「集思广益一波」

CleanBlue

2022 年 12 月 3 日

目录

1 基本算法

1.1 快速幂/整数类/取模

```
constexpr int P = 1000000007;
int norm(int x) {
    if (x < 0) x += P;
    if (x >= P) x -= P;
    return x;
}
template <class T> constexpr T power(T a,
    int64_t b) {
    T res = 1;
    for (; b; b /= 2, a *= a)
    if (b % 2) res *= a;
    return res;
}

struct Z {
    int x;
    Z(int x = 0) : x(norm(x)) {}
    Z(int64_t x) : x(norm(x % P)) {}
    int val() const { return x; }
    Z operator-() const { return Z(norm(P - x));
        }
    Z inv() const {
        assert(x != 0);
        return power(*this, P - 2);
    }
    Z &operator*=(const Z &rhs) {
        x = int64_t(x) * rhs.x % P;
        return *this;
    }
    Z &operator+=(const Z &rhs) {
        x = norm(x + rhs.x);
        return *this;
    }
    Z &operator-=(const Z &rhs) {
        x = norm(x - rhs.x);
        return *this;
    }
    Z &operator/=(const Z &rhs) { return *this
        *= rhs.inv(); }
    friend Z operator*(const Z &lhs, const Z &
        rhs) {
        Z res = lhs;
        res *= rhs;
        return res;
    }
    friend Z operator+(const Z &lhs, const Z &
        rhs) {
        Z res = lhs;
        res += rhs;
        return res;
    }
    friend Z operator-(const Z &lhs, const Z &
        rhs) {
        Z res = lhs;
        res -= rhs;
        return res;
    }
    friend Z operator/(const Z &lhs, const Z &
        rhs) {
        Z res = lhs;
        res /= rhs;
        return res;
    }
    friend std::istream &operator>>(std::istream
        &is, Z &a) {
        int64_t v;
        is >> v;
        a = Z(v);
        return is;
    }
    friend std::ostream &operator<<(std::ostream
        &os, const Z &a) { return os << a.val();
        }
};
```

1.2 一二三维差分/前缀和

一维：区间 $[l, r]$ 每个元素加上 x：

```
d[l] += x;
d[r + 1] -= x;
```

二维：$(x1, y1), (x2, y2)$ 每个元素加上 x：

```
d[x1][y1] += x;
d[x1][y2 + 1] -= x;
d[x2 + 1][y1] -= x;
d[x2 + 1][y2 + 1] += x;
```

三维：$(x1, y1, z1), (x2, y2, z2)$ 每个元素加上 x：

```
d[x1][y1][z1] += x;
d[x2 + 1][y1][z1] -= x;
d[x1][y1][z2 + 1] -= x;
d[x2 + 1][y1][z2 + 1] += x;
d[x1][y2 + 1][z1] -= x;
d[x2 + 1][y2 + 1][z1] += x;
d[x1][y2 + 1][z2 + 1] += x;
d[x2 + 1][y2 + 1][z2 + 1] -=x;
```

1.3 二分

```
int binary_search_1(const std::vector<int> &a,
    int x) {
    // a 单调递增
    // 在 [0, n) 中查找 x 或 x 的后继的位置
    int n = a.size(), l = 0, r = n;
    while (l < r) {
        int mid = (l + r) / 2;
        if (a[mid] >= x)
            r = mid;
        else
            l = mid + 1;
    }
    return l;
}

int binary_search_2(const std::vector<int> &a,
    int x) {
    // a 单调递增
    //在 [0, n) 中查找 x 或 x 的前驱的位置
    int n = a.size(), l = 0, r = n;
    while (l < r) {
        int mid = (l + r + 1) / 2;
        if (a[mid] <= x)
```

```
            l = mid;
        else
            r = mid - 1;
    }
    return l;
}

double float_binary_search(/*...*/) {
    constexpr double eps = 1e-7;
    double l = 0, r = 1e18;
    while (r - l > eps) {
        double mid = (l + r) / 2;
        if (/*...*/)
            l = mid;
        else
            r = mid;
    }
    return l;
}
```

1.4 三分

如何取 mid1 和 mid2? 有以下两种基本方法。

(1) 三等分：mid1 和 mid2 为 $[l,r]$ 的三等分点，区间每次可以缩小 1 / 3。

(2) 近似三等分：计算 $[l,r]$ 的中间点 mid $=(l+r)/2$，然后让 mid1 和 mid2 非常接近 mid，如 mid1 = mid - eps, mid2 = mid + eps。区间每次可以缩小接近一半。

近似三等分比三等分要稍微快一点。不过，在有些情况下 eps 过小可能导致 mid1 和 mid2 算出的结果相等，从而判断错方向。

三等分：

```
while (r - l > eps) {
    double k = (r - l) / 3;
    double mid1 = l + k, mid2 = r - k;
    if (/*...*/)
        r = mid2;
    else
        l = mid1;
}
```

近似三等分：

```
while (r - l > eps) {
    double mid1 = (l + r) / 2;
    double mid2 = mid1 + eps;
    if (/*...*/)
        r = mid2;
    else
        l = mid1;
}
```

整数三分：

```
while (r - l > 2) {
    int mid1 = l + (r - l) / 3;
    int mid2 = r - (r - l) / 3;
    if (/*...*/)
        r = mid2;
    else
        l = mid1;
}
```

