点的增量之和为 0，恰是单位元，创建结点和合并信息都无需再考虑区间端点；但施加区间加时仍需知道区间长度，因此它只能由递归参数提供，不能并入区间信息.

由于本题没有强制在线，也可以先把所有操作存下来，将端点离散化，再用普通线段树处理. 离散化后，每个叶结点对应原序列的一段连续区间，其信息需要按该段的长度和元素之和初始化.

- 实现 1

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

using u64 = unsigned long long;

// --8<-- [start:info]
// Interval length and sum of elements.
struct Info {
  u64 len, sum;
```

```
12
13    Info(u64 _len = 0, u64 _sum = 0) : len(_len), sum(_sum) {}
14
15    Info operator+(const Info& o) const { return Info(len + o.len, sum
    + o.sum); }
16
17    Info& operator+=(const Info& o) { return *this = *this + o; }
18  };
19
20  // --8<-- [end:info]
21  // --8<-- [start:transform]
22  // Range add.
23  struct Transform {
24    u64 a;
25
26    Transform(u64 _a = 0) : a(_a) {}
27
28    explicit operator bool() const { return a != 0; }
29
30    Transform operator+(const Transform& o) const { return Transform(a
    + o.a); }
31
32    Transform& operator+=(const Transform& o) { return *this = *this +
    o; }
33
34    Info operator()(const Info& v) const {
35      return Info(v.len, v.sum + a * v.len);
36    }
37  };
38
39  // --8<-- [end:transform]
40  // --8<-- [start:seg-tree]
41  // Segment Tree Implementation. Dynamic node allocation.
42  // --8<-- [start:build]
43  // Recursive structure, stored in a full binary tree.
44  #define lc(x) ch[(x)][0]
45  #define rc(x) ch[(x)][1]
46
47  int rt, id, L, R;
48  std::vector<std::array<int, 2>> ch;
49  std::vector<Info> val;
50  std::vector<Transform> lazy;
51
52  Info blank(int ll, int rr) {
53    return Info(rr - ll + 1, rr * (rr + 1ULL) / 2 - ll * (ll - 1ULL) /
    2);
54  }
55
56  Info info_of(int cr, int ll, int rr) { return cr ? val[cr] : blank(ll,
    rr); }
57
58  void push_up(int cr, int ll, int rr) {
```

```cpp
  int mm = ll + ((rr - ll) >> 1);
  val[cr] = info_of(lc(cr), ll, mm) + info_of(rc(cr), mm + 1, rr);
}

// Build an empty tree.
void build(int l, int r, int n) {
  rt = 0, id = 0, L = l, R = r;
  ch.resize(n), val.resize(n), lazy.resize(n);
}

// --8<-- [end:build]
// --8<-- [start:tag]
// Lazy update.
void lazy_update(int& cr, int ll, int rr, const Transform& f) {
  if (!cr) val[cr = ++id] = blank(ll, rr);
  val[cr] = f(val[cr]);
  lazy[cr] += f;
}

// Push down lazy tag.
void push_down(int cr, int ll, int rr) {
  if (!lazy[cr]) return;
  int mm = ll + ((rr - ll) >> 1);
  lazy_update(lc(cr), ll, mm, lazy[cr]);
  lazy_update(rc(cr), mm + 1, rr, lazy[cr]);
  lazy[cr] = Transform();
}

// --8<-- [end:tag]
// --8<-- [start:range-get]
// Query info in [l, r].
Info query(int cr, int ll, int rr, int tl, int tr) {
  if (!cr) return blank(std::max(ll, tl), std::min(rr, tr));
  if (tl <= ll && rr <= tr) return val[cr];
  push_down(cr, ll, rr);
  int mm = ll + ((rr - ll) >> 1);
  Info res;
  if (tl <= mm) res = query(lc(cr), ll, mm, tl, tr);
  if (mm < tr) res += query(rc(cr), mm + 1, rr, tl, tr);
  return res;
}

Info query(int l, int r) { return query(rt, L, R, l, r); }

// --8<-- [end:range-get]
// --8<-- [start:range-set]
// Apply transformation f to the range [l, r].
void modify(int& cr, int ll, int rr, int tl, int tr, const Transform&
f) {
  if (!cr) val[cr = ++id] = blank(ll, rr);
  if (tl <= ll && rr <= tr) return lazy_update(cr, ll, rr, f);
  push_down(cr, ll, rr);
```

```
  int mm = ll + ((rr - ll) >> 1);
  if (tl <= mm) modify(lc(cr), ll, mm, tl, tr, f);
  if (mm < tr) modify(rc(cr), mm + 1, rr, tl, tr, f);
  push_up(cr, ll, rr);
}

void modify(int l, int r, const Transform& f) { modify(rt, L, R, l, r, f); }

// --8<-- [end:range-set]
// --8<-- [end:seg-tree]
int main() {
  std::ios::sync_with_stdio(false), std::cin.tie(nullptr);
  int n, m;
  std::cin >> n >> m;
  build(1, n, 1.5e7);
  for (; m; --m) {
    int op;
    std::cin >> op;
    if (op == 1) {
      int l, r, k;
      std::cin >> l >> r >> k;
      modify(l, r, Transform(k));
    } else if (op == 2) {
      int l, r;
      std::cin >> l >> r;
      std::cout << query(l, r).sum << '\n';
    }
  }
  return 0;
}

```

- 实现 2

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

class SegmentTree {
  static constexpr int N = 2e7;
  int rt, id, L, R;
  std::vector<int> lc, rc;
  std::vector<unsigned long long> sum;
  std::vector<unsigned long long> lz;

  void lazy_update(int& cr, int ll, int rr, unsigned long long v) {
    if (!cr) cr = ++id;
    sum[cr] += v * (rr - ll + 1);
    lz[cr] += v;
  }

  void push_down(int cr, int ll, int rr) {
```

```
    if (!lz[cr]) return;
    int mm = (ll + rr) >> 1;
    lazy_update(lc[cr], ll, mm, lz[cr]);
    lazy_update(rc[cr], mm + 1, rr, lz[cr]);
    lz[cr] = 0;
  }

  void add(int& cr, int ll, int rr, int tl, int tr, int k) {
    if (!cr) cr = ++id;
    if (tl <= ll && rr <= tr) return lazy_update(cr, ll, rr, k);
    push_down(cr, ll, rr);
    int mm = (ll + rr) / 2;
    if (tl <= mm) add(lc[cr], ll, mm, tl, tr, k);
    if (mm < tr) add(rc[cr], mm + 1, rr, tl, tr, k);
    sum[cr] = sum[lc[cr]] + sum[rc[cr]];
  }

  unsigned long long query(int cr, int ll, int rr, int tl, int tr) {
    if (!cr) return 0;
    if (tl <= ll && rr <= tr) return sum[cr];
    push_down(cr, ll, rr);
    int mm = (ll + rr) / 2;
    unsigned long long res = 0;
    if (tl <= mm) res = query(lc[cr], ll, mm, tl, tr);
    if (mm < tr) res += query(rc[cr], mm + 1, rr, tl, tr);
    return res;
  }

 public:
  SegmentTree(int n) : rt(0), id(0), L(1), R(n), lc(N), rc(N), sum(N),
lz(N) {}

  void add(int l, int r, int k) { add(rt, L, R, l, r, k); }

  unsigned long long query(int l, int r) { return query(rt, L, R,
l, r); }
};

int main() {
  std::ios::sync_with_stdio(false), std::cin.tie(nullptr);
  int n, m;
  std::cin >> n >> m;
  SegmentTree seg(n);
  for (; m; --m) {
    int op;
    std::cin >> op;
    if (op == 1) {
      int l, r, k;
      std::cin >> l >> r >> k;
      seg.add(l, r, k);
    } else if (op == 2) {
      int l, r;
```

```
      std::cin >> l >> r;
      auto res = (r + 1ULL) * r / 2 - (l - 1ULL) * l / 2;
      std::cout << (res + seg.query(l, r)) << '\n';
    }
  }
  return 0;
}

```

- 实现 3

```
#include <algorithm>
#include <array>
#include <iostream>
#include <unordered_map>
#include <vector>

using u64 = unsigned long long;

// --8<-- [start:info]
// Interval length and sum of elements.
struct Info {
  u64 len, sum;

  Info(u64 _len = 0, u64 _sum = 0) : len(_len), sum(_sum) {}

  Info operator+(const Info& o) const { return Info(len + o.len, sum + o.sum); }

  Info& operator+=(const Info& o) { return *this = *this + o; }
};

// --8<-- [end:info]
// --8<-- [start:transform]
// Range add.
struct Transform {
  u64 a;

  Transform(u64 _a = 0) : a(_a) {}

  explicit operator bool() const { return a != 0; }

  Transform operator+(const Transform& o) const { return Transform(a + o.a); }

  Transform& operator+=(const Transform& o) { return *this = *this + o; }

  Info operator()(const Info& v) const {
    return Info(v.len, v.sum + a * v.len);
  }
};

```

```cpp
// --8<-- [end:transform]
// --8<-- [start:seg-tree]
// Segment Tree.
// --8<-- [start:build]
// Recursive structure, stored in a full binary tree.
#define lc(x) ch[(x)][0]
#define rc(x) ch[(x)][1]

int rt, id, L, R;
std::vector<std::array<int, 2>> ch;
std::vector<Info> val;
std::vector<Transform> lazy;

void push_up(int cr) { val[cr] = val[lc(cr)] + val[rc(cr)]; }

// Build the tree based on info stored in vec (0-indexed).
void build(int cr, int ll, int rr, const std::vector<Info>& vec) {
  if (ll == rr) return (void)(val[cr] = vec[ll - L]);
  int mm = ll + ((rr - ll) >> 1);
  build(lc(cr) = ++id, ll, mm, vec);
  build(rc(cr) = ++id, mm + 1, rr, vec);
  push_up(cr);
}

void build(int l, int r, const std::vector<Info>& vec) {
  rt = 1, id = 0, L = l, R = r;
  int n = R - L + 1;
  ch.resize(n << 1), val.resize(n << 1), lazy.resize(n << 1);
  build(rt = ++id, L, R, vec);
}

// --8<-- [end:build]
// --8<-- [start:tag]
// Lazy update.
void lazy_update(int cr, const Transform& f) {
  val[cr] = f(val[cr]);
  lazy[cr] += f;
}

// Push down lazy tag.
void push_down(int cr) {
  if (!lazy[cr]) return;
  lazy_update(lc(cr), lazy[cr]);
  lazy_update(rc(cr), lazy[cr]);
  lazy[cr] = Transform();
}

// --8<-- [end:tag]
// --8<-- [start:range-get]
// Query info in [l, r].
Info query(int cr, int ll, int rr, int tl, int tr) {
  if (tl <= ll && rr <= tr) return val[cr];
```

```
  push_down(cr);
  int mm = ll + ((rr - ll) >> 1);
  Info res;
  if (tl <= mm) res = query(lc(cr), ll, mm, tl, tr);
  if (mm < tr) res += query(rc(cr), mm + 1, rr, tl, tr);
  return res;
}

Info query(int l, int r) { return query(rt, L, R, l, r); }

// --8<-- [end:range-get]
// --8<-- [start:range-set]
// Apply transformation f to the range [l, r].
void modify(int cr, int ll, int rr, int tl, int tr, const Transform&
f) {
  if (tl <= ll && rr <= tr) return lazy_update(cr, f);
  push_down(cr);
  int mm = ll + ((rr - ll) >> 1);
  if (tl <= mm) modify(lc(cr), ll, mm, tl, tr, f);
  if (mm < tr) modify(rc(cr), mm + 1, rr, tl, tr, f);
  push_up(cr);
}

void modify(int l, int r, const Transform& f) { modify(rt, L, R, l,
r, f); }

// --8<-- [end:range-set]
// --8<-- [end:seg-tree]
int main() {
  std::ios::sync_with_stdio(false), std::cin.tie(nullptr);
  int n, m;
  std::cin >> n >> m;
  // Offline queries and discretization.
  std::vector<std::array<int, 4>> queries(m);
  std::vector<int> loc;
  loc.reserve((m << 1) | 1);
  loc.push_back(0);
  for (auto& q : queries) {
    std::cin >> q[0] >> q[1] >> q[2];
    if (q[0] == 1) std::cin >> q[3];
    loc.push_back(q[1] - 1);
    loc.push_back(q[2]);
  }
  std::sort(loc.begin(), loc.end());
  loc.erase(std::unique(loc.begin(), loc.end()), loc.end());
  int sz = loc.size();
  std::unordered_map<int, int> ids;
  for (int i = 0; i < sz; ++i) ids[loc[i]] = i;
  std::vector<Info> vec(sz);
  for (int i = 1; i < sz; ++i) {
    vec[i].len = loc[i] - loc[i - 1];
    vec[i].sum = vec[i].len * (loc[i] + loc[i - 1] + 1ULL) / 2;
```

```
  }
  // Ordinary seg tree operations.
  build(0, sz - 1, vec);
  for (auto q : queries) {
    if (q[0] == 1) {
      modify(ids[q[1] - 1] + 1, ids[q[2]], Transform(q[3]));
    } else if (q[0] == 2) {
      std::cout << query(ids[q[1] - 1] + 1, ids[q[2]]).sum << '\n';
    }
  }
  return 0;
}

```

Luogu P5490【模板】扫描线 & 矩形面积并[8]

求 n 个四边平行于坐标轴的矩形的面积并.

解答

这是一道 扫描线→12.8@p3125 模板题. 利用扫描线转化问题的细节请参考所引页面, 此处只讨论转化后的数据结构. 转化后需要维护的是一个支持区间加减覆盖层数的结构: 每次操作给某段区间的覆盖层数加一或减一, 每次查询覆盖层数至少为一的位置的总长度.

考虑利用线段树实现这些区间操作. 一种自然的想法是, 每个结点处维护该区间的覆盖层数和覆盖层数至少为一的总长度. 但是, 区间一旦整体覆盖, 子区间各自的覆盖情况就从总长度中丢失了, 之后撤去这层覆盖便无从恢复. 解决办法是把两类操作分开记录: 覆盖层数只计入恰好整体覆盖该区间的操作, 总长度只统计记录在子孙结点上的覆盖长度. 于是, 该区间实际覆盖到的长度就是: 覆盖层数大于零时为区间长度, 否则为记录的总长度. 合并时, 把两个子结点各自算出的实际长度相加, 即为本结点的总长度. 这样一来, 覆盖层数就不能下传了, 因为每次覆盖的加入与移除落在同一组结点处, 只有让层数留在原处才能相互抵消. 这正是标记永久化.

当然, 本题也可以用前文的线段树模板实现. 只需将区间信息设为该区间内的最小覆盖层数, 以及取得该最小值的部分的总长度. 容易验证, 这样的信息空间是幺半群, 而加减覆盖层数是其上的自同态, 于是区间修改和区间查询都可以照搬模板. 要得到覆盖到的总长度, 用整段区间的长度减去覆盖层数为零的部分的长度即可; 而后者就是最小覆盖层数为零时所记录的长度, 否则为零.

- 实现 1

```
#include <algorithm>
#include <array>
#include <iostream>
#include <unordered_map>
#include <vector>

struct Info {
  int len;  // Length of this interval.
  int cnt;  // Number of operations that cover this interval exactly.
  int tot;  // Length covered by operations recorded in the
descendants.
};
```

```cpp

// Segment tree.
int rt, id, L, R;
std::vector<int> lc, rc;
std::vector<Info> val;

// Build the tree.
void build(int cr, int ll, int rr, const std::vector<int>& vec) {
  if (ll == rr) return (void)(val[cr].len = (ll ? vec[ll] - vec[ll - 1] : 0));
  int mm = ll + ((rr - ll) >> 1);
  build(lc[cr] = ++id, ll, mm, vec);
  build(rc[cr] = ++id, mm + 1, rr, vec);
  val[cr].len = val[lc[cr]].len + val[rc[cr]].len;
}

void build(int n, const std::vector<int>& vec) {
  rt = 0, id = 0, L = 0, R = n - 1;
  lc.resize(n << 1), rc.resize(n << 1), val.resize(n << 1);
  build(rt = ++id, 0, n - 1, vec);
}

// Query.
int query(int cr) { return val[cr].cnt ? val[cr].len : val[cr].tot; }

int query() { return query(rt); }

// Cover.
void cover(int cr, int ll, int rr, int tl, int tr, int v) {
  if (tl <= ll && rr <= tr) return (void)(val[cr].cnt += v);
  int mm = ll + ((rr - ll) >> 1);
  if (tl <= mm) cover(lc[cr], ll, mm, tl, tr, v);
  if (mm < tr) cover(rc[cr], mm + 1, rr, tl, tr, v);
  val[cr].tot = query(lc[cr]) + query(rc[cr]);
}

void cover(int l, int r, int v) { cover(rt, L, R, l, r, v); }

int main() {
  std::ios::sync_with_stdio(false), std::cin.tie(nullptr);
  int n;
  std::cin >> n;
  // Scanning and discretizing.
  std::vector<std::array<int, 4>> ops;
  ops.reserve(n << 1);
  std::vector<int> locs;
  locs.reserve(n << 1);
  for (int i = 0; i < n; ++i) {
    int x1, y1, x2, y2;
    std::cin >> x1 >> y1 >> x2 >> y2;
    ops.push_back({x1, y1, y2, 1});
    ops.push_back({x2, y1, y2, -1});
```

```
    locs.push_back(y1);
    locs.push_back(y2);
  }
  std::sort(ops.begin(), ops.end());
  std::sort(locs.begin(), locs.end());
  locs.erase(std::unique(locs.begin(), locs.end()), locs.end());
  int m = locs.size();
  std::unordered_map<int, int> ids;
  for (int i = 0; i < m; ++i) ids[locs[i]] = i;
  // Segment tree operations.
  build(m, locs);
  long long res = 0;
  for (int l = 0, r; l < (n << 1); l = r) {
    for (r = l; r < (n << 1) && ops[r][0] == ops[l][0]; ++r) {
      cover(ids[ops[r][1]] + 1, ids[ops[r][2]], ops[r][3]);
    }
    if (r < (n << 1)) res += (long long)(ops[r][0] - ops[l][0]) *
query();
  }
  std::cout << res << std::endl;
  return 0;
}

```

- 实现 2

```
#include <algorithm>
#include <array>
#include <climits>
#include <iostream>
#include <unordered_map>
#include <vector>

struct Info {
  int cnt;  // Minimum cover count over this interval.
  int len;  // Total length of the parts attaining that minimum.

  Info(int _cnt = INT_MAX, int _len = 0) : cnt(_cnt), len(_len) {}

  Info operator+(const Info& o) const {
    return cnt < o.cnt ? *this : (cnt > o.cnt ? o : Info(cnt, len
+ o.len));
  }

  Info& operator+=(const Info& o) { return *this = *this + o; }
};

struct Transform {
  int v;

  Transform(int _v = 0) : v(_v) {}

  explicit operator bool() const { return v != 0; }
```

```
27
28   Transform operator+(const Transform& o) const { return Transform(v
+ o.v); }
29
30   Transform& operator+=(const Transform& o) { return *this = *this +
o; }
31
32   Info operator()(const Info& x) const {
33     return Info(x.cnt == INT_MAX ? INT_MAX : x.cnt + v, x.len);
34   }
35 };
36
37 class SegmentTree {
38   int rt, id, L, R;
39   std::vector<int> lc, rc;
40   std::vector<Info> val;
41   std::vector<Transform> lz;
42
43   void push_up(int cr) { val[cr] = val[lc[cr]] + val[rc[cr]]; }
44
45   void lazy_update(int cr, const Transform& f) {
46     val[cr] = f(val[cr]);
47     lz[cr] += f;
48   }
49
50   void push_down(int cr) {
51     if (!lz[cr]) return;
52     lazy_update(lc[cr], lz[cr]);
53     lazy_update(rc[cr], lz[cr]);
54     lz[cr] = Transform();
55   }
56
57   void build(int cr, int ll, int rr, const std::vector<int>& locs) {
58     if (ll == rr)
59       return (void)(val[cr] = Info(0, ll ? locs[ll] - locs[ll - 1] :
0));
60     int mm = ll + ((rr - ll) >> 1);
61     build(lc[cr] = ++id, ll, mm, locs);
62     build(rc[cr] = ++id, mm + 1, rr, locs);
63     push_up(cr);
64   }
65
66   void modify(int cr, int ll, int rr, int tl, int tr, const Transform&
f) {
67     if (tl <= ll && rr <= tr) return lazy_update(cr, f);
68     push_down(cr);
69     int mm = ll + ((rr - ll) >> 1);
70     if (tl <= mm) modify(lc[cr], ll, mm, tl, tr, f);
71     if (mm < tr) modify(rc[cr], mm + 1, rr, tl, tr, f);
72     push_up(cr);
73   }
74
```

```
 public:
  SegmentTree(int n, const std::vector<int>& locs) {
    rt = 0, id = 0, L = 0, R = n - 1;
    lc.resize(n << 1), rc.resize(n << 1), val.resize(n << 1),
lz.resize(n << 1);
    build(rt = ++id, L, R, locs);
  }

  void modify(int l, int r, const Transform& f) { modify(rt, L, R, l,
r, f); }

  Info query() const { return val[rt]; }
};

int main() {
  std::ios::sync_with_stdio(false), std::cin.tie(nullptr);
  int n;
  std::cin >> n;
  // Scanning and discretizing.
  std::vector<std::array<int, 4>> ops;
  ops.reserve(n << 1);
  std::vector<int> locs;
  locs.reserve(n << 1);
  for (int i = 0; i < n; ++i) {
    int x1, y1, x2, y2;
    std::cin >> x1 >> y1 >> x2 >> y2;
    ops.push_back({x1, y1, y2, 1});
    ops.push_back({x2, y1, y2, -1});
    locs.push_back(y1);
    locs.push_back(y2);
  }
  std::sort(ops.begin(), ops.end());
  std::sort(locs.begin(), locs.end());
  locs.erase(std::unique(locs.begin(), locs.end()), locs.end());
  int m = locs.size();
  std::unordered_map<int, int> ids;
  for (int i = 0; i < m; ++i) ids[locs[i]] = i;
  // Segment tree operations.
  long long all = locs.back() - locs[0];
  SegmentTree seg(m, locs);
  long long res = 0;
  for (int l = 0, r; l < (n << 1); l = r) {
    for (r = l; r < (n << 1) && ops[r][0] == ops[l][0]; ++r) {
      seg.modify(ids[ops[r][1]] + 1, ids[ops[r][2]], Transform(ops[r]
[3]));
    }
    if (r < (n << 1)) res += (ops[r][0] - ops[l][0]) * (all -
seg.query().len);
  }
  std::cout << res << std::endl;
  return 0;
}
```

```
123
```

Luogu P2894【USACO08FEB】Hotel G[9]

有 n 个房间，初始都是空房．需要进行如下操作：

- 找到长度为 x 的连续空房．如果存在，让它们住上人．
- 房间 $[x, x+y-1]$ 退房．

每次进行第一个操作时，输出连续 x 个空房中最小的房间号；若不存在，返回 0.

解答

考虑构造线段树，维护每个区间中连续空房的最大长度．为了合并这一信息，还需维护最长空房前缀、最长空房后缀和区间长度：跨越中点的连续空房，恰是左子区间的最长后缀接上右子区间的最长前缀．懒惰标记只需实现区间赋值，即整段置为空或置为已住；不修改的情形记录为特殊值（例如 -1）．要找最小的可用房号，只需查询最小的 r，使得 $[1, r]$ 中连续空房的最大长度不小于 x，房号即 $r-x+1$．为此，利用线段树二分实现即可．

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

// Interval info.
// lt - longest prefix run.
// rt - longest suffix run.
// ma - longest run.
// len - interval length.
struct Info {
  int lt, rt, ma, len;

  Info(int _lt = 0, int _rt = 0, int _ma = 0, int _len = 0)
      : lt(_lt), rt(_rt), ma(_ma), len(_len) {}

  Info operator+(const Info& o) const {
    auto _lt = lt == len ? len + o.lt : lt;
    auto _rt = o.rt == o.len ? rt + o.len : o.rt;
    return Info(_lt, _rt, std::max({_lt, _rt, rt + o.lt, ma, o.ma}),
                len + o.len);
  }

  Info& operator+=(const Info& o) { return *this = *this + o; }
};

// Assignment.
struct Transform {
  int v;

  Transform(int _v = -1) : v(_v) {}

  explicit operator bool() const { return v != -1; }

```

```
  Transform operator+(const Transform& o) const { return o ? o : *this; }

  Transform& operator+=(const Transform& o) { return *this = *this + o; }

  Info operator()(const Info& x) const {
    return v == 1 ? Info(0, 0, 0, x.len)
                  : (v == 0 ? Info(x.len, x.len, x.len, x.len) : x);
  }
};

// Segment tree.
class SegmentTree {
  int rt, id, L, R;
  std::vector<int> lc, rc;
  std::vector<Info> val;
  std::vector<Transform> lz;

  void push_up(int cr) { val[cr] = val[lc[cr]] + val[rc[cr]]; }

  void lazy_update(int cr, const Transform& f) {
    val[cr] = f(val[cr]);
    lz[cr] += f;
  }

  void push_down(int cr) {
    if (!lz[cr]) return;
    lazy_update(lc[cr], lz[cr]);
    lazy_update(rc[cr], lz[cr]);
    lz[cr] = Transform();
  }

  void build(int cr, int ll, int rr) {
    if (ll == rr) return (void)(val[cr] = Info(1, 1, 1, 1));
    int mm = ll + ((rr - ll) >> 1);
    build(lc[cr] = ++id, ll, mm);
    build(rc[cr] = ++id, mm + 1, rr);
    push_up(cr);
  }

  void modify(int cr, int ll, int rr, int tl, int tr, const Transform& f) {
    if (tl <= ll && rr <= tr) return lazy_update(cr, f);
    push_down(cr);
    int mm = ll + ((rr - ll) >> 1);
    if (tl <= mm) modify(lc[cr], ll, mm, tl, tr, f);
    if (mm < tr) modify(rc[cr], mm + 1, rr, tl, tr, f);
    push_up(cr);
  }

  template <typename G>
  int lower_bound(int cr, int ll, int rr, const G& g, const Info& acc) {
    if (ll == rr) return ll;
    push_down(cr);
```

```
    int mm = ll + ((rr - ll) >> 1);
    return g(acc + val[lc[cr]])
               ? lower_bound(lc[cr], ll, mm, g, acc)
               : lower_bound(rc[cr], mm + 1, rr, g, acc + val[lc[cr]]);
  }

 public:
  SegmentTree(int n) : rt(0), id(0), L(1), R(n) {
    lc.resize(n << 1), rc.resize(n << 1), val.resize(n << 1), lz.resize(n << 1);
    build(rt = ++id, L, R);
  }

  // Apply f to [l,r].
  void modify(int l, int r, const Transform& f) { modify(rt, L, R, l, r, f); }

  // Find the lowest r such that g(info([1,r])) is true.
  template <typename G>
  int lower_bound(const G& g) {
    return g(val[rt]) ? lower_bound(rt, L, R, g, Info()) : R + 1;
  }
};

int main() {
  int n, m;
  std::cin >> n >> m;
  SegmentTree seg(n);
  for (; m; --m) {
    int op;
    std::cin >> op;
    if (op == 1) {
      int x;
      std::cin >> x;
      auto y =
          seg.lower_bound([&](const Info& v) -> bool { return v.ma >= x; });
      if (y <= n) {
        std::cout << (y - x + 1) << '\n';
        seg.modify(y - x + 1, y, Transform(1));
      } else {
        std::cout << 0 << '\n';
      }
    } else if (op == 2) {
      int x, y;
      std::cin >> x >> y;
      seg.modify(x, x + y - 1, Transform(0));
    }
  }
  return 0;
}

```

Luogu P1168 中位数[10]

给定长度为 n 的数列．对于每一个奇数 $i \le n$，输出前 i 个数的中位数．

解答

中位数只与元素的大小关系有关，与它们在数列中的位置无关，因此可以用权值线段树维护．依次插入各个元素，每插入奇数个后查询第 $(i+1)/2$ 小的元素即可．只需要实现插入和查询第 k 小两种操作．

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

class SegmentTree {
  int rt, id, L, R;
  std::vector<int> lc, rc;
  std::vector<int> cnt;

  void insert(int& cr, int ll, int rr, int v) {
    if (!cr) cr = ++id;
    ++cnt[cr];
    if (ll == rr) return;
    int mm = ll + ((rr - ll) >> 1);
    if (v <= mm)
      insert(lc[cr], ll, mm, v);
    else
      insert(rc[cr], mm + 1, rr, v);
  }

  int find_kth(int cr, int ll, int rr, int k) {
    if (ll == rr) return ll;
    int mm = ll + ((rr - ll) >> 1);
    if (k <= cnt[lc[cr]])
      return find_kth(lc[cr], ll, mm, k);
    else
      return find_kth(rc[cr], mm + 1, rr, k - cnt[lc[cr]]);
  }

 public:
  SegmentTree(int l, int r, int n)
      : rt(0), id(0), L(l), R(r), lc(n), rc(n), cnt(n) {}

  void insert(int v) { insert(rt, L, R, v); }

  int find_kth(int k) { return find_kth(rt, L, R, k); }
};

int main() {
  int n;
  std::cin >> n;
  SegmentTree seg(0, 1e9, 5e6);
  for (int i = 0; i < n; ++i) {
    int v;
```

```
    std::cin >> v;
    seg.insert(v);
    if (~i & 1) std::cout << seg.find_kth((i >> 1) + 1) << '\n';
  }
  return 0;
}

```

习题

基础实现与区间信息设计：

- Luogu P2574 XOR 的艺术[11]
- Luogu P4588【TJOI2018】数学计算[12]
- Luogu P1253 扶苏的问题[13]
- Luogu P1471 方差[14]
- Luogu P4513 小白逛公园[15]
- Luogu P2572【SCOI2010】序列操作[16]

动态开点：

- Codeforces 915 E. Physical Education Lessons[17]
- Library Checker - Range Affine Range Sum (Large Array)[18]
- Codeforces 817 F. MEX Queries[19]

标记永久化：

- Luogu P1502 窗口的星星[20]
- 2018 Multi-University Training Contest 5 G. Glad You Came[21]

线段树二分：

- AtCoder Library Practice Contest J - Segment Tree[22]
- AtCoder Beginner Contest 292 Ex - Rating Estimator[23]
- Luogu P4137 Rmq Problem/mex[24]
- Codeforces 773 E. Blog Post Rating[25]
- Codeforces 407 E. k-d-sequence[26]
- Codeforces 671 E. Organizing a Race[27]

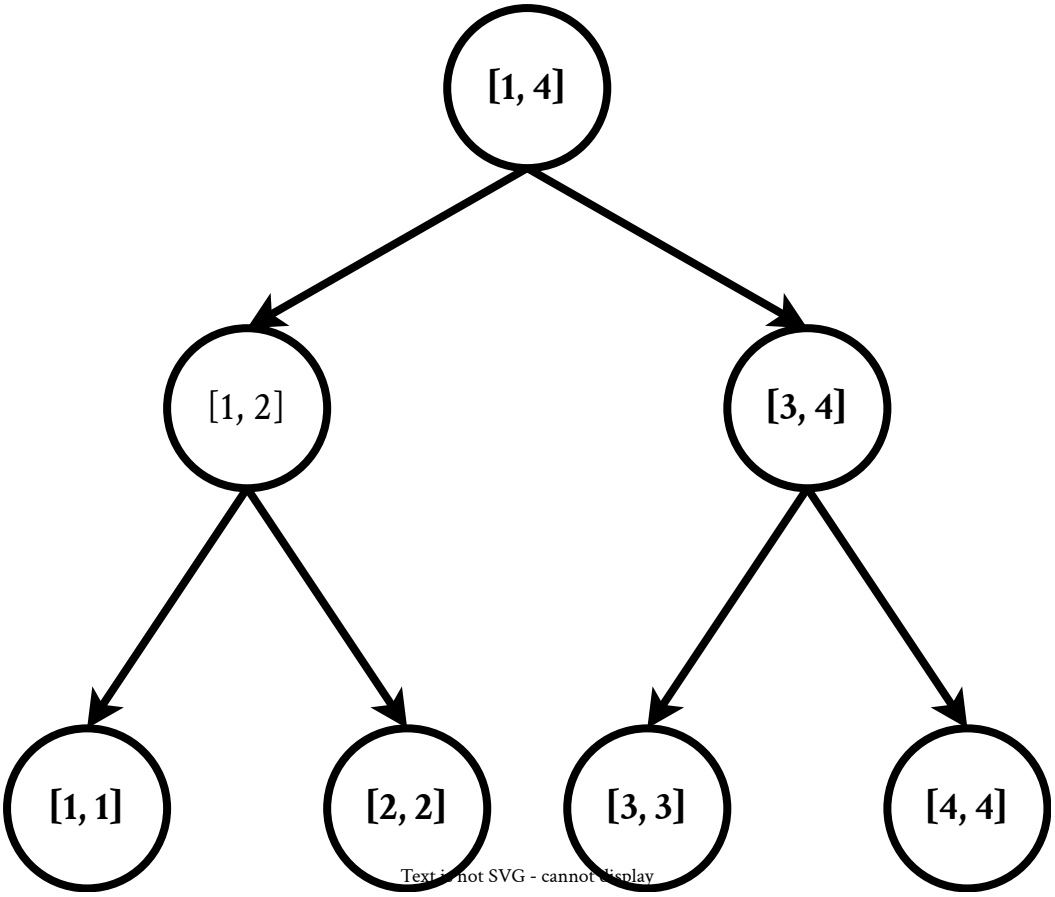
权值线段树：

- Luogu P1908 逆序对[28]
- Luogu P1801 黑匣子[29]
- Luogu P1637 三元上升子序列[30]
- Luogu P2286【HNOI2004】宠物收养场[31]

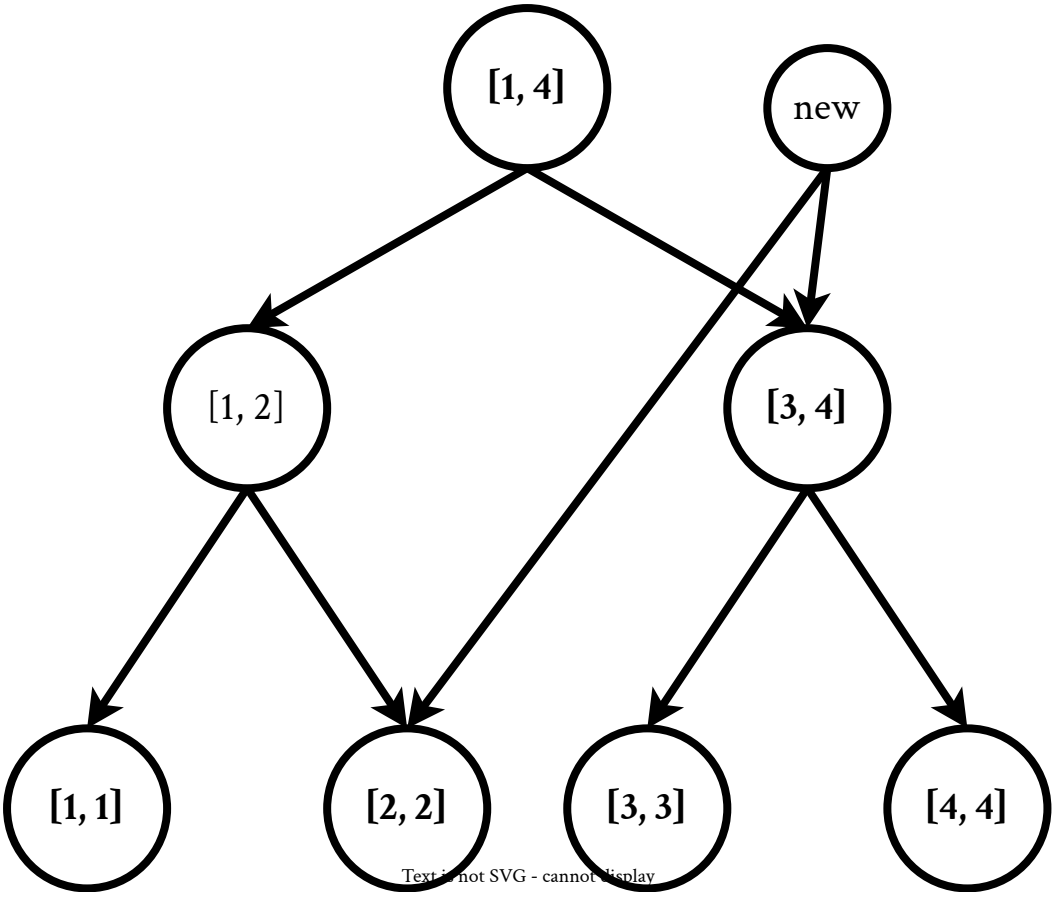
应用：线段树优化建图

在建图连边的过程中，我们有时会碰到这种题目，一个点向一段连续的区间中的点连边或者一个连续的区间向一个点连边，如果我们真的一条一条连过去，那一旦点的数量多了复杂度就爆炸了，这里就需要用线段树的区间性质来优化我们的建图了.

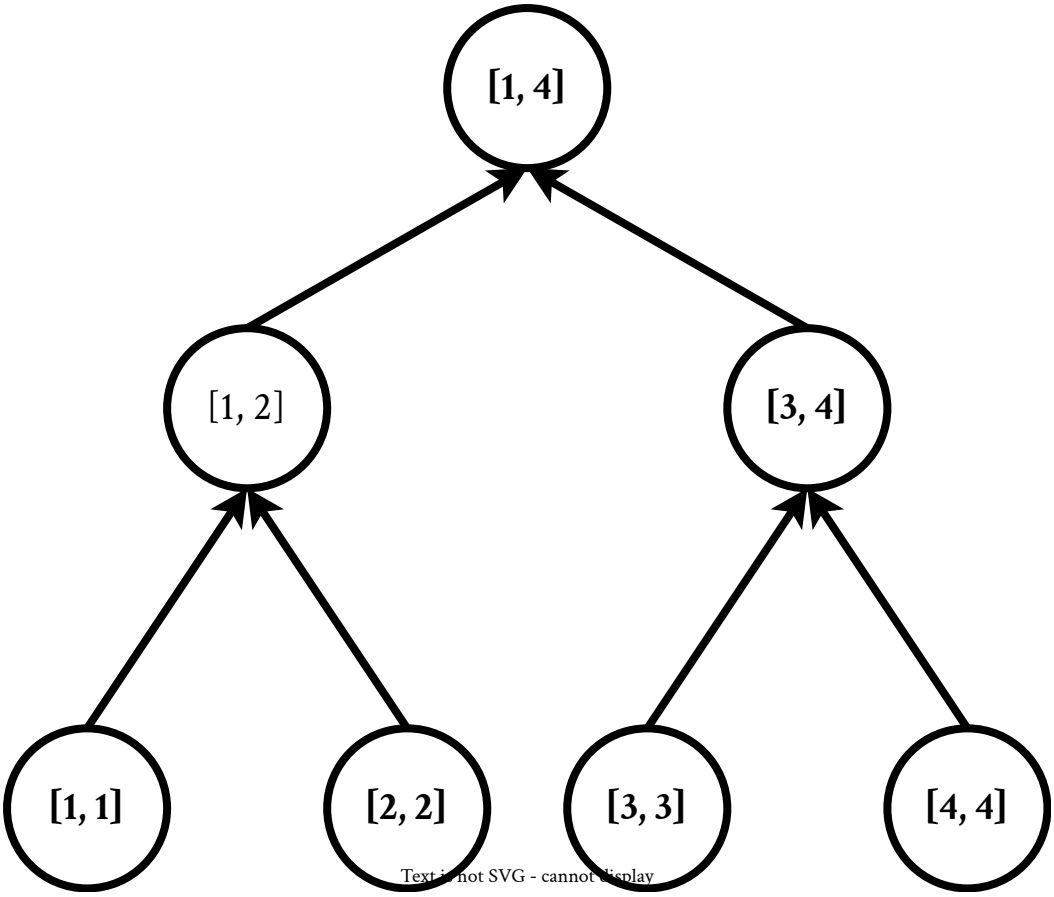
下面是一个线段树.



每个结点都代表了一段区间，假设我们要向区间 $[2, 4]$ 连边.



在一些题目中, 还会出现一段区间连向一个点的情况, 则我们将上面第一张图的有向边全部反过来即可，上面的树叫做入树，下面这个叫做出树.



Legacy[32]

题目大意：有 n 个点、q 次操作．每一种操作为以下三种类型中的一种：

- 操作一：连一条 $u \to v$ 的有向边，权值为 w.
- 操作二：对于所有 $i \in [l, r]$ 连一条 $u \to i$ 的有向边，权值为 w.
- 操作三：对于所有 $i \in [l, r]$ 连一条 $i \to u$ 的有向边，权值为 w.

求从点 s 到其他点的最短路.

$1 \le n, q \le 10^5, 1 \le w \le 10^9$.

参考代码

```cpp
#include <bitset>
#include <iostream>
#include <queue>
#include <vector>
using namespace std;
using ll = long long;

constexpr int N = 1e5 + 5;

using pil = pair<int, ll>;
using pli = pair<ll, int>;

int n, q, s, tot, rt1, rt2;
```

```
int pos[N];
ll dis[N << 3];
vector<pil> e[N << 3];
bitset<(N << 3)> vis;

struct seg {
  int l, r, lson, rson;
} t[N << 3];

int ls(int u) {  // 左儿子
  return t[u].lson;
}

int rs(int u) {  // 右儿子
  return t[u].rson;
}

void build(int &u, int l, int r) {  // 动态开点建造入树
  u = ++tot;
  t[u] = seg{l, r};
  if (l == r) {
    pos[l] = u;
    return;
  }
  int mid = (l + r) >> 1;
  build(t[u].lson, l, mid);
  build(t[u].rson, mid + 1, r);
  e[u].emplace_back(ls(u), 0);
  e[u].emplace_back(rs(u), 0);
}

void build2(int &u, int l, int r) {  // 动态开点建造出树
  if (l == r) {
    u = pos[l];
    return;
  }
  u = ++tot;
  t[u] = seg{l, r};
  int mid = (l + r) >> 1;
  build2(t[u].lson, l, mid);
  build2(t[u].rson, mid + 1, r);
  e[ls(u)].emplace_back(u, 0);
  e[rs(u)].emplace_back(u, 0);
}

void add1(int u, int lr, int rr, int v, ll w) {  // 点向区间连边
  if (lr <= t[u].l && t[u].r <= rr) {
    e[v].emplace_back(u, w);
    return;
  }
  int mid = (t[u].l + t[u].r) >> 1;
```

```
  if (lr <= mid) {
    add1(ls(u), lr, rr, v, w);
  }
  if (rr > mid) {
    add1(rs(u), lr, rr, v, w);
  }
}

void add2(int u, int lr, int rr, int v, ll w) {  // 区间向点连边
  if (lr <= t[u].l && t[u].r <= rr) {
    e[u].emplace_back(v, w);
    return;
  }
  int mid = (t[u].l + t[u].r) >> 1;
  if (lr <= mid) {
    add2(ls(u), lr, rr, v, w);
  }
  if (rr > mid) {
    add2(rs(u), lr, rr, v, w);
  }
}

void dij(int S) {
  priority_queue<pli, vector<pli>, greater<pli>> q;
  int tot = (n << 2);
  for (int i = 1; i <= tot; ++i) {
    dis[i] = 1e18;
  }
  dis[S] = 0;
  q.emplace(dis[S], S);
  while (!q.empty()) {
    pli fr = q.top();
    q.pop();
    int u = fr.second;
    if (vis[u]) continue;
    for (pil it : e[u]) {
      int v = it.first;
      ll w = it.second;
      if (dis[v] > dis[u] + w) {
        dis[v] = dis[u] + w;
        q.emplace(dis[v], v);
      }
    }
  }
}

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  cin >> n >> q >> s;
  build(rt1, 1, n);
  build2(rt2, 1, n);
```

```
  for (int i = 1, op, u; i <= q; ++i) {
    cin >> op >> u;
    if (op == 1) {
      int v;
      ll w;
      cin >> v >> w;
      e[pos[u]].emplace_back(pos[v], w);
    } else if (op == 2) {
      int l, r;
      ll w;
      cin >> l >> r >> w;
      add1(rt1, l, r, pos[u], w);
    } else {
      int l, r;
      ll w;
      cin >> l >> r >> w;
      add2(rt2, l, r, pos[u], w);
    }
  }
  dij(pos[s]);
  for (int i = 1; i <= n; ++i) {
    if (dis[pos[i]] == 1e18) {
      cout << "-1 ";
    } else {
      cout << dis[pos[i]] << ' ';
    }
  }
  return 0;
}

```

参考资料与注释

- 统计的力量 - 张昆玮[33]
- Generalizing Segment Trees by Eklavya[34]
- 线段树进阶 Part 1 by Alex_Wei - 洛谷[35]

[1] AtCoder Library

[2] AtCoder Library

[3] Library Checker - Point Add Range Sum

[4] Luogu P3372【模板】线段树 1

[5] Luogu P3373【模板】线段树 2

[6] SPOJ GSS3 - Can you answer these queries III

[7] Luogu P13825【模板】线段树 1.5

[8] Luogu P5490【模板】扫描线 & 矩形面积并

[9] Luogu P2894【USACO08FEB】Hotel G

[10] Luogu P1168 中位数

[11] Luogu P2574 XOR 的艺术

[12] Luogu P4588【TJOI2018】数学计算

[13] Luogu P1253 扶苏的问题

[14] Luogu P1471 方差

[15] Luogu P4513 小白逛公园

[16] Luogu P2572【SCOI2010】序列操作 

[17] Codeforces 915 E. Physical Education Lessons 

[18] Library Checker - Range Affine Range Sum (Large Array) 

[19] Codeforces 817 F. MEX Queries 

[20] Luogu P1502 窗口的星星 

[21] 2018 Multi-University Training Contest 5 G. Glad You Came 

[22] AtCoder Library Practice Contest J - Segment Tree 

[23] AtCoder Beginner Contest 292 Ex - Rating Estimator 

[24] Luogu P4137 Rmq Problem/mex 

[25] Codeforces 773 E. Blog Post Rating 

[26] Codeforces 407 E. k-d-sequence 

[27] Codeforces 671 E. Organizing a Race 

[28] Luogu P1908 逆序对 

[29] Luogu P1801 黑匣子 

[30] Luogu P1637 三元上升子序列 

[31] Luogu P2286【HNOI2004】宠物收养场 

[32] Legacy 

[33] 统计的力量 - 张昆玮 

[34] Generalizing Segment Trees by Eklavya 

[35] 线段树进阶 Part 1 by Alex_Wei - 洛谷 

10.13.2 线段树合并 & 分裂

> **Authors:** ChungZH, billchenchina, Chrogeek, Early0v0, ethan-enhe, HeRaNO, hsfzLZH1, iamtwz, Ir1d, konnyakuxzy, luoguojie, Marcythm, orzAtalod, StudyingFather, wy-luke, Xeonacid, CCXXXI, chenryang, chenzheAya, CJSoft, cjsoft, countercurrent-time, DawnMagnet, Enter-tainer, GavinZhengOI, Haohu Shen, Henry-ZHR, hjsjhn, hly1204, jaxvanyang, Jebearssica, kenlig, ksyx, megakite, Menci, moon-dim, NachtgeistW, onelittlechildawa, ouuan, shadowice1984, shawlleyw, shuzhouliu, SukkaW, Tiphereth-A, x2e6, Ycrpro, yifan0305, zeningc

线段树的合并与分裂是线段树的常用技巧，常见于权值线段树维护可重集的场景.

例如，树上某些结点处有若干操作，如果需要自下而上地将子节点信息传递给亲节点，而单个结点处的信息又方便用线段树维护时，就可以应用线段树合并的技巧控制整体的复杂度.

线段树合并

过程

顾名思义，线段树合并是指建立一棵新的线段树，这棵线段树的每个节点都是两棵原线段树对应节点合并后的结果. 它常常被用于维护树上或是图上的信息.

显然，我们不可能真的每次建满一颗新的线段树，因此我们需要使用上文的动态开点线段树.

线段树合并的过程本质上相当暴力：

假设两颗线段树为 A 和 B，我们从 1 号节点开始递归合并.

递归到某个节点时，如果 A 树或者 B 树上的对应节点为空，直接返回另一个树上对应节点，这里运用了动态开点线段树的特性.

如果递归到叶子节点，我们合并两棵树上的对应节点.

最后，根据子节点更新当前节点并且返回.

线段树合并的复杂度

显然，对于两颗满的线段树，单次合并操作的复杂度是 $O(n)$ 的. 但实际情况下使用的常常是权值线段树，所有需要合并的线段树的总点数和 n 的规模相差并不大. 并且合并时一般不会重复地合并

某个线段树，所以我们最终增加的点数大致是 $n \log n$ 级别的．这样，合并所有线段树总的复杂度就是 $O(n \log n)$ 级别的．当然，在一些情况下，可并堆可能是更好的选择．

实现

```
int merge(int a, int b, int l, int r) {
  if (!a) return b;
  if (!b) return a;
  if (l == r) {
    // do something...
    return a;
  }
  int mid = (l + r) >> 1;
  tr[a].l = merge(tr[a].l, tr[b].l, l, mid);
  tr[a].r = merge(tr[a].r, tr[b].r, mid + 1, r);
  pushup(a);
  return a;
}
```

例题

luogu P4556 [Vani 有约会] 雨天的尾巴/【模板】线段树合并[1]

解题思路

线段树合并模板题，用差分把树上修改转化为单点修改，然后向上 dfs 线段树合并统计答案即可．

参考代码

```
#include <iostream>
#include <vector>
using namespace std;
int n, fa[100005][22], dep[100005], rt[100005];
int sum[5000005], cnt = 0, res[5000005], ls[5000005], rs[5000005];
int m, ans[100005];
vector<int> v[100005];

void update(int x) {
  if (sum[ls[x]] < sum[rs[x]]) {
    res[x] = res[rs[x]];
    sum[x] = sum[rs[x]];
  } else {
    res[x] = res[ls[x]];
    sum[x] = sum[ls[x]];
  }
}

int merge(int a, int b, int x, int y) {
  if (!a) return b;
```

```
  if (!b) return a;
  if (x == y) {
    sum[a] += sum[b];
    return a;
  }
  int mid = (x + y) >> 1;
  ls[a] = merge(ls[a], ls[b], x, mid);
  rs[a] = merge(rs[a], rs[b], mid + 1, y);
  update(a);
  return a;
}

int add(int id, int x, int y, int co, int val) {
  if (!id) id = ++cnt;
  if (x == y) {
    sum[id] += val;
    res[id] = co;
    return id;
  }
  int mid = (x + y) >> 1;
  if (co <= mid)
    ls[id] = add(ls[id], x, mid, co, val);
  else
    rs[id] = add(rs[id], mid + 1, y, co, val);
  update(id);
  return id;
}

void initlca(int x) {
  for (int i = 0; i <= 20; i++) fa[x][i + 1] = fa[fa[x][i]][i];
  for (int i : v[x]) {
    if (i == fa[x][0]) continue;
    dep[i] = dep[x] + 1;
    fa[i][0] = x;
    initlca(i);
  }
}

int lca(int x, int y) {
  if (dep[x] < dep[y]) swap(x, y);
  for (int d = dep[x] - dep[y], i = 0; d; d >>= 1, i++)
    if (d & 1) x = fa[x][i];
  if (x == y) return x;
  for (int i = 20; i >= 0; i--)
    if (fa[x][i] != fa[y][i]) x = fa[x][i], y = fa[y][i];
  return fa[x][0];
}

void cacl(int x) {
  for (int i : v[x]) {
    if (i == fa[x][0]) continue;
```

```
    cacl(i);
    rt[x] = merge(rt[x], rt[i], 1, 100000);
  }
  ans[x] = res[rt[x]];
  if (sum[rt[x]] == 0) ans[x] = 0;
}

int main() {
  ios::sync_with_stdio(false);
  cin >> n >> m;
  for (int i = 0; i < n - 1; i++) {
    int a, b;
    cin >> a >> b;
    v[a].push_back(b);
    v[b].push_back(a);
  }
  initlca(1);
  for (int i = 0; i < m; i++) {
    int a, b, c;
    cin >> a >> b >> c;
    rt[a] = add(rt[a], 1, 100000, c, 1);
    rt[b] = add(rt[b], 1, 100000, c, 1);
    int t = lca(a, b);
    rt[t] = add(rt[t], 1, 100000, c, -1);
    rt[fa[t][0]] = add(rt[fa[t][0]], 1, 100000, c, -1);
  }
  cacl(1);
  for (int i = 1; i <= n; i++) cout << ans[i] << endl;
  return 0;
}

```

线段树分裂

过程

线段树分裂实质上是线段树合并的逆过程. 线段树分裂只适用于有序的序列, 无序的序列是没有意义的, 常用在动态开点的权值线段树.

注意当分裂和合并都存在时, 我们在合并的时候必须回收节点, 以避免分裂时会可能出现节点重复占用的问题.

从一颗区间为 $[1,N]$ 的线段树中分裂出 $[l,r]$, 建一颗新的树:

从 1 号结点开始递归分裂, 当节点不存在或者代表的区间 $[s,t]$ 与 $[l,r]$ 没有交集时直接回溯.

当 $[s,t]$ 与 $[l,r]$ 有交集时需要开一个新结点.

当 $[s,t]$ 包含于 $[l,r]$ 时, 需要将当前结点直接接到新的树下面, 并把旧边断开.

线段树分裂的复杂度

可以发现被断开的边最多只会有 $\log n$ 条，所以最终每次分裂的时间复杂度就是 $O(\log n)$，相当于区间查询的复杂度.

实现

```
void split(int &p, int &q, int s, int t, int l, int r) {
  if (t < l || r < s) return;
  if (!p) return;
  if (l <= s && t <= r) {
    q = p;
    p = 0;
    return;
  }
  if (!q) q = New();
  int m = s + t >> 1;
  if (l <= m) split(ls[p], ls[q], s, m, l, r);
  if (m < r) split(rs[p], rs[q], m + 1, t, l, r);
  push_up(p);
  push_up(q);
}
```

例题

P5494【模板】线段树分裂[2]

解题思路

线段树分裂模板题，将 $[x,y]$ 分裂出来.

- 将 t 树合并入 p 树：单次合并即可.
- p 树中插入 x 个 q：单点修改.
- 查询 $[x,y]$ 中数的个数：区间求和.
- 查询第 k 小.

参考代码

```
#include <iostream>
using namespace std;
constexpr int N = 2e5 + 10;
int n, m;
int idx = 1;
long long sum[N << 5];
int ls[N << 5], rs[N << 5], root[N << 2], rub[N << 5], cnt, tot;

// 内存分配与回收
int New() { return cnt ? rub[cnt--] : ++tot; }

void Del(int &p) {
  ls[p] = rs[p] = sum[p] = 0;
```

```
  rub[++cnt] = p;
  p = 0;
}

void push_up(int p) { sum[p] = sum[ls[p]] + sum[rs[p]]; }

void build(int s, int t, int &p) {
  if (!p) p = New();
  if (s == t) {
    cin >> sum[p];
    return;
  }
  int m = (s + t) >> 1;
  build(s, m, ls[p]);
  build(m + 1, t, rs[p]);
  push_up(p);
}

// 单点修改
void update(int x, int c, int s, int t, int &p) {
  if (!p) p = New();
  if (s == t) {
    sum[p] += c;
    return;
  }
  int m = (s + t) >> 1;
  if (x <= m)
    update(x, c, s, m, ls[p]);
  else
    update(x, c, m + 1, t, rs[p]);
  push_up(p);
}

// 合并
int merge(int p, int q, int s, int t) {
  if (!p || !q) return p + q;
  if (s == t) {
    sum[p] += sum[q];
    Del(q);
    return p;
  }
  int m = (s + t) >> 1;
  ls[p] = merge(ls[p], ls[q], s, m);
  rs[p] = merge(rs[p], rs[q], m + 1, t);
  push_up(p);
  Del(q);
  return p;
}

// 分裂
void split(int &p, int &q, int s, int t, int l, int r) {
```

```
  if (t < l || r < s) return;
  if (!p) return;
  if (l <= s && t <= r) {
    q = p;
    p = 0;
    return;
  }
  if (!q) q = New();
  int m = (s + t) >> 1;
  if (l <= m) split(ls[p], ls[q], s, m, l, r);
  if (m < r) split(rs[p], rs[q], m + 1, t, l, r);
  push_up(p);
  push_up(q);
}

long long query(int l, int r, int s, int t, int p) {
  if (!p) return 0;
  if (l <= s && t <= r) return sum[p];
  int m = (s + t) >> 1;
  long long ans = 0;
  if (l <= m) ans += query(l, r, s, m, ls[p]);
  if (m < r) ans += query(l, r, m + 1, t, rs[p]);
  return ans;
}

int kth(int s, int t, int k, int p) {
  if (s == t) return s;
  int m = (s + t) >> 1;
  long long left = sum[ls[p]];
  if (k <= left)
    return kth(s, m, k, ls[p]);
  else
    return kth(m + 1, t, k - left, rs[p]);
}

int main() {
  cin >> n >> m;
  build(1, n, root[1]);
  while (m--) {
    int op;
    cin >> op;
    if (!op) {
      int p, x, y;
      cin >> p >> x >> y;
      split(root[p], root[++idx], 1, n, x, y);
    } else if (op == 1) {
      int p, t;
      cin >> p >> t;
      root[p] = merge(root[p], root[t], 1, n);
    } else if (op == 2) {
      int p, x, q;
```

```
116       cin >> p >> x >> q;
117       update(q, x, 1, n, root[p]);
118     } else if (op == 3) {
119       int p, x, y;
120       cin >> p >> x >> y;
121       cout << query(x, y, 1, n, root[p]) << endl;
122     } else {
123       int p, k;
124       cin >> p >> k;
125       if (sum[root[p]] < k)
126         cout << -1 << endl;
127       else
128         cout << kth(1, n, k, root[p]) << endl;
129     }
130   }
131 }
132
```

习题

- Luogu P4556 [Vani 有约会] 雨天的尾巴/【模板】线段树合并[3]
- Luogu P5494【模板】线段树分裂[4]
- Luogu P1600 天天爱跑步[5]
- Luogu P4577 [FJOI2018] 领导集团问题[6]
- Luogu P2824 [HEOI2016/TJOI2016] 排序[7]

外部参考资料

[1] luogu P4556 [Vani 有约会] 雨天的尾巴/【模板】线段树合并

[2] P5494【模板】线段树分裂

[3] Luogu P4556 [Vani 有约会] 雨天的尾巴/【模板】线段树合并

[4] Luogu P5494【模板】线段树分裂

[5] Luogu P1600 天天爱跑步

[6] Luogu P4577 [FJOI2018] 领导集团问题

[7] Luogu P2824 [HEOI2016/TJOI2016] 排序

10.13.3 李超线段树

引入

洛谷 4097 [HEOI2013]Segment[1]

要求在平面直角坐标系下维护两个操作（强制在线）：

1. 在平面上加入一条线段. 记第 i 条被插入的线段的标号为 i, 该线段的两个端点分别为 (x_0, y_0), (x_1, y_1).
2. 给定一个数 k，询问与直线 $x = k$ 相交的线段中，交点纵坐标最大的线段的编号（若有多条线段与查询直线的交点纵坐标都是最大的，则输出编号最小的线段）. 特别地，若不存在线段与给定直线相交，输出 0.

数据满足：操作总数 $1 \le n \le 10^5$，$1 \le k, x_0, x_1 \le 39989$，$1 \le y_0, y_1 \le 10^9$.

我们发现，传统的线段树无法很好地维护这样的信息．这种情况下，**李超线段树** 便应运而生．

过程

我们可以把任务转化为维护如下操作：

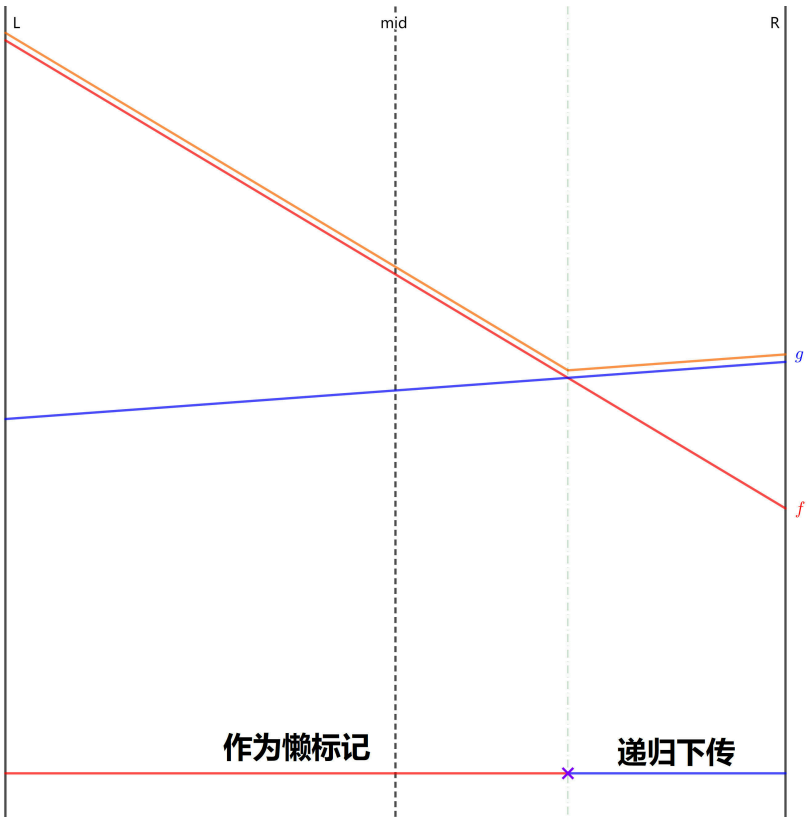
- 加入一个一次函数，定义域为 $[l,r]$；
- 给定 k，求定义域包含 k 的所有一次函数中，在 $x=k$ 处取值最大的那个，如果有多个函数取值相同，选编号最小的．

> ⚠ **注意**
>
> 当线段垂直于 x 轴时，会出现除以零的情况．假设线段两端点分别为 (x,y_0) 和 (x,y_1)，$y_0<y_1$，则插入定义域为 $[x,x]$ 的一次函数 $f(x)=0\cdot x+y_1$．

看到区间修改，我们按照线段树解决区间问题的常见方法，给每个节点一个懒标记．每个节点 i 的懒标记都是一条线段，记为 l_i，表示要用 l_i 更新该节点所表示的整个区间．

现在我们需要插入一条线段 f，考虑某个被新线段 f 完整覆盖的线段树区间．若该区间无标记，直接打上用该线段更新的标记．

如果该区间已经有标记了，由于标记难以合并，只能把标记下传．但是子节点也有自己的标记，也可能产生冲突，所以我们要递归下传标记．



如图，按新线段 f 取值是否大于原标记 g，我们可以把当前区间分为两个子区间．其中 **肯定有一个子区间被左区间或右区间完全包含**，也就是说，在两条线段中，肯定有一条线段，只可能成为左区间的答案，或者只可能成为右区间的答案．我们用这条线段递归更新对应子树，用另一条线段作为懒标记更新整个区间，这就保证了递归下传的复杂度．当一条线段只可能成为左或右区间的答案时，才会被下传，所以不用担心漏掉某些线段．

具体来说，设当前区间的中点为 m，我们拿新线段 f 在中点处的值与原最优线段 g 在中点处的值作比较.

如果新线段 f 更优，则将 f 和 g 交换. 那么现在考虑在中点处 f 不如 g 优的情况：

1. 若在左端点处 f 更优，那么 f 和 g 必然在左半区间中产生了交点，f 只有在左区间才可能优于 g，递归到左儿子中进行下传；
2. 若在右端点处 f 更优，那么 f 和 g 必然在右半区间中产生了交点，f 只有在右区间才可能优于 g，递归到右儿子中进行下传；
3. 若在左右端点处 g 都更优，那么 f 不可能成为答案，不需要继续下传.

除了这两种情况之外，还有一种情况是 f 和 g 刚好交于中点，在程序实现时可以归入中点处 f 不如 g 优的情况，结果会往 f 更优的一个端点进行递归下传.

最后将 g 作为当前区间的懒标记.

