XCPC 算法模板和结论 (补充)

October 15, 2025

Contents

Chapter 1

几何

1.1. 计算两个扇形区域的交

atan2() 下。

```cpp

using ld = long double;
/**
 * @brief 计算两个不跨越 (-PI, PI] 边界的区间 [a1, b1] 和 [a2, b2] 的交集长度。
 */
ld intersect_non_crossing(ld a1, ld b1, ld a2, ld b2) {
  // 交集的左端点是 max(a1, a2)
  ld start = max(a1, a2);
  // 交集的右端点是 min(b1, b2)
  ld end = min(b1, b2);

  // 只有当 start < end 时，交集才存在
  if (start < end) {
    return end - start;
  } else {
    return 0.0;
  }
}

ld ints(ld l1, ld r1, ld l2, ld r2) {
  ld total_intersection = 0.0;

  vector<pair<ld, ld>> segs1;
  if (l1 <= r1) {
    segs1.push_back({l1, r1});
  } else {
    segs1.push_back({l1, PI});
    segs1.push_back({-PI, r1});
  }

  vector<pair<ld, ld>> segs2;
  if (l2 <= r2) {
    segs2.push_back({l2, r2});
  } else {
    segs2.push_back({l2, PI});
    segs2.push_back({-PI, r2});
  }
  for (const auto& seg1 : segs1) {
    ld a1 = seg1.first;
    ld b1 = seg1.second;

    for (const auto& seg2 : segs2) {
      ld a2 = seg2.first;
      ld b2 = seg2.second;
      total_intersection += intersect_non_crossing(a1, b1, a2, b2);
    }
  }
  return min(total_intersection, PI * 2);
}
```

1.2. 极角排序

```cpp
#include <bits/stdc++.h>
using ld = long double;
const ld PI = acos(-1);
const ld EPS = 1e-7;
const ld INF = numeric_limits<ld>::max();
```

```cpp
#define cc(x) cout << fixed << setprecision(x);

template <class T>
struct Point {  // 在C++17下使用 emplace_back 绑定可能会导致CE!
  T x, y;
  Point(T x_ = 0, T y_ = 0) : x(x_), y(y_) {}  // 初始化
  template <class U>
  operator Point<U>() {  // 自动类型匹配
    return Point<U>(U(x), U(y));
  }
  Point &operator+=(Point p) & { return x += p.x, y += p.y, *this; }
  Point &operator+=(T t) & { return x += t, y += t, *this; }
  Point &operator-=(Point p) & { return x -= p.x, y -= p.y, *this; }
  Point &operator-=(T t) & { return x -= t, y -= t, *this; }
  Point &operator*=(T t) & { return x *= t, y *= t, *this; }
  Point &operator/=(T t) & { return x /= t, y /= t, *this; }
  Point operator-() const { return Point(-x, -y); }
  friend Point operator+(Point a, Point b) { return a += b; }
  friend Point operator+(Point a, T b) { return a += b; }
  friend Point operator-(Point a, Point b) { return a -= b; }
  friend Point operator-(Point a, T b) { return a -= b; }
  friend Point operator*(Point a, T b) { return a *= b; }
  friend Point operator*(T a, Point b) { return b *= a; }
  friend Point operator/(Point a, T b) { return a /= b; }
  friend bool operator<(Point a, Point b) {
    return equal(a.x, b.x) ? a.y < b.y - EPS : a.x < b.x - EPS;
  }
  friend bool operator>(Point a, Point b) { return b < a; }
  friend bool operator==(Point a, Point b) { return !(a < b) && !(b < a); }
  friend bool operator!=(Point a, Point b) { return a < b || b < a; }
  friend auto &operator>>(istream &is, Point &p) { return is >> p.x >> p.y; }
  friend auto &operator<<(ostream &os, Point p) {
    return os << "(" << p.x << ", " << p.y << ")";
  }
};

using Points = vector<Point<int>>;

double theta(auto p) { return atan2(p.y, p.x); }
void psort(Points &ps, Point<int> c = {0, 0}) {
  sort(ps.begin(), ps.end(),
       [&](auto p1, auto p2) { return theta(p1 - c) < theta(p2 - c); });
}
```

'atan2(y,x)'函数，从第三象限向第二象限递增，值域 $[-\pi, +\pi]$。

1.3. 旋转卡壳

```cpp
#include <bits/stdc++.h>

template <class T>
pair<vector<Point<T>>, T> maxInscribedQuadrilateral(vector<Point<T>>& p) {
  int n = (int)p.size();
  // 至少需要4个顶点
  if (n < 4) return {{}, 0};
  // 计算三角形的双倍面积（绝对值）
  auto area2 = [&](int i, int j, int k) {
    T dx1 = p[j].x - p[i].x;
    T dy1 = p[j].y - p[i].y;
    T dx2 = p[k].x - p[i].x;
    T dy2 = p[k].y - p[i].y;
    return abs(dx1 * dy2 - dy1 * dx2);
  };

  // 找最低且最左顶点 (a0)，最高且最右顶点 (c0)
  int a0 = 0, c0 = 0;
  for (int i = 1; i < n; i++) {
    if (p[i].y < p[a0].y || (p[i].y == p[a0].y && p[i].x < p[a0].x)) a0 = i;
    if (p[i].y > p[c0].y || (p[i].y == p[c0].y && p[i].x > p[c0].x)) c0 = i;
  }

  int a = a0, b = a0, c = c0, d = c0;
  T maxArea = 0;
  int bestA = a, bestB = b, bestC = c, bestD = d;

  // 旋转卡尺主循环
  while (true) {
    // 移动 b 指针，使 A-B-C 三角形面积最大
    while (true) {
      int nb = (b + 1) % n;
      if (area2(a, nb, c) > area2(a, b, c))
```

```
        b = nb;
      else
        break;
    }
    // 移动 d 指针，使 C-D-A 三角形面积最大
    while (true) {
      int nd = (d + 1) % n;
      if (area2(c, nd, a) > area2(c, d, a))
        d = nd;
      else
        break;
    }
    // 计算四边形面积（注意这里 area2 返回双倍面积）
    T areaQuad = (area2(a, b, c) + area2(a, c, d)) * T(0.5);
    if (areaQuad > maxArea) {
      maxArea = areaQuad;
      bestA = a;
      bestB = b;
      bestC = c;
      bestD = d;
    }
    // 判断旋转方向：比较移动 C 后与移动 A 后的面积
    int a_next = (a + 1) % n;
    int c_next = (c + 1) % n;
    // 如果 area2(a,a_next,c_next) <= area2(a,a_next,c)，则移动 A，否则移动 C
    if (area2(a, a_next, c_next) <= area2(a, a_next, c)) {
      a = a_next;
    } else {
      c = c_next;
    }
    // 退出条件：回到初始对换位置
    if (a == c0 && c == a0) break;
  }

  vector<Point<T>> quad = {p[bestA], p[bestB], p[bestC], p[bestD]};
  return {quad, maxArea};
}
```

1.4. 切比雪夫距离

$$d(A,B) = \max(|x_a - x_b|, |y_a - y_b|)$$

1. 曼哈顿坐标系是通过切比雪夫坐标系旋转45度后，再缩小到原来的一半得到的。
2. 将一个点 (x, y) 的坐标变为 $(x+y, x-y)$ 后，原坐标系中的曼哈顿距离等于新坐标系中的切比雪夫距离。
3. 将一个点 (x, y) 的坐标变为 $(\frac{x+y}{2}, \frac{x-y}{2})$ 后，原坐标系中的切比雪夫距离等于新坐标系中的曼哈顿距离。

Chapter 2

树

2.1. 树的重心

计算以无根树每个点为根节点时的最大子树大小，这个值最小的点称为无根树的重心。
一些性质:
1. 某个点是树的重心等价于它最大子树大小不大于整棵树大小的一半。
2. 树至多有两个重心。如果树有两个重心，那么它们相邻。此时树一定有偶数个节点，且可以被划分为两个大小相等的分支，每个分支各自包含一个重心。
3. 树中所有点到某个点的距离和中，到重心的距离和是最小的；如果有两个重心，那么到它们的距离和一样。反过来，距离和最小的点一定是重心。
4. 往树上增加或减少一个叶子，如果原节点数是奇数，那么重心可能增加一个，原重心仍是重心；如果原节点数是偶数，重心可能减少一个，另一个重心仍是重心。
5. 把两棵树通过一条边相连得到一棵新的树，则新的重心在较大的一棵树一侧的连接点与原重心之间的简单路径上。如果两棵树大小一样，则重心就是两个连接点。
利用性质1, 可以很快的找到重心.

2.2. 树的直径

若树上所有边边权均为正，则树的所有直径中点重合
由此可以知道找出 (端点) 字典序最大的直径的方法：从任意顶点出发：找到离它最远其字典序最大的点。然后从这个点出发找离他最远且字典序最大的点。这条直径就是答案。

2.3. O(1)LCA

预处理 $O(n \log n)$

```
#include <bits/stdc++.h>

vector<int> g[N];
int len[N], dep[N], mxs[N], eord[N << 1], fp[N], idx, fa[N];
void dfs(int x, int pre) {
  fa[x] = pre;
  dep[x] = dep[pre] + 1;
  mxs[x] = 1;
  eord[++idx] = x;
  if (fp[x] == 0) fp[x] = idx;
  for (auto& y : g[x]) {
    if (y == pre) continue;
    dfs(y, x);
    eord[++idx] = x;
    mxs[x] = max(mxs[x], mxs[y] + 1);
  }
}
int f[N << 1][35];
void init() {
  for (int i = 1; i <= idx; ++i) f[i][0] = eord[i];
  for (int len = 1; len < 30; ++len) {
    for (int l = 1; l <= idx; ++l) {
      int r = l + (1ll << len) - 1;
      if (r > idx) break;
      f[l][len] =
          min(f[l][len - 1], f[l + (1ll << len - 1)][len - 1],
              [&](const int& x, const int& y) { return dep[x] < dep[y]; });
    }
  }
}
int lca(int x, int y) {
  int L = fp[x], R = fp[y];
  if (L > R) swap(L, R);
  int len = __lg(R - L + 1);
  // cout<<L<<" "<<R<<"\n" ;
  return min(f[L][len], f[R - (1 << len) + 1][len],
```

```
             [&](const int& x, const int& y) { return dep[x] < dep[y]; });
}
```

Chapter 3

数学

3.1. 卷积

SOSDP

$$G(K) = \sum_{L \subseteq K} F(L) \qquad (K \subseteq U)$$

求所有的 $G(K)$ 狄利克雷前缀和

$$G(n) = \sum_{d|n} F(d)$$

狄利克雷后缀和

$$G(d) = \sum_{d|n} F(n)$$

卷积

$$H(\mathcal{S}) = \sum_{\mathcal{I} \cup \mathcal{J} = \mathcal{S}} F(\mathcal{I}) G(\mathcal{J})$$