1.5 ST/离线区间最值

定义 $dp[i][k]$ 表示左断点为 i，区间长度为 2^k 的区间最值。递推关系为:

$$dp[i][k] = merge\{dp[i][k-1], dp[i+(1<<(k-1))][k]\}$$

$O(nlogn)$ 预处理、$O(1)$ 查询。

```
int N, M;
std::cin >> N >> M;

std::vector<int> a(N);
for (int &i : a) std::cin >> i;

std::vector<int> LOG2(N + 1);
LOG2[0] = -1;
for (int i = 1; i <= N; ++i)
    LOG2[i] = LOG2[i / 2] + 1;

int dp[N][21];
for (int i = 0; i < N; ++i)
    dp[i][0] = a[i];

int p = LOG2[N];
// 可倍增区间的最大次方: 2^p <= N

for (int k = 1; k <= p; ++k)
    for (int i = 0; i + (1 << k) - 1 < N; ++i)
        dp[i][k] = std::max(dp[i][k - 1], dp[i +
            (1 << (k - 1))][k - 1]);

while (M--) {
    int l, r;
    //求 [l, r] 中最大值
    std::cin >> l >> r;
    --l;

    int len = r - l;
    int k = LOG2[len];

    std::cout << std::max(dp[l][k], dp[r - (1 <<
        k)][k]) << '\n';
}
```

ST 类:

```
struct ST {
    size_t n;
    std::vector<std::vector<int>> info;

    int merge(const int &lhs, const int &rhs) {
        // TODO: 修改merge
        return std::max(lhs, rhs);
    }

    ST(const std::vector<int> &init) : n(init.
        size()), info(init.size() + 1, std::
        vector<int>(std::__lg(n) + 1, INT_MIN)) {
        //TODO : 修改初值
        for (size_t x = 0; x <= std::__lg(n); ++
            x)
        for (int i = 0; i + (1ull << x) <= n; ++
            i)
        if (x == 0)
        info[i][x] = init[i];
        else
```

```
        info[i][x] = merge(info[i][x - 1], info[
            i + (1 << (x - 1))][x - 1]);
    }

    int query(int l, int r) {  //[l, r)
        int len = r - l;
        assert(len > 0);
        return merge(info[l][std::__lg(len)],
            info[r - (1 << std::__lg(len))][std::
            __lg(len)]);
    }
};
```

1.6 归并排序/逆序对

```
int n;
std::cin >> n;

std::vector<int> a(n), b(n);
for (int &i : a) std::cin >> i;

int64_t ans = 0;
std::function<void(int, int)> merge_sort = [&](
    int l, int r) {
    //排序区间 [l, r)
    if (r - l <= 1) return;
    int mid = (l + r) / 2;
    merge_sort(l, mid);
    merge_sort(mid, r);
    for (int i = l, j = mid, k = l; k < r; ++k)
        {
        if (j == r || (i < mid && a[i] <= a[j]))
            b[k] = a[i++];
        else {
            ans += mid - i;
            b[k] = a[j++];
        }
    }
    for (int i = l; i < r; ++i) a[i] = b[i];
};

merge_sort(0, n);
std::cout << ans << '\n';
```

1.7 分块

```
int block = std::sqrt(n);  //块的大小: 每块有
    block个元素
int t = n / block;         //块的数量: 共分为t块
if (n % block != 0) t += 1;

for (int i = 1; i <= t; ++i) {
    st[i] = (i - 1) * block + 1;
    ed[i] = i * block;
}
ed[t] = n;

for (int i = 1; i <= n; ++i)
    pos[i] = (i - 1) / block + 1;  // 每个元素所
        在块的编号
```

1.8 莫队

假设 $n = m$，那么对于序列上的区间询问问题，如果从 $[l, r]$ 的答案能够 $O(1)$ 扩展到 $[l-1, r], [l+1, r], [l, r+1], [l, r-1]$（即与 $[l, r]$ 相邻的区间）的答案，那么可以在 $O(n\sqrt{n})$ 的复杂度内求出所有询问的答案。

离线后排序，顺序处理每个询问，暴力从上一个区间的答案转移到下一个区间答案（一步一步移动即可）。

对于区间 $[l, r]$，以 l 所在块的编号为第一关键字，r 为第二关键字从小到大排序。

```
inline void move(int pos, int sign) {
    // update nowAns
}

void solve() {
    BLOCK_SIZE = int(ceil(pow(n, 0.5)));
    sort(querys, querys + m);
    for (int i = 0; i < m; ++i) {
        const query &q = querys[i];
        while (l > q.l) move(--l, 1);
        while (r < q.r) move(r++, 1);
        while (l < q.l) move(l++, -1);
        while (r > q.r) move(--r, -1);
        ans[q.id] = nowAns;
    }
}
```

例题 [「国家集训队」小 Z 的袜子

题目大意：有一个长度为 n 的序列 $\{c_i\}$。现在给出 m 个询问，每次给出两个数 l, r，从编号在 l 到 r 之间的数中随机选出两个不同的数，求两个数相等的概率。

对于区间 $[l, r]$，以 l 所在块的编号为第一关键字，r 为第二关键字从小到大排序。

然后从序列的第一个询问开始计算答案，第一个询问通过直接暴力算出，复杂度为 $O(n)$，后面的询问在前一个询问的基础上得到答案。

具体做法：

对于区间 $[i, i]$，由于区间只有一个元素，我们很容易就能知道答案。然后一步一步从当前区间（已知答案）向下一个区间靠近。

我们设 $col[i]$ 表示当前颜色 i 出现了多少次，ans 当前共有多少种可行的配对方案（有多少种可以选到一双颜色相同的袜子），表示然后每次移动的时候更新答案——设当前颜色为 k，如果是增长区间就是 ans 加上 $C^2_{col[k]+1} - C^2_{col[k]}$，如果是缩短就是 ans 减去 $C^2_{col[k]} - C^2_{col[k]-1}$。