下传标记：

实现

```cpp
constexpr double eps = 1e-9;

int cmp(double x, double y) {  // 因为用到了浮点数，所以会有精度误差
  if (x - y > eps) return 1;
  if (y - x > eps) return -1;
  return 0;
}

//...

void upd(int root, int cl, int cr, int u) {  // 对线段完全覆盖到的区间进行修改
  int &v = s[root], mid = (cl + cr) >> 1;
  int bmid = cmp(calc(u, mid), calc(v, mid));
  if (bmid == 1 || (!bmid && u < v))  // 在此题中记得判线段编号
    swap(u, v);
  int bl = cmp(calc(u, cl), calc(v, cl)), br = cmp(calc(u, cr), calc(v,
cr));
  if (bl == 1 || (!bl && u < v)) upd(root << 1, cl, mid, u);
  if (br == 1 || (!br && u < v)) upd(root << 1 | 1, mid + 1, cr, u);
  // 上面两个 if 的条件最多只有一个成立，这保证了李超树的时间复杂度
}
```

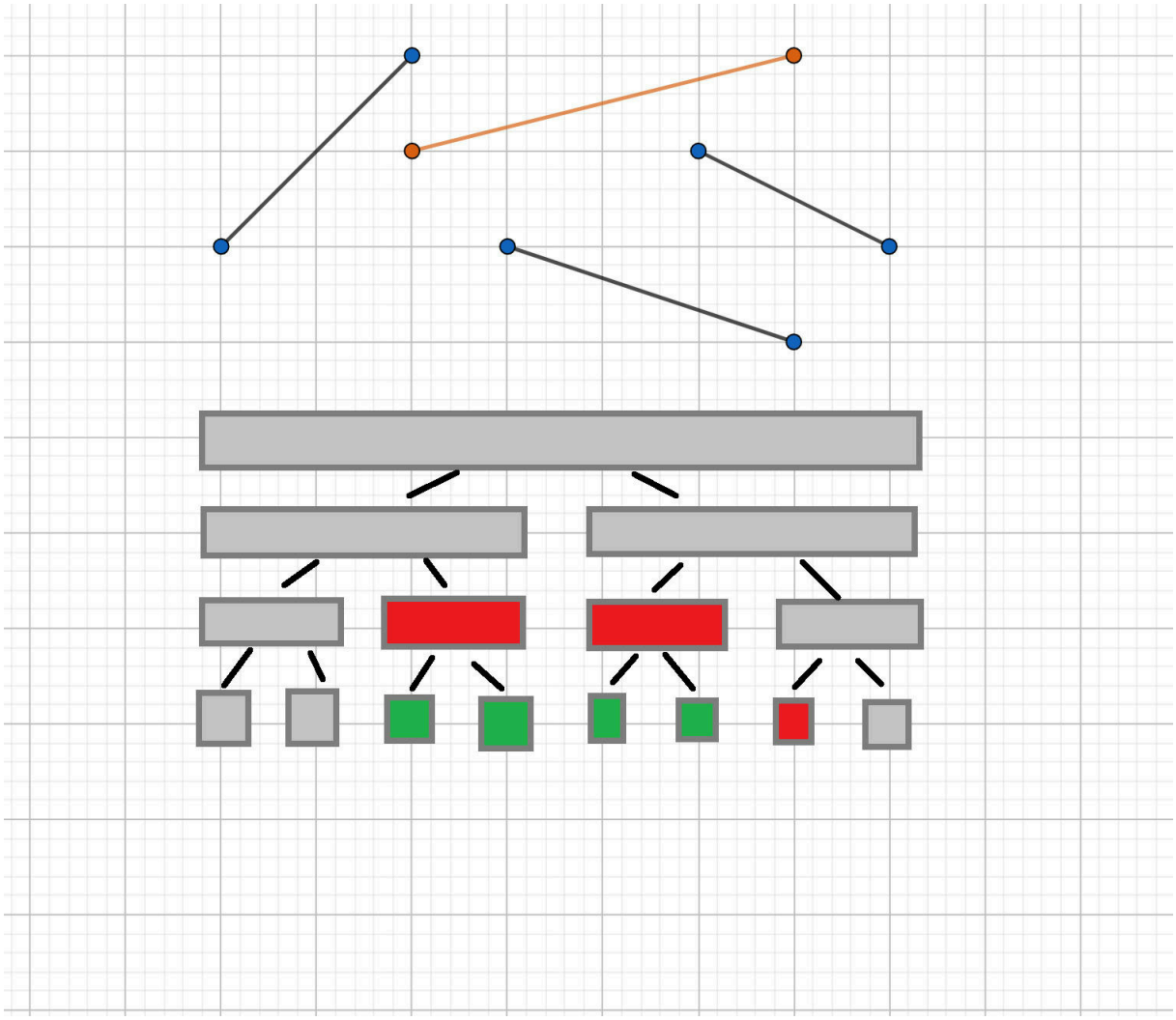
拆分线段：

实现

```cpp
void update(int root, int cl, int cr, int l, int r,
            int u) {  // 定位插入线段完全覆盖到的区间
  if (l <= cl && cr <= r) {
    upd(root, cl, cr, u);  // 完全覆盖当前区间，更新当前区间的标记
    return;
  }
  int mid = (cl + cr) >> 1;
  if (l <= mid) update(root << 1, cl, mid, l, r, u);  // 递归拆分区间
  if (mid < r) update(root << 1 | 1, mid + 1, cr, l, r, u);
```

```
10 }
```

注意懒标记并不等价于在区间中点处取值最大的线段.



如图，加入黄色线段后，只有红色节点的标记被更新，而绿色节点的标记还未被改变. 但在第二、三、四个绿色区间的中点处显然黄色线段取值最大.

查询时，我们可以利用标记永久化思想，在包含 x 的所有线段树区间（不超过 $O(\log n)$ 个）的标记线段中，比较得出最终答案.

查询:

实现

```
pdi query(int root, int l, int r, int d) {  // 查询
  if (r < d || d < l) return {0, 0};
  int mid = (l + r) >> 1;
  double res = calc(s[root], d);
  if (l == r) return {res, s[root]};
  return pmax({res, s[root]}, pmax(query(root << 1, l, mid, d),
                                   query(root << 1 | 1, mid + 1, r, d)));
```

```
}
```

根据上面的描述，查询过程的时间复杂度显然为 $O(\log n)$，而插入过程中，我们需要将原线段拆分到 $O(\log n)$ 个区间中，对于每个区间，我们又需要花费 $O(\log n)$ 的时间递归下传，从而插入过程的时间复杂度为 $O(\log^2 n)$.

[HEOI2013]Segment[2] 参考代码

```
#include <iostream>
constexpr int MOD1 = 39989;
constexpr int MOD2 = 1000000000;
constexpr int MAXT = 40000;
using namespace std;
using pdi = pair<double, int>;

constexpr double eps = 1e-9;

int cmp(double x, double y) {
  if (x - y > eps) return 1;
  if (y - x > eps) return -1;
  return 0;
}

struct line {
  double k, b;
} p[100005];

int s[160005];
int cnt;

double calc(int id, int d) { return p[id].b + p[id].k * d; }

void add(int x0, int y0, int x1, int y1) {
  cnt++;
  if (x0 == x1)  // 特判直线斜率不存在的情况
    p[cnt].k = 0, p[cnt].b = max(y0, y1);
  else
    p[cnt].k = 1.0 * (y1 - y0) / (x1 - x0), p[cnt].b = y0 - p[cnt].k * x0;
}

void upd(int root, int cl, int cr, int u) {  // 对线段完全覆盖到的区间进行修改
  int &v = s[root], mid = (cl + cr) >> 1;
  int bmid = cmp(calc(u, mid), calc(v, mid));
  if (bmid == 1 || (!bmid && u < v)) swap(u, v);
  int bl = cmp(calc(u, cl), calc(v, cl)), br = cmp(calc(u, cr), calc(v,
cr));
  if (bl == 1 || (!bl && u < v)) upd(root << 1, cl, mid, u);
  if (br == 1 || (!br && u < v)) upd(root << 1 | 1, mid + 1, cr, u);
}

void update(int root, int cl, int cr, int l, int r,
```

```
43              int u) {  // 定位插入线段完全覆盖到的区间
44    if (l <= cl && cr <= r) {
45      upd(root, cl, cr, u);
46      return;
47    }
48    int mid = (cl + cr) >> 1;
49    if (l <= mid) update(root << 1, cl, mid, l, r, u);
50    if (mid < r) update(root << 1 | 1, mid + 1, cr, l, r, u);
51  }
52
53  pdi pmax(pdi x, pdi y) {  // pair max函数
54    if (cmp(x.first, y.first) == -1)
55      return y;
56    else if (cmp(x.first, y.first) == 1)
57      return x;
58    else
59      return x.second < y.second ? x : y;
60  }
61
62  pdi query(int root, int l, int r, int d) {  // 查询
63    if (r < d || d < l) return {0, 0};
64    int mid = (l + r) >> 1;
65    double res = calc(s[root], d);
66    if (l == r) return {res, s[root]};
67    return pmax({res, s[root]}, pmax(query(root << 1, l, mid, d),
68                                     query(root << 1 | 1, mid + 1, r, d)));
69  }
70
71  int main() {
72    ios::sync_with_stdio(false);
73    int n, lastans = 0;
74    cin >> n;
75    while (n--) {
76      int op;
77      cin >> op;
78      if (op == 1) {
79        int x0, y0, x1, y1;
80        cin >> x0 >> y0 >> x1 >> y1;
81        x0 = (x0 + lastans - 1 + MOD1) % MOD1 + 1,
82        x1 = (x1 + lastans - 1 + MOD1) % MOD1 + 1;
83        y0 = (y0 + lastans - 1 + MOD2) % MOD2 + 1,
84        y1 = (y1 + lastans - 1 + MOD2) % MOD2 + 1;
85        if (x0 > x1) swap(x0, x1), swap(y0, y1);
86        add(x0, y0, x1, y1);
87        update(1, 1, MOD1, x0, x1, cnt);
88      } else {
89        int x;
90        cin >> x;
91        x = (x + lastans - 1 + MOD1) % MOD1 + 1;
92        cout << (lastans = query(1, 1, MOD1, x).second) << endl;
93      }
94    }
```

```
95   return 0;
96 }
97
```

合并

类似于普通线段树的合并, 我们定义以下过程来将两个李超线段树节点 u, v 合并, 并以 u 作为新的根.

1. 如果 v 为空，结束过程.
2. 如果 u 为空，将 v 复制给 u.
3. 将 v 对应线段插入到 u 为根的子树.
4. 递归将 u, v 的左右子树对应合并.

若合并若干李超线段树涉及的总点数为 n，则该过程的复杂度为 $O(n \log n)$：对于任意线段在树上对应的节点, 每次涉及移动它时, 我们要么使其深度 $+1$, 要么直接从树上删除, 这两个操作的代价都是 $O(1)$ 的，而每个点深度至多为 $O(\log n)$，于是复杂度如上.

实现

```
void upd(int &root, int cl, int cr,
         int u) {  // 涉及多棵李超线段树合并，使用动态开点.
  static int idx = 0;
  if (!root) {
    s[root = ++idx] = u;
    return;
  }
  int &v = s[root], mid = (cl + cr) >> 1;
  int bmid = cmp(calc(u, mid), calc(v, mid));
  if (bmid == 1 || (!bmid && u < v)) swap(u, v);
  int bl = cmp(calc(u, cl), calc(v, cl)), br = cmp(calc(u, cr), calc(v,
cr));
  if (bl == 1 || (!bl && u < v)) upd(ls[root], cl, mid, u);
  if (br == 1 || (!br && u < v)) upd(rs[root], mid + 1, cr, u);
}

int merge(int &u, int &v, int l, int r) {
  if (!u || !v) {
    return u + v;
  }
  if (l == r) {
    int b = cmp(calc(s[v], l), calc(s[u], l));
    if (b == 1 || (!b && s[v] < s[u])) return v;
    return u;
  }
  upd(u, l, r, s[v]);
  int mid = (l + r) >> 1;
  ls[u] = merge(ls[u], ls[v], l, mid);
  rs[u] = merge(rs[u], rs[v], mid + 1, r);
  return u;
}
```

习题

「JSOI2008」Blue Mary 开公司[3]

「CodeChef」TSUM2 Sum on Tree[4]

「USACO13MAR」Hill Walk G[5]

「CF932F」Escape Through Leaf[6]

外部参考资料

[1] 洛谷 4097 [HEOI2013]Segment 

[2] [HEOI2013]Segment 

[3] 「JSOI2008」Blue Mary 开公司 

[4] 「CodeChef」TSUM2 Sum on Tree 

[5] 「USACO13MAR」Hill Walk G 

[6] 「CF932F」Escape Through Leaf 

10.13.4 猫树

> **Authors:** ChungZH, billchenchina, Chrogeek, Early0v0, ethan-enhe, HeRaNO, hsfzLZH1, iamtwz, Ir1d, konnyakuxzy, luoguojie, Marcythm, orzAtalod, StudyingFather, wy-luke, Xeonacid, CCXXXI, chenryang, chenzheAya, CJSoft, cjsoft, countercurrent-time, DawnMagnet, Enter-tainer, GavinZhengOI, Haohu Shen, Henry-ZHR, hjsjhn, hly1204, jaxvanyang, Jebearssica, kenlig, ksyx, megakite, Menci, moon-dim, NachtgeistW, onelittlechildawa, ouuan, shadowice1984, shawlleyw, shuzhouliu, SukkaW, Tiphereth-A, x2e6, Ycrpro, yifan0305, zeningc, hcx2012Git, karsl-program

引入

众所周知线段树可以支持高速查询某一段区间的信息和，比如区间最大子段和，区间和，区间矩阵的连乘积等等.

但是有一个问题在于普通线段树的区间询问在某些毒瘤的眼里可能还是有些慢了.

简单来说就是线段树建树的时候需要做 $O(n)$ 次合并操作，而每一次区间询问需要做 $O(\log n)$ 次合并操作，询问区间和这种东西的时候还可以忍受，但是当我们需要询问区间矩阵乘法这种合并（即一次矩阵乘法）复杂度高达 $O(k^3)$ 的信息的话，此时就算是做 $O(\log n)$ 次合并有些时候在时间上也是不可接受的.

而所谓「猫树」就是一种不支持修改，仅仅支持快速区间询问的一种静态线段树. 在询问次数较多（$m = \Omega(n)$）时，该做法才有显著价值.

构造一棵这样的静态线段树需要 $O(n \log n)$ 次合并操作，但是此时的查询复杂度被加速至 $O(1)$ 次合并操作.

在处理矩阵乘法这样合并代价高昂的信息的时候，使用猫树可以将单次查询复杂度从 $O(k^3 \log n)$ 降至 $O(k^3)$.

注意，在处理静态区间线性基查询这个问题上，虽然猫树可以优化普通线段树实现的区间线性基的复杂度，但有一种时空复杂度都更优的做法：前缀线性基→9.14.8@p1738. 而且普通线段树可以处理动态区间线性基问题，但对于静态区间线性基查询，猫树不仅实现较为复杂，不是最优解，在功能上也没有差别.

原理

在查询 $[l, r]$ 这段区间的信息和的时候，将线段树树上代表 $[l, l]$ 的节点和代表 $[r, r]$ 这段区间的节点在线段树上的 LCA 求出来，设这个节点 p 代表的区间为 $[L, R]$，我们会发现一些非常有趣的性质：

1. $[L, R]$ 这个区间一定包含 $[l, r]$. 显然，因为它既是 l 的祖先又是 r 的祖先.
2. $[l, r]$ 这个区间一定跨越 $[L, R]$ 的中点. 由于 p 是 l 和 r 的 LCA，这意味着 p 的左儿子是 l 的祖先而不是 r 的祖先，p 的右儿子是 r 的祖先而不是 l 的祖先. 因此，l 一定在 $[L, mid]$ 这个区间内，r 一定在 $(mid, R]$ 这个区间内.

有了这两个性质，我们就可以将询问的复杂度降至 $O(1)$ 了.

实现

具体来讲我们建树的时候对于线段树树上的一个节点，设它代表的区间为 $(l,r]$.

不同于传统线段树在这个节点里只保留 $[l,r]$ 的和，我们在这个节点里面额外保存 $(l,mid]$ 的后缀和数组和 $(mid,r]$ 的前缀和数组.

这样的话建树的复杂度为 $T(n)=2T(n/2)+O(n)=O(n\log n)$ 同理空间复杂度也从原来的 $O(n)$ 变成了 $O(n\log n)$.

下面是最关键的询问了.

如果我们询问的区间是 $[l,r]$ 那么我们把代表 $[l,l]$ 的节点和代表 $[r,r]$ 的节点的 LCA 求出来，记为 p.

根据刚才的两个性质，l,r 在 p 所包含的区间之内并且一定跨越了 p 的中点.

这意味这一个非常关键的事实是我们可以使用 p 里面的前缀和数组和后缀和数组，将 $[l,r]$ 拆成 $[l,mid]+(mid,r]$ 从而拼出来 $[l,r]$ 这个区间.

而这个过程仅仅需要 $O(1)$ 次合并操作!

不过我们好像忽略了点什么?

似乎求 LCA 的复杂度似乎还不是 $O(1)$，暴力求是 $O(\log n)$ 的，倍增法则是 $O(\log\log n)$ 的，转 ST 表的代价又太大……

堆式建树

具体来将我们将这个序列补成 2 的整次幂，然后建线段树.

此时我们发现线段树上两个节点的 LCA 编号，就是两个节点二进制编号的最长公共前缀 LCP.

稍作思考即可发现在 x 和 y 的二进制下 `lcp(x,y)=x>>digits[x^y]`.（其中 `digits[x]` 表示二进制下 x 的位数，即 $\lfloor\log_2 x\rfloor+1$）

所以我们预处理一个 `digits` 数组即可轻松完成求 LCA 的工作.

这样我们就构建了一个猫树.

由于建树的时候涉及到求前缀和和求后缀和，所以对于矩阵乘法这种合并复杂度为 $O(k^3)$ 的信息，建树复杂度为 $O(n\log n\cdot k^3)$. 在此基础上，猫树将静态区间矩阵乘法查询的单次复杂度从 $O(k^3\log n)$ 降至 $O(k^3)$，从而在询问次数较多（$m=\Omega(n)$）时，将处理 m 次查询的总复杂度从 $O(n\cdot k^3+m\cdot k^3\log n)$ 优化至 $O(n\log n\cdot k^3+m\cdot k^3)$.

参考

- immortalCO 的博客[1]
- [Kle77][2] V. Klee, "Can the Measure of be Computed in Less than O (n log n) Steps?," Am. Math. Mon., vol. 84, no. 4, pp. 284–285, Apr. 1977.
- [BeW80][3] Bentley and Wood, "An Optimal Worst Case Algorithm for Reporting Intersections of Rectangles," IEEE Trans. Comput., vol. C–29, no. 7, pp. 571–577, Jul. 1980.

外部参考资料

[1] immortalCO 的博客  [2] [Kle77]  [3] [BeW80] 

10.13.5 区间最值操作 & 区间历史最值

本文讲解吉老师在 2016 年国家集训队论文[1] 中提到的线段树处理历史区间最值的问题.

区间最值

笼统地说，区间最值操作指，将区间 $[l,r]$ 的数全部对 x 取 max 或 min，即 $a_i=\max(a_i,x)$ 或者 $a_i=\min(a_i,x)$.

HDU5306 Gorgeous Sequence[2]

维护一个序列 a，执行以下操作：

1. `0 l r t` $\forall l\le i\le r,\ a_i=\min(a_i,t)$.
2. `1 l r` 输出 $\max\limits_{i=l}^{r} a_i$.
3. `2 l r` 输出 $\sum\limits_{i=l}^{r} a_i$.

多组测试数据，保证 $T\le 100,\ \sum n,\sum m\le 10^6$.

区间取 min，意味着只对那些大于 t 的数有更改．因此这个操作的对象不再是整个区间，而是「这个区间中大于 t 的数」．于是我们可以有这样的思路：每个结点维护该区间的最大值 Max、次大值 Se、区间和 Sum 以及最大值的个数 Cnt．接下来我们考虑区间对 t 取 min 的操作．

1. 如果 $Max\le t$，显然这个 t 是没有意义的，直接返回；
2. 如果 $Se<t<Max$，那么这个 t 就能更新当前区间中的最大值．于是我们让区间和加上 $Cnt(t-Max)$，然后更新 Max 为 t，并打一个标记．
3. 如果 $t\le Se$，那么这时你发现你不知道有多少个数涉及到更新的问题．于是我们的策略就是，暴力递归向下操作．然后上传信息．

这个算法的复杂度如何？使用势能分析法可以得到复杂度是 $O(m\log n)$ 的．具体分析过程见论文．

```cpp
#include <algorithm>
#include <cctype>
#include <iostream>
using namespace std;
constexpr int N = 1e6 + 6;

int t, n, m;
int a[N];
int mx[N << 2], se[N << 2], cn[N << 2], tag[N << 2];
long long sum[N << 2];

void pushup(int u) {  // 向上更新标记
  const int ls = u << 1, rs = u << 1 | 1;
  sum[u] = sum[ls] + sum[rs];
  if (mx[ls] == mx[rs]) {
    mx[u] = mx[rs];
    se[u] = max(se[ls], se[rs]);
    cn[u] = cn[ls] + cn[rs];
  } else if (mx[ls] > mx[rs]) {
    mx[u] = mx[ls];
    se[u] = max(se[ls], mx[rs]);
    cn[u] = cn[ls];
  } else {
    mx[u] = mx[rs];
    se[u] = max(mx[ls], se[rs]);
    cn[u] = cn[rs];
  }
}

void pushtag(int u, int tg) {  // 单纯地打标记，不暴搜
```

```
  if (mx[u] <= tg) return;
  sum[u] += (1ll * tg - mx[u]) * cn[u];
  mx[u] = tag[u] = tg;
}

void pushdown(int u) {  // 下传标记
  if (tag[u] == -1) return;
  pushtag(u << 1, tag[u]), pushtag(u << 1 | 1, tag[u]);
  tag[u] = -1;
}

void build(int u = 1, int l = 1, int r = n) {  // 建树
  tag[u] = -1;
  if (l == r) {
    sum[u] = mx[u] = a[l], se[u] = -1, cn[u] = 1;
    return;
  }
  int mid = (l + r) >> 1;
  build(u << 1, l, mid), build(u << 1 | 1, mid + 1, r);
  pushup(u);
}

void modify_min(int L, int R, int v, int u = 1, int l = 1, int r = n) {
  if (mx[u] <= v) return;
  if (L <= l && r <= R && se[u] < v) return pushtag(u, v);
  int mid = (l + r) >> 1;
  pushdown(u);
  if (L <= mid) modify_min(L, R, v, u << 1, l, mid);
  if (mid < R) modify_min(L, R, v, u << 1 | 1, mid + 1, r);
  pushup(u);
}

int query_max(int L, int R, int u = 1, int l = 1, int r = n) {  // 查询最值
  if (L <= l && r <= R) return mx[u];
  int mid = (l + r) >> 1, r1 = -1, r2 = -1;
  pushdown(u);
  if (L <= mid) r1 = query_max(L, R, u << 1, l, mid);
  if (mid < R) r2 = query_max(L, R, u << 1 | 1, mid + 1, r);
  return max(r1, r2);
}

long long query_sum(int L, int R, int u = 1, int l = 1, int r = n) {  // 数值
  if (L <= l && r <= R) return sum[u];
  int mid = (l + r) >> 1;
  long long res = 0;
  pushdown(u);
  if (L <= mid) res += query_sum(L, R, u << 1, l, mid);
  if (mid < R) res += query_sum(L, R, u << 1 | 1, mid + 1, r);
  return res;
}

void go() {  // 根据题意
  cin >> n >> m;
  for (int i = 1; i <= n; i++) cin >> a[i];
```

```
  build();
  for (int i = 1; i <= m; i++) {
    int op, x, y, z;
    cin >> op >> x >> y;
    if (op == 0)
      cin >> z, modify_min(x, y, z);
    else if (op == 1)
      cout << query_max(x, y) << '\n';
    else
      cout << query_sum(x, y) << '\n';
  }
}

signed main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  cin >> t;
  while (t--) go();
  return 0;
}
```

BZOJ4695 最假女选手[3]

维护一个序列 a，执行以下操作：

1. `1 l r x` $\forall l \le i \le r,\ a_i = a_i + x$.
2. `2 l r x` $\forall l \le i \le r,\ a_i = \max(a_i, x)$.
3. `3 l r x` $\forall l \le i \le r,\ a_i = \min(a_i, x)$.
4. `4 l r` 输出 $\sum_{i=l}^{r} a_i$.
5. `5 l r` 输出 $\max_{i=l}^{r} a_i$.
6. `6 l r` 输出 $\min_{i=l}^{r} a_i$.

$n, m \le 5 \times 10^5,\ |a_i| \le 10^8$. 所有类型 1 操作有 $|x| \le 10^3$，其余操作满足 $|x| \le 10^8$.

同样的方法，我们维护最大、次大、最大个数、最小、次小、最小个数、区间和. 除了这些信息，我们还需要维护区间 max、区间 min、区间加的标记. 相比上一道题，这就涉及到标记下传的顺序问题了. 我们采用这样的策略：

1. 我们认为区间加的标记是最优先的，其余两种标记地位平等.
2. 对一个结点加上一个 v 标记，除了用 v 更新卫星信息和当前结点的区间加标记外，我们用这个 v 更新区间 max 和区间 min 的标记.
3. 对一个结点取 v 的 min（这里忽略暴搜的过程，假定标记满足添加的条件），除了更新卫星信息，我们要与区间 max 的标记做比较. 如果 v 小于区间 max 的标记，则所有的数最后都会变成 v，那么把区间 max 的标记也变成 v. 否则不管.
4. 区间取 v 的 max 同理.

在维护信息的时候，当只有一个数或两个数的时候可能发生数集重合，比如一个数既是最大值又是次小值，需要特判.

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

constexpr int N = 5e5 + 5, SZ = N << 2, INF = 0x7fffffff;

int n, m;
```

```
int a[N];

struct data {
  int mx, mx2, mn, mn2, cmx, cmn, tmx, tmn, tad;
  long long sum;
} t[SZ];

void pushup(int u) {
  const int lu = u << 1, ru = u << 1 | 1;
  t[u].sum = t[lu].sum + t[ru].sum;
  if (t[lu].mx == t[ru].mx) {
    t[u].mx = t[lu].mx, t[u].cmx = t[lu].cmx + t[ru].cmx;
    t[u].mx2 = max(t[lu].mx2, t[ru].mx2);
  } else if (t[lu].mx > t[ru].mx) {
    t[u].mx = t[lu].mx, t[u].cmx = t[lu].cmx;
    t[u].mx2 = max(t[lu].mx2, t[ru].mx);
  } else {
    t[u].mx = t[ru].mx, t[u].cmx = t[ru].cmx;
    t[u].mx2 = max(t[lu].mx, t[ru].mx2);
  }
  if (t[lu].mn == t[ru].mn) {
    t[u].mn = t[lu].mn, t[u].cmn = t[lu].cmn + t[ru].cmn;
    t[u].mn2 = min(t[lu].mn2, t[ru].mn2);
  } else if (t[lu].mn < t[ru].mn) {
    t[u].mn = t[lu].mn, t[u].cmn = t[lu].cmn;
    t[u].mn2 = min(t[lu].mn2, t[ru].mn);
  } else {
    t[u].mn = t[ru].mn, t[u].cmn = t[ru].cmn;
    t[u].mn2 = min(t[lu].mn, t[ru].mn2);
  }
}

void push_add(int u, int l, int r, int v) {
  // 更新加法标记的同时，更新 $\min$ 和 $\max$ 标记
  t[u].sum += (r - l + 1ll) * v;
  t[u].mx += v, t[u].mn += v;
  if (t[u].mx2 != -INF) t[u].mx2 += v;
  if (t[u].mn2 != INF) t[u].mn2 += v;
  if (t[u].tmx != -INF) t[u].tmx += v;
  if (t[u].tmn != INF) t[u].tmn += v;
  t[u].tad += v;
}

void push_min(int u, int tg) {
  // 注意比较 $\max$ 标记
  if (t[u].mx <= tg) return;
  t[u].sum += (tg * 1ll - t[u].mx) * t[u].cmx;
  if (t[u].mn2 == t[u].mx) t[u].mn2 = tg;  // !!!
  if (t[u].mn == t[u].mx) t[u].mn = tg;    // !!!!!
  if (t[u].tmx > tg) t[u].tmx = tg;        // 更新取 $\max$ 标记
  t[u].mx = tg, t[u].tmn = tg;
}

void push_max(int u, int tg) {
```

```
  if (t[u].mn > tg) return;
  t[u].sum += (tg * 1ll - t[u].mn) * t[u].cmn;
  if (t[u].mx2 == t[u].mn) t[u].mx2 = tg;
  if (t[u].mx == t[u].mn) t[u].mx = tg;
  if (t[u].tmn < tg) t[u].tmn = tg;
  t[u].mn = tg, t[u].tmx = tg;
}

void pushdown(int u, int l, int r) {
  const int lu = u << 1, ru = u << 1 | 1, mid = (l + r) >> 1;
  if (t[u].tad)
    push_add(lu, l, mid, t[u].tad), push_add(ru, mid + 1, r, t[u].tad);
  if (t[u].tmx != -INF) push_max(lu, t[u].tmx), push_max(ru, t[u].tmx);
  if (t[u].tmn != INF) push_min(lu, t[u].tmn), push_min(ru, t[u].tmn);
  t[u].tad = 0, t[u].tmx = -INF, t[u].tmn = INF;
}

void build(int u = 1, int l = 1, int r = n) {
  t[u].tmn = INF, t[u].tmx = -INF;  // 取极限
  if (l == r) {
    t[u].sum = t[u].mx = t[u].mn = a[l];
    t[u].mx2 = -INF, t[u].mn2 = INF;
    t[u].cmx = t[u].cmn = 1;
    return;
  }
  int mid = (l + r) >> 1;
  build(u << 1, l, mid), build(u << 1 | 1, mid + 1, r);
  pushup(u);
}

void add(int L, int R, int v, int u = 1, int l = 1, int r = n) {
  if (R < l || r < L) return;
  if (L <= l && r <= R) return push_add(u, l, r, v);  // !!! 忘 return
  int mid = (l + r) >> 1;
  pushdown(u, l, r);
  add(L, R, v, u << 1, l, mid), add(L, R, v, u << 1 | 1, mid + 1, r);
  pushup(u);
}

void tomin(int L, int R, int v, int u = 1, int l = 1, int r = n) {
  if (R < l || r < L || t[u].mx <= v) return;
  if (L <= l && r <= R && t[u].mx2 < v) return push_min(u, v);  // BUG: 忘了返回
  int mid = (l + r) >> 1;
  pushdown(u, l, r);
  tomin(L, R, v, u << 1, l, mid), tomin(L, R, v, u << 1 | 1, mid + 1, r);
  pushup(u);
}

void tomax(int L, int R, int v, int u = 1, int l = 1, int r = n) {
  if (R < l || r < L || t[u].mn >= v) return;
  if (L <= l && r <= R && t[u].mn2 > v) return push_max(u, v);
  int mid = (l + r) >> 1;
  pushdown(u, l, r);
  tomax(L, R, v, u << 1, l, mid), tomax(L, R, v, u << 1 | 1, mid + 1, r);
```

```
  pushup(u);
}

long long qsum(int L, int R, int u = 1, int l = 1, int r = n) {
  if (R < l || r < L) return 0;
  if (L <= l && r <= R) return t[u].sum;
  int mid = (l + r) >> 1;
  pushdown(u, l, r);
  return qsum(L, R, u << 1, l, mid) + qsum(L, R, u << 1 | 1, mid + 1, r);
}

long long qmax(int L, int R, int u = 1, int l = 1, int r = n) {
  if (R < l || r < L) return -INF;
  if (L <= l && r <= R) return t[u].mx;
  int mid = (l + r) >> 1;
  pushdown(u, l, r);
  return max(qmax(L, R, u << 1, l, mid), qmax(L, R, u << 1 | 1, mid + 1, r));
}

long long qmin(int L, int R, int u = 1, int l = 1, int r = n) {
  if (R < l || r < L) return INF;
  if (L <= l && r <= R) return t[u].mn;
  int mid = (l + r) >> 1;
  pushdown(u, l, r);
  return min(qmin(L, R, u << 1, l, mid), qmin(L, R, u << 1 | 1, mid + 1, r));
}

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  cin >> n;
  for (int i = 1; i <= n; i++) cin >> a[i];
  build();
  cin >> m;
  for (int i = 1; i <= m; i++) {
    int op, l, r, x;
    cin >> op >> l >> r;
    if (op <= 3) cin >> x;
    if (op == 1)
      add(l, r, x);
    else if (op == 2)
      tomax(l, r, x);
    else if (op == 3)
      tomin(l, r, x);
    else if (op == 4)
      cout << qsum(l, r) << '\n';
    else if (op == 5)
      cout << qmax(l, r) << '\n';
    else
      cout << qmin(l, r) << '\n';
  }
  return 0;
}
```

吉老师证出来这个算法的复杂度是 $O(m \log^2 n)$ 的.

Mzl loves segment tree

两个序列 A, B，一开始 B 中的数都是 0．维护的操作是：

1. 对 A 做区间取 min
2. 对 A 做区间取 max
3. 对 A 做区间加
4. 询问 B 的区间和

每次操作完后，如果 A_i 的值发生变化，就给 B_i 加 1．$n, m \leq 3 \times 10^5$．

先考虑最容易的区间加操作．只要 $x \neq 0$ 那么整个区间的数都变化，所以给 B 作一次区间加即可．

对于区间取最值的操作，你发现你打标记与下传标记是与 B 数组一一对应的．本质上你将序列的数分成三类：最大值、最小值、非最值．并分别维护（只不过你没有建出具体的最值集合而已，但这并不妨碍维护的操作）．因此在打标记的时候顺便给 B 更新信息即可（注意不是给 B 打标记！是更新信息！）．查询的时候，你在 A 上查询，下传标记的时候顺便给 B 更新信息．找到需要的结点后，返回 B 的信息即可．这种操作本质上就是把最值的信息拿给 B 去维护了．另外仍要处理数集的重复问题．

CTSN loves segment tree[4]

维护两个序列 a, b，执行以下操作：

1. `1 l r x` $\forall l \leq i \leq r,\ a_i = \min(a_i, x)$.
2. `2 l r x` $\forall l \leq i \leq r,\ b_i = \min(b_i, x)$.
3. `3 l r x` $\forall l \leq i \leq r,\ a_i = a_i + x$.
4. `4 l r x` $\forall l \leq i \leq r,\ b_i = b_i + x$.
5. `5 l r` 输出 $\max_{i=l}^{r}(a_i + b_i)$.

$n, m \leq 3 \times 10^5,\ |a_i|, |b_i|, |x| \leq 10^9$.

我们把区间 $[l, r]$ 中的备选答案 $A_i + B_i$ 分成四类：A_i, B_i 均不是序列 A, B 区间最大值、A_i 是序列 A 区间最大值但是 B_i 不是序列 B 区间最大值、A_i 不是序列 A 区间最大值但是 B_i 是序列 B 区间最大值、A_i, B_i 均是序列 A, B 区间最大值．我们不妨分别设为 $C_{0,0}, C_{1,0}, C_{0,1}, C_{1,1}$．此外我们正常维护序列 A, B 的区间最大值和次大值．下传区间加法标记和 min 标记时对 A, B 最大值和次大值的处理与上述两个例题一致．对 A 的 min 标记会影响到 $C_{1,1}$ 和 $C_{1,0}$，对 B 的标记会影响到 $C_{1,1}$ 和 $C_{0,1}$．对 A, B 的加法则会对 $C_{0,0}, C_{1,0}, C_{0,1}, C_{1,1}$ 均产生影响．只需要注意 $C_{0,0}, C_{1,0}, C_{0,1}$ 不存在的边界情况即可（例如区间 $[i, i]$ 只有 A, B 的最大值与 $C_{1,1}$ 存在）．

接下来需要考虑在 pushup 时如何维护 $C_{0,0}, C_{1,0}, C_{0,1}, C_{1,1}$．我们可以考虑一下完成 A, B 最大值的更新之后，讨论左右儿子的 A, B 最大值是否与当前节点 A, B 最大值相等．我们以左儿子为例进行讲解，右儿子类似处理：

- 当左儿子的 A, B 最大值与当前节点的 A, B 最大值均相等时，左儿子的 $C_{0,0}, C_{1,0}, C_{0,1}, C_{1,1}$ 会分别对当前节点的 $C_{0,0}, C_{1,0}, C_{0,1}, C_{1,1}$ 产生贡献．
- 当左儿子的 A 最大值与当前节点 A 最大值相等，但是 B 最大值不相等时，左儿子的 $C_{1,0}, C_{1,1}$ 会对该节点的 $C_{1,0}$ 产生贡献，$C_{0,0}, C_{0,1}$ 会对该节点的 $C_{0,0}$ 产生贡献．
- 当左儿子的 A 最大值与当前节点 A 最大值不相等，但是 B 最大值相等时，左儿子的 $C_{0,1}, C_{1,1}$ 会对该节点的 $C_{0,1}$ 产生贡献，$C_{0,0}, C_{1,0}$ 会对该节点的 $C_{0,0}$ 产生贡献．
- 当左儿子的 A, B 最大值与当前节点的 A, B 最大值均不相等时，左儿子的 $C_{0,0}, C_{1,0}, C_{0,1}, C_{1,1}$ 仅会对该节点的 $C_{0,0}$ 产生贡献．

区间查询结果的 $\max(C_{0,0}, C_{1,0}, C_{0,1}, C_{1,1})$ 即为所求．

由于需要同时维护区间 min 和区间加法，所以复杂度仍是 $O(m \log^2 n)$．

```cpp
#include <algorithm>
#include <cstdint>
#include <iostream>
using i64 = int64_t;

constexpr int N = 300010;
constexpr i64 INF = 1145141919810114514ll;
int n, q, op, l, r;
int a[N], b[N];

int lc(int u) { return (u << 1); }

int rc(int u) { return (u << 1) | 1; }

struct node {
  i64 mx_ab[2][2];  // 0: not max A or B; 1: is max A or B

  i64 max_val() {
    return std::max({mx_ab[0][0], mx_ab[0][1], mx_ab[1][0], mx_ab[1][1]});
  }

  i64 mx1[2], mx2[2];  // 0: A; 1: B
  i64 tag_add[2], tag_mn[2];

  void clr_tag() { tag_add[0] = tag_add[1] = 0, tag_mn[0] = tag_mn[1] = INF; }

  void init(i64 a, i64 b) {
    mx1[0] = a, mx1[1] = b, mx2[0] = mx2[1] = -INF;
    mx_ab[1][1] = a + b, mx_ab[0][1] = mx_ab[1][0] = mx_ab[0][0] = -INF;
  }

  void modify_mn(i64 tg, bool id)  // id 0: A; 1: B
  {
    if (mx1[id] <= tg) return;
    mx_ab[1][1] += (mx_ab[1][1] > -INF) ? (tg - mx1[id]) : 0;
    mx_ab[!id][id] += (mx_ab[!id][id] > -INF) ? (tg - mx1[id]) : 0;
    mx1[id] = tag_mn[id] = tg;
  }

  void modify_add(i64 tg, bool id)  // id 0: A; 1: B
  {
    for (int i = 0; i < 2; ++i)
      for (int j = 0; j < 2; ++j) mx_ab[i][j] += (mx_ab[i][j] > -INF) ? tg : 0;
    mx1[id] += tg, mx2[id] += (mx2[id] > -INF) ? tg : 0;
    tag_mn[id] += (tag_mn[id] < INF) ? tg : 0, tag_add[id] += tg;
  }

  void pushup(const node& l, const node& r) {
    for (int i = 0; i < 2; ++i) {
      if (l.mx1[i] == r.mx1[i])
        mx1[i] = l.mx1[i], mx2[i] = std::max(l.mx2[i], r.mx2[i]);
      else if (l.mx1[i] > r.mx1[i])
        mx1[i] = l.mx1[i], mx2[i] = std::max(l.mx2[i], r.mx1[i]);
      else
```

```
55          mx1[i] = r.mx1[i], mx2[i] = std::max(l.mx1[i], r.mx2[i]);
56      }
57      if (l.mx1[0] == mx1[0] && l.mx1[1] == mx1[1]) {
58        mx_ab[1][1] = l.mx_ab[1][1], mx_ab[0][1] = l.mx_ab[0][1];
59        mx_ab[1][0] = l.mx_ab[1][0], mx_ab[0][0] = l.mx_ab[0][0];
60      } else if (l.mx1[0] == mx1[0] && l.mx1[1] != mx1[1]) {
61        mx_ab[1][1] = mx_ab[0][1] = -INF;
62        mx_ab[1][0] = std::max(l.mx_ab[1][1], l.mx_ab[1][0]);
63        mx_ab[0][0] = std::max(l.mx_ab[0][1], l.mx_ab[0][0]);
64      } else if (l.mx1[0] != mx1[0] && l.mx1[1] == mx1[1]) {
65        mx_ab[1][1] = mx_ab[1][0] = -INF;
66        mx_ab[0][1] = std::max(l.mx_ab[1][1], l.mx_ab[0][1]);
67        mx_ab[0][0] = std::max(l.mx_ab[1][0], l.mx_ab[0][0]);
68      } else {
69        mx_ab[1][1] = mx_ab[0][1] = mx_ab[1][0] = -INF;
70        mx_ab[0][0] = std::max(
71            {l.mx_ab[0][0], l.mx_ab[0][1], l.mx_ab[1][0], l.mx_ab[1][1]});
72      }
73
74      if (r.mx1[0] == mx1[0] && r.mx1[1] == mx1[1]) {
75        mx_ab[1][1] = std::max(mx_ab[1][1], r.mx_ab[1][1]);
76        mx_ab[0][1] = std::max(mx_ab[0][1], r.mx_ab[0][1]);
77        mx_ab[1][0] = std::max(mx_ab[1][0], r.mx_ab[1][0]);
78        mx_ab[0][0] = std::max(mx_ab[0][0], r.mx_ab[0][0]);
79      } else if (r.mx1[0] == mx1[0] && r.mx1[1] != mx1[1]) {
80        mx_ab[1][0] = std::max({mx_ab[1][0], r.mx_ab[1][1], r.mx_ab[1][0]});
81        mx_ab[0][0] = std::max({mx_ab[0][0], r.mx_ab[0][1], r.mx_ab[0][0]});
82      } else if (r.mx1[0] != mx1[0] && r.mx1[1] == mx1[1]) {
83        mx_ab[0][1] = std::max({mx_ab[0][1], r.mx_ab[1][1], r.mx_ab[0][1]});
84        mx_ab[0][0] = std::max({mx_ab[0][0], r.mx_ab[1][0], r.mx_ab[0][0]});
85      } else
86        mx_ab[0][0] = std::max({mx_ab[0][0], r.mx_ab[0][0], r.mx_ab[0][1],
87                                r.mx_ab[1][0], r.mx_ab[1][1]});
88    }
89  } tr[N << 2];
90
91  void pushup(int u) { tr[u].pushup(tr[lc(u)], tr[rc(u)]); }
92
93  void pushdown(int u) {
94    for (int i = 0; i < 2; ++i)
95      if (tr[u].tag_add[i])
96        tr[lc(u)].modify_add(tr[u].tag_add[i], i),
97            tr[rc(u)].modify_add(tr[u].tag_add[i], i);
98    for (int i = 0; i < 2; ++i)
99      if (tr[u].tag_mn[i] < INF)
100       tr[lc(u)].modify_mn(tr[u].tag_mn[i], i),
101           tr[rc(u)].modify_mn(tr[u].tag_mn[i], i);
102   tr[u].clr_tag();
103 }
104
105 void _build(int u, int L, int R) {
106   tr[u].clr_tag();
107   if (L == R) return tr[u].init(a[L], b[L]);
108   int M = (L + R) >> 1;
```

```
  _build(lc(u), L, M), _build(rc(u), M + 1, R), pushup(u);
}

void build() { _build(1, 1, n); }

void _modify_mn(int u, int l, int r, int L, int R, int v, bool i) {
  if (L > r || R < l || tr[u].mx1[i] <= v) return;
  if (l <= L && R <= r && tr[u].mx2[i] < v) return tr[u].modify_mn(v, i);
  pushdown(u);
  int M = (L + R) >> 1;
  _modify_mn(lc(u), l, r, L, M, v, i), _modify_mn(rc(u), l, r, M + 1, R, v, i);
  pushup(u);
}

void modify_mn(int l, int r, int v, bool i) { _modify_mn(1, l, r, 1, n, v, i); }

void _modify_add(int u, int l, int r, int L, int R, int v, bool i) {
  if (L > r || R < l) return;
  if (l <= L && R <= r) return tr[u].modify_add(v, i);
  pushdown(u);
  int M = (L + R) >> 1;
  _modify_add(lc(u), l, r, L, M, v, i),
      _modify_add(rc(u), l, r, M + 1, R, v, i);
  pushup(u);
}

void modify_add(int l, int r, int v, bool i) {
  _modify_add(1, l, r, 1, n, v, i);
}

node _query(int u, int l, int r, int L, int R) {
  if (l <= L && R <= r) return tr[u];
  pushdown(u);
  int M = (L + R) >> 1;
  if (r <= M) return _query(lc(u), l, r, L, M);
  if (l > M) return _query(rc(u), l, r, M + 1, R);
  node ret;
  node _l = _query(lc(u), l, M, L, M), _r = _query(rc(u), M + 1, r, M + 1, R);
  ret.pushup(_l, _r);
  return ret;
}

node query(int l, int r) { return _query(1, l, r, 1, n); }

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

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  cin >> n >> q;
  for (int i = 1; i <= n; ++i) cin >> a[i];
  for (int i = 1; i <= n; ++i) cin >> b[i];
  build();
  while (q--) {
```

```
163     cin >> op >> l >> r;
164     int x;
165     switch (op) {
166       case 1:
167         cin >> x;
168         modify_mn(l, r, x, 0);
169         break;
170       case 2:
171         cin >> x;
172         modify_mn(l, r, x, 1);
173         break;
174       case 3:
175         cin >> x;
176         modify_add(l, r, x, 0);
177         break;
178       case 4:
179         cin >> x;
180         modify_add(l, r, x, 1);
181         break;
182       case 5:
183         cout << query(l, r).max_val() << '\n';
184         break;
185       default:
186         break;
187     }
188   }
189 }
190
```

小结

在第本章节中我们给出了四道例题，分别讲解了基本区间最值操作的维护、多个标记的优先级处理、数集分类的思想以及多个分类的维护．本质上处理区间最值的基本思想就是数集信息的分类维护与高效合并．在下一章节中，我们将探讨历史区间最值的相关问题．

历史最值问题

历史最值不等于可持久化

注意，本章所讲到的历史最值问题不同于所谓的可持久化数据结构．这类特殊的问题我们将其称为历史最值问题．历史最值的问题可以分为三类．

历史最大值

简单地说，一个位置的历史最大值就是当前位置下曾经出现过的数的最大值．形式化地定义，我们定义一个辅助数组 B，一开始与 A 完全相同．在 A 的每次操作后，我们对整个数组取 max：

$$\forall i \in [1,n],\ B_i = \max(B_i, A_i)$$

这时，我们将 B_i 称作这个位置的历史最大值，

历史最小值

定义与历史最大值类似，在 A 的每次操作后，我们对整个数组取 min．这时，我们将 B_i 称作这个位置的历史最小值，

历史版本和

辅助数组 B 一开始全部是 0．在每一次操作后，我们把整个 A 数组累加到 B 数组上

$$\forall i \in [1,n],\ B_i = B_i + A_i$$

我们称 B_i 为 i 这个位置上的历史版本和.

接下来，我们将历史最值问题分成四类讨论.

可以用标记处理的问题

CPU 监控[5]

序列 A, B 一开始相同:

1. 对 A 做区间覆盖 x
2. 对 A 做区间加 x
3. 询问 A 的区间 max
4. 询问 B 的区间 max

每次操作后，我们都进行一次更新，$\forall i \in [1,n],\ B_i = \max(B_i, A_i)$. $n, m \leq 10^5$.

我们先不考虑操作 1. 那么只有区间加的操作，我们维护标记 Add 表示当前区间增加的值，这个标记可以解决区间 max 的问题. 接下来考虑历史区间 max. 我们定义标记 Pre，该标记的含义是：在该标记的生存周期内，Add 标记的历史最大值.

这个定义可能比较模糊. 因此我们先解释一下标记的生存周期. 一个标记会经历这样的过程:

1. 在结点 u 被建立.
2. 在结点 u 接受若干个新的标记的同时，与新的标记合并（指同类标记）
3. 结点 u 的标记下传给 u 的儿子，u 的标记清空

我们认为在这个过程中，从 1 开始到 3 之前，都是结点 u 的标记的生存周期. 两个标记合并后，成为同一个标记，那么他们的生存周期也会合并（即取建立时间较早的那个做为生存周期的开始）. 一个与之等价的说法是，从上次把这个结点的标记下传的时刻到当前时刻这一时间段.

为什么要定义生存周期?利用这个概念, 我们可以证明: 在一个结点标记的生存周期内, 其子结点均不会发生任何变化, 并保留在这个生存周期之前的状态. 道理很简单, 因为在这个期间你是没有下传标记的.

于是，你就可以保证，在当前标记生存周期内的历史 Add 的最大值是可以更新到子结点的标记和信息上的. 因为子结点的标记和信息在这个时间段内都没有变过. 于是我们把 u 的标记下传给它的儿子 s, 不难发现

$$Pre_s = \max(Pre_s, Pre_u + Add_s), Add_s = Add_u + Add_s$$

那么信息的更新也是类似的，拿对应的标记更新即可.

接下来，我们考虑操作 1.

区间覆盖操作，会把所有的数变成一个数. 在这之后，无论是区间加减还是覆盖，整个区间的数仍是同一个（除非你结束当前标记的生存周期，下传标记）. 因此我们可以把第一次区间覆盖后的所有标记都看成区间覆盖标记. 也就是说一个标记的生存周期被大致分成两个阶段:

1. 若干个加减操作标记的合并，没有接收过覆盖标记.
2. 覆盖操作的标记，没有所谓的加减标记（加减标记转化为覆盖标记）

于是我们把这个结点的 Pre 标记拆成 (P_1, P_2). P_1 表示第一阶段的最大加减标记; P_2 表示第二阶段的最大覆盖标记. 利用相似的方法, 我们可以对这个做标记下传和信息更新. 时间复杂度是 $O(m \log n)$ 的（这个问题并没有区间对 x 取最值的操作哦～）

```cpp
#include <algorithm>
#include <climits>
#include <iostream>
using namespace std;
using ll = long long;
```

```
6
7  constexpr int N = 1e5 + 7;
8
9  struct Tree {
10   int mx, _mx;  // 区间最大值 区间历史最大值
11   int ad, _ad;  // 区间加标记 区间阶段历史最大加标记
12   int st, _st;  // 区间修改值 区间阶段历史最大修改标记
13 } g[N * 4];
14
15 int a[N];
16 int n, m;
17 #define ls u * 2
18 #define rs u * 2 + 1
19 #define mid (l + r) / 2
20
21 void pushup(int u) {
22   g[u].mx = max(g[ls].mx, g[rs].mx);
23   g[u]._mx = max(g[ls]._mx, g[rs]._mx);
24 }
25
26 void pushadd(int u, int v, int _v) {
27   g[u]._mx = max(g[u]._mx, g[u].mx + _v), g[u].mx += v;
28   if (g[u].st == INT_MIN)
29     g[u]._ad = max(g[u]._ad, g[u].ad + _v), g[u].ad += v;
30   else
31     g[u]._st = max(g[u]._st, g[u].st + _v), g[u].st += v;
32 }
33
34 void pushset(int u, int v, int _v) {
35   g[u]._mx = max(g[u]._mx, _v), g[u].mx = v;
36   g[u]._st = max(g[u]._st, _v), g[u].st = v;
37 }
38
39 void pushdown(int u, int l, int r) {
40   if (g[u].ad || g[u]._ad)
41     pushadd(ls, g[u].ad, g[u]._ad), pushadd(rs, g[u].ad, g[u]._ad),
42         g[u].ad = g[u]._ad = 0;
43   if (g[u].st != INT_MIN || g[u]._st != INT_MIN)
44     pushset(ls, g[u].st, g[u]._st), pushset(rs, g[u].st, g[u]._st),
45         g[u].st = g[u]._st = INT_MIN;
46 }
47
48 void build(int u = 1, int l = 1, int r = n) {
49   g[u]._st = g[u].st = INT_MIN;
50   if (l == r) {
51     g[u].mx = a[l];
52     g[u]._mx = a[l];
53     return;
54   }
55   build(ls, l, mid), build(rs, mid + 1, r);
56   pushup(u);
57 }
58
59 int L, R;
```

```
60
61 void add(int v, int u = 1, int l = 1, int r = n) {
62   if (R < l || r < L) return;
63   if (L <= l && r <= R) return pushadd(u, v, max(v, 0));
64   pushdown(u, l, r);
65   add(v, ls, l, mid), add(v, rs, mid + 1, r);
66   pushup(u);
67 }
68
69 void tset(int v, int u = 1, int l = 1, int r = n) {
70   if (R < l || r < L) return;
71   if (L <= l && r <= R) return pushset(u, v, v);
72   pushdown(u, l, r);
73   tset(v, ls, l, mid), tset(v, rs, mid + 1, r);
74   pushup(u);
75 }
76
77 int qmax(int u = 1, int l = 1, int r = n) {
78   if (R < l || r < L) return INT_MIN;
79   if (L <= l && r <= R) return g[u].mx;
80   pushdown(u, l, r);
81   return max(qmax(ls, l, mid), qmax(rs, mid + 1, r));
82 }
83
84 int qmaxh(int u = 1, int l = 1, int r = n) {
85   if (R < l || r < L) return INT_MIN;
86   if (L <= l && r <= R) return g[u]._mx;
87   pushdown(u, l, r);
88   return max(qmaxh(ls, l, mid), qmaxh(rs, mid + 1, r));
89 }
90
91 int main() {
92   cin.tie(nullptr)->sync_with_stdio(false);
93   cin >> n;
94   for (int i = 1; i <= n; ++i) cin >> a[i];
95   build();
96   int m, z;
97   cin >> m;
98   for (int i = 1; i <= m; ++i) {
99     char op;
100    cin >> op;
101    while (op == ' ' || op == '\r' || op == '\n') cin >> op;
102    cin >> L >> R;
103    int x;
104    if (op == 'Q')
105      cout << qmax() << '\n';
106    else if (op == 'A')
107      cout << qmaxh() << '\n';
108    else if (op == 'P')
109      cin >> x, add(x);
110    else
111      cin >> x, tset(x);
112  }
113  return 0;
```

```
114 }
115
```

外部参考资料

[1] 2016 年国家集训队论文 

[2] HDU5306 Gorgeous Sequence 

[3] BZOJ4695 最假女选手 

[4] CTSN loves segment tree 

[5] CPU 监控 

10.13.6 Kinetic Tournament Tree

Authors: Jerry3128

前置知识：线段树→10.13.1@p2098

问题引入

给定单变量线性函数序列 $F = \{f_1, ..., f_n\}$：$f_i : \mathbf{R} \to \mathbf{R}$. 其中 $f_i(x) = k_i x + b_i$ 且 $k_i, b_i \in \mathbf{R}$. 我们需要维护以下操作：

- QueryMax(l, r)：给定 l 和 r, 返回 $\max_{i=l}^{r} f_i(0)$.
- TranslateLeft(l, r, δ)：给定 l,r 和 δ, 对于所有 $i \in [l, r]$，执行操作 $f_i(x) \leftarrow f_i(x + \delta)$，这个操作等价于执行 $b_i \leftarrow b_i + k_i\delta$. 其中 $\delta > 0$.

为了方便，我们假定所有函数互不相同.

一次函数区间向左平移的本质是 $b_i \leftarrow k_i \cdot \delta$，也就是对于 b_i 常数项，让它加上斜率 k_i 乘上横坐标的平移量 δ；这种操作等价于我们在许多数据结构问题中遇到的「按位置系数加权区间加」：即对于区间 $[l, r]$ 内的每个下标 i，给其值加上一个固定的数 δ 乘上该位置特有的系数 k_i. 因此本质上一次函数区间平移就是所谓的，按位置系数加权区间加法.

为了展示 KTT 独特的二叉树形分治结构，我们将直接从区间平移入手.

Kinetic Data Structures

Kinetic Data Structures 简称 KDS. KDS 用于维护几何对象系统在连续运动过程中的属性.

事件队列

我们假设每一个点都有一个已知的运动计划, 这个计划可以提供它的完整或者部分运动信息, 例如函数 $f_i(x)$ 形成的曲线或直线能够很好的描述动点 i 的运动轨迹. 运动计划随时可能变化，它可能是由于碰撞，或者环境交互的原因；我们称造成运动计划更改的原因为事件. 事件队列会按时间顺序给出事件.

KDS 的一个关键方面是需要拥有容易维护的事件, 即事件队列中事件类型对应于可能的组合变化, 这些变化涉及数量恒定且通常较少的物体. 例如，在本题的维护中，我们使用的一种事件类型是「函数 $f_i(0)$ 与函数 $f_j(0)$ 的大小发生变化」.

事件队列可以隐式维护.

证书

这些事件应当可以等价于通过一系列低阶代数条件的交保证，每个代数条件都涉及有限数量的对象. 我们将这些条件称为 KDS 的证书. 例如 $[f_i(0) > f_j(0)]$.

Kinetic Tournament Tree

简介

Kinetic Tournament Tree（简称 KTT），属于 Kinetic Data Structures，首次出现于 1999 年的 Data Structures for Mobile Data[1]，用于维护连续变化的数据．更普遍地，每一个采用如下动态化策略（kinetization strategy）的结构，都可以称为 Kinetic Tournament：

- 为静态算法中的关键操作（例如比较）生成正确性证书，并将每个证书与一个全局事件队列关联，记录该证书可能失效的时间点．
- 当某个证书失效时，我们能够高效地更新算法输出并维护证书集合．

在算法竞赛社区，它兴起于 2020 年国家集训队论文《浅谈函数最值的动态维护[2]》．学术界的 KTT 与算法竞赛界的 KTT 在应用领域和实现上有所不同，因此我们将介绍为算法竞赛界进行一些优化过后的 KTT．

基本结构

首先我们考虑设计一个线段树结构相似的数据结构用于维护静态最大值．我们将线段树的结构建立出来，对于每个非叶节点，它的权值为两个孩子节点中较大的权值．在执行了 $O(n)$ 次比较过后，我们得到根节点的权值就是全局的最大值．现在，权值开始变化．只要 KTT 能探测到每一次树上点的最大值来源改变，我们就能够维护全局最大值．

为了让 KTT 能够探测到每一次树上最大值来源变换，对于树上节点 x 以及其左、右儿子提供的函数 f_L 和 f_R，我们定义证书为「f_L 和 f_R 的大小关系保持不变」．当证书失效的时候，我们就需要通过树上路径走到当前证书失效的节点来更新它的信息．为了维护每个证书失效的时间，我们发现证书失效的时刻正是两个函数拥有相同值的时刻，那么问题就变成了找到两个线性函数的交点的横坐标，可以在 $O(1)$ 时间内解决．

对于每个树上的节点维护出了它在 0 处取到最大值的函数，以及当前证书失效的时间和整个子树内最早失效证书的失效时间；那么对于每个节点证书失效的时刻，我们就可以在这个时刻找到它并更新它的信息．这些信息是用来记录函数本身的信息的．接下来我们考虑维护区间平移操作，因为它是可以简单累加的，我们就可以使用懒标记去处理区间平移操作．

我们定义懒标记 Δ_v 表示 v 节点子树内的所有其他节点上的函数都应该向左平移 Δ_v 个单位．此时，对于树上节点 v，一个新的操作要将其子树内的所有函数区间向左平移 δ，即 $f(x) \leftarrow f(x+\delta)$．我们需要更新懒标记：$\Delta_v \leftarrow \Delta_v + \delta$，即将子树内所有其他节点的偏移量进行累加．同时向左平移也意味着 0 点函数值的变化．我们可以发现如果证书的失效横坐标为 t，那么平移后的失效横坐标应为 $t-\delta$；此时，如果 $t-\delta$ 越过 0 点，就代表着证书失效，我们就需要往下递归找到当前证书所在节点，更新当前节点，并将新的信息向上更新到根．这个过程可以跟随着修改一起做．

因此我们便可以得到简单实现．

参考实现

```
struct node {
  int l, r;
  int tag;   // the lazy propagation tag
  int k, b;  // the linear function
  int swc;   // the time of certificate violation
};

vector<node> v;

int IntegerPart(double x) {
```

```
  if (x >= 0 && x <= inf) return int(ceil(x));
  return inf;
}

void push_up(int rt) {
  int mx = v[rt << 1].b > v[rt << 1 | 1].b ? rt << 1 : rt << 1 | 1, mi = mx
^ 1;
  v[rt].k = v[mx].k, v[rt].b = v[mx].b;
  v[rt].swc = v[mx].k < v[mi].k
                  ? IntegerPart(1.0 * (v[mx].b - v[mi].b) / (v[mi].k
- v[mx].k))
                  : inf;
  v[rt].swc = min({v[rt].swc, v[mx].swc, v[mi].swc});
}

void push_tag(int rt, int val) {
  v[rt].tag += val, v[rt].swc -= val, v[rt].b += v[rt].k * val;
}

void push_down(int rt) {
  if (v[rt].tag)
    push_tag(rt << 1, v[rt].tag), push_tag(rt << 1 | 1, v[rt].tag),
        v[rt].tag = 0;
}

void checkswitch(int rt) {
  if (v[rt].l == v[rt].r) return;
  push_down(rt);
  if (v[rt].swc <= 0) checkswitch(rt << 1), checkswitch(rt << 1 | 1);
  push_up(rt);
}

void build(int rt, int l, int r) {
  v[rt].l = l, v[rt].r = r;
  if (l == r) {
    v[rt].k = k[l];
    v[rt].b = b[l];
    v[rt].swc = inf;
    return;
  }
  int mid = (l + r) >> 1;
  build(rt << 1, l, mid);
  build(rt << 1 | 1, mid + 1, r);
  push_up(rt);
}

void TranslateLeft(int rt, int l, int r, int val) {
  if (l <= v[rt].l && v[rt].r <= r) return push_tag(rt, val),
checkswitch(rt);
  int mid = v[rt << 1].r;
  push_down(rt);
  if (l <= mid) TranslateLeft(rt << 1, l, r, val);
```

```
  if (mid < r) TranslateLeft(rt << 1 | 1, l, r, val);
  push_up(rt);
}

int QueryMax(int rt, int l, int r) {
  if (l <= v[rt].l && v[rt].r <= r) return v[rt].b;
  int mid = v[rt << 1].r, res = 0;
  push_down(rt);
  if (l <= mid) res = max(res, QueryMax(rt << 1, l, r));
  if (mid < r) res = max(res, QueryMax(rt << 1 | 1, l, r));
  return res;
}

```

复杂度分析

证明 KTT 的时间复杂度需要用到势能分析.

设 $d(x)$ 为节点 x 在线段树上的深度，根节点的深度为 1. 定义节点 x 在线段树上的势能为:

$$\alpha(x)=\begin{cases}d(x) & \text{if the lower slope function has larger value}\\ 0 & \text{otherwise}\end{cases}$$

即在 x 比较的两个函数，如果拥有较小斜率的函数的值在 0 点更大，那么当前节点势能为 $d(x)$，否则为 0.

定义整个 KTT 的势能为所有节点势能之和:

$$\Phi=\sum_x \alpha(x)$$

考虑某次对于节点 x，和它的父亲 p 的实际更新代价 $c=1$，更新前后的势能分别为 Φ 和 Φ'. 我们去计算更新节点 x 的均摊更新代价. 由于当前节点 x 被更新，那么在当下它的势能一定是由 $d(x)$ 下降到 0. 而对于 p，它的势能最坏情况可能由 0 上升到 $d(p)$:

$$\begin{aligned}\hat{c}&=1+\Phi'-\Phi\\&=1+(\alpha'(p)+\alpha'(x))-(\alpha(p)+\alpha(x))\\&=1+(\alpha'(p)-\alpha(p))+(\alpha'(x)-\alpha(x))\\&\le 1+d(p)-d(x)\\&=0\end{aligned}$$

对实际代价求和，定义初始势能 Φ_s 以及最终势能 Φ_t:

$$\begin{aligned}\sum c&=\sum \hat{c}+\Phi_s-\Phi_t\\&\le \Phi_s-\Phi_t\\&=O(n\log n)\end{aligned}$$

这部分为 KTT 在仅存在全局修改的情况下，它将所有证书失效更新完毕的次数.

额外的，考虑区间平移对势能的影响. 对于某次区间平移，我们需要考虑的节点应当为，其子树当中存在但不是所有树节点被执行区间平移操作的节点. 这样的节点也就是我们在执行修改操作的时候在树上经过的节点，它的数量不超过 $O(\log n)$ 个，最坏情况下每个节点的势能上涨 $d(x)\le\log n$，因此每次操作上涨 $O(\log^2 n)$ 的势能.

为了维护区间平移，更新证书的操作将会被执行 $O(n\log n+m\log^2 n)$ 次. 每次更新证书我们需要从树上沿着路径走到证书失效的节点，这部分是 $O(\log n)$ 的. 因此总时间复杂度为 $O(n\log^2 n+m\log^3 n)$.

这个方法的优秀之处在于他已经触及问题时间复杂度的下界 $O(\lambda_s(n)\log^2 n)$. $\lambda_s(n)$ 表示长度最长的 (n, s) Davenport-Schinzel 序列. 其中线性函数对应 $s=1$ 的 $\lambda_1(n)=n$. 这部分属于计算几何内容，本篇不在这里赘述.

高次情况

如果我们维护的不是线性函数而是多项式函数, 或者更加复杂的函数我们如何应对. 两个复杂函数之间可能拥有多个交点. 给定一个连续、完全定义的单变量函数序列 $F=\{f_1,...,f_n\}$: $f_i:\mathbf{R}\to\mathbf{R}$. 其中每对函数的图像至多相交于 s 个点. 具有代表性的，s 次多项式函数集合符合这个要求.

对于同样的问题，我们使用势能分析.

$d(x)$ 为节点 x 在线段树上的深度，根节点的深度为 1. 定义 $I(x)$ 表示在节点 x 比较的两个函数，在 0 点过后有几个交点. 定义节点 x 在线段树上的势能为:

$$\alpha(x)=d(x)^{\log_2(s+1)}I(x)$$

定义整个 KTT 的势能为所有节点势能之和:

$$\Phi=\sum_x\alpha(x)$$

考虑某次对于节点 x，和它的父亲 p 的实际更新代价 $c=1$，更新前后的势能分别为 Φ 和 Φ'. 我们去计算更新节点 x 的均摊更新代价. 由于当前节点 x 被更新, 那么在当下它的势能一定是由 $d(x)^{\log_2(s+1)}I(x)$ 下降到 $d(x)^{\log_2(s+1)}(I(x)-1)$. 而对于 p，它的势能最坏情况可能由 0 上升到 $d(p)^{\log_2(s+1)}$:

$$\begin{aligned}\hat{c}&=1+\Phi'-\Phi\\&=1+(\alpha'(x)-\alpha(x))+(\alpha'(p)-\alpha(p))\\&\le 1-d(x)^{\log_2(s+1)}+s(d(x)-1)^{\log_2(s+1)}\\&\le 0\end{aligned}$$

由第三行到第四行我们使用了 $d(x)$ 为正整数的限制.

对实际代价求和，定义初始势能 Φ_s 以及最终势能 Φ_t:

$$\begin{aligned}\sum c&=\sum\hat{c}-\Phi_t+\Phi_s\\&\le\Phi_s-\Phi_t\\&=O(ns(\log n)^{\log_2(s+1)})\end{aligned}$$

我们得到复杂度的上界 $O(ns(\log n)^{1+\log_2(s+1)}+ms(\log n)^{2+\log_2(s+1)})$. [6]

近似情况

给定一个连续、完全定义的单变量函数序列 $F=\{f_1,...,f_n\}$，定义 $\mathfrak{U}_F(x)$，$\mathfrak{L}_F(x)$ 和 $\mathfrak{E}_F(x)$ 分别为上包络、下包络和幅度.

$$\begin{aligned}\mathfrak{U}_F(x)&=\max\{f_i(x)\mid f_i\in F\}\\\mathfrak{L}_F(x)&=\min\{f_i(x)\mid f_i\in F\}\\\mathfrak{E}_F(x)&=\mathfrak{U}_F(x)-\mathfrak{L}_F(x)\end{aligned}$$

我们只要求程序返回 $\tilde{\mathfrak{U}}_F(x)$ 满足

$$\mathfrak{U}_F(x)\ge\tilde{\mathfrak{U}}_F(x)\ge\mathfrak{U}_F(x)-\epsilon\mathfrak{E}_F(x)$$

那么在复杂情况下我们可以做到 $O((1/\epsilon^2)n\log^3 n)$，与多项式次数无关，以及我们允许函数同时进行区间左移或者右移.

[6]需要注意的是，这仅仅给出的是上界，复杂度的下界应为 $O(\lambda_s(n)\log n)$. 笔者猜测这里的势能分析构造应当参考 Davenport-Schinzel 序列对应的 $\lambda_s(n)$ 的通项公式以获取更紧的上界.

参考文献与注释

- P. K. Agarwal, S. Har-Peled, and K. R. Varadarajan. Approximating extent measures of points. J. ACM, 51(4):606–635, July 2004.
- J. Basch, L. J. Guibas, and J. Hershberger. Data structures for mobile data. Journal of Algorithms, 31(1):1–28, 1999.
- G. Alexandron, H. Kaplan, and M. Sharir. Kinetic and dynamic data structures for convex hulls and upper envelopes. Computational Geometry, 36(2):144–158, 2007.

[1] Data Structures for Mobile Data  [2] 浅谈函数最值的动态维护 

10.14 划分树

Authors: Xarfa

引入

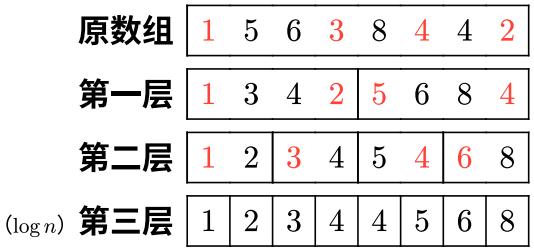
划分树是一种来解决区间第 K 大的一种数据结构，其常数、理解难度都要比主席树低很多．同时，划分树紧贴「第 K 大」，所以是一种基于排序的一种数据结构．

前置知识：主席树→10.17.2@p2424

过程

建树

划分树的建树比较简单，但是相对于其他树来说比较复杂．



如图，每一层都有一个看似无序的数组．其实，每一个被红色标记的数字都是 **要分配到左儿子的**．而分配的规则是什么？就是与 **这一层的中位数** 做比较，如果小于等于中位数，则分到左边，否则分到右边．但是这里要注意一下：并不是严格的 **小于等于就分到左边，否则分到右边**．因为中位数可能有相同，而且与 N 的奇偶有一定关系．下面的代码展示会有一个巧妙的运用，大家可以参照代码．

我们不可能每一次都对每一层排序，这样子不说常数，就算是理论复杂度也过不去．我们想，找中位数，一次排序就够了．为什么？比如，我们求 l, r 的中位数，其实就是在排完序过后的 `num[mid]`．

两个关键数组：

tree[log(N),N]: 也就是树，要存下所有的值，空间复杂度 $O(n \log n)$． toleft[log(N),n]: 也就是每一层 1~i 进入左儿子的数量，这里需要理解一下，这是一个前缀和．

实现