子集卷积

$$H_i = \sum_{jork=i,jandk=0} F_j G_k$$

差卷积

$$c_k = \sum_{i=0} a_i b_{i+k}$$

考虑翻转整个 a 数组, 有 $aa_{n-i} = a_i$, 就变成了和卷积，NTT/FTT即可。

```
template <class T>
struct Convo {
  // 快速幂
  long long qpow(long long a, long long b) const {
    long long res = 1;
    while (b) {
      if (b & 1) res = res * a % mod;
      a = a * a % mod;
      b >>= 1;
    }
    return res;
  }

  // SOS 前缀和
  void sos_prefix(vector<T>& f, int n) {
    int N = 1 << n;
    for (int i = 0; i < n; ++i) {
      for (int mask = 0; mask < N; ++mask) {
        if (mask & (1 << i)) {
          f[mask] = (f[mask] + f[mask ^ (1 << i)]) % mod;
        }
      }
    }
  }
  // SOS 逆
  void sos_inverse(vector<T>& f, int n) {
    int N = 1 << n;
```

```cpp
  for (int i = 0; i < n; ++i) {
    for (int mask = 0; mask < N; ++mask) {
      if (mask & (1 << i)) {
        f[mask] = (f[mask] - f[mask ^ (1 << i)] + mod) % mod;
      }
    }
  }
}
// SOS 后缀和
void sos_suffix(vector<T>& f, int n) {
  int N = 1 << n;
  for (int i = 0; i < n; ++i) {
    for (int mask = 0; mask < N; ++mask) {
      if (!(mask & (1 << i))) {
        f[mask] = (f[mask] + f[mask | (1 << i)]) % mod;
      }
    }
  }
}
// SOS 后缀逆
void sos_suffix_inverse(vector<T>& f, int n) {
  int N = 1 << n;
  for (int i = 0; i < n; ++i) {
    for (int mask = 0; mask < N; ++mask) {
      if (!(mask & (1 << i))) {
        f[mask] = (f[mask] - f[mask | (1 << i)] + mod) % mod;
      }
    }
  }
}

// 线性筛及 Möbius
vector<int> primes, mu;
vector<bool> is_comp;
void init_sieve(int N) {
  mu.assign(N + 1, 1);
  is_comp.assign(N + 1, false);
  for (int i = 2; i <= N; ++i) {
    if (!is_comp[i]) {
      primes.push_back(i);
      mu[i] = -1;
    }
    for (int p : primes) {
      if ((long long)i * p > N) break;
      is_comp[i * p] = true;
      mu[i * p] = (i % p == 0 ? 0 : -mu[i]);
      if (i % p == 0) break;
    }
  }
}
// Dirichlet 前缀和
void dirichlet_prefix(vector<T>& F, int n) {
  for (int p : primes) {
    if (p > n) break;
    for (int i = 1; i * p <= n; ++i) {
      F[i * p] = (F[i * p] + F[i]) % mod;
    }
  }
}
// Dirichlet 前缀逆
void dirichlet_prefix_inverse(vector<T>& G, int n) {
  for (int i = (int)primes.size() - 1; i >= 0; --i) {
    int p = primes[i];
    if (p > n) continue;
    for (int j = 1; j * p <= n; ++j) {
      G[j * p] = (G[j * p] - G[j] + mod) % mod;
    }
  }
}
// Dirichlet 后缀和
void dirichlet_suffix(vector<T>& F, int n) {
  for (int p : primes) {
    if (p > n) break;
    for (int i = n / p; i >= 1; --i) {
      F[i] = (F[i] + F[i * p]) % mod;
    }
  }
}
// Dirichlet 后缀逆
void dirichlet_suffix_inverse(vector<T>& H, int n) {
  for (int i = (int)primes.size() - 1; i >= 0; --i) {
    int p = primes[i];
```

```cpp
      if (p > n) continue;
      for (int j = n / p; j >= 1; --j) {
        H[j] = (H[j] - H[j * p] + mod) % mod;
      }
    }
  }

  // OR 卷积
  void OR(vector<T>& F, vector<T>& G, int n) {
    sos_prefix(F, n);
    sos_prefix(G, n);
    int N = 1 << n;
    for (int i = 0; i < N; ++i) F[i] = (long long)F[i] * G[i] % mod;
    sos_inverse(F, n);
  }
  // AND 卷积
  void AND(vector<T>& F, vector<T>& G, int n) {
    sos_suffix(F, n);
    sos_suffix(G, n);
    int N = 1 << n;
    for (int i = 0; i < N; ++i) F[i] = (long long)F[i] * G[i] % mod;
    sos_suffix_inverse(F, n);
  }
  // GCD 卷积
  void GCD(vector<T>& F, vector<T>& G, int n) {
    dirichlet_suffix(F, n);
    dirichlet_suffix(G, n);
    for (int i = 1; i <= n; ++i) F[i] = (long long)F[i] * G[i] % mod;
    dirichlet_suffix_inverse(F, n);
  }
  // LCM 卷积
  void LCM(vector<T>& F, vector<T>& G, int n) {
    dirichlet_prefix(F, n);
    dirichlet_prefix(G, n);
    for (int i = 1; i <= n; ++i) F[i] = (long long)F[i] * G[i] % mod;
    dirichlet_prefix_inverse(F, n);
  }
  // 子集卷积
  void SUBSET(vector<T>& A, vector<T>& B, int n) {
    int N = 1 << n;
    vector<vector<T>> f(n + 1, vector<T>(N)), g(n + 1, vector<T>(N));
    for (int mask = 0; mask < N; ++mask) {
      int pc = __builtin_popcount(mask);
      f[pc][mask] = A[mask];
      g[pc][mask] = B[mask];
    }
    for (int i = 0; i <= n; ++i) {
      sos_prefix(f[i], n);
      sos_prefix(g[i], n);
    }
    vector<vector<T>> h(n + 1, vector<T>(N));
    for (int mask = 0; mask < N; ++mask) {
      for (int i = 0; i <= n; ++i) {
        long long sum = 0;
        for (int j = 0; j <= i; ++j) {
          sum += (long long)f[j][mask] * g[i - j][mask] % mod;
        }
        h[i][mask] = sum % mod;
      }
    }
    for (int i = 0; i <= n; ++i) sos_inverse(h[i], n);
    for (int mask = 0; mask < N; ++mask)
      A[mask] = h[__builtin_popcount(mask)][mask];
  }

  // FWT 异或卷积
  void FWT(vector<T>& F, int n, bool inverse = false) {
    int N = 1 << n;
    for (int len = 1; len < N; len <<= 1) {
      for (int i = 0; i < N; i += len << 1) {
        for (int j = 0; j < len; ++j) {
          T u = F[i + j], v = F[i + j + len];
          F[i + j] = (u + v) % mod;
          F[i + j + len] = (u - v + mod) % mod;
        }
      }
    }
    if (inverse) {
      long long inv = qpow(N, mod - 2);
      for (int i = 0; i < N; ++i) F[i] = (long long)F[i] * inv % mod;
    }
  }
```

```cpp
  // 异或卷积接口
  void XOR(vector<T>& F, vector<T>& G, int n) {
    FWT(F, n, false);
    FWT(G, n, false);
    int N = 1 << n;
    for (int i = 0; i < N; ++i) F[i] = (long long)F[i] * G[i] % mod;
    FWT(F, n, true);
  }
};
```

3.2. 复数域高斯消元

```cpp
#include <bits/stdc++.h>
using cd = complex<long double>;
cd ar[10][10];
const double eps = 1e-8;
int gauss(int n) {
  int c, r;
  for (c = 0, r = 0; c < n; ++c) {
    int t = r;
    for (int i = r; i < n; ++i) {
      if (std::abs(ar[i][c]) > std::abs(ar[t][c])) t = i;
    }
    if (std::abs(ar[t][c]) < eps) continue;
    for (int j = c; j < n + 1; ++j) std::swap(ar[t][j], ar[r][j]);
    for (int j = n; j >= c; --j) ar[r][j] /= ar[r][c];
    for (int i = r + 1; i < n; ++i) {
      if (std::abs(ar[i][c]) > eps) {
        for (int j = n; j >= c; --j) {
          ar[i][j] -= ar[r][j] * ar[i][c];
        }
      }
    }
    r++;
  }
  if (r < n) {
    for (int i = r; i < n; ++i) {
      bool allZero = true;
      for (int j = 0; j < n; ++j) {
        if (std::abs(ar[i][j]) > eps) {
          allZero = false;
          break;
        }
      }
      if (allZero && std::abs(ar[i][n]) > eps) return 2;
    }
    return 1;
  }
  for (int i = n - 1; i >= 0; --i) {
    for (int j = i + 1; j < n; ++j) {
      ar[i][n] -= ar[i][j] * ar[j][n];
    }
  }
  return 0;
}
```

3.3. 积和式

一个 $n \times n$ 矩阵 $A = (a_{i,j})$ 的积和式定义为

$$\operatorname{perm}(A) = \sum_{\sigma \in S_n} \prod_{i=1}^{n} a_{i,\sigma(i)}$$

也就是说，对所有排列 σ 把对应位置的乘积求和。
形式上与行列式相似，但行列式在每项前有置换符号 $\operatorname{sgn}(\sigma)$，而积和式不带符号。
因此两者在代数与计算性质上有明显不同（例如交换两行不改变积和式、但会改变行列式的符号）。积和式是多线性的，且对行（或列）置换不变。
对于二分图，把左、右两侧各 n 个顶点的邻接矩阵记为 A，则 $\operatorname{perm}(A)$ 等于该二分图中**完美匹配**的数目（每个排列对应一种匹配，非边对应项为 0）。
对有向图，邻接矩阵的积和式等于该图的**圈覆盖**（vertex-disjoint cycle cover）的数目。
2×2 矩阵 $\begin{pmatrix} a & b \\ c & d \end{pmatrix}$ 的积和式为 $ad + bc$。
全 1 的 $n \times n$ 矩阵的积和式就是棋盘上放置 n 个互不攻击的车（rook）的排列数（与置换的计数相关）。

特别地，对于二分图完美匹配计数问题，模2下其数目同余于行列式的值。

3.4. 多项式封装

一些常见模数:

$998244353 = 7 \times 17 \times 2^{23} + 1, g = 3$

$469762049 = 7 \times 2^{26} + 1, g = 3$

$1004535809 = 479 \times 2^{21} + 1, g = 3$

大模数:

$1,945,555,039,024,054,273 = 27 \times 2^{56} + 1, g = 5$

$4,179,340,454,199,820,289 = 29 \times 2^{57} + 1, g = 3$

```
#include <bits/stdc++.h>
using namespace std;

typedef long long LL;
template <unsigned umod>
struct modint {
  static constexpr int mod = umod;
  unsigned v;
  modint() : v(0) {}
  template <class T, enable_if_t<is_integral<T>::value>* = nullptr>
  modint(T x) {
    x %= mod;
    if (x < 0) x += mod;
    v = x;
  }
  modint(const string& str) {
    v = 0;
    size_t i = 0;
    if (str.front() == '-') i += 1;
    while (i < str.size()) {
      assert(isdigit(str[i]));
      v = (v * 10ull % umod + str[i] - '0') % umod;
      i += 1;
    }
    if (str.front() == '-' && v) v = umod - v;
  }
  modint operator+() const { return *this; }
  modint operator-() const { return modint() - *this; }
  friend int raw(const modint& self) { return self.v; }
  friend istream& operator>>(istream& is, modint& self) {
    string str;
    is >> str;
    self = str;
    return is;
  }
  friend ostream& operator<<(ostream& os, const modint& self) {
    return os << raw(self);
  }
  modint& operator+=(const modint& rhs) {
    v += rhs.v;
    if (v >= umod) v -= umod;
    return *this;
  }
  modint& operator-=(const modint& rhs) {
    v -= rhs.v;
    if (v >= umod) v += umod;
    return *this;
  }
  modint& operator*=(const modint& rhs) {
    v = static_cast<unsigned>(1ull * v * rhs.v % umod);
    return *this;
  }
  modint& operator/=(const modint& rhs) {
    static constexpr size_t ilim = 1 << 20;
    static modint inv[ilim + 10];
    static int sz = 0;
    assert(rhs.v);
    if (rhs.v > ilim) return *this *= qpow(rhs, mod - 2);
    if (!sz) inv[1] = sz = 1;
    while (sz < (int)rhs.v) {
      for (int i = sz + 1; i <= sz << 1; i++) inv[i] = -mod / i * inv[mod % i];
      sz <<= 1;
```

```cpp
    }
    return *this *= inv[rhs.v];
  }
  template <class T>
  friend modint qpow(modint a, T b) {
    modint r = 1;
    for (; b; b >>= 1, a *= a)
      if (b & 1) r *= a;
    return r;
  }
  friend modint operator+(modint lhs, const modint& rhs) { return lhs += rhs; }
  friend modint operator-(modint lhs, const modint& rhs) { return lhs -= rhs; }
  friend modint operator*(modint lhs, const modint& rhs) { return lhs *= rhs; }
  friend modint operator/(modint lhs, const modint& rhs) { return lhs /= rhs; }
  friend bool operator==(const modint& lhs, const modint& rhs) {
    return lhs.v == rhs.v;
  }
  friend bool operator!=(const modint& lhs, const modint& rhs) {
    return lhs.v != rhs.v;
  }
};

typedef modint<998244353> mint;
// 返回大于等于 x 的最小 2 的幂
int glim(const int& x) { return 1 << (32 - __builtin_clz(x - 1)); }
// 返回 x 尾部连续 0 的个数。
int bitctz(const int& x) { return __builtin_ctz(x); }
struct poly : vector<mint> {
  poly() {}
  explicit poly(int n) : vector<mint>(n) {}
  poly(const vector<mint>& vec) : vector<mint>(vec) {}
  // 列表初始化
  poly(initializer_list<mint> il) : vector<mint>(il) {}
  mint operator()(const mint& x) const;
  poly& cut(int lim);
  void ntt(int op);
};
// 输入多项式系数
istream& operator>>(istream& is, poly& a) {
  for (auto& x : a) is >> x;
  return is;
}
// 输出一个多项式
ostream& operator<<(ostream& os, const poly& a) {
  bool flag = false;
  for (auto& x : a) {
    if (flag)
      os << " ";
    else
      flag = true;
    os << x;
  }
  return os;
}
// 单点求值
mint poly::operator()(const mint& x) const {
  const auto& a = *this;
  mint res = 0;
  for (int i = (int)a.size() - 1; i >= 0; i--) {
    res = res * x + a[i];
  }
  return res;
}
// 截断到lim
poly& poly::cut(int lim) {
  resize(lim);
  return *this;
}
// 传入-1，逆变换。
void poly::ntt(int op) {
  static bool wns_flag = false;
  static vector<mint> wns;
  if (!wns_flag) {
    wns_flag = true;
    for (int j = 1; j <= 23; j++) {
      wns.push_back(qpow(mint(3), raw(mint(-1)) >> j));
    }
  }
  vector<mint>& a = *this;
  int n = a.size();
  for (int i = 1, r = 0; i < n; i++) {
    r ^= n - (1 << (bitctz(n) - bitctz(i) - 1));
```

```cpp
    if (i < r) std::swap(a[i], a[r]);
  }
  vector<mint> w(n);
  for (int k = 1, len = 2; len <= n; k <<= 1, len <<= 1) {
    mint wn = wns[bitctz(k)];
    for (int i = raw(w[0] = 1); i < k; i++) w[i] = w[i - 1] * wn;
    for (int i = 0; i < n; i += len) {
      for (int j = 0; j < k; j++) {
        mint x = a[i + j], y = a[i + j + k] * w[j];
        a[i + j] = x + y, a[i + j + k] = x - y;
      }
    }
  }
  if (op == -1) {
    mint iz = mint(1) / n;
    for (int i = 0; i < n; i++) a[i] *= iz;
    reverse(a.begin() + 1, a.end());
  }
}
// 牛顿迭代, vec是多项式, func是计算的函数
poly concalc(int n, vector<poly> vec,
             const function<mint(vector<mint>)>& func) {
  int lim = glim(n);
  int m = vec.size();
  for (auto& f : vec) f.resize(lim), f.ntt(1);
  vector<mint> tmp(m);
  poly ret(lim);
  for (int i = 0; i < lim; i++) {
    for (int j = 0; j < m; j++) tmp[j] = vec[j][i];
    ret[i] = func(tmp);
  }
  ret.ntt(-1);
  return ret;
}

poly getInv(const poly& a, int lim) {
  poly b{1 / a[0]};
  for (int len = 2; len <= glim(lim); len <<= 1) {
    poly c = vector<mint>(a.begin(), a.begin() + min(len, (int)a.size()));
    b = concalc(len << 1, {b, c}, [](vector<mint> vec) {
          return vec[0] * (2 - vec[0] * vec[1]);
        }).cut(len);
  }
  return b.cut(lim);
}

poly operator+=(poly& a, const poly& b) {
  if (a.size() < b.size()) a.resize(b.size());
  for (size_t i = 0; i < b.size(); i++) a[i] += b[i];
  return a;
}

poly operator-=(poly& a, const poly& b) {
  if (a.size() < b.size()) a.resize(b.size());
  for (size_t i = 0; i < b.size(); i++) a[i] -= b[i];
  return a;
}

poly operator*=(poly& a, const mint& k) {
  if (k == 1) return a;
  for (size_t i = 0; i < a.size(); i++) a[i] *= k;
  return a;
}
poly operator/=(poly& a, const mint& k) { return a *= 1 / k; }

poly operator<<=(poly& a, const int& k) {
  // mnltiple by x^k
  a.insert(a.begin(), k, 0);
  return a;
}

poly operator>>=(poly& a, const int& k) {
  // divide by x^k
  a.erase(a.begin(), a.begin() + min(k, (int)a.size()));
  return a;
}
poly operator*(const poly& a, const poly& b) {
  if (a.empty() || b.empty()) return {};
  int rlen = a.size() + b.size() - 1;
  int len = glim(rlen);
  if (1ull * a.size() * b.size() <= 1ull * len * bitctz(len)) {
    poly ret(rlen);
```

```
    for (size_t i = 0; i < a.size(); i++)
      for (size_t j = 0; j < b.size(); j++) ret[i + j] += a[i] * b[j];
    return ret;
  } else {
    return concalc(len, {a, b},
                   [](vector<mint> vec) { return vec[0] * vec[1]; })
        .cut(rlen);
  }
}
poly operator/(poly a, poly b) {
  if (a.size() < b.size()) return {};
  int rlen = a.size() - b.size() + 1;
  reverse(a.begin(), a.end());
  reverse(b.begin(), b.end());
  a = (a * getInv(b, rlen)).cut(rlen);
  reverse(a.begin(), a.end());
  return a;
}
poly operator-(poly a, const poly& b) { return a -= b; }
poly operator%(const poly& a, const poly& b) {
  return (a - (a / b) * b).cut(b.size() - 1);
}
poly operator*=(poly& a, const poly& b) { return a = a * b; }
poly operator/=(poly& a, const poly& b) { return a = a / b; }
poly operator%=(poly& a, const poly& b) { return a = a % b; }
poly operator+(poly a, const poly& b) { return a += b; }
poly operator*(poly a, const mint& k) { return a *= k; }
poly operator*(const mint& k, poly a) { return a *= k; }
poly operator/(poly a, const mint& k) { return a /= k; }
poly operator<<(poly a, const int& k) { return a <<= k; }
poly operator>>(poly a, const int& k) { return a >>= k; }
// 形式导数
poly getDev(poly a) {
  a >>= 1;
  for (size_t i = 1; i < a.size(); i++) a[i] *= i + 1;
  return a;
}
// 不定积分
poly getInt(poly a) {
  a <<= 1;
  for (size_t i = 1; i < a.size(); i++) a[i] /= i;
  return a;
}
// 对数函数
poly getLn(const poly& a, int lim) {
  assert(a[0] == 1);
  return getInt(getDev(a) * getInv(a, lim)).cut(lim);
}
// 指数函数
poly getExp(const poly& a, int lim) {
  assert(a[0] == 0);
  poly b{1};
  for (int len = 2; len <= glim(lim); len <<= 1) {
    poly c = vector<mint>(a.begin(), a.begin() + min(len, (int)a.size()));
    b = concalc(len << 1, {b, getLn(b, len), c}, [](vector<mint> vec) {
          return vec[0] * (1 - vec[1] + vec[2]);
        }).cut(len);
  }
  return b.cut(lim);
}
// 快速幂
poly qpow(const poly& a, string k, int lim) {
  size_t i = 0;
  while (i < a.size() && a[i] == 0) i += 1;
  if (i == a.size() || (i > 0 && k.size() >= 9) ||
      1ull * i * raw(mint(k)) >= 1ull * lim)
    return poly(lim);
  lim -= i * raw(mint(k));
  return getExp(getLn(a / a[i] >> i, lim) * k, lim) *
             qpow(a[i], raw(modint<mint::mod - 1>(k)))
         << i * raw(mint(k));
}
poly qpow(const poly& a, LL k, int lim) {
  size_t i = 0;
  while (i < a.size() && a[i] == 0) i += 1;
  if (i == a.size() || (i > 0 && k >= 1e9) || 1ull * i * k >= 1ull * lim)
    return poly(lim);
  lim -= i * k;
  return getExp(getLn(a / a[i] >> i, lim) * k, lim) *
             qpow(a[i], raw(modint<mint::mod - 1>(k)))
         << i * k;
}
```

```cpp

mint sqrt(const mint& c) {
  static const auto check = [](mint c) {
    return qpow(c, (mint::mod - 1) >> 1) == 1;
  };
  if (raw(c) <= 1) return 1;
  if (!check(c)) throw "No solution!";
  static mt19937 rng{random_device{}()};
  mint a = rng();
  while (check(a * a - c)) a = rng();
  typedef pair<mint, mint> number;
  const auto mul = [=](number x, number y) {
    return make_pair(x.first * y.first + x.second * y.second * (a * a - c),
                     x.first * y.second + x.second * y.first);
  };
  const auto qpow = [=](number a, int b) {
    number r = {1, 0};
    for (; b; b >>= 1, a = mul(a, a))
      if (b & 1) r = mul(r, a);
    return r;
  };
  mint ret = qpow({a, 1}, (mint::mod + 1) >> 1).first;
  return min(raw(ret), raw(-ret));
}
// 开根号
poly getSqrt(const poly& a, int lim) {
  poly b{sqrt(a[0])};
  for (int len = 2; len <= glim(lim); len <<= 1) {
    poly c = vector<mint>(a.begin(), a.begin() + min(len, (int)a.size()));
    b = (c * getInv(b * 2, len) + b / 2).cut(len);
  }
  return b.cut(lim);
}
template <class T>
mint divide_at(poly f, poly g, T n) {
  for (; n; n >>= 1) {
    poly r = g;
    for (size_t i = 1; i < r.size(); i += 2) r[i] *= -1;
    f *= r;
    g *= r;
    int i;
    for (i = n & 1; i < (int)f.size(); i += 2) f[i >> 1] = f[i];
    f.resize(i >> 1);
    for (i = 0; i < (int)g.size(); i += 2) g[i >> 1] = g[i];
    g.resize(i >> 1);
  }
  return f.empty() ? 0 : f[0] / g[0];
}

template <class T>
mint linear_rec(poly a, poly f, T n) {
  // a[n] = sum_i f[i] * a[n - i]
  a.resize(f.size() - 1);
  f = poly{1} - f;
  poly g = a * f;
  g.resize(a.size());
  return divide_at(g, f, n);
}
poly BM(poly a) {
  poly ans, lst;
  int w = 0;
  mint delta = 0;
  for (size_t i = 0; i < a.size(); i++) {
    mint tmp = -a[i];
    for (size_t j = 0; j < ans.size(); j++) tmp += ans[j] * a[i - j - 1];
    if (tmp == 0) continue;
    if (ans.empty()) {
      w = i;
      delta = tmp;
      ans = vector<mint>(i + 1, 0);
    } else {
      auto now = ans;
      mint mul = -tmp / delta;
      if (ans.size() < lst.size() + i - w) ans.resize(lst.size() + i - w);
      ans[i - w - 1] -= mul;
      for (size_t j = 0; j < lst.size(); j++) ans[i - w + j] += lst[j] * mul;
      if (now.size() <= lst.size() + i - w) {
        w = i;
        lst = now;
        delta = tmp;
      }
    }
```

```cpp
  }
  return ans << 1;
}

poly lagrange(const vector<pair<mint, mint>>& a) {
  poly ans(a.size()), product{1};
  for (size_t i = 0; i < a.size(); i++) {
    product *= poly{-a[i].first, 1};
  }
  auto divide2 = [&](poly a, mint b) {
    poly res(a.size() - 1);
    for (size_t i = (int)a.size() - 1; i >= 1; i--) {
      res[i - 1] = a[i];
      a[i - 1] -= a[i] * b;
    }
    return res;
  };
  for (size_t i = 0; i < a.size(); i++) {
    mint denos = 1;
    for (size_t j = 0; j < a.size(); j++) {
      if (i != j) denos *= a[i].first - a[j].first;
    }
    poly numes = divide2(product, -a[i].first);
    ans += a[i].second / denos * numes;
  }
  return ans;
}
```