而这个询问的答案就是 $\dfrac{ans}{C^2_{r-l+1}}$。

这里有个优化：$C^2_a = \dfrac{a(a-1)}{2}$。

所以 $C^2_{a+1} - C^2_a = \dfrac{(a+1)a}{2} - \dfrac{a(a-1)}{2} = \dfrac{a}{2} \cdot (a+1-a+1) = \dfrac{a}{2} \cdot 2 = a$。

所以 $C^2_{col[k]+1} - C^2_{col[k]} = col[k]$。

算法总复杂度：$O(n\sqrt{n})$

下面的代码中 'deno' 表示答案的分母 (denominator)，'nume' 表示分子 (numerator)，'sqn' 表示块的大小：$\sqrt{n}$，'arr' 是输入的数组，'node' 是存储询问的结构体，'tab' 是询问序列（排序后的），'col' 同上所述。

```
#include <algorithm>
#include <cmath>
#include <cstdio>
using namespace std;
const int N = 50005;
int n, m, maxn;
int c[N];
```

```
long long sum;
int cnt[N];
long long ans1[N], ans2[N];

struct query {
    int l, r, id;

    bool operator<(const query &x) const {  // 重载<运算符
        if (l / maxn != x.l / maxn) return l < x.l;
        return (l / maxn) & 1 ? r < x.r : r > x.r;
    }
} a[N];

void add(int i) {
    sum += cnt[i];
    cnt[i]++;
}

void del(int i) {
    cnt[i]--;
    sum -= cnt[i];
}

long long gcd(long long a, long long b) { return b ? gcd(b, a % b) : a; }

int main() {
    scanf("%d%d", &n, &m);
    maxn = sqrt(n);
    for (int i = 1; i <= n; i++) scanf("%d", &c[i]);
    for (int i = 0; i < m; i++) scanf("%d%d", &a[i].l, &a[i].r), a[i].id = i;
    sort(a, a + m);
    for (int i = 0, l = 1, r = 0; i < m; i++) {
        // 具体实现
        if (a[i].l == a[i].r) {
            ans1[a[i].id] = 0, ans2[a[i].id] = 1;
            continue;
        }
        while (l > a[i].l) add(c[--l]);
        while (r < a[i].r) add(c[++r]);
        while (l < a[i].l) del(c[l++]);
        while (r > a[i].r) del(c[r--]);
        ans1[a[i].id] = sum;
        ans2[a[i].id] = (long long)(r - l + 1) * (r - l) / 2;
    }
    for (int i = 0; i < m; i++) {
        if (ans1[i] != 0) {
            long long g = gcd(ans1[i], ans2[i]);
            ans1[i] /= g, ans2[i] /= g;
        } else
        ans2[i] = 1;
        printf("%lld/%lld\n", ans1[i], ans2[i]);
    }
    return 0;
}
```

2 数据结构

2.1 单调队列/滑动窗口

有一个长为 n 的序列 a，以及一个大小为 k 的窗口。现在这个从左边开始向右滑动，每次滑动一个单位，求出每次滑动后窗口中的最大值和最小值。

```
int n, k;
std::cin >> n >> k;

std::vector<int> a(n);
for (int &i : a)
    std::cin >> i;

std::deque<int> q;
for (int i = 0; i < n; ++i) {  //输出最小值
    while (!q.empty() && a[q.back()] >= a[i])
        q.pop_back();
    q.emplace_back(i);
    if (i >= k - 1) {
        while (!q.empty() && q.front() <= i - k)
            q.pop_front();
        std::cout << a[q.front()] << ' ';
    }
}
std::cout << '\n';

while (!q.empty())
    q.pop_back();

for (int i = 0; i < n; ++i) {  //输出最大值
    while (!q.empty() && a[q.back()] <= a[i])
        q.pop_back();
    q.emplace_back(i);
    if (i >= k - 1) {
        while (!q.empty() && q.front() <= i - k)
            q.pop_front();
        std::cout << a[q.front()] << ' ';
    }
}
std::cout << '\n';
```

2.2 单调栈

输出右边第一个比 a_i 大的数的下标。

```
int n;
std::cin >> n;

std::vector<int> a(n), ans(n);
for (int &i : a)
    std::cin >> i;

std::stack<int> s;
for (int i = n - 1; i >= 0; --i) {
    while (!s.empty() && a[s.top()] <= a[i])
        s.pop();
    if (s.empty())
        ans[i] = -1;
    else
        ans[i] = s.top();
    s.emplace(i);
}
```

2.3 并查集

```
struct dsu {
    // 元素范围 [0, n)
    int n;
    std::vector<int> f, sz;

    dsu(int n) : n(n), f(n), sz(n, 1) { std::iota(f.begin(), f.end(), 0); }
    int father(int u) {
        if (u >= n) return -1;
        while (u != f[u]) u = f[u] = f[f[u]];
        return u;
    }
    bool same(int u, int v) {
        int a = father(u), b = father(v);
        return a == b && a != -1;
    }
    void merge(int u, int v) {
        int a = father(u), b = father(v);
        if (sz[a] > sz[b])
            f[b] = a, sz[a] += sz[b];
        else
            f[a] = b, sz[b] += sz[a];
    }
    int size(int u) { return sz[father(u)]; }
};
```