```
procedure Build(left,right,deep:longint); // left,right 表示区间左右端点,deep是第几层
```

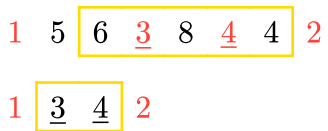
```
var
  i,mid,same,ls,rs,flag:longint; // 其中 flag 是用来平衡左右两边的数量的
begin
  if left=right then exit; // 到底层了
  mid:=(left+right) >> 1;
  same:=mid-left+1;
  for i:=left to right do
    if tree[deep,i]<num[mid] then
      dec(same);

  ls:=left; // 分配到左儿子的第一个指针
  rs:=mid+1; // 分配到右儿子的第一个指针
  for i:=left to right do
  begin
    flag:=0;
    if (tree[deep,i]<num[mid])or((tree[deep,i]=num[mid])and(same>0)) then // 分配到左边的条件
    begin
      flag:=1; tree[deep+1,ls]:=tree[deep,i]; inc(ls);
      if tree[deep,i]=num[mid] then // 平衡左右个数
        dec(same);
    end
    else
    begin
      tree[deep+1,rs]:=tree[deep,i]; inc(rs);
    end;
    toleft[deep,i]:=toleft[deep,i-1]+flag;
  end;
  Build(left,mid,deep+1); // 继续
  Build(mid+1,right,deep+1);
end;
```

查询

那我们先扯一下主席树的内容．在用主席树求区间第 K 小的时候，我们以 K 为基准，向左就向左，向右要减去向左的值，在划分树中也是这样子的．

查询难理解的，在于 **区间缩小** 这种东西．下图，查询的是 3 到 7，那么下一层就只需要查询 2 到 3 了．当然，我们定义 $[\text{left},\text{right}]$ 为缩小后的区间（目标区间），$[l,r]$ 还是所在节点的区间．那为什么要标出目标区间呢？因为那是 **判定答案在左边还是右边的基准**．



实现

```
function Query(left,right,k,l,r,deep:longint):longint;
```

```
var
  mid,x,y,cnt,rx,ry:longint;
begin
  if left=right then // 写成 l=r 也无妨,因为目标区间也一定有答案
    exit(tree[deep,left]);
  mid:=(l+r) >> 1;
  x:=toleft[deep,left-1]-toleft[deep,l-1]; // l 到 left 的去左儿子的个数
  y:=toleft[deep,right]-toleft[deep,l-1]; // l 到 right 的去左儿子的个数
  ry:=right-l-y; rx:=left-l-x; // ry 是 l 到 right 去右儿子的个数,rx 则是 l 到 left 去右儿子的个数
  cnt:=y-x; // left 到 right 左儿子的个数
  if cnt>=k then // 主席树常识啦
    Query:=Query(l+x,l+y-1,k,l,mid,deep+1) // l+x 就是缩小左边界,l+y-1 就是缩小右区间. 对于上图来说,就是把节点 1 和 2 放弃了.
  else
    Query:=Query(mid+rx+1,mid+ry+1,k-cnt,mid+1,r,deep+1); // 同样是缩小区间,只不过变成了右边而已. 注意要将 k 减去 cnt.
end;
```

性质

时间复杂度：一次查询只需要 $O(\log n)$，m 次询问，就是 $O(m \log n)$.

空间复杂度：只需要存储 $O(n \log n)$ 个数字.

亲测结果：主席树 :1482 ms、划分树 :889 ms.（非递归，常数比较小）

划分树的应用

例题：Luogu P3157[CQOI2011] 动态逆序对[1]

> 题意简述：给定一个 n 个元素的排列（$n \leq 10^5$），有 m 次询问（$m \leq 5 \times 10^4$），每次删去排列中的一个数，求删去这个数之后排列的逆序对个数.

这题可以使用 CDQ 在 $\Theta(n \log^2 n)$ 的时间及 $\Theta(n)$ 的空间内解决，并且 CDQ 的常数也很优秀.

如果这道题改为强制在线，则一般使用树状数组 + 主席树的树套树解法解决，时间复杂度为 $\Theta(n \log^2 n)$，空间复杂度为 $\Theta(n \log^2 n)$，常数略大，同样可以过此题.

而使用划分树的话就可以在 $\Theta(n \log^2 n)$ 的时间及 $\Theta(n \log n)$ 的空间内在线解决本题，同时常数也比树套树解法少很多.（大致与 CDQ 相当.）

> ⚠ **注意**
>
> 为了编程实现方便, 本文依照位置的中间值将大数组划分为两个小数组, 即下文中的划分树相当于是归并排序的过程，而非快速排序的过程. 最顶层的大数组为有序数组，最底层为原数组.

对于每一个划分树中的节点, 我们称他为右节点当且仅当他在下一层会被划分到右孩子, 即原数组中位置比较靠后的那些数, 相似的可以定义左节点. 如果在建树的过程中将最顶层排为有序的, 类似于归并排序求逆序对，可以发现一个数组的逆序对个数就是在每个左节点之前的右节点的个树和.

再考虑删除操作. 删除一个左节点会将整个数组的逆序对减少在他之前右结点的个数, 而删除一个右节点会减少在他之后的左节点个数. 那么可以考虑每次动态维护「每一个左节点之前的右结点个数」和「每一个右节点之后的左节点个数」. 这可以使用树状数组简单维护.

需要注意的是，在使用树状数组维护时只能计算在划分树中同一块内的贡献，而不能跳出块．对于树状数组来说有一个较为巧妙的处理方式．

考虑划分树上每一块的下标范围肯定为 $[c \times 2^k + 1, (c+1) \times 2^k]$ 的形式，列举如下（由于代码中不会涉及到划分树最底层的处理，因此只枚举到倒数第二层）：

```
[0001 0010] [0011 0100] [0101 0110] [0111 1000] [1001 1010] [1011 1100] [1101 1110] [1111 10000]  lev=1
[0001 0010 0011 0100]   [0101 0110 0111 1000]   [1001 1010 1011 1100]   [1101 1110 1111 10000]    lev=2
[0001 0010 0011 0100 0101 0110 0111 1000]       [1001 1010 1011 1100 1101 1110 1111 10000]        lev=3
[0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111 10000]                lev=4
```

回忆一下树状数组的原理，在向上跳的时候，我们每次 `x += lowbit(x)`．如果在向上跳的时候可以保证不跳出块，就可以保证只会影响到块内元素的值．向上查询也类似．

而如果要在向上跳的同时保证不跳出块，只需要保证在跳的时候满足 $lowbit(x) < 2^{lev}$ 即可．

而向下跳则是完全不同的处理方式．每一块的下标如果使用 0-index 表示的话，即为 $[c \times 2^k, (c+1) \times 2^k)$ 的形式．那么，只需将某一个下标的值右位移 k，即可得出它在哪一块中．在向下跳的时候时刻判断是否跳出块即可．

需要注意的是，按这一方法实现的树状数组会访问到的最大下标是距离 n 最近的 2 的整次幂，因此数组下标不能开 n．

由于需要在 $\log n$ 层修改，在第 k 层修改的时间复杂度为 $\Theta(k)$，最终时间复杂度即为 $\Theta(n \log n + m \log^2 n)$．

附代码：

```cpp
#include <algorithm>
#include <cstring>
#include <iostream>
#if defined(_MSC_VER) && !defined(__clang__)
#include <immintrin.h>
#define __builtin_clz _lzcnt_u32
#endif
using namespace std;
using lld = long long;

int ti[2][17][131073];

int lowbit(int x) { return x & (-x); }

void fixup(int k, int x) {
  for (; lowbit(x) < 1 << k; x += lowbit(x)) {
    ++ti[0][k - 1][x];
  }
  ++ti[0][k - 1][x];
}

void fixdown(int k, int x) {
  for (; ((x ^ lowbit(x)) - 1) >> k == (x - 1) >> k; x ^= lowbit(x)) {
    ++ti[1][k - 1][x];
  }
```

```
  ++ti[1][k - 1][x];
}

int queryup(int k, int x) {
  int res = 0;
  for (; lowbit(x) < 1 << k; x += lowbit(x)) {
    res += ti[1][k - 1][x];
  }
  return res + ti[1][k - 1][x];
}

int querydown(int k, int x) {
  int res = 0;
  for (; ((x ^ lowbit(x)) - 1) >> k == (x - 1) >> k; x ^= lowbit(x)) {
    res += ti[0][k - 1][x];
  }
  return res + ti[0][k - 1][x];
}

int ai[100005];
int tx[100005];

lld mgx(int* npi, int* nai, int* rxi, int* lxi, int al, int ar, int bl,
        int br) {
  int rx = 1;
  int lx = ar - al;
  lld res = 0;
  for (; al != ar || bl != br; ++npi, ++nai, ++rxi, ++lxi) {
    if (al != ar && (bl == br || tx[al] < tx[bl])) {
      --lx;
      *rxi = rx;
      *nai = tx[al];
      *npi = al;
      ++al;
    } else {
      ++rx;
      *lxi = lx;
      res += ar - al;
      *nai = tx[bl];
      *npi = bl;
      ++bl;
    }
  }
  return res;
}

int npi[1700005];
int nri[1700005];
int nli[1700005];

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

```
  for (int i = 1; i <= n; ++i) cin >> ai[i];

  if (n == 1) {
    for (int i = 1; i <= m; ++i) cout << "0\n";
    return 0;
  }

  const int logn = 31 - __builtin_clz(n - 1);

  lld ans = 0;
  for (int i = logn; i >= 0; --i) {
    memcpy(tx, ai, (n + 1) * sizeof(int));
    for (int j = 1; j <= n; j += 1 << (logn - i + 1)) {
      ans += mgx(npi + n * i + j, ai + j, nri + n * i + j, nli + n * i + j, j,
                 min(n + 1, j + (1 << (logn - i))),
                 min(n + 1, j + (1 << (logn - i))),
                 min(n + 1, j + (1 << (logn - i + 1))));
    }
  }
  cout << ans << '\n';

  for (int asdf = 1; asdf < m; ++asdf) {
    int x;
    cin >> x;
    for (int i = 0, p = 0; i <= logn; ++i, p += n) {
      if (nri[p + x]) {
        ans -= nri[p + x] - querydown(logn - i + 1, x) - 1;
        fixdown(logn - i + 1, x);
      } else {
        ans -= nli[p + x] - queryup(logn - i + 1, x);
        fixup(logn - i + 1, x);
      }
      x = npi[p + x];
    }
    cout << ans << '\n';
  }
}
```

后记

参考博文 :传送门[2].

外部参考资料

[1] Luogu P3157[CQOI2011] 动态逆序对

[2] 传送门

10.15 二叉搜索树 & 平衡树

10.15.1 二叉搜索树 & 平衡树

> **Authors:** 2323122, aofall, AtomAlpaca, Bocity, CoelacanthusHex, countercurrent-time, Early0v0, Enter-tainer, fearlessxjdx, Great-designer, H-J-Granger, hsfzLZH1, iamtwz, Ir1d, ksyx, Marcythm, NachtgeistW, ouuan, Persdre, shuzhouliu, StudyingFather, SukkaW, Tiphereth-A, wsyhb, Yesphet, yuhuoji, lingkerio, bililateral, q-wind

定义

二叉搜索树是一种二叉树的树形数据结构，其定义如下：

1. 空树是二叉搜索树.
2. 若二叉搜索树的左子树不为空，则其左子树上所有点的附加权值均小于其根节点的值.
3. 若二叉搜索树的右子树不为空，则其右子树上所有点的附加权值均大于其根节点的值.
4. 二叉搜索树的左右子树均为二叉搜索树.

二叉搜索树上的基本操作所花费的时间与这棵树的高度成正比. 对于一个有 n 个结点的二叉搜索树中，这些操作的最优时间复杂度为 $O(\log n)$，最坏为 $O(n)$. 随机构造这样一棵二叉搜索树的期望高度为 $O(\log n)$.

过程

二叉搜索树节点的定义

实现

```
struct TreeNode {
  int key;
  TreeNode* left;
  TreeNode* right;
  // 维护其他信息，如高度，节点数量等
  int size;   // 当前节点为根的子树大小
  int count;  // 当前节点的重复数量

  TreeNode(int value)
      : key(value), size(1), count(1), left(nullptr), right(nullptr) {}
};
```

遍历二叉搜索树

由二叉搜索树的递归定义可得，二叉搜索树的中序遍历权值的序列为非降的序列. 时间复杂度为 $O(n)$.

遍历一棵二叉搜索树的代码如下：

实现

```
void inorderTraversal(TreeNode* root) {
  if (root == nullptr) {
    return;
  }
  inorderTraversal(root->left);
  std::cout << root->key << " ";
  inorderTraversal(root->right);
}
```

查找最小/最大值

由二叉搜索树的性质可得，二叉搜索树上的最小值为二叉搜索树左链的顶点，最大值为二叉搜索树右链的顶点. 时间复杂度为 $O(h)$.

实现

```cpp
int findMin(TreeNode* root) {
  if (root == nullptr) {
    return -1;
  }
  while (root->left != nullptr) {
    root = root->left;
  }
  return root->key;
}

int findMax(TreeNode* root) {
  if (root == nullptr) {
    return -1;
  }
  while (root->right != nullptr) {
    root = root->right;
  }
  return root->key;
}
```

搜索元素

在以 `root` 为根节点的二叉搜索树中搜索一个值为 `value` 的节点.

分类讨论如下:

- 若 `root` 为空，返回 `false`.
- 若 `root` 的权值等于 `value`，返回 `true`.
- 若 `root` 的权值大于 `value`，在 `root` 的左子树中继续搜索.
- 若 `root` 的权值小于 `value`，在 `root` 的右子树中继续搜索.

时间复杂度为 $O(h)$.

实现

```cpp
bool search(TreeNode* root, int target) {
  if (root == nullptr) {
    return false;
  }
  if (root->key == target) {
    return true;
  } else if (target < root->key) {
    return search(root->left, target);
  } else {
    return search(root->right, target);
  }
}
```

插入，删除，修改都需要先在二叉搜索树中进行搜索.

插入一个元素

在以 `root` 为根节点的二叉搜索树中插入一个值为 `value` 的节点.

分类讨论如下：

- 若 `root` 为空，直接返回一个值为 `value` 的新节点.
- 若 `root` 的权值等于 `value`，该节点的附加域该值出现的次数自增 1.
- 若 `root` 的权值大于 `value`，在 `root` 的左子树中插入权值为 `value` 的节点.
- 若 `root` 的权值小于 `value`，在 `root` 的右子树中插入权值为 `value` 的节点.

时间复杂度为 $O(h)$.

实现

```
TreeNode* insert(TreeNode* root, int value) {
  if (root == nullptr) {
    return new TreeNode(value);
  }
  if (value < root->key) {
    root->left = insert(root->left, value);
  } else if (value > root->key) {
    root->right = insert(root->right, value);
  } else {
    root->count++;  // 节点值相等，增加重复数量
  }
  root->size = root->count + (root->left ? root->left->size : 0) +
               (root->right ? root->right->size : 0);  // 更新节点的子树大小
  return root;
}
```

删除一个元素

在以 `root` 为根节点的二叉搜索树中删除一个值为 `value` 的节点.

先在二叉搜索树中搜索权值为 `value` 的节点，分类讨论如下：

- 若该节点的附加 `count` 大于 1，只需要减少 `count`.
- 若该节点的附加 `count` 为 1:
  - 若 `root` 为叶子节点，直接删除该节点即可.
  - 若 `root` 为链节点，即只有一个儿子的节点，返回这个儿子.
  - 若 `root` 有两个非空子节点，一般是用它左子树的最大值（左子树最右的节点）或右子树的最小值（右子树最左的节点）代替它，然后将它删除.

时间复杂度 $O(h)$.

实现

方法使用 `root = remove(root, 1)` 表示删除根节点为 `root` 树中值为 1 的节点，并返回新的根节点.

```
// 此处返回值为删除 value 后的新 root
TreeNode* remove(TreeNode* root, int value) {
  if (root == nullptr) {
    return root;
  }
  if (value < root->key) {
    root->left = remove(root->left, value);
  } else if (value > root->key) {
    root->right = remove(root->right, value);
```

```
  } else {
    if (root->count > 1) {
      root->count--;  // 节点重复数量大于1，减少重复数量
    } else {
      if (root->left == nullptr) {
        TreeNode* temp = root->right;
        delete root;
        return temp;
      } else if (root->right == nullptr) {
        TreeNode* temp = root->left;
        delete root;
        return temp;
      } else {
        TreeNode* successor = findMinNode(root->right);
        root->key = successor->key;
        root->count = successor->count;  // 更新重复数量
        // 当 successor->count > 1时，也应该删除该节点，否则
        // 后续的删除只会减少重复数量
        successor->count = 1;
        root->right = remove(root->right, successor->key);
      }
    }
  }
  // 继续维护size，不写成 --root->size;
  // 是因为value可能不在树中，从而可能未发生删除
  root->size = root->count + (root->left ? root->left->size : 0) +
               (root->right ? root->right->size : 0);
  return root;
}

// 此处以右子树的最小值为例
TreeNode* findMinNode(TreeNode* root) {
  while (root->left != nullptr) {
    root = root->left;
  }
  return root;
}
```

求元素的排名

排名定义为将数组元素升序排序后第一个相同元素之前的数的个数加一.

查找一个元素的排名，首先从根节点跳到这个元素，若向右跳，答案加上左儿子节点个数加当前节点重复的数个数，最后答案加上终点的左儿子子树大小加一.

时间复杂度 $O(h)$.

实现

```
int queryRank(TreeNode* root, int v) {
  if (root == nullptr) return 0;
  if (root->key == v) return (root->left ? root->left->size : 0) + 1;
  if (root->key > v) return queryRank(root->left, v);
```

```
  return queryRank(root->right, v) + (root->left ? root->left->size : 0) +
         root->count;
}
```

查找排名为 k 的元素

在一棵子树中，根节点的排名取决于其左子树的大小.

- 若其左子树的大小大于等于 k，则该元素在左子树中;
- 若其左子树的大小在区间 $[k-count, k-1]$（count 为当前结点的值的出现次数）中，则该元素为子树的根节点;
- 若其左子树的大小小于 $k-count$，则该元素在右子树中.

时间复杂度 $O(h)$.

实现

```
int querykth(TreeNode* root, int k) {
  if (root == nullptr) return -1;  // 或者根据需求返回其他合适的值
  if (root->left) {
    if (root->left->size >= k) return querykth(root->left, k);
    if (root->left->size + root->count >= k) return root->key;
  } else {
    if (k <= root->count) return root->key;
  }
  return querykth(root->right,
                  k - (root->left ? root->left->size : 0) - root->count);
}
```

平衡树简介

使用搜索树的目的之一是缩短插入、删除、修改和查找（插入、删除、修改都包括查找操作）节点的时间.

关于查找效率，如果一棵树的高度为 h，在最坏的情况，查找一个关键字需要对比 h 次，查找时间复杂度（也为平均查找长度 ASL，Average Search Length）不超过 $O(h)$. 一棵理想的二叉搜索树所有操作的时间可以缩短到 $O(\log n)$（n 是节点总数）.

然而 $O(\log n)$ 的时间复杂度仅为理想情况. 在最坏情况下，搜索树有可能退化为链表. 想象一棵每个结点只有右孩子的二叉搜索树，那么它的性质就和链表一样，所有操作（增删改查）的时间是 $O(n)$.

可以发现操作的复杂度与树的高度 h 有关. 由此引出了平衡树, 通过一定操作维持树的高度（平衡性）来降低操作的复杂度.

平衡性的定义

关于一棵搜索树是否「**平衡**」，不同的平衡树中对「**平衡**」有着不同的定义. 比如以 T 为根节点的二叉搜索树，左子树和右子树的高度相差很大，或者左子树的节点个数远大于右子树的节点个数，这棵树显然不具有平衡性.

对于二叉搜索树来说，常见的平衡性的定义是指：以 T 为根节点的树，每一个结点的左子树和右子树高度差最多为 1.

- Splay 树→10.15.3@p2254 中，对于任意节点的访问操作（搜索、插入还是删除），都会将被访问的节点移动到树的根节点位置.

- AVL 树→10.15.8@p2341 每个节点 N 维护以 N 为根节点的树的高度信息. AVL 树对平衡性的定义：如果 T 是一棵 AVL 树，当且仅当左右子树也是 AVL 树，且 $|height(T->left) - height(T->right)| \leq 1$.
- Size Balanced Tree→10.15.7@p2315 每个节点 N 维护以 N 为根节点的树中节点个数 `size`. 对平衡性的定义：任意节点的 `size` 不小于其兄弟节点（Sibling）的所有子节点（Nephew）的 `size`.

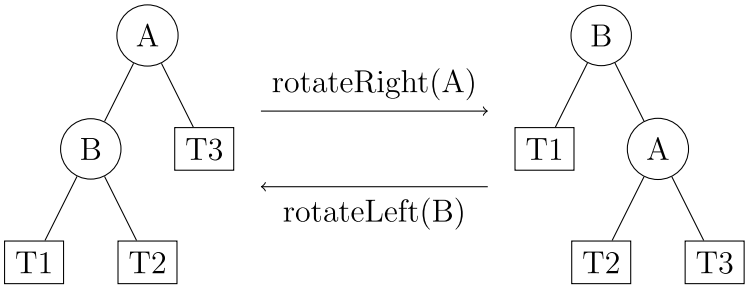
此外，对于拥有同样元素值集合的搜索树，平衡状态可能是不唯一的. 也就是说，可能两棵不同的搜索树，含有的元素值集合相同，并且都是平衡的.

平衡的调整过程

对不满足平衡条件的搜索树进行调整操作，可以使不平衡的搜索树重新具有平衡性.

关于二叉平衡树，平衡的调整操作分为包括 **左旋（Left Rotate 或者 zag）** 和 **右旋（Right Rotate 或者 zig）** 两种. 由于二叉平衡树在调整时需要保证中序遍历序列不变. 这两种操作均不改变中序遍历序列.

在这里先介绍右旋，右旋也称为「右单旋转」或「LL 平衡旋转」. 对于结点 A 的右旋操作是指：将 A 的左孩子 B 向右上旋转，代替 A 成为根节点，将 A 结点向右下旋转成为 B 的右子树的根结点，B 的原来的右子树变为 A 的左子树.



右旋操作只改变了三组结点关联, 相当于对三组边进行循环置换一下, 因此需要暂存一个结点再进行轮换更新.

对于右旋操作一般的更新顺序是：暂存 B 结点（新的根节点），让 A 的左孩子指向 B 的右子树 $T2$，再让 B 的右孩子指针指向 A，最后让 A 的父结点指向暂存的 B.

完全同理, 有对应的左旋操作, 也称为「左单旋转」或「RR 平衡旋转」. 左旋操作与右旋操作互为镜像.

下面给出左旋和右旋的代码.

实现

```
TreeNode* rotateLeft(TreeNode* root) {
  TreeNode* newRoot = root->right;
  root->right = newRoot->left;
  newRoot->left = root;
  // 更新相关节点的信息
  updateHeight(root);
  updateHeight(newRoot);
  return newRoot;  // 返回新的根节点
}

TreeNode* rotateRight(TreeNode* root) {
  TreeNode* newRoot = root->left;
  root->left = newRoot->right;
  newRoot->right = root;
  updateHeight(root);
```

```
16    updateHeight(newRoot);
17    return newRoot;
18 }
```

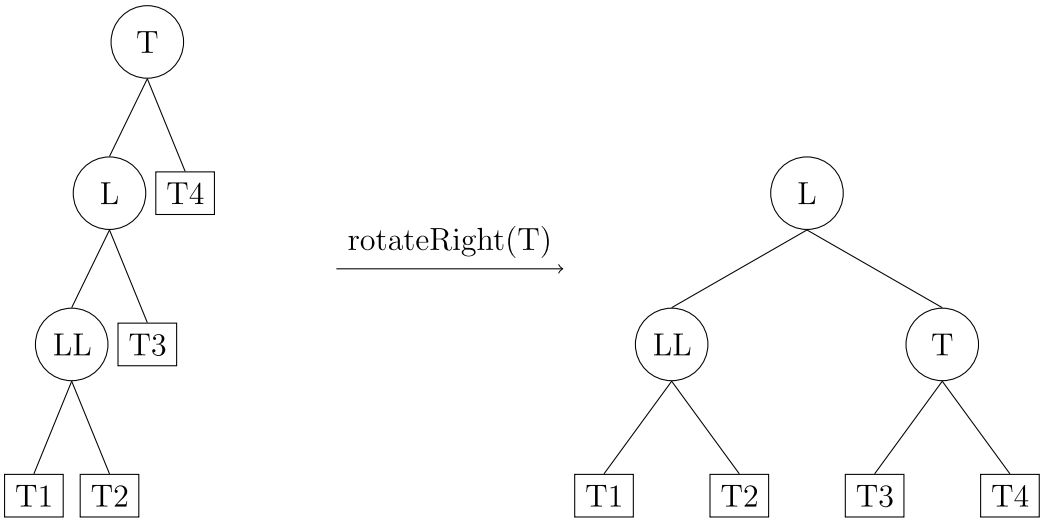
对于这段示例代码，在调用时需要保存 `root` 的父节点 `pre`. 方法返回指向新的根节点的指针，只需要将 `pre` 指向新的根节点即可.

四种平衡性破坏的情况

虽然不同的二叉平衡树的定义有所区别，不同二叉平衡树区别只在于节点维护的信息不同，以及旋转调整后节点更新的信息不同. 二叉平衡树平衡性被破坏的情况只有以下四种. 进行平衡性调整的操作只包括左旋和右旋. 以下先介绍四种情况，再对不同的二叉平衡树进行对比.

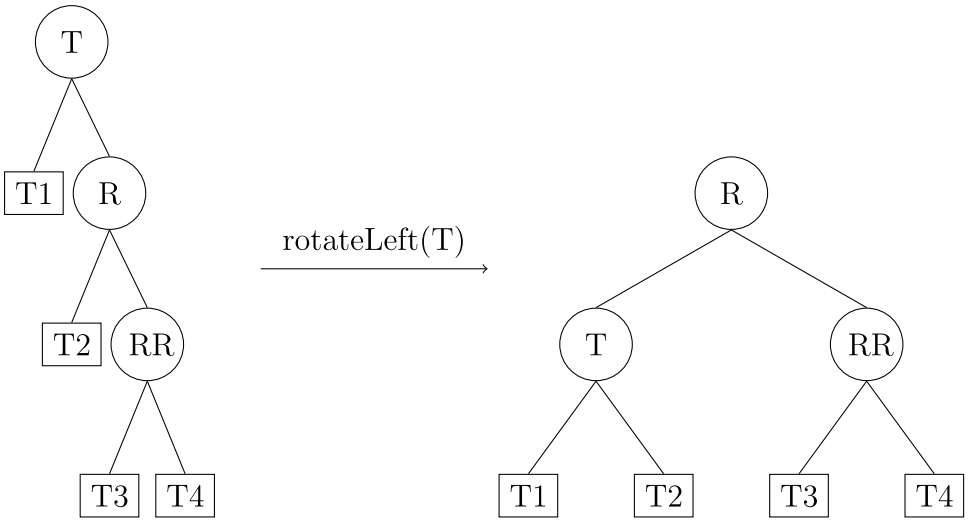
LL 型：T 的左孩子的左子树过长导致平衡性破坏.

调整方式：右旋节点 T.



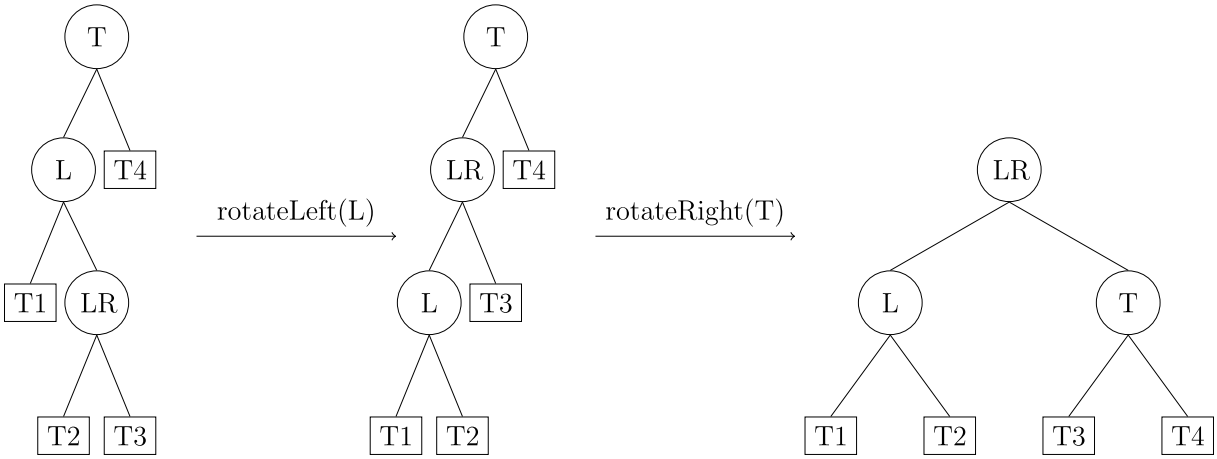
RR 型：与 LL 型类似，T 的右孩子的右子树过长导致平衡性破坏.

调整方式：左旋节点 T.



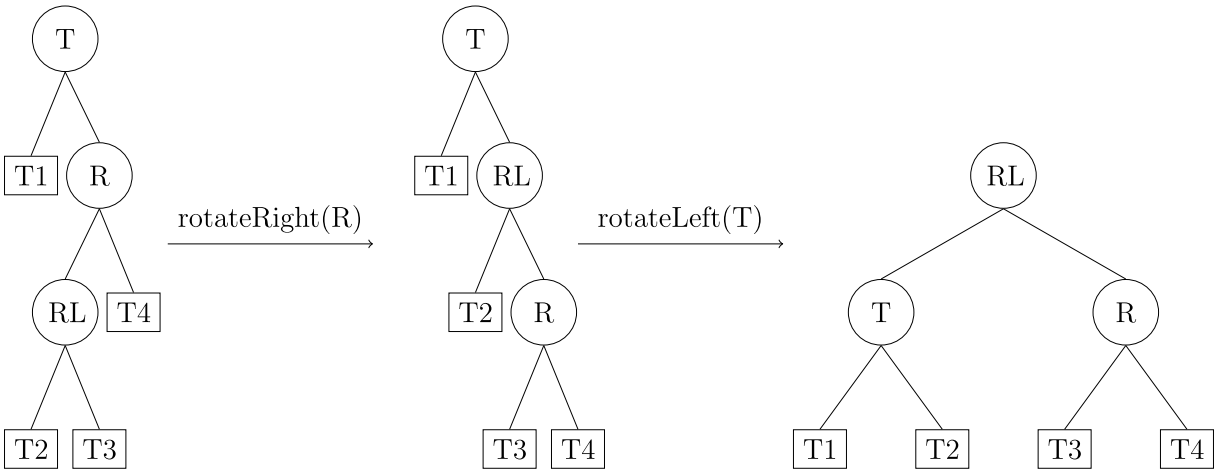
LR 型：T 的左孩子的右子树过长导致平衡性破坏.

调整方式：先左旋节点 L，成为 LL 型，再右旋节点 T.



RL 型：与 LR 型类似，T 的右孩子的左子树过长导致平衡性破坏.

调整方式：先右旋节点 R，成为 RR 型，再左旋节点 T.



10.15.2 Treap

Authors: Dev-XYS, ttzytt, Sora233, qwqAutomaton

前置知识：朴素二叉搜索树→10.15.1@p2213，堆基础→10.7.1@p2025.

简介

Treap（树堆）是一种 **弱平衡** 的 **二叉搜索树**.

Treap 的结点除了被维护的 **权值**（*val*）之外，还附加了一个随机的 **优先级**（*priority*）. 其中，权值满足二叉搜索树性质，优先级满足堆性质（小根堆或大根堆）.

其中，二叉搜索树的性质是指：

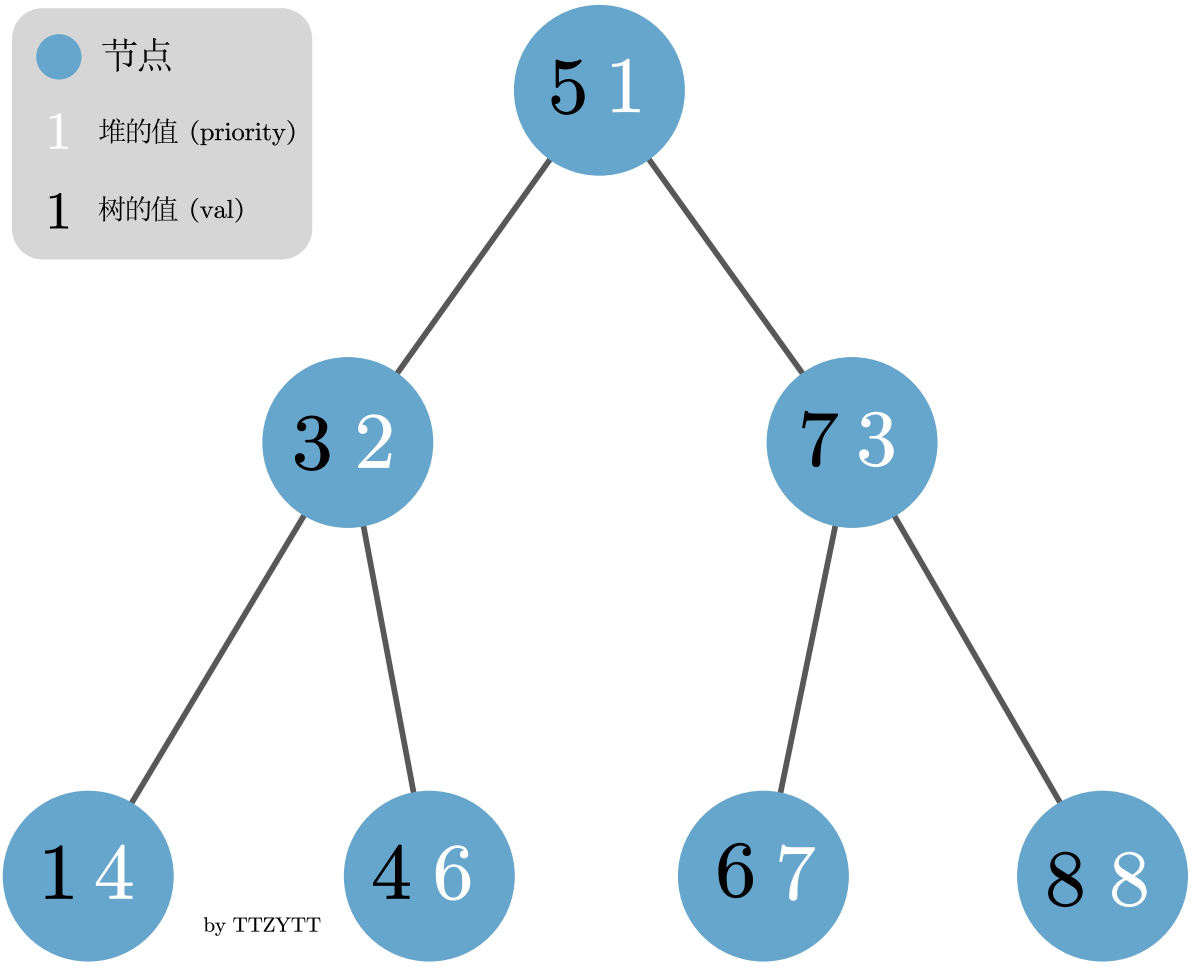
- 左子树所有节点的权值（*val*）比父节点小.
- 右子树所有节点的权值（*val*）比父节点大.

堆的性质是：

- 子节点优先级（*priority*）比父节点大或小（取决于是小根堆还是大根堆）.

不难看出，如果用的是同一个值，那么这两种数据结构在组合后会变成一条链，所以我们再在搜索树的基础上，引入一个给堆的值 *priority*. 对于 *val* 值，我们维护搜索树的性质，对于 *priority* 值，我们维护堆的性质. 其中 *priority* 这个值是随机给出的.

下图就是一个 Treap 的例子（这里使用的是小根堆，即根节点的优先级最小）.



那我们为什么需要大费周章的去让这个数据结构符合树和堆的性质，并且随机给出堆的值呢?

要理解这个，首先需要理解朴素二叉搜索树的问题．在给朴素搜索树插入一个新节点时，我们需要从这个搜索树的根节点开始递归，如果新节点比当前节点小，那就向左递归，反之亦然．

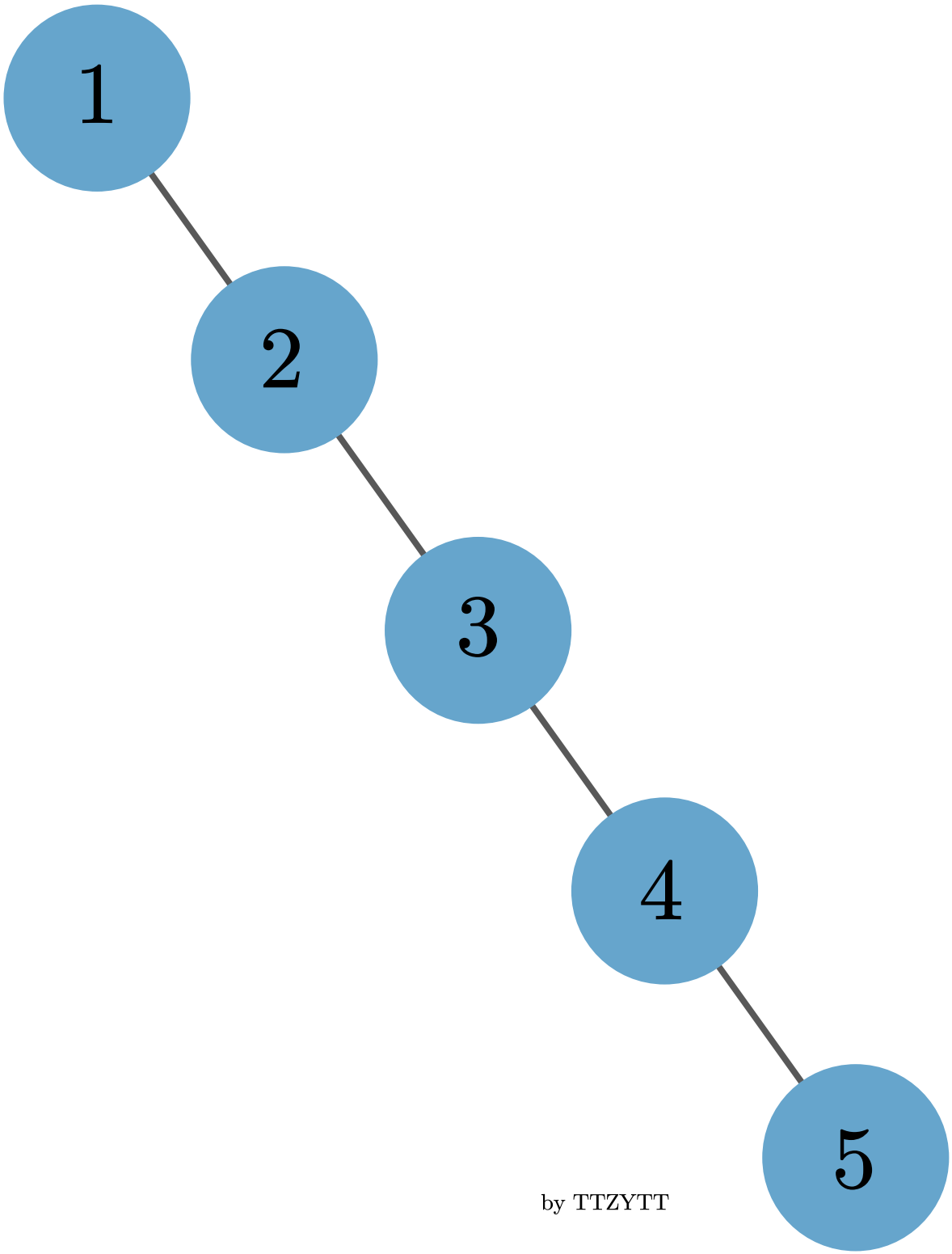
最后当发现当前节点没有子节点时，就根据新节点的值的大小，让新节点成为当前节点的左或右子节点．

如果插入结点的权值是随机的（换言之，是随机插入的），那这个朴素搜索树的高度较小（接近 $\log n$，其中 n 为结点数），而每一层的节点数较多，即它的形状会非常的「胖」．上图的 Treap 就是一个例子．因此此时的任意操作复杂度都将会是 $O(\log n)$ 左右．

不过，这只是在随机情况下的复杂度，如果我们按照下面这个非常有序的顺序给一个朴素的搜索树插入节点：

```
1 1 2 3 4 5
```

那么这棵树将会退化成链，即变得非常「瘦长」（每次插入的节点都比前面的大，所以都被安排到右子节点了）：



不难看出，查询的复杂度也从 $O(\log n)$ 变成了 $O(n)$.

而 treap 为了解决这个问题、达到一个较为「平衡」的状态，通过维护随机的优先级满足堆性质，「打乱」了节点的插入顺序，从而让二叉搜索树达到了理想的复杂度，避免了退化成链的问题.

Treap 复杂度的证明

由于 treap 各种操作的复杂度都和所操作的节点的深度有关，我们首先证明，所有节点的期望深度都是 $O(\log n)$.

记号约定

为了方便表述，我们约定：

- n 是节点个数.
- Treap 节点中满足二叉搜索树性质的称为 **权值**，满足堆性质的（也就是随机的）称为 **优先级**. 不妨设优先级满足小根堆性质.
- x_k 表示权值第 k 小的节点.
- $X_{i,j}$ 表示集合 $\{x_i, x_{i+1}, \cdots, x_{j-1}, x_j\}$，即按权值升序排列后第 i 个到第 j 个的节点构成的集合.
- $\mathrm{dep}(x)$ 表示节点 x 的深度. 规定根节点的深度是 0.
- $Y_{i,j}$ 是一个指示器随机变量，当 x_i 是 x_j 的祖先时值为 1，否则为 0. 特别地，$Y_{i,i} = 0$.
- $\Pr(A)$ 表示事件 A 发生的概率.

节点期望深度的证明

由于节点 x_i 的深度等于它祖先的个数，因此有

$$\mathrm{dep}(x_i) = \sum_{k=1}^{n} Y_{k,i}.$$

那么根据期望的线性性，有

$$E(\mathrm{dep}(x_i)) = E\left(\sum_{k=1}^{n} Y_{k,i}\right) = \sum_{k=1}^{n} E(Y_{k,i}).$$

由于 $Y_{k,i}$ 是指示器随机变量，它的期望就等于它为 1 的概率，因此

$$E(\mathrm{dep}(x_i)) = \sum_{k=1}^{n} \Pr(Y_{k,i} = 1).$$

我们先证明引理：$Y_{i,j} = 1$ 当且仅当 x_i 的优先级是 $X_{i,j}$ 中最小的.

引理的证明

考虑分类讨论 x_i 和 x_j 的情况.

1. 若 x_i 是根节点：由于优先级满足小根堆性质，x_i 的优先级最小，并且对于任意的 x_j，x_i 都是 x_j 的祖先.
2. 若 x_j 是根节点：同理，x_j 优先级最小，因此 x_i 不是 $X_{i,j}$ 中优先级最小的；同时 x_i 也不是 x_j 的祖先.
3. 若 x_i 和 x_j 在根节点的两个子树中（一左一右），那么根节点 $r \in X_{i,j}$. 因此 x_i 的优先级不可能是 $X_{i,j}$ 中最小的（因为根节点的比它小）. 同时，由于 x_i 和 x_j 分属两个子树，x_i 也不是 x_j 的祖先.
4. 若 x_i 和 x_j 在根节点的同一个子树中, 此时可以将这个子树单独拿出来作为一棵新的 treap, 递归进行上面的证明即可.

那么根据引理，深度的期望可以转化成

$$E(\mathrm{dep}(x_i)) = \sum_{k=1}^{n} \Pr(x_k = \min X_{i,k} \wedge k \neq i).$$

又因为节点的优先级是随机的, 我们假定集合 $X_{i,j}$ 中任何一个节点的优先级最小的概率都相同, 那么

$$
\begin{aligned}
E(\operatorname{dep}(x_i)) &= \sum_{k=1}^{n} \Pr(x_k = \min X_{i,k} \wedge k \neq i) \\
&= \sum_{k=1}^{n} \Pr(x_k = \min X_{i,k}) - 1 \\
&= \sum_{k=1}^{n} \frac{1}{|\,i-k\,|+1} - 1 \\
&= \sum_{k=1}^{i-1} \frac{1}{i-k+1} + \sum_{k=i+1}^{n} \frac{1}{k-i+1} \\
&= \sum_{j=2}^{i} \frac{1}{j} + \sum_{j=2}^{n-i+1} \frac{1}{j} \\
&\leq 2\sum_{j=2}^{n} \frac{1}{j} < 2\sum_{j=2}^{n} \int_{j-1}^{j} \frac{1}{x}\mathrm{d}x \\
&= 2\int_{1}^{n} \frac{1}{x}\mathrm{d}x = 2\ln n = O(\log n).
\end{aligned}
$$

因此每个节点的期望深度都是 $O(\log n)$.

而朴素的二叉搜索树的操作的复杂度均是 $O(h)$，同时 treap 维护堆性质的复杂度也是 $O(h)$ 的，因此 treap 各种操作的期望复杂度都是 $O(\log n)$.

期望复杂度的感性理解

首先，我们需要认识到一个节点的 *priority* 属性是和它所在的层数有直接关联的. 再回忆堆的性质:

- 子节点值（*priority*）比父节点大或小（取决于是小根堆还是大根堆）

我们发现层数低的节点, 比如整个树的根节点, 它的 *priority* 属性也会更小（在小根堆中）. 并且，在朴素的搜索树中，先被插入的节点，也更有可能会有比较小的层数. 我们可以把这个 *priority* 属性和被插入的顺序关联起来理解，这样，也就理解了为什么 treap 可以把节点插入的顺序通过 *priority* 打乱.

给 treap 插入新节点时，需要同时维护树和堆的性质. 其中，搜索树的性质可以在插入时维护，而堆性质的维护则有两种处理方法，分别是旋转和分裂、合并. 使用这两种方法的 treap 被分别称为 **旋转 treap** 和 **无旋 treap**.

旋转 treap

旋转 treap 维护平衡的方式为旋转，和 AVL 树的旋转操作类似，分为 **左旋** 和 **右旋**. 即在满足二叉搜索树的条件下根据堆的优先级对 treap 进行平衡操作.

旋转 treap 在做普通平衡树题的时候，是所有平衡树中常数较小的.

下面的讲解中的代码用指针实现了旋转 treap，文末附有数组形式的完整实现.

Info

代码中的 `rank` 代表前面讲的优先级（*priority* 属性），该属性满足的是小根堆性质.

节点结构

```
struct Node {
  Node *ch[2];  // 两个子节点的地址
```

```
  int val, rank;
  int rep_cnt;  // 当前这个值（val）重复出现的次数
  int siz;      // 以当前节点为根的子树大小

  Node(int val) : val(val), rep_cnt(1), siz(1) {
    ch[0] = ch[1] = nullptr;
    rank = rand();
    // 注意初始化的时候，rank 是随机给出的
  }

  void upd_siz() {
    // 用于旋转和删除过后，重新计算 siz 的值
    siz = rep_cnt;
    if (ch[0] != nullptr) siz += ch[0]->siz;
    if (ch[1] != nullptr) siz += ch[1]->siz;
  }
};
```

旋转

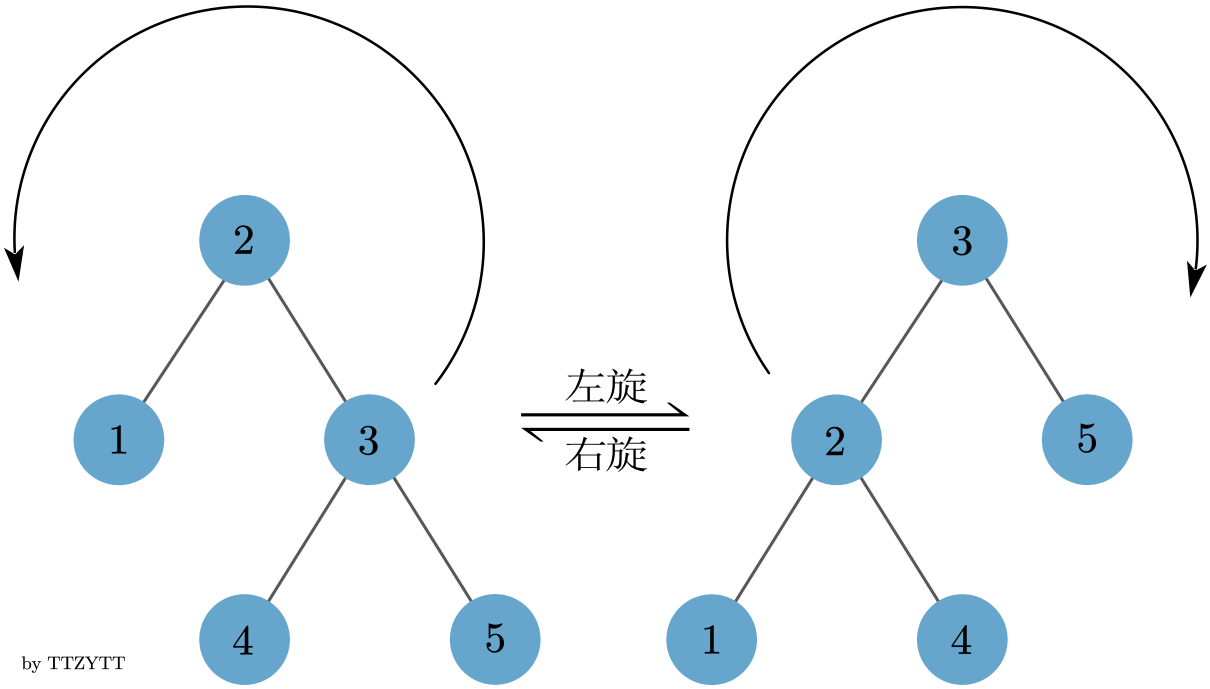
旋转操作是 treap 的一个非常重要的操作，主要用来在保持 treap 树性质的同时，调整不同节点的层数，以达到维护堆性质的作用.

旋转操作的左旋和右旋可能不是特别容易区分，以下是两个较为明显的特点：

旋转操作的含义：

- 在不影响搜索树性质的前提下，把和旋转方向相反的子树变成根节点（如左旋，就是把右子树变成根节点）
- 不影响性质，并且在旋转过后，跟旋转方向相同的子节点变成了原来的根节点（如左旋，旋转完之后的左子节点是旋转前的根节点）

左旋和右旋操作是相互的，如下图.



```cpp
enum rot_type { LF = 1, RT = 0 };

void _rotate(Node *&cur,
             rot_type dir) {  // dir参数代表旋转的方向 0为右旋，1为左旋
  // 注意传进来的 cur 是指针的引用，也就是改了这个
  // cur，变量是跟着一起改的，如果这个 cur 是别的 树的子节点，根据 ch
  // 找过来的时候，也是会找到这里的

  // 以下的代码解释的均是左旋时的情况
  Node *tmp = cur->ch[dir];  // 让 C 变成根节点，
                             // 这里的 tmp
                             // 是一个临时的节点指针，指向成为新的根节点的节点

  /* 左旋：也就是让右子节点变成根节点
   *         A                 C
   *        / \               / \
   *       B  C    ---->     A   E
   *         / \             / \
   *        D   E           B   D
   */
  cur->ch[dir] = tmp->ch[!dir];    // 让 A 的右子节点变成 D
  tmp->ch[!dir] = cur;             // 让 C 的左子节点变成 A
  cur->upd_siz(), tmp->upd_siz();  // 更新大小信息
  cur = tmp;  // 最后把临时储存 C 树的变量赋值到当前根节点上（注意 cur 是引用）
}
```

插入

类似普通二叉搜索树的插入，但是需要在插入的过程中通过旋转来维护优先级的堆性质.

```cpp
void _insert(Node *&cur, int val) {
  if (cur == nullptr) {
    // 没这个节点直接新建
    cur = new Node(val);
    return;
  } else if (val == cur->val) {
    // 如果有这个值相同的节点，就把重复数量加一
    cur->rep_cnt++;
    cur->siz++;
  } else if (val < cur->val) {
    // 维护搜索树性质，val 比当前节点小就插到左边，反之亦然
    _insert(cur->ch[0], val);
    if (cur->ch[0]->rank < cur->rank) {
      // 小根堆中，上面节点的优先级一定更小
      // 因为新插的左子节点比父节点小，现在需要让左子节点变成父节点
      _rotate(cur, RT);  // 注意前面的旋转性质，要把左子节点转上来，需要右旋
    }
    cur->upd_siz();  // 插入之后大小会变化，需要更新
  } else {
    _insert(cur->ch[1], val);
    if (cur->ch[1]->rank < cur->rank) {
      _rotate(cur, LF);
    }
    cur->upd_siz();
  }
```

```
26 }
```

删除

主要就是分类讨论，不同的情况有不同的处理方法，删完了树的大小会有变化，要注意更新．并且如果要删的节点有左子树和右子树，就要考虑删除之后让谁来当父节点（维护 rank 小的节点在上面）．

```
1  void _del(Node *&cur, int val) {
2    if (val > cur->val) {
3      _del(cur->ch[1], val);
4      // 值更大就在右子树，反之亦然
5      cur->upd_siz();
6    } else if (val < cur->val) {
7      _del(cur->ch[0], val);
8      cur->upd_siz();
9    } else {
10     if (cur->rep_cnt > 1) {
11       // 如果要删除的节点是重复的，可以直接把重复值减小
12       cur->rep_cnt--, cur->siz--;
13       return;
14     }
15     uint8_t state = 0;
16     state |= (cur->ch[0] != nullptr);
17     state |= ((cur->ch[1] != nullptr) << 1);
18     // 00都无，01有左无右，10，无左有右，11都有
19     Node *tmp = cur;
20     switch (state) {
21       case 0:
22         delete cur;
23         cur = nullptr;
24         // 没有任何子节点，就直接把这个节点删了
25         break;
26       case 1:  // 有左无右
27         cur = tmp->ch[0];
28         // 把根变成左儿子，然后把原来的根节删了，注意这里的 tmp 是从 cur
29         // 复制的，而 cur 是引用
30         delete tmp;
31         break;
32       case 2:  // 有右无左
33         cur = tmp->ch[1];
34         delete tmp;
35         break;
36       case 3:
37         rot_type dir = cur->ch[0]->rank < cur->ch[1]->rank
38                            ? RT
39                            : LF;  // dir 是 rank 更小的那个儿子
40         _rotate(cur, dir);  // 这里的旋转可以把优先级更小的儿子转上去，rt 是 0，
41                             // 而 lf 是 1，刚好跟实际的子树下标反过来
42         _del(
43             cur->ch[!dir],
44             val);  // 旋转完成后原来的根节点就在旋方向那边，所以需要
45                    // 继续把这个原来的根节点删掉
46                    // 如果说要删的这个节点是在整个树的「上层的」，那我们会一直通过这
47                    // 这里的旋转操作，把它转到没有子树了（或者只有一个），再删掉它．
48         cur->upd_siz();
```

```
49          // 删除会造成大小改变
50          break;
51      }
52    }
53  }
```

根据值查询排名

操作含义：查询以 cur 为根节点的子树中，val 这个值的大小的排名（该子树中小于 val 的节点的个数 + 1）

```
1  int _query_rank(Node *cur, int val) {
2    int less_siz = cur->ch[0] == nullptr ? 0 : cur->ch[0]->siz;
3    // 这个树中小于 val 的节点的数量
4    if (val == cur->val)
5      // 如果这个节点就是要查的节点
6      return less_siz + 1;
7    else if (val < cur->val) {
8      if (cur->ch[0] != nullptr)
9        return _query_rank(cur->ch[0], val);
10     else
11       return 1;  // 如果左子树是空的，说比最小的节点还要小，那这个数字就是最小的
12   } else {
13     if (cur->ch[1] != nullptr)
14       // 如果要查的值比这个节点大，那这个节点的左子树以及这个节点自身肯定都比要查的值小
15       // 所以要加上这两个值，再加上往右边找的结果
16       //  （以右子树为根的子树中，val 这个值的大小的排名）
17       return less_siz + cur->rep_cnt + _query_rank(cur->ch[1], val);
18     else
19       return cur->siz + 1;
20     // 没有右子树的话直接整个树 + 1 相当于 less_siz + cur->rep_cnt + 1
21   }
22 }
```

根据排名查询值

要根据排名查询值，我们首先要知道如何判断要查的节点在树的哪个部分：

以下是一个判断方法的表：

左子树	根节点/当前节点	右子树
排名 ≤ 左子树的大小	排名 > 左子树的大小, 并且 ≤ 左子树的大小 + 根节点的重复次数	排名 > 左子树的大小 + 根节点的重复次数

注意如果在右子树，递归的时候需要对原来的 `rank` 进行处理．递归的时候就相当去查，在右子树中为这个排名的值，为了把排名转换成基于右子树的，需要把原来的 `rank` 减去左子树的大小和根节点的重复次数．

可以把所有节点想象成一个排好序的数组，或者数轴（如下），

```
1 -> |左子树的节点|根节点|右子树的节点| -> n
                              ^
                            要查的排名
                          ⬇转换成基于右子树的排名
1 -> |右子树的节点| -> n
       ^
```

```
7         要查的排名
```

这里的转换方法就是直接把排名减去左子树的大小和根节点的重复数量.

```
int _query_val(Node *cur, int rank) {
  // 查询树中第 rank 大的节点的值
  int less_siz = cur->ch[0] == nullptr ? 0 : cur->ch[0]->siz;
  // less siz 是左子树的大小
  if (rank <= less_siz)
    return _query_val(cur->ch[0], rank);
  else if (rank <= less_siz + cur->rep_cnt)
    return cur->val;
  else
    return _query_val(cur->ch[1], rank - less_siz - cur->rep_cnt);  // 见前文
}
```

查询第一个比 val 小的节点

注意这里使用了一个类中的全局变量，`q_prev_tmp`.

这个值是只有在 val 比当前节点值大的时候才会被更改的，所以返回这个变量就是返回 val 最后一次比当前节点的值大，之后就是更小了.

```
int _query_prev(Node *cur, int val) {
  if (val <= cur->val) {
    // 还是比 val 大，所以往左子树找
    if (cur->ch[0] != nullptr) return _query_prev(cur->ch[0], val);
  } else {
    // 只有能进到这个 else 里，才会更新 q_prev_tmp 的值
    q_prev_tmp = cur->val;
    // 当前节点已经比 val，小了，但是不确定是否是最大的，所以要到右子树继续找
    if (cur->ch[1] != nullptr) _query_prev(cur->ch[1], val);
    // 接下来的递归可能不会更改 q_prev_tmp
    // 了，那就直接返回这个值，总之返回的就是最后一次进到 这个 else 中的
    // cur->val
    return q_prev_tmp;
  }
  return NIL;
}
```

查询第一个比 val 大的节点

跟前一个很相似，只是大于小于号换了一下.

```
int _query_nex(Node *cur, int val) {
  if (val >= cur->val) {
    if (cur->ch[1] != nullptr) return _query_nex(cur->ch[1], val);
  } else {
    q_nex_tmp = cur->val;
    if (cur->ch[0] != nullptr) _query_nex(cur->ch[0], val);
    return q_nex_tmp;
  }
  return NIL;
}
```

无旋 treap

无旋 treap 的操作方式使得它天生支持维护序列、可持久化等特性.

无旋 treap 又称分裂合并 treap. 它仅有两种核心操作，即为 **分裂** 与 **合并**. 通过这两种操作，在很多情况下可以比旋转 treap 更方便的实现别的操作. 下面逐一介绍这两种操作.

> **注释**
>
> 讲解无旋 treap 应当提到 FHQ-Treap（by 范浩强）. 即可持久化，支持区间操作的无旋 Treap. 更多内容请参照《范浩强谈数据结构》ppt.

分裂（split）

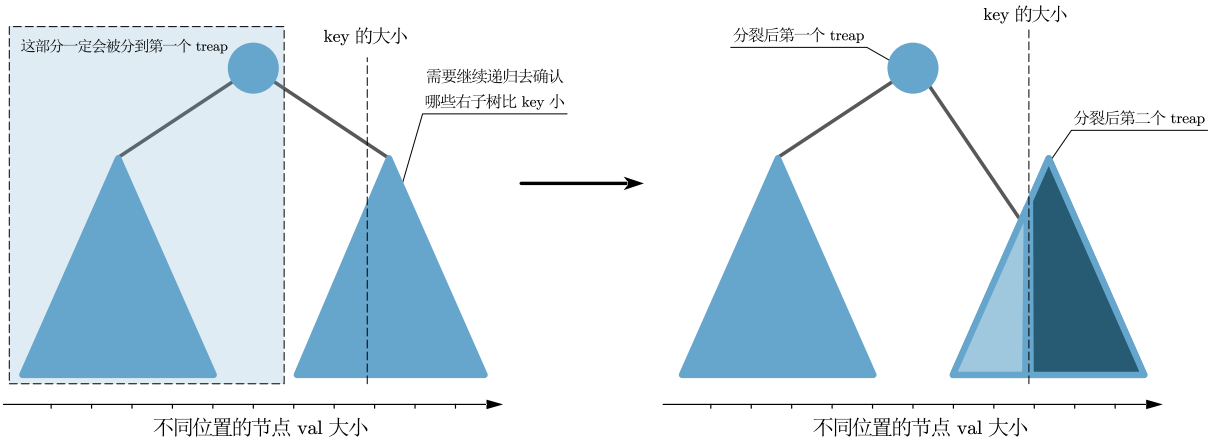
按值分裂

分裂过程接受两个参数：根指针 cur、关键值 key. 结果为将根指针指向的 treap 分裂为两个 treap，第一个 treap 所有结点的值（val）小于等于 key，第二个 treap 所有结点的值大于 key.

该过程首先判断 key 是否小于 cur 的值，若小于，则说明 cur 及其右子树全部大于 key，属于第二个 treap. 当然，也可能有一部分的左子树的值大于 key，所以还需要继续向左子树递归地分裂. 对于大于 key 的那部分左子树，我们把它作为 cur 的左子树，这样，整个 cur 上的节点都是大于 key 的.