3.5. 异或线性基

时间复杂度 $O(n \log \max a)$ 异或问题, 同时又可以找到“子集”“子序列”等字眼，或者是图论中的某条路径的异或和时，就可以往线性基方向想了。我们可以利用异或线性基实现:

1. 判断一个数能否表示成某数集子集的异或和;
2. 求一个数表示成某数集子集异或和的方案数;
3. 求某数集子集的最大/最小/第 k 大/第 k 小异或和；(注意01Tire是求点对区间的第k大异或对)
4. 求一个数在某数集子集异或和中的排名。

```cpp
/*异或线性基
 *求数集子集异或和第k小 (k从0开始计数)
 */
struct Basis {
    vector<u64> B;
    int cntz=0;//0的个数
    bool ok=false;
    void insert(u64 x) {
      for (auto b:B) x=min(x,x^b);
      for (auto& b: B) b=min(b,b^x);
      if (x) B.push_back(x);
      else cntz++;
    }
    void _min() {
      sort(B.begin(),B.end());
      for (int i=1;i<B.size();++i) {
        for (int j=i-1;j>=0;--j) {
          B[i]=min(B[i],B[i]^B[j]);
        }
      }
    }
    //第k小的异或和
    u64 query(int k,bool overphi) {//第k小, 包含空集?(k从0开始数)
      if (!ok)_min(),ok=true;
      if (!overphi and !cntz) k++;
      if (k>=(1ll<<(B.size()))) return -1;
      int res=0;
      for (int i=0;i<B.size();++i) {
        if ((k>>i)&1) res^=B[i];
      }
      return res;
    }

    u64 querymx() {
      return query((1ll<<B.size())-1,1);
    }

    void print() {
      for (int i=0;i<B.size();++i) cout<<B[i]<<" ";
      cout<<"\n";
    }

```

```
    //线性基的合并（双log）
    void operator+=(Basis& _B) {
      for (auto &b:_B.B) this->insert(b);
    }
    friend Basis operator+(Basis& b1,Basis& b2) {
      Basis res=b1;
      for (auto& b:b2.B)res.insert(b);
      return res;
    }
};
```

模板题: 最大异或和:https://www.luogu.com.cn/record/204660302

Chapter 4

数据结构

4.1. 线段树二分

```
/*
线段树（LazySegmentTree）
左闭右开
*/
template <class Info, class Tag>
struct LazySegmentTree {
  int n;  // n+1
  vector<Info> info;
  vector<Tag> tag;
  // init begin
  LazySegmentTree() : n(0) {}
  LazySegmentTree(int n_, Info v_ = Info()) {
    init(n_ + 1, v_);  // 下标从1开始
  }
  template <class T>
  LazySegmentTree(vector<T> init_) {
    init(init_);
  }
  void init(int n_, Info v_ = Info()) { init(vector(n_, v_)); }
  template <class T>
  void init(vector<T> init_) {
    n = init_.size();
    info.assign(4 << __lg(n), Info());
    tag.assign(4 << __lg(n), Tag());
    std::function<void(int, int, int)> build = [&](int p, int l, int r) {
      if (r - l == 1) {
        info[p] = init_[l];
        return;
      }
      int m = (l + r) / 2;
      build(2 * p, l, m);
      build(2 * p + 1, m, r);
      pull(p);
    };
    build(1, 1, n);
  }
  // init end
  //  up
  void pull(int p) { info[p] = info[2 * p] + info[2 * p + 1]; }
  // 修改
  void apply(int p, const Tag &v, int len) {
    info[p].apply(v, len);
    tag[p].apply(v);
  }
  // down
  void push(int p, int len) {
    apply(2 * p, tag[p], len / 2);
    apply(2 * p + 1, tag[p], len - len / 2);
    tag[p] = Tag();
  }
  // 单点修改
  void modify(int p, int l, int r, int x, const Info &v) {
    if (r - l == 1) {
      info[p] = v;
      return;
    }
    int m = (l + r) / 2;
    push(p, r - l);
    if (x < m) {
      modify(2 * p, l, m, x, v);
```

```
  } else {
    modify(2 * p + 1, m, r, x, v);
  }
  pull(p);
}
void modify(int p, const Info &v) { modify(1, 1, n, p, v); }
// 区间查询
Info rangeQuery(int p, int l, int r, int x, int y) {
  if (l >= y || r <= x) {
    return Info();
  }
  if (l >= x && r <= y) {
    return info[p];
  }
  int m = (l + r) / 2;
  push(p, r - l);
  return rangeQuery(2 * p, l, m, x, y) + rangeQuery(2 * p + 1, m, r, x, y);
}
Info rangeQuery(int l, int r) { return rangeQuery(1, 1, n, l, r); }

// 区间修改
void rangeApply(int p, int l, int r, int x, int y, const Tag &v) {
  if (l >= y || r <= x) {
    return;
  }
  if (l >= x && r <= y) {
    apply(p, v, r - l);
    return;
  }
  int m = (l + r) / 2;
  push(p, r - l);
  rangeApply(2 * p, l, m, x, y, v);
  rangeApply(2 * p + 1, m, r, x, y, v);
  pull(p);
}
void rangeApply(int l, int r, const Tag &v) {
  return rangeApply(1, 1, n, l, r, v);
}
//线段树二分
template <class F>
int findFirst(int p, int l, int r, int x, int y, F &&pred) {
  if (l >= y || r <= x) {
    return -1;
  }
  if (l >= x && r <= y && !pred(info[p])) {
    return -1;
  }
  if (r - l == 1) {
    return l;
  }
  int m = (l + r) / 2;
  push(p,r-l);
  int res = findFirst(2 * p, l, m, x, y, pred);
  if (res == -1) {
    res = findFirst(2 * p + 1, m, r, x, y, pred);
  }
  return res;
}
//第一个满足条件F的位置
template <class F>
int findFirst(int l, int r, F &&pred) {
  return findFirst(1, 1, n, l, r, pred);
}
template <class F>
int findLast(int p, int l, int r, int x, int y, F &&pred) {
  if (l >= y || r <= x) {
    return -1;
  }
  if (l >= x && r <= y && !pred(info[p])) {
    return -1;
  }
  if (r - l == 1) {
    return l;
  }
  int m = (l + r) / 2;
  push(p,r-l);
  int res = findLast(2 * p + 1, m, r, x, y, pred);
  if (res == -1) {
    res = findLast(2 * p, l, m, x, y, pred);
  }
  return res;
}
```

```cpp
  //最后一个满足条件F的位置
  template <class F>
  int findLast(int l, int r, F &&pred) {
    return findLast(1, 1, n, l, r, pred);
  }
};

struct Tag {
  int x = 0;
  void apply(const Tag &t) & { x += t.x; }
};

struct Info {
  int sum = 0,mx=-iinf,mi=iinf;
  void apply(const Tag &t, int len) & {
    sum += t.x * len;
    mx +=t.x;
    mi +=t.x;
  }
};
// merge
Info operator+(const Info &a, const Info &b) {
  Info res={};
  res.sum=a.sum+b.sum;
  res.mx=max(a.mx,b.mx);
  res.mi=min(a.mi,b.mi);
  return res;
}
```

4.2. 线段树维护区间gcd

$$\gcd_{i=l}^{r} = \gcd(a_l, \gcd_{i=l+1}^{r}(a[i] - a[i-1]))$$

这意味着我们无须维护区间加，只要做差分数组并维护单点加就可以了。

```cpp
int mygcd(int a,int b) {
  return __gcd(abs(a),abs(b));
}
template<class T>
struct Segt {
  struct node {
    int l,r;
    T w;// gcd
    T sum;
  };
  vector<T> w;
  vector<node> t;
  Segt(){}
  Segt(int n) {init(n);}
  Segt(vector<int> in) {
    int n=in.size()-1;
    w.resize(n+1);
    for (int i=1;i<=n;++i) {
      w[i]=in[i];
    }
    init(in.size()-1);
  }
#define GL k<<1
#define GR k<<1|1
  void init(int n) {
    t.resize(4*n +1);
    auto build=[&](auto self ,int l, int r,int k=1) {
      if (l==r) {
        t[k]={l,r,w[l],w[l]};
        return ;
      }
      t[k]={l,r,0,0};
      int mid=(l+r)/2;
      self(self,l,mid,GL);
      self(self,mid+1,r,GR);
      pushup(k);
    };
    build(build,1,n);
  }
  void pushup(int k) {
    auto pushup=[&](node& p,node& l, node &r) {
      p.w=mygcd(l.w,r.w);
      p.sum=l.sum+r.sum;
```

```
    };
    pushup(t[k],t[GL],t[GR]);
  }
  void add(int pos,T val,int k=1) {
    if (t[k].l==t[k].r) {
      t[k].w+=val;
      t[k].sum+=val;
      return ;
    }
    int mid=(t[k].l+t[k].r)/2;
    if (pos<=mid) add(pos,val,GL);
    else add(pos,val,GR);
    pushup(k);
  }
  // 单点赋值, 不用管sum
  void upd(int pos,T val,int k=1) {
    if (t[k].l==t[k].r) {
      t[k].w=val;
      return ;
    }
    int mid=(t[k].l+t[k].r)/2;
    if (pos<=mid) upd(pos,val,GL);
    else upd(pos,val,GR);
    pushup(k);
  }
  T askgcd(int l,int r,int k=1) {
    if (l<=t[k].l and t[k].r<=r) return t[k].w;
    int mid=(t[k].l+t[k].r)/2;
    T ans=0;
    if (l<=mid) ans=mygcd(ans,askgcd(l,r,GL));
    if (mid<r)  ans=mygcd(ans,askgcd(l,r,GR));
    return ans;
  }
  T asksum(int l,int r,int k=1) {
    if (l<=t[k].l and t[k].r<=r) return t[k].sum;
    int mid=(t[k].l+t[k].r)/2;
    T ans=0;
    if (l<=mid) ans+=asksum(l,r,GL);
    if (mid<r)  ans+=asksum(l,r,GR);
    return ans;
  }
};

void solve() {
  int n,m;cin>>n>>m;
  vector<int> a(n+1);
  for (int i=1;i<=n;++i) cin>>a[i];
  for (int i=n;i;--i) a[i]-=a[i-1];
  Segt<int> sgt(a);
  for (int i=1;i<=m;++i) {
    char op;cin>>op;
    if (op=='C') {// 区间修改
      int l,r,d;cin>>l>>r>>d;
      sgt.add(l,d);
      if (r<n) sgt.add(r+1,-d);
    }else {//区间查询
      int l,r;cin>>l>>r;
      cout<<mygcd(sgt.asksum(1,l),sgt.askgcd(l+1,r))<<"\n";
    }
  }
}
```

4.3. 对顶堆

```
using namespace std;
struct Maxheap {
  int n;
  vector<int> w;
  Maxheap(auto &_init):w(_init) {
    n=static_cast<int>(_init.size())-1;
    w.resize(n+1);
    for (int i=1;i<=n;++i) up(i);
  }
  void up(int x) {
    while (x>1 and w[x]>w[x/2]) swap(w[x],w[x/2]),x/=2;
  }
  void down(int x) {
    while (x*2<=n) {
      int t=x*2;
      if (t+1<=n and w[t+1]>w[t]) t++;
      if (w[x]>=w[t]) return ;
```

```
      swap(w[x],w[t]);
      x=t;
    }
  }
};
```