2.4 可撤销并查集

```
#include <iostream>
#include <stack>
#include <utility>

class UFS {
    private:
    int *fa, *rank;
    std::stack <std::pair <int*, int> > stk ;
    public:
    UFS() {}
    UFS(int n) {
        fa = new int[(const int)n + 1];
        rank = new int[(const int)n + 1];
        memset (rank, 0, n+1);
        for (int i = 1; i <= n; ++i) {
            fa [i] = i;
        }
    }
    inline int find(int x) {
        while (x ^ fa[x]) {
            x = fa[x];
        }
        return x ;
    }
    inline int Join (int x, int y) {
        x = find(x), y = find(y);
        if (x == y) {
            return 0;
        }
        if (rank[x] <= rank[y]) {
            stk.push(std::make_pair (fa + x, fa[x]));
            fa[x] = y;
            if (rank[x] == rank[y]) {
                stk.push(std::make_pair (rank + y, rank[y]));
                ++rank[y];
                return 2;
            }
            return 1 ;
        }
        stk.push(std::make_pair(fa + y, fa [y]));
        return fa[y] = x, 1;
    }
    inline void Undo ( )  {
        *stk.top( ).first = stk.top( ).second ;
        stk.pop( ) ;
    }
}T;
```

2.5 树状数组

```
struct fenwick {
    int n;
    std::vector<int> info;

    //[1, n]
    fenwick(int n) : n(n + 1), info(n + 1) {}
    void add(int pos, int val) {
        for (; pos < n; pos += pos & -pos)
        info[pos] += val;
    }
    int query(int pos) {
        int res = 0;
        for (; pos; pos -= pos & -pos)
        res += info[pos];
        return res;
    }
};
```

```
struct fenwick_2d {
    int n, m;
    std::vector<vector<int>> info;

    fenwick_2d(int n, int m) : n(n + 1), m(m + 1), info(n + 1, std::vector<int>(m + 1)) {}
    void add(int x, int y, int val) {
        for (; x < n; x += x & -x)
        for (; y < m; y += y & -y)
        info[x][y] += val;
    }
    int query(int x, int y) {
        int res = 0;
        for (; x; x -= x & -x)
        for (; y; y -= y & -y)
        res += info[x][y];
        return res;
    }
};
```

2.6 字典树

```
template <char MIN_CHAR = 'a', int ALPHABET = 26> struct trie {
```

```
    struct trie_node {
        std::array<int, ALPHABET> child;
        int words = 0;
        trie_node() { child.fill(-1); }
    };

    static const int ROOT = 0;
    std::vector<trie_node> nodes = {trie_node()
        };

    int get_or_create_child(int node, int c) {
        if (nodes[node].child[c] < 0) {
            nodes[node].child[c] = int(nodes.
                size());
            nodes.emplace_back();
        }

        return nodes[node].child[c];
    }

    int add_word(const std::string &word) {
        int node = ROOT;
        for (char c : word) node =
            get_or_create_child(node, c -
            MIN_CHAR);
        nodes[node].words++;
        return node;
    }

    int count_prefixes(const std::string &word,
        bool include_word) {
        int node = ROOT, count = 0;
        for (char c : word) {
            count += nodes[node].words;
            node = nodes[node].child[c -
                MIN_CHAR];
            if (node < 0) break;
        }
        if (include_word && node >= 0) count +=
            nodes[node].words;
        return count;
    }
};
```

2.7 线段树

普通线段数。单调修改、查询区间和：

```
template <class Info, class Merge = std::plus<
    Info>>
struct SegmentTree {
    const int n;
    const Merge merge;
    std::vector<Info> info;

    SegmentTree(int n) :
    n(n), merge(Merge()), info(4 << (31 -
        __builtin_clz(n))) {}
    SegmentTree(std::vector<Info> init) :
    SegmentTree(init.size()) {
        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, 0, n);
    }
    void pull(int p) { info[p] = merge(info[2 *
        p], info[2 * p + 1]); }
    void modify(int p, int l, int r, int x,
        const Info &v) {
        if (r - l == 1) {
            info[p] = info[p] + v;
            return;
        }
        int m = (l + r) / 2;
        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, 0, n, p, v); }
    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;
        return
            merge(rangeQuery(2 * p, l, m, x, y),
                 rangeQuery(2 * p + 1, m, r, x, y
                ));
    }
    Info rangeQuery(int l, int r) {
        return rangeQuery(1, 0, n, l, r);
    }
};

struct Info {
    int64_t x;
    Info(int x = 0) : x(x){};
};

Info operator+(const Info &lhs, const Info &rhs)
     {
    return lhs.x + rhs.x;
}
```

懒标记。区间修改和查询：

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

struct Info {
    int64_t x;
    Info(int64_t x = 0) : x(x) {}
};

Info operator+(const Info &lhs, const Info &rhs)
     { return Info(lhs.x + rhs.x); }

void apply(Info &a, int64_t b) { a.x += b; }
void apply(int64_t &a, int64_t b) { a += b; }
```

```cpp

template <class Info, class Tag, class Merge = std::plus<Info>> struct LazySegmentTree {
    const int n;
    const Merge merge;
    std::vector<Info> info;
    std::vector<Tag> tag;