相应的，如果 key 大于等于 cur 的值，说明 cur 的整个左子树以及其自身都小于等于 key，属于分裂后的第一个 treap. 并且，cur 的部分右子树也可能有部分小于等于 key，因此我们需要继续递归地分裂右子树. 把小于等于 key 的那部分作为 cur 的右子树，这样，整个 cur 上的节点都小于等于 key.

下图展示了 cur 的值小于等于 key 时按值分裂的情况. [1]



by TTZYTT

$$\text{if cur} \to \text{val} \le \text{key}$$

```cpp
pair<Node *, Node *> split(Node *cur, int key) {
  if (cur == nullptr) return {nullptr, nullptr};
  if (cur->val <= key) {
    // cur 以及它的左子树一定属于分裂后的第一个树
    auto temp = split(cur->ch[1], key);
    // 但是它可能有部分右子树也比 key 小
    cur->ch[1] = temp.first;
    // 我们把小于 key 的那部分拿出来，作为 cur 的右子树，这样整个 cur 都是小于
    // key 的 剩下的那部分右子树成为分裂后的第二个 treap
    cur->upd_siz();
```

[1]本图的设计参考了 维基百科 treap 词条的配图[1]

```
    // 分裂过后树的大小会变化，需要更新
    return {cur, temp.second};
  } else {
    // 同上
    auto temp = split(cur->ch[0], key);
    cur->ch[0] = temp.second;
    cur->upd_siz();
    return {temp.first, cur};
  }
}
```

按排名分裂

比起按值分裂，这个操作更像是旋转 treap 中的根据排名（某个节点的排名是树中所有小于此节点值的节点的数量 $+1$）查询值：

此函数接受两个参数，节点指针 cur 和排名 rk，返回分裂后的三个 treap.

其中，第一个 treap 中每个节点的排名都小于 rk，第二个的排名等于 rk，并且第二个 treap 只有一个节点（不可能有多个等于的，如果有的话会增加 `Node` 结构体中的 `cnt`），第三个则是大于.

此操作的重点在于判断排名和 cur 相等的节点在树的哪个部分，这也是旋转 treap 根据排名查询值操作时的重要部分，在前文有非常详细的解释，这里不过多讲解.

并且，此操作的递归部分和按值分裂也非常相似，这里不赘述.

```
tuple<Node *, Node *, Node *> split_by_rk(Node *cur, int rk) {
  if (cur == nullptr) return {nullptr, nullptr, nullptr};
  int ls_siz = cur->ch[0] == nullptr ? 0 : cur->ch[0]->siz;
  if (rk <= ls_siz) {
    // 排名和 cur 相等的节点在左子树
    Node *l, *mid, *r;
    tie(l, mid, r) = split_by_rk(cur->ch[0], rk);
    cur->ch[0] = r;  // 返回的第三个 treap 中的排名都大于 rk
    // cur 的左子树被设成 r 后，整个 cur 中节点的排名都大于 rk
    cur->upd_siz();
    return {l, mid, cur};
  } else if (rk <= ls_siz + cur->cnt) {
    // 和 cur 相等的就是当前节点
    Node *lt = cur->ch[0];
    Node *rt = cur->ch[1];
    cur->ch[0] = cur->ch[1] = nullptr;
    // 分裂后第二个 treap 只有一个节点，所有要把它的子树设置为空
    return {lt, cur, rt};
  } else {
    // 排名和 cur 相等的节点在右子树
    // 递归过程同上
    Node *l, *mid, *r;
    tie(l, mid, r) = split_by_rk(cur->ch[1], rk - ls_siz - cur->cnt);
    cur->ch[1] = l;
    cur->upd_siz();
    return {cur, mid, r};
  }
}
```

合并（merge）

合并过程接受两个参数：左 treap 的根指针 u、右 treap 的根指针 v．必须满足 u 中所有结点的值小于等于 v 中所有结点的值．一般来说，我们合并的两个 treap 都是原来从一个 treap 中分裂出去的，所以不难满足 u 中所有节点的值都小于 v

在旋转 treap 中，我们借助旋转操作来维护 $priority$ 符合堆的性质，同时旋转时还不能改变树的性质．在无旋 treap 中，我们用合并达到相同的效果．

因为两个 treap 已经有序，所以我们在合并的时候只需要考虑把哪个树「放在上面」，把哪个「放在下面」，也就是需要判断将哪个一个树作为子树．显然，根据堆的性质，我们需要把 $priority$ 小的放在上面（这里采用小根堆）．

同时，我们还需要满足搜索树的性质，所以若 u 的根结点的 $priority$ 小于 v 的，那么 u 即为新根结点，并且 v 因为值比 u 更大，应与 u 的右子树合并；反之，则 v 作为新根结点，然后因为 u 的值比 v 小，与 v 的左子树合并．

```
Node *merge(Node *u, Node *v) {
  // 传进来的两个树的内部已经符合搜索树的性质了
  // 并且 u 内所有节点的值 < v 内所有节点的值
  // 所以在合并的时候需要维护堆的性质
  // 这里用的是小根堆
  if (u == nullptr && v == nullptr) return nullptr;
  if (u != nullptr && v == nullptr) return u;
  if (v != nullptr && u == nullptr) return v;

  if (u->prio < v->prio) {
    // u 的 prio 比较小，u应该作为父节点
    u->ch[1] = merge(u->ch[1], v);
    // 因为 v 比 u 大，所以把 v 作为 u 的右子树
    u->upd_siz();
    return u;
  } else {
    // v 比较小，v应该作为父节点
    v->ch[0] = merge(u, v->ch[0]);
    // u 比 v 小，所以递归时的参数是这样的
    v->upd_siz();
    return v;
  }
}
```

插入

在无旋 treap 中，插入，删除，根据值查询排名等基础操作既可以用普通二叉查找树的方法实现，也可以用分裂和合并来实现．通常来说，使用分裂和合并来实现更加简洁，但是速度会慢一点[2]．为了帮助更好的理解无旋 treap，下面的操作全部使用分裂和合并实现．

在实现插入操作时，我们利用了分裂操作的一些性质．也就是值小于等于 val 的节点会被分到第一个 treap．

所以，假设我们根据 val 分裂当前这个 treap．会有下面两棵树，并符合以下条件：

$$T_1 \leq val$$

$$T_2 > val$$

其中 T_1 表示分裂后所有被分到第一个 treap 的节点的集合，T_2 则是第二个．

[2] https://charleswu.site/archives/1051[2]

如果我们再按照 $val-1$ 继续分裂 T_1，那么会产生下面两棵树，并符合以下条件：

$$T_{1\text{ left}} \leq val-1$$

$$T_{1\text{ right}} > val-1 \ \&\ T_{1\text{ right}} \leq val$$

其中 $T_{1\text{ left}}$ 表示 T_1 分裂后所有被分到第一个 treap 的节点的集合，$T_{1\text{ right}}$ 则是第二个．并且上面的式子中，后半部分的 $\& \ T_{1\text{ right}} \leq val$ 来自于 T_1 所符合的条件 $T_1 \leq val$.

不难发现，只要 val 和节点的值是一个整数（大多数使用场景下会使用整数）那么符合 $T_{1\text{ right}}$ 条件的节点只有一个，也就是值等于 val 的节点.

在插入时，如果我们发现符合 $T_{1\text{ right}}$ 的节点存在，那就可以直接增加重复次数，否则，就新开一个节点.

注意把树分裂好了还需要用合并操作把它「粘」回去，这样下次还能继续使用．并且，还需要注意合并操作的参数顺序是有要求的，第一个树的所有节点的值都需要小于第二个.

```
void insert(int val) {
  auto temp = split(root, val);
  // 根据 val 的值把整个树分成两个
  // 注意 split 的实现，等于 val 的子树是在左子树的
  auto l_tr = split(temp.first, val - 1);
  // l_tr 的左子树 <= val - 1，如果有 = val 的节点，那一定在右子树
  Node *new_node;
  if (l_tr.second == nullptr) {
    // 没有这个节点就新开，否则直接增加重复次数.
    new_node = new Node(val);
  } else {
    l_tr.second->cnt++;
    l_tr.second->upd_siz();
  }
  Node *l_tr_combined =
      merge(l_tr.first, l_tr.second == nullptr ? new_node : l_tr.second);
  // 合并 T_1 left 和 T_1 right
  root = merge(l_tr_combined, temp.second);
  // 合并 T_1 和 T_2
}
```

删除

删除操作也使用和插入操作相似的方法，找到值和 val 相等的节点，并且删除它.

```
void del(int val) {
  auto temp = split(root, val);
  auto l_tr = split(temp.first, val - 1);
  if (l_tr.second->cnt > 1) {
    // 如果这个节点的重复次数大于 1，减小即可
    l_tr.second->cnt--;
    l_tr.second->upd_siz();
    l_tr.first = merge(l_tr.first, l_tr.second);
  } else {
    if (temp.first == l_tr.second) {
      // 有可能整个 T_1 只有这个节点，所以也需要把这个点设成 null 来标注已经删除
      temp.first = nullptr;
    }
    delete l_tr.second;
    l_tr.second = nullptr;
```

```
  }
  root = merge(l_tr.first, temp.second);
}
```

根据值查询排名

排名是比这个值小的节点的数量 $+1$，所以我们根据 $val-1$ 分裂当前树，那么分裂后的第一个树就符合：

$$T_1 \leq val-1$$

如果树的值和 val 为整数，那么 T_1 就包含了所有值小于 val 的节点.

```
int qrank_by_val(Node* cur, int val) {
  auto temp = split(cur, val - 1);
  int ret = (temp.first == nullptr ? 0 : temp.first->siz) + 1;  // 根据定义 + 1
  root = merge(temp.first, temp.second);  // 拆好了再粘回去
  return ret;
}
```

根据排名查询值

调用 `split_by_rk()` 函数后，会返回分裂好的三个 treap，其中第二个只包含一个节点，它的排名等于 rk，所以我们直接返回这个节点的 val.

```
int qval_by_rank(Node *cur, int rk) {
  Node *l, *mid, *r;
  tie(l, mid, r) = split_by_rk(cur, rk);
  int ret = mid->val;
  root = merge(merge(l, mid), r);
  return ret;
}
```

查询第一个比 val 小的节点

可以把这个问题转化为，在比 val 小的所有节点中，找出排名最大的. 我们根据 val 来分裂这个 treap，返回的第一个 treap 中的节点的值就全部小于 val，然后我们调用 `qval_by_rank()` 找出这个树中值最大的节点.

```
int qprev(int val) {
  auto temp = split(root, val - 1);
  // temp.first 就是值小于 val 的子树
  int ret = qval_by_rank(temp.first, temp.first->siz);
  // 这里查询的是，所有小于 val 的节点里面，最大的那个的值
  root = merge(temp.first, temp.second);
  return ret;
}
```

查询第一个比 val 大的节点

和上个操作类似，可以把这个问题转化为，在比 val 大的所有节点中，找出排名最小的. 那么根据 val 分裂后，返回的第二个 treap 中的所有节点的值就大于 val.

然后我们去查询这个树中排名为 1 的节点（也就是值最小的节点）的值，就可以成功查到第一个比 val 大的节点.

```
int qnex(int val) {
  auto temp = split(root, val);
```

```
  int ret = qval_by_rank(temp.second, 1);
  // 查询所有大于 val 的子树里面，值最小的那个
  root = merge(temp.first, temp.second);
  return ret;
}
```

建树（build）

将一个有 n 个节点的序列 $\{a_n\}$ 转化为一棵 treap.

可以依次暴力插入这 n 个节点，每次插入一个权值为 v 的节点时，将整棵 treap 按照权值分裂成权值小于等于 v 的和权值大于 v 的两部分，然后新建一个权值为 v 的节点，将两部分和新节点按从小到大的顺序依次合并，单次插入时间复杂度 $O(\log n)$，总时间复杂度 $O(n \log n)$.

在某些题目内，可能会有多次插入一段有序序列的操作，这是就需要在 $O(n)$ 的时间复杂度内完成建树操作.

方法一：在递归建树的过程中，每次选取当前区间的中点作为该区间的树根，并对每个节点钦定合适的优先值，使得新树满足堆的性质．这样能保证树高为 $O(\log n)$.

方法二：在递归建树的过程中，每次选取当前区间的中点作为该区间的树根，然后给每个节点一个随机优先级．这样能保证树高为 $O(\log n)$，但不保证其满足堆的性质．这样也是正确的，因为无旋式 treap 的优先级是用来使 `merge` 操作更加随机一点，而不是用来保证树高的.

方法三：观察到 treap 是笛卡尔树，利用笛卡尔树的 $O(n)$ 建树方法即可，用单调栈维护右链即可.

无旋 treap 的区间操作

建树

无旋 treap 相比旋转 treap 的一大好处就是可以实现各种区间操作，下面我们以文艺平衡树的 模板题[3] 为例，介绍 treap 的区间操作.

> 您需要写一种数据结构（可参考题目标题），来维护一个有序数列.
>
> 其中需要提供以下操作：翻转一个区间，例如原有序序列是 5 4 3 2 1，翻转区间是 $[2,4]$ 的话，结果是 5 2 3 4 1． 对于 100% 的数据，$1 \le n$（初始区间长度）m（翻转次数）$\le 10^5$

在这道题目中，我们需要实现的是区间翻转，那么我们首先需要考虑如何建树，建出来的树需要是初始的区间.

我们只需要把区间的下标依次插入 treap 中，这样在中序遍历（先遍历左子树，然后当前节点，最后右子树）时，就可以得到这个区间[3].

我们知道在朴素的二叉查找树中按照递增的顺序插入节点，建出来的树是一个长链，按照中序遍历，自然可以得到这个区间.

如上图，按照 1 2 3 4 5 的顺序给朴素搜索树插入节点，中序遍历时，得到的也是 1 2 3 4 5.

但是在 treap 中，按增序插入节点后，在合并操作时还会根据 *priority* 调整树的结构，在这样的情况下，如何确保中序遍历一定能正确的输出呢?

可以参考 笛卡尔树的单调栈建树方法→10.15.6@p2311 来理解这个问题.

设新插入的节点为 u.

首先，因为是递增地插入节点，每一个新插入的节点肯定会被连接到 treap 的右链（即从根结点一直往右子树走，经过的结点形成的链）上.

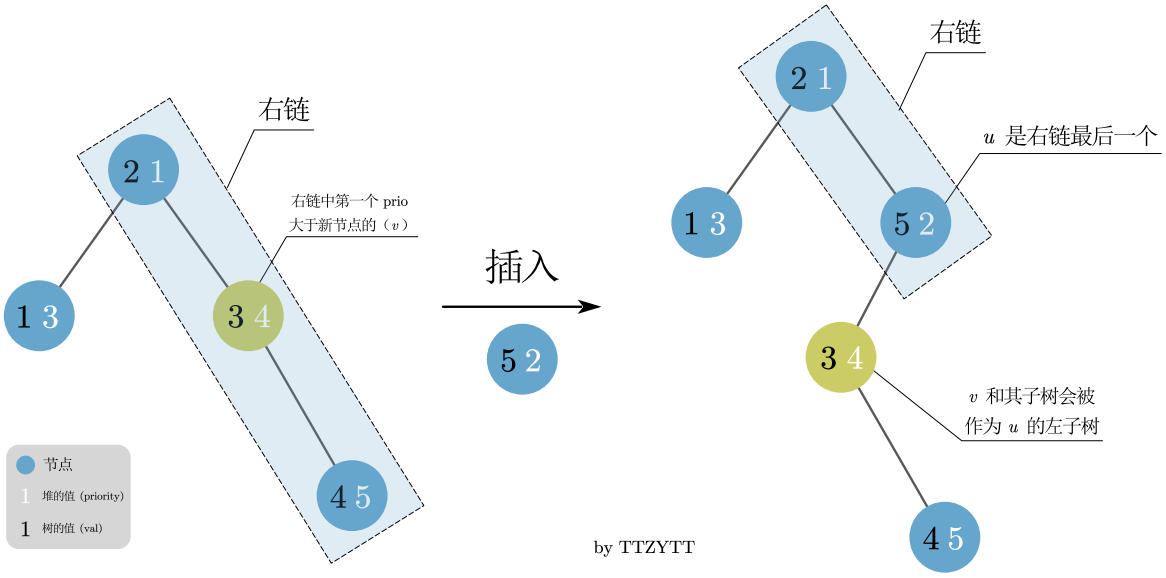
[3] https://www.cnblogs.com/Equinox-Flower/p/10785292.html[4]

从根节点开始，右链上的节点的 $priority$ 是递增的（小根堆）. 那我们可以找到右链上第一个 $priority$ 大于 u 的节点，我们叫这个节点 v，并把这个节点换成 u.

因为 u 一定大于这个树上其他的全部节点，我们需要把 v 以及它的子树作为 u 的左子树. 并且此时 u 没有右子树.

可以发现，中序遍历时 u 一定是最后一个被遍历到的（因为 u 是右链中的最后一个，而中序遍历中，右子树是最后被遍历到的）.

下图是一个 treap 根据递增顺序插入 $1 \sim 5$ 号节点时，插入 5 号节点时的变化，可以用这张图更好的理解按照增序插入的过程.

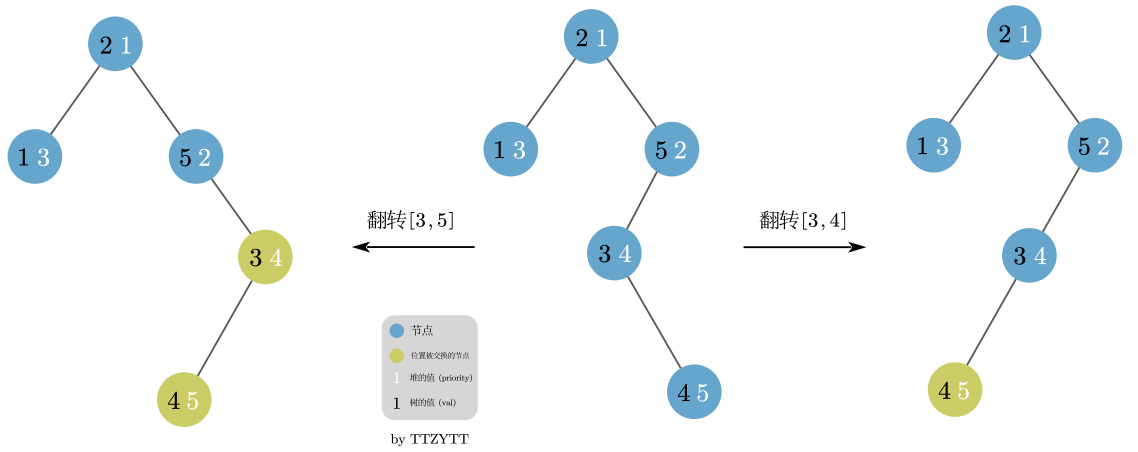


区间翻转

翻转 $[l, r]$ 这个区间时，基本思路是将树分裂成 $[1, l-1]$, $[l, r]$, $[r+1, n]$ 三个区间，再对中间的 $[l, r]$ 进行翻转[4].

翻转的具体操作是把区间内的子树的每一个左，右子节点交换位置. 如下图就展示了翻转上图中 treap 的 $[3, 4]$ 和 $[3, 5]$ 区间后的 treap.

[4] https://www.cnblogs.com/Equinox-Flower/p/10785292.html[5]



注意如果按照这个方法翻转，那么每次翻转 $[l,r]$ 区间时，就会有 $r-l$ 个节点会被交换位置，这样频繁的操作显然不能满足 10^5 的数据范围，其 $O(n \times \log_2 n)$ 的单次翻转复杂度甚至不如暴力（因为我们除了需要花线性时间交换节点外，还需要在树中花费 $O(\log_2 n)$ 的时间找到需要交换的节点）.

再观察题目要求，可以发现因为只需要最后输出操作完的区间，所以并不需要每次都真的去交换. 如此一来，便可以使用线段树中常用的懒标记（lazy tag）来优化复杂度. 交换时，只需要在父节点打上标记，代表这个子树下的每个左右子节点都需要交换就行了.

在线段树中，我们一般在更新和查询时下传懒标记. 这是因为，在更新和查询时，我们想要更新/查询的范围不一定和懒标记代表的范围重合，所以要先下传标记，确保查到和更新后的值是正确的.

在无旋 treap 中也是一样. 具体操作时我们会把 treap 分裂成前文讲到的三个树，然后给中间的树打上懒标记后合并这三棵树. 因为我们想要翻转的区间和懒标记代表的区间不一定重合，所以要在分裂时下传标记. 并且，分裂和合并操作会造成每个节点及其懒标记所代表的节点发生变动，所以也需要在合并前下传懒标记.

换句话说，是当树的结构发生改变的时候，当我们进行分裂或合并操作时需要改变某一个点的左右儿子信息时之前，应该下放标记，而非之后，因为懒标记是需要下传给儿子节点的，但更改左右儿子信息之后若懒标记还未下放，则懒标记就丢失了下放的对象. [5]

以下为代码讲解，代码参考了 [6].

因为区间操作中大部分操作都和普通的无旋 treap 相同，所以这里只讲解和普通无旋 treap 不同的地方.

下传标记

需要注意这里的懒标记代表需要把这个树中的每一个子节点交换位置. 所以如果当前节点的子节点也有懒标记，那两次翻转就抵消了. 如果子节点不需要翻转，那么这个懒标记就需要继续被下传到子节点上.

```
// 这里这个 pushdown 是 Node 类的成员函数，其中 to_rev 是懒标记
void pushdown() {
  swap(ch[0], ch[1]);
  if (ch[0] != nullptr) ch[0]->to_rev ^= 1;
  if (ch[1] != nullptr) ch[1]->to_rev ^= 1;
  to_rev = false;
```

[5] https://www.luogu.com.cn/blog/85514/fhq-treap-xue-xi-bi-ji[6]
[6] https://www.cnblogs.com/Equinox-Flower/p/10785292.html[7]

```
}

void check_tag() {
  if (to_rev) pushdown();
}
```

分裂

注意在这个题目中, 因为翻转操作, treap 中的 val 会不符合二叉搜索树的性质(见区间翻转部分的图), 所以我们不能根据 val 来判断应该往左子树还是右子树递归.

所以这里的分裂跟普通无旋 treap 中的按排名分裂更相似，是根据当前树的大小判断往左还是右子树递归的，换言之，我们是按照开始时这个节点在树中的位置来判断的.

返回的第一个 treap 中节点的排名全部小于等于 sz，而第二个 treap 中节点的排名则全部大于 sz.

```
#define siz(_) (_ == nullptr ? 0 : _->siz)

pair<Node*, Node*> split(Node* cur, int sz) {
  // 按照树的大小判断
  if (cur == nullptr) return {nullptr, nullptr};
  cur->check_tag();
  // 分裂前先下传
  if (sz <= siz(cur->ch[0])) {
    auto temp = split(cur->ch[0], sz);
    cur->ch[0] = temp.second;
    cur->upd_siz();
    return {temp.first, cur};
  } else {
    auto temp =
        split(cur->ch[1],
              sz - siz(cur->ch[0]) -
                  1);  // 这里的转换在有旋 treap 的 「根据排名查询值有讲」
    cur->ch[1] = temp.first;
    cur->upd_siz();
    return {cur, temp.second};
  }
}
```

合并

唯一需要注意的是在合并前下传懒标记

```
Node *merge(Node *sm, Node *bg) {
  // small, big
  if (sm == nullptr && bg == nullptr) return nullptr;
  if (sm != nullptr && bg == nullptr) return sm;
  if (sm == nullptr && bg != nullptr) return bg;
  sm->check_tag(), bg->check_tag();
  if (sm->prio < bg->prio) {
    sm->ch[1] = merge(sm->ch[1], bg);
    sm->upd_siz();
    return sm;
  } else {
    bg->ch[0] = merge(sm, bg->ch[0]);
    bg->upd_siz();
```

```
    return bg;
  }
}
```

区间翻转

和前面介绍的一样，分裂出 $[1,l-1]$，$[l,r]$，$[r+1,n]$ 三个区间，然后对中间的区间打上标记后再合并.

```
void seg_rev(int l, int r) {
  // 这里的 less 和 more 是相对于 l 的
  auto less = split(root, l - 1);
  // 所有小于等于 l - 1 的会在 less 的左子树
  auto more = split(less.second, r - l + 1);
  // 从 l 开始的前 r - l + 1 个元素的区间
  more.first->to_rev = true;
  root = merge(less.first, merge(more.first, more.second));
}
```

中序遍历打印

要注意在打印时要下传标记.

```
void print(Node* cur) {
  if (cur == nullptr) return;
  cur->check_tag();
  // 中序遍历 -> 先左子树，再自己，最后右子树
  print(cur->ch[0]);
  cout << cur->val << " ";
  print(cur->ch[1]);
}
```

完整代码

旋转 treap

指针实现

完整代码

以下是前文讲解的代码的完整版本，是普通平衡树的模板代码.

```
// author: (ttzytt)[ttzytt.com]
#include <cstdint>
#include <cstdio>
#include <cstdlib>
using namespace std;

struct Node {
  Node *ch[2];
  int val, rank;
  int rep_cnt;
  int siz;

  Node(int val) : val(val), rep_cnt(1), siz(1) {
    ch[0] = ch[1] = nullptr;
    rank = rand();
```

```
  }

  void upd_siz() {
    siz = rep_cnt;
    if (ch[0] != nullptr) siz += ch[0]->siz;
    if (ch[1] != nullptr) siz += ch[1]->siz;
  }
};

class Treap {
 private:
  Node *root;

  constexpr static int NIL = -1;  // 用于表示查询的值不存在

  enum rot_type { LF = 1, RT = 0 };

  int q_prev_tmp = 0, q_nex_tmp = 0;

  void _rotate(Node *&cur, rot_type dir) {  // 0为右旋，1为左旋
    Node *tmp = cur->ch[dir];
    cur->ch[dir] = tmp->ch[!dir];
    tmp->ch[!dir] = cur;
    cur->upd_siz(), tmp->upd_siz();
    cur = tmp;
  }

  void _insert(Node *&cur, int val) {
    if (cur == nullptr) {
      cur = new Node(val);
      return;
    } else if (val == cur->val) {
      cur->rep_cnt++;
      cur->siz++;
    } else if (val < cur->val) {
      _insert(cur->ch[0], val);
      if (cur->ch[0]->rank < cur->rank) {
        _rotate(cur, RT);
      }
      cur->upd_siz();
    } else {
      _insert(cur->ch[1], val);
      if (cur->ch[1]->rank < cur->rank) {
        _rotate(cur, LF);
      }
      cur->upd_siz();
    }
  }

  void _del(Node *&cur, int val) {
    if (val > cur->val) {
      _del(cur->ch[1], val);
```

```
68        cur->upd_siz();
69      } else if (val < cur->val) {
70        _del(cur->ch[0], val);
71        cur->upd_siz();
72      } else {
73        if (cur->rep_cnt > 1) {
74          cur->rep_cnt--, cur->siz--;
75          return;
76        }
77        uint8_t state = 0;
78        state |= (cur->ch[0] != nullptr);
79        state |= ((cur->ch[1] != nullptr) << 1);
80        // 00都无，01有左无右，10，无左有右，11都有
81        Node *tmp = cur;
82        switch (state) {
83          case 0:
84            delete cur;
85            cur = nullptr;
86            break;
87          case 1:  // 有左无右
88            cur = tmp->ch[0];
89            delete tmp;
90            break;
91          case 2:  // 有右无左
92            cur = tmp->ch[1];
93            delete tmp;
94            break;
95          case 3:
96            rot_type dir = cur->ch[0]->rank < cur->ch[1]->rank ? RT : LF;
97            _rotate(cur, dir);
98            _del(cur->ch[!dir], val);
99            cur->upd_siz();
100           break;
101       }
102     }
103   }
104
105   int _query_rank(Node *cur, int val) {
106     int less_siz = cur->ch[0] == nullptr ? 0 : cur->ch[0]->siz;
107     if (val == cur->val)
108       return less_siz + 1;
109     else if (val < cur->val) {
110       if (cur->ch[0] != nullptr)
111         return _query_rank(cur->ch[0], val);
112       else
113         return 1;
114     } else {
115       if (cur->ch[1] != nullptr)
116         return less_siz + cur->rep_cnt + _query_rank(cur->ch[1], val);
117       else
118         return cur->siz + 1;
119     }
```

```
120   }
121
122   int _query_val(Node *cur, int rank) {
123     int less_siz = cur->ch[0] == nullptr ? 0 : cur->ch[0]->siz;
124     if (rank <= less_siz)
125       return _query_val(cur->ch[0], rank);
126     else if (rank <= less_siz + cur->rep_cnt)
127       return cur->val;
128     else
129       return _query_val(cur->ch[1], rank - less_siz - cur->rep_cnt);
130   }
131
132   int _query_prev(Node *cur, int val) {
133     if (val <= cur->val) {
134       if (cur->ch[0] != nullptr) return _query_prev(cur->ch[0], val);
135     } else {
136       q_prev_tmp = cur->val;
137       if (cur->ch[1] != nullptr) _query_prev(cur->ch[1], val);
138       return q_prev_tmp;
139     }
140     return NIL;
141   }
142
143   int _query_nex(Node *cur, int val) {
144     if (val >= cur->val) {
145       if (cur->ch[1] != nullptr) return _query_nex(cur->ch[1], val);
146     } else {
147       q_nex_tmp = cur->val;
148       if (cur->ch[0] != nullptr) _query_nex(cur->ch[0], val);
149       return q_nex_tmp;
150     }
151     return NIL;
152   }
153
154  public:
155   void insert(int val) { _insert(root, val); }
156
157   void del(int val) { _del(root, val); }
158
159   int query_rank(int val) { return _query_rank(root, val); }
160
161   int query_val(int rank) { return _query_val(root, rank); }
162
163   int query_prev(int val) { return _query_prev(root, val); }
164
165   int query_nex(int val) { return _query_nex(root, val); }
166 };
167
168 Treap tr;
169
170 int main() {
171   srand(0);
```

```
  int t;
  scanf("%d", &t);
  while (t--) {
    int mode;
    int num;
    scanf("%d%d", &mode, &num);
    switch (mode) {
      case 1:
        tr.insert(num);
        break;
      case 2:
        tr.del(num);
        break;
      case 3:
        printf("%d\n", tr.query_rank(num));
        break;
      case 4:
        printf("%d\n", tr.query_val(num));
        break;
      case 5:
        printf("%d\n", tr.query_prev(num));
        break;
      case 6:
        printf("%d\n", tr.query_nex(num));
        break;
    }
  }
}
```

数组实现

以下是 bzoj 普通平衡树模板代码，使用数组实现.

完整代码

```
#include <iostream>

constexpr int MAXN = 100005;
constexpr int INF = 1 << 30;

int n;

struct treap {  // 直接维护成数据结构，可以直接用
  int l[MAXN], r[MAXN], val[MAXN], rnd[MAXN], size_[MAXN], w[MAXN];
  int sz, ans, rt;

  void pushup(int x) { size_[x] = size_[l[x]] + size_[r[x]] + w[x]; }

  void lrotate(int &k) {
    int t = r[k];
    r[k] = l[t];
    l[t] = k;
```

```
    size_[t] = size_[k];
    pushup(k);
    k = t;
  }

  void rrotate(int &k) {
    int t = l[k];
    l[k] = r[t];
    r[t] = k;
    size_[t] = size_[k];
    pushup(k);
    k = t;
  }

  void insert(int &k, int x) {  // 插入
    if (!k) {
      sz++;
      k = sz;
      size_[k] = 1;
      w[k] = 1;
      val[k] = x;
      rnd[k] = rand();
      return;
    }
    size_[k]++;
    if (val[k] == x) {
      w[k]++;
    } else if (val[k] < x) {
      insert(r[k], x);
      if (rnd[r[k]] < rnd[k]) lrotate(k);
    } else {
      insert(l[k], x);
      if (rnd[l[k]] < rnd[k]) rrotate(k);
    }
  }

  bool del(int &k, int x) {  // 删除节点
    if (!k) return false;
    if (val[k] == x) {
      if (w[k] > 1) {
        w[k]--;
        size_[k]--;
        return true;
      }
      if (l[k] == 0 || r[k] == 0) {
        k = l[k] + r[k];
        return true;
      } else if (rnd[l[k]] < rnd[r[k]]) {
        rrotate(k);
        return del(k, x);
      } else {
        lrotate(k);
```

```
70          return del(k, x);
71        }
72      } else if (val[k] < x) {
73        bool succ = del(r[k], x);
74        if (succ) size_[k]--;
75        return succ;
76      } else {
77        bool succ = del(l[k], x);
78        if (succ) size_[k]--;
79        return succ;
80      }
81    }
82
83    int queryrank(int k, int x) {
84      if (!k) return 0;
85      if (val[k] == x)
86        return size_[l[k]] + 1;
87      else if (x > val[k]) {
88        return size_[l[k]] + w[k] + queryrank(r[k], x);
89      } else
90        return queryrank(l[k], x);
91    }
92
93    int querynum(int k, int x) {
94      if (!k) return 0;
95      if (x <= size_[l[k]])
96        return querynum(l[k], x);
97      else if (x > size_[l[k]] + w[k])
98        return querynum(r[k], x - size_[l[k]] - w[k]);
99      else
100       return val[k];
101   }
102
103   void querypre(int k, int x) {
104     if (!k) return;
105     if (val[k] < x)
106       ans = k, querypre(r[k], x);
107     else
108       querypre(l[k], x);
109   }
110
111   void querysub(int k, int x) {
112     if (!k) return;
113     if (val[k] > x)
114       ans = k, querysub(l[k], x);
115     else
116       querysub(r[k], x);
117   }
118 } T;
119
120 using std::cin;
121 using std::cout;
```

```

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  srand(123);
  cin >> n;
  int opt, x;
  for (int i = 1; i <= n; i++) {
    cin >> opt >> x;
    if (opt == 1)
      T.insert(T.rt, x);
    else if (opt == 2)
      T.del(T.rt, x);
    else if (opt == 3) {
      cout << T.queryrank(T.rt, x) << '\n';
    } else if (opt == 4) {
      cout << T.querynum(T.rt, x) << '\n';
    } else if (opt == 5) {
      T.ans = 0;
      T.querypre(T.rt, x);
      cout << T.val[T.ans] << '\n';
    } else if (opt == 6) {
      T.ans = 0;
      T.querysub(T.rt, x);
      cout << T.val[T.ans] << '\n';
    }
  }
  return 0;
}
```

无旋 treap

指针实现

完整代码

以下是前文讲解的代码的完整版本，是普通平衡树的模板代码.

```

// author: (ttzytt)[ttzytt.com]
#include <cstdio>
#include <cstdlib>
#include <ctime>
#include <tuple>
using namespace std;

struct Node {
  Node *ch[2];
  int val, prio;
  int cnt;
  int siz;

  Node(int _val) : val(_val), cnt(1), siz(1) {
```

```
    ch[0] = ch[1] = nullptr;
    prio = rand();
  }

  Node(Node *_node) {
    val = _node->val, prio = _node->prio, cnt = _node->cnt, siz = _node->siz;
  }

  void upd_siz() {
    siz = cnt;
    if (ch[0] != nullptr) siz += ch[0]->siz;
    if (ch[1] != nullptr) siz += ch[1]->siz;
  }
};

struct none_rot_treap {
#define _3 second.second
#define _2 second.first
  Node *root;

  pair<Node *, Node *> split(Node *cur, int key) {
    if (cur == nullptr) return {nullptr, nullptr};
    if (cur->val <= key) {
      auto temp = split(cur->ch[1], key);
      cur->ch[1] = temp.first;
      cur->upd_siz();
      return {cur, temp.second};
    } else {
      auto temp = split(cur->ch[0], key);
      cur->ch[0] = temp.second;
      cur->upd_siz();
      return {temp.first, cur};
    }
  }

  tuple<Node *, Node *, Node *> split_by_rk(Node *cur, int rk) {
    if (cur == nullptr) return {nullptr, nullptr, nullptr};
    int ls_siz = cur->ch[0] == nullptr ? 0 : cur->ch[0]->siz;
    if (rk <= ls_siz) {
      Node *l, *mid, *r;
      tie(l, mid, r) = split_by_rk(cur->ch[0], rk);
      cur->ch[0] = r;
      cur->upd_siz();
      return {l, mid, cur};
    } else if (rk <= ls_siz + cur->cnt) {
      Node *lt = cur->ch[0];
      Node *rt = cur->ch[1];
      cur->ch[0] = cur->ch[1] = nullptr;
      return {lt, cur, rt};
    } else {
      Node *l, *mid, *r;
```

```
      tie(l, mid, r) = split_by_rk(cur->ch[1], rk - ls_siz - cur->cnt);
      cur->ch[1] = l;
      cur->upd_siz();
      return {cur, mid, r};
    }
  }

  Node *merge(Node *u, Node *v) {
    if (u == nullptr && v == nullptr) return nullptr;
    if (u != nullptr && v == nullptr) return u;
    if (v != nullptr && u == nullptr) return v;
    if (u->prio < v->prio) {
      u->ch[1] = merge(u->ch[1], v);
      u->upd_siz();
      return u;
    } else {
      v->ch[0] = merge(u, v->ch[0]);
      v->upd_siz();
      return v;
    }
  }

  void insert(int val) {
    auto temp = split(root, val);
    auto l_tr = split(temp.first, val - 1);
    Node *new_node;
    if (l_tr.second == nullptr) {
      new_node = new Node(val);
    } else {
      l_tr.second->cnt++;
      l_tr.second->upd_siz();
    }
    Node *l_tr_combined =
        merge(l_tr.first, l_tr.second == nullptr ? new_node : l_tr.second);
    root = merge(l_tr_combined, temp.second);
  }

  void del(int val) {
    auto temp = split(root, val);
    auto l_tr = split(temp.first, val - 1);
    if (l_tr.second->cnt > 1) {
      l_tr.second->cnt--;
      l_tr.second->upd_siz();
      l_tr.first = merge(l_tr.first, l_tr.second);
    } else {
      if (temp.first == l_tr.second) {
        temp.first = nullptr;
      }
      delete l_tr.second;
      l_tr.second = nullptr;
    }
    root = merge(l_tr.first, temp.second);
```

```
  }

  int qrank_by_val(Node *cur, int val) {
    auto temp = split(cur, val - 1);
    int ret = (temp.first == nullptr ? 0 : temp.first->siz) + 1;
    root = merge(temp.first, temp.second);
    return ret;
  }

  int qval_by_rank(Node *cur, int rk) {
    Node *l, *mid, *r;
    tie(l, mid, r) = split_by_rk(cur, rk);
    int ret = mid->val;
    root = merge(merge(l, mid), r);
    return ret;
  }

  int qprev(int val) {
    auto temp = split(root, val - 1);
    int ret = qval_by_rank(temp.first, temp.first->siz);
    root = merge(temp.first, temp.second);
    return ret;
  }

  int qnex(int val) {
    auto temp = split(root, val);
    int ret = qval_by_rank(temp.second, 1);
    root = merge(temp.first, temp.second);
    return ret;
  }
};

none_rot_treap tr;

int main() {
  srand(time(nullptr));
  int t;
  scanf("%d", &t);
  while (t--) {
    int mode;
    int num;
    scanf("%d%d", &mode, &num);
    switch (mode) {
      case 1:
        tr.insert(num);
        break;
      case 2:
        tr.del(num);
        break;
      case 3:
        printf("%d\n", tr.qrank_by_val(tr.root, num));
        break;
```

```
171       case 4:
172         printf("%d\n", tr.qval_by_rank(tr.root, num));
173         break;
174       case 5:
175         printf("%d\n", tr.qprev(num));
176         break;
177       case 6:
178         printf("%d\n", tr.qnex(num));
179         break;
180     }
181   }
182 }
```

无旋 treap 的区间操作

指针实现

完整代码

以下是前文讲解的代码的完整版本，是文艺平衡树题目的模板代码.

```
1
2  // author: (ttzytt)[ttzytt.com]
3  #include <cstdlib>
4  #include <ctime>
5  #include <iostream>
6  using namespace std;
7
8  // 参考: https://www.cnblogs.com/Equinox-Flower/p/10785292.html
9  struct Node {
10   Node* ch[2];
11   int val, prio;
12   int cnt;
13   int siz;
14   bool to_rev = false;  // 需要把这个子树下的每一个节点都翻转过来
15
16   Node(int _val) : val(_val), cnt(1), siz(1) {
17     ch[0] = ch[1] = nullptr;
18     prio = rand();
19   }
20
21   int upd_siz() {
22     siz = cnt;
23     if (ch[0] != nullptr) siz += ch[0]->siz;
24     if (ch[1] != nullptr) siz += ch[1]->siz;
25     return siz;
26   }
27
28   void pushdown() {
29     swap(ch[0], ch[1]);
30     if (ch[0] != nullptr) ch[0]->to_rev ^= 1;
31     // 如果原来子节点也要翻转，那两次翻转就抵消了，如果子节点不翻转，那这个
```

```
32      //  tag 就需要继续被 push 到子节点上
33      if (ch[1] != nullptr) ch[1]->to_rev ^= 1;
34      to_rev = false;
35    }
36
37    void check_tag() {
38      if (to_rev) pushdown();
39    }
40  };
41
42  struct Seg_treap {
43    Node* root;
44  #define siz(_) (_ == nullptr ? 0 : _->siz)
45
46    pair<Node*, Node*> split(Node* cur, int sz) {
47      // 按照树的大小划分
48      if (cur == nullptr) return {nullptr, nullptr};
49      cur->check_tag();
50      if (sz <= siz(cur->ch[0])) {
51        // 左边的子树就够了
52        auto temp = split(cur->ch[0], sz);
53        // 左边的子树不一定全部需要，temp.second 是不需要的
54        cur->ch[0] = temp.second;
55        cur->upd_siz();
56        return {temp.first, cur};
57      } else {
58        // 左边的加上右边的一部分（当然也包括这个节点本身）
59        auto temp = split(cur->ch[1], sz - siz(cur->ch[0]) - 1);
60        cur->ch[1] = temp.first;
61        cur->upd_siz();
62        return {cur, temp.second};
63      }
64    }
65
66    Node* merge(Node* sm, Node* bg) {
67      // small, big
68      if (sm == nullptr && bg == nullptr) return nullptr;
69      if (sm != nullptr && bg == nullptr) return sm;
70      if (sm == nullptr && bg != nullptr) return bg;
71      sm->check_tag(), bg->check_tag();
72      if (sm->prio < bg->prio) {
73        sm->ch[1] = merge(sm->ch[1], bg);
74        sm->upd_siz();
75        return sm;
76      } else {
77        bg->ch[0] = merge(sm, bg->ch[0]);
78        bg->upd_siz();
79        return bg;
80      }
81    }
82
83    void insert(int val) {
```

```
    auto temp = split(root, val);
    auto l_tr = split(temp.first, val - 1);
    Node* new_node;
    if (l_tr.second == nullptr) new_node = new Node(val);
    Node* l_tr_combined =
        merge(l_tr.first, l_tr.second == nullptr ? new_node : l_tr.second);
    root = merge(l_tr_combined, temp.second);
  }

  void seg_rev(int l, int r) {
    // 这里的 less 和 more 是相对于 l 的
    auto less = split(root, l - 1);
    // 所有小于等于 l - 1 的会在 less 的左边
    auto more = split(less.second, r - l + 1);
    // 拿出从 l 开始的前 r - l + 1 个
    more.first->to_rev = true;
    root = merge(less.first, merge(more.first, more.second));
  }

  void print(Node* cur) {
    if (cur == nullptr) return;
    cur->check_tag();
    print(cur->ch[0]);
    cout << cur->val << " ";
    print(cur->ch[1]);
  }
};

Seg_treap tr;

int main() {
  srand(time(nullptr));
  int n, m;
  cin >> n >> m;
  for (int i = 1; i <= n; i++) tr.insert(i);
  while (m--) {
    int l, r;
    cin >> l >> r;
    tr.seg_rev(l, r);
  }
  tr.print(tr.root);
}
```

例题

普通平衡树[8]

文艺平衡树（Splay）[9]

「ZJOI2006」书架[10]

「NOI2005」维护数列[11]

CF 702F T-Shirts[12]

参考资料与注释

[1] 维基百科 treap 词条的配图 
[2] https://charleswu.site/archives/1051 
[3] 模板题 
[4] https://www.cnblogs.com/Equinox-Flower/p/10785292.html 
[5] https://www.cnblogs.com/Equinox-Flower/p/10785292.html 
[6] https://www.luogu.com.cn/blog/85514/fhq-treap-xue-xi-bi-ji 
[7] https://www.cnblogs.com/Equinox-Flower/p/10785292.html 
[8] 普通平衡树 
[9] 文艺平衡树（Splay） 
[10] 「ZJOI2006」书架 
[11] 「NOI2005」维护数列 
[12] CF 702F T-Shirts 

10.15.3 Splay 树

本页面将简要介绍如何用 Splay 维护二叉查找树.

定义

Splay 树，或 **伸展树**，是一种平衡二叉查找树，它通过 **伸展（splay）操作** 不断将某个节点旋转到根节点，使得整棵树仍然满足二叉查找树的性质，能够在均摊 $O(\log n)$ 时间内完成插入、查找和删除操作.[7]

Splay 树由 Daniel Sleator 和 Robert Tarjan 于 1985 年发明.

基本结构与操作

本节讨论 Splay 树的基本结构和它的核心操作，其中最为重要的是伸展操作.

Splay 树是一棵二叉查找树，即树中任意节点都满足性质：左子树任意节点的值 $<$ 该节点的值 $<$ 右子树任意节点的值.

维护信息

本文使用数组模拟指针来实现 Splay 树，需要维护如下信息：

rt	id	fa[i]	ch[i][0/1]	val[i]	cnt[i]	sz[i]
根节点编号	已使用节点个数	父亲	左右儿子编号	节点权值	权值出现次数	子树大小

初始化时，所有信息都置零即可.

辅助操作

首先是一些简单的辅助操作：

- `dir(x)`：判断节点 x 是父亲节点的左儿子还是右儿子；
- `push_up(x)`：在改变节点位置后，根据子节点信息更新节点 x 的信息.

实现

```
bool dir(int x) { return x == ch[fa[x]][1]; }

void push_up(int x) { sz[x] = cnt[x] + sz[ch[x][0]] + sz[ch[x][1]]; }

```

旋转操作

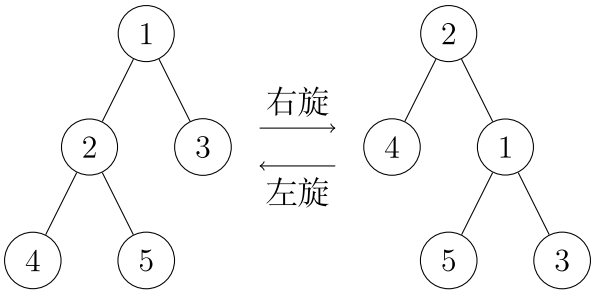
为了使 Splay 保持平衡，需要进行旋转操作. 旋转的作用是将某个节点上移一个位置.

[7] Splay 树只保证均摊复杂度为 $O(\log n)$. 它不维护 AVL 树或红黑树那样的平衡条件，甚至可能在一系列操作后退化为链. 因此，单次操作的最坏复杂度为 $O(n)$.

旋转需要保证：

- 整棵 Splay 的中序遍历不变（不能破坏二叉查找树的性质）；
- 受影响的节点维护的信息依然正确有效；
- `rt` 必须指向旋转后的根节点.

在 Splay 中旋转分为两种：左旋和右旋.



观察图示可知，如果要通过旋转将节点 x（左旋时的 1 和右旋时的 2）上移，则旋转的方向由该节点是其父节点的左节点还是右节点唯一确定. 因此，实现旋转操作时，只需要将要上移的节点 x 传入即可.

具体分析旋转步骤：（假设需要上移的节点为 x，以右旋为例）

1. 首先，记录节点 x 的父节点 y，以及 y 的父节点 z（可能为空），并记录 x 是 y 的左子节点还是右子节点；
2. 按照旋转后的树中自下向上的顺序，依次更新 y 的左子节点为 x 的右子节点，x 的右子节点为 y，以及若 z 非空，z 的子节点为 x；
3. 按照同样的顺序，依次更新当前 y 的左子节点（若存在）的父节点为 y，y 的父节点为 x，以及 x 的父节点为 z；
4. 自下而上维护节点信息.

实现

```
void rotate(int x) {
  int y = fa[x], z = fa[y];
  bool r = dir(x);
  ch[y][r] = ch[x][!r];
  ch[x][!r] = y;
  if (z) ch[z][dir(y)] = x;
  if (ch[y][r]) fa[ch[y][r]] = y;
  fa[y] = x;
  fa[x] = z;
  push_up(y);
  push_up(x);
}

```

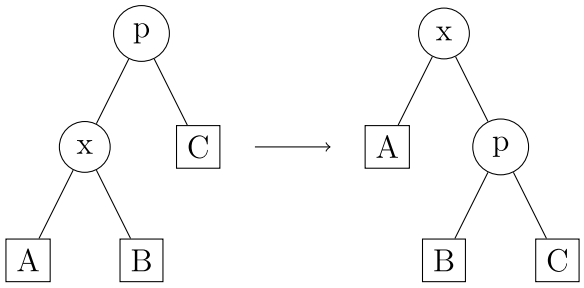
在实现所有函数时，都应注意不要修改节点 0 的信息.

伸展操作

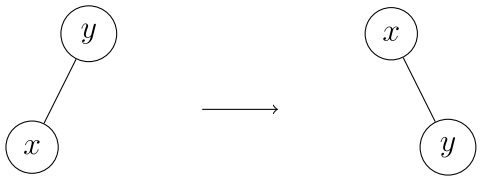
Splay 树要求每访问一个节点 x 后都要强制将其旋转到根节点. 该操作也称为伸展操作.

设刚访问的节点为 x. 要做伸展操作，就是要对 x 做一系列的 **伸展步骤**. 每次对 x 做一次伸展步骤，x 到根节点的距离都会更近. 定义 p 为 x 的父节点. 伸展步骤有三种：

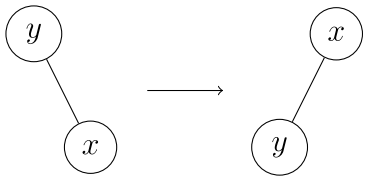
1. **zig**: 在 p 是根节点时操作. Splay 树会根据 x 和 p 间的边旋转. **zig** 用于处理深度的奇偶性问题，仅当 x 在伸展操作开始时具有奇数深度时，才作为伸展操作的最后一步执行.



即直接将 x 右旋或左旋（图 1, 2）.

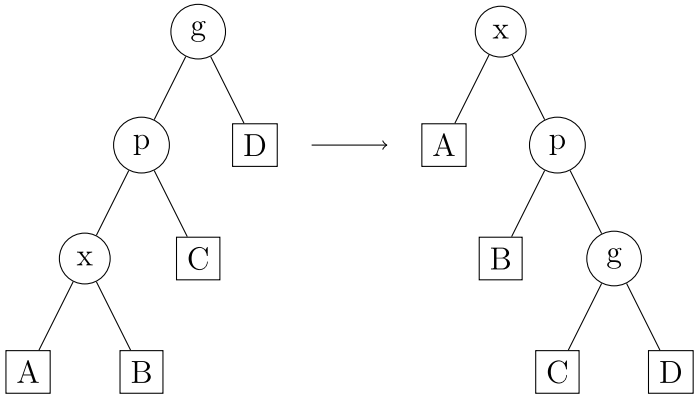


1

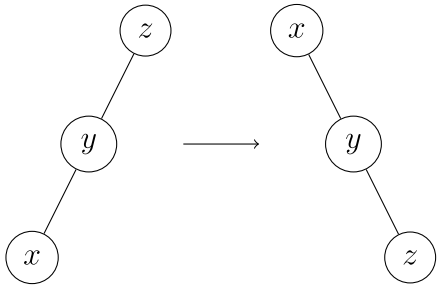


2

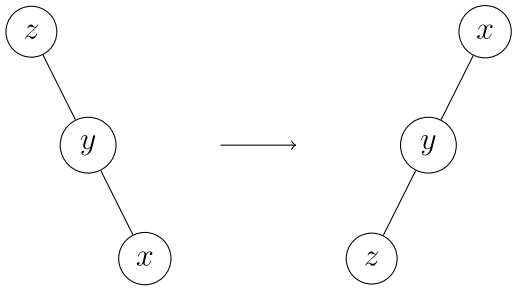
2. **zig-zig**: 在 p 不是根节点且 x 和 p 都是右侧子节点或都是左侧子节点时操作. 下方例图显示了 x 和 p 都是左侧子节点时的情况. Splay 树首先按照连接 p 与其父节点 g 的边旋转，然后按照连接 x 和 p 的边旋转.



即首先将 p 右旋或左旋，然后将 x 右旋或左旋（图 3, 4）.

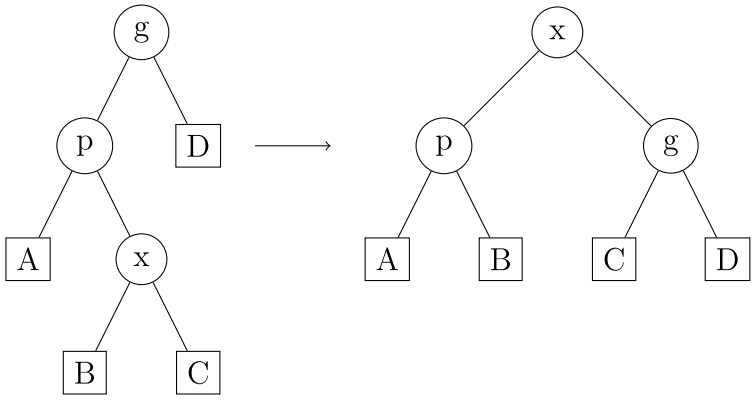


3

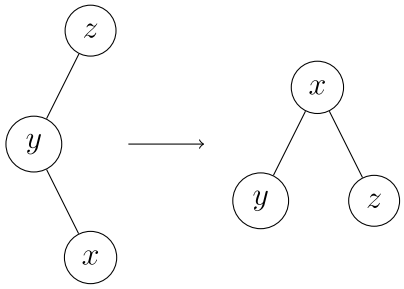


4

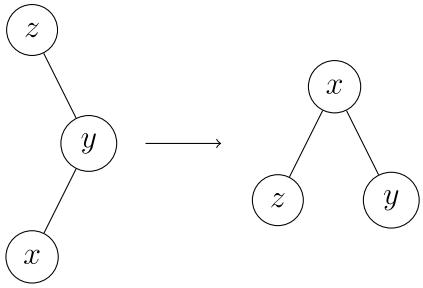
3. **zig-zag**: 在 p 不是根节点且 x 和 p 一个是右侧子节点一个是左侧子节点时操作. Splay 树首先按 p 和 x 之间的边旋转，然后按旋转后 x 和 g 之间新生成的边旋转.



即将 x 先左旋再右旋或先右旋再左旋（图 5, 6）.



5



6

> **Tip**
>
> 请读者尝试自行模拟 6 种旋转情况，以理解伸展操作的基本思想.

比较三种伸展步骤可知，要区分此时应使用哪种操作，关键是要判断 x 是否是根节点的子节点，以及 x 和它父节点是否在各自的父节点同侧.

此处提供的实现，可以指定任意根节点 z，并将它的子树内任意节点 x 上移至 z 处：

1. 首先记录根节点 z 的父节点 w，从而可以利用 `fa[x] == w` 判断 x 已经位于根节点处；
2. 记录 x 当前的父节点 y，如果 y 和 w 相同，说明 x 已经到达根节点；
3. 否则，利用 `fa[y] == w` 判断 y 是否是根节点．如果是，直接做 zig 操作将 x 旋转；如果不是，利用 `dir(x) == dir(y)` 判断使用 zig-zig 还是 zig-zag，前者先旋转 y 再旋转 x，后者直接旋转两次 x．

实现

```
void splay(int& z, int x) {
  int w = fa[z];
  for (int y; (y = fa[x]) != w; rotate(x)) {
    if (fa[y] != w) rotate(dir(x) == dir(y) ? y : x);
  }
  z = x;
}

```

伸展操作是 Splay 树的核心操作，也是它的时间复杂度能够得到保证的关键步骤．请务必保证每次向下访问节点后，都进行一次伸展操作．

另外，伸展操作会将当前节点 x 到根节点 z 的路径上的所有节点信息自下而上地更新一遍．正是因为这一点，才可以修改非根节点，再通过伸展操作将它上移至根来完成整个树的信息更新．

时间复杂度

对大小为 n 的 Splay 树做 m 次伸展操作的复杂度是 $O((m+n)\log n)$ 的，单次均摊复杂度是 $O(\log n)$ 的．

基于势能分析的复杂度证明

为此只需分析 **zig**、**zig-zig** 和 **zig-zag** 三种操作的复杂度．为此，我们采用 **势能分析法**，通过研究势能的变化来推导操作的均摊复杂度．假设对一棵包含 n 个节点的 Splay 树进行了 m 次伸展操作，可以通过如下方式进行分析：

定义：

1. **单个节点的势能**：$w(x)=\log(\text{size}(x))$，其中 $\text{size}(x)$ 表示以节点 x 为根的子树大小．
2. **整棵树的势能**：$\varphi=\sum w(x)$，即树中所有节点势能的总和，初始势能满足 $\varphi_0 \le n\log n$．
3. **第 i 次操作的均摊成本**：$c_i=t_i+\varphi_i-\varphi_{i-1}$，其中 t_i 为实际操作代价，φ_i 和 φ_{i-1} 分别为操作后和操作前的势能．

性质：

1. 如果 p 是 x 的父节点，则有 $w(p)\ge w(x)$，即父节点的势能不小于子节点的势能．
2. 由于根节点的子树大小在操作前后保持不变，因此根节点的势能在操作过程中不变．
3. 如果 $\text{size}(p)\ge\text{size}(x)+\text{size}(y)$，那么有 $2w(p)-w(x)-w(y)\ge 2$．

性质 3 的证明

根据均值不等式可知

$$\begin{aligned}2w(p)-w(x)-w(y)&=\log\frac{\mathrm{size}(p)^2}{\mathrm{size}(x)\cdot\mathrm{size}(y)}\\&\geq\log\frac{(\mathrm{size}(x)+\mathrm{size}(y))^2}{\mathrm{size}(x)\cdot\mathrm{size}(y)}\\&\geq\log 4\\&=2.\end{aligned}$$

接下来, 分别对 **zig**、**zig-zig** 和 **zig-zag** 操作进行势能分析. 设操作前后的节点 x 的势能分别是 $w(x)$ 和 $w'(x)$. 节点的记号与 上文→10@p1977 一致.

zig: 根据性质 1 和 2, 有 $w(p)=w'(x)$, 且 $w'(x)\geq w'(p)$. 由此, 均摊成本为

$$\begin{aligned}c_i&=1+w'(x)+w'(p)-w(x)-w(p)\\&=1+w'(p)-w(x)\\&\leq 1+w'(x)-w(x).\end{aligned}$$

zig-zig: 根据性质 1 和 2, 有 $w(g)=w'(x)$, 且 $w'(x)\geq w'(p)$, $w(x)\leq w(p)$. 因为

$$\begin{aligned}\mathrm{size}(x)&=3+\mathrm{size}(A)+\mathrm{size}(B)+\mathrm{size}(C)+\mathrm{size}(D)\\&>(1+\mathrm{size}(A)+\mathrm{size}(B))+(1+\mathrm{size}(C)+\mathrm{size}(D))\\&=\mathrm{size}(x)+\mathrm{size}(g),\end{aligned}$$

根据性质 3 可得

$$2w'(x)-w(x)-w'(g)\geq 2.$$

由此, 均摊成本为

$$\begin{aligned}c_i&=2+w'(x)+w'(p)+w'(g)-w(x)-w(p)-w(g)\\&=2+w'(p)+w'(g)-w(x)-w(p)\\&\leq(2w'(x)-w(x)-w'(g))+w'(p)+w'(g)-w(x)-w(p)\\&=2(w'(x)-w(x))+w'(p)-w(p)\\&\leq 3(w'(x)-w(x)).\end{aligned}$$

zig-zag: 根据性质 1 和 2, 有 $w(g)=w'(x)$, 且 $w(p)\geq w(x)$. 因为 $\mathrm{size}(x)>\mathrm{size}(p)+\mathrm{size}(g)$, 根据性质 3, 可得

$$2\cdot w'(x)-w'(g)-w'(p)\geq 2.$$

由此, 均摊成本为

$$\begin{aligned}c_i&=2+w'(x)+w'(p)+w'(g)-w(x)-w(p)-w(g)\\&=2+w'(p)+w'(g)-w(x)-w(p)\\&\leq(2w'(x)-w'(g)-w'(p))+w'(p)+w'(g)-w(x)-w(p)\\&=2w'(x)-w(x)-w(p)\\&\leq 2(w'(x)-w(x)).\end{aligned}$$

单次伸展操作:

令 $w^{(j)}(x)=(w^{(j-1)})'(x)$ 且 $w^{(0)}(x)=w(x)$. 假设一次伸展操作共包含 k 个伸展步骤, 最终将节点 x_1 上移至根节点. 这必然经过若干次 **zig-zig** 和 **zig-zag** 操作和至多一次 **zig** 操作, 前两种操作的均摊成本均不超过 $3(w'(x)-w(x))$, 而最后一次操作的均摊成本不超过 $3(w'(x)-w(x))+1$, 所以总的均摊成本不超过

$$3(w^{(k)}(x_1)-w^{(0)}(x_1))+1\leq 3\log n+1.$$

因此，一次伸展操作的均摊复杂度是 $O(\log n)$ 的．从而，基于伸展的插入、查询、删除等操作的时间复杂度也为均摊 $O(\log n)$．

结论：

在进行 m 次伸展操作之后，实际成本

$$\begin{aligned}\sum_{i=1}^{m} t_i &= \sum_{i=1}^{m}(c_i + \varphi_{i-1} - \varphi_i)\\ &= \sum_{i=1}^{m} c_i + \varphi_0 - \varphi_m\\ &\le m(3\log n + 1) + n\log n.\end{aligned}$$

因此，m 次伸展操作的实际时间复杂度是 $O((m+n)\log n)$ 的．

为什么 Splay 树的再平衡操作可以获得 $O(\log n)$ 的均摊复杂度？

朴素的再平衡思路就是对节点反复进行旋转操作使其上升，直到它成为根节点．这种朴素思路的问题在于，对于所有子节点都是左（右）节点的链状树来说，它相当于反复进行 zig 操作，因而 zig 操作的均摊复杂度中的常数项 1 会不断累积，造成最终的均摊复杂度达到 $O(n)$ 级别．Splay 树的再平衡操作的设计，避免了连续 zig 的情形中的常数累积，使得一次完整的伸展操作中，至多进行一次单独的 zig 操作，从而优化了时间复杂度．

平衡树操作

本节讨论基于 Splay 树实现平衡树的常见操作的方法．其中，较为重要的是按照值或排名查找元素，它们可以将某个特定的元素找到，并上移至根节点处，以便后续处理．

作为例子，本节将讨论模板题目 普通平衡树[1] 的实现．

按照值查找

作为二叉查找树，可以通过值 v 查找到相应的节点，只需要将待查找的值 v 和当前节点的值比较即可，找到后将该元素上移至根部即可．

应注意，经常存在树中不存在相应的节点的情形．对于这种情形，要记录最后一个访问的节点（即实现中的 y），并将 y 上移至根部．此时，节点 y 存储的值必然要么是所有小于 v 的元素中最大的（即 v 的前驱），要么是所有大于 v 的元素中最小的（即 v 的后继）．这是因为查找过程保证，左子树总是存储小于 v 的值，而右子树总是存储大于 v 的值．

实现

```
void find(int& z, int v) {
  int x = z, y = fa[x];
  for (; x && val[x] != v; x = ch[y = x][v > val[x]]);
  splay(z, x ? x : y);
}

```

该实现允许指定任何节点 z 作为根节点，并在它的子树内按值查找．

按照排名访问

因为记录了子树大小信息，所以 Splay 树还可以通过排名访问元素，即查找树中第 k 小的元素．

设 k 为剩余排名，具体步骤如下：

- 如果左子树非空且剩余排名 k 不大于左子树的大小，那么向左子树查找；
- 否则，如果 k 不大于左子树的大小与根节点的权值出现次数之和，那么根节点就是要寻找的；
- 否则，将 k 减去左子树的大小与根节点的权值出现次数之和，继续向右子树查找；
- 将最终找到的元素上移至根部.

实现

```
void loc(int& z, int k) {
  int x = z;
  for (;;) {
    if (sz[ch[x][0]] >= k) {
      x = ch[x][0];
    } else if (sz[ch[x][0]] + cnt[x] >= k) {
      break;
    } else {
      k -= sz[ch[x][0]] + cnt[x];
      x = ch[x][1];
    }
  }
  splay(z, x);
}

```

该实现需要保证排名 k 不超过根 z 处的树大小.

模板题目中操作 4 要求按照排名返回值，直接调用该方法，并返回值即可.

实现

```
int find_kth(int k) {
  if (k > sz[rt]) return -1;
  loc(rt, k);
  return val[rt];
}

```

合并操作

有些时候需要合并两棵 Splay 树.

设两棵树的根节点分别为 x 和 y，那么为了保证结果仍是二叉查找树，需要要求 x 树中的最大值小于 y 树中的最小值. 这一条件通常都可以满足，因为两棵树往往是从更大的子树中分裂出的.

合并操作如下：

- 如果 x 和 y 其中之一或两者都为空树，直接返回不为空的那一棵树的根节点或空树；
- 否则，通过 `loc(y, 1)` 将 y 树中的最小值上移至根 y 处，再将它的左节点（此时必然为空）设置为 x，并更新节点信息，返回节点 y.

实现

```
int merge(int x, int y) {
  if (!x || !y) return x | y;
  loc(y, 1);
```

```
  ch[y][0] = x;
  fa[x] = y;
  push_up(y);
  return y;
}

```

分裂操作类似. 因而，Splay 树可以模拟 无旋 treap→10.15.2@p2221 的思路做各种操作，包括区间操作. 后文→10@p1977 会介绍更具有 Splay 树风格的区间操作处理方法.