对顶堆可以动态维护一个序列上的中位数, 或者第k大的数,(k的值可能变化).
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。
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;
变化k：根据新的 k 值直接维护对顶堆。
时间复杂度 $O(\log n)$

```
#include <bits/stdc++.h>
using namespace std;
struct mset {
  const int kInf = 1e9 + 2077;
  multiset<int> less, greater;
  void init() {
    less.clear(), greater.clear();
    less.insert(-kInf), greater.insert(kInf);
  }
  void adjust() {
    while (less.size() > greater.size() + 1) {
      multiset<int>::iterator it = (--less.end());
      greater.insert(*it);
      less.erase(it);
    }
    while (greater.size() > less.size()) {
      multiset<int>::iterator it = greater.begin();
      less.insert(*it);
      greater.erase(it);
    }
  }
  void add(int val_) {
    if (val_ <= *greater.begin())
      less.insert(val_);
    else
      greater.insert(val_);
    adjust();
  }
  void del(int val_) {
    multiset<int>::iterator it = less.lower_bound(val_);
    if (it != less.end()) {
      less.erase(it);
    } else {
      it = greater.lower_bound(val_);
      greater.erase(it);
    }
    adjust();
  }
  int get_middle() { return *less.rbegin(); }
};
```

4.4. 手写Bitset

```
u64 mi[200];
// for (int i = 0; i < 64; i++) mi[i] = (1ULL << i);
struct Bit {
  vector<u64> bit;
  int len;

  Bit(int sz = 0) {
    len = (sz + 63) >> 6;
    bit.assign(len, 0);
  }

#define I inline
  I void reset() { fill(bit.begin(), bit.end(), 0); }
  Bit() { fill(bit.begin(), bit.end(), 0); }
  I void set1(int x) { bit[x >> 6] |= mi[x & 63]; }
  I void set0(int x) { bit[x >> 6] &= ~mi[x & 63]; }
  I void flip(int x) { bit[x >> 6] ^= mi[x & 63]; }
  bool operator[](int x) { return (bit[x >> 6] >> (x & 63)) & 1; }
#define re register
```

```cpp
  Bit operator~(void) const {
    Bit res;
    for (re int i = 0; i < len; i++) res.bit[i] = ~bit[i];
    return res;
  }
  Bit operator&(const Bit &b) const {
    Bit res;
    for (re int i = 0; i < len; i++) res.bit[i] = bit[i] & b.bit[i];
    return res;
  }
  Bit operator|(const Bit &b) const {
    Bit res;
    for (re int i = 0; i < len; i++) res.bit[i] = bit[i] | b.bit[i];
    return res;
  }
  Bit operator^(const Bit &b) const {
    Bit res;
    for (re int i = 0; i < len; i++) res.bit[i] = bit[i] ^ b.bit[i];
    return res;
  }
  void operator&=(const Bit &b) {
    for (re int i = 0; i < len; i++) bit[i] &= b.bit[i];
  }
  void operator|=(const Bit &b) {
    for (re int i = 0; i < len; i++) bit[i] |= b.bit[i];
  }
  void operator^=(const Bit &b) {
    for (re int i = 0; i < len; i++) bit[i] ^= b.bit[i];
  }
  Bit operator<<(const int t) const {
    Bit res;
    int high = t >> 6, low = t & 63;
    u64 last = 0;
    for (register int i = 0; i + high < len; i++) {
      res.bit[i + high] = (last | (bit[i] << low));
      if (low) last = (bit[i] >> (64 - low));
    }
    return res;
  }
  Bit operator>>(const int t) const {
    Bit res;
    int high = t >> 6, low = t & 63;
    u64 last = 0;
    for (register int i = len - 1; i >= high; i--) {
      res.bit[i - high] = last | (bit[i] >> low);
      if (low) last = bit[i] << (64 - low);
    }
    return res;
  }
  void operator<<=(const int t) {
    int high = t >> 6, low = t & 63;
    for (register int i = len - high - 1; ~i; i--) {
      bit[i + high] = (bit[i] << low);
      if (low && i) bit[i + high] |= bit[i - 1] >> (64 - low);
    }
    for (register int i = 0; i < high; i++) bit[i] = 0;
  }
};
```

4.5. 笛卡尔树

常用于数数题。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。
k 有序的话, 可以线性建树。

```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;
}
```

性质
1. 以u为根的子树是一段连续的区间（由BST性质），且u是这段区间的最小值，且不能再向两端延伸使得最小值不变（即，这一段区间是极长的）
2. 在 u 左右子树里任选两个点，两点间的区间最小值必定是 w_u

3. a,b间的区间最小值为：$w_{LCA(a,b)}$

Chapter 5

trick

5.1. 枚举子集

用于循环枚举子集，**注意枚举不了空集**

```
for(int j=st;j;j=(j-1)&st)
```

st，为要枚举子集的集合，j为子集
本质是将每一位设为0，1后枚举后面的。
时间复杂度 $O(3^n)$

5.2. 求所有因数

利用类似筛法的原理

```
 for (int i = 1; i <= MX; ++i) {
  for (int j = i; j <= MX; j += i) {
    d[j].push_back(i);
  }
}
```

时间复杂度 $O(n \log n)$

Chapter 6

杂项

6.1. 格雷码

构造格雷码

$$G(n) = n \oplus \left\lfloor \frac{n}{2} \right\rfloor$$

格雷码构造原数

$$n_{k-i} = \bigoplus_{0 \le j \le i} g_{k-j}$$

6.2. pbds

```
#include<ext/pb_ds/assoc_container.hpp>
#include<ext/pb_ds/tree_policy.hpp>//用tree
#include<ext/pb_ds/hash_policy.hpp>//用hash
#include<ext/pb_ds/trie_policy.hpp>//用trie
#include<ext/pb_ds/priority_queue.hpp>//用priority_queue
```

6.3. 哈希表

```
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/hash_policy.hpp>
const int RANDOM = time(NULL);
struct MyHash {int operator() (int x) const {return x ^ RANDOM;}};
template <class T1, class T2>
struct std::tr1::hash <std::pair <T1, T2> > {
  size_t operator() (std::pair <T1, T2> x) const {
    std::tr1::hash <T1> H1; std::tr1::hash <T2> H2;
    return H1(x.first) ^ H2(x.second); // 你自定义的 hash 函数。
  }
};
__gnu_pbds::gp_hash_table <std::pair <int, int>, int> Table;
```

直接当没有 `emplace()`,`cbegin()`,`cend()` `unordered_map` 用就好了。

Chapter 7

动态规划

7.1. 树上背包

7.1.1 dfs序优化

```cpp
#include <bits/stdc++.h>
using namespace std;
int n, m, v[100010];
vector<int> G[100010], f[100010];
int siz[100010], awa[100010], len;
void dfs(int pos) {
  siz[pos] = 1;
  for (auto i : G[pos]) {
    dfs(i);
    siz[pos] += siz[i];
  }
  awa[++len] = pos;
}
// 前i个点，代价j，最大价值。
int main() {
  cin >> n >> m;
  for (int i = 1; i <= n; i++) {
    int u;
    cin >> u >> v[i];
    G[u].emplace_back(i);
  }
  dfs(0);
  f[0].resize(m + 1);
  for (int i = 1; i <= n; i++) {
    f[i].resize(m + 1);
    for (int j = 1; j <= m; j++) {
      f[i][j] = max(f[i - 1][j - 1] + v[awa[i]], f[i - siz[awa[i]]][j]);
    }
  }
  cout << f[n][m];
  return 0;
}
```

7.1.2 多叉转二叉优化

具体方法就是不断递归，找到根节点，把它与最左边的子节点之间的边保留，其他全部断掉。然后从这个保留的孩子开始，连一条链到所有断开的点。

```cpp
#include <bits/stdc++.h>
#define NO 100009
using namespace std;
int n, m, v[100010], lc[100010], rc[100010];
int siz[100010];
vector<int> G[100010], f[100010];
void dfs(int pos) {
  if (pos == NO) return;
  dfs(lc[pos]);
  dfs(rc[pos]);
  for (int i = 1; i <= min(m, siz[pos]); i++) {
    f[pos][i] = f[rc[pos]][i];
    // 不需要  ...][i]  =>  ...][min(i,siz[rc[pos]])]
    // 原因是如果i>siz[rc[pos]]，就说明把右节点分配满，
    // 也有剩余的课程可以加到pos和pos的左子节点
    // 这个语句就相当于没用了
    int lj, rj;
    rj = min(i - 1, siz[lc[pos]]);
    // 左节点最多分配siz[lc[pos]] 个
    lj = max(0, i - 1 - siz[rc[pos]]);
    // 右节点最多分配siz[rc[pos]] 个
```

```
    // 而右节点个数是i-1-j,
    // 所以j最大枚举到i-1-siz[rc[pos]]
    for (int j = lj; j <= rj; j++) {
      // 左 j个    右 i-1-j个
      int l = f[lc[pos]][j];
      int r = f[rc[pos]][i - 1 - j];
      f[pos][i] = max(f[pos][i], l + r + v[pos]);
    }
  }
}
void conv(int pos) {
  siz[pos] = 1;
  int prei = -1;
  for (auto i : G[pos]) {
    if (prei == -1)
      lc[pos] = i;
    else
      rc[prei] = i;
    prei = i;
    conv(i);
  }
}
void cntsiz(int pos) {
  if (pos == NO) return;
  cntsiz(lc[pos]);
  cntsiz(rc[pos]);
  siz[pos] = 1 + siz[lc[pos]] + siz[rc[pos]];
}
int main() {
  cin >> n >> m;
  m++;
  for (int i = 1; i <= n; i++) {
    int u;
    cin >> u >> v[i];
    G[u].emplace_back(i);
  }
  for (int i = 0; i <= n; i++) f[i].resize(m + 1), lc[i] = rc[i] = NO;
  f[NO].resize(m + 1);
  conv(0);    // 转二叉
  cntsiz(0);  // 计算大小
  dfs(0);
  cout << f[0][m];
  return 0;
}
```