    LazySegmentTree(int n) : n(n), merge(Merge()), info(4 << (31 - __builtin_clz(n))), tag(4 << (31 - __builtin_clz(n))) {}
    LazySegmentTree(std::vector<Info> init) : LazySegmentTree(init.size()) {
        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, 0, n);
    }
    void pull(int p) { info[p] = merge(info[2 * p], info[2 * p + 1]); }
    void apply(int p, int l, int r, const Tag &v) {
        ::apply(info[p], (r - l) * v);
        ::apply(tag[p], v);
    }
    void push(int p, int l, int r) {
        int mid = (l + r) / 2;
        apply(2 * p, l, mid, tag[p]);
        apply(2 * p + 1, mid, r, tag[p]);
        tag[p] = Tag();
    }
    void modify(int p, int l, int r, int x, const Info &v) {
        if (r - l == 1) {
            info[p] = info[p] + v;
            return;
        }
        int m = (l + r) / 2;
        push(p, l, r);
        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, 0, n, p, v); }
    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, l, r);
        return merge(rangeQuery(2 * p, l, m, x, y), rangeQuery(2 * p + 1, m, r, x, y));
    }
    Info rangeQuery(int l, int r) { return rangeQuery(1, 0, n, l, r); }
    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, l, r, v);
            return;
        }
        int m = (l + r) / 2;
        push(p, l, r);
        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, 0, n, l, r, v); }
};

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);

    int n, m;
    std::cin >> n >> m;

    std::vector<Info> a(n);
    for (Info &i : a) {
        std::cin >> i.x;
    }

    LazySegmentTree<Info, int64_t> seg(a);

    while (m--) {
        int op;
        std::cin >> op;

        if (op == 1) {
            int x, y, k;
            std::cin >> x >> y >> k;
            --x;
            seg.rangeApply(x, y, k);
        } else {
            int x, y;
            std::cin >> x >> y;
            --x;

            std::cout << seg.rangeQuery(x, y).x << '\n';
        }
    }
}
```

2.8 可持久化线段树

先引入一道题目：给定 n 个整数构成的序列 a，将对于指定的闭区间 $[l,r]$ 查询其区间内的第 k 小值。

主席树的主要思想就是：保存每次插入操作时的历史版本，以便查询区间第 k 小。

我们把问题简化一下：每次求 $[1,r]$ 区间内的 k 小值。

怎么做呢？只需要找到插入 r 时的根节点版本，然后用普通权值线段树（有的叫键值线段树/值域线段树）做就行了。

所以……如果需要得到 $[l,r]$ 的统计信息，只需要用 $[1,r]$ 的信息减去 $[1,l-1]$ 的信息就行了。

至此，该问题解决！

关于空间问题，我们分析一下：由于我们是动态开点的，所以一棵线段树只会出现 $2n-1$ 个结点。然后，有 n 次修改，每次至多增加 $\lceil \log_2 n \rceil + 1$ 个结点。因此，最坏情况下 n 次修改后的结点总数会达到 $2n-1+n(\lceil \log_2 n \rceil + 1)$。此题的 $n \le 10^5$，单次修改至多增加 $\lceil \log_2 10^5 \rceil + 1 = 18$ 个结点，故 n 次修改后的结点总数为 $2\times 10^5 - 1 + 18 \times 10^5$，忽略掉 -1，大概就是 20×10^5。

最后给一个忠告：千万不要吝啬空间（大多数题目中空间限制都较为宽松，因此一般不用担心空间超限的问题）！大胆一点，直接上个 $2^5 \times 10^5$，接近原空间的两倍（即"n « 5"）。

```cpp
#include <algorithm>
#include <cstdio>
#include <cstring>
using namespace std;
const int maxn = 1e5;  // 数据范围
int tot, n, m;
int sum[(maxn << 5) + 10], rt[maxn + 10], ls[(maxn << 5) + 10],
rs[(maxn << 5) + 10];
int a[maxn + 10], ind[maxn + 10], len;

inline int getid(const int &val) {  // 离散化
    return lower_bound(ind + 1, ind + len + 1, val) - ind;
}

int build(int l, int r) {  // 建树
    int root = ++tot;
    if (l == r) return root;
    int mid = l + r >> 1;
    ls[root] = build(l, mid);
    rs[root] = build(mid + 1, r);
    return root;  // 返回该子树的根节点
}

int update(int k, int l, int r, int root) {  // 插入操作
    int dir = ++tot;
    ls[dir] = ls[root], rs[dir] = rs[root], sum[dir] = sum[root] + 1;
    if (l == r) return dir;
    int mid = l + r >> 1;
    if (k <= mid)
    ls[dir] = update(k, l, mid, ls[dir]);
    else
    rs[dir] = update(k, mid + 1, r, rs[dir]);
    return dir;
}

int query(int u, int v, int l, int r, int k) {  // 查询操作
    int mid = l + r >> 1,
    x = sum[ls[v]] - sum[ls[u]];  // 通过区间减法得到左儿子中所存储的数值个数
    if (l == r) return l;
    if (k <= x)  // 若 k 小于等于 x ，则说明第 k 小的数字存储在在左儿子中
    return query(ls[u], ls[v], l, mid, k);
    else  // 否则说明在右儿子中
    return query(rs[u], rs[v], mid + 1, r, k - x);
}

inline void init() {
    scanf("%d%d", &n, &m);
    for (int i = 1; i <= n; ++i) scanf("%d", a + i);
    memcpy(ind, a, sizeof ind);
    sort(ind + 1, ind + n + 1);
    len = unique(ind + 1, ind + n + 1) - ind - 1;
    rt[0] = build(1, len);
    for (int i = 1; i <= n; ++i) rt[i] = update(getid(a[i]), 1, len, rt[i - 1]);
}