插入操作

插入操作是一个比较复杂的过程. 具体步骤如下：（假设插入的值为 v）

- 类似按值查找的过程，根据 v 向下查找到存储 v 的节点或者空节点，过程中记录父节点 y；
- 如果存在存储 v 的节点 x，直接更新信息，否则就新建节点 x；
- 做伸展操作，将最后一个节点 x 上移至根部.

实现

```
void insert(int v) {
  int x = rt, y = 0;
  for (; x && val[x] != v; x = ch[y = x][v > val[x]]);
  if (x) {
    ++cnt[x];
    ++sz[x];
  } else {
    x = ++id;
    val[x] = v;
    cnt[x] = sz[x] = 1;
    fa[x] = y;
    if (y) ch[y][v > val[y]] = x;
  }
  splay(rt, x);
}

```

该实现允许直接向空树内插入值. 若不想处理空树，可以在树中提前插入哑节点.

删除操作

删除操作也是一个比较复杂的操作. 具体步骤如下：（假设删除的值为 v）

- 首先按照值 v 查找存储它的节点，并上移至根部；
- 如果不存在存储它的节点，直接返回；（上一步已经做了伸展操作）
- 否则，更新节点信息；
- 如果根节点的权值出现次数减为零，就删除该节点，即合并左右子树作为新的根节点，注意合并前需要更新两个子树的根的父节点为空.

实现

```
bool remove(int v) {
  find(rt, v);
```

```
  if (!rt || val[rt] != v) return false;
  --cnt[rt];
  --sz[rt];
  if (!cnt[rt]) {
    int x = ch[rt][0];
    int y = ch[rt][1];
    fa[x] = fa[y] = 0;
    rt = merge(x, y);
  }
  return true;
}

```

查询排名

直接按照值 v 访问节点（并上移至根），然后返回相应的排名即可.

注意，当 v 不存在时，方法 `find(rt, v)` 返回的根和 v 的大小关系无法确定，需要单独讨论.

实现

```
int find_rank(int v) {
  find(rt, v);
  return sz[ch[rt][0]] + (val[rt] < v ? cnt[rt] : 0) + 1;
}

```

查询前驱

前驱定义为小于 v 的最大的数. 具体步骤如下:

- 按照值 v 访问节点（并上移至根部）;
- 如果根部的值小于 v，那么它必然是最大的那个，直接返回;
- 否则，在左子树中找到最大值，并上移至根部.

最后一步相当于直接调用 `loc(ch[rt][0], sz[ch[rt][0]])`，只是省去了不必要的判断.

实现

```
int find_prev(int v) {
  find(rt, v);
  if (rt && val[rt] < v) return val[rt];
  int x = ch[rt][0];
  if (!x) return -1;
  for (; ch[x][1]; x = ch[x][1]);
  splay(rt, x);
  return val[rt];
}

```

该实现允许前驱不存在，此时返回 -1.

查询后继

后继定义为大于 v 的最小的数. 查询方法和前驱类似, 只是将左子树的最大值换成了右子树的最小值, 即调用 `loc(ch[rt][1], 1)`.

实现

```
int find_next(int v) {
  find(rt, v);
  if (rt && val[rt] > v) return val[rt];
  int x = ch[rt][1];
  if (!x) return -1;
  for (; ch[x][0]; x = ch[x][0]);
  splay(rt, x);
  return val[rt];
}

```

参考实现

本节的最后，给出模板题目 普通平衡树[2] 的参考实现.

参考实现

```
#include <iostream>

constexpr int N = 2e6;
int id, rt;
int fa[N], val[N], cnt[N], sz[N], ch[N][2];

bool dir(int x) { return x == ch[fa[x]][1]; }

void push_up(int x) { sz[x] = cnt[x] + sz[ch[x][0]] + sz[ch[x][1]]; }

void rotate(int x) {
  int y = fa[x], z = fa[y];
  bool r = dir(x);
  ch[y][r] = ch[x][!r];
  ch[x][!r] = y;
  if (z) ch[z][dir(y)] = x;
  if (ch[y][r]) fa[ch[y][r]] = y;
  fa[y] = x;
  fa[x] = z;
  push_up(y);
  push_up(x);
}

void splay(int& z, int x) {
  int w = fa[z];
  for (int y; (y = fa[x]) != w; rotate(x)) {
    if (fa[y] != w) rotate(dir(x) == dir(y) ? y : x);
  }
  z = x;
}
```

```
31
32 void find(int& z, int v) {
33   int x = z, y = fa[x];
34   for (; x && val[x] != v; x = ch[y = x][v > val[x]]);
35   splay(z, x ? x : y);
36 }
37
38 void loc(int& z, int k) {
39   int x = z;
40   for (;;) {
41     if (sz[ch[x][0]] >= k) {
42       x = ch[x][0];
43     } else if (sz[ch[x][0]] + cnt[x] >= k) {
44       break;
45     } else {
46       k -= sz[ch[x][0]] + cnt[x];
47       x = ch[x][1];
48     }
49   }
50   splay(z, x);
51 }
52
53 int merge(int x, int y) {
54   if (!x || !y) return x | y;
55   loc(y, 1);
56   ch[y][0] = x;
57   fa[x] = y;
58   push_up(y);
59   return y;
60 }
61
62 void insert(int v) {
63   int x = rt, y = 0;
64   for (; x && val[x] != v; x = ch[y = x][v > val[x]]);
65   if (x) {
66     ++cnt[x];
67     ++sz[x];
68   } else {
69     x = ++id;
70     val[x] = v;
71     cnt[x] = sz[x] = 1;
72     fa[x] = y;
73     if (y) ch[y][v > val[y]] = x;
74   }
75   splay(rt, x);
76 }
77
78 bool remove(int v) {
79   find(rt, v);
80   if (!rt || val[rt] != v) return false;
81   --cnt[rt];
82   --sz[rt];
```

```
  if (!cnt[rt]) {
    int x = ch[rt][0];
    int y = ch[rt][1];
    fa[x] = fa[y] = 0;
    rt = merge(x, y);
  }
  return true;
}

int find_rank(int v) {
  find(rt, v);
  return sz[ch[rt][0]] + (val[rt] < v ? cnt[rt] : 0) + 1;
}

int find_kth(int k) {
  if (k > sz[rt]) return -1;
  loc(rt, k);
  return val[rt];
}

int find_prev(int v) {
  find(rt, v);
  if (rt && val[rt] < v) return val[rt];
  int x = ch[rt][0];
  if (!x) return -1;
  for (; ch[x][1]; x = ch[x][1]);
  splay(rt, x);
  return val[rt];
}

int find_next(int v) {
  find(rt, v);
  if (rt && val[rt] > v) return val[rt];
  int x = ch[rt][1];
  if (!x) return -1;
  for (; ch[x][0]; x = ch[x][0]);
  splay(rt, x);
  return val[rt];
}

int main() {
  int n;
  std::cin >> n;
  for (; n; --n) {
    int op, x;
    std::cin >> op >> x;
    switch (op) {
      case 1:
        insert(x);
        break;
      case 2:
        remove(x);
```

```
135         break;
136       case 3:
137         std::cout << find_rank(x) << '\n';
138         break;
139       case 4:
140         std::cout << find_kth(x) << '\n';
141         break;
142       case 5:
143         std::cout << find_prev(x) << '\n';
144         break;
145       case 6:
146         std::cout << find_next(x) << '\n';
147         break;
148     }
149   }
150   return 0;
151 }
152
```

序列操作

Splay 树也可以运用在序列上，用于维护区间信息．与线段树对比，Splay 树常数较大，但是支持更复杂的序列操作，如区间翻转等．上文提到 Splay 树同样支持分裂和合并操作，因而可以模拟 无旋 treap→10.15.2@p2221 进行区间操作，在此不再过多讨论．本节主要讨论基于伸展操作的区间操作实现方法．

将序列建成的 Splay 树有如下性质：

- Splay 树的中序遍历相当于原序列从左到右的遍历；
- Splay 树上的一个节点代表原序列的一个元素；
- Splay 树上的一棵子树，代表原序列的一段区间．

因为有伸展操作，可以快速提取出代表某个区间的 Splay 子树．

作为例子，本节将讨论模板题目 文艺平衡树[3] 的实现．

根据序列建树

在操作之前，需要根据所给的序列先把 Splay 树建出来．根据 Splay 树的特性，直接建出一棵只有左儿子的链即可．时间复杂度是 $O(n)$ 的．

参考实现

```
1  void build(int n) {
2    for (int i = 1; i <= n + 2; ++i) {
3      ++id;
4      ch[id][0] = rt;
5      if (rt) fa[rt] = id;
6      rt = id;
7      val[id] = i - 1;
8    }
9    splay(rt, 1);
10 }
11
```

最后的伸展操作自下而上地更新了节点信息. 为了后文区间操作方便, 序列左右两侧添加了两个哨兵节点.

区间翻转

以区间翻转为例，可以理解区间操作的方法：（设区间为 $[L, R]$）

- 首先将节点 $L-1$ 上移到根节点，再在其右子树中，将节点 $R+1$ 上移到右子树的根节点；
- 此时，设 x 为根节点的右子节点的左子节点，则以 x 为根的子树就对应着区间 $[L, R]$；
- 在 x 处对区间 $[L, R]$ 做操作，并打上懒标记；
- 在 x 处将标记下传一次，然后利用伸展操作将 x 上移到根.

第一步需要的操作就是前文平衡树操作中的「按照排名访问」，因为元素的标号就是它的排名. 因为涉及懒标记的管理，它的实现与上文略有不同.

参考实现

```
void reverse(int l, int r) {
  loc(rt, l);
  loc(ch[rt][1], r - l + 2);
  int x = ch[ch[rt][1]][0];
  lazy_reverse(x);
  push_down(x);
  splay(rt, x);
}

```

最后一步的伸展操作并非为了保证复杂度正确，而是为了更新节点信息. 因为伸展操作涉及到节点 x 的左右子节点，所以之前需要将节点 x 处的标记先下传一次. 当然，仅对于区间翻转操作而言，子区间的翻转不会对祖先节点产生影响，所以省去这一步骤也是正确的. 此处实现保留这两行，是为了说明一般的情形下的操作方法.

懒标记管理

首先，需要辅助函数 `lazy_reverse(x)` 和 `push_down(x)`. 前者交换左右节点，并更新懒标记；后者将标记下传.

参考实现

```
void lazy_reverse(int x) {
  std::swap(ch[x][0], ch[x][1]);
  lz[x] ^= 1;
}

void push_down(int x) {
  if (lz[x]) {
    if (ch[x][0]) lazy_reverse(ch[x][0]);
    if (ch[x][1]) lazy_reverse(ch[x][1]);
    lz[x] = 0;
  }
}

```

然后，只需要在向下经过节点时下传标记即可. 模板题要求的操作比较简单，只有按照排名寻找的操作（即 `loc`）涉及向下访问节点. 注意，需要在函数每次访问一个新的节点 **前** 下传标记.

参考实现

```
void loc(int& z, int k) {
  int x = z;
  for (push_down(x); sz[ch[x][0]] != k - 1; push_down(x)) {
    if (sz[ch[x][0]] >= k) {
      x = ch[x][0];
    } else {
      k -= sz[ch[x][0]] + 1;
      x = ch[x][1];
    }
  }
  splay(z, x);
}

```

因为向下访问节点时已经移除了经过的路径的所有懒标记，所以利用伸展操作上移节点时不再需要处理懒标记. 但是，对于区间操作的那一个节点要谨慎处理：因为它同样位于伸展操作的路径上，但是刚刚操作完，可能存在尚未下传的标记，需要首先下传再做伸展操作，正如同上文所做的那样.

参考实现

本节的最后，给出模板题目 文艺平衡树[4] 的参考实现.

参考实现

```
#include <iostream>

constexpr int N = 2e6;
int id, rt;
int fa[N], val[N], sz[N], lz[N], ch[N][2];

bool dir(int x) { return x == ch[fa[x]][1]; }

void push_up(int x) { sz[x] = 1 + sz[ch[x][0]] + sz[ch[x][1]]; }

void lazy_reverse(int x) {
  std::swap(ch[x][0], ch[x][1]);
  lz[x] ^= 1;
}

void push_down(int x) {
  if (lz[x]) {
    if (ch[x][0]) lazy_reverse(ch[x][0]);
    if (ch[x][1]) lazy_reverse(ch[x][1]);
    lz[x] = 0;
  }
}

void rotate(int x) {
  int y = fa[x], z = fa[y];
  bool r = dir(x);
  ch[y][r] = ch[x][!r];
```

```
  ch[x][!r] = y;
  if (z) ch[z][dir(y)] = x;
  if (ch[y][r]) fa[ch[y][r]] = y;
  fa[y] = x;
  fa[x] = z;
  push_up(y);
  push_up(x);
}

void splay(int& z, int x) {
  int w = fa[z];
  for (int y; (y = fa[x]) != w; rotate(x)) {
    if (fa[y] != w) rotate(dir(x) == dir(y) ? y : x);
  }
  z = x;
}

void loc(int& z, int k) {
  int x = z;
  for (push_down(x); sz[ch[x][0]] != k - 1; push_down(x)) {
    if (sz[ch[x][0]] >= k) {
      x = ch[x][0];
    } else {
      k -= sz[ch[x][0]] + 1;
      x = ch[x][1];
    }
  }
  splay(z, x);
}

void build(int n) {
  for (int i = 1; i <= n + 2; ++i) {
    ++id;
    ch[id][0] = rt;
    if (rt) fa[rt] = id;
    rt = id;
    val[id] = i - 1;
  }
  splay(rt, 1);
}

void reverse(int l, int r) {
  loc(rt, l);
  loc(ch[rt][1], r - l + 2);
  int x = ch[ch[rt][1]][0];
  lazy_reverse(x);
  push_down(x);
  splay(rt, x);
}

void print(int x) {
  if (!x) return;
```

```
  push_down(x);
  print(ch[x][0]);
  std::cout << val[x] << ' ';
  print(ch[x][1]);
}

void print() {
  loc(rt, 1);
  loc(ch[rt][1], sz[rt] - 1);
  print(ch[ch[rt][1]][0]);
}

int main() {
  int n, m;
  std::cin >> n >> m;
  build(n);
  for (; m; --m) {
    int l, r;
    std::cin >> l >> r;
    reverse(l, r);
  }
  print();
  return 0;
}

```

习题

这些题目都是裸的 Splay 树维护二叉查找树：

- 【模板】普通平衡树[5]
- 【模板】文艺平衡树[6]
- 「HNOI2002」营业额统计[7]
- 「HNOI2004」宠物收养所[8]

Splay 树还出现在更复杂的应用场景中：

- 「Cerc2007」robotic sort 机械排序[9]
- 「HNOI2011」括号修复/「JSOI2011」括号序列[10]
- 二逼平衡树（树套树）[11]
- BZOJ 2827 千山鸟飞绝[12]
- 「Lydsy1706 月赛」K 小值查询[13]
- POJ3580 SuperMemo[14]

参考资料与注释

本文部分内容引用自 algocode 算法博客，特别鸣谢！

[1] 普通平衡树 

[2] 普通平衡树 

[3] 文艺平衡树 

[4] 文艺平衡树 

[5] 【模板】普通平衡树 

[6] 【模板】文艺平衡树 

[7]「HNOI2002」营业额统计

[8]「HNOI2004」宠物收养所

[9]「Cerc2007」robotic sort 机械排序

[10]「HNOI2011」括号修复/「JSOI2011」括号序列

[11] 二逼平衡树（树套树）

[12] BZOJ 2827 千山鸟飞绝

[13]「Lydsy1706 月赛」K 小值查询

[14] POJ3580 SuperMemo

10.15.4 WBLT

> **Authors:** hsfzLZH1, cesonic, AtomAlpaca, caijianhong, Persdre, aofall, CoelacanthusHex, Marcythm, shuzhouliu, Tiphereth-A

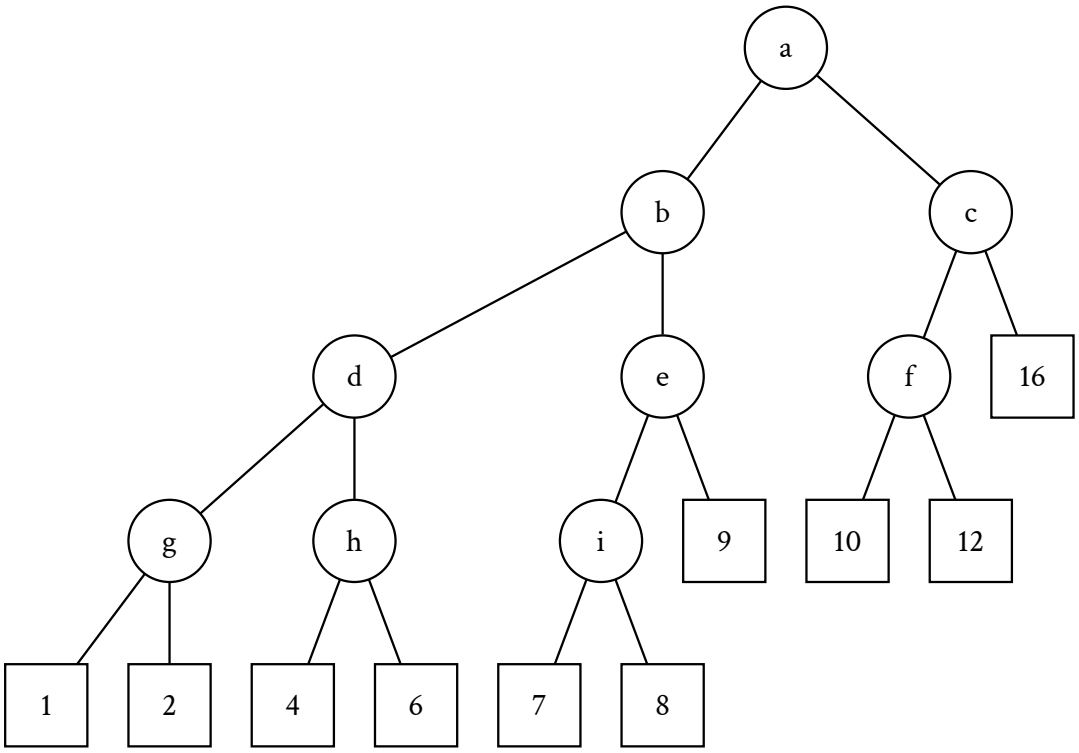
引入

Weight Balanced Leafy Tree，下称 **WBLT**，是一种平衡树，比起其它平衡树主要有实现简单、常数小的优点．它支持区间操作，而且可持久化．

Weight Balanced Leafy Tree 顾名思义是 Weight Balanced Tree 和 Leafy Tree 的结合．

Weight Balanced Tree 的每个节点储存这个节点下子树的大小，并且通过保持左右子树的大小关系在一定范围来保证树高．

Leafy Tree 维护的原始信息仅存储在树的 **叶子节点** 上，而非叶子节点仅用于维护子节点信息和维持数据结构的形态．我们熟知的线段树就是一种 Leafy Tree．



本文的树均指的是二叉的 Leafy Tree，即每个节点的子节点数目只能是 0 或者 2．本文中的 n，指的是树的叶子节点的数目．叶子节点数目为 n 的树，总的节点数量是 $2n-1$，因此，WBLT 占用的空间是 $\Theta(n)$ 的．

基本结构及平衡维护

本节介绍 WBLT 的基本结构，定义树的 α-平衡的概念，并解释如何通过旋转或合并的方式维护树的平衡.

节点信息

要实现一个基本的 WBLT，只需要记录每个节点的如下信息：

- `lc[x]`、`rc[x]`：左、右子节点；
- `sz[x]`：以 x 为根的子树中的叶子节点的数目.

利用 WBLT 实现平衡树，还需要在每个节点处记录与键值相关的信息：

- `val[x]`：节点 x 处的键值.

因为只有叶子节点实际存储键值，所以其他节点处存储的信息是由它们的子节点合并得到的，以方便后续查询.

比如，一种常用的合并方式就是将两个子节点的键值中较大的那个存储于该节点. 这样，每个节点存储的就是以它为根的子树中，所有叶子节点的键值的最大值. 基于此，节点信息的更新方法如下：

参考代码

```
void push_up(int x) {
  sz[x] = sz[ch[x][0]] + sz[ch[x][1]];
  val[x] = val[ch[x][1]];
}

```

当然，如果需要，还可以实现相应的 `push_down(x)` 函数.

辅助函数

除了基本的节点信息维护外，WBLT 通常还需要实现如下辅助函数，用于内存管理：

- `new_node()`：新建节点；
- `del_node(x)`：删除节点 x；
- `new_leaf(v)`：新建以 v 为键值的叶子节点；
- `join(x, y)`：连接子树，即分别以 x、y 为左右子节点，新建节点 z；
- `cut(x)`：拆分子树，即获得节点 x 的两个子节点，并删除节点 x.

如果 WBLT 的实现十分依赖于拆分和连接子树，会建立较多的新节点，并释放等量的旧节点. 如果不及时回收旧的无用节点，会导致空间不再是线性的. 以下是这些辅助函数的数组实现：

参考代码

```
// Return a new empty node.
int new_node() {
  int x = top ? pool[--top] : ++id;
  sz[x] = val[x] = ch[x][0] = ch[x][1] = 0;
  return x;
}

// Release a node for later use.
void del_node(int& x) {
  pool[top++] = x;
  x = 0;
}
```

```
13
14 // Return a new leaf node of value v.
15 int new_leaf(int v) {
16   int x = new_node();
17   val[x] = v;
18   sz[x] = 1;
19   return x;
20 }
21
22 // Return a new node with subtrees x and y.
23 int join(int x, int y) {
24   int z = new_node();
25   ch[z][0] = x;
26   ch[z][1] = y;
27   push_up(z);
28   return z;
29 }
30
31 // Return subtrees of x and release x.
32 auto cut(int& x) {
33   int y = ch[x][0];
34   int z = ch[x][1];
35   del_node(x);
36   return std::make_pair(y, z);
37 }
38
```

封装好这些辅助函数后，数组实现和指针实现在后续函数中就没有区别了.

平衡的概念

对于一个树，可以定义它在一个非叶节点 x 处的 **平衡度** 为

$$\rho(x) = \frac{\min\{w(T_{\text{left}(x)}), w(T_{\text{right}(x)})\}}{w(T_x)}.$$

其中，T_x 表示以 x 为根的子树，$w(\cdot)$ 表示子树的权重（它的叶子节点的数目），而 $\text{left}(x)$ 和 $\text{right}(x)$ 分别表示 x 的左右子节点. 特别地，叶子节点处规定 $\rho(x) = 1/2$.

对于 $\alpha \in (0, 1/2]$，如果某个节点 x 处平衡度 $\rho(x) \geq \alpha$，就称该节点是 α-**平衡** 的. 如果树的每个节点处都是 α-平衡的，就称树是 α-**平衡** 的. 这样的树的集合记作 $BB[\alpha]$. 一棵树是 α-**平衡** 的，当且仅当它是叶子节点，或者它的根节点处是 α-平衡的且它的左右子树都是 α-平衡的.

树是 α-平衡的，有一个显然的好处是，它的高度是 $O(\log n)$ 的. 这是因为，从叶子节点每向根移动一步，子树所包含的叶子节点数目就至少扩大到原来的 $1/(1-\alpha)$ 倍，因此只能移动 $O(\log_{\frac{1}{1-\alpha}} n) = O(\log n)$ 次. 这就保证了在 α-平衡的树中，单次查询的复杂度总是严格 $O(\log n)$ 的，且算法的常数与 $\log(1/(1-\alpha))$（以 2 为底）成反比. 当 α 位于下文提供的合理范围内时，这个常数大致为 $2 \sim 3.5$.

WBLT 的平衡维护通常可以通过旋转或合并的方式进行. 两种方式实现的 WBLT，单次插入、删除等操作，复杂度都是严格 $O(\log n)$ 的. 但是，与固定优先级的 Treap→10.15.2@p2221 不同，WBLT 的结构并不具有唯一性，因此，两种方式维护得到的树的结构并不相同，虽然这并不影响它们的使用. 当然，平衡的维护还可以采取类似 替罪羊树→10.15.5@p2301 的策略，利用重构达到均摊 $O(\log n)$ 的复杂度，但是这样就失去了 WBLT 可持久化和区间操作等优势，因而并不推荐.

下文分别介绍了通过旋转和合并维护平衡的方法，并实现了相应的平衡维护和合并操作的函数. 封装好这些函数后，两种维护树平衡的方式在后续具体的平衡树的实现中再无区别. 而且，无论使用哪种方式，单次维护平衡的操作的时间复杂度都是 $O(1)$ 的，单次合并树 T_1 和树 T_2 的复杂度都是 $O\left(\left|\log \frac{w(T_1)}{w(T_2)}\right|\right)$ 的.

省略权重的记号

为了维护树的平衡，只需要保留子树的权重信息. 因此，为了表达方便，下面讨论平衡维护的两节将混用树和它的权重的记号. 比如，子树 x 的权重也由 x 表示，而不是 $w(x)$. 类似地，子树 x 和 y 合并得到的树也用它的权重表示，直接写作树 $x+y$.

通过旋转维护

WBLT 的旋转操作和 Treap 的旋转操作→10.15.2@p2221 完全相同，可以采取与 Treap 完全一致的旋转策略. 当然，旋转本身同样可以看作是重新分配子树权重的过程，因此也可以利用拆分和连接子树完成. 两种实现的结果是完全一致的，但是第二种实现更方便 WBLT 的持久化.

参考代码

- 不依赖连接

```
// Rotate the subtree at x such that ch[x][r] is the new root.
void rotate(int& x, bool r) {
  int y = ch[x][r];
  ch[x][r] = ch[y][!r];
  ch[y][!r] = x;
  push_up(x);
  push_up(y);
  x = y;
}

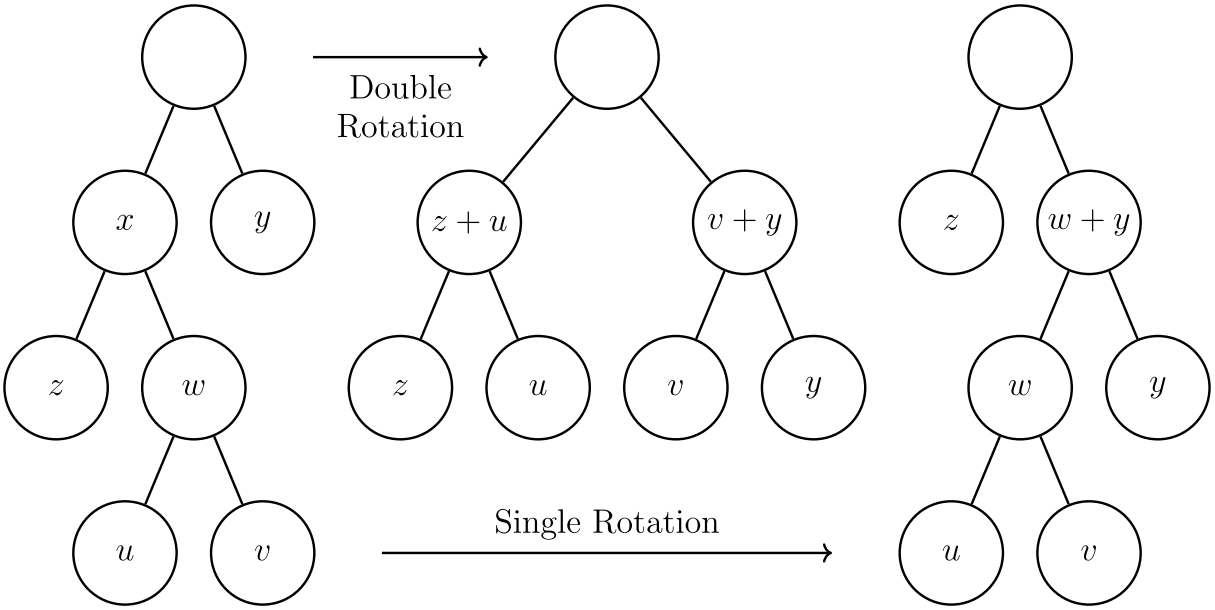
```

- 依赖连接

```
// Rotate the subtree at x such that ch[x][r] is the new root.
void rotate(int& x, bool r) {
  int a, b, c, d;
  std::tie(a, b) = cut(x);
  if (r) {
    std::tie(c, d) = cut(b);
    x = join(join(a, c), d);
  } else {
    std::tie(c, d) = cut(a);
    x = join(c, join(d, b));
  }
}

```

假设在某个树的修改操作后，正在自下而上地恢复树的平衡. 现在，左右子树 x 和 y 不再平衡，但是它们自身都是平衡的. 不妨设右子树 y 过轻，即 $y<\alpha(x+y)$. 此时，树的形态如图中左侧的树所示.



一种朴素的平衡维护策略是将 x 旋转到根节点处，这样它原先的右子节点 w 就和 y 一起成为了新树的右子节点，而它原先的左子节点 z 成为了新树的左子节点．这相当于将原来的树左侧中 w 的权重移动到它的右侧．如果 w 的权重合适，这样的操作就可以恢复树的平衡．这样得到的树如图中右侧的树所示．

但是，如果 w 本身过重，这样的操作可能移动了太多的权重到右子树，从而使得新树中左子树过轻，即 $z < \alpha(x+y)$．对于这种情形，因为子树 z 和子树 y 的权重都太小，只能考虑将 w 分拆为两个子树，分别与 z 和 y 连接，成为新树的两个子树．这相当于首先将节点 w 旋转到节点 x 处，再将它旋转到根节点处．同样，可以期待这样得到的树能够达到平衡，形态如图中间的树所示．

这两种旋转的策略分别称为单旋和双旋．单旋和双旋策略的选取，主要取决于子树 w 相对于子树 x 的比重，即存在阈值 β，使得

- 当 $w \le \beta x$ 时，应选取单旋策略；
- 当 $w > \beta x$ 时，应选取双旋策略．

难点在于阈值 β 的选择，这就需要做一些具体的计算．Blum 和 Mehlhorn 证明了，对于参数[8]

$$\alpha \in \left(\frac{2}{11}, 1-\frac{\sqrt{2}}{2}\right] \approx (0.182, 0.292],\ \beta = \frac{1}{2-\alpha},$$

能够通过上述单旋和双旋结合的策略，维护因为单次插入或删除而失衡的 WBLT 的平衡．

证明

需要证明的是，如果树在单次插入或删除后失衡，可以通过上述策略恢复它的平衡．结合上述图示，令

$$\rho_1 = \frac{y}{x+y},\ \rho_2 = \frac{w}{x},\ \rho_3 = \frac{v}{w}.$$

那么，有 $\rho_1 < \alpha \le \rho_2, \rho_3 \le 1-\alpha$．此处还有一个隐含条件，是关于 ρ_1 的取值范围的：

- 如果失衡是由插入单个元素引起的，那么，应该有
  $$\frac{y}{x-1+y} \ge \alpha \Longrightarrow \rho_1 \ge \frac{\alpha y}{y+\alpha} \ge \frac{\alpha}{1+\alpha}.$$
- 如果失衡是由删除单个元素引起的，那么，应该有

[8] Nievergelt 和 Reingold 的原始论文中给出的参数范围 $\alpha < 1-\frac{\sqrt{2}}{2},\ \beta = \frac{1-2\alpha}{1-\alpha}$ 是错误的．Hirai 和 Yamamoto 的文章中提供了相应的反例，问题主要出现在某些很小的树上，从而导致整个归纳证明失效．当然，实际算法竞赛中，很难造出能卡掉这些错误参数的数据，所以实践中可能并不会有太大影响．

$$\frac{y+1}{x+y+1} \geq \alpha \Longrightarrow \rho_1 \geq \frac{\alpha y}{y+1-\alpha} \geq \frac{\alpha}{2-\alpha}.$$

因为对于 $0 < \alpha < 1/2$，总有 $\alpha/(2-\alpha) < \alpha/(1+\alpha)$，所以删除元素会导致比增添元素更严重的失衡，尤其是对于树的规模很小的情形.

接下来，恢复平衡的操作分为两种情形：

情形一：w 没有过重，即 $\rho_2 \leq \beta$ 时，单旋

首先，z 和 $w+y$ 平衡．这是因为

$$\left(1-\frac{\alpha}{2-\alpha}\right)\alpha + \frac{\alpha}{2-\alpha} \leq \frac{w+y}{x+y} = (1-\rho_1)\rho_2 + \rho_1 < (1-\alpha)\frac{1}{2-\alpha} + \alpha.$$

左侧表达式在 $\alpha \in (0,1)$ 时总大于 α，右侧表达式在 $\alpha \in (0, 1-\sqrt{2}/2]$ 时总不大于 $(1-\alpha)$.

其次，w 和 y 平衡．同样地，考虑

$$\frac{y}{w+y} = \frac{\rho_1}{(1-\rho_1)\rho_2 + \rho_1}.$$

一方面，对于所有 $\alpha \in (0, (3-\sqrt{5})/2)$，有

$$\frac{y}{w+y} < \frac{\alpha}{(1-\alpha)\alpha + \alpha} < 1-\alpha.$$

另一方面，对于所有 $\alpha \in (0, 1/3)$，除了删除元素且 $y=1$ 的情形外，都有

$$\rho_1 \geq \min\left\{\frac{\alpha}{1+\alpha}, \frac{2\alpha}{3-\alpha}\right\} = \frac{2\alpha}{3-\alpha},$$

所以，有

$$\frac{y}{w+y} \geq \frac{\dfrac{2\alpha}{3-\alpha}}{\left(1-\dfrac{2\alpha}{3-\alpha}\right)\dfrac{1}{2-\alpha} + \dfrac{2\alpha}{3-\alpha}} > \alpha.$$

最后，考虑剩余的情形，即删除元素且 $y=1$ 时．最可能失衡的情形发生在 $x = \lfloor 2/\alpha \rfloor - 2$ 且 $w = \lfloor \beta x \rfloor$ 时．树可以恢复平衡，当且仅当

$$\frac{1}{1+\lfloor \beta x \rfloor} \geq \alpha \Longleftrightarrow \lfloor \beta x \rfloor \leq \frac{1}{\alpha} - 1 \Longleftrightarrow \beta x < \frac{1}{\alpha} \Longleftrightarrow x < \frac{2}{\alpha} - 1.$$

而这总是成立的. 这就完成了该情形的证明. 注意, 最后一种情形的证明利用了权重总是整数这一性质，并不能合并到之前的讨论中.

情形二：w 过重，即 $\rho_2 > \beta$ 时，双旋

首先，$z+u$ 和 $v+y$ 平衡．这是因为

$$\frac{\alpha}{2-\alpha} + \left(1-\frac{\alpha}{2-\alpha}\right)\frac{1}{2-\alpha}\alpha < \frac{v+y}{x+y} = \rho_1 + (1-\rho_1)\rho_2\rho_3 < \alpha + (1-\alpha)^3$$

左侧表达式在 $\alpha \in (0,1)$ 时总大于 α，右侧表达式在 $\alpha \in (0, (3-\sqrt{5})/2)$ 时总小于 $(1-\alpha)$.

然后，z 和 u 平衡．这是因为对于 $\alpha \in (0,1)$，总是成立

$$\alpha = \frac{\dfrac{1}{2-\alpha}\alpha}{1-\dfrac{1}{2-\alpha}(1-\alpha)} < \frac{u}{z+u} = \frac{\rho_2(1-\rho_3)}{1-\rho_2\rho_3} < \frac{(1-\alpha)^2}{1-(1-\alpha)\alpha} < 1-\alpha.$$

最后，v 和 y 平衡．类似其他的情形，考虑

$$\frac{y}{v+y} = \frac{\rho_1}{\rho_1 + (1-\rho_1)\rho_2\rho_3}.$$

一方面，对于所有 $\alpha \in (0, 1-\sqrt{2}/2]$，都有

$$\frac{y}{v+y} < \frac{\alpha}{\alpha + (1-\alpha)\dfrac{1}{2-\alpha}\alpha} \le 1-\alpha.$$

另一方面，

$$\frac{y}{v+y} \ge \frac{\rho_1}{\rho_1 + (1-\rho_1)(1-\alpha)^2}.$$

右侧表达式不小于 α，当且仅当

$$\rho_1 \ge \frac{\alpha(1-\alpha)}{1+\alpha(1-\alpha)}.$$

如果失衡是由插入引起的，那么 $\rho_1 \ge \alpha/(1+\alpha)$，显然成立．否则，情形有些复杂：

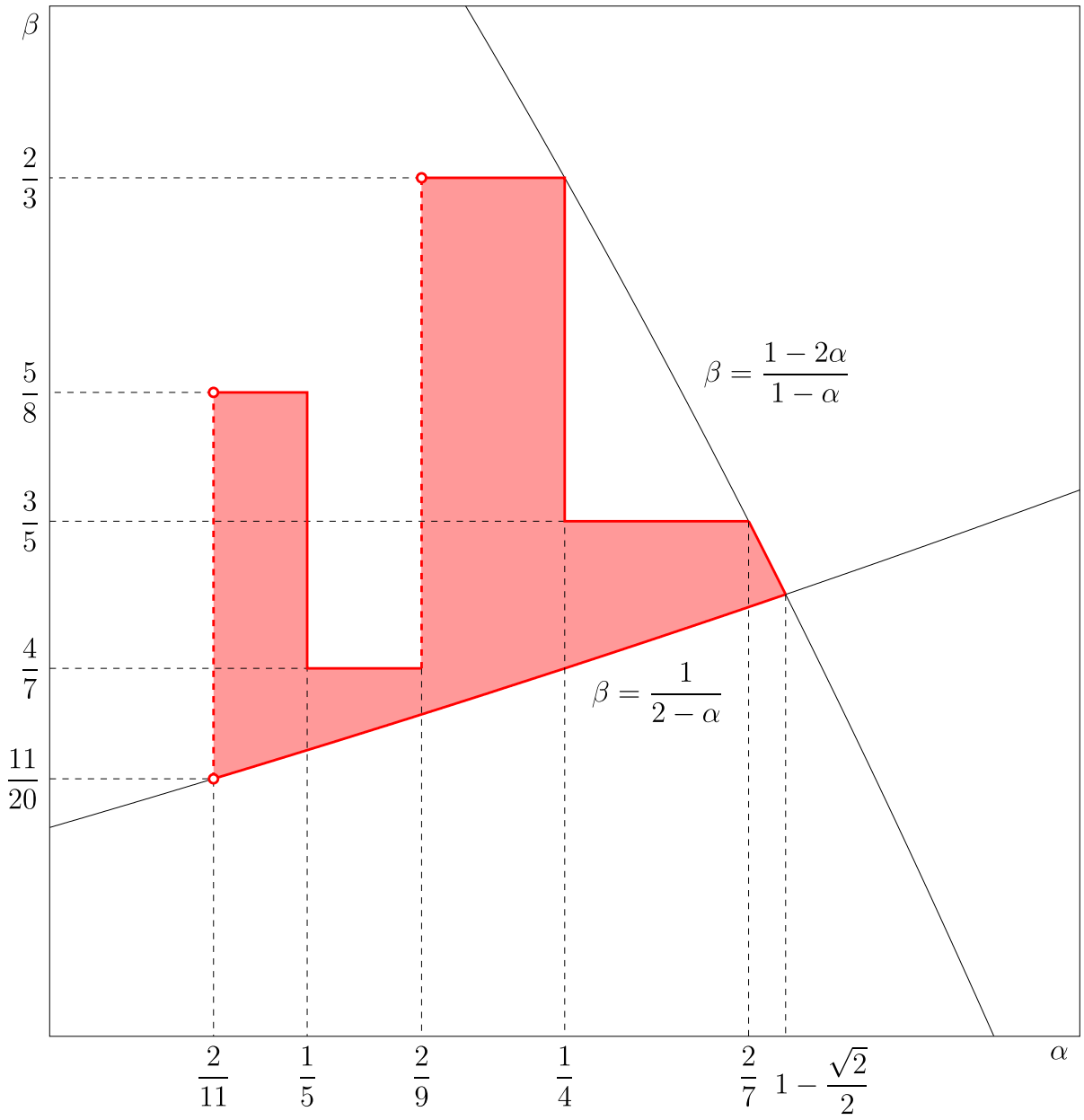
- 当 $y \ge 3$ 时，$\rho_1 \ge 3\alpha/(4-\alpha)$，且 $3\alpha/(4-\alpha) \ge \alpha(1-\alpha)/(1+\alpha(1-\alpha))$ 对于所有 $\alpha \in [1-\sqrt{3}/2, 1)$ 都成立；
- 当 $y = 2$ 时，最可能失衡的情形发生在 $x = \lfloor 3/\alpha \rfloor - 3$，$w = \lfloor (1-\alpha)x \rfloor$ 且 $v = \lfloor (1-\alpha)w \rfloor$ 时，此时 $y/(v+y) \ge \alpha$ 对于所有 $\alpha \in (3/22, 1)$ 都成立；
- 当 $y = 1$ 时，最可能失衡的情形发生在 $x = \lfloor 2/\alpha \rfloor - 2$，$w = \lfloor (1-\alpha)x \rfloor$ 且 $v = \lfloor (1-\alpha)w \rfloor$ 时，此时 $y/(v+y) \ge \alpha$ 对于所有 $\alpha \in (2/11, 1)$ 都成立．

最后两种情形的讨论，同样利用了所有节点的权重都是整数这一点．

综合两种情形，当 $\alpha \in (2/11, 1-\sqrt{2}/2]$ 时，前述单旋和双旋结合的策略可以保证树的平衡．

从这个分析过程中可以看出，最难保持平衡的情形发生在从小规模的树中删除节点时．除了 $\beta = 1/(2-\alpha)$ 之外，对于其他参数的选择的正确性证明，同样可以重复上述的过程，只是用到的一些不等式需要相应地调整．

随后，Hirai 和 Yamamoto 通过机器证明完整地确定了所有可行的 (α, β) 的范围，结果是一个相当复杂的二维图形：



他们在文章中推荐使用如下策略维持平衡：

- 当 $x > 3y$ 时，判断失衡；
- 当 $w \leq 2z$ 时，选取单旋策略，否则，选取双旋策略.

原因是，这是可行的参数范围内唯一可以用简单整数表示的策略，从而避免了浮点数运算造成的效率损失. 他们推荐的策略相当于取 $(\alpha, \beta) = (1/4, 2/3)$. 实践中，可以根据具体情况，选择合适的参数.

参考实现如下：

参考代码

```
// Check whether a subtree of weight SX is too heavy
//     in a tree of weight SX + SY.
bool too_heavy(int sx, int sy) {
  // or sx > sy * 3;
  return sy < ALPHA * (sx + sy);
}
```

```
7
8  // Check if ch[x][!r] is too heavy so that a double rotation is needed.
9  bool need_double_rotation(int x, bool r) {
10   // or sz[ch[x][!r]] > sz[ch[x][r]] * 2;
11   return sz[ch[x][!r]] > sz[x] / (2 - ALPHA);
12 }
13
14 // Balance the subtree at x;
15 void balance(int& x) {
16   if (sz[x] == 1) return;
17   bool r = sz[ch[x][1]] > sz[ch[x][0]];
18   if (!too_heavy(sz[ch[x][r]], sz[ch[x][!r]])) return;
19   if (need_double_rotation(ch[x][r], r)) {
20     rotate(ch[x][r], !r);
21   }
22   rotate(x, r);
23 }
24
```

实现了维护平衡的策略后，合并两树的算法就非常简单．仍然设 $x > y$，合并的策略如下：

- 如果右子树 y 是空的，直接返回左子树 x；
- 如果左右子树 x 和 y 已经平衡，即 $y \geq \alpha(x + y)$，直接连接两子树；
- 否则，将 x 的右子树 w 与 y 合并，将左子树 z 与它们合并的结果连接，并调整新树的平衡．

参考实现如下：

参考代码

```
1  // Merge two subtrees.
2  int merge(int x, int y) {
3    if (!x || !y) return x | y;
4    int a, b;
5    if (too_heavy(sz[x], sz[y])) {
6      std::tie(a, b) = cut(x);
7      int z = join(a, merge(b, y));
8      balance(z);
9      return z;
10   } else if (too_heavy(sz[y], sz[x])) {
11     std::tie(a, b) = cut(y);
12     int z = join(merge(x, a), b);
13     balance(z);
14     return z;
15   } else {
16     return join(x, y);
17   }
18 }
19
```

可以证明，这样可以维持合并后树的平衡，且这样操作的复杂度是 $O(|\log(x/y)|)$ 的．

平衡和复杂度的证明

只需要考虑 y 过轻的情形，即 $y < \alpha(x+y)$. 此时，先合并 w 和 y，再连接 z 和 $w+y$. 需要证明的是，只要在树根处调整树的平衡，就能够保证树的平衡. 假设树 $w+y$ 的左、右子树分别是 c 和 d，且 c 的左、右子树分别是 a 和 b. 在树根处调整平衡，可以分为三种情形：

情形一：z 和 $w+y$ 已经平衡，无需进一步调整，即 $z \geq \alpha(x+y)$

根据平衡的定义，子树 z 和 $w+y$ 都是平衡的，且它们互相也是平衡的，那么整棵树也是平衡的.

情形二：z 过轻且 c 没有过重，可以通过单旋恢复平衡，即 $z < \alpha(x+y)$ 且 $c \leq \beta(w+y)$

此时，因为 z 和 w 平衡，但 y 相较于 $x = z+w$ 过轻，所以，子树 z 的权重满足

$$\alpha(1-\alpha)(x+y) < \alpha(z+w) \leq z \leq \alpha(x+y).$$

而 c 的权重则满足

$$\alpha(w+y) \leq c \leq \beta(w+y).$$

由此，z 和 c 互相平衡，只要

$$\frac{\alpha}{1-\alpha} < \frac{1-\alpha}{\alpha}\alpha < \frac{c}{z} = \frac{w+y}{z}\frac{c}{w+y} < \frac{1-\alpha(1-\alpha)}{\alpha(1-\alpha)}\beta \leq \frac{1-\alpha}{\alpha},$$

这要求

$$\beta \leq \frac{(1-\alpha)^2}{1-\alpha(1-\alpha)}.$$

以及 $z+c$ 和 d 互相平衡，只要

$$\alpha \leq (1-\beta)(1-\alpha) \leq \frac{d}{w+y}\frac{w+y}{x+y} = \frac{d}{x+y} < \frac{d}{c+d} \leq 1-\alpha,$$

这要求

$$\beta \leq \frac{1-2\alpha}{1-\alpha}.$$

情形三：z 过轻且 c 过重，可以通过双旋恢复平衡，即 $z < \alpha(x+y)$ 且 $c > \beta(w+y)$

类似情形二，有

$$\alpha(1-\alpha)(x+y) < z \leq \alpha(x+y),$$
$$\beta(w+y) < c \leq (1-\alpha)(w+y),$$
$$\alpha c \leq a, b \leq (1-\alpha)c.$$

由此，z 和 a 互相平衡，只要

$$\frac{\alpha}{1-\alpha} \leq \frac{1-\alpha}{\alpha}\beta\alpha \leq \frac{a}{z} = \frac{w+y}{z}\frac{a}{c+d} < \frac{1-\alpha(1-\alpha)}{\alpha(1-\alpha)}(1-\alpha)^2,$$

这要求

$$\beta \geq \frac{\alpha}{(1-\alpha)^2}.$$

其次，b 和 d 互相平衡，只要

$$\frac{\alpha}{1-\alpha} \leq \frac{\beta}{1-\beta}\alpha \leq \frac{b}{d} = \frac{c}{d}\frac{b}{c} \leq \frac{1-\alpha}{\alpha}(1-\alpha) < \frac{1-\alpha}{\alpha},$$

这要求

$$\beta \geq \frac{1}{2-\alpha}.$$

最后，$z+a$ 和 $b+d$ 互相平衡，只要

$$\alpha < (1-\alpha)(1-(1-\alpha)^2) \leq \frac{b+d}{x+y} = \frac{w+y}{x+y}\frac{b+d}{w+y} < (1-\alpha(1-\alpha))(1-\beta\alpha) \leq 1-\alpha,$$

这要求

$$\beta \geq \frac{\alpha}{1-\alpha+\alpha^2}.$$

综合三种情形，只要

$$0 < \alpha \leq 1-\frac{\sqrt{2}}{2},\ \frac{1}{2-\alpha} \leq \beta \leq \frac{1-2\alpha}{1-\alpha},$$

就能保证合并后的树可以利用单旋和双旋结合的策略调整到平衡．这显然包含正文给出的参数范围．

最后，简单说明一下该算法的复杂度为什么是 $O(|\log(x/y)|)$ 的．合并的流程中，如果 y 相较于 x 过轻，就尝试与 x 的右子树合并，这个过程一直持续到以 x 某个子孙节点为根的子树与 y 平衡为止．因为每向下加深一层，子树权重就减少到不超过原来的 $(1-\alpha)$ 倍，所以至多只要 $\log_{\frac{1}{1-\alpha}}(x/y)$ 次迭代，就能找到与 y 平衡的子树．因此，该合并算法调用 $O(|\log(x/y)|)$ 次平衡算法[9]，复杂度也就是 $O(|\log(x/y)|)$．

虽然并不明显，但是这个论证过程依赖于这样一个结论：不断取右子树的过程中，y 不会在一次迭代前后，从相较于左侧的子树过轻，变为相较于它过重．这是因为能够与 y 平衡的子树权重范围位于 $\alpha y/(1-\alpha)$ 与 $(1-\alpha)y/\alpha$ 之间．因此，如果在一次迭代时，就从 y 过轻变成 y 过重，则 x 的子树的权重在该次迭代过程中至少缩小到了原来的 $\alpha^2/(1-\alpha)^2$ 倍．但是，单次迭代，子树权重至多只能缩小到原来的 α 倍，而在上述 α 的范围中，$\alpha > \alpha^2/(1-\alpha)^2$．这说明，前设情形是不可能的，某次迭代之后一定会有 y 与 x 的某个子树平衡的情形发生．

通过合并维护

合并两子树是指，保证左子树的键值总是不大于右子树的键值的情况下，建立新树，使其所有叶子节点的信息恰为左右子树叶子节点信息的并，且保证树的平衡．

为此，有如下策略[10]：（仍然设 $x \geq y$）

- 如果右子树 y 是空的，直接返回左子树 x；
- 如果左右子树 x 和 y 已经平衡，即 $y \geq \alpha(x+y)$，直接连接两子树；
- 否则，右子树 y 过轻，但如果 x 的左子树 z 和 $w+y$ 可以平衡，即 $z \geq \alpha(x+y)$，就将 w 和 y 先合并，再合并 z 和 $w+y$；
- 否则，z 和 y 都过轻，此时，需要首先合并 z 和 w 的左子树 u，再合并 w 的右子树 v 和 y，再将两次合并的结果 **合并** 为新树．

将这一策略与前文的平衡策略对比，可以看到后两种情形中节点的组合方式分别和前述平衡策略中单旋和双旋的结果相似，只是将子树的连接换作了合并．

可以证明，当

[9]因为单次平衡操作至多相当于连接两次子树，而且最后两子树已经平衡时还需要调用一次连接子树的算法，所以如果以调用连接子树的算法的次数计算，基于平衡实现的合并操作和下文直接合并的平衡操作的算法的常数是一致的．

[10]通过稍后的证明可以看出：第三种情形中，z 和 $w+y$ 总是平衡的；第四种情形中，z 和 u 总是平衡的．它们都可以直接连接，而不需要合并．

$$0 < \alpha \le 1 - \frac{\sqrt{2}}{2} \approx 0.292$$

时，这样得到的树总是平衡的，且这样操作的复杂度是 $O(|\log(x/y)|)$ 的．也就是说，合并两个树的成本，与两个树的绝对大小无关，而只与它们的相对大小有关．

平衡和复杂度的证明

设合并权重分别为 x 和 y 的两棵子树时，需要直接连接两棵子树的次数为 $\tau(x,y)$．严格来说，需要证明当 $0 < \alpha \le 1 - \sqrt{2}/2$ 时，存在常数 $C > 0$，对于任意 $x \ge y > 0$，都有

$$\tau(x,y) \le 1 + C \log^+ \frac{\alpha x}{(1-\alpha)^2 y},$$

其中，$\log^+ x = \max\{0, \log x\}$；而且，对于所有 $x/y \le (1-\alpha)/\alpha$，都有 $\tau(x,y) = 1$．实际上，式子中的常数可以取作

$$C = -\frac{2}{\log(1-\alpha)}.$$

这就说明了合并算法的复杂度是 $O(|\log(x/y)|)$ 的．

为了证明合并算法得到的树总是平衡的，且上述复杂度的表达式成立，需要使用归纳法．对于所有第一象限的格点 $(x,y) \in \mathbf{N}_+^2$，可以赋以 $(x+y, |\ x-y\ |)$ 的字典序，这显然是该集合上的良序，可以沿着该顺序进行归纳．归纳起点是 $(x,y) = (1,1)$，此时，两子树都只有一个叶子节点，直接连接得到的子树必然是平衡的，且 $\tau(x,y) = 1$，符合上式．下面假设归纳进行到 (x,y)，且结论对于所有 (x,y) 之前的点都成立．这分为三种情形：

情形一：树 x 和 y 平衡，即 $y \ge \alpha(x+y)$

此时，直接连接得到树也是平衡的，且只调用树的连接算法一次，所以有 $\tau(x,y) = 1$．

情形二：树 y 过轻，但是 z 并不过轻，即 $y < \alpha(x+y) \le z$

此时，首先合并 w 和 y，然后合并 z 和 $w+y$，所以

$$\tau(x,y) = \tau(w,y) + \tau(z, w+y).$$

由归纳假设，子树 $w+y$ 已经是平衡的．对于第二步合并，其实可以直接证明 z 和 $w+y$ 是平衡的：

$$\alpha \le \frac{z}{z+(w+y)} = \frac{z}{x+y} < \frac{z}{z+w} \le 1-\alpha.$$

因此，合并 z 和 $w+y$ 其实是直接连接两个子树，有 $\tau(z, w+y) = 1$．因此，最后得到的树也是平衡的．

现在估计 $\tau(w,y)$ 的大小．因为 $y < (\alpha/(1-\alpha))x$ 且 $\alpha x \le w \le (1-\alpha)x$，所以，经放缩可知

$$\frac{\alpha}{1-\alpha} < 1 - \alpha = \frac{\alpha x}{(\alpha/(1-\alpha))x} < \frac{w}{y} \le \frac{(1-\alpha)x}{y}.$$

这说明，w 和 y 不平衡，只出现在 $w > y$ 时，所以，有

$$\tau(w,y) \le 1 + C \log^+ \frac{\alpha w}{(1-\alpha)^2 y} \le 1 + C \log^+ \frac{\alpha x}{(1-\alpha)y}$$

$$= 1 + C \log(1-\alpha) + C \log^+ \frac{\alpha x}{(1-\alpha)^2 y}.$$

最后一步的等式成立，是因为 $x/y > (1-\alpha)/\alpha$．

因此，有

$$\tau(x,y)\le 2+C\log(1-\alpha)+C\log^+\frac{\alpha x}{(1-\alpha)^2y}.$$

情形三：树 y 和 z 都过轻，即 $y,z<\alpha(x+y)$

此时，首先合并 z 和 u，再合并 v 和 y，最后合并 $z+u$ 和 $v+y$．因此，

$$\tau(x,y)=\tau(z,u)+\tau(v,y)+\tau(z+u,v+y).$$

类似前文情形，可以估计每一步合并时两个子树的权重比值．

因为 $z,y<\alpha(x+y)$，所以 $w>(1-2\alpha)(x+y)$．同时，利用平衡条件，有 $\alpha\le z/x,w/x,u/w,v/w\le 1-\alpha$．这说明

$$\frac{\alpha}{1-\alpha}<\frac{\alpha}{1-\alpha}\frac{1}{1-\alpha}\le\frac{z}{u}=\frac{z}{w}\frac{w}{u}<\frac{\alpha}{1-2\alpha}\frac{1}{\alpha}\le\frac{1-\alpha}{\alpha},$$

$$\frac{\alpha}{1-\alpha}\le\alpha\frac{1-2\alpha}{\alpha}<\frac{v}{y}=\frac{v}{w}\frac{w}{y}\le(1-\alpha)\frac{(1-\alpha)x}{y}=(1-\alpha)^2\frac{x}{y}.$$

对于最后一项，有

$$\frac{z+u}{v+y}=\frac{x+y}{v+y}-1=\frac{x+y}{y}\frac{y}{v+y}-1<(1-\alpha)\left(\frac{x}{y}+1\right)-1<(1-\alpha)\frac{x}{y}.$$

反过来，有

$$\frac{z+u}{v+y}=\frac{x+y}{v+y}-1\ge\frac{x+y}{(1-\alpha)^2x+y}-1>\frac{1}{(1-\alpha)^3+\alpha}-1>\frac{\alpha}{1-\alpha}.$$

利用这些不等式，可以说明最后得到的树必然是平衡的．利用归纳假设可知，z 和 u 合并，v 和 y 合并，都可以保证得到的树是平衡的．而且，其中第一步 z 和 u 合并实际上是直接连接两棵树．对于树 $z+u$ 和树 $v+y$ 的合并，又有两种子情形：

- 如果 $z+u\le v+y$，那么它们的权重比值严格大于 $\alpha/(1-\alpha)$，故而可以直接连接，结果是平衡的；
- 否则，它们的权重比值必然严格小于 x/y，但是 $(z+u)+(v+y)=x+y$，所以 $|(z+u)-(v+y)|<|x-y|$，由前文给出的字典序判断，这种情形也可以应用归纳假设，结果也是平衡的．

进一步应用归纳假设可知：

$$\tau(z,u)=1,$$

$$\tau(v,y)\le 1+C\log^+\frac{\alpha x}{y},$$

$$\tau(z+u,v+y)\le 1+C\log^+\frac{\alpha x}{(1-\alpha)y}.$$

三个不等式直接相加，会导致对数项前面的系数变成 $2C$，无法完成归纳．因此，此处需要更为细致的估计．

当 $\min\{v/y,(z+u)/(v+y)\}\le(1-\alpha)/\alpha$ 时，$\tau(v,y)$ 和 $\tau(z+u,v+y)$ 中必然有一项为 1，所以，有

$$\begin{aligned}\tau(v,y)+\tau(z+u,v+y)&\le 2+C\log^+\frac{\alpha x}{(1-\alpha)y}\\&=2+C\log(1-\alpha)+C\log^+\frac{\alpha x}{(1-\alpha)^2y}.\end{aligned}$$

否则，应该有

$$\tau(v,y)+\tau(z+u,v+y)\le 2+C\log^+\frac{\alpha v}{(1-\alpha)^2y}+C\log^+\frac{\alpha(z+u)}{(1-\alpha)^2(v+y)}$$
$$=2+C\log\frac{\alpha}{(1-\alpha)^2}+C\log^+\frac{\alpha v(z+u)}{(1-\alpha)^2y(v+y)}.$$

对于 $0<\alpha\le 1-\sqrt{2}/2$，有

$$\frac{\alpha}{(1-\alpha)^2}<1-\alpha.$$

而且，有

$$\frac{v(z+u)}{y(v+y)}=\left(\frac{v+y}{y}-1\right)\left(\frac{x+y}{y}\frac{y}{v+y}-1\right)$$
$$=\frac{x+y}{y}+1-\frac{x+y}{y}\frac{y}{v+y}-\frac{v+y}{y}<\frac{x}{y}.$$

这就说明，对于后面这种情形，也有

$$\tau(v,y)+\tau(z+u,v+y)<2+C\log(1-\alpha)+C\log^+\frac{\alpha x}{(1-\alpha)^2y}.$$

整体的合并复杂度为

$$\tau(x,y)\le 3+C\log(1-\alpha)+C\log^+\frac{\alpha x}{(1-\alpha)^2y}.$$

综合所有情形，有

$$\tau(x,y)\le 3+C\log(1-\alpha)+C\log^+\frac{\alpha x}{(1-\alpha)^2y}.$$

因此，只要取 $2+C\log(1-\alpha)\le 0$，就可以完成复杂度的归纳．一个显然的选择为

$$C=-\frac{2}{\log(1-\alpha)}.$$

这个常数说明，两个树合并时，直接连接子树的次数大致不会超过树高差值的二倍．

参考实现如下：

参考代码

```cpp
// Merge two subtrees.
int merge(int x, int y) {
  if (!x || !y) return x | y;
  int a, b, c, d;
  if (too_heavy(sz[x], sz[y])) {
    std::tie(a, b) = cut(x);
    if (too_heavy(sz[b] + sz[y], sz[a])) {
      std::tie(c, d) = cut(b);
      return merge(merge(a, c), merge(d, y));
    } else {
      return merge(a, merge(b, y));
    }
  } else if (too_heavy(sz[y], sz[x])) {
    std::tie(a, b) = cut(y);
    if (too_heavy(sz[a] + sz[x], sz[b])) {
      std::tie(c, d) = cut(a);
      return merge(merge(x, c), merge(d, b));
```

```
    } else {
      return merge(merge(x, a), b);
    }
  } else {
    return join(x, y);
  }
}

```

利用该合并策略，同样可以很容易实现树的平衡维护：失衡时，直接合并左右子树即可.

参考代码

```
// Balance the subtree at x;
void balance(int& x) {
  if (sz[x] == 1) return;
  if (too_heavy(sz[ch[x][0]], sz[ch[x][1]]) ||
      too_heavy(sz[ch[x][1]], sz[ch[x][0]])) {
    int a, b;
    std::tie(a, b) = cut(x);
    x = merge(a, b);
  }
}

```

因为需要再平衡的两个树的大小总是近乎平衡的，因此维护平衡的复杂度是 $O(1)$ 的.

平衡树基础操作

利用前文实现的函数，WBLT 可以支持平衡树的所有基础操作. 本节以可重集为例，讨论 WBLT 实现平衡树的方法.

建树

建树操作与线段树十分相似，只需要向下递归二分区间，直至区间长度为 1 时把要维护的信息放在叶子节点上，回溯的时候合并区间信息即可.

参考代码

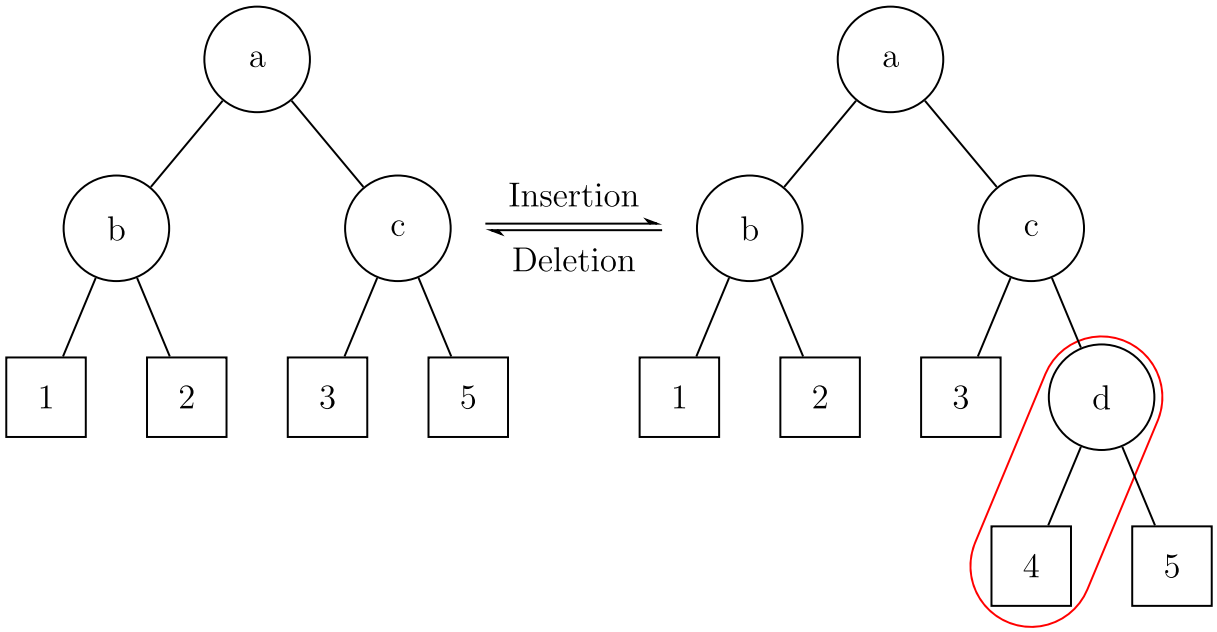
```
// Build the tree for the interval [ll, rr].
int build(int ll, int rr) {
  if (ll == rr) return new_leaf(ll);
  int mm = (ll + rr) / 2;
  return join(build(ll, mm), build(mm + 1, rr));
}

```

时间复杂度为 $O(n)$.

插入与删除

对于插入操作，需要从根节点开始向下递归，直到找到权值大于等于待插入元素的叶子节点中权值最小的那个，再新建两个节点，其中一个用来存储新插入的值，另一个作为两个叶子的新父亲替代这个最小叶子节点的位置，再将这两个叶子连接到这个父亲上. 回溯时，需要维护树的平衡.



如图所示，要向左侧的树中插入值为 4 的元素．首先找到值为 5 的叶子节点，然后新建叶子节点 4 和非叶子节点 d，并将 4 和 5 连接到 d 上．这就得到右侧的树．

对于删除，考虑上面过程的逆过程．即找到与要删除的值权值相等的一个叶子节点，将它和它的父亲节点删除，并用其父亲的另一个儿子代替父亲的位置．回溯时，同样需要维护树的平衡．

参考实现如下：

参考代码

```
// Insert v to the subtree at x.
void insert(int& x, int v) {
  if (!x) {
    x = new_leaf(v);
  } else if (sz[x] == 1) {
    bool r = v >= val[x];
    ch[x][r] = new_leaf(v);
    ch[x][!r] = new_leaf(val[x]);
    push_up(x);
  } else {
    bool r = v > val[ch[x][0]];
    insert(ch[x][r], v);
    push_up(x);
    balance(x);
  }
}

// Insert v.
void insert(int v) { insert(rt, v); }

// Remove v from the subtree at x.
bool remove(int& x, int v) {
  if (!x) return false;
  if (sz[x] == 1) {
    if (val[x] == v) {
```

```
      del_node(x);
      return true;
    } else {
      return false;
    }
  } else {
    bool r = v > val[ch[x][0]];
    bool succ = remove(ch[x][r], v);
    if (!ch[x][r]) {
      x = ch[x][!r];
    } else {
      push_up(x);
      balance(x);
    }
    return succ;
  }
}

// Remove v.
bool remove(int v) { return remove(rt, v); }

```

注意空树的处理．如果不想处理空树，可以提前向树内插入 ∞ 元素．

两种操作的时间复杂度均为 $O(\log n)$．

查询排名

因为 WBLT 的形态和线段树十分相似，因此查询排名可以使用类似线段树上二分的方式：如果左子树的最大值大于等于待查值就往左子节点跳；否则，就向右子节点跳，同时答案加上左子树的权重．

参考实现如下：

参考代码

```
// Count the number of nodes less than v in the subtree at x.
int count_less_than(int x, int v) {
  if (!x) return 0;
  int res = 0;
  while (sz[x] > 1) {
    if (val[ch[x][0]] < v) {
      res += sz[ch[x][0]];
      x = ch[x][1];
    } else {
      x = ch[x][0];
    }
  }
  return res + (val[x] < v ? 1 : 0);
}

// Find the rank of v.
int find_rank(int v) { return count_less_than(rt, v) + 1; }

```

时间复杂度为 $O(\log n)$.