7.2. 最长单调子序列

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

int main() {
  int n;
  vector<int> a(n + 1);
  vector<int> stk(n + 10);
  int top = 0, ans = 0;
  vector<int> pre(n + 1);
  // 最长不上升子序列
  for (int i = 1; i <= n; ++i) {
    auto pos = lower_bound(stk.begin() + 1, stk.begin() + 1 + top, a[i],
                           [&](int u, int v) { return u >= v; }) -
               stk.begin();
    if (pos > top) top++;
    stk[pos] = a[i];
    ans = max(ans, top);
    // 序列恢复
    if (top > 1) pre[i] = stk[top - 1];
  }
  cout << ans << "\n";
  top = 0, ans = 0;
  // 最长上升子序列
  for (int i = 1; i <= n; ++i) {
    auto pos =
        lower_bound(stk.begin() + 1, stk.begin() + 1 + top, a[i]) - stk.begin();
    if (pos > top) top++;
    stk[pos] = a[i];
    ans = max(ans, top);
  }
}
```

Chapter 8

图论

8.1. 最小生成树

8.1.1 kruskal

思路：将所有边按权重升序，依次考虑，若连接不同分量则加入（用并查集判定/合并）。
适用场景：稀疏图

8.1.2 prim

Prim 从某个起点开始，把当前生成树与外部的所有边中权重最小的那条边加入生成树，重复直到所有顶点都被包含。等价地，维护每个未加入顶点到已加入顶点集合的最短边（关键值），每次选关键值最小的顶点加入并更新相邻顶点的关键值。
对密集图 (EV^2) 邻接矩阵实现优于 Kruskal。
易于在线/增量场景（可以边读取边扩展树）。
适用场景：密集图、需要快速局部扩展或图以邻接矩阵存储、在线/流式构建 MST 情况。

8.1.3 boruvka

boruvka算法流程：
1. **初始化**：每个节点自成一个连通分量。
2. **并行探索**：每一轮迭代下，对每个连通分量，找到其连接外界的**最小权重边**（类似 Prim 的切割性质）。
3. **批量合并**：将所有找到的最小边加入 MST，合并连通分量。
4. **循环迭代**：重复步骤 2-3，直至只剩一个连通分量。

完全图以 $(t_u + t_v) \mod k$ 为边权的 MST 问题，使用 **Boruvka 算法**。**Boruvka 算法非常适合处理这类边是隐式定义的、无需显式构建所有边的图。**
若图在几轮后快速收缩，能显著减少问题规模

8.2. 竞赛图

竞赛图是一种特殊的有向图，它模拟了一场“循环赛”的结果：每个参赛者都与其他所有参赛者比赛一次，且比赛必有胜负，没有平局。竞赛图是一种特殊的有向图，它模拟了一场“循环赛”的结果：每个参赛者都与其他所有参赛者比赛一次，且比赛必有胜负，没有平局。
一个 n 个顶点的竞赛图是一个有向图，其中每对不同的顶点 u 和 v 之间都恰好含有一条有向边。也就是说，要么存在边 (u, v)（表示 u 战胜了 v），要么存在边 (v, u)（表示 v 战胜了 u）。
1. 任何竞赛图都必定存在一条哈密顿路径。
2. 一个顶点数 $n \geq 3$ 的竞赛图是强连通的，当且仅当它含有一条哈密顿回路。
3. 在任何竞赛图中，都至少存在一个“王者” (King) 节点。一个顶点 k 被称为王者，如果对于图中任何其他顶点 v，k 要么直接战胜 v（即存在边 (k,v)），要么 k 战胜了某个顶点 w，而 w 又战胜了 v（即存在长度为 2 的路径 k⟶w⟶v）。
4. 任何出度（胜场数）最大的顶点必然是一个王者。
5. 一个由 n 个整数组成的非降序序列 $s_1, s_2, \cdots, s_n$ 是某个竞赛图的得分序列，当且仅当对于任意 $k \in 1, 2, , n$，都满足：

$$\sum_{i=1}^{k} s_i \geq \binom{k}{2}$$

且

$$\sum_{i=1}^{n} s_i = \binom{n}{2}$$

6. 如果一个竞赛图满足传递性关系（即若 u⟶v 且 v⟶w，则必有 u⟶w），则称之为传递性竞赛图。
传递性竞赛图是无环的 (DAG)。
它有唯一的哈密顿路径。
它的得分序列是 $0, 1, 2, , n1$。这意味着存在一个“全胜冠军”，一个“仅负于冠军”的亚军，以此类推，直到一个“全败”的选手。这对应了一个完全的线性排名。

7. 竞赛图缩点后的DAG是一条链，前面的所有点向后面的所有点连边。
拓扑序在前的SCC的任意一节点的入度严格小于拓扑序在后的SCC的任意一节点入度。

8. 若 x 点的出度大于或等于 y 点的出度，则 x 一定可以到达 y。

9. 若 x 点的入度小于或等于 y 点的入度，则 x 一定可以到达 y。

10. 按照入度从小到大排序，如果到前 i 个的入度和恰好为 $\binom{i}{2}$，则出现了一个新的强连通分量，假设上一次符合条件的是lst，则 $[lst+1,i]$ 构成了一个新的强连通分量。

8.3. dijsktra求有向图最小瓶颈路（AI, 未验证）

$O(nm\log n)$

```cpp
#include <bits/stdc++.h>
using namespace std;
using pii = pair<int, int>;
const int INF = 1e9;

int main() {
  ios::sync_with_stdio(false);
  cin.tie(nullptr);
  int n, m;
  if (!(cin >> n >> m)) return 0;
  vector<vector<pair<int, int>>> G(n);
  for (int i = 0; i < m; i++) {
    int u, v, w;
    cin >> u >> v >> w;
    --u;
    --v;
    G[u].push_back({v, w});
  }
  // result matrix
  vector<vector<int>> res(n, vector<int>(n, INF));
  for (int s = 0; s < n; s++) {
    // Dijkstra-like for minimax
    vector<int> dist(n, INF);
    dist[s] = 0;
    priority_queue<pair<int, int>, vector<pair<int, int>>,
                   greater<pair<int, int>>>
        pq;
    pq.push({0, s});
    while (!pq.empty()) {
      auto [d, u] = pq.top();
      pq.pop();
      if (d != dist[u]) continue;
      for (auto &e : G[u]) {
        int v = e.first, w = e.second;
        int cand = max(dist[u], w);
        if (cand < dist[v]) {
          dist[v] = cand;
          pq.push({dist[v], v});
        }
      }
    }
    // store
    for (int v = 0; v < n; ++v) res[s][v] = dist[v];
  }

  // 输出
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      if (res[i][j] == INF)
        cout << -1;
      else
        cout << res[i][j];
      if (j + 1 < n) cout << ' ';
    }
    cout << '\n';
  }
  return 0;
}
```

如果边比较少，图比较稀疏，可用堆优化掉log。

$O(n(m+K))$

```cpp
// compile: g++ -O2 -std=c++17 bucket_allpairs_discrete.cpp -o bucket_allpairs
#include <bits/stdc++.h>
using namespace std;
using pii = pair<int, int>;
const int INF = 1e9;

```

```
int main() {
  ios::sync_with_stdio(false);
  cin.tie(nullptr);
  int n, m;
  if (!(cin >> n >> m)) return 0;
  struct Edge {
    int u, v;
    long long w;
  };
  vector<Edge> edges;
  edges.reserve(m);
  vector<long long> vals;
  for (int i = 0; i < m; ++i) {
    int u, v;
    long long w;
    cin >> u >> v >> w;
    --u;
    --v;
    edges.push_back({u, v, w});
    vals.push_back(w);
  }
  if (n == 0) return 0;
  // special: if no edges
  if (m == 0) {
    for (int i = 0; i < n; i++) {
      for (int j = 0; j < n; j++) {
        if (i == j)
          cout << 0;
        else
          cout << -1;
        if (j + 1 < n) cout << ' ';
      }
      cout << '\n';
    }
    return 0;
  }

  // 离散化权值到 rank [1..K]
  sort(vals.begin(), vals.end());
  vals.erase(unique(vals.begin(), vals.end()), vals.end());
  int K = (int)vals.size();
  // map weight -> rank (1..K)
  auto rank_of = [&](long long w) -> int {
    int idx = int(lower_bound(vals.begin(), vals.end(), w) - vals.begin());
    return idx + 1;  // ranks start from 1
  };

  // build graph with ranked weights
  vector<vector<pair<int, int>>> G(n);
  for (auto &e : edges) {
    int r = rank_of(e.w);
    G[e.u].push_back({e.v, r});
  }

  // bucketed (Dial-like) single-source minimax Dijkstra
  auto bottleneck_from = [&](int s) {
    // dist in [0..K], where 0 means source itself (no edges)
    vector<int> dist(n, INT_MAX);
    dist[s] = 0;
    vector<vector<int>> buckets(K + 1);  // 0..K
    vector<int> head(K + 1, 0);
    buckets[0].push_back(s);
    int cur = 0;
    while (true) {
      while (cur <= K && head[cur] >= (int)buckets[cur].size()) ++cur;
      if (cur > K) break;
      int u = buckets[cur][head[cur]++];
      if (dist[u] != cur) continue;  // lazy skip
      for (auto &ed : G[u]) {
        int v = ed.first, wr = ed.second;
        int cand = max(dist[u], wr);
        if (cand < dist[v]) {
          dist[v] = cand;
          if (cand <= K) buckets[cand].push_back(v);
          // cand should always be <=K (weights ranks are within 1..K)
        }
      }
    }
    return dist;
  };

  // 全源求解并输出（输出原始权值; dist==0 且 i==s 表示 0; dist==INF
```

```cpp
  // 表示不可达）
  vector<int> dist;
  for (int s = 0; s < n; ++s) {
    dist = bottleneck_from(s);
    for (int t = 0; t < n; ++t) {
      if (s == t)
        cout << 0;
      else if (dist[t] == INT_MAX)
        cout << -1;
      else {
        int r = dist[t];
        // r in 1..K
        cout << vals[r - 1];
      }
      if (t + 1 < n) cout << ' ';
    }
    cout << '\n';
  }
  return 0;
}
```

在稠密图，$n \leq 2000$，可以用bitset优化传递闭包。

8.4. tarjan有重边找桥

```cpp

int low[MAXN], dfn[MAXN], idx;
bool isbridge[MAXN];
vector<int> G[MAXN];
int cnt_bridge;
int father[MAXN];

void tarjan(int u, int fa) {
  bool flag = false;
  father[u] = fa;
  low[u] = dfn[u] = ++idx;
  for (const auto &v : G[u]) {
    if (!dfn[v]) {
      tarjan(v, u);
      low[u] = min(low[u], low[v]);
      if (low[v] > dfn[u]) {
        isbridge[v] = true;
        ++cnt_bridge;
      }
    } else {
      if (v != fa || flag)
        low[u] = min(low[u], dfn[v]);
      else
        flag = true;
    }
  }
}
```

Chapter 9

语法

9.1. 复数

引入与类型

```
// 头文件与常用别名
#include <complex>
using std::complex;
using cd = complex<double>;
using cf = complex<float>;
using cld = complex<long double>;
```

构造与访问

- 构造：

```
cd z1(1.0, 2.0);      // 实部 1，虚部 2
cd z2 = {3.0, -1.5};  // 列表初始化
cd z3 = cd(4.0);      // 虚部为 0
```

- 访问／修改实部与虚部：

```
double a = z1.real();          // 成员函数读取实部
double b = z1.imag();          // 成员函数读取虚部
std::real(z1);                 // free function，等价 z1.real()
std::imag(z1);                 // free function，等价 z1.imag()
z1.real() = 5.0;               // 可写（修改实部）
```

四则运算与标量混合

- 支持 + - * /，既有 complex op complex 也有 complex op scalar。

```
cd s = z1 + z2;
cd p = z1 * cd(0,1);   // 乘以 i
cd d = z1 / 2.0;       // 除以标量
```

常用数学函数（<complex>）

- 模长、模平方、相位：

```
double r     = std::abs(z);   // |z|
double r2    = std::norm(z);  // |z|^2，避免开根号的高效计算
double theta = std::arg(z);   // 相位（弧度）
```

- 共轭、极坐标、投影：

```
cd conjz = std::conj(z);           // 复共轭
cd fromPolar = std::polar(r,theta); // 从极坐标构造 r*e^{i theta}
cd projz = std::proj(z);           // 投影到黎曼球面（处理无穷）
```

- 复数版本的初等/超越函数：

```
std::exp(z); std::log(z); std::sqrt(z); std::pow(z, w);
std::sin(z); std::cos(z); std::tan(z);
std::sinh(z); std::cosh(z); std::asin(z); std::acos(z); std::atan(z);
```

I/O 与比较

- 流支持：

```
std::cout << z; // 输出格式依实现（常见形式 "(real,imag)"）
std::cin  >> z; // 读取，格式由实现决定
```

- 比较：不要用 == 做严格相等判断（浮点误差）。

```
bool approx_equal = (std::abs(z1 - z2) < 1e-9);
```

性能与数值注意事项

- 若只需模的平方，优先使用 `std::norm(z)`，比 `std::abs(z)*std::abs(z)` 更快且更稳定。
- 判断是否为 0 时应使用容差 `eps`：

```
if (std::abs(z) < eps) { /* 视为 0 */ }
```

- `std::complex<T>` 使用模板参数 `T`（float/double/long double），根据精度与性能需求选择合适类型。
- 对于列主元或比较大小时，通常用 `std::abs`（模）比较大小。

常见小例子

用复数表示二维向量并旋转

```
// 旋转复数（2D 向量） by angle theta:
cd v = {1.0, 0.0};
cd rotated = v * std::polar(1.0, theta);
```

在高斯消元中使用 std::complex<double>