int l, r, k;

inline void work() {
    while (m--) {
        scanf("%d%d%d", &l, &r, &k);
        printf("%d\n", ind[query(rt[l - 1], rt[r], 1, len, k)]);  // 回答询问
    }
}

int main() {
    init();
    work();
    return 0;
}
```

3 字符串

3.1 字符串哈希

```cpp
struct hash_base_pow {
    int N, base;
    std::vector<uint64_t> val;

    hash_base_pow(int N, int base) : N(N), base(base), val(N + 1, 1) {
        for (int i = 1; i <= N; ++i) {
            val[i] = base * val[i - 1];
        }
    }
};

struct string_hash {
    size_t N;
    int base;
    std::vector<uint64_t> pre;
    std::vector<uint64_t>::const_iterator it_begin;

    string_hash(const std::string &str, const hash_base_pow &hbp, int base)
    : N(str.size()), base(base), pre(str.size() + 1, 0), it_begin(hbp.val.begin()) {
        for (size_t i = 1; i <= N; ++i) {
            pre[i] = base * pre[i - 1] + str[i - 1] + base;
        }
    }

```

```
    uint64_t sub(size_t l, size_t r) {  // [1, n]
        assert(l >= 1 && r <= N);
        return pre[r] - pre[l - 1] * *(it_begin + r - l + 1);
    }
};
```

3.2 KMP

```
/*
* Author: Simon
* 复杂度: O(n)
*/
int Next[maxn]; /*i之前相同前缀后缀的长度, 例如ababc,Next[5]=2; */
void getNext(int m,char p[]){
    memset(Next,0,sizeof(int)*(m+5));
    int k=-1,j=0;
    Next[0]=-1;
    while(j<m){
        if(k==-1||p[k]==p[j]){
            k++,j++;
            if(p[k]!=p[j]) Next[j]=k;
            else Next[j]=Next[k];
        }
        else k=Next[k];
    }
}
int KMP(int n,int m,char s[],char p[]){
    int i=0,j=0,ans=0;
    while(i<n){
        if(j==-1||s[i]==p[j]) i++,j++;
        else j=Next[j];
        if(j==m) ans++; /*计数(可重叠)*/
        //if(j==m) ans++,j=0;/*计数(不可重叠)*/
        //if(j==m) return i-m+1; /*返回第一个匹配的位置 */
    }
    //return j;/*返回s后缀与p前缀匹配的最长长度*/
    return ans;
}
```

3.3 exKMP

```
/*
* Author: nuchenghao
* 复杂度: O(n+m)
* 功能: 解决如下问题: 定义母串s和子串p, 设s的长度为n,p的长度为m,求p与s的每一个后缀的最长公共前缀
*       extend[i]表示p与s[i,n-1]的最长公共前缀。(0<=i<n)
*       一个辅助工具为nxt[i]表示p[i,m-1]和p本身的最长公共前缀长度
*       下标都从0开始
*/

int nxt[maxn], ex[maxn]; /*ex数组即为extend数组*/
/*预处理计算next数组 */
void getnxt(char *s) {
    memset(nxt,0,sizeof(nxt));
    int i = 0, j, po, len = strlen(s);
    nxt[0] = len;   /*初始化next[0] */
    while (s[i] == s[i + 1] && i + 1 < len) i++;
     /*计算next[1] */
    nxt[1] = i; po = 1; /*初始化po的位置 */
    for (i = 2; i < len; i++) {
        if (nxt[i - po] + i < nxt[po] + po) nxt[i] = nxt[i - po];/*第一种情况, 可以直接得到next[i]的值 */
        else { /*第二种情况, 要继续匹配才能得到next[i]的值 */
            j = nxt[po] + po - i;
            if (j < 0) j = 0; /*如果i>po+nxt[po],则要从头开始匹配 */
            while (i + j < len && s[j] == s[j + i]) j++; /*计算next[i] */
            nxt[i] = j;
            po = i; /*更新po的位置 */
        }
    }
}
/*计算extend数组 */
void exkmp(char *s, char *p) {
    memset(ex,0,sizeof(ex));
    int i = 0, j, po, len = strlen(s), l2 = strlen(p);
    getnxt(p); /*计算子串的next数组 */
    while (s[i] == p[i] && i < l2 && i < len) i++;/*计算ex[0] */
    ex[0] = i; po = 0; //初始化po的位置
    for (i = 1; i < len; i++) {
        if (nxt[i - po] + i < ex[po] + po) ex[i] = nxt[i - po]; /*第一种情况, 直接可以得到ex[i]的值 */
        else {  /*第二种情况, 要继续匹配才能得到ex[i]的值 */
            j = ex[po] + po - i;
            if (j < 0) j = 0; /*如果i>ex[po]+po则要从头开始匹配 */
            while (i + j < len && j < l2 && s[j + i] == p[j]) j++;  /*计算ex[i]*/
            ex[i] = j; po = i; //更新po的位置
        }
    }
}
```

3.4 Manacher

```
#include <bits/stdc++.h>
using namespace std;
// Manacher
/*
* 求最长回文子串
*/
const int MAXN = 110010;

char Ma[MAXN * 2];
int Mp[MAXN * 2];
void Manacher(char s[], int len) {
    int l = 0;
```