根据排名查询

依然是利用线段树上二分的思想，只不过这里比较的是节点的权重.

参考实现如下:

参考代码

```
// Find the k-th element in the subtree at x.
// It is guaranteed that such an element exists.
int find_kth(int x, int k) {
  while (sz[x] > 1) {
    if (sz[ch[x][0]] >= k) {
      x = ch[x][0];
    } else {
      k -= sz[ch[x][0]];
      x = ch[x][1];
    }
  }
  return val[x];
}

// Find the k-th element.
int find_kth(int k) { return k > sz[rt] || k <= 0 ? -1 : find_kth(rt, k); }

```

时间复杂度为 $O(\log n)$.

查找前驱、后继

以上两种功能结合即可.

参考实现如下:

参考代码

```
// Find the predecessor of v.
int find_prev(int v) { return find_kth(find_rank(v) - 1); }

// Find the successor of v.
int find_next(int v) { return find_kth(find_rank(v + 1)); }

```

如果想直接实现，需要注意键值相同的节点可能存储于多个叶子节点.

分裂操作

WBLT 的分裂与 无旋 Treap→10.15.2@p2221 类似，根据子树大小或权值决定向下递归分裂左子树或右子树. 不同的是，WBLT 需要对分裂出来的子树进行 **合并**，以维护最终分裂的树的平衡.

根据子树大小分裂的参考实现如下:

参考代码

```

```

```cpp
// Split the subtree at x.
// The left half will have k elements.
std::pair<int, int> split(int x, int k) {
  if (!x) return {0, 0};
  if (!k) return {0, x};
  if (k == sz[x]) return {x, 0};
  int a, b;
  std::tie(a, b) = cut(x);
  if (k <= sz[a]) {
    int ll, rr;
    std::tie(ll, rr) = split(a, k);
    return {ll, merge(rr, b)};
  } else {
    int ll, rr;
    std::tie(ll, rr) = split(b, k - sz[a]);
    return {merge(a, ll), rr};
  }
}

```

时间复杂度为 $O(\log n)$.

复杂度证明

向下递归的层数显然不超过树高，是 $O(\log n)$ 的. 需要证明的是，将左右两侧分裂出来的子树分别合并起来的复杂度是 $O(\log n)$ 的. 不妨仅考虑左侧的子树，因为右侧是对称的. 设左侧分裂出来的子树自下而上分别是 $T_1, T_2, \cdots, T_\ell$，这些子树的数量 $\ell \in O(\log n)$. 合并的过程可以描述为，自 $T'_1 = T_1$ 开始，将 T'_{i-1} 与 T_i 合并得到 T'_i，递归地合并完所有子树为止. 合并的总复杂度可以表示为

$$\sum_{i=2}^{\ell} \tau(T_i, T'_{i-1}),$$

其中，$\tau(T_i, T'_{i-1})$ 是将 T_i 和 T'_{i-1} 合并起来的复杂度.

如果总是有 $w(T_i) \geq w(T'_{i-1})$，那么根据合并两子树的复杂度表达式，有

$$\tau(T_i, T'_{i-1}) \in O\left(\log \frac{w(T_i)}{w(T'_{i-1})}\right) \subseteq O\left(\log \frac{w(T'_i)}{w(T'_{i-1})}\right).$$

因为这些大 O 记号中的常数都是一致的，所以可以直接相加，裂项相消.

但是，应该注意的是，$w(T_i) \geq w(T'_{i-1})$ 并不总是成立，因为 T'_{i-1} 是从 T_i 在原来的树中对应的右子树分裂出来的，而这个右子树可能比左子树 T_i 更大. 尽管如此，即使 T'_{i-1} 比 T_i 大，作为右子树的一部分，权重 $w(T'_{i-1})$ 也不会超过 $w(T_i)$ 的 $(1-\alpha)/\alpha$ 倍，这意味着，此时 T'_{i-1} 和 T_i 一定是平衡的，合并的复杂度是 $O(1)$ 的.

将这两种情形总结在一起，单次合并的复杂度可以写为

$$\tau(T_i, T'_{i-1}) \in O\left(\log \frac{w(T'_i)}{w(T'_{i-1})}\right) + O(1).$$

由此，合并的总复杂度为

$$O\left(\sum_{i=2}^{\ell}\left(1 + \log \frac{w(T'_i)}{w(T'_{i-1})}\right)\right) \subseteq O(\ell + \log w(T'_\ell)) \subseteq O(\log n).$$

这也说明，分裂算法的总复杂度是 $O(\log n)$ 的.

参考实现

本文介绍了如何利用 WBLT 完成平衡树的基本操作. 下面是用 WBLT 实现的 普通平衡树模板[1].

参考代码

```cpp
#include <iostream>
#include <tuple>

// Change here to use different strategies to balance and to merge.
#define BALANCE_BY_ROTATING 1
#define ROTATE_BY_JOINING 0

constexpr int N = 2e6;
constexpr double ALPHA = 0.292;

int id, rt, ch[N][2], sz[N], val[N];
int pool[N], top;

void push_up(int x) {
  sz[x] = sz[ch[x][0]] + sz[ch[x][1]];
  val[x] = val[ch[x][1]];
}

// Return a new empty node.
int new_node() {
  int x = top ? pool[--top] : ++id;
  sz[x] = val[x] = ch[x][0] = ch[x][1] = 0;
  return x;
}

// Release a node for later use.
void del_node(int& x) {
  pool[top++] = x;
  x = 0;
}

// Return a new leaf node of value v.
int new_leaf(int v) {
  int x = new_node();
  val[x] = v;
  sz[x] = 1;
  return x;
}

// Return a new node with subtrees x and y.
int join(int x, int y) {
  int z = new_node();
  ch[z][0] = x;
  ch[z][1] = y;
  push_up(z);
  return z;
}

```

```cpp
// Return subtrees of x and release x.
auto cut(int& x) {
  int y = ch[x][0];
  int z = ch[x][1];
  del_node(x);
  return std::make_pair(y, z);
}

// Check whether a subtree of weight SX is too heavy
//     in a tree of weight SX + SY.
bool too_heavy(int sx, int sy) {
  // or sx > sy * 3;
  return sy < ALPHA * (sx + sy);
}

#if BALANCE_BY_ROTATING

#if ROTATE_BY_JOINING

// Rotate the subtree at x such that ch[x][r] is the new root.
void rotate(int& x, bool r) {
  int a, b, c, d;
  std::tie(a, b) = cut(x);
  if (r) {
    std::tie(c, d) = cut(b);
    x = join(join(a, c), d);
  } else {
    std::tie(c, d) = cut(a);
    x = join(c, join(d, b));
  }
}

#else

// Rotate the subtree at x such that ch[x][r] is the new root.
void rotate(int& x, bool r) {
  int y = ch[x][r];
  ch[x][r] = ch[y][!r];
  ch[y][!r] = x;
  push_up(x);
  push_up(y);
  x = y;
}

#endif

// Check if ch[x][!r] is too heavy so that a double rotation is needed.
bool need_double_rotation(int x, bool r) {
  // or sz[ch[x][!r]] > sz[ch[x][r]] * 2;
  return sz[ch[x][!r]] > sz[x] / (2 - ALPHA);
}

```

```cpp
// Balance the subtree at x;
void balance(int& x) {
  if (sz[x] == 1) return;
  bool r = sz[ch[x][1]] > sz[ch[x][0]];
  if (!too_heavy(sz[ch[x][r]], sz[ch[x][!r]])) return;
  if (need_double_rotation(ch[x][r], r)) {
    rotate(ch[x][r], !r);
  }
  rotate(x, r);
}

// Merge two subtrees.
int merge(int x, int y) {
  if (!x || !y) return x | y;
  int a, b;
  if (too_heavy(sz[x], sz[y])) {
    std::tie(a, b) = cut(x);
    int z = join(a, merge(b, y));
    balance(z);
    return z;
  } else if (too_heavy(sz[y], sz[x])) {
    std::tie(a, b) = cut(y);
    int z = join(merge(x, a), b);
    balance(z);
    return z;
  } else {
    return join(x, y);
  }
}

#else

// Merge two subtrees.
int merge(int x, int y) {
  if (!x || !y) return x | y;
  int a, b, c, d;
  if (too_heavy(sz[x], sz[y])) {
    std::tie(a, b) = cut(x);
    if (too_heavy(sz[b] + sz[y], sz[a])) {
      std::tie(c, d) = cut(b);
      return merge(merge(a, c), merge(d, y));
    } else {
      return merge(a, merge(b, y));
    }
  } else if (too_heavy(sz[y], sz[x])) {
    std::tie(a, b) = cut(y);
    if (too_heavy(sz[a] + sz[x], sz[b])) {
      std::tie(c, d) = cut(a);
      return merge(merge(x, c), merge(d, b));
    } else {
      return merge(merge(x, a), b);
    }
```

```
153   } else {
154     return join(x, y);
155   }
156 }
157
158 // Balance the subtree at x;
159 void balance(int& x) {
160   if (sz[x] == 1) return;
161   if (too_heavy(sz[ch[x][0]], sz[ch[x][1]]) ||
162       too_heavy(sz[ch[x][1]], sz[ch[x][0]])) {
163     int a, b;
164     std::tie(a, b) = cut(x);
165     x = merge(a, b);
166   }
167 }
168
169 #endif
170
171 // Insert v to the subtree at x.
172 void insert(int& x, int v) {
173   if (!x) {
174     x = new_leaf(v);
175   } else if (sz[x] == 1) {
176     bool r = v >= val[x];
177     ch[x][r] = new_leaf(v);
178     ch[x][!r] = new_leaf(val[x]);
179     push_up(x);
180   } else {
181     bool r = v > val[ch[x][0]];
182     insert(ch[x][r], v);
183     push_up(x);
184     balance(x);
185   }
186 }
187
188 // Insert v.
189 void insert(int v) { insert(rt, v); }
190
191 // Remove v from the subtree at x.
192 bool remove(int& x, int v) {
193   if (!x) return false;
194   if (sz[x] == 1) {
195     if (val[x] == v) {
196       del_node(x);
197       return true;
198     } else {
199       return false;
200     }
201   } else {
202     bool r = v > val[ch[x][0]];
203     bool succ = remove(ch[x][r], v);
204     if (!ch[x][r]) {
```

```
      x = ch[x][!r];
    } else {
      push_up(x);
      balance(x);
    }
    return succ;
  }
}

// Remove v.
bool remove(int v) { return remove(rt, v); }

// Count the number of nodes less than v in the subtree at x.
int count_less_than(int x, int v) {
  if (!x) return 0;
  int res = 0;
  while (sz[x] > 1) {
    if (val[ch[x][0]] < v) {
      res += sz[ch[x][0]];
      x = ch[x][1];
    } else {
      x = ch[x][0];
    }
  }
  return res + (val[x] < v ? 1 : 0);
}

// Find the rank of v.
int find_rank(int v) { return count_less_than(rt, v) + 1; }

// Find the k-th element in the subtree at x.
// It is guaranteed that such an element exists.
int find_kth(int x, int k) {
  while (sz[x] > 1) {
    if (sz[ch[x][0]] >= k) {
      x = ch[x][0];
    } else {
      k -= sz[ch[x][0]];
      x = ch[x][1];
    }
  }
  return val[x];
}

// Find the k-th element.
int find_kth(int k) { return k > sz[rt] || k <= 0 ? -1 : find_kth(rt, k); }

// Find the predecessor of v.
int find_prev(int v) { return find_kth(find_rank(v) - 1); }

// Find the successor of v.
int find_next(int v) { return find_kth(find_rank(v + 1)); }
```

```
257
258 int main() {
259   int n;
260   std::cin >> n;
261   for (; n; --n) {
262     int op, x;
263     std::cin >> op >> x;
264     switch (op) {
265       case 1:
266         insert(x);
267         break;
268       case 2:
269         remove(x);
270         break;
271       case 3:
272         std::cout << find_rank(x) << '\n';
273         break;
274       case 4:
275         std::cout << find_kth(x) << '\n';
276         break;
277       case 5:
278         std::cout << find_prev(x) << '\n';
279         break;
280       case 6:
281         std::cout << find_next(x) << '\n';
282         break;
283     }
284   }
285   return 0;
286 }
287
```

利用合并和分裂，也可以实现文艺平衡树．下面是用 WBLT 实现的 文艺平衡树模板[2]，需要在向下访问节点时下传懒标记．

参考代码

```
1 #include <iostream>
2 #include <tuple>
3
4 // Change here to use different strategies to balance and to merge.
5 #define BALANCE_BY_ROTATING 0
6 #define ROTATE_BY_JOINING 1
7
8 constexpr int N = 2e5;
9 constexpr double ALPHA = 0.292;
10
11 int id, rt, ch[N][2], sz[N], val[N], lz[N];
12 int pool[N], top;
13
14 void push_up(int x) { sz[x] = sz[ch[x][0]] + sz[ch[x][1]]; }
15
```

```cpp
void lazy_reverse(int x) {
  if (!x) return;
  std::swap(ch[x][0], ch[x][1]);
  lz[x] ^= 1;
}

void push_down(int x) {
  if (lz[x]) {
    lazy_reverse(ch[x][0]);
    lazy_reverse(ch[x][1]);
    lz[x] = 0;
  }
}

// Return a new empty node.
int new_node() {
  int x = top ? pool[--top] : ++id;
  sz[x] = val[x] = ch[x][0] = ch[x][1] = 0;
  return x;
}

// Release a node for later use.
void del_node(int& x) {
  pool[top++] = x;
  x = 0;
}

// Return a new leaf node of value v.
int new_leaf(int v) {
  int x = new_node();
  val[x] = v;
  sz[x] = 1;
  return x;
}

// Return a new node with subtrees x and y.
int join(int x, int y) {
  int z = new_node();
  ch[z][0] = x;
  ch[z][1] = y;
  push_up(z);
  return z;
}

// Return subtrees of x and release x.
auto cut(int& x) {
  push_down(x);
  int y = ch[x][0];
  int z = ch[x][1];
  del_node(x);
  return std::make_pair(y, z);
}
```

```
68
69 // Check whether a subtree of weight SX is too heavy
70 //     in a tree of weight SX + SY.
71 bool too_heavy(int sx, int sy) {
72   // or sx > sy * 3;
73   return sy < ALPHA * (sx + sy);
74 }
75
76 #if BALANCE_BY_ROTATING
77
78 #if ROTATE_BY_JOINING
79
80 // Rotate the subtree at x such that ch[x][r] is the new root.
81 void rotate(int& x, bool r) {
82   int a, b, c, d;
83   std::tie(a, b) = cut(x);
84   if (r) {
85     std::tie(c, d) = cut(b);
86     x = join(join(a, c), d);
87   } else {
88     std::tie(c, d) = cut(a);
89     x = join(c, join(d, b));
90   }
91 }
92
93 #else
94
95 // Rotate the subtree at x such that ch[x][r] is the new root.
96 void rotate(int& x, bool r) {
97   int y = ch[x][r];
98   ch[x][r] = ch[y][!r];
99   ch[y][!r] = x;
100  push_up(x);
101  push_up(y);
102  x = y;
103 }
104
105 #endif
106
107 // Check if ch[x][!r] is too heavy so that a double rotation is needed.
108 bool need_double_rotation(int x, bool r) {
109   // or sz[ch[x][!r]] > sz[ch[x][r]] * 2;
110   return sz[ch[x][!r]] > sz[x] / (2 - ALPHA);
111 }
112
113 // Balance the subtree at x;
114 void balance(int& x) {
115   if (sz[x] == 1) return;
116   push_down(x);
117   bool r = sz[ch[x][1]] > sz[ch[x][0]];
118   if (!too_heavy(sz[ch[x][r]], sz[ch[x][!r]])) return;
119   push_down(ch[x][r]);
```

```
  if (need_double_rotation(ch[x][r], r)) {
    push_down(ch[ch[x][r]][!r]);
    rotate(ch[x][r], !r);
  }
  rotate(x, r);
}

// Merge two subtrees.
int merge(int x, int y) {
  if (!x || !y) return x | y;
  int a, b;
  if (too_heavy(sz[x], sz[y])) {
    std::tie(a, b) = cut(x);
    int z = join(a, merge(b, y));
    balance(z);
    return z;
  } else if (too_heavy(sz[y], sz[x])) {
    std::tie(a, b) = cut(y);
    int z = join(merge(x, a), b);
    balance(z);
    return z;
  } else {
    return join(x, y);
  }
}

#else

// Merge two subtrees.
int merge(int x, int y) {
  if (!x || !y) return x | y;
  int a, b, c, d;
  if (too_heavy(sz[x], sz[y])) {
    std::tie(a, b) = cut(x);
    if (too_heavy(sz[b] + sz[y], sz[a])) {
      std::tie(c, d) = cut(b);
      return merge(merge(a, c), merge(d, y));
    } else {
      return merge(a, merge(b, y));
    }
  } else if (too_heavy(sz[y], sz[x])) {
    std::tie(a, b) = cut(y);
    if (too_heavy(sz[a] + sz[x], sz[b])) {
      std::tie(c, d) = cut(a);
      return merge(merge(x, c), merge(d, b));
    } else {
      return merge(merge(x, a), b);
    }
  } else {
    return join(x, y);
  }
}
```

```
172
173 // Balance the subtree at x;
174 void balance(int& x) {
175   if (sz[x] == 1) return;
176   if (too_heavy(sz[ch[x][0]], sz[ch[x][1]]) ||
177       too_heavy(sz[ch[x][1]], sz[ch[x][0]])) {
178     int a, b;
179     std::tie(a, b) = cut(x);
180     x = merge(a, b);
181   }
182 }
183
184 #endif
185
186 // Split the subtree at x.
187 // The left half will have k elements.
188 std::pair<int, int> split(int x, int k) {
189   if (!x) return {0, 0};
190   if (!k) return {0, x};
191   if (k == sz[x]) return {x, 0};
192   int a, b;
193   std::tie(a, b) = cut(x);
194   if (k <= sz[a]) {
195     int ll, rr;
196     std::tie(ll, rr) = split(a, k);
197     return {ll, merge(rr, b)};
198   } else {
199     int ll, rr;
200     std::tie(ll, rr) = split(b, k - sz[a]);
201     return {merge(a, ll), rr};
202   }
203 }
204
205 // Reverse the interval [l, r].
206 void reverse(int l, int r) {
207   int ll, rr;
208   std::tie(rt, rr) = split(rt, r);
209   std::tie(ll, rt) = split(rt, l - 1);
210   lazy_reverse(rt);
211   rt = merge(ll, merge(rt, rr));
212 }
213
214 // Output the subtree at x.
215 void print(int x) {
216   if (sz[x] == 1) {
217     std::cout << val[x] << ' ';
218   } else {
219     push_down(x);
220     print(ch[x][0]);
221     print(ch[x][1]);
222   }
223 }
```

```
224
225 // Output the tree.
226 void print() {
227   print(rt);
228   std::cout << '\n';
229 }
230
231 // Build the tree for the interval [ll, rr].
232 int build(int ll, int rr) {
233   if (ll == rr) return new_leaf(ll);
234   int mm = (ll + rr) / 2;
235   return join(build(ll, mm), build(mm + 1, rr));
236 }
237
238 int main() {
239   int n, m;
240   std::cin >> n >> m;
241   rt = build(1, n);
242   for (; m; --m) {
243     int l, r;
244     std::cin >> l >> r;
245     reverse(l, r);
246   }
247   print();
248   return 0;
249 }
250
```

注意 WBLT 需要两倍的空间；涉及分裂与合并时，需要注意垃圾回收，及时回收无用的节点，否则空间不是线性的.

参考资料与注释

- Weight-balanced tree - Wikipedia[3]
- Nievergelt, J.; Reingold, E. M. (1973). "Binary Search Trees of Bounded Balance". SIAM Journal on Computing. 2: 33–43.
- Blum, Norbert; Mehlhorn, Kurt (1980). "On the average number of rebalancing operations in weight-balanced trees". Theoretical Computer Science. 11 (3): 303–320.
- Hirai, Y.; Yamamoto, K. (2011). "Balancing weight-balanced trees". Journal of Functional Programming. 21 (3): 287.
- Blelloch, Guy E.; Ferizovic, Daniel; Sun, Yihan (2016), "Just Join for Parallel Ordered Sets", Symposium on Parallel Algorithms and Architectures, Proc. of 28th ACM Symp. Parallel Algorithms and Architectures (SPAA 2016), ACM, pp. 253–264.
- Straka, Milan. (2011). "Adams'Trees Revisited: Correctness Proof and Efficient Implementation." International Symposium on Trends in Functional Programming. Berlin, Heidelberg: Springer Berlin Heidelberg.

[1] 普通平衡树模板  [2] 文艺平衡树模板  [3] Weight-balanced tree - Wikipedia 

10.15.5 替罪羊树

Authors: Ir1d, 0xis-cn

引入

替罪羊树 是一种依靠重构操作维持平衡的重量平衡树. 替罪羊树会在插入、删除操作后, 检测树是否发生失衡; 如果失衡, 将有针对性地进行重构以恢复平衡.

一般地, 替罪羊树不支持区间操作, 且无法完全持久化; 但它具有实现简单、常数较小的优点.

基本结构和操作

替罪羊树的核心操作是重构、插入和删除操作.

节点信息

替罪羊树需要存储以下信息, 用于树的自平衡操作:

- 树的结构信息:
  - `id`: 已使用节点数目;
  - `rt`: 根节点;
  - `lc[x]`, `rc[x]`: 左、右子节点;
  - `tot[x]`: 以 x 为根的子树大小 (每个节点计数为 1)[11];
  - `tot_active`: 整个树中未删除 (即 `cnt[x] != 0`) 的节点的数目.

当使用替罪羊树实现平衡树时, 还需要存储如下信息:

- 平衡树的节点信息:
  - `val[x]`: 节点存储的值;
  - `cnt[x]`: 节点存储的值的计数 (可能为 0);
  - `sz[x]`: 以 x 为根的子树存储的值的计数之和.

为了维护节点信息, 可以实现 `push_up` 操作:

参考实现

```
// Update node info from its children.
void push_up(int x) {
  tot[x] = 1 + tot[lc[x]] + tot[rc[x]];
  sz[x] = cnt[x] + sz[lc[x]] + sz[rc[x]];
}

```

应注意 `tot[x]` 和 `sz[x]` 的更新方式的不同.

重构操作

当树发生失衡时, 需要对某个子树进行重构, 使之尽可能平衡. 重构分为两步:

- 对要重构的子树做中序遍历, 将所有未删除节点存到序列中;
- 二分建树, 即取中点为根, 左右两侧递归地建子树, 并更新节点信息.

参考实现如下:

参考实现

```
// Tree -> Array.
void flatten(int x) {
  if (!x) return;
  flatten(lc[x]);
```

[11] 也可以只统计未删除节点数目, 此时不再需要统计 `tot_active`, 而需要统计所有占用节点数目 `tot_max`, 代码相应调整即可.

```
  if (cnt[x]) tmp[n_tmp++] = x;
  flatten(rc[x]);
}

// Array -> Tree.
int build(int ll, int rr) {
  if (ll > rr) return 0;
  int mm = (ll + rr) / 2;
  int x = tmp[mm];
  lc[x] = build(ll, mm - 1);
  rc[x] = build(mm + 1, rr);
  push_up(x);
  return x;
}

// Rebuild a subtree.
void rebuild(int& x) {
  n_tmp = 0;
  flatten(x);
  x = build(0, n_tmp - 1);
}

```

建树时注意维护节点信息，包括叶子节点的信息.

单次重构的复杂度是 $\Theta(|\ T_x|)$ 的，因此如果每次插入、删除时都进行重构，复杂度将难以接受. 替罪羊树的核心思想就在于对重构时机的选择，进而实现了 $O(\log n)$ 的均摊复杂度.

插入操作

插入操作时，可能会引起树的失衡. 为了判断树的失衡，需要引入参数 $\alpha \in (0.5, 1)$，通常的选择在 $0.7 \sim 0.8$ 之间.

如果新插入的节点的深度超过了 $\left\lfloor \log_{1/\alpha} |\ T\ | \right\rfloor$，其中，$|\ T\ |$ 为更新后的树的大小，就需要在回溯时寻找失衡发生的节点并进行重构. 此时，需要根据如下条件判断以 x 为根的子树失衡：

$$\max\{|\ T_{\text{left}(x)}|, |\ T_{\text{right}(x)}|\} > \alpha \cdot |\ T_x|,$$

其中，$\text{left}(x)$ 和 $\text{right}(x)$ 分别为 x 的左、右子节点，$|\ T_x|$ 为以 x 为根的子树大小.

插入操作的具体步骤如下：

- 首先利用二叉查找树的性质向下找到插入值的位置，下探时记录深度；
- 如果已经有节点，直接修改节点信息，否则新建节点；
- 自下而上回溯到根，更新节点信息；如果新建节点过深，还需要记录回溯过程中第一个（或任意一个）子树失衡的节点；
- 如果存在失衡节点，重构失衡节点的子树.

参考实现如下：

参考实现

```
// Insert v into subtree of x.
bool insert(int& x, int v, int dep) {
  bool check = false;
  if (!x) {
```

```
    x = ++id;
    val[x] = v;
    check = dep > std::log(tot[rt] + 1) / std::log(1 / ALPHA);
  }
  if (val[x] == v) {
    if (!cnt[x]) ++tot_active;
    ++cnt[x];
  } else if (v < val[x]) {
    check = insert(lc[x], v, dep + 1);
  } else {
    check = insert(rc[x], v, dep + 1);
  }
  push_up(x);
  if (check && (tot[lc[x]] > ALPHA * tot[x] || tot[rc[x]] > ALPHA * tot[x]))
{
    rebuild(x);
    return false;
  }
  return check;
}

// Insert v into the tree.
void insert(int v) { insert(rt, v, 0); }

```

注意，单次插入至多引起一次重构．如果没有新增节点或是新增节点并没有过深，又或是本次回溯过程中已经执行过重构，就不需要继续判断失衡了．多余的重构可能会导致效率损失[12]．回溯过程中的第一个失衡节点，就是所谓的「替罪羊」．

删除操作

删除操作的处理则非常简单．替罪羊树的删除策略是「懒删除」，即节点为空时，不移除节点，而是留待后续处理．

当然，如果树中空节点过多，树的访问效率会大大下降．因此，替罪羊树维护两个计数，整个树中未删除节点的数目和整个树实际使用的节点数目．对于选定的阈值[13] $\alpha \in (0,1)$，当前者与后者的比值下降到 α 以下时，就对整个树做一次重构，重构时删除所有空节点．

参考实现

```
// Remove v from subtree of x.
bool remove(int x, int v) {
  if (!x) return false;
  bool succ = true;
  if (v < val[x]) {
    succ = remove(lc[x], v);
  } else if (v > val[x]) {
    succ = remove(rc[x], v);
```

[12] 根据后文的复杂度分析可知，这些效率损失仅意味着更大的常数因子，而复杂度依然是正确的．因为判断树深可能会涉及较多的浮点数对数运算，不判断树深只判断失衡的代码在某些数据中可能更快．

[13] 不必与上文插入操作时选取的参数相同．尽管原论文做了这样的假定，但是选取不同的参数只会导致单次操作的复杂度中常数项的变化，整体复杂度依然是正确的．

```
  } else if (cnt[x]) {
    --cnt[x];
    if (!cnt[x]) --tot_active;
  } else {
    succ = false;
  }
  push_up(x);
  return succ;
}

// Remove v from the tree.
bool remove(int v) {
  bool succ = remove(rt, v);
  if (!tot_active) {
    rt = 0;
  } else if (succ && tot_active < tot[rt] * ALPHA) {
    rebuild(rt);
  }
  return succ;
}

```

时间复杂度

大小为 n 的替罪羊树中，单次访问节点的时间复杂度是 $O(\log n)$ 的；$\Theta(n)$ 次插入和删除操作的时间复杂度也是均摊单次 $O(\log n)$ 的.

本节对替罪羊树的时间复杂度仅做简要论证，详细证明请参考原论文.

替罪羊树的时间复杂度的论证

由于采用懒删除的策略，未删除节点数目为 n 的替罪羊树可能占用了 $\alpha^{-1}n$ 个节点. 由于仅仅相差一个常数因子，本文在表述中并不区分替罪羊树的未删除节点数目和占用节点数目，而统一称为「树的大小」.

1. **访问操作**：访问操作的复杂度得以保证，是因为大小为 n 的替罪羊树的树高总是 $O(\log n)$ 的.

   首先，区分两个概念：

   - α-重量平衡：所有节点处，左、右子节点的子树的大小均不超过该节点处子树的大小的 α 倍；
   - α-高度平衡：树的高度不超过 $\left\lfloor \log_{1/\alpha} \mid T \mid \right\rfloor$，其中，$\mid T \mid$ 为树的大小.

   α-重量平衡可以推出 α-高度平衡，因为深度每增加一，子树大小就减少到不超过原来的 α 倍；反过来则不一定成立. 更严格地说，每次操作结束后，替罪羊树都总是 α-高度平衡的[14]，这就保证了访问操作的复杂度.

   只有插入操作会改变树的结构，所以只需要说明每次插入操作后，替罪羊树都仍是 α-高度平衡的. 如果新插入的节点过深，造成了整个树不再 α-高度平衡，那么自该节点回溯至根时，至少会碰上一个节点，即「替罪羊」，它的子树不再 α-重量平衡. 将其重构后，子树的高度将降低至少一，故而新插入的节点将不再过深.
2. **插入操作**：插入操作的复杂度是均摊 $O(\log n)$ 的.

[14] 按原文定义，n 指未删除节点的数目，故而只能保证树高不超过 $\left\lfloor \log_{1/\alpha} n \right\rfloor + 1$，这称为弱 α-高度平衡. 此处没有细究该常数项的差异.

设在某次插入操作后，节点 x 处发生一次子树的重构，时间成本为 $\Theta(|\ T_x|)$. 因为节点 x 刚插入时，或者它刚刚经历了（自身或祖先节点的）上一次重构之后，它的左右子树至多只相差一个节点. 而在这次重构之前，节点 x 处必然成立

$$\max\{|\ T_{\text{left}(x)}|, |\ T_{\text{right}(x)}|\} > \alpha \cdot |\ T_x|\ .$$

这一条件保证左右子树的大小的差值至少为 $(2\alpha - 1)|\ T_x|$. 因而，这两次重构之间，子树 T_x 中插入了 $\Omega(|\ T_x|)$ 个节点.

利用摊还分析可知[15]，如果每次插入节点时，都在（可能的重构前）自根到该节点的路径上的每个节点处增加 $\Theta(1)$ 的势能，那么到节点 x 处子树重构前，必然已经在节点 x 处累积了 $\Omega(|\ T_x|)$ 的势能，足以用于偿还 x 处子树重构的成本 $\Theta(|\ T_x|)$. 因为树的深度都是 $O(\log n)$ 的，所以单次插入增加的势能是 $O(\log n)$ 的；这说明，$\Theta(n)$ 次插入操作中势能增加的总和是 $O(n \log n)$ 的. 由此，子树重构的总成本也是 $O(n \log n)$ 的，单次插入操作（含重构）的均摊时间复杂度就是 $O(\log n)$ 的.

注意，分析中没有假定在节点 x 处的两次重构之间，子树 T_x 内部没有发生其它的重构. 因此，只要只重构满足失衡条件的节点处的子树，就能保证复杂度正确.

3. **删除操作**：删除操作的复杂度也是均摊 $O(\log n)$ 的.

   删除引起的重构会导致整个树不含空节点. 而某次删除引起重构之前，整个树中已经有 $\Theta(n)$ 个空节点，这意味着至少进行了 $\Theta(n)$ 次删除操作. 因为每次删除操作的寻址的复杂度是 $O(\log n)$ 的，且单次重构的复杂度是 $\Theta(n)$，所以这 $\Theta(n)$ 次删除操作的实际时间成本为 $\Theta(n)O(\log n) + \Theta(n)$ 的. 故而，单次删除的均摊复杂度为 $O(\log n)$ 的.

平衡树操作

本节介绍用替罪羊树维护可重集的方法.

除上节介绍的操作外，其余操作均为平衡树的常见操作. 但是，因为替罪羊树中可能存在空节点，这些操作也需要相应调整.

查询排名

利用二叉查找树的性质向下查找节点位置，过程中记录路径左侧存储的值的数目即可.

参考实现

```
// Find the rank of v, i.e., #{val < v} + 1.
int find_rank(int v) {
  int res = 0;
  int x = rt;
  while (x && val[x] != v) {
    if (v < val[x]) {
      x = lc[x];
    } else {
      res += sz[lc[x]] + cnt[x];
      x = rc[x];
```

[15] 有些文章会简单分析成 $\Omega(|\ T_x|)$ 次插入对应一次重构，故而均摊复杂度为 $\dfrac{\Omega(|\ T_x|)O(\log n) + \Theta(|\ T_x|)}{\Omega(|\ T_x|)} = O(\log n)$. 这样的思路可以辅助理解均摊复杂度为什么正确，但并不严谨. 这是因为，一次插入可能对应着多个祖先节点的重构，故而当节点 x 发生重构时，子树内未引起重构的节点数目并不显然是 $\Omega(|\ T_x|)$ 的.

```
    }
  }
  return res + sz[lc[x]] + 1;
}

```

根据排名查询值

利用节点记录的子树存储值的数目信息向下查找即可．注意可能存在计数为零的节点．

参考实现

```
// Find the k-th smallest element.
int find_kth(int k) {
  if (k <= 0 || sz[rt] < k) return -1;
  int x = rt;
  for (;;) {
    if (k <= sz[lc[x]]) {
      x = lc[x];
    } else if (k <= sz[lc[x]] + cnt[x]) {
      return val[x];
    } else {
      k -= sz[lc[x]] + cnt[x];
      x = rc[x];
    }
  }
  return -1;
}

```

查询前驱、后继

以上两种功能结合即可．

参考实现

```
// Find predecessor.
int find_prev(int x) { return find_kth(find_rank(x) - 1); }

// Find successor.
int find_next(int x) { return find_kth(find_rank(x + 1)); }

```

如果想直接实现，应注意处理计数为零的节点．

参考实现

本节的最后，给出模板题 普通平衡树[1] 的参考实现．

参考实现

```
#include <cmath>
```

```cpp
#include <iostream>

constexpr int N = 2e6;
constexpr double ALPHA = 0.7;
int id, rt, lc[N], rc[N], tot[N], tot_active;  // Tree structure info.
int tmp[N], n_tmp;                             // Space for rebuilding trees.
int val[N], cnt[N], sz[N];                     // Node info.

// Update node info from its children.
void push_up(int x) {
  tot[x] = 1 + tot[lc[x]] + tot[rc[x]];
  sz[x] = cnt[x] + sz[lc[x]] + sz[rc[x]];
}

// Tree -> Array.
void flatten(int x) {
  if (!x) return;
  flatten(lc[x]);
  if (cnt[x]) tmp[n_tmp++] = x;
  flatten(rc[x]);
}

// Array -> Tree.
int build(int ll, int rr) {
  if (ll > rr) return 0;
  int mm = (ll + rr) / 2;
  int x = tmp[mm];
  lc[x] = build(ll, mm - 1);
  rc[x] = build(mm + 1, rr);
  push_up(x);
  return x;
}

// Rebuild a subtree.
void rebuild(int& x) {
  n_tmp = 0;
  flatten(x);
  x = build(0, n_tmp - 1);
}

// Insert v into subtree of x.
bool insert(int& x, int v, int dep) {
  bool check = false;
  if (!x) {
    x = ++id;
    val[x] = v;
    check = dep > std::log(tot[rt] + 1) / std::log(1 / ALPHA);
  }
  if (val[x] == v) {
    if (!cnt[x]) ++tot_active;
    ++cnt[x];
```

```
  } else if (v < val[x]) {
    check = insert(lc[x], v, dep + 1);
  } else {
    check = insert(rc[x], v, dep + 1);
  }
  push_up(x);
  if (check && (tot[lc[x]] > ALPHA * tot[x] || tot[rc[x]] > ALPHA * tot[x]))
{
    rebuild(x);
    return false;
  }
  return check;
}

// Insert v into the tree.
void insert(int v) { insert(rt, v, 0); }

// Remove v from subtree of x.
bool remove(int x, int v) {
  if (!x) return false;
  bool succ = true;
  if (v < val[x]) {
    succ = remove(lc[x], v);
  } else if (v > val[x]) {
    succ = remove(rc[x], v);
  } else if (cnt[x]) {
    --cnt[x];
    if (!cnt[x]) --tot_active;
  } else {
    succ = false;
  }
  push_up(x);
  return succ;
}

// Remove v from the tree.
bool remove(int v) {
  bool succ = remove(rt, v);
  if (!tot_active) {
    rt = 0;
  } else if (succ && tot_active < tot[rt] * ALPHA) {
    rebuild(rt);
  }
  return succ;
}

// Find the rank of v, i.e., #{val < v} + 1.
int find_rank(int v) {
  int res = 0;
  int x = rt;
  while (x && val[x] != v) {
    if (v < val[x]) {
```

```
      x = lc[x];
    } else {
      res += sz[lc[x]] + cnt[x];
      x = rc[x];
    }
  }
  return res + sz[lc[x]] + 1;
}

// Find the k-th smallest element.
int find_kth(int k) {
  if (k <= 0 || sz[rt] < k) return -1;
  int x = rt;
  for (;;) {
    if (k <= sz[lc[x]]) {
      x = lc[x];
    } else if (k <= sz[lc[x]] + cnt[x]) {
      return val[x];
    } else {
      k -= sz[lc[x]] + cnt[x];
      x = rc[x];
    }
  }
  return -1;
}

// Find predecessor.
int find_prev(int x) { return find_kth(find_rank(x) - 1); }

// Find successor.
int find_next(int x) { return find_kth(find_rank(x + 1)); }

int main() {
  int n;
  std::cin >> n;
  for (; n; --n) {
    int op, x;
    std::cin >> op >> x;
    switch (op) {
      case 1:
        insert(x);
        break;
      case 2:
        remove(x);
        break;
      case 3:
        std::cout << find_rank(x) << '\n';
        break;
      case 4:
        std::cout << find_kth(x) << '\n';
        break;
      case 5:
```

```
156         std::cout << find_prev(x) << '\n';
157         break;
158       case 6:
159         std::cout << find_next(x) << '\n';
160         break;
161     }
162   }
163   return 0;
164 }
165
```

参考资料

- Galperin, Igal, and Ronald L. Rivest. "Scapegoat trees." Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms. 1993.
- Scapegoat Tree - Wikipedia[2]
- 替罪羊树 - riteme 的博客[3]

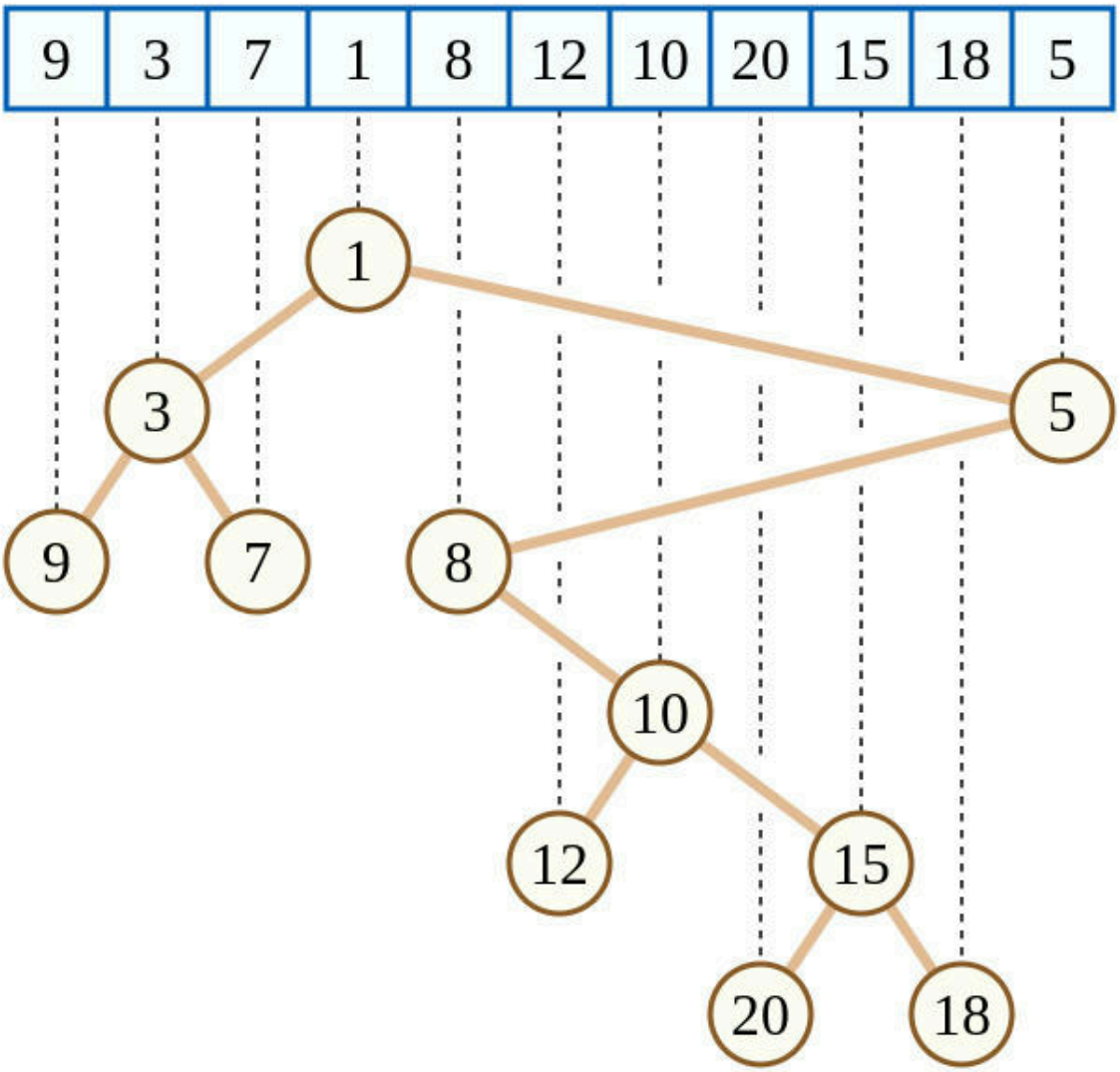
[1] 普通平衡树  [2] Scapegoat Tree - Wikipedia  [3] 替罪羊树 - riteme 的博客 

10.15.6 笛卡尔树

Authors: sshwy, zhouyuyang2002, StudyingFather, Ir1d, ouuan, Enter-tainer, AtomAlpaca

引入

笛卡尔树是一种二叉树，每一个节点由一个键值二元组 (k, w) 构成．要求 k 满足二叉搜索树（BST）的性质，而 w 满足堆的性质．如果笛卡尔树的 k, w 键值确定，且 k 互不相同，w 也互不相同，那么这棵笛卡尔树的结构是唯一的．如下图：



（图源自维基百科）

上面这棵笛卡尔树相当于把数组元素值当作键值 w，而把数组下标当作键值 k．可以发现，这棵树的键值 k 满足 BST 的性质，而键值 w 满足小根堆的性质．同时根据二叉搜索树的性质，可以发现这种特殊的笛卡尔树满足一棵子树内的下标是一个连续区间．

竞赛中使用笛卡尔树时，常用数组下标作为二元组的键值 k，数组下标 k 满足 BST 性质．

下文使用 k, w 时，默认 k 满足 BST 性质，w 满足堆的性质．

单调栈构建笛卡尔树

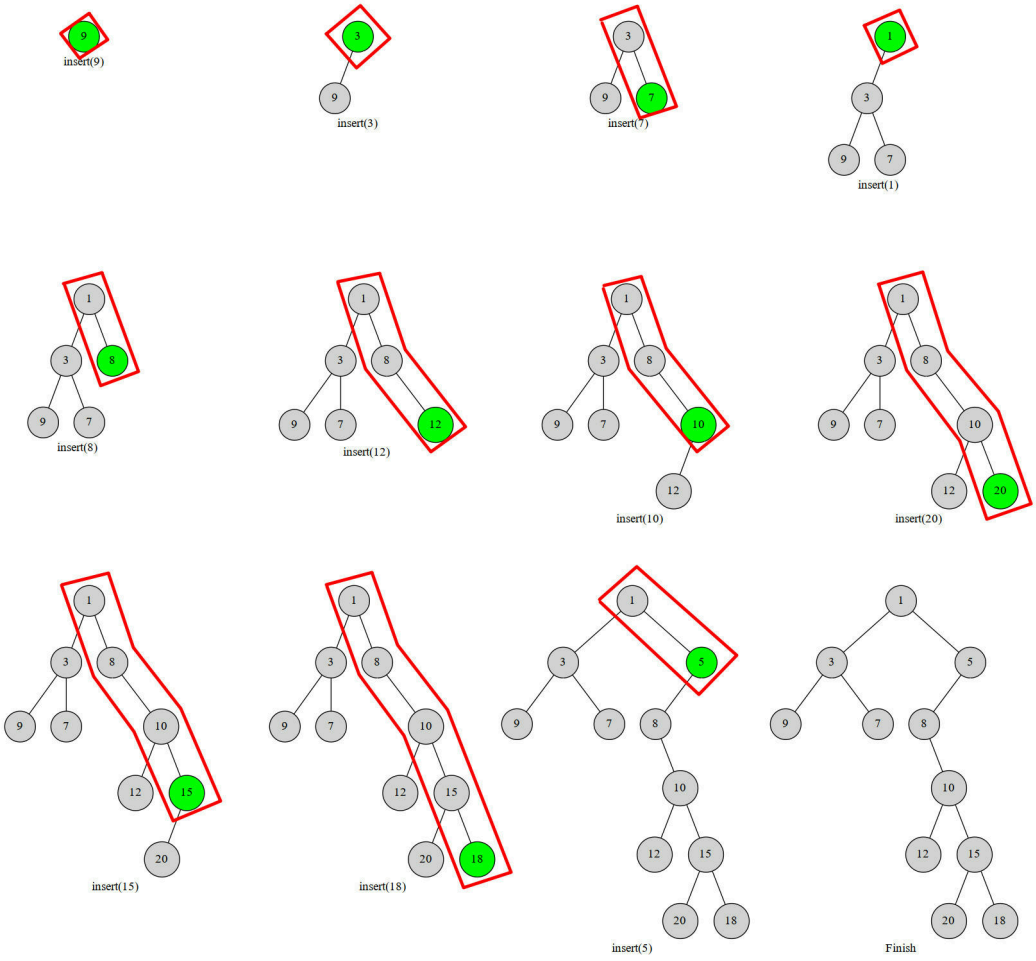
过程

我们考虑将元素按 k 升序依次插入到当前的笛卡尔树中．

对于一棵笛卡尔树，定义「右链」为从根节点开始一直走右儿子，直到走到一个没有右儿子的节点形成的链．则插入节点后，这个节点一定在右链上．因为是按照满足 BST 性质的 k 升序插入，那么这个新插入的节点必然在树的 **最右端**．这个节点不可能是一个左儿子，也没有右儿子．

于是我们执行这样一个过程，从下往上比较右链节点与当前节点 u 的 w，如果找到了一个右链上的节点 x 满足 $w_x < w_u$，就把 u 接到 x 的右儿子上，而 x 原本的右子树就变成 u 的左子树．

图中红框部分就是我们始终维护的右链：



显然每个数最多进出右链一次（或者说每个点在右链中存在的是一段连续的时间）．这个过程可以用单调栈维护，栈中维护当前笛卡尔树的右链上的节点．一个点不在右链上了就把它弹掉．这样每个点最多进出一次，复杂度 $O(n)$．

> **笛卡尔树与 Treap**
>
> 实际上，Treap 是笛卡尔树的一种，只不过 Treap 中 w 的值完全随机．Treap 有线性的构建算法，如果提前将键值 k 排好序，是可以使用上述单调栈算法完成构建过程的，只不过很少会这么用．

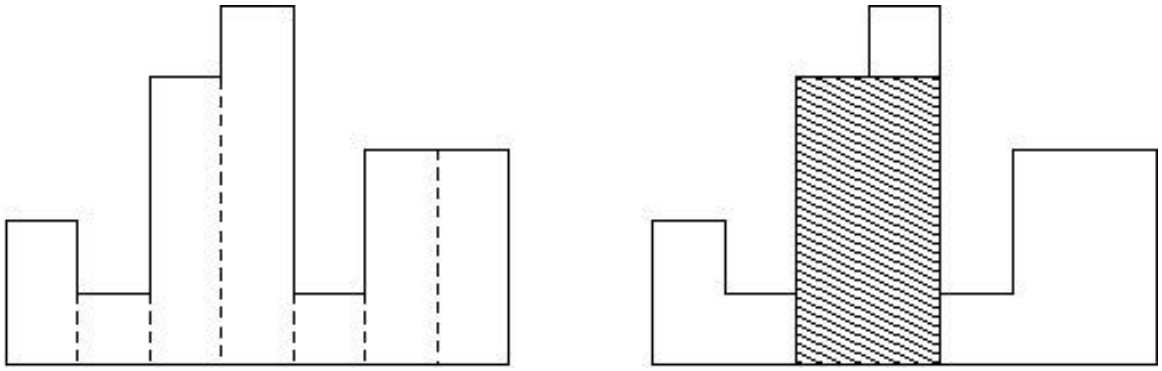
C++ 实现

```cpp
// stk 维护笛卡尔树中节点对应到序列中的下标
for (int i = 1; i <= n; i++) {
  int k = top;  // top 表示操作前的栈顶，k 表示当前栈顶
  while (k > 0 && w[stk[k]] > w[i]) k--;  // 维护右链上的节点
  if (k) rs[stk[k]] = i;  // 栈顶元素.右儿子 := 当前元素
  if (k < top) ls[i] = stk[k + 1];  // 当前元素.左儿子 := 上一个被弹出的元素
  stk[++k] = i;                     // 当前元素入栈
  top = k;
}
```

例题

HDU 1506. Largest Rectangle in a Histogram[1]

n 个位置，每个位置上的高度是 h_i，求最大子矩形．如下图：



阴影部分就是图中的最大子矩阵．

解题思路

具体地，我们把下标作为键值 k，h_i 作为键值 w 满足小根堆性质，构建一棵 (i, h_i) 的笛卡尔树．

这样我们枚举每个节点 u，把 w_u（即节点 u 的高度 h）作为最大子矩阵的高度．由于我们建立的笛卡尔树满足小根堆性质，因此 u 的子树内的节点的高度都大于等于 u．而我们又知道 u 子树内的下标是一段连续的区间．于是我们只需要知道子树的大小，然后就可以算这个区间的最大子矩阵的面积了．用每一个点计算出来的值更新答案即可．显然这个可以一次 DFS 完成，因此复杂度是 $O(n)$ 的．

参考实现

```cpp
#include <algorithm>
#include <cstring>
#include <iostream>
using namespace std;
using ll = long long;
constexpr int N = 100000 + 10, INF = 0x3f3f3f3f;

struct node {
  int idx, val, par, ch[2];

  friend bool operator<(node a, node b) { return a.idx < b.idx; }

  void init(int _idx, int _val, int _par) {
    idx = _idx, val = _val, par = _par, ch[0] = ch[1] = 0;
  }
} tree[N];

int root, top, stk[N];
ll ans;

int cartesian_build(int n) {  // 建树，满足小根堆性质
  for (int i = 1; i <= n; i++) {
```

```
    int k = i - 1;
    while (tree[k].val > tree[i].val) k = tree[k].par;
    tree[i].ch[0] = tree[k].ch[1];
    tree[k].ch[1] = i;
    tree[i].par = k;
    tree[tree[i].ch[0]].par = i;
  }
  return tree[0].ch[1];
}

int dfs(int x) {  // 一次dfs更新答案就可以了
  if (!x) return 0;
  int sz = dfs(tree[x].ch[0]);
  sz += dfs(tree[x].ch[1]);
  ans = max(ans, (ll)(sz + 1) * tree[x].val);
  return sz + 1;
}

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  int n, hi;
  while (cin >> n, n) {
    tree[0].init(0, 0, 0);
    for (int i = 1; i <= n; i++) {
      cin >> hi;
      tree[i].init(i, hi, 0);
    }
    root = cartesian_build(n);
    ans = 0;
    dfs(root);
    cout << ans << '\n';
  }
  return 0;
}
```

参考资料

笛卡尔树 - 维基百科[2]

外部参考资料

[1] HDU 1506. Largest Rectangle in a Histogram

[2] 笛卡尔树 - 维基百科

10.15.7 Size Balanced Tree

Size Balanced Tree (SBT) 是由中国 OI 选手陈启峰在 2007 年提出的一种自平衡二叉搜索树 (Self-Balanced Binary Search Tree, SBBST), 通过检查子树的节点数量进行自身的平衡维护. 相比于红黑树, AVL 等主流自平衡二叉搜索树而言， Size Balanced Tree 支持在 $O(\log n)$ 的时间复杂度内查询某个键值在树中的排名 (rank).

节点定义

相比与普通二叉搜索树， SBT 的每个节点 N 仅需要多维护一个整数字段 `size`, 用于储存以 N 为根的子树中节点的个数. 节点类型 `Node` 的具体定义如下:

Identifier	Type	Description
`left`	`Node*`	左子节点引用
`right`	`Node*`	右子节点引用
`size`	`int`	以该节点为根的子树中节点的个数

性质

Size Balanced Tree 中任意节点 N 满足如下几条性质：

```
size(N.left) >= size(N.right.left)
size(N.left) >= size(N.right.right)
size(N.right) >= size(N.left.left)
size(N.right) >= size(N.left.right)
```

使用自然语言可描述为：任意节点的 `size` 不小于其兄弟节点（Sibling）的所有子节点（Nephew）的 `size`.

平衡维护

旋转

SBT 主要通过旋转操作改变自身高度从而进行平衡维护. 其旋转操作与绝大部分自平衡二叉搜索树类似, 唯一区别在于在完成旋转之后需要对旋转过程中左右子节点发生改变的节点更新 `size`. 示例代码如下：

```
void updateSize() {
  USize leftSize = this->left != nullptr ? this->left->size : 0;
  USize rightSize = this->right != nullptr ? this->right->size : 0;
  this->size = leftSize + rightSize + 1;
}

static void rotateLeft(NodePtr& node) {
  assert(node != nullptr);
  // clang-format off
  //     |                      |
  //     N                      S
  //    / \     l-rotate(N)    / \
  //   L   S    ==========>    N   R
  //      / \                 / \
  //     M   R               L   M
  // clang-format on
  NodePtr successor = node->right;
  node->right = successor->left;
  successor->left = node;

  node->updateSize();
  successor->updateSize();

  node = successor;
}

static void rotateRight(NodePtr& node) {
  assert(node != nullptr);
  // clang-format off
  //       |                    |
```

```
  //       N                   S
  //      / \   r-rotate(N)   / \
  //     S   R  ==========>  L   N
  //    / \                     / \
  //   L   M                   M   R
  // clang-format on
  NodePtr successor = node->left;
  node->left = successor->right;
  successor->right = node;

  node->updateSize();
  successor->updateSize();

  node = successor;
}
```

维护

Case 1

`size(N.left) < size(N.right.left)`

```
if (size(node->right->left) > size(node->left)) {
  // clang-format off
  //     |                     |                      |
  //     N                     N                    [M]
  //    / \    r-rotate(R)    / \     l-rotate(N)    / \
  //  <L>  R   ==========>  <L> [M]   ==========>   N   R
  //      /                       \               /
  //    [M]                        R            <L>
  // clang-format on
  rotateRight(node->right);
  rotateLeft(node);
  fixBalance(node->left);
  fixBalance(node->right);
  fixBalance(node);
  return;
}
```

Case 2

`size(N.left) < size(N.right.right)`

```
if (size(node->right->right) > size(node->left)) {
  // clang-format off
  //     |                       |
  //     N                       R
  //    / \     l-rotate(N)     / \
  //  <L>  R    ==========>    N  [M]
  //        \                 /
  //        [M]             <L>
  // clang-format on
  rotateLeft(node);
  fixBalance(node->left);
  fixBalance(node);
  return;
```

```
14 }
```

Case 3

`size(N.right) < size(N.left.left)`

```
1  if (size(node->left->left) > size(node->right)) {
2    // clang-format off
3    //       |                       |
4    //       N                       L
5    //      / \     r-rotate(N)     / \
6    //     L  <R>   ==========>   [M]  N
7    //    /                           \
8    //  [M]                           <R>
9    // clang-format on
10   rotateRight(node);
11   fixBalance(node->right);
12   fixBalance(node);
13   return;
14 }
```

Case 4

`size(N.right) < size(N.left.right)`

```
1  if (size(node->left->right) > size(node->right)) {
2    // clang-format off
3    //     |                     |                     |
4    //     N                     N                    [M]
5    //    / \    l-rotate(L)    / \     r-rotate(N)    / \
6    //   L  <R>  ==========>  [M] <R>   ==========>   L   N
7    //    \                   /                         \
8    //    [M]                L                          <R>
9    // clang-format on
10   rotateLeft(node->left);
11   rotateRight(node);
12   fixBalance(node->left);
13   fixBalance(node->right);
14   fixBalance(node);
15   return;
16 }
```

操作

插入

SBT 的插入操作需要在完成普通二叉搜索树的插入操作的基础上递归地进行节点 `size` 字段的更新及平衡维护. 示例代码如下:

```
1  if (compare(key, node->key)) {
2    /* key < node->key */
3    if (node->left == nullptr) {
4      node->left = Node::from(key, value);
5      node->updateSize();
6    } else {
7      insert(node->left, key, value, replace);
8      node->updateSize();
```

```
    fixBalance(node);
  }
} else {
  /* key > node->key */
  if (node->right == nullptr) {
    node->right = Node::from(key, value);
    node->updateSize();
  } else {
    insert(node->right, key, value, replace);
    node->updateSize();
    fixBalance(node);
  }
}
```

删除

根据 Size Balanced Tree 的提出者陈启峰在其论文中对于删除操作的描述：

> It can result in a destroyed SBT. But with the insertion above, a BST is still kept at the height of $O(\log n)$ where n is the total number of insertions, not the current size.