```
// 假设增广矩阵 ar 为 complex<double>:
cd ar[MAXN][MAXN+1]; // 每行长度为 n+1
// 判主元、交换、消元时可直接使用 std::abs, std::conj 等
```

额外提示

- `std::abs` 返回的是模（非平方），内部对极端情况有一定数值稳定性的处理，但仍需注意大/小量级混合的问题。
- 在数值线性代数中，若频繁调用共轭、模等，注意避免不必要的临时对象以减少开销（视编译器做内联优化情况）。
- 如果需要可移植的输出格式，自己实现格式化（例如 `printf("%.12g + %.12gi", z.real(), z.imag())`）会更稳定一致。

Chapter 10

网络流

10.1. SPFA费用流

```
#include <bits/stdc++.h>
using namespace std;

template <class T>
struct MinCostFlow {
  struct _Edge {
    int to;
    T cap;
    T cost;
    _Edge(int to_, T cap_, T cost_) : to(to_), cap(cap_), cost(cost_) {}
  };
  int n;
  vector<_Edge> e;
  vector<vector<int>> g;
  vector<T> dis;     // 最短路距离（SPFA）
  vector<int> pre;   // 记录到某点的边索引
  vector<char> inq;  // SPFA 是否在队列中

  MinCostFlow() {}
  MinCostFlow(int n_) { init(n_); }

  void init(int n_) {
    n = n_;
    e.clear();
    g.assign(n, {});
  }

  void addEdge(int u, int v, T cap, T cost) {
    g[u].push_back((int)e.size());
    e.emplace_back(v, cap, cost);
    g[v].push_back((int)e.size());
    e.emplace_back(u, 0, -cost);
  }

  // 使用 SPFA 求 s->t 的最短路（边权为 cost，且只考虑 cap>0 的边）
  bool spfa(int s, int t) {
    // INF 取为 numeric_limits<T>::max()/4 避免加法溢出
    const T INF = numeric_limits<T>::max() / 4;
    dis.assign(n, INF);
    pre.assign(n, -1);
    inq.assign(n, 0);

    queue<int> q;
    dis[s] = 0;
    q.push(s);
    inq[s] = 1;

    while (!q.empty()) {
      int u = q.front();
      q.pop();
      inq[u] = 0;
      for (int idx : g[u]) {
        const _Edge &ed = e[idx];
        int v = ed.to;
        T cap = ed.cap;
        T cost = ed.cost;
        if (cap > 0 && dis[v] > dis[u] + cost) {
          dis[v] = dis[u] + cost;
          pre[v] = idx;
          if (!inq[v]) {
```

```
      q.push(v);
      inq[v] = 1;
     }
    }
   }
  }
  return pre[t] != -1;  // 或者 dis[t] < INF
 }

 pair<T, T> flow(int s, int t) {
  T flow = 0;
  T cost = 0;
  // 不再使用 potentials h，直接用 SPFA
  while (spfa(s, t)) {
   // 找增广量
   T aug = numeric_limits<T>::max();
   for (int i = t; i != s; i = e[pre[i] ^ 1].to) {
    aug = min(aug, e[pre[i]].cap);
   }
   // 改变残量网络
   for (int i = t; i != s; i = e[pre[i] ^ 1].to) {
    e[pre[i]].cap -= aug;
    e[pre[i] ^ 1].cap += aug;
   }
   flow += aug;
   // dis[t] 是此次最短路径的代价
   cost += aug * dis[t];
  }
  return make_pair(flow, cost);
 }

 struct Edge {
  int from;
  int to;
  T cap;
  T cost;
  T flow;
 };

 // 返回原始图的边（与原实现兼容）
 vector<Edge> edges() {
  vector<Edge> a;
  for (int i = 0; i < (int)e.size(); i += 2) {
   Edge x;
   x.from = e[i + 1].to;
   x.to = e[i].to;
   x.cap = e[i].cap + e[i + 1].cap;  // 原始容量（被分配前的）
   x.cost = e[i].cost;
   x.flow = e[i + 1].cap;  // 反向边的容量表示已经流过的量
   a.push_back(x);
  }
  return a;
 }
};
```

spfa初始化。

```
void spfa_init_potentials(int s) {
    const T INF = numeric_limits<T>::max() / 4;
    h.assign(n, INF);
    vector<char> inq(n, 0);
    queue<int> q;
    h[s] = 0;
    q.push(s);
    inq[s] = 1;
    while (!q.empty()) {
      int u = q.front(); q.pop();
      inq[u] = 0;
      for (int idx : g[u]) {
        const _Edge &ed = e[idx];
        int v = ed.to;
        if (ed.cap > 0 && h[v] > h[u] + ed.cost) {
          h[v] = h[u] + ed.cost;
          if (!inq[v]) {
            q.push(v);
            inq[v] = 1;
          }
        }
      }
    }
    // 把不可达的点势设为 0，避免后续计算中出现 INF
    for (int i = 0; i < n; ++i) if (h[i] == INF) h[i] = 0;
```

```
    }
```

10.2. 网络流封装 (已验证)

由GPT，Gemini，Grok生成，已通过模板题。

```
#include <algorithm>
#include <iostream>
#include <limits>
#include <numeric>
#include <queue>
#include <vector>
using namespace std;

/**
 * @brief 网络流封装
 *
 * 使用原始对偶算法 (Primal-Dual)，配合 Dijkstra 和势函数。
 * 同时集成一个普通的最大流（Dinic）用于不带费用的最大流场景，
 * 并把所有“没有 cost 的最大流接口”改为使用普通最大流以获得更好常数与简洁性。
 *
 * @tparam T 容量和费用的数据类型 (例如 int, long long)。
 */
template <class T>
struct MinCostFlow {
  // ----- 内部最小费用边结构（保留用于带费用的算法） -----
  struct _Edge {
    int to;  // 终点
    T cap;   // 容量 / 当前残量
    T cost;  // 费用（最大流算法忽略）
    _Edge(int to_ = 0, T cap_ = 0, T cost_ = 0)
        : to(to_), cap(cap_), cost(cost_) {}
  };

  // 对外返回的边信息
  struct Edge {
    int from, to;
    T cap, cost, flow;
  };

  // 方便的参数结构体
  struct E_Cap {
    int u, v;
    T cap;
  };  // 仅有容量的边
  struct E_Cost {
    int u, v;
    T cap, cost;
  };  // 有容量和费用的边
  struct E_Bound {
    int u, v;
    T low, cap;
  };  // 有流量下界的边
  struct E_Full {
    int u, v;
    T low, cap, cost;
  };  // 完整信息的边

  int n;
  vector<_Edge> e;        // 用于最小费用流（也被最大流复用：仅 cap 字段）
  vector<vector<int>> g;  // 邻接表：存储边在 e 中的索引（成对出现：正向、反向）
  vector<T> h, dis;       // 势函数 / Dijkstra 距离
  vector<int> pre;        // Dijkstra 的前驱（存储边索引）

  // 为 Dinic 复用的结构体（减少重复分配）
  vector<int> level;
  vector<int> it_ptr;

  const T INF = numeric_limits<T>::max() / 4;

 public:
  MinCostFlow() : n(0) {}
  MinCostFlow(int n_) { init(n_); }

  // init 支持期望边数以便预分配
  void init(int n_, size_t expected_edges = 0) {
    n = n_;
    e.clear();
    if (expected_edges) e.reserve(expected_edges * 2 + 4);
    g.assign(n, {});
    if (expected_edges && n > 0) {
```

```
    size_t avg = max<size_t>(1, expected_edges / (size_t)n);
    for (int i = 0; i < n; ++i) g[i].reserve(avg + 1);
  }
  h.assign(n, 0);
  dis.assign(n, 0);
  pre.assign(n, -1);
  level.assign(n, -1);
  it_ptr.assign(n, 0);
}

// 加边 (u->v, 容量 cap, 费用 cost)
inline void addEdge(int u, int v, T cap, T cost) {
  g[u].push_back((int)e.size());
  e.emplace_back(v, cap, cost);
  g[v].push_back((int)e.size());
  e.emplace_back(u, (T)0, -cost);  // 反向边（初始 cap=0）
}

// ---------------- 最小费用流部分（与之前保持兼容） ----------------

// SPFA 初始化势函数 h，用于处理负权边
bool spfa_init_h(int s) {
  T localINF = INF;
  std::fill(h.begin(), h.end(), localINF);
  vector<char> inq(n, 0);
  vector<int> cnt(n, 0);
  vector<int> q;
  q.reserve(n * 2 + 4);
  int qh = 0;
  q.push_back(s);
  inq[s] = 1;
  cnt[s] = 1;
  h[s] = 0;
  while (qh < (int)q.size()) {
    int u = q[qh++];
    inq[u] = 0;
    for (int idx : g[u]) {
      const _Edge& ed = e[idx];
      if (ed.cap > 0) {
        T nv = h[u] + ed.cost;
        if (h[ed.to] > nv) {
          h[ed.to] = nv;
          if (!inq[ed.to]) {
            inq[ed.to] = 1;
            q.push_back(ed.to);
            if (++cnt[ed.to] > n) return false;  // 检测到负环
          }
        }
      }
    }
  }
  return true;
}

// Dijkstra 寻找最短增广路（使用势函数）
bool dijkstra(int s, int t) {
  T localINF = INF;
  std::fill(dis.begin(), dis.end(), localINF);
  std::fill(pre.begin(), pre.end(), -1);
  using P = pair<T, int>;
  priority_queue<P, vector<P>, greater<P>> pq;
  dis[s] = 0;
  pq.emplace((T)0, s);
  while (!pq.empty()) {
    auto [d, u] = pq.top();
    pq.pop();
    if (d != dis[u]) continue;
    for (int i : g[u]) {
      const _Edge& ed = e[i];
      if (ed.cap <= 0) continue;
      T nd = d + h[u] - h[ed.to] + ed.cost;
      if (dis[ed.to] > nd) {
        dis[ed.to] = nd;
        pre[ed.to] = i;
        pq.emplace(nd, ed.to);
      }
    }
  }
  return dis[t] != localINF;
}

// 求解最小费用最大流（默认版本，边权非负）
```

```
  pair<T, T> flow(int s, int t) {
    T flow_val = 0;
    T cost_val = 0;
    std::fill(h.begin(), h.end(), (T)0);
    while (dijkstra(s, t)) {
      for (int i = 0; i < n; ++i) {
        if (dis[i] != INF) h[i] += dis[i];
      }
      T aug = INF;
      for (int i = t; i != s; i = e[pre[i] ^ 1].to) {
        aug = min(aug, e[pre[i]].cap);
      }
      for (int i = t; i != s; i = e[pre[i] ^ 1].to) {
        e[pre[i]].cap -= aug;
        e[pre[i] ^ 1].cap += aug;
      }
      flow_val += aug;
      cost_val += aug * h[t];
    }
    return {flow_val, cost_val};
  }

  // 可处理负权边的 flow（警告：图中不能有从源点可达的负费用环）
  pair<T, T> flow_neg(int s, int t) {
    if (!spfa_init_h(s)) {
      return {0, -INF};  // 有从源可达的负环，费用无下界
    }
    return flow(s, t);
  }

  // 获取最终的流网络信息（基于 e/g：假定边以成对方式存储）
  vector<Edge> edges() {
    vector<Edge> res;
    res.reserve(e.size() / 2);
    for (size_t i = 0; i + 1 < e.size(); i += 2) {
      // forward edge is at i, reverse at i+1
      res.push_back({e[i + 1].to, e[i].to, e[i].cap + e[i + 1].cap, e[i].cost,
                     e[i + 1].cap});
    }
    return res;
  }