```
    Ma[l++] = '$';
    Ma[l++] = '#';
    for (int i = 0; i < len; i++) {
        Ma[l++] = s[i];
        Ma[l++] = '#';
    }
    Ma[l] = 0;
    int mx = 0, id = 0;
    for (int i = 0; i < l; i++) {
        Mp[i] = mx > i ? min(Mp[2 * id - i], mx - i) : 1;
        while (Ma[i + Mp[i]] == Ma[i - Mp[i]]) Mp[i]++;
        if (i + Mp[i] > mx) {
            mx = i + Mp[i];
            id = i;
        }
    }
}
/*
* abaaba
* i: 0 1 2 3 4 5 6 7 8 9 10 11 12 13
* Ma[i]: $ # a # b # a # a # b # a #
* Mp[i]: 1 1 2 1 4 1 2 7 2 1 4 1 2 1
*/
char s[MAXN];
int main() {
    while (scanf("%s", s) == 1) {
        int len = strlen(s);
        Manacher(s, len);
        int ans = 0;
        for (int i = 0; i < 2 * len + 2; i++) ans = max(ans, Mp[i] - 1);
        printf("%d\n", ans);
    }
    return 0;
}
```

4 图论

4.1 LCA

倍增:

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

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);

    int n, m, s;
    std::cin >> n >> m >> s;
    --s;

    std::vector G(n, std::vector<int>());

    for (int i = 1; i < n; ++i) {
        int u, v;
        std::cin >> u >> v;
        --u, --v;

        G[u].emplace_back(v);
        G[v].emplace_back(u);
    }

    std::vector fa(n, std::vector<int>(20, -1));
    //fa[u][i] 表示u的第2^i个父亲
    std::vector<int> depth(n);

    std::function<void(int, int)>
    dfs = [&](int u, int f) {
        depth[u] = depth[f] + 1;
        fa[u][0] = f;

        for (int i = 1; (1 << i) <= depth[u]; ++i)
        fa[u][i] = fa[fa[u][i - 1]][i - 1];

        for (int v : G[u])
        if (v != f) dfs(v, u);
    };

    auto lca = [&](int x, int y) {
        if (depth[x] < depth[y]) std::swap(x, y);

        for (int i = 19; i >= 0; --i)
        if (depth[x] - (1 << i) >= depth[y])
        x = fa[x][i];

        if (x == y) return x;

        for (int i = 19; i >= 0; --i)
        if (fa[x][i] != fa[y][i]) {
            x = fa[x][i];
            y = fa[y][i];
        }

        return fa[x][0];
    };

    dfs(s, s);

    while (m--) {
        int x, y;
        std::cin >> x >> y;
        --x, --y;

        std::cout << lca(x, y) + 1 << '\n';
    }
}
```

targan:

```
int n, root, cnt;
int pre[maxn], ans[maxn];
vector<int> v[maxn], s[maxn], num[maxn];

int find(int x) { return pre[x] == x ? x : pre[x] = find(pre[x]); }

void dfs(int u) {
    pre[u] = u;
    for (int i = 0; i < v[u].size(); i++) {
        int to = v[u][i];
        dfs(to);
        pre[find(pre[to])] = find(pre[u]);
    }
    for (int i = 0; i < s[u].size(); i++) {
        int to = s[u][i];
        if (pre[to] != to)
```

```
        ans[num[u][i]] = find(pre[to]);
    }
}

/*
for (int i = 1; i <= q; i++) {
    scanf("%d%d", &x, &y);
    if (x == y) ans[i] = x;
    s[x].push_back(y);
    s[y].push_back(x);
    num[x].push_back(i);
    num[y].push_back(i);
}
dfs(root);
*/
```

4.2 最小生成树

Kruscal:

```
/*
* Author: Simon
* 复杂度: mlog(m)
* 功能: 适用于稀疏图求MST
*/
namespace Kruskal{
    int Set[maxn], Rank[maxn],cnt=0;//并查集 rank
    struct node{
        int u,v,w;
        node(){}
        node(int u,int v,int w):u(u),v(v),w(w){}
        bool operator <(const node&a)const{
            return w<a.w;
        }
    }g[maxm];//结构体储存
    void init(int n) {
        for (int i = 0; i <= n; i++) {
            Rank[i] = 0; Set[i] = i;
        }
    }
    void addedge(int u,int v,int w){
        g[cnt++]=node(u,v,w);
    }
    int find(int x) {
        int root = x;
        while (root != Set[root]) root = Set[root];
        while (x != root) {
            int t = Set[x];
            Set[x] = root;
            x = t;
        }
        return root;
    }
    void unite(int x, int y) {
        x = find(x); y = find(y);
        if (Rank[x] < Rank[y]) {
            Set[x] = y;
        } else {
            Set[y] = x;
            if (Rank[x] == Rank[y]) Rank[x]++;
        }
    }
    //n为边的数量, m为村庄的数量
    int Run(int n, int m) {
        int num = 0, res = 0; //将边按照权值从小到大排序
        sort(g,g+cnt);
        for (int i = 0; i < n && num != m - 1; i++) {
            int u=g[i].u,v=g[i].v,w=g[i].w;
            //判断当前这条边的两个端点是否属于同一棵树
            if (find(u) != find(v)) {
                unite(u, v);
                res += w;
                num++;
            }
        }
        //如果加入边的数量小于m - 1,则表明该无向图不连通,等价于不存在最小生成树
        if (num < m - 1) res = -1;
        return res;
    }
}
```