删除操作虽然有可能使得 SBT 的性质被打破，但并不会使树的高度增高，因此不会影响后续操作的效率．但在实际情况下，如果在一次批量插入操作后只进行大量的删除和查询操作，依然有可能由于树的失衡影响整体效率，因此本文在实现 SBT 的删除操作时依然选择加入平衡维护．参考代码如下：

```
bool remove(NodePtr& node, K key, NodeConsumer action) {
  assert(node != nullptr);

  if (key != node->key) {
    if (compare(key, node->key)) {
      /* key < node->key */
      NodePtr& left = node->left;
      if (left != nullptr && remove(left, key, action)) {
        node->updateSize();
        fixBalance(node);
        return true;
      } else {
        return false;
      }
    } else {
      /* key > node->key */
      NodePtr& right = node->right;
      if (right != nullptr && remove(right, key, action)) {
        node->updateSize();
        fixBalance(node);
        return true;
      } else {
        return false;
      }
    }
  }

  assert(key == node->key);
  action(node);

```

```
  if (node->isLeaf()) {
    // Case 1: no child
    node = nullptr;
  } else if (node->right == nullptr) {
    // Case 2: left child only
    // clang-format off
    //     P
    //     |  remove(N)  P
    //     N  ========>  |
    //    /              L
    //   L
    // clang-format on
    node = node->left;
  } else if (node->left == nullptr) {
    // Case 3: right child only
    // clang-format off
    //   P
    //   |    remove(N)  P
    //   N    ========>  |
    //    \              R
    //     R
    // clang-format on
    node = node->right;
  } else if (node->right->left == nullptr) {
    // Case 4: both left and right child, right child has no left child
    // clang-format off
    //    |                |
    //    N    remove(N)    R
    //   / \   ========>   /
    //  L   R             L
    // clang-format on
    NodePtr right = node->right;
    swapNode(node, right);
    right->right = node->right;
    node = right;
    node->updateSize();
    fixBalance(node);
  } else {
    // Case 5: both left and right child, right child is not a leaf
    // clang-format off
    //   Step 1. find the node N with the smallest key
    //           and its parent P on the right subtree
    //   Step 2. swap S and N
    //   Step 3. remove node N like Case 1 or Case 3
    //   Step 4. update size for all nodes on the path
    //           from S to P
    //     |                 |
    //     N                 S                 |
    //    / \               / \                S
    //   L  ..  swap(N, S)  L  ..  remove(N)   / \
    //       |  =========>      |  ========>  L  ..
    //       P                 P               |
    //      / \               / \              P
    //     S  ..             N  ..            / \
```

```
85      //      \                  \                R  ..
86      //       R                  R
87      //
88      // clang-format on
89
90      std::stack<NodePtr> path;
91
92      // Step 1
93      NodePtr successor = node->right;
94      NodePtr parent = node;
95      path.push(node);
96
97      while (successor->left != nullptr) {
98        path.push(successor);
99        parent = successor;
100       successor = parent->left;
101     }
102
103     // Step 2
104     swapNode(node, successor);
105
106     // Step 3
107     parent->left = node->right;
108     // Restore node
109     node = successor;
110
111     // Step 4
112     while (!path.empty()) {
113       path.top()->updateSize();
114       path.pop();
115     }
116   }
117
118   return true;
119 }
```

值得注意的是，在上述代码的 Case 5 中使用后继节点 S（也可以选择前驱节点）替换待删除节点 N 并删除替换后的 N 以后，需要更新替换前 S 节点的父节点 P 到替换后的 S 节点这条路径（如代码中注释所示）上的所有节点的 `size` 字段．本文的实现选择使用栈依次记录路径上的节点，最后再按遍历的相反顺序出栈进行更新．

查询排名

由于 SBT 节点中储存了子树节点个数的信息，因此可以在 $O(\log n)$ 的时间复杂度下查询某个 `key` 的排名（或者大于/小于某个 `key` 的节点个数）．示例代码如下：

```
1 USize countLess(ConstNodePtr node, K key, bool countEqual = false) const {
2   if (node == nullptr) {
3     return 0;
4   } else if (key < node->key) {
5     return countLess(node->left, key, countEqual);
6   } else if (key > node->key) {
7     return size(node->left) + 1 + countLess(node->right, key, countEqual);
8   } else {
9     return size(node->left) + (countEqual ? 1 : 0);
```

```
  }
}

USize countGreater(ConstNodePtr node, K key, bool countEqual = false) const {
  if (node == nullptr) {
    return 0;
  } else if (key < node->key) {
    return size(node->right) + 1 + countGreater(node->left, key, countEqual);
  } else if (key > node->key) {
    return countGreater(node->right, key, countEqual);
  } else {
    return size(node->right) + (countEqual ? 1 : 0);
  }
}
```

参考代码

下面的代码是用 SBT 实现的 `Map`，即有序不可重映射：

完整代码

```
/**
 * @file SizeBalancedTreeMap.hpp
 * @brief An SizeBalancedTree-based map implementation
 * @details The map is sorted according to the natural ordering of its
 *  keys or by a {@code Compare} function provided; This implementation
 *  provides guaranteed log(n) time cost for the contains, get, insert
 *  and remove operations.
 * @author [r.ivance](https://github.com/RIvance)
 */

#ifndef SIZE_BALANCED_TREE_MAP_HPP
#define SIZE_BALANCED_TREE_MAP_HPP

#include <cassert>
#include <cstddef>
#include <cstdint>
#include <functional>
#include <memory>
#include <stack>
#include <utility>
#include <vector>

/**
 * An SizeBalancedTree-based map implementation
 * http://wcipeg.com/wiki/Size_Balanced_Tree
 * @tparam Key the type of keys maintained by this map
 * @tparam Value the type of mapped values
 * @tparam Compare the compare function
 */
template <typename Key, typename Value, typename Compare = std::less<Key> >
class SizeBalancedTreeMap {
 private:
```

```
33   using USize = size_t;
34
35   Compare compare = Compare();
36
37  public:
38   struct Entry {
39     Key key;
40     Value value;
41
42     bool operator==(const Entry &rhs) const noexcept {
43       return this->key == rhs.key && this->value == rhs.value;
44     }
45
46     bool operator!=(const Entry &rhs) const noexcept {
47       return this->key != rhs.key || this->value != rhs.value;
48     }
49   };
50
51  private:
52   struct Node {
53     using Ptr = std::shared_ptr<Node>;
54     using Provider = const std::function<Ptr(void)> &;
55     using Consumer = const std::function<void(const Ptr &)> &;
56
57     Key key;
58     Value value{};
59
60     Ptr left = nullptr;
61     Ptr right = nullptr;
62
63     USize size = 1;
64
65     explicit Node(Key k) : key(std::move(k)) {}
66
67     explicit Node(Key k, Value v) : key(std::move(k)), value(std::move(v))
   {}
68
69     ~Node() = default;
70
71     inline bool isLeaf() const noexcept {
72       return this->left == nullptr && this->right == nullptr;
73     }
74
75     inline void updateSize() {
76       USize leftSize = this->left != nullptr ? this->left->size : 0;
77       USize rightSize = this->right != nullptr ? this->right->size : 0;
78       this->size = leftSize + rightSize + 1;
79     }
80
81     inline Entry entry() const { return Entry{key, value}; }
82
```

```
    static Ptr from(const Key &k) { return
std::make_shared<Node>(Node(k)); }

    static Ptr from(const Key &k, const Value &v) {
      return std::make_shared<Node>(Node(k, v));
    }
  };

  using NodePtr = typename Node::Ptr;
  using ConstNodePtr = const NodePtr &;
  using NodeProvider = typename Node::Provider;
  using NodeConsumer = typename Node::Consumer;

  NodePtr root = nullptr;

  using K = const Key &;
  using V = const Value &;

 public:
  using EntryList = std::vector<Entry>;
  using KeyValueConsumer = const std::function<void(K, V)> &;
  using MutKeyValueConsumer = const std::function<void(K, Value &)> &;
  using KeyValueFilter = const std::function<bool(K, V)> &;

  class NoSuchMappingException : protected std::exception {
   private:
    const char *message;

   public:
    explicit NoSuchMappingException(const char *msg) : message(msg) {}

    const char *what() const noexcept override { return message; }
  };

  SizeBalancedTreeMap() noexcept = default;

  /**
   * Returns the number of entries in this map.
   * @return size_t
   */
  inline USize size() const noexcept {
    if (this->root != nullptr) {
      return this->root->size;
    } else {
      return 0;
    }
  }

  /**
   * Returns true if this collection contains no elements.
   * @return bool
   */
```

```
134   inline bool empty() const noexcept { return this->root == nullptr; }
135
136   /**
137    * Removes all of the elements from this map.
138    */
139   void clear() noexcept { this->root = nullptr; }
140
141   /**
142    * Returns the value to which the specified key is mapped; If this map
143    * contains no mapping for the key, a {@code NoSuchMappingException} will
144    * be thrown.
145    * @param key
146    * @return SizeBalancedTreeMap<Key, Value>::Value
147    * @throws NoSuchMappingException
148    */
149   Value get(K key) const {
150     if (this->root == nullptr) {
151       throw NoSuchMappingException("Invalid key");
152     } else {
153       NodePtr node = this->getNode(this->root, key);
154       if (node != nullptr) {
155         return node->value;
156       } else {
157         throw NoSuchMappingException("Invalid key");
158       }
159     }
160   }
161
162   /**
163    * Returns the value to which the specified key is mapped; If this map
164    * contains no mapping for the key, a new mapping with a default value
165    * will be inserted.
166    * @param key
167    * @return SizeBalancedTreeMap<Key, Value>::Value &
168    */
169   Value &getOrDefault(K key) {
170     if (this->root == nullptr) {
171       this->root = Node::from(key);
172       return this->root->value;
173     } else {
174       return this
175           ->getNodeOrProvide(this->root, key,
176                              [&key]() { return Node::from(key); })
177           ->value;
178     }
179   }
180
181   /**
182    * Returns true if this map contains a mapping for the specified key.
183    * @param key
184    * @return bool
185    */
```

```
  bool contains(K key) const {
    return this->getNode(this->root, key) != nullptr;
  }

  /**
   * Associates the specified value with the specified key in this map.
   * @param key
   * @param value
   */
  void insert(K key, V value) {
    if (this->root == nullptr) {
      this->root = Node::from(key, value);
    } else {
      this->insert(this->root, key, value);
    }
  }

  /**
   * If the specified key is not already associated with a value, associates
   * it with the given value and returns true, else returns false.
   * @param key
   * @param value
   * @return bool
   */
  bool insertIfAbsent(K key, V value) {
    USize sizeBeforeInsertion = this->size();
    if (this->root == nullptr) {
      this->root = Node::from(key, value);
    } else {
      this->insert(this->root, key, value, false);
    }
    return this->size() > sizeBeforeInsertion;
  }

  /**
   * If the specified key is not already associated with a value, associates
   * it with the given value and returns the value, else returns the
associated
   * value.
   * @param key
   * @param value
   * @return SizeBalancedTreeMap<Key, Value>::Value &
   */
  Value &getOrInsert(K key, V value) {
    if (this->root == nullptr) {
      this->root = Node::from(key, value);
      return root->value;
    } else {
      NodePtr node = getNodeOrProvide(this->root, key,
                                      [&]() { return Node::from(key,
value); });
      return node->value;
```

```
236     }
237   }
238
239   Value operator[](K key) const { return this->get(key); }
240
241   Value &operator[](K key) { return this->getOrDefault(key); }
242
243   /**
244    * Removes the mapping for a key from this map if it is present;
245    * Returns true if the mapping is present else returns false
246    * @param key the key of the mapping
247    * @return bool
248    */
249   bool remove(K key) {
250     if (this->root == nullptr) {
251       return false;
252     } else {
253       return this->remove(this->root, key, [](ConstNodePtr) {});
254     }
255   }
256
257   /**
258    * Removes the mapping for a key from this map if it is present and
    returns
259    * the value which is mapped to the key; If this map contains no mapping
    for
260    * the key, a {@code NoSuchMappingException} will be thrown.
261    * @param key
262    * @return SizeBalancedTreeMap<Key, Value>::Value
263    * @throws NoSuchMappingException
264    */
265   Value getAndRemove(K key) {
266     Value result;
267     NodeConsumer action = [&](ConstNodePtr node) { result = node->value; };
268
269     if (root == nullptr) {
270       throw NoSuchMappingException("Invalid key");
271     } else {
272       if (remove(this->root, key, action)) {
273         return result;
274       } else {
275         throw NoSuchMappingException("Invalid key");
276       }
277     }
278   }
279
280   /**
281    * Gets the entry corresponding to the specified key; if no such entry
282    * exists, returns the entry for the least key greater than the specified
283    * key; if no such entry exists (i.e., the greatest key in the Tree
    is less
```

```
    * than the specified key), a {@code NoSuchMappingException} will be
284 thrown.
    * @param key
    * @return SizeBalancedTreeMap<Key, Value>::Entry
    * @throws NoSuchMappingException
    */
   Entry getCeilingEntry(K key) const {
     if (this->root == nullptr) {
       throw NoSuchMappingException("No ceiling entry in this map");
     }

     NodePtr node = this->root;
     std::stack<NodePtr> ancestors;

     while (node != nullptr) {
       if (key == node->key) {
         return node->entry();
       }

       if (compare(key, node->key)) {
         /* key < node->key */
         if (node->left != nullptr) {
           ancestors.push(node);
           node = node->left;
         } else {
           return node->entry();
         }
       } else {
         /* key > node->key */
         if (node->right != nullptr) {
           ancestors.push(node);
           node = node->right;
         } else {
           if (ancestors.empty()) {
             throw NoSuchMappingException("No ceiling entry in this map");
           }

           NodePtr parent = ancestors.top();
           ancestors.pop();

           while (node == parent->right) {
             node = parent;
             if (!ancestors.empty()) {
               parent = ancestors.top();
               ancestors.pop();
             } else {
               throw NoSuchMappingException("No ceiling entry in this map");
             }
           }

           return parent->entry();
         }
```

```
335       }
336     }
337
338     throw NoSuchMappingException("No ceiling entry in this map");
339   }
340
341   /**
342    * Gets the entry corresponding to the specified key; if no such entry
    exists,
343    * returns the entry for the greatest key less than the specified key;
344    * if no such entry exists, a {@code NoSuchMappingException} will be
    thrown.
345    * @param key
346    * @return SizeBalancedTreeMap<Key, Value>::Entry
347    * @throws NoSuchMappingException
348    */
349   Entry getFloorEntry(K key) const {
350     if (this->root == nullptr) {
351       throw NoSuchMappingException("No floor entry exists in this map");
352     }
353
354     NodePtr node = this->root;
355     std::stack<NodePtr> ancestors;
356
357     while (node != nullptr) {
358       if (key == node->key) {
359         return node->entry();
360       }
361
362       if (compare(key, node->key)) {
363         /* key < node->key */
364         if (node->left != nullptr) {
365           ancestors.push(node);
366           node = node->left;
367         } else {
368           if (ancestors.empty()) {
369             throw NoSuchMappingException("No floor entry exists in this
    map");
370           }
371
372           NodePtr parent = ancestors.top();
373           ancestors.pop();
374
375           while (node == parent->left) {
376             node = parent;
377             if (!ancestors.empty()) {
378               parent = ancestors.top();
379               ancestors.pop();
380             } else {
381               throw NoSuchMappingException("No floor entry exists in
    this map");
382             }
```

```
          }

          return parent->entry();
        }
      } else {
        /* key > node->key */
        if (node->right != nullptr) {
          ancestors.push(node);
          node = node->right;
        } else {
          return node->entry();
        }
      }
    }

    throw NoSuchMappingException("No floor entry exists in this map");
  }

  /**
   * Gets the entry for the least key greater than the specified
   * key; if no such entry exists, returns the entry for the least
   * key greater than the specified key; if no such entry exists,
   * a {@code NoSuchMappingException} will be thrown.
   * @param key
   * @return SizeBalancedTreeMap<Key, Value>::Entry
   * @throws NoSuchMappingException
   */
  Entry getHigherEntry(K key) {
    if (this->root == nullptr) {
      throw NoSuchMappingException("No higher entry exists in this map");
    }

    NodePtr node = this->root;
    std::stack<NodePtr> ancestors;

    while (node != nullptr) {
      if (compare(key, node->key)) {
        /* key < node->key */
        if (node->left != nullptr) {
          ancestors.push(node);
          node = node->left;
        } else {
          return node->entry();
        }
      } else {
        /* key >= node->key */
        if (node->right != nullptr) {
          ancestors.push(node);
          node = node->right;
        } else {
          if (ancestors.empty()) {
```

```
            throw NoSuchMappingException("No higher entry exists in this
map");
          }

          NodePtr parent = ancestors.top();
          ancestors.pop();

          while (node == parent->right) {
            node = parent;
            if (!ancestors.empty()) {
              parent = ancestors.top();
              ancestors.pop();
            } else {
              throw NoSuchMappingException(
                  "No higher entry exists in this map");
            }
          }

          return parent->entry();
        }
      }
    }

    throw NoSuchMappingException("No higher entry exists in this map");
  }

  /**
   * Returns the entry for the greatest key less than the specified key; if
   * no such entry exists (i.e., the least key in the Tree is greater than
   * the specified key), a {@code NoSuchMappingException} will be thrown.
   * @param key
   * @return SizeBalancedTreeMap<Key, Value>::Entry
   * @throws NoSuchMappingException
   */
  Entry getLowerEntry(K key) const {
    if (this->root == nullptr) {
      throw NoSuchMappingException("No lower entry exists in this map");
    }

    NodePtr node = this->root;
    std::stack<NodePtr> ancestors;

    while (node != nullptr) {
      if (compare(key, node->key) || key == node->key) {
        /* key <= node->key */
        if (node->left != nullptr) {
          ancestors.push(node);
          node = node->left;
        } else {
          if (ancestors.empty()) {
            throw NoSuchMappingException("No lower entry exists in this
map");
```

```
484           }
485
486           NodePtr parent = ancestors.top();
487           ancestors.pop();
488
489           while (node == parent->left) {
490             node = parent;
491             if (!ancestors.empty()) {
492               parent = ancestors.top();
493               ancestors.pop();
494             } else {
495               throw NoSuchMappingException("No lower entry exists in
    this map");
496             }
497           }
498
499           return parent->entry();
500         }
501       } else {
502         /* key > node->key */
503         if (node->right != nullptr) {
504           ancestors.push(node);
505           node = node->right;
506         } else {
507           return node->entry();
508         }
509       }
510     }
511
512     throw NoSuchMappingException("No lower entry exists in this map");
513   }
514
515   /**
516    * Count the number of entries that are less than the given key
517    * @param key
518    * @return USize
519    */
520   USize countLessThan(K key) { return this->countLess(this->root, key); }
521
522   /**
523    * Count the number of entries that are less or equal to the given key
524    * @param key
525    * @return USize
526    */
527   USize countLessOrEqualTo(K key) {
528     return this->countLess(this->root, key, true);
529   }
530
531   /**
532    * Count the number of entries that are greater than the given key
533    * @param key
534    * @return USize
```

```
535     */
536    USize countGreaterThan(K key) { return this->countGreater(this->root,
    key); }
537
538    /**
539     * Count the number of entries that are greater or equal to the given key
540     * @param key
541     * @return USize
542     */
543    USize countGreaterOrEqualTo(K key) {
544      return this->countGreater(this->root, key, true);
545    }
546
547    /**
548     * Remove all entries that satisfy the filter condition.
549     * @param filter
550     */
551    void removeAll(KeyValueFilter filter) {
552      std::vector<Key> keys;
553      this->inorderTraversal([&](ConstNodePtr node) {
554        if (filter(node->key, node->value)) {
555          keys.push_back(node->key);
556        }
557      });
558      for (const Key &key : keys) {
559        this->remove(key);
560      }
561    }
562
563    /**
564     * Performs the given action for each key and value entry in this map.
565     * The value is immutable for the action.
566     * @param action
567     */
568    void forEach(KeyValueConsumer action) const {
569      this->inorderTraversal(
570          [&](ConstNodePtr node) { action(node->key, node->value); });
571    }
572
573    /**
574     * Performs the given action for each key and value entry in this map.
575     * The value is mutable for the action.
576     * @param action
577     */
578    void forEachMut(MutKeyValueConsumer action) {
579      this->inorderTraversal(
580          [&](ConstNodePtr node) { action(node->key, node->value); });
581    }
582
583    /**
584     * Returns a list containing all of the entries in this map.
585     * @return SizeBalancedTreeMap<Key, Value>::EntryList
```

```
586    */
587   EntryList toEntryList() const {
588     EntryList entryList;
589     this->inorderTraversal(
590         [&](ConstNodePtr node) { entryList.push_back(node->entry()); });
591     return entryList;
592   }
593
594  private:
595   static USize size(ConstNodePtr node) {
596     return node != nullptr ? node->size : 0;
597   }
598
599   static void rotateLeft(NodePtr &node) {
600     assert(node != nullptr);
601     // clang-format off
602     //     |                       |
603     //     N                       S
604     //    / \     l-rotate(N)     / \
605     //   L   S    ==========>    N   R
606     //      / \                 / \
607     //     M   R               L   M
608     // clang-format on
609     NodePtr successor = node->right;
610     node->right = successor->left;
611     successor->left = node;
612
613     node->updateSize();
614     successor->updateSize();
615
616     node = successor;
617   }
618
619   static void rotateRight(NodePtr &node) {
620     assert(node != nullptr);
621     // clang-format off
622     //       |                   |
623     //       N                   S
624     //      / \   r-rotate(N)   / \
625     //     S   R  ==========>  L   N
626     //    / \                     / \
627     //   L   M                   M   R
628     // clang-format on
629     NodePtr successor = node->left;
630     node->left = successor->right;
631     successor->right = node;
632
633     node->updateSize();
634     successor->updateSize();
635
636     node = successor;
637   }
```

```
638
639   static void swapNode(NodePtr &lhs, NodePtr &rhs) {
640     std::swap(lhs->key, rhs->key);
641     std::swap(lhs->value, rhs->value);
642     std::swap(lhs, rhs);
643   }
644
645   static void fixBalance(NodePtr &node) {
646     if (node == nullptr) {
647       return;
648     }
649
650     if (node->left != nullptr) {
651       if (size(node->left->left) > size(node->right)) {
652         // clang-format off
653         //       |                        |
654         //       N                        L
655         //      / \     r-rotate(N)      / \
656         //     L  <R>   ==========>    [M]  N
657         //    /                            \
658         //  [M]                            <R>
659         // clang-format on
660         rotateRight(node);
661         fixBalance(node->right);
662         fixBalance(node);
663         return;
664       } else if (size(node->left->right) > size(node->right)) {
665         // clang-format off
666         //     |                     |                       |
667         //     N                     N                     [M]
668         //    / \    l-rotate(L)    / \     r-rotate(N)    / \
669         //   L  <R>  ==========>  [M] <R>   ==========>   L   N
670         //    \                   /                          \
671         //    [M]               L                            <R>
672         // clang-format on
673         rotateLeft(node->left);
674         rotateRight(node);
675         fixBalance(node->left);
676         fixBalance(node->right);
677         fixBalance(node);
678         return;
679       }
680     }
681
682     if (node->right != nullptr) {
683       if (size(node->right->right) > size(node->left)) {
684         // clang-format off
685         //     |                       |
686         //     N                       R
687         //    / \     l-rotate(N)     / \
688         //  <L>  R    ==========>    N  [M]
689         //        \                 /
```

```
690         //        [M]             <L>
691         // clang-format on
692         rotateLeft(node);
693         fixBalance(node->left);
694         fixBalance(node);
695         return;
696       } else if (size(node->right->left) > size(node->left)) {
697         // clang-format off
698         //     |                     |                      |
699         //     N                     N                    [M]
700         //    / \    r-rotate(R)    / \     l-rotate(N)    / \
701         //  <L>  R   ==========>  <L> [M]   ==========>   N   R
702         //      /                     \                /
703         //    [M]                      R             <L>
704         // clang-format on
705         rotateRight(node->right);
706         rotateLeft(node);
707         fixBalance(node->left);
708         fixBalance(node->right);
709         fixBalance(node);
710         return;
711       }
712     }
713   }
714
715   NodePtr getNodeOrProvide(NodePtr &node, K key, NodeProvider provide) {
716     assert(node != nullptr);
717
718     if (key == node->key) {
719       return node;
720     }
721
722     assert(key != node->key);
723
724     NodePtr result;
725
726     if (compare(key, node->key)) {
727       /* key < node->key */
728       if (node->left == nullptr) {
729         result = node->left = provide();
730         node->updateSize();
731       } else {
732         result = getNodeOrProvide(node->left, key, provide);
733         node->updateSize();
734         fixBalance(node);
735       }
736     } else {
737       /* key > node->key */
738       if (node->right == nullptr) {
739         result = node->right = provide();
740         node->updateSize();
741       } else {
```

```
742         result = getNodeOrProvide(node->right, key, provide);
743         node->updateSize();
744         fixBalance(node);
745       }
746     }
747
748     return result;
749   }
750
751   NodePtr getNode(ConstNodePtr node, K key) const {
752     assert(node != nullptr);
753
754     if (key == node->key) {
755       return node;
756     }
757
758     if (compare(key, node->key)) {
759       /* key < node->key */
760       return node->left == nullptr ? nullptr : getNode(node->left, key);
761     } else {
762       /* key > node->key */
763       return node->right == nullptr ? nullptr : getNode(node->right, key);
764     }
765   }
766
767   void insert(NodePtr &node, K key, V value, bool replace = true) {
768     assert(node != nullptr);
769
770     if (key == node->key) {
771       if (replace) {
772         node->value = value;
773       }
774       return;
775     }
776
777     assert(key != node->key);
778
779     if (compare(key, node->key)) {
780       /* key < node->key */
781       if (node->left == nullptr) {
782         node->left = Node::from(key, value);
783         node->updateSize();
784       } else {
785         insert(node->left, key, value, replace);
786         node->updateSize();
787         fixBalance(node);
788       }
789     } else {
790       /* key > node->key */
791       if (node->right == nullptr) {
792         node->right = Node::from(key, value);
793         node->updateSize();
```

```
794       } else {
795         insert(node->right, key, value, replace);
796         node->updateSize();
797         fixBalance(node);
798       }
799     }
800   }
801
802   bool remove(NodePtr &node, K key, NodeConsumer action) {
803     assert(node != nullptr);
804
805     if (key != node->key) {
806       if (compare(key, node->key)) {
807         /* key < node->key */
808         NodePtr &left = node->left;
809         if (left != nullptr && remove(left, key, action)) {
810           node->updateSize();
811           fixBalance(node);
812           return true;
813         } else {
814           return false;
815         }
816       } else {
817         /* key > node->key */
818         NodePtr &right = node->right;
819         if (right != nullptr && remove(right, key, action)) {
820           node->updateSize();
821           fixBalance(node);
822           return true;
823         } else {
824           return false;
825         }
826       }
827     }
828
829     assert(key == node->key);
830     action(node);
831
832     if (node->isLeaf()) {
833       // Case 1: no child
834       node = nullptr;
835     } else if (node->right == nullptr) {
836       // clang-format off
837       // Case 2: left child only
838       //     P
839       //     |  remove(N)  P
840       //     N  ========>  |
841       //    /              L
842       //   L
843       // clang-format on
844       node = node->left;
845     } else if (node->left == nullptr) {
```

```
846       // clang-format off
847       // Case 3: right child only
848       //   P
849       //   |    remove(N)  P
850       //   N    ========>  |
851       //    \              R
852       //     R
853       // clang-format on
854       node = node->right;
855     } else if (node->right->left == nullptr) {
856       // clang-format off
857       // Case 4: both left and right child, right child has no left child
858       //    |                 |
859       //    N    remove(N)    R
860       //   / \   ========>   /
861       //  L   R             L
862       // clang-format on
863       NodePtr right = node->right;
864       swapNode(node, right);
865       right->right = node->right;
866       node = right;
867       node->updateSize();
868       fixBalance(node);
869     } else {
870       // clang-format off
871       // Case 5: both left and right child, right child is not a leaf
872       //   Step 1. find the node N with the smallest key
873       //           and its parent P on the right subtree
874       //   Step 2. swap S and N
875       //   Step 3. remove node N like Case 1 or Case 3
876       //   Step 4. update size for all nodes on the path
877       //           from S to P
878       //     |                 |
879       //     N                 S                 |
880       //    / \               / \                S
881       //   L  ..  swap(N, S)  L  ..  remove(N)   / \
882       //       |  =========>     |  ========>  L  ..
883       //       P                 P                |
884       //      / \               / \               P
885       //     S  ..             N  ..             / \
886       //      \                 \               R  ..
887       //       R                 R
888       // clang-format on
889
890       std::stack<NodePtr> path;
891
892       // Step 1
893       NodePtr successor = node->right;
894       NodePtr parent = node;
895       path.push(node);
896
897       while (successor->left != nullptr) {
```

```
898           path.push(successor);
899           parent = successor;
900           successor = parent->left;
901         }
902
903         // Step 2
904         swapNode(node, successor);
905
906         // Step 3
907         parent->left = node->right;
908         // Restore node
909         node = successor;
910
911         // Step 4
912         while (!path.empty()) {
913           path.top()->updateSize();
914           path.pop();
915         }
916       }
917
918       return true;
919     }
920
921     USize countLess(ConstNodePtr node, K key, bool countEqual = false) const {
922       if (node == nullptr) {
923         return 0;
924       } else if (key < node->key) {
925         return countLess(node->left, key, countEqual);
926       } else if (key > node->key) {
927         return size(node->left) + 1 + countLess(node->right, key, countEqual);
928       } else {
929         return size(node->left) + (countEqual ? 1 : 0);
930       }
931     }
932
933     USize countGreater(ConstNodePtr node, K key, bool countEqual = false)
    const {
934       if (node == nullptr) {
935         return 0;
936       } else if (key < node->key) {
937         return size(node->right) + 1 + countGreater(node->left, key,
    countEqual);
938       } else if (key > node->key) {
939         return countGreater(node->right, key, countEqual);
940       } else {
941         return size(node->right) + (countEqual ? 1 : 0);
942       }
943     }
944
945     void inorderTraversal(NodeConsumer action) const {
946       if (this->root == nullptr) {
947         return;
```

```
948     }
949
950     std::stack<NodePtr> stack;
951     NodePtr node = this->root;
952
953     while (node != nullptr || !stack.empty()) {
954       while (node != nullptr) {
955         stack.push(node);
956         node = node->left;
957       }
958       if (!stack.empty()) {
959         node = stack.top();
960         stack.pop();
961         action(node);
962         node = node->right;
963       }
964     }
965   }
966 };
967
968 #endif  // SIZE_BALANCED_TREE_MAP_HPP
969
```

10.15.8 AVL 树

AVL 树，是一种平衡的二叉搜索树．由于各种算法教材上对 AVL 的介绍十分冗长，造成了很多人对 AVL 树复杂、不实用的印象．但实际上，AVL 树的原理简单，实现也并不复杂．

性质

1. 空二叉树是一个 AVL 树
2. 如果 T 是一棵 AVL 树，那么其左右子树也是 AVL 树，并且 $|h(ls) - h(rs)| \leq 1$，h 是其左右子树的高度
3. 树高为 $O(\log n)$

平衡因子：右子树高度 - 左子树高度

树高的证明

设 f_n 为高度为 n 的 AVL 树所包含的最少节点数，则有

$$f_n = \begin{cases} 1 & (n = 1) \\ 2 & (n = 2) \\ f_{n-1} + f_{n-2} + 1 & (n > 2) \end{cases}$$

根据常系数非齐次线性差分方程的解法，$\{f_n + 1\}$ 是一个斐波那契数列．这里 f_n 的通项为：

$$f_n = \frac{5 + 2\sqrt{5}}{5}\left(\frac{1 + \sqrt{5}}{2}\right)^n + \frac{5 - 2\sqrt{5}}{5}\left(\frac{1 - \sqrt{5}}{2}\right)^n - 1$$

斐波那契数列以指数的速度增长，对于树高 n 有：

$$n < \log_{\frac{1+\sqrt{5}}{2}}(f_n + 1) < \frac{3}{2}\log_2(f_n + 1)$$

因此 AVL 树的高度为 $O(\log f_n)$，这里的 f_n 为结点数．

过程

插入结点

与 BST（二叉搜索树）中类似，先进行一次失败的查找来确定插入的位置，插入节点后根据平衡因子来决定是否需要调整.

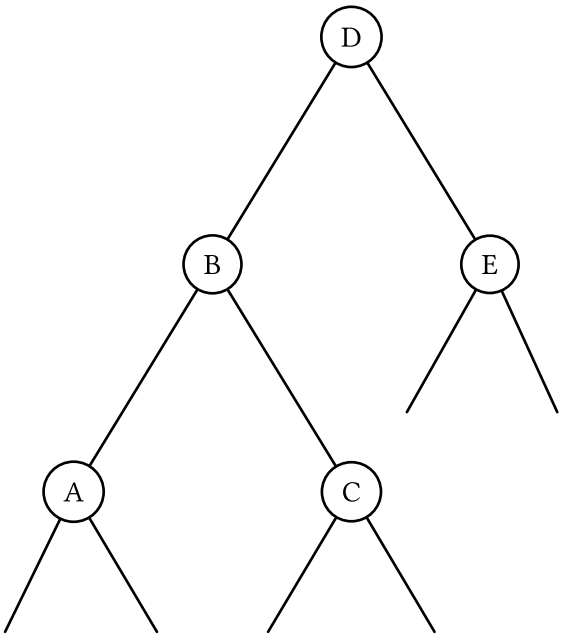
删除结点

删除和 BST 类似，将结点与后继交换后再删除.

删除会导致树高以及平衡因子变化，这时需要沿着被删除结点到根的路径来调整这种变化.

平衡的维护

插入或删除节点后，可能会造成 AVL 树的性质 2 被破坏．因此，需要沿着从被插入/删除的节点到根的路径对树进行维护．如果对于某一个节点，性质 2 不再满足，由于我们只插入/删除了一个节点，对树高的影响不超过 1，因此该节点的平衡因子的绝对值至多为 2．由于对称性，我们在此只讨论左子树的高度比右子树大 2 的情况，即下图中 $h(B)-h(E)=2$．此时，还需要根据 $h(A)$ 和 $h(C)$ 的大小关系分两种情况讨论．需要注意的是，由于我们是自底向上维护平衡的，因此对节点 D 的所有后代来说，性质 2 仍然是被满足的.

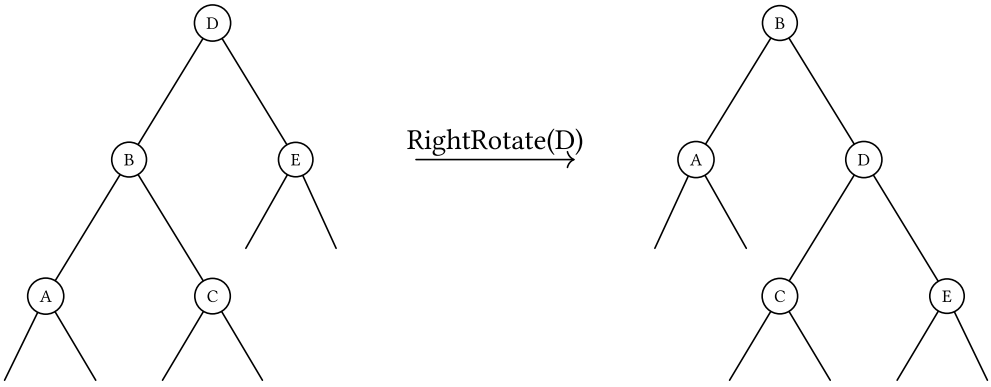


情况一：A 点树高不小于 C 点树高

设 $h(E)=x$，则有

$$\begin{cases} h(B) = x + 2 \\ h(A) = x + 1 \\ x \le h(C) \le x + 1 \end{cases}$$

其中 $h(C) \ge x$ 是由于节点 B 满足性质 2，因此 $h(C)$ 和 $h(A)$ 的差不会超过 1. 此时我们对节点 D 进行一次右旋操作（旋转操作与其它类型的平衡二叉搜索树相同），如下图所示.



显然节点 A、C、E 的高度不发生变化，并且有

$$\begin{cases} 0 \le h(C) - h(E) \le 1 \\ x + 1 \le h'(D) = \max(h(C), h(E)) + 1 = h(C) + 1 \le x + 2 \\ 0 \le h'(D) - h(A) \le 1 \end{cases}$$

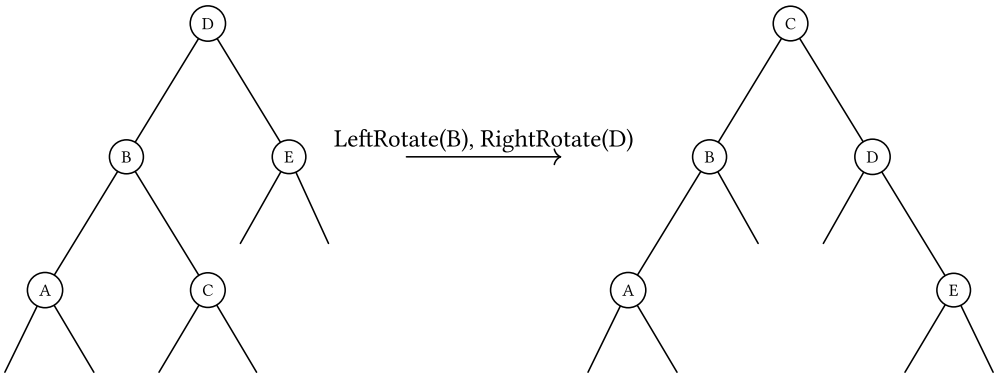
因此旋转后的节点 B 和 D 也满足性质 2.

情况二：A 点树高小于 C 点树高

设 $h(E) = x$，则与刚才同理，有

$$\begin{cases} h(B) = x + 2 \\ h(C) = x + 1 \\ h(A) = x \end{cases}$$

此时我们先对节点 B 进行一次左旋操作，再对节点 D 进行一次右旋操作，如下图所示.



显然节点 A、E 的高度不发生变化，并且 B 的新右儿子和 D 的新左儿子分别为 C 原来的左右儿子，则有

$$\begin{cases} x-1 \le h'(rs_B), h'(ls_D) \le x \\ 0 \le h(A) - h'(rs_B) \le 1 \\ 0 \le h(E) - h'(ls_D) \le 1 \\ h'(B) = \max(h(A), h'(rs_B)) + 1 = x+1 \\ h'(D) = \max(h(E), h'(ls_D)) + 1 = x+1 \\ h'(B) - h'(D) = 0 \end{cases}$$

因此旋转后的节点 B、C、D 也满足性质 2.

维护平衡操作：伪代码

```
1  function MaintainBalance(p)
2      l ← ls_p, r ← rs_p
3      if h(l) − h(r) = 2
4          if h(ls_l) ≥ h(rs_l)
5              RightRotate(p)
6          else
7              LeftRotate(l)
8              RightRotate(p)
9      else if h(l) − h(r) = −2
10         if h(ls_r) ≤ h(rs_r)
11             LeftRotate(p)
12         else
13             RightRotate(r)
14             LeftRotate(p)
```

与其他平衡二叉搜索树相同，AVL 树中节点的高度、子树大小等信息需要在旋转时进行维护.

其他操作

AVL 树的其他操作（Predecessor、Successor、Select、Rank 等）与普通的二叉搜索树相同.

参考代码

下面的代码是用 AVL 树实现的 `Map`，即有序不可重映射：

参考代码

```cpp
/**
 * @brief An AVLTree-based map implementation
 * @details The map is sorted according to the natural ordering of its
 *  keys or by a {@code Compare} function provided; This implementation
 *  provides guaranteed log(n) time cost for the contains, get, insert
 *  and remove operations.
 */

#ifndef AVLTREE_MAP_HPP
#define AVLTREE_MAP_HPP

#include <cassert>
#include <cstddef>
#include <cstdint>
#include <functional>
#include <memory>
#include <stack>
#include <utility>
#include <vector>

/**
 * An AVLTree-based map implementation
 * https://en.wikipedia.org/wiki/AVL_tree
 * @tparam Key the type of keys maintained by this map
 * @tparam Value the type of mapped values
 * @tparam Compare
 */
template <typename Key, typename Value, typename Compare = std::less<Key> >
class AvlTreeMap {
 private:
  using USize = size_t;
  using Factor = int64_t;

  Compare compare = Compare();

 public:
  struct Entry {
    Key key;
    Value value;

    bool operator==(const Entry &rhs) const noexcept {
      return this->key == rhs.key && this->value == rhs.value;
    }

    bool operator!=(const Entry &rhs) const noexcept {
      return this->key != rhs.key || this->value != rhs.value;
    }
  };

 private:
  struct Node {
    using Ptr = std::shared_ptr<Node>;
```

```
53      using Provider = const std::function<Ptr(void)> &;
54      using Consumer = const std::function<void(const Ptr &)> &;
55
56      Key key;
57      Value value{};
58
59      Ptr left = nullptr;
60      Ptr right = nullptr;
61
62      USize height = 1;
63
64      explicit Node(Key k) : key(std::move(k)) {}
65
66      explicit Node(Key k, Value v) : key(std::move(k)), value(std::move(v))
    {}
67
68      ~Node() = default;
69
70      inline bool isLeaf() const noexcept {
71        return this->left == nullptr && this->right == nullptr;
72      }
73
74      inline void updateHeight() noexcept {
75        if (this->isLeaf()) {
76          this->height = 1;
77        } else if (this->left == nullptr) {
78          this->height = this->right->height + 1;
79        } else if (this->right == nullptr) {
80          this->height = this->left->height + 1;
81        } else {
82          this->height = std::max(left->height, right->height) + 1;
83        }
84      }
85
86      inline Factor factor() const noexcept {
87        if (this->isLeaf()) {
88          return 0;
89        } else if (this->left == nullptr) {
90          return (Factor)this->right->height;
91        } else if (this->right == nullptr) {
92          return (Factor) - this->left->height;
93        } else {
94          return (Factor)(this->right->height - this->left->height);
95        }
96      }
97
98      inline Entry entry() const { return Entry{key, value}; }
99
100     static Ptr from(const Key &k) { return
    std::make_shared<Node>(Node(k)); }
101
102     static Ptr from(const Key &k, const Value &v) {
```

```
103         return std::make_shared<Node>(Node(k, v));
104       }
105     };
106
107     using NodePtr = typename Node::Ptr;
108     using ConstNodePtr = const NodePtr &;
109     using NodeProvider = typename Node::Provider;
110     using NodeConsumer = typename Node::Consumer;
111
112     NodePtr root = nullptr;
113     USize count = 0;
114
115     using K = const Key &;
116     using V = const Value &;
117
118   public:
119     using EntryList = std::vector<Entry>;
120     using KeyValueConsumer = const std::function<void(K, V)> &;
121     using MutKeyValueConsumer = const std::function<void(K, Value &)> &;
122     using KeyValueFilter = const std::function<bool(K, V)> &;
123
124     class NoSuchMappingException : protected std::exception {
125      private:
126       const char *message;
127
128      public:
129       explicit NoSuchMappingException(const char *msg) : message(msg) {}
130
131       const char *what() const noexcept override { return message; }
132     };
133
134     AvlTreeMap() noexcept = default;
135
136     /**
137      * Returns the number of entries in this map.
138      * @return size_t
139      */
140     inline USize size() const noexcept { return this->count; }
141
142     /**
143      * Returns true if this collection contains no elements.
144      * @return bool
145      */
146     inline bool empty() const noexcept { return this->count == 0; }
147
148     /**
149      * Removes all of the elements from this map.
150      */
151     void clear() noexcept {
152       this->root = nullptr;
153       this->count = 0;
154     }
```

```
155
156   /**
157    * Returns the value to which the specified key is mapped; If this map
158    * contains no mapping for the key, a {@code NoSuchMappingException} will
159    * be thrown.
160    * @param key
161    * @return AvlTreeMap<Key, Value>::Value
162    * @throws NoSuchMappingException
163    */
164   Value get(K key) const {
165     if (this->root == nullptr) {
166       throw NoSuchMappingException("Invalid key");
167     } else {
168       NodePtr node = this->getNode(this->root, key);
169       if (node != nullptr) {
170         return node->value;
171       } else {
172         throw NoSuchMappingException("Invalid key");
173       }
174     }
175   }
176
177   /**
178    * Returns the value to which the specified key is mapped; If this map
179    * contains no mapping for the key, a new mapping with a default value
180    * will be inserted.
181    * @param key
182    * @return AvlTreeMap<Key, Value>::Value &
183    */
184   Value &getOrDefault(K key) {
185     if (this->root == nullptr) {
186       this->root = Node::from(key);
187       this->count += 1;
188       return this->root->value;
189     } else {
190       return this
191           ->getNodeOrProvide(this->root, key,
192                              [&key]() { return Node::from(key); })
193           ->value;
194     }
195   }
196
197   /**
198    * Returns true if this map contains a mapping for the specified key.
199    * @param key
200    * @return bool
201    */
202   bool contains(K key) const {
203     return this->getNode(this->root, key) != nullptr;
204   }
205
206   /**
```

```
207     * Associates the specified value with the specified key in this map.
208     * @param key
209     * @param value
210     */
211    void insert(K key, V value) {
212      if (this->root == nullptr) {
213        this->root = Node::from(key, value);
214        this->count += 1;
215      } else {
216        this->insert(this->root, key, value);
217      }
218    }
219
220    /**
221     * If the specified key is not already associated with a value, associates
222     * it with the given value and returns true, else returns false.
223     * @param key
224     * @param value
225     * @return bool
226     */
227    bool insertIfAbsent(K key, V value) {
228      USize sizeBeforeInsertion = this->size();
229      if (this->root == nullptr) {
230        this->root = Node::from(key, value);
231        this->count += 1;
232      } else {
233        this->insert(this->root, key, value, false);
234      }
235      return this->size() > sizeBeforeInsertion;
236    }
237
238    /**
239     * If the specified key is not already associated with a value, associates
240     * it with the given value and returns the value, else returns the
    associated
241     * value.
242     * @param key
243     * @param value
244     * @return
245     */
246    Value &getOrInsert(K key, V value) {
247      if (this->root == nullptr) {
248        this->root = Node::from(key, value);
249        this->count += 1;
250        return root->value;
251      } else {
252        NodePtr node = getNodeOrProvide(this->root, key,
                                           [&]() { return Node::from(key,
253     value); });
254        return node->value;
255      }
256    }
```

```
257
258   Value operator[](K key) const { return this->get(key); }
259
260   Value &operator[](K key) { return this->getOrDefault(key); }
261
262   /**
263    * Removes the mapping for a key from this map if it is present;
264    * Returns true if the mapping is present else returns false
265    * @param key the key of the mapping
266    * @return bool
267    */
268   bool remove(K key) {
269     if (this->root == nullptr) {
270       return false;
271     } else {
272       return this->remove(this->root, key, [](ConstNodePtr) {});
273     }
274   }
275
276   /**
277    * Removes the mapping for a key from this map if it is present and
    returns
278    * the value which is mapped to the key; If this map contains no mapping
    for
279    * the key, a {@code NoSuchMappingException} will be thrown.
280    * @param key
281    * @return AvlTreeMap<Key, Value>::Value
282    * @throws NoSuchMappingException
283    */
284   Value getAndRemove(K key) {
285     Value result;
286     NodeConsumer action = [&](ConstNodePtr node) { result = node->value; };
287
288     if (root == nullptr) {
289       throw NoSuchMappingException("Invalid key");
290     } else {
291       if (remove(this->root, key, action)) {
292         return result;
293       } else {
294         throw NoSuchMappingException("Invalid key");
295       }
296     }
297   }
298
299   /**
300    * Gets the entry corresponding to the specified key; if no such entry
301    * exists, returns the entry for the least key greater than the specified
302    * key; if no such entry exists (i.e., the greatest key in the Tree
    is less
303    * than the specified key), a {@code NoSuchMappingException} will be
    thrown.
304    * @param key
```

```
305    * @return AvlTreeMap<Key, Value>::Entry
306    * @throws NoSuchMappingException
307    */
308   Entry getCeilingEntry(K key) const {
309     if (this->root == nullptr) {
310       throw NoSuchMappingException("No ceiling entry in this map");
311     }
312
313     NodePtr node = this->root;
314     std::stack<NodePtr> ancestors;
315
316     while (node != nullptr) {
317       if (key == node->key) {
318         return node->entry();
319       }
320
321       if (compare(key, node->key)) {
322         /* key < node->key */
323         if (node->left != nullptr) {
324           ancestors.push(node);
325           node = node->left;
326         } else {
327           return node->entry();
328         }
329       } else {
330         /* key > node->key */
331         if (node->right != nullptr) {
332           ancestors.push(node);
333           node = node->right;
334         } else {
335           if (ancestors.empty()) {
336             throw NoSuchMappingException("No ceiling entry in this map");
337           }
338
339           NodePtr parent = ancestors.top();
340           ancestors.pop();
341
342           while (node == parent->right) {
343             node = parent;
344             if (!ancestors.empty()) {
345               parent = ancestors.top();
346               ancestors.pop();
347             } else {
348               throw NoSuchMappingException("No ceiling entry in this map");
349             }
350           }
351
352           return parent->entry();
353         }
354       }
355     }
356
```

```
357     throw NoSuchMappingException("No ceiling entry in this map");
358   }
359
360   /**
361    * Gets the entry corresponding to the specified key; if no such entry
exists,
362    * returns the entry for the greatest key less than the specified key;
363    * if no such entry exists, a {@code NoSuchMappingException} will be
thrown.
364    * @param key
365    * @return AvlTreeMap<Key, Value>::Entry
366    * @throws NoSuchMappingException
367    */
368   Entry getFloorEntry(K key) const {
369     if (this->root == nullptr) {
370       throw NoSuchMappingException("No floor entry exists in this map");
371     }
372
373     NodePtr node = this->root;
374     std::stack<NodePtr> ancestors;
375
376     while (node != nullptr) {
377       if (key == node->key) {
378         return node->entry();
379       }
380
381       if (compare(key, node->key)) {
382         /* key < node->key */
383         if (node->left != nullptr) {
384           ancestors.push(node);
385           node = node->left;
386         } else {
387           if (ancestors.empty()) {
388             throw NoSuchMappingException("No floor entry exists in this
map");
389           }
390
391           NodePtr parent = ancestors.top();
392           ancestors.pop();
393
394           while (node == parent->left) {
395             node = parent;
396             if (!ancestors.empty()) {
397               parent = ancestors.top();
398               ancestors.pop();
399             } else {
400               throw NoSuchMappingException("No floor entry exists in
this map");
401             }
402           }
403
404           return parent->entry();
```

```
405         }
406       } else {
407         /* key > node->key */
408         if (node->right != nullptr) {
409           ancestors.push(node);
410           node = node->right;
411         } else {
412           return node->entry();
413         }
414       }
415     }
416
417     throw NoSuchMappingException("No floor entry exists in this map");
418   }
419
420   /**
421    * Gets the entry for the least key greater than the specified
422    * key; if no such entry exists, returns the entry for the least
423    * key greater than the specified key; if no such entry exists,
424    * a {@code NoSuchMappingException} will be thrown.
425    * @param key
426    * @return AvlTreeMap<Key, Value>::Entry
427    * @throws NoSuchMappingException
428    */
429   Entry getHigherEntry(K key) {
430     if (this->root == nullptr) {
431       throw NoSuchMappingException("No higher entry exists in this map");
432     }
433
434     NodePtr node = this->root;
435     std::stack<NodePtr> ancestors;
436
437     while (node != nullptr) {
438       if (compare(key, node->key)) {
439         /* key < node->key */
440         if (node->left != nullptr) {
441           ancestors.push(node);
442           node = node->left;
443         } else {
444           return node->entry();
445         }
446       } else {
447         /* key >= node->key */
448         if (node->right != nullptr) {
449           ancestors.push(node);
450           node = node->right;
451         } else {
452           if (ancestors.empty()) {
453             throw NoSuchMappingException("No higher entry exists in this
    map");
454           }
455
```

```
456           NodePtr parent = ancestors.top();
457           ancestors.pop();
458
459           while (node == parent->right) {
460             node = parent;
461             if (!ancestors.empty()) {
462               parent = ancestors.top();
463               ancestors.pop();
464             } else {
465               throw NoSuchMappingException(
466                   "No higher entry exists in this map");
467             }
468           }
469
470           return parent->entry();
471         }
472       }
473     }
474
475     throw NoSuchMappingException("No higher entry exists in this map");
476   }
477
478   /**
479    * Returns the entry for the greatest key less than the specified key; if
480    * no such entry exists (i.e., the least key in the Tree is greater than
481    * the specified key), a {@code NoSuchMappingException} will be thrown.
482    * @param key
483    * @return AvlTreeMap<Key, Value>::Entry
484    * @throws NoSuchMappingException
485    */
486   Entry getLowerEntry(K key) const {
487     if (this->root == nullptr) {
488       throw NoSuchMappingException("No lower entry exists in this map");
489     }
490
491     NodePtr node = this->root;
492     std::stack<NodePtr> ancestors;
493
494     while (node != nullptr) {
495       if (compare(key, node->key) || key == node->key) {
496         /* key <= node->key */
497         if (node->left != nullptr) {
498           ancestors.push(node);
499           node = node->left;
500         } else {
501           if (ancestors.empty()) {
502             throw NoSuchMappingException("No lower entry exists in this
    map");
503           }
504
505           NodePtr parent = ancestors.top();
506           ancestors.pop();
```

```
507
508           while (node == parent->left) {
509             node = parent;
510             if (!ancestors.empty()) {
511               parent = ancestors.top();
512               ancestors.pop();
513             } else {
514               throw NoSuchMappingException("No lower entry exists in
    this map");
515             }
516           }
517
518           return parent->entry();
519         }
520       } else {
521         /* key > node->key */
522         if (node->right != nullptr) {
523           ancestors.push(node);
524           node = node->right;
525         } else {
526           return node->entry();
527         }
528       }
529     }
530
531     throw NoSuchMappingException("No lower entry exists in this map");
532   }
533
534   /**
535    * Remove all entries that satisfy the filter condition.
536    * @param filter
537    */
538   void removeAll(KeyValueFilter filter) {
539     std::vector<Key> keys;
540     this->inorderTraversal([&](ConstNodePtr node) {
541       if (filter(node->key, node->value)) {
542         keys.push_back(node->key);
543       }
544     });
545     for (const Key &key : keys) {
546       this->remove(key);
547     }
548   }
549
550   /**
551    * Performs the given action for each key and value entry in this map.
552    * The value is immutable for the action.
553    * @param action
554    */
555   void forEach(KeyValueConsumer action) const {
556     this->inorderTraversal(
557         [&](ConstNodePtr node) { action(node->key, node->value); });
```

```
  }

  /**
   * Performs the given action for each key and value entry in this map.
   * The value is mutable for the action.
   * @param action
   */
  void forEachMut(MutKeyValueConsumer action) {
    this->inorderTraversal(
        [&](ConstNodePtr node) { action(node->key, node->value); });
  }

  /**
   * Returns a list containing all of the entries in this map.
   * @return AvlTreeMap<Key, Value>::EntryList
   */
  EntryList toEntryList() const {
    EntryList entryList;
    this->inorderTraversal(
        [&](ConstNodePtr node) { entryList.push_back(node->entry()); });
    return entryList;
  }

private:
  static NodePtr rotateLeft(ConstNodePtr node) {
    // clang-format off
    //     |                       |
    //     N                       S
    //    / \     l-rotate(N)     / \
    //   L   S    ==========>    N   R
    //      / \                 / \
    //     M   R               L   M
    NodePtr successor = node->right;
    // clang-format on
    node->right = successor->left;
    successor->left = node;

    node->updateHeight();
    successor->updateHeight();

    return successor;
  }

  static NodePtr rotateRight(ConstNodePtr node) {
    // clang-format off
    //       |                   |
    //       N                   S
    //      / \   r-rotate(N)   / \
    //     S   R  ==========>  L   N
    //    / \                     / \
    //   L   M                   M   R
    NodePtr successor = node->left;
```

```
610     // clang-format on
611     node->left = successor->right;
612     successor->right = node;
613
614     node->updateHeight();
615     successor->updateHeight();
616
617     return successor;
618   }
619
620   static void swapNode(NodePtr &lhs, NodePtr &rhs) {
621     std::swap(lhs->key, rhs->key);
622     std::swap(lhs->value, rhs->value);
623     std::swap(lhs, rhs);
624   }
625
626   static void fixBalance(NodePtr &node) {
627     if (node->factor() < -1) {
628       if (node->left->factor() < 0) {
629         // clang-format off
630         //  Left-Left Case
631         //       |
632         //       C                 |
633         //      /   r-rotate(C)    B
634         //     B    ==========>   / \
635         //    /                 A   C
636         //   A
637         // clang-format on
638         node = rotateRight(node);
639       } else {
640         // clang-format off
641         //  Left-Right Case
642         //     |                   |
643         //     C                   C                 |
644         //    /   l-rotate(A)     /   r-rotate(C)    B
645         //   A    ==========>    B    ==========>   / \
646         //    \                 /                 A   C
647         //     B               A
648         // clang-format on
649         node->left = rotateLeft(node->left);
650         node = rotateRight(node);
651       }
652     } else if (node->factor() > 1) {
653       if (node->right->factor() > 0) {
654         // clang-format off
655         //  Right-Right Case
656         //   |
657         //   C                   |
658         //    \     l-rotate(C)    B
659         //     B    ==========>   / \
660         //      \                A   C
661         //       A
```

```
662         // clang-format on
663         node = rotateLeft(node);
664       } else {
665         // clang-format off
666         //  Right-Left Case
667         //   |                |
668         //   A                A                    |
669         //    \   r-rotate(C)  \     l-rotate(A)    B
670         //     C  ==========>    B    ==========>   / \
671         //    /                   \                A   C
672         //   B                     C
673         // clang-format on
674         node->right = rotateRight(node->right);
675         node = rotateLeft(node);
676       }
677     }
678   }
679
680   NodePtr getNodeOrProvide(NodePtr &node, K key, NodeProvider provide) {
681     assert(node != nullptr);
682
683     if (key == node->key) {
684       return node;
685     }
686
687     assert(key != node->key);
688
689     NodePtr result;
690
691     if (compare(key, node->key)) {
692       /* key < node->key */
693       if (node->left == nullptr) {
694         result = node->left = provide();
695         this->count += 1;
696         node->updateHeight();
697       } else {
698         result = getNodeOrProvide(node->left, key, provide);
699         node->updateHeight();
700         fixBalance(node);
701       }
702     } else {
703       /* key > node->key */
704       if (node->right == nullptr) {
705         result = node->right = provide();
706         this->count += 1;
707         node->updateHeight();
708       } else {
709         result = getNodeOrProvide(node->right, key, provide);
710         node->updateHeight();
711         fixBalance(node);
712       }
713     }
```

```
714
715     return result;
716   }
717
718   NodePtr getNode(ConstNodePtr node, K key) const {
719     assert(node != nullptr);
720
721     if (key == node->key) {
722       return node;
723     }
724
725     if (compare(key, node->key)) {
726       /* key < node->key */
727       return node->left == nullptr ? nullptr : getNode(node->left, key);
728     } else {
729       /* key > node->key */
730       return node->right == nullptr ? nullptr : getNode(node->right, key);
731     }
732   }
733
734   void insert(NodePtr &node, K key, V value, bool replace = true) {
735     assert(node != nullptr);
736
737     if (key == node->key) {
738       if (replace) {
739         node->value = value;
740       }
741       return;
742     }
743
744     assert(key != node->key);
745
746     if (compare(key, node->key)) {
747       /* key < node->key */
748       if (node->left == nullptr) {
749         node->left = Node::from(key, value);
750         this->count += 1;
751         node->updateHeight();
752       } else {
753         insert(node->left, key, value, replace);
754         node->updateHeight();
755         fixBalance(node);
756       }
757     } else {
758       /* key > node->key */
759       if (node->right == nullptr) {
760         node->right = Node::from(key, value);
761         this->count += 1;
762         node->updateHeight();
763       } else {
764         insert(node->right, key, value, replace);
765         node->updateHeight();
```

```
766         fixBalance(node);
767       }
768     }
769   }
770
771   bool remove(NodePtr &node, K key, NodeConsumer action) {
772     assert(node != nullptr);
773
774     if (key != node->key) {
775       if (compare(key, node->key)) {
776         /* key < node->key */
777         NodePtr &left = node->left;
778         if (left != nullptr && remove(left, key, action)) {
779           node->updateHeight();
780           fixBalance(node);
781           return true;
782         } else {
783           return false;
784         }
785       } else {
786         /* key > node->key */
787         NodePtr &right = node->right;
788         if (right != nullptr && remove(right, key, action)) {
789           node->updateHeight();
790           fixBalance(node);
791           return true;
792         } else {
793           return false;
794         }
795       }
796     }
797
798     assert(key == node->key);
799     action(node);
800
801     if (node->isLeaf()) {
802       // Case 1: no child
803       node = nullptr;
804     } else if (node->right == nullptr) {
805       // clang-format off
806       // Case 2: left child only
807       //     P
808       //     |  remove(N)  P
809       //     N  ========>  |
810       //    /              L
811       //   L
812       // clang-format on
813       node = node->left;
814       node->updateHeight();
815     } else if (node->left == nullptr) {
816       // clang-format off
817       // Case 3: right child only
```

```
818       //   P
819       //   |    remove(N)  P
820       //   N    ========>  |
821       //    \              R
822       //     R
823       // clang-format on
824       node = node->right;
825       node->updateHeight();
826     } else if (node->right->left == nullptr) {
827       // clang-format off
828       // Case 4: both left and right child, right child has no left child
829       //    |                 |
830       //    N    remove(N)    R
831       //   / \   ========>   /
832       //  L   R             L
833       // clang-format on
834       NodePtr right = node->right;
835       swapNode(node, right);
836       right->right = node->right;
837       node = right;
838       node->updateHeight();
839       fixBalance(node);
840     } else {
841       // clang-format off
842       // Case 5: both left and right child, right child is not a leaf
843       //   Step 1. find the node N with the smallest key
844       //           and its parent P on the right subtree
845       //   Step 2. swap S and N
846       //   Step 3. remove node N like Case 1 or Case 3
847       //   Step 4. update height for P
848       //     |                 |
849       //     N                 S                 |
850       //    / \               / \                S
851       //   L  ..  swap(N, S)  L  ..  remove(N)   / \
852       //       |  =========>     |  ========>  L  ..
853       //       P                 P                |
854       //      / \               / \               P
855       //     S  ..             N  ..             / \
856       //      \                 \               R  ..
857       //       R                 R
858       // clang-format on
859
860       // Step 1
861       NodePtr successor = node->right;
862       NodePtr parent = node;
863       while (successor->left != nullptr) {
864         parent = successor;
865         successor = parent->left;
866       }
867       // Step 2
868       swapNode(node, successor);
869       // Step 3
```

```
870       parent->left = node->right;
871       // Restore node
872       node = successor;
873       // Step 4
874       parent->updateHeight();
875     }
876
877     this->count -= 1;
878     return true;
879   }
880
881   void inorderTraversal(NodeConsumer action) const {
882     if (this->root == nullptr) {
883       return;
884     }
885
886     std::stack<NodePtr> stack;
887     NodePtr node = this->root;
888
889     while (node != nullptr || !stack.empty()) {
890       while (node != nullptr) {
891         stack.push(node);
892         node = node->left;
893       }
894       if (!stack.empty()) {
895         node = stack.top();
896         stack.pop();
897         action(node);
898         node = node->right;
899       }
900     }
901   }
902 };
903
904 #endif  // AVLTREE_MAP_HPP
905
```

其他资料

在 AVL Tree Visualization[1] 可以观察 AVL 树维护平衡的过程.

维基百科 -- AVL 树[2]

外部参考资料

[1] AVL Tree Visualization 

[2] 维基百科 -- AVL 树 

10.15.9 红黑树

> **Authors:** 0x03A6, abc1763613206, auuuu4, CCXXXI, Conless, Enter-tainer, fanenr, happyZYM, hsfzLZH1, iamtwz, LeverImmy, leverimmy, Lhcfl, Marcythm, RIvance, Tiphereth-A, trudbot, Xeniume, Xeonacid, YBYCS, yuhuoji

红黑树是一种自平衡的二叉搜索树. 每个节点额外存储了一个 color 字段 ("RED" or "BLACK"), 用于确保树在插入和删除时保持平衡.

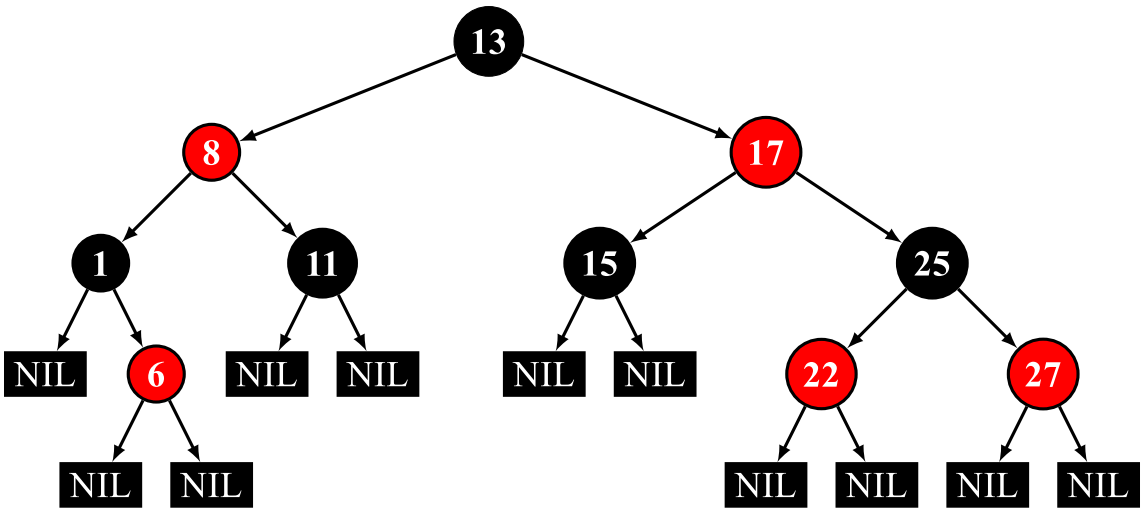
红黑树是 4 阶 B 树（2-3-4 树[1]）的变体. [16]

性质

一棵合法的红黑树必须遵循以下四条性质:

1. 节点为红色或黑色
2. NIL 节点（空叶子节点）为黑色
3. 红色节点的子节点为黑色
4. 从根节点到 NIL 节点的每条路径上的黑色节点数量相同

下图为一棵合法的红黑树:



Note

部分资料中还加入了第五条性质, 即根节点必须为黑色, 这条性质要求完成插入操作后若根节点为红色则将其染黑, 但由于将根节点染黑的操作也可以延迟至删除操作时进行, 因此, 该条性质并非必须满足（本文给出的代码实现中满足该性质）. 为严谨起见, 这里同时引用 维基百科原文[3] 进行说明:

> Some authors, e.g. Cormen & al.,[17]claim "the root is black" as fifth requirement; but not Mehlhorn & Sanders[18]or Sedgewick & Wayne.[19]Since the root can always be changed from red to black, this rule has little effect on analysis. This article also omits it, because it slightly disturbs the recursive algorithms and proofs.

红黑树类的定义

```
1 /**
2  * An RBTree-based set implementation
3  *
```

[16]L. J. Guibas and R. Sedgewick, "A dichromatic framework for balanced trees,"*19th Annual Symposium on Foundations of Computer Science (sfcs 1978)*, Ann Arbor, MI, USA, 1978, pp. 8-21, doi:10.1109/SFCS.1978.3[2].

[17]https://en.wikipedia.org/wiki/Red–black_tree#cite_note-Cormen2009-18[4]

[18]https://en.wikipedia.org/wiki/Red–black_tree#cite_note-Mehlhorn2008-17[5]

[19]https://en.wikipedia.org/wiki/Red–black_tree#cite_note-Algs4-16[6]: 432–447

```
 * @tparam key_t key type
 * @tparam compare_t compare function
 */
template <typename key_t, typename compare_t = std::less<key_t>>
struct rb_tree {
  /**
   * Tree node
   */
  struct node_t {
    node_t *fa;     // == nullptr if root of the tree, otherwise parent
    node_t *ch[2];  // == nullptr if child is empty
                    // ch[0]: left child, ch[1]: right child
    key_t data;
    size_t sz;  // Size of subtree
    bool red;   // == true if node is red, otherwise black

    /**
     * Get child direction of current non-null node
     *
     * @return true if this node is right child of its parent, otherwise false
     */
    auto child_dir() const -> bool { return this == fa->ch[1]; }
  };

  using pointer = node_t *;
  using const_pointer = const node_t *;
  using pointer_const = node_t *const;

  const compare_t compare;
  pointer root;

  rb_tree() : compare{}, root{nullptr} {}

  // ...
};

```

Note

在红黑树节点的存储中，用数组来存储子节点指针可以提高代码复用率.

操作

Note

红黑树的插入/删除有多种实现方式，本文采用《算法导论》的实现方式，将插入后的平衡维护分为 3 种情况，删除后的平衡维护分为 4 种情况.

红黑树的遍历、查找最小/最大值、搜索元素、求元素的排名、根据排名反查元素、查找前驱/后继等操作和 二叉搜索树→10.15.1@p2213 一致，此处不再赘述.

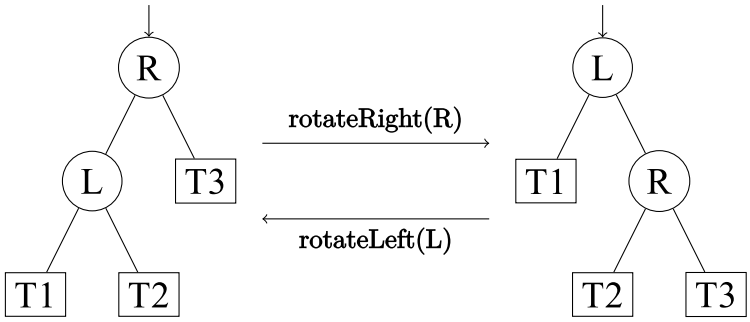
另外，在下文插入/删除平衡维护的代码注释中，我们作如下约定：

- 用 `p` 表示节点 `p` 为黑色；
- 用 `[p]` 表示节点 `p` 为红色；

- 用 `{p}` 表示节点 `p` 为红色或黑色；
- 用 `|p|` 表示节点 `p` 为 NIL 节点或颜色为黑色.

旋转

旋转操作是多数平衡树能够维持平衡的关键，它能在不改变一棵合法 BST 中序遍历结果的情况下改变局部节点的深度.



实现

```
/**
 * @param p root of subtree (may be same as {@code root})
 * @param dir direction. 0: left rotate; 1: right rotate
 * @return new root of subtree
 */
auto rotate(pointer p, bool dir) -> pointer {
  auto g = p->fa;
  auto s = p->ch[!dir];  // new root of subtree
  assert(s);             // pointer to true node required
  s->sz = p->sz, p->sz = size(p->ch[dir]) + size(s->ch[dir]) + 1;
  auto c = s->ch[dir];
  if (c) c->fa = p;
  p->ch[!dir] = c, s->ch[dir] = p;
  p->fa = s, s->fa = g;
  if (g)
    g->ch[p == g->ch[1]] = s;
  else
    root = s;
  return s;
}

```

插入

红黑树的插入操作与普通的 BST 类似，对于红黑树来说，新插入的节点初始为红色，完成插入后需根据插入节点及相关节点的状态进行修正以满足上文提到的四条性质.

实现

```
/**
```

```
   * @return nullptr if insert failed, otherwise pointer of inserted node
   */
  auto insert(const key_t &data) -> const_pointer {
    pointer n = new node_t;
    n->fa = n->ch[0] = n->ch[1] = nullptr;
    n->data = data, n->sz = 1;
    pointer now = root, p = nullptr;
    bool dir = 0;
    while (now) {
      p = now;
      dir = compare(now->data, data);
      now = now->ch[dir];
    }
    insert_fixup_leaf(p, n, dir);
    return n;
  }

  /**
   * Insert leaf node {@code n} to {@code p}
   *
   * @param p parent of node which will be inserted
   * @param n leaf node which will be inserted
   * @param dir direction of n, 0: left; 1: right
   */
  void insert_leaf(pointer_const p, pointer_const n, bool dir) {
    if (!p) {
      root = n;
      return;
    }
    p->ch[dir] = n, n->fa = p;
    auto now = p;
    while (now) now->sz++, now = now->fa;
  }

  /**
   * Insert leaf node {@code n} to {@code p}, then fixup
   *
   * @param p parent of node which will be inserted
   * @param n node which will be inserted
   * @param dir direction of n, 0: left; 1: right
   */
  void insert_fixup_leaf(pointer p, pointer n, bool dir) {
    n->red = p;
    insert_leaf(p, n, dir);
    // Fix double red
    // ...
    // Post process: color root black
    root->red = false;
  }

```

插入后的平衡维护

> **Note**
>
> 为加深理解，请读者自行验证平衡维护后是否满足性质 4.

由于插入的节点若不为根节点则必为红色，所以插入后可能违反性质 3，需要维护平衡性.