  // ---------------- 普通最大流（Dinic）实现（复用 e/g 的 cap 字段）
  // ----------------

  // 为兼容性：在当前已使用 addEdge 构建好的图上运行 Dinic，修改 e 中的 cap &
  // 反向 cap
  T maxflow_on_current_graph(int s, int t) {
    // level / it_ptr 已在 init 时分配
    auto bfs = [&](void) -> bool {
      std::fill(level.begin(), level.end(), -1);
      vector<int> q;
      q.reserve(n);
      int qh = 0;
      q.push_back(s);
      level[s] = 0;
      while (qh < (int)q.size()) {
        int u = q[qh++];
        for (int idx : g[u]) {
          if (e[idx].cap > 0 && level[e[idx].to] == -1) {
            level[e[idx].to] = level[u] + 1;
            q.push_back(e[idx].to);
          }
        }
      }
      return level[t] != -1;
    };

    function<T(int, T)> dfs = [&](int u, T pushed) -> T {
      if (u == t || pushed == 0) return pushed;
      for (int& cid = it_ptr[u]; cid < (int)g[u].size(); ++cid) {
        int ei = g[u][cid];
        _Edge& ed = e[ei];
        if (ed.cap > 0 && level[ed.to] == level[u] + 1) {
          T tr = dfs(ed.to, min(pushed, ed.cap));
          if (tr > 0) {
            ed.cap -= tr;
            e[ei ^ 1].cap += tr;
            return tr;
          }
        }
      }
```

```
      return (T)0;
    };

    T flow = 0;
    while (bfs()) {
      std::fill(it_ptr.begin(), it_ptr.end(), 0);
      while (true) {
        T pushed = dfs(s, INF);
        if (pushed == 0) break;
        flow += pushed;
      }
    }
    return flow;
  }

  // 一个便捷的接口：根据传入的边列表构建图并求最大流（与旧接口保持相同签名）
  T max_flow(int _n, int s, int t, const vector<E_Cap>& es) {
    init(_n, es.size());
    for (const auto& edge : es) addEdge(edge.u, edge.v, edge.cap, 0);
    return maxflow_on_current_graph(s, t);
  }

  // ---------------- 封装接口（把“无费用最大流”相关操作改为使用 Dinic）
  // ----------------

  // 1. 无源汇可行流（循环流）
  // 使用 Dinic 来判断可行性并且保留 e/g 中的流量信息，便于后续读取 edges()
  bool feasible_circulation(int _n, const vector<E_Bound>& es) {
    // 仍然在 this (mcf 对象) 上构建图，以便 edges() 能反映流
    init(_n + 2, es.size());
    int SS = _n, ST = _n + 1;
    vector<T> diff(_n, 0);

    for (const auto& edge : es) {
      diff[edge.u] -= edge.low;
      diff[edge.v] += edge.low;
      addEdge(edge.u, edge.v, edge.cap - edge.low, 0);
    }

    T sup_sum = 0;
    for (int i = 0; i < _n; ++i) {
      if (diff[i] > 0) {
        addEdge(SS, i, diff[i], 0);
        sup_sum += diff[i];
      } else if (diff[i] < 0) {
        addEdge(i, ST, -diff[i], 0);
      }
    }

    T flow_val = maxflow_on_current_graph(SS, ST);
    return flow_val == sup_sum;
  }

  // 2. 有源汇可行流
  // 返回 {是否存在，一个可行的流值}
  pair<bool, T> feasible_flow(int _n, int s, int t, const vector<E_Bound>& es) {
    vector<E_Bound> circ_es = es;
    circ_es.push_back({t, s, 0, INF});
    if (feasible_circulation(_n, circ_es)) {
      T res = 0;
      // 流值为 t->s 的反向边的流量（edges() 中的 flow 值）
      for (const auto& ed : this->edges()) {
        if (ed.from == t && ed.to == s) {
          res = ed.flow;
          break;
        }
      }
      return {true, res};
    }
    return {false, 0};
  }

  // 3. 有源汇上下界最大流
  // 返回 {是否存在可行流，最大流值}
  // 说明：先判断是否存在满足上下界的可行流（添加 t->s 无限边），若存在则读取
  // t->s 的流量 flow1，然后在残量图（去掉人工 t->s 边）上用 Dinic 再增广得到
  // flow2，最终最大流为 flow1 + flow2。
  pair<bool, T> bounded_max_flow(int _n, int s, int t,
                                 const vector<E_Bound>& es) {
    // 在一个临时 mcf1 对象上构造并判断可行性（但为了方便后续读取
    // edges()，这里直接用当前对象）
    MinCostFlow<T> mcf1;
```

```
    vector<E_Bound> circ_es = es;
    circ_es.push_back({t, s, 0, INF});
    // 使用 mcf1 的 feasible_circulation（其内部会 init 并运行 Dinic，并保留
    // e/g）
    if (!mcf1.feasible_circulation(_n, circ_es)) {
      return {false, 0};
    }

    // 找到 t->s 边的流量（作为初始 s->t 流）
    T flow1 = 0;
    for (const auto& ed : mcf1.edges()) {
      if (ed.from == t && ed.to == s) {
        flow1 = ed.flow;
        break;
      }
    }

    // 构建残量图：将 mcf1 中剩余容量 >0 的边作为 residual_edges（跳过人工 t->s
    // 边）
    vector<E_Cap> residual_edges;
    residual_edges.reserve(mcf1.e.size() / 2);
    for (const auto& ed : mcf1.edges()) {
      if (ed.from < _n && ed.to < _n) {
        if (ed.from == t && ed.to == s) continue;  // 跳过人工边
        T rem_cap = ed.cap - ed.flow;              // 剩余容量
        if (rem_cap > 0) residual_edges.push_back({ed.from, ed.to, rem_cap});
      }
    }

    // 在新图上用 Dinic 再跑一次最大流（增广 s->t）
    MinCostFlow<T> mcf2;
    T flow2 = mcf2.max_flow(_n, s, t, residual_edges);

    return {true, flow1 + flow2};
  }

  // 4. 有源汇最小流
  // 返回 {是否存在, 最小流值}
  pair<bool, T> min_flow(int _n, int s, int t, const vector<E_Bound>& es) {
    MinCostFlow<T> mcf1;
    vector<E_Bound> circ_es = es;
    circ_es.push_back({t, s, 0, INF});  // 添加 t->s 的边构成循环

    bool ok = mcf1.feasible_circulation(_n, circ_es);
    if (!ok) return {false, 0};

    T flow1 = 0;  // 初始可行流 (t->s)
    for (const auto& ed : mcf1.edges()) {
      if (ed.from == t && ed.to == s) {
        flow1 = ed.flow;
        break;
      }
    }

    // 在残量图上，从 t 到 s 跑最大流，即可退回最多的流
    MinCostFlow<T> mcf2;
    mcf2.init(_n, mcf1.e.size() / 2);

    for (const auto& ed : mcf1.edges()) {
      // 只考虑原始节点范围内的边
      if (ed.from < _n && ed.to < _n) {
        // 跳过我们人为加入的 t->s 边（否则会误导增广）
        if (ed.from == t && ed.to == s) continue;

        T forward_rem = ed.cap - ed.flow;  // 前向剩余容量 = C - f
        T backward_rem = ed.flow;          // 反向剩余容量 = f (可以退回的流)

        if (forward_rem > 0) mcf2.addEdge(ed.from, ed.to, forward_rem, 0);
        if (backward_rem > 0) mcf2.addEdge(ed.to, ed.from, backward_rem, 0);
      }
    }

    T flow2 = mcf2.maxflow_on_current_graph(t, s);
    return {true, flow1 - flow2};
  }

  // 5. 有源汇上下界最小费用可行流
  // 返回 {是否存在, 流值, 费用值}，如果有负环则费用为 -INF 表示无下界
  tuple<bool, T, T> min_cost_feasible_flow(int _n, int s, int t,
                                           const vector<E_Full>& es) {
    vector<E_Full> circ_es = es;
    circ_es.push_back({t, s, (T)0, INF, (T)0});
```

```cpp
  auto [ok, cost] = min_cost_circulation(_n, circ_es);
  if (!ok) return {false, (T)0, (T)0};
  T f = 0;
  for (const auto& ed : edges()) {
    if (ed.from == t && ed.to == s) {
      f = ed.flow;
      break;
    }
  }
  return {true, f, cost};
}

// 6. 无源汇上下界最小费用可行流
// 返回 {是否存在, 费用值}, 如果有负环则费用为 -INF 表示无下界
pair<bool, T> min_cost_circulation(int _n, const vector<E_Full>& es) {
  init(_n + 2, es.size());
  int SS = _n, ST = _n + 1;
  vector<T> diff(_n, 0);
  T base_cost = 0;
  bool has_neg = false;
  for (const auto& edge : es) {
    if (edge.low > edge.cap) return {false, (T)0};
    diff[edge.u] -= edge.low;
    diff[edge.v] += edge.low;
    base_cost += edge.low * edge.cost;
    if (edge.cap > edge.low) {
      addEdge(edge.u, edge.v, edge.cap - edge.low, edge.cost);
      if (edge.cost < 0) has_neg = true;
    }
  }
  T sup_sum = 0;
  for (int i = 0; i < _n; ++i) {
    if (diff[i] > 0) {
      addEdge(SS, i, diff[i], (T)0);
      sup_sum += diff[i];
    } else if (diff[i] < 0) {
      addEdge(i, ST, -diff[i], (T)0);
    }
  }
  pair<T, T> res{(T)0, (T)0};
  if (has_neg) {
    res = flow_neg(SS, ST);
    if (res.second == -INF) return {false, -INF};
  } else {
    res = flow(SS, ST);
  }
  if (res.first < sup_sum) return {false, (T)0};
  return {true, base_cost + res.second};
}

// 7. 有负环的费用流（最小费用最大流，支持负费用，可能有负环时计算有限费用）
// 返回 {是否存在解, 流值,
// 费用值}，如果无法平衡（可能由于负环导致无界）则不存在解
tuple<bool, T, T> min_cost_max_flow_neg(int _n, int s, int t,
                                        const vector<E_Cost>& es) {
  init(_n + 2, es.size());
  int SS = _n, ST = _n + 1;
  vector<T> diff(_n, 0);
  T base_cost = 0;
  for (const auto& edge : es) {
    if (edge.cost >= 0) {
      addEdge(edge.u, edge.v, edge.cap, edge.cost);
    } else {
      if (edge.cap == 0) continue;
      diff[edge.u] -= edge.cap;
      diff[edge.v] += edge.cap;
      base_cost += edge.cap * edge.cost;
      addEdge(edge.v, edge.u, edge.cap, -edge.cost);
    }
  }
  T sup_sum = 0;
  for (int i = 0; i < _n; ++i) {
    if (diff[i] > 0) {
      addEdge(SS, i, diff[i], (T)0);
      sup_sum += diff[i];
    } else if (diff[i] < 0) {
      addEdge(i, ST, -diff[i], (T)0);
    }
  }
  addEdge(t, s, INF, (T)0);
  pair<T, T> res1 = flow(SS, ST);
  if (res1.first < sup_sum) {
```

```
      return {false, (T)0, (T)0};
    }
    size_t last_rev = e.size() - 1;
    T flow_back = e[last_rev].cap;
    // 禁用超级源汇相关边
    for (int idx : g[SS]) {
      e[idx].cap = 0;
      e[idx ^ 1].cap = 0;
    }
    for (int idx : g[ST]) {
      e[idx].cap = 0;
      e[idx ^ 1].cap = 0;
    }
    // 禁用 t->s 边
    size_t ts_fwd = e.size() - 2;
    e[ts_fwd].cap = 0;
    e[ts_fwd ^ 1].cap = 0;
    pair<T, T> res2 = flow(s, t);
    T total_flow = flow_back + res2.first;
    T total_cost = base_cost + res1.second + res2.second;
    return {true, total_flow, total_cost};
  }
};
```

1.**二分图的最大匹配**就是在二分图上跑出来的**最大流**。

2. 我们有**最小覆盖数=最大匹配数**，**最大独立集=总点数-最小覆盖集**两个性质。

3. 在一个二分图中，如果删去一条边能够使这个图的最大匹配减小1的话，那么这条边一定在残量网络中满流，并且它所连接的两个点一定不在同一个强连通分量当中。

4. 有一个DAG，要求用尽量少的不相交的简单路径覆盖所有的节点。有**最小路径覆盖=原图节点数-新图最大匹配**新图指的是，将原来的点拆成两个点，如果 $u \to v$，那么连接 $u_x \to v_y$，这样得到的二分图的最大匹配。