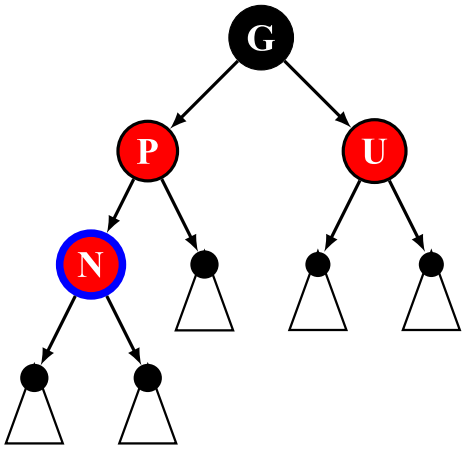
令插入的节点为 n，其父节点为 p，祖父节点为 g，叔节点为 u. 由性质 3 可知 g 必为黑色.

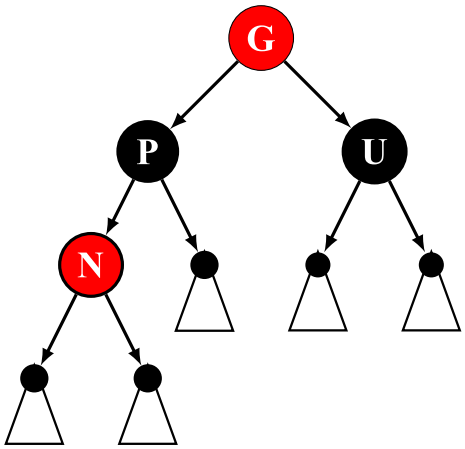
我们从插入的位置开始向上递归维护，若 p 为黑色即可终止，否则分为 3 种情况.

```
    while (is_red(p = n->fa)) {
      bool p_dir = p->child_dir();
      auto g = p->fa, u = g->ch[!p_dir];
      // ...
    }
```

Insert case 1

p 和 u 均为红色. 此时我们只需重新染色即可.





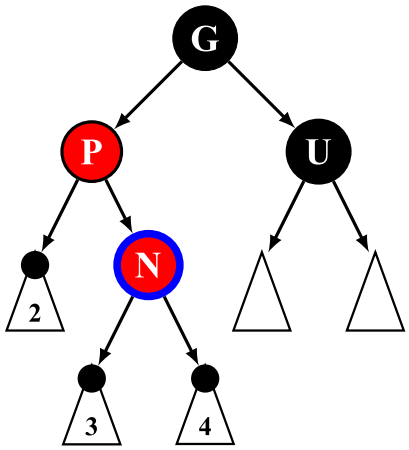
> **实现**

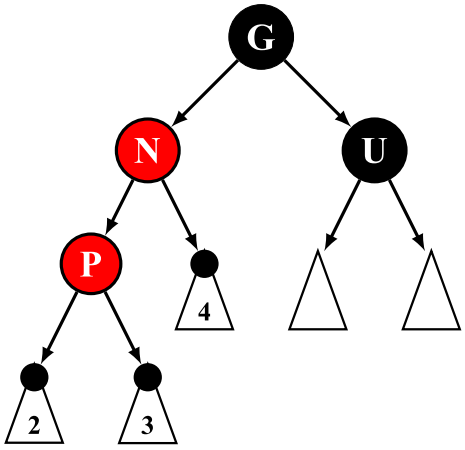
```
      // Case 1: both p and u are red
      //      g              [g]
      //     / \             / \
      //   [p] [u]   ==>    p   u
      //   /               /
      // [n]             [n]
      if (is_red(u)) {
        p->red = u->red = false;
        g->red = true;
        n = g;
        continue;
      }
```

Insert case 2

p 为红色，u 为黑色，p 的方向和 n 的方向不同.

此时我们需要旋转 p 节点来转为第三种情况.





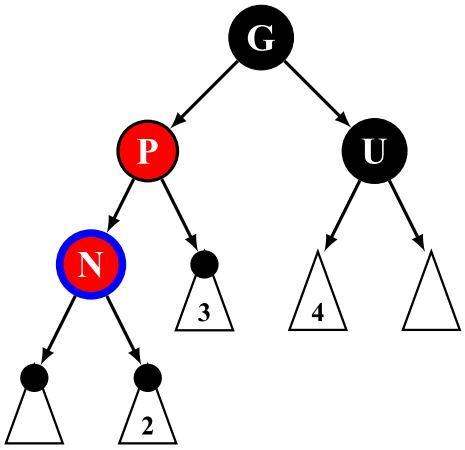
实现

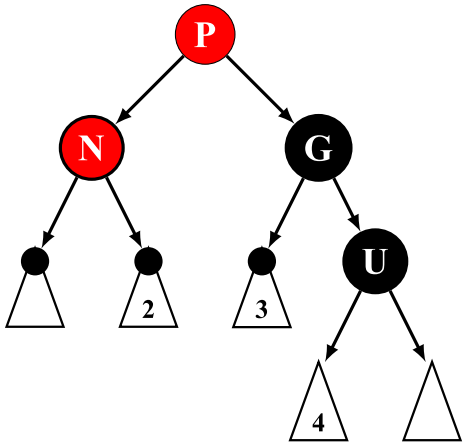
```
1       // p is red and u is black
2       // Case 2: dir of n is different with dir of p
3       //    g              g
4       //   / \            / \
5       // [p]  u   ==>   [n]  u
6       //   \              /
7       //   [n]         [p]
8       if (n->child_dir() != p_dir) rotate(p, p_dir), std::swap(n, p);
```

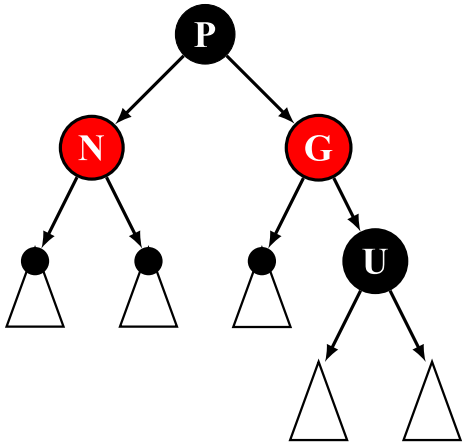
Insert case 3

p 为红色，u 为黑色，p 的方向和 n 的方向相同.

此时我们需要旋转 g 节点以将 p 转为子树的根，之后交换 p 和 g 的颜色即可.







实现

```
1       // Case 3: p is red, u is black and dir of n is same as dir of p
2       //      g             p
3       //     / \           / \
4       //   [p]  u   ==>  [n] [g]
```

```
      //   /                   \
      // [n]                    u
      p->red = false, g->red = true;
      rotate(g, !p_dir);
```

删除

红黑树的删除操作与普通的 BST 相比要多一些步骤．具体而言：

- 若待删除的节点 n 有两个子节点，则交换 n 和右子树中最小节点 s 的数据，并将 n 设为 s．此时 n 不可能有两个子节点．
- 若待删除的节点 n 有一个子节点 s．由性质 4 可知 s 必为红色，再由性质 3 可知 n 必为黑色．所以只需将 n 在父节点 p 中对应的指针替换为 s 的地址，以及将 s 的父节点指针替换为 p 的地址，之后再将 s 染黑即可．
- 若待删除的节点 n 没有子节点．若 n 是根节点或 n 是红色节点，则直接删除即可，否则直接删除会违反性质 4，需要维护平衡性．

实现

```
/**
 * @return succeed or not
 */
auto erase(const key_t &key) -> bool {
  auto p = lower_bound(key);
  if (!p || p->data != key) return false;
  erase(p);
  return true;
}

/**
 * @return {@code next(p)}
 */
auto erase(pointer p) -> const_pointer {
  if (!p) return nullptr;
  pointer result;
  if (p->ch[0] && p->ch[1]) {
    auto s = leftmost(p->ch[1]);
    std::swap(s->data, p->data);
    result = p, p = s;
  } else
    result = next(p);
  erase_fixup_branch_or_leaf(p);
  delete p;
  return result;
}

/**
 * Erase node {@code n}
 *
 * @param n node which will be deleted, must have no more than 2 child
 */
void erase_branch_or_leaf(pointer_const n) {
```

```
    auto p = n->fa, s = n->ch[0] ? n->ch[0] : n->ch[1];
    if (s) s->fa = p;
    if (!p) {
      root = s;
      return;
    }
    p->ch[n->child_dir()] = s;
    auto now = p;
    while (now) now->sz--, now = now->fa;
  }

  /**
   * Erase node {@code n}, then fixup
   *
   * @param n node which will be deleted, must have no more than 2 child
   */
  void erase_fixup_branch_or_leaf(pointer n) {
    bool n_dir = n == root ? false : n->child_dir();
    erase_branch_or_leaf(n);
    auto p = n->fa;
    if (!p) {  // n is root
      if (root) root->red = false;
      return;
    } else {
      auto s = p->ch[n_dir];
      if (s) {  // n has 1 child
        // n must be black and s must be red, so we need to color s black
        s->red = false;
        return;
      }
    }
    // n is not root but leaf with black color, need to be fixup
    // ...
    // Post process: see case 2 & case 4
    n->red = false;
  }

```

删除后的平衡维护

Note

为加深理解，请读者自行验证平衡维护后是否满足性质 4.

由上文讨论可知 n 是黑色叶子节点且不为根节点. 我们设 n 的父节点为 p，兄弟节点为 s，侄节点分别为 c 和 d.

删除的维护也是从 n 开始向上递归维护，若 n 是根或 n 为红色即可终止，否则分为 4 种情况.

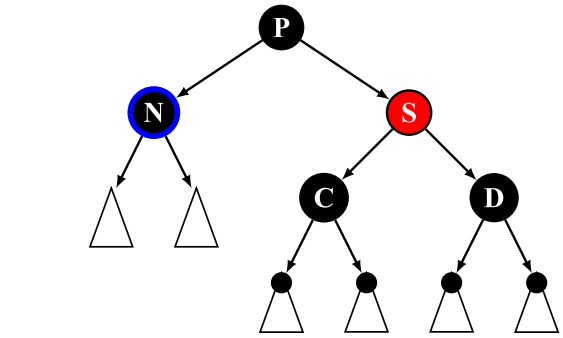
```
    while (p && !n->red) {
      auto s = p->ch[!n_dir];
      // Delete case 1
      // ...
```

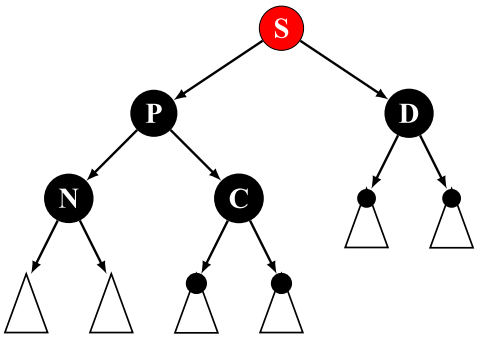
```
      // Other cases
      // s must be black
      auto c = s->ch[n_dir], d = s->ch[!n_dir];
      // ...
    end_erase_fixup:
      p = n->fa;
      if (!p) break;
      n_dir = n->child_dir();
    }
```

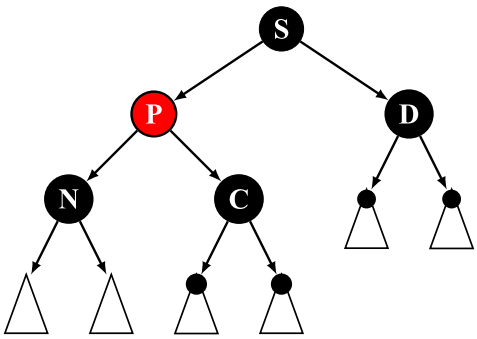
Delete case 1

s 为红色.

此时我们旋转 p，将 s 转为子树根节点，之后交换 s 和 p 的颜色来转为其余三种情况之一.







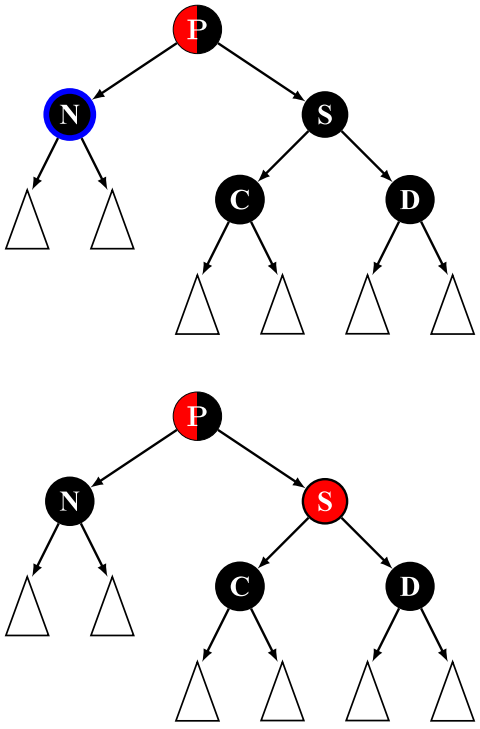
实现

```
      // Case 1: s is red
      //    p               s
      //   / \             / \
      // |n| [s]   ==>   [p]  d
      //     / \         / \
      //    c   d      |n|  c
      if (is_red(s)) {
        s->red = false, p->red = true;
        rotate(p, n_dir);
        s = p->ch[!n_dir];
      }
```

Delete case 2

p 的颜色不确定，s、c、d 均为黑色.

此时只需将 s 染红即可.



需要注意的是，若 p 为红色则会违反性质 3，但是若 p 为红色则会直接退出循环，所以我们在最后将其染黑.

实现

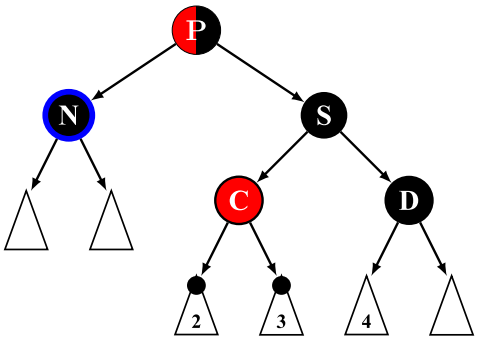
```
      // Case 2: both c and d are black
      //   {p}          {p}
      //   / \          / \
      // |n|  s   ==> |n| [s]
      //     / \          / \
      //    c   d        c   d
```

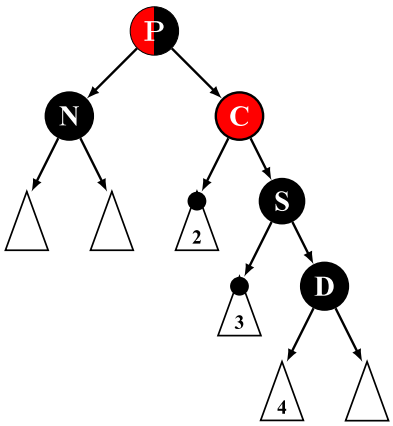
```
7       // p will be colored black in the end
8       if (!is_red(c) && !is_red(d)) {
9         s->red = true;
10        n = p;
11        goto end_erase_fixup;
12      }
```

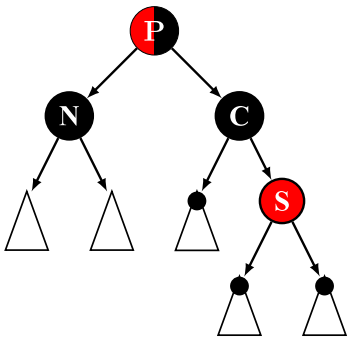
Delete case 3

p 的颜色不确定，s、d 均为黑色，c 为红色.

此时需要旋转 s 使 c 为原来 s 对应子树的根节点，并交换 s 和 c 的颜色转为第四种情况即可.







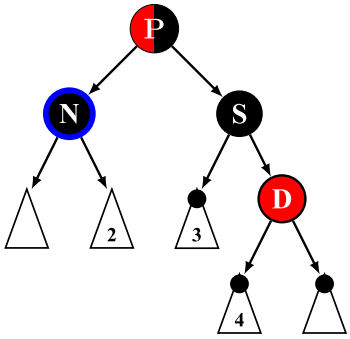
实现

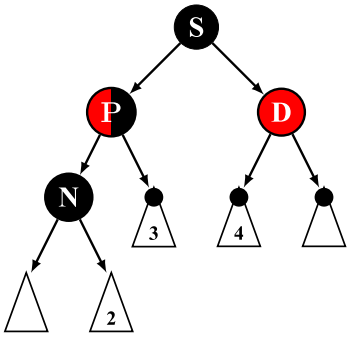
```
      // Case 3: c is red and d is black
      //   {p}          {p}
      //   / \          / \
      // |n|  s   ==> |n|  c
      //     / \           \
      //   [c]  d          [s]
      //                     \
      //                      d
      if (!is_red(d)) {
        c->red = false, s->red = true;
        rotate(s, !n_dir);
        s = p->ch[!n_dir], c = s->ch[n_dir], d = s->ch[!n_dir];
      }
```

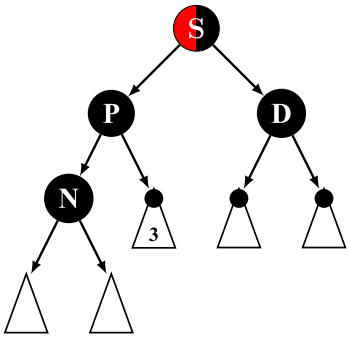
Delete case 4

p、c 的颜色不确定，s 为黑色，d 为红色.

此时需要旋转 p 使 s 为子树的根节点，交换 s 和 p 的颜色，并将 d 染黑即可终止维护平衡.







实现

```
1       // Case 4: d is red
2       //   {p}            {s}
3       //   / \            / \
4       // |n|  s   ==>    p   d
5       //     / \        / \
6       //   {c} [d]    |n| {c}
7       s->red = p->red, p->red = d->red = false;
8       rotate(p, n_dir), n = root;
```

参考代码

下面的代码是用红黑树实现的 set：

实现

```cpp
/**
 * @file rbtree.hpp
 * @brief An RBTree-based set implementation
 * @details The set is sorted according to the {@code compare_t} function
 * provided; This implementation provides find, insert, remove, find order,
find
 * key by order in O(log(n)) time.
 * @author [Tiphereth-A](https://github.com/Tiphereth-A)
 */

#ifndef RBTREE_HPP
#define RBTREE_HPP

#include <cassert>
#include <cstdint>
#include <functional>

using std::size_t;

/**
 * An RBTree-based set implementation
 *
 * @tparam key_t key type
 * @tparam compare_t compare function
 */
template <typename key_t, typename compare_t = std::less<key_t>>
struct rb_tree {
  /**
   * Tree node
   */
  struct node_t {
    node_t *fa;     // == nullptr if root of the tree, otherwise parent
    node_t *ch[2];  // == nullptr if child is empty
                    // ch[0]: left child, ch[1]: right child
    key_t data;
    size_t sz;  // Size of subtree
    bool red;   // == true if node is red, otherwise black

    /**
     * Get child direction of current non-null node
     *
     * @return true if this node is right child of its parent, otherwise
false
     */
    auto child_dir() const -> bool { return this == fa->ch[1]; }
  };

  using pointer = node_t *;
  using const_pointer = const node_t *;
  using pointer_const = node_t *const;

  const compare_t compare;
```

```
51    pointer root;
52
53    rb_tree() : compare{}, root{nullptr} {}
54
55
56    ~rb_tree() {
57      post_order([](auto it) { delete it; });
58    }
59
60    auto size() const -> size_t { return size(root); }
61
62    template <typename F>
63    void pre_order(F callback) {
64      auto f = [&](auto &&f, pointer p) {
65        if (!p) return;
66        callback(p), f(f, p->ch[0]), f(f, p->ch[1]);
67      };
68      f(f, root);
69    }
70
71    template <typename F>
72    void in_order(F callback) {
73      auto f = [&](auto &&f, pointer p) {
74        if (!p) return;
75        f(f, p->ch[0]), callback(p), f(f, p->ch[1]);
76      };
77      f(f, root);
78    }
79
80    template <typename F>
81    void post_order(F callback) {
82      auto f = [&](auto &&f, pointer p) {
83        if (!p) return;
84        f(f, p->ch[0]), f(f, p->ch[1]), callback(p);
85      };
86      f(f, root);
87    }
88
89    auto leftmost(const_pointer p) const { return most(p, 0); }
90
91    auto rightmost(const_pointer p) const { return most(p, 1); }
92
93    auto prev(const_pointer p) const { return neighbour(p, 0); }
94
95    auto next(const_pointer p) const { return neighbour(p, 1); }
96
97    auto lower_bound(const key_t &key) const -> pointer {
98      const_pointer now = root, ans = nullptr;
99      while (now) {
100       if (!compare(now->data, key))
101         ans = now, now = now->ch[0];
102       else
```

```
103         now = now->ch[1];
104       }
105       return (pointer)ans;
106     }
107
108     auto upper_bound(const key_t &key) const -> pointer {
109       const_pointer now = root, ans = nullptr;
110       while (now) {
111         if (compare(key, now->data))
112           ans = now, now = now->ch[0];
113         else
114           now = now->ch[1];
115       }
116       return (pointer)ans;
117     }
118
119     // Order start from 0
120     auto order_of_key(const key_t &key) const -> size_t {
121       size_t ans = 0;
122       auto now = root;
123       while (now) {
124         if (!compare(now->data, key))
125           now = now->ch[0];
126         else
127           ans += size(now->ch[0]) + 1, now = now->ch[1];
128       }
129       return ans;
130     }
131
132     // Order start from 0
133     auto find_by_order(size_t order) const -> const_pointer {
134       const_pointer now = root, ans = nullptr;
135       while (now && now->sz >= order) {
136         auto lsize = size(now->ch[0]);
137         if (order < lsize)
138           now = now->ch[0];
139         else {
140           ans = now;
141           if (order == lsize) break;
142           now = now->ch[1], order -= lsize + 1;
143         }
144       }
145       return ans;
146     }
147
148     /**
149      * @return nullptr if insert failed, otherwise pointer of inserted node
150      */
151     auto insert(const key_t &data) -> const_pointer {
152       pointer n = new node_t;
153       n->fa = n->ch[0] = n->ch[1] = nullptr;
154       n->data = data, n->sz = 1;
```

```
155     pointer now = root, p = nullptr;
156     bool dir = 0;
157     while (now) {
158       p = now;
159       dir = compare(now->data, data);
160       now = now->ch[dir];
161     }
162     insert_fixup_leaf(p, n, dir);
163     return n;
164   }
165
166   /**
167    * @return succeed or not
168    */
169   auto erase(const key_t &key) -> bool {
170     auto p = lower_bound(key);
171     if (!p || p->data != key) return false;
172     erase(p);
173     return true;
174   }
175
176   /**
177    * @return {@code next(p)}
178    */
179   auto erase(pointer p) -> const_pointer {
180     if (!p) return nullptr;
181     pointer result;
182     if (p->ch[0] && p->ch[1]) {
183       auto s = leftmost(p->ch[1]);
184       std::swap(s->data, p->data);
185       result = p, p = s;
186     } else
187       result = next(p);
188     erase_fixup_branch_or_leaf(p);
189     delete p;
190     return result;
191   }
192
193 private:
194   static auto size(const_pointer p) -> size_t { return p ? p->sz : 0; }
195
196   static auto is_red(const_pointer p) -> bool { return p ? p->red : false; }
197
198   /**
199    * @param dir 0: leftmost, 1: rightmost
200    */
201   auto most(const_pointer p, bool dir) const -> pointer {
202     if (!p) return nullptr;
203     while (p->ch[dir]) p = p->ch[dir];
204     return (pointer)p;
205   }
206
```

```
207   /**
208    * @param dir 0: prev, 1: next
209    */
210   auto neighbour(const_pointer p, bool dir) const -> pointer {
211     if (!p) return nullptr;
212     if (p->ch[dir]) return most(p->ch[dir], !dir);
213     if (p == root) return nullptr;
214     while (p && p->fa && p->child_dir() == dir) p = p->fa;
215     return p ? p->fa : nullptr;
216   }
217
218   /**
219    * Insert leaf node {@code n} to {@code p}
220    *
221    * @param p parent of node which will be inserted
222    * @param n leaf node which will be inserted
223    * @param dir direction of n, 0: left; 1: right
224    */
225   void insert_leaf(pointer_const p, pointer_const n, bool dir) {
226     if (!p) {
227       root = n;
228       return;
229     }
230     p->ch[dir] = n, n->fa = p;
231     auto now = p;
232     while (now) now->sz++, now = now->fa;
233   }
234
235   /**
236    * Erase node {@code n}
237    *
238    * @param n node which will be deleted, must have no more than 2 child
239    */
240   void erase_branch_or_leaf(pointer_const n) {
241     auto p = n->fa, s = n->ch[0] ? n->ch[0] : n->ch[1];
242     if (s) s->fa = p;
243     if (!p) {
244       root = s;
245       return;
246     }
247     p->ch[n->child_dir()] = s;
248     auto now = p;
249     while (now) now->sz--, now = now->fa;
250   }
251
252   /**
253    * @param p root of subtree (may be same as {@code root})
254    * @param dir direction. 0: left rotate; 1: right rotate
255    * @return new root of subtree
256    */
257   auto rotate(pointer p, bool dir) -> pointer {
258     auto g = p->fa;
```

```
259     auto s = p->ch[!dir];  // new root of subtree
260     assert(s);             // pointer to true node required
261     s->sz = p->sz, p->sz = size(p->ch[dir]) + size(s->ch[dir]) + 1;
262     auto c = s->ch[dir];
263     if (c) c->fa = p;
264     p->ch[!dir] = c, s->ch[dir] = p;
265     p->fa = s, s->fa = g;
266     if (g)
267       g->ch[p == g->ch[1]] = s;
268     else
269       root = s;
270     return s;
271   }
272
273 #pragma GCC diagnostic ignored "-Wcomment"
274
275   /**
276    * Insert leaf node {@code n} to {@code p}, then fixup
277    *
278    * @param p parent of node which will be inserted
279    * @param n node which will be inserted
280    * @param dir direction of n, 0: left; 1: right
281    */
282   void insert_fixup_leaf(pointer p, pointer n, bool dir) {
283     n->red = p;
284     insert_leaf(p, n, dir);
285     // Fix double red
286     while (is_red(p = n->fa)) {
287       bool p_dir = p->child_dir();
288       auto g = p->fa, u = g->ch[!p_dir];
289       // Case 1: both p and u are red
290       //      g              [g]
291       //     / \             / \
292       //   [p] [u]   ==>    p   u
293       //   /               /
294       // [n]             [n]
295       if (is_red(u)) {
296         p->red = u->red = false;
297         g->red = true;
298         n = g;
299         continue;
300       }
301       // p is red and u is black
302       // Case 2: dir of n is different with dir of p
303       //    g              g
304       //   / \            / \
305       // [p]  u   ==>   [n]  u
306       //   \            /
307       //   [n]        [p]
308       if (n->child_dir() != p_dir) rotate(p, p_dir), std::swap(n, p);
309       // Case 3: p is red, u is black and dir of n is same as dir of p
310       //      g             p
```

```
      //     / \           / \
      //   [p]  u   ==>  [n] [g]
      //   /                 \
      // [n]                  u
      p->red = false, g->red = true;
      rotate(g, !p_dir);
    }
    // Post process: color root black
    root->red = false;
  }

  /**
   * Erase node {@code n}, then fixup
   *
   * @param n node which will be deleted, must have no more than 2 child
   */
  void erase_fixup_branch_or_leaf(pointer n) {
    bool n_dir = n == root ? false : n->child_dir();
    erase_branch_or_leaf(n);
    auto p = n->fa;
    if (!p) {  // n is root
      if (root) root->red = false;
      return;
    } else {
      auto s = p->ch[n_dir];
      if (s) {  // n has 1 child
        // n must be black and s must be red, so we need to color s black
        s->red = false;
        return;
      }
    }
    // n is not root but leaf with black color, need to be fixup
    while (p && !n->red) {
      auto s = p->ch[!n_dir];
      // Case 1: s is red
      //    p               s
      //   / \             / \
      // |n| [s]   ==>   [p]  d
      //     / \         / \
      //    c   d      |n|  c
      if (is_red(s)) {
        s->red = false, p->red = true;
        rotate(p, n_dir);
        s = p->ch[!n_dir];
      }
      // s must be black
      auto c = s->ch[n_dir], d = s->ch[!n_dir];
      // Case 2: both c and d are black
      //   {p}          {p}
      //   / \          / \
      // |n|  s   ==> |n| [s]
      //     / \          / \
```

```
      //    c   d        c   d
      // p will be colored black in the end
      if (!is_red(c) && !is_red(d)) {
        s->red = true;
        n = p;
        goto end_erase_fixup;
      }
      // Case 3: c is red and d is black
      //   {p}          {p}
      //   / \          / \
      // |n|  s   ==> |n|  c
      //     / \            \
      //   [c]  d           [s]
      //                      \
      //                       d
      if (!is_red(d)) {
        c->red = false, s->red = true;
        rotate(s, !n_dir);
        s = p->ch[!n_dir], c = s->ch[n_dir], d = s->ch[!n_dir];
      }
      // Case 4: d is red
      //   {p}            {s}
      //   / \            / \
      // |n|  s   ==>    p   d
      //     / \        / \
      //   {c} [d]    |n| {c}
      s->red = p->red, p->red = d->red = false;
      rotate(p, n_dir), n = root;
    end_erase_fixup:
      p = n->fa;
      if (!p) break;
      n_dir = n->child_dir();
    }
    // Post process: see case 2 & case 4
    n->red = false;
  }

};

#pragma GCC diagnostic warning "-Wcomment"

#endif  // RBTREE_HPP
```

例题：Luogu P3369【模板】普通平衡树[7] 与 Luogu P6136【模板】普通平衡树（数据加强版）[8]

```
#include <cassert>
#include <cstdint>
#include <functional>

using std::size_t;

```

```cpp
/**
 * An RBTree-based set implementation
 *
 * @tparam key_t key type
 * @tparam compare_t compare function
 */
template <typename key_t, typename compare_t = std::less<key_t>>
struct rb_tree {
  /**
   * Tree node
   */
  struct node_t {
    node_t *fa;     // == nullptr if root of the tree, otherwise parent
    node_t *ch[2];  // == nullptr if child is empty
                    // ch[0]: left child, ch[1]: right child
    key_t data;
    size_t sz;  // Size of subtree
    bool red;   // == true if node is red, otherwise black

    /**
     * Get child direction of current non-null node
     *
     * @return true if this node is right child of its parent, otherwise false
     */
    auto child_dir() const -> bool { return this == fa->ch[1]; }
  };

  using pointer = node_t *;
  using const_pointer = const node_t *;
  using pointer_const = node_t *const;

  const compare_t compare;
  pointer root;

  rb_tree() : compare{}, root{nullptr} {}


  ~rb_tree() {
    post_order([](auto it) { delete it; });
  }

  auto size() const -> size_t { return size(root); }

  template <typename F>
  void pre_order(F callback) {
    auto f = [&](auto &&f, pointer p) {
      if (!p) return;
      callback(p), f(f, p->ch[0]), f(f, p->ch[1]);
    };
    f(f, root);
  }
```

```
58
59    template <typename F>
60    void in_order(F callback) {
61      auto f = [&](auto &&f, pointer p) {
62        if (!p) return;
63        f(f, p->ch[0]), callback(p), f(f, p->ch[1]);
64      };
65      f(f, root);
66    }
67
68    template <typename F>
69    void post_order(F callback) {
70      auto f = [&](auto &&f, pointer p) {
71        if (!p) return;
72        f(f, p->ch[0]), f(f, p->ch[1]), callback(p);
73      };
74      f(f, root);
75    }
76
77    auto leftmost(const_pointer p) const { return most(p, 0); }
78
79    auto rightmost(const_pointer p) const { return most(p, 1); }
80
81    auto prev(const_pointer p) const { return neighbour(p, 0); }
82
83    auto next(const_pointer p) const { return neighbour(p, 1); }
84
85    auto lower_bound(const key_t &key) const -> pointer {
86      const_pointer now = root, ans = nullptr;
87      while (now) {
88        if (!compare(now->data, key))
89          ans = now, now = now->ch[0];
90        else
91          now = now->ch[1];
92      }
93      return (pointer)ans;
94    }
95
96    auto upper_bound(const key_t &key) const -> pointer {
97      const_pointer now = root, ans = nullptr;
98      while (now) {
99        if (compare(key, now->data))
100         ans = now, now = now->ch[0];
101       else
102         now = now->ch[1];
103     }
104     return (pointer)ans;
105   }
106
107   // Order start from 0
108   auto order_of_key(const key_t &key) const -> size_t {
109     size_t ans = 0;
```

```
    auto now = root;
    while (now) {
      if (!compare(now->data, key))
        now = now->ch[0];
      else
        ans += size(now->ch[0]) + 1, now = now->ch[1];
    }
    return ans;
  }

  // Order start from 0
  auto find_by_order(size_t order) const -> const_pointer {
    const_pointer now = root, ans = nullptr;
    while (now && now->sz >= order) {
      auto lsize = size(now->ch[0]);
      if (order < lsize)
        now = now->ch[0];
      else {
        ans = now;
        if (order == lsize) break;
        now = now->ch[1], order -= lsize + 1;
      }
    }
    return ans;
  }

  /**
   * @return nullptr if insert failed, otherwise pointer of inserted node
   */
  auto insert(const key_t &data) -> const_pointer {
    pointer n = new node_t;
    n->fa = n->ch[0] = n->ch[1] = nullptr;
    n->data = data, n->sz = 1;
    pointer now = root, p = nullptr;
    bool dir = 0;
    while (now) {
      p = now;
      dir = compare(now->data, data);
      now = now->ch[dir];
    }
    insert_fixup_leaf(p, n, dir);
    return n;
  }

  /**
   * @return succeed or not
   */
  auto erase(const key_t &key) -> bool {
    auto p = lower_bound(key);
    if (!p || p->data != key) return false;
    erase(p);
    return true;
```

```
162   }
163
164   /**
165    * @return {@code next(p)}
166    */
167   auto erase(pointer p) -> const_pointer {
168     if (!p) return nullptr;
169     pointer result;
170     if (p->ch[0] && p->ch[1]) {
171       auto s = leftmost(p->ch[1]);
172       std::swap(s->data, p->data);
173       result = p, p = s;
174     } else
175       result = next(p);
176     erase_fixup_branch_or_leaf(p);
177     delete p;
178     return result;
179   }
180
181 private:
182   static auto size(const_pointer p) -> size_t { return p ? p->sz : 0; }
183
184   static auto is_red(const_pointer p) -> bool { return p ? p->red : false; }
185
186   /**
187    * @param dir 0: leftmost, 1: rightmost
188    */
189   auto most(const_pointer p, bool dir) const -> pointer {
190     if (!p) return nullptr;
191     while (p->ch[dir]) p = p->ch[dir];
192     return (pointer)p;
193   }
194
195   /**
196    * @param dir 0: prev, 1: next
197    */
198   auto neighbour(const_pointer p, bool dir) const -> pointer {
199     if (!p) return nullptr;
200     if (p->ch[dir]) return most(p->ch[dir], !dir);
201     if (p == root) return nullptr;
202     while (p && p->fa && p->child_dir() == dir) p = p->fa;
203     return p ? p->fa : nullptr;
204   }
205
206   /**
207    * Insert leaf node {@code n} to {@code p}
208    *
209    * @param p parent of node which will be inserted
210    * @param n leaf node which will be inserted
211    * @param dir direction of n, 0: left; 1: right
212    */
213   void insert_leaf(pointer_const p, pointer_const n, bool dir) {
```

```
214     if (!p) {
215       root = n;
216       return;
217     }
218     p->ch[dir] = n, n->fa = p;
219     auto now = p;
220     while (now) now->sz++, now = now->fa;
221   }
222
223   /**
224    * Erase node {@code n}
225    *
226    * @param n node which will be deleted, must have no more than 2 child
227    */
228   void erase_branch_or_leaf(pointer_const n) {
229     auto p = n->fa, s = n->ch[0] ? n->ch[0] : n->ch[1];
230     if (s) s->fa = p;
231     if (!p) {
232       root = s;
233       return;
234     }
235     p->ch[n->child_dir()] = s;
236     auto now = p;
237     while (now) now->sz--, now = now->fa;
238   }
239
240   /**
241    * @param p root of subtree (may be same as {@code root})
242    * @param dir direction. 0: left rotate; 1: right rotate
243    * @return new root of subtree
244    */
245   auto rotate(pointer p, bool dir) -> pointer {
246     auto g = p->fa;
247     auto s = p->ch[!dir];  // new root of subtree
248     assert(s);             // pointer to true node required
249     s->sz = p->sz, p->sz = size(p->ch[dir]) + size(s->ch[dir]) + 1;
250     auto c = s->ch[dir];
251     if (c) c->fa = p;
252     p->ch[!dir] = c, s->ch[dir] = p;
253     p->fa = s, s->fa = g;
254     if (g)
255       g->ch[p == g->ch[1]] = s;
256     else
257       root = s;
258     return s;
259   }
260
261 #pragma GCC diagnostic ignored "-Wcomment"
262
263   /**
264    * Insert leaf node {@code n} to {@code p}, then fixup
265    *
```

```
266    * @param p parent of node which will be inserted
267    * @param n node which will be inserted
268    * @param dir direction of n, 0: left; 1: right
269    */
270   void insert_fixup_leaf(pointer p, pointer n, bool dir) {
271     n->red = p;
272     insert_leaf(p, n, dir);
273     // Fix double red
274     while (is_red(p = n->fa)) {
275       bool p_dir = p->child_dir();
276       auto g = p->fa, u = g->ch[!p_dir];
277       // Case 1: both p and u are red
278       //      g              [g]
279       //     / \             / \
280       //   [p] [u]   ==>    p   u
281       //   /               /
282       // [n]             [n]
283       if (is_red(u)) {
284         p->red = u->red = false;
285         g->red = true;
286         n = g;
287         continue;
288       }
289       // p is red and u is black
290       // Case 2: dir of n is different with dir of p
291       //    g              g
292       //   / \            / \
293       // [p]  u   ==>   [n]  u
294       //   \            /
295       //   [n]        [p]
296       if (n->child_dir() != p_dir) rotate(p, p_dir), std::swap(n, p);
297       // Case 3: p is red, u is black and dir of n is same as dir of p
298       //      g             p
299       //     / \           / \
300       //   [p]  u   ==>  [n] [g]
301       //   /                 \
302       // [n]                  u
303       p->red = false, g->red = true;
304       rotate(g, !p_dir);
305     }
306     // Post process: color root black
307     root->red = false;
308   }
309
310   /**
311    * Erase node {@code n}, then fixup
312    *
313    * @param n node which will be deleted, must have no more than 2 child
314    */
315   void erase_fixup_branch_or_leaf(pointer n) {
316     bool n_dir = n == root ? false : n->child_dir();
317     erase_branch_or_leaf(n);
```

```
    auto p = n->fa;
    if (!p) {  // n is root
      if (root) root->red = false;
      return;
    } else {
      auto s = p->ch[n_dir];
      if (s) {  // n has 1 child
        // n must be black and s must be red, so we need to color s black
        s->red = false;
        return;
      }
    }
    // n is not root but leaf with black color, need to be fixup
    while (p && !n->red) {
      auto s = p->ch[!n_dir];
      // Case 1: s is red
      //    p               s
      //   / \             / \
      // |n| [s]   ==>   [p]  d
      //     / \         / \
      //    c   d      |n|  c
      if (is_red(s)) {
        s->red = false, p->red = true;
        rotate(p, n_dir);
        s = p->ch[!n_dir];
      }
      // s must be black
      auto c = s->ch[n_dir], d = s->ch[!n_dir];
      // Case 2: both c and d are black
      //   {p}          {p}
      //   / \          / \
      // |n|  s   ==> |n| [s]
      //     / \          / \
      //    c   d        c   d
      // p will be colored black in the end
      if (!is_red(c) && !is_red(d)) {
        s->red = true;
        n = p;
        goto end_erase_fixup;
      }
      // Case 3: c is red and d is black
      //   {p}          {p}
      //   / \          / \
      // |n|  s   ==> |n|  c
      //     / \            \
      //   [c]  d           [s]
      //                      \
      //                       d
      if (!is_red(d)) {
        c->red = false, s->red = true;
        rotate(s, !n_dir);
        s = p->ch[!n_dir], c = s->ch[n_dir], d = s->ch[!n_dir];
```

```
      }
      // Case 4: d is red
      //   {p}            {s}
      //   / \            / \
      // |n|  s   ==>    p   d
      //     / \        / \
      //   {c} [d]    |n| {c}
      s->red = p->red, p->red = d->red = false;
      rotate(p, n_dir), n = root;
    end_erase_fixup:
      p = n->fa;
      if (!p) break;
      n_dir = n->child_dir();
    }
    // Post process: see case 2 & case 4
    n->red = false;
  }

};

#pragma GCC diagnostic warning "-Wcomment"

#include <iostream>
#include <numeric>
#include <vector>

template <bool P6136>  // == false: P3369, == true: P6136
struct IO_luogu_P3369_P6136 {
  int last_ans = 0;
  std::size_t n, m;
  std::vector<int> a;
  std::vector<int> ans_list;

  void init() {
    std::cin >> n;
    if (!P6136) {
      m = n;
      return;
    }
    std::cin >> m;
    a.resize(n);
    for (auto &i : a) std::cin >> i;
  }

  int opt() {
    int x;
    std::cin >> x;
    return x;
  }

  int x() {
    int x;
```

```
    std::cin >> x;
    return P6136 ? x ^ last_ans : x;
  }

  void print(int ans) { ans_list.push_back(last_ans = ans); }

  void print_total_ans() {
    if (P6136)
      std::cout << std::accumulate(ans_list.begin(), ans_list.end(), 0,
                                   std::bit_xor<int>{})
                << '\n';
    else
      for (auto &&i : ans_list) std::cout << i << '\n';
  }
};

// 把这里的模板参数改为 false 即为「Luogu P3369【模板】普通平衡树」的代码
IO_luogu_P3369_P6136<true> io;

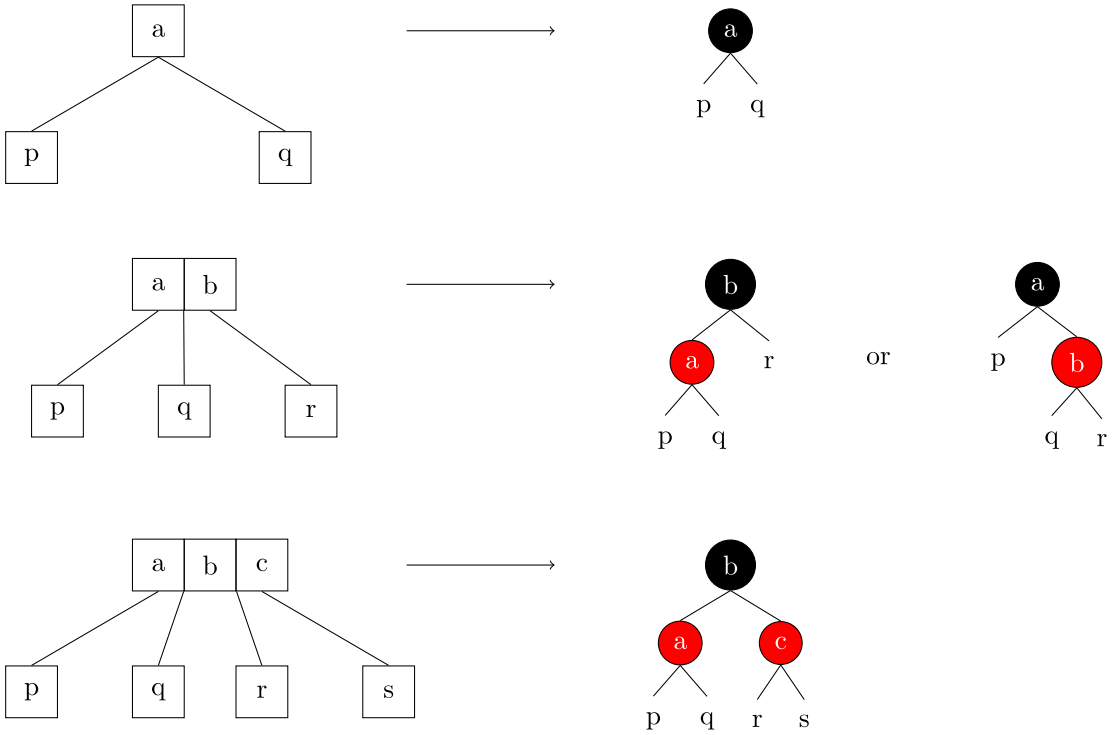
int main() {
  std::cin.tie(nullptr)->sync_with_stdio(false);
  io.init();
  std::size_t n = io.n, m = io.m;
  rb_tree<std::pair<int, int>> bt;
  int cnt = 0;
  for (auto i : io.a) bt.insert(std::make_pair(i, cnt++));
  for (int i = 0, opt, x; (std::size_t)i < m; ++i) {
    opt = io.opt(), x = io.x();
    switch (opt) {
      case 1:
        bt.insert({x, cnt++});
        break;
      case 2:
        bt.erase(bt.lower_bound({x, 0}));
        break;
      case 3:
        io.print((int)bt.order_of_key({x, 0}) + 1);
        break;
      case 4:
        io.print(bt.find_by_order((uint32_t)x - 1)->data.first);
        break;
      case 6:
        io.print(bt.upper_bound({x, n + m})->data.first);
        break;
      case 5:
        auto it = bt.lower_bound({x, 0});
        io.print((it ? bt.prev(it) : bt.rightmost(bt.root))->data.first);
        break;
    }
  }
  io.print_total_ans();
  return 0;
```

```
474 }
475
```

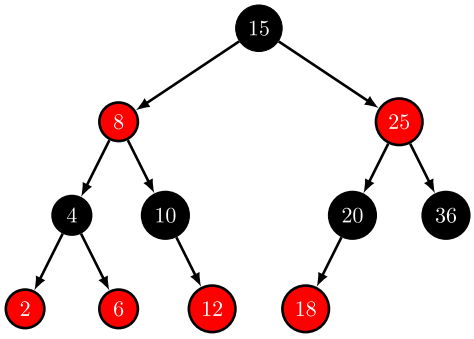
与 2-3-4 树的关系

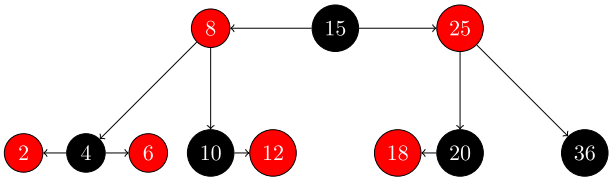
2-3-4 树是 4 阶 B 树，与一般的 B 树一样，2-3-4 树可以实现在 $O(\log n)$ 时间内进行搜索、插入和删除操作．2-3-4 树的节点分为三种，2 节点、3 节点和 4 节点，分别包含一个、两个或三个数据元素．所有的叶子节点都处于同一深度（最底层），所有数据都有序存储．

2-3-4 树和红黑树是同构的，任意一棵红黑树都唯一对应一棵 2-3-4 树．在 2-3-4 树上的插入和删除操作导致节点的扩展、分裂和合并，相当于红黑树中的变色和旋转．下图是 2-3-4 树的 2 节点、3 节点和 4 节点对应的红黑树节点．注意到 2-3-4 树的 3 节点对应红黑树中红色节点左偏和右偏两种情况，所以一棵红黑树可能对应多棵 2-3-4 树．



下图是一棵红黑树和与之对应的 2-3-4 树．将红黑树中的红色节点上移到父节点的左右两侧，形成一个 B 树节点，就可以得到与之对应的 2-3-4 树．可以发现，红黑树的节点数等于 2-3-4 树的节点个数．





可以通过对比 2-3-4 树来理解红黑树的插入和删除操作. [20]

实际工程项目中的使用

由于红黑树是目前主流工业界综合效率最高的内存型平衡树，其在实际的工程项目中有着广泛的使用，这里列举几个实际的使用案例并给出相应的源码链接，以便读者进行对比学习.

Linux

源码:

- `linux/lib/rbtree.c`[10]

Linux 中的红黑树所有操作均使用循环迭代进行实现，保证效率的同时又增加了大量的注释来保证代码可读性，十分建议读者阅读学习. Linux 内核中的红黑树使用非常广泛，这里仅列举几个经典案例.

- CFS 非实时任务调度[11]

  Linux 的稳定内核版本在 2.6.24 之后，使用了新的调度程序 CFS，所有非实时可运行进程都以虚拟运行时间为键值用一棵红黑树进行维护，以完成更公平高效地调度所有任务. CFS 弃用 active/expired 数组和动态计算优先级，不再跟踪任务的睡眠时间和区别是否交互任务，而是在调度中采用基于时间计算键值的红黑树来选取下一个任务，根据所有任务占用 CPU 时间的状态来确定调度任务优先级.

- epoll[12]

  epoll 全称 event poll，是 Linux 内核实现 IO 多路复用 (IO multiplexing) 的一个实现，是原先 poll/select 的改进版. Linux 中 epoll 的实现选择使用红黑树来储存文件描述符.

Nginx

源码:

- `nginx/src/core/ngx_rbtree.h`[13]
- `nginx/src/core/ngx_rbtree.c`[14]

nginx 中的用户态定时器是通过红黑树实现的. 在 nginx 中，所有 timer 节点都由一棵红黑树进行维护，在 worker 进程的每一次循环中都会调用 `ngx_process_events_and_timers` 函数，在该函数中就会调用处理定时器的函数 `ngx_event_expire_timers`，每次该函数都不断的从红黑树中取出时间值最小的，查看他们是否已经超时，然后执行他们的函数，直到取出的节点的时间没有超时为止.

关于 nginx 中红黑树的源码分析公开资源很多，读者可以自行查找学习.

C++

源码:

- GNU libstdc++
  - `libstdc++-v3/include/bits/stl_tree.h`[15]

[20] 这篇博文[9] 提供了详细的描述.

    - `libstdc++-v3/src/c++98/tree.cc`[16]

  另外, `libstdc++` 在 `<ext/rb_tree>` 中提供了 `__gnu_cxx::rb_tree`[17], 其继承了 `std::_Rb_tree`, 可以认为是供外部使用的类型别名. 需要注意的是, 该头文件 **不是** C++ 标准的一部分, 所以非必要不推荐使用.

  `libstdc++` 的 `pb_ds`→4.4.9.3@p454 中也提供了红黑树.

- LLVM libcxx
    - `libcxx/include/__tree`[18]
- Microsoft STL
    - `stl/inc/xtree`[19]

大多数 STL 中的 `std::set` 和 `std::map` 的内部数据结构就是红黑树 (例如上面提到的这些). 不过值得注意的是, C++ 标准并未规定必须以红黑树实现 `std::set` 和 `std::map`, 所以不应该在工程项目中直接使用 `std::set` 和 `std::map` 的内部数据结构.

OpenJDK

源码:

- `java.util.TreeMap<K, V>`[20]
- `java.util.TreeSet<K, V>`[21]
- `java.util.HashMap<K, V>`[22]

JDK 中的 `TreeMap` 和 `TreeSet` 都是使用红黑树作为底层数据结构的. 同时在 JDK 1.8 之后 `HashMap` 内部哈希表中每个表项的链表长度超过 8 时也会自动转变为红黑树以提升查找效率.

参考资料

- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022).*Introduction to algorithms*. MIT press.
- Red-Black Tree - Wikipedia[23]
- Red-Black Tree Visualization[24]

[1] 2-3-4 树

[2] 10.1109/SFCS.1978.3

[3] 维基百科原文

[4] https://en.wikipedia.org/wiki/Red–black_tree#cite_note-Cormen2009-18

[5] https://en.wikipedia.org/wiki/Red–black_tree#cite_note-Mehlhorn2008-17

[6] https://en.wikipedia.org/wiki/Red–black_tree#cite_note-Algs4-16

[7] Luogu P3369【模板】普通平衡树

[8] Luogu P6136【模板】普通平衡树（数据加强版）

[9] 这篇博文

[10] `linux/lib/rbtree.c`

[11] CFS 非实时任务调度

[12] epoll

[13] `nginx/src/core/ngx_rbtree.h`

[14] `nginx/src/core/ngx_rbtree.c`

[15] `libstdc++-v3/include/bits/stl_tree.h`

[16] `libstdc++-v3/src/c++98/tree.cc`

[17] `__gnu_cxx::rb_tree`

[18] `libcxx/include/__tree`

[19] `stl/inc/xtree`

[20] `java.util.TreeMap<K, V>`

[21] `java.util.TreeSet<K, V>`

[22] `java.util.HashMap<K, V>`

[23] Red-Black Tree - Wikipedia

[24] Red-Black Tree Visualization

10.15.10 左偏红黑树

> **Authors:** c-forrest, Enter-tainer, giiiiiithub, hly1204, iamtwz, Ir1d, kigawas, ksyx, luxuryspark567, mgt, orzAtalod, sandyzikun, SunsetGlow95, Tiphereth-A, current2020, untitledunrevised, yuhuoji

左偏红黑树是 红黑树→10.15.9@p2362 的一种变体，它的对红边（点）的位置做了一定限制，使得其插入与删除操作可以与 2-3 树[1] 构成一一对应.

我们假设读者已经至少掌握了一种基于旋转的平衡树，因此本文不会对旋转操作进行讲解.

红黑树

性质

一棵红黑树满足如下性质:

1. 节点是红色或黑色;
2. NIL 节点（空叶子节点）为黑色;
3. 红色的节点的所有儿子的颜色必须是黑色，即从每个叶子到根的所有路径上不能有两个连续的红色节点;
4. 从任一节点到其子树中的每个叶子的所有简单路径上都包含相同数目的黑色节点.（黑高平衡）

这保证了从根节点到任意叶子的最长路径（红黑交替）不会超过最短路径（全黑）的二倍. 从而保证了树的平衡性.

维护这些性质是比较复杂的，如果我们要插入一个节点，首先，它一定会被染色成红色，否则会破坏性质 4. 即使这样，我们还是有可能会破坏性质 3. 因此需要进行调整. 而删除节点就更加麻烦，与插入类似，我们不能删除黑色节点，否则会破坏黑高的平衡. 如何方便地解决这些问题呢?

左偏红黑树（Left Leaning Red Black Tree）

解释

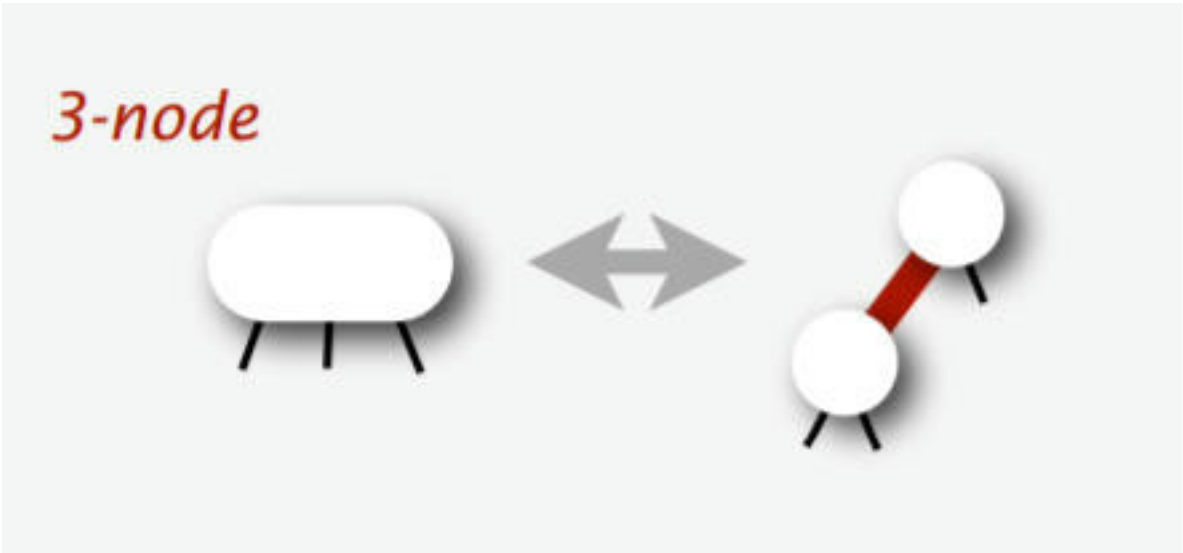
左偏红黑树是一种容易实现的红黑树变体.

在以下左偏红黑树示意图中, 是边具有颜色而不是节点具有颜色. 我们习惯用一个节点的颜色代指它的父亲边的颜色.

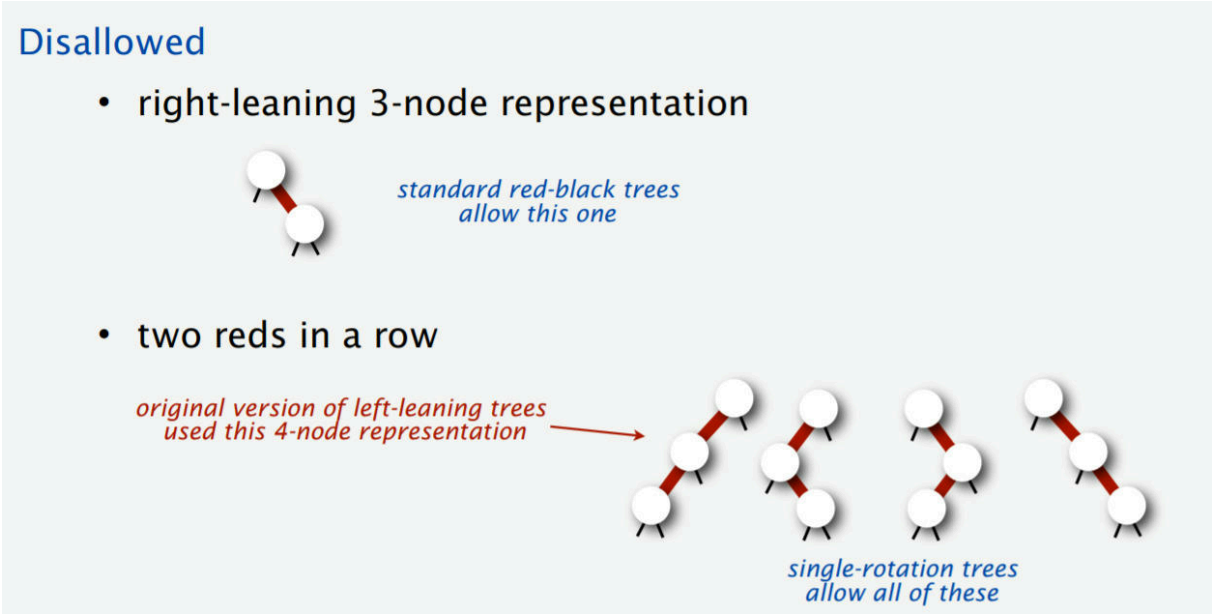
左偏红黑树对红黑树进行了进一步限制，一个黑色节点的左右儿子:

- 要么全是黑色;
- 要么左儿子是红色，右儿子是黑色.

符合条件的情况:



不符合条件的情况:

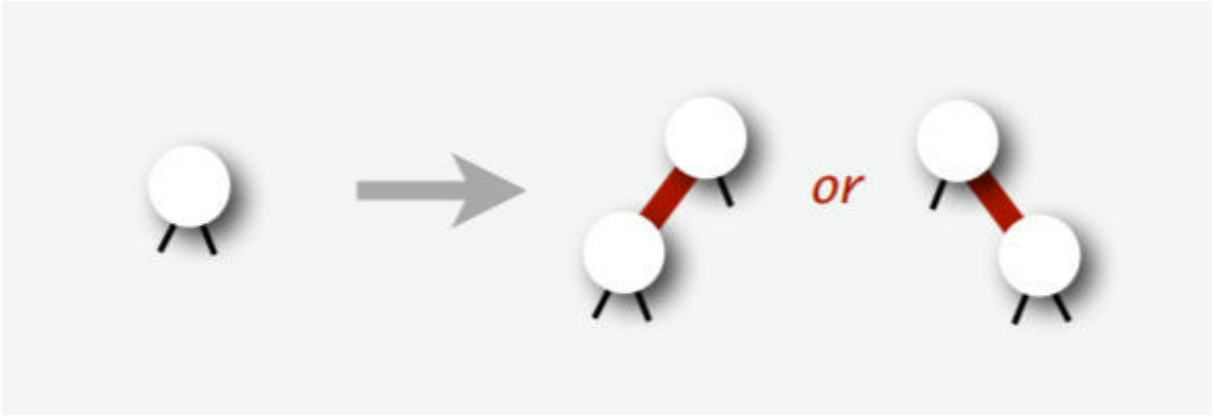


这是左偏树的「左偏」性质：红色边只能是左偏的.

过程

插入

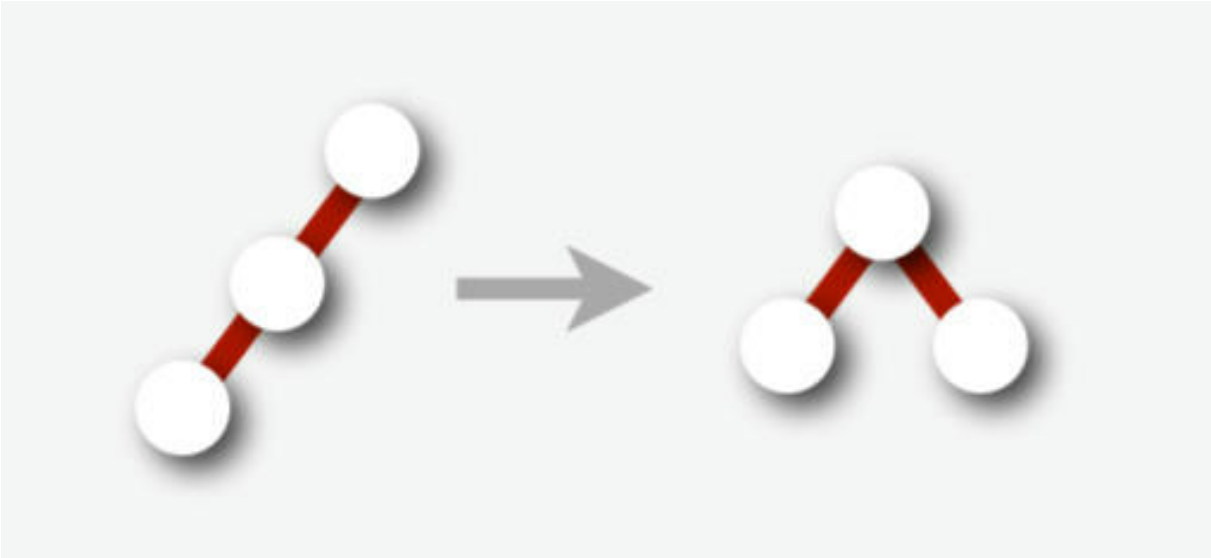
我们首先使用普通的 BST 插入方法，在树的底部插入一个红色的叶子节点，然后通过从下向上的调整，使得插入后的树仍然符合左偏红黑树的性质. 下面描述调整的过程：



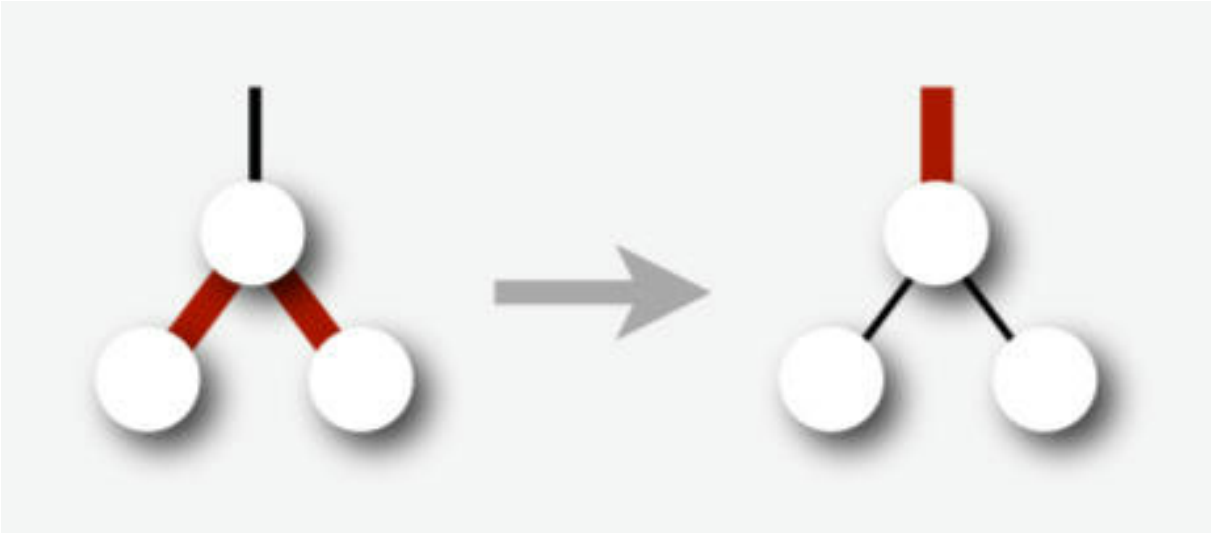
插入后，可能会产生一条右偏的红色边，因此需要对红边右偏的情况进行一次左旋：



考虑左旋后会产生两条连续的左偏红色边：



因此需要把它进行一次右旋．而对于右旋后的情况，我们应该对它进行 `color_flip`：即翻转该节点和它的两个儿子的颜色



从而消灭右偏的红边．

参考代码（部分）

```
template <class Key, class Compare>
typename Set<Key, Compare>::Node *Set<Key, Compare>::fix_up(
    Set::Node *root) const {
  if (is_red(root->rc) && !is_red(root->lc))  // fix right leaned red link
    root = rotate_left(root);
  if (is_red(root->lc) &&
      is_red(root->lc->lc))  // fix doubly linked left leaned red link
    // if (root->lc == nullptr), then the second expr won't be evaluated
    root = rotate_right(root);
  if (is_red(root->lc) && is_red(root->rc))
    // break up 4 node
```