prim:

```
/*
* Author: Simon
* 复杂度: nlog(m)
* 功能: 适用于求稠密图求MST
*/
namespace Prim{
    int head[maxn],cnt=0;
    bool vis[maxn];
    struct node{
        int v,w,next;
        node(){}
        node(int v,int w,int next):v(v),w(w),next(next){}
        bool operator <(const node a)const{
            return w>a.w;
        }
    }g[maxm];
    void init(){
        memset(vis,0,sizeof(vis));
        memset(head,-1,sizeof(head));cnt=0;
    }
    void addedge(int u,int v,int w){
        g[cnt]=node(v,w,head[u]);
        head[u]=cnt++;
    }
    int Run(int s,int n){
        priority_queue<node>q;
        q.push(node(s,0));int ans=0;
        while(!q.empty()){
            node now=q.top();q.pop();
            if(vis[now.v]) continue;
            vis[now.v]=1;ans+=now.w;
            for(int i=head[now.v];~i;i=g[i].next){
                if(!vis[g[i].v]) q.push(g[i]);
            }
        }
        return ans;
    }
}
```

4.3 最短路

SPFA:

```
/*
* Author: Simon
* 复杂度: nlog(m)
* 功能: 适用于求稠密图求MST
*/
namespace Prim{
    int head[maxn],cnt=0;
    bool vis[maxn];
    struct node{
        int v,w,next;
        node(){}
        node(int v,int w,int next):v(v),w(w),
            next(next){}
        bool operator <(const node a)const{
            return w>a.w;
        }
    }g[maxm];
    void init(){
        memset(vis,0,sizeof(vis));
        memset(head,-1,sizeof(head));cnt=0;
    }
    void addedge(int u,int v,int w){
        g[cnt]=node(v,w,head[u]);
        head[u]=cnt++;
    }
    int Run(int s,int n){
        priority_queue<node>q;
        q.push(node(s,0));int ans=0;
        while(!q.empty()){
            node now=q.top();q.pop();
            if(vis[now.v]) continue;
            vis[now.v]=1;ans+=now.w;
            for(int i=head[now.v];~i;i=g[i].next
                ){
                if(!vis[g[i].v]) q.push(g[i]);
            }
        }
        return ans;
    }
}
```

dijkstra:

```
/*
* Author: Simon
* 复杂度: O(e · log(n))
*/
namespace Dijkstra{
    struct node{
        int v,w,next;
        node(){}
        node(int v,int w,int next=-1):v(v),w(w),
            next(next){}
        bool operator <(const node&a)const{
            return w>a.w;
        }
    }g[maxn];
    int head[maxn],cnt=0,dis[maxn];
    bool vis[maxn];
    void addedge(int u,int v,int w){
        g[cnt]=node(v,w,head[u]);
        head[u]=cnt++;
    }
    void init(){
        memset(head,-1,sizeof(head)); cnt=0;
    }
    void Run(int n,int r){
        fill(dis,dis+n+5,INF);dis[r]=0;
        memset(vis,0,sizeof(vis));
        priority_queue<node>q;
        q.push({r,0});
        while(!q.empty()){
            node now=q.top();q.pop();
            int &u=now.v;
            if(vis[u]) continue; vis[u]=1;
            for(int i=head[u];~i;i=g[i].next){
                int &v=g[i].v,&w=g[i].w;
                if(!vis[v]&&dis[u]+w<dis[v]){
                    dis[v]=dis[u]+w;
                    q.push({v,dis[v]});
                }
            }
        }
    }
}
```

floyd:

```
//#define inf maxn*maxw+10
for(int i = 0; i < n; i++) {
    for(int j = 0; j < n; j++) {
        d[i][j] = inf;
    }
}
d[0][0] = 0;
for(int k = 0; k < n; k++) {
    for(int i = 0; i < n; i++) {
        for(int j = 0; j < n; j++) {
            d[i][j] = std::min(d[i][j], d[i][k]
                + d[k][j]);
        }
    }
}
/*
* Author: Simon
* 功能: 传递闭包
*/
for(int k=1;k<=n;k++){
    for(int i=1;i<=n;i++){
        for(int j=1;j<=n;j++){
            if(a[i][k]&&a[k][j]) a[i][j]=1;  /*
                传递闭包 */
        }
    }
}\texttt{}
```

4.4 欧拉回路

```
const int maxn = 100;

int n;
int step;
int path[maxn];

void find_path_u(int now, int mat[][maxn]) {
    for (int i=n-1; i>=0; i--) {
        while (mat[now][i]) {
```

```
            mat[now][i]--, mat[i][now]--;
            find_path_u(i, mat);
        }
    }
    path[step++] = now;
}

void find_path_d(int now, int mat[][maxn]) {
    for (int i=n-1; i>=0; i--) {
        while (mat[now][i]) {
            mat[now][i]--;
            find_path_d(i, mat);
        }
    }
    path[step++] = now;
}


int euler_circuit(int start, int mat[][maxn]) {
    step = 0;
    find_path_u(start, mat);
    // find_path_d(start, mat);
    return step;
}

int main() {

}
```

4.5 K 短路

```
#include <cstdio>
#include <cstring>
#include <queue>
#include <vector>
#include <algorithm>
using namespace std;

const int maxn = 10000 + 5;
const int INF = 0x3f3f3f3f;
int s, t, k;

bool vis[maxn];
int dist[maxn];

struct Node {
    int v, c;
    Node (int _v = 0, int _c = 0) : v(_v), c(_c) {}
    bool operator < (const Node &rhs) const {
        return c + dist[v] > rhs.c + dist[rhs.v];
    }
};

struct Edge {
    int v, cost;
    Edge (int _v = 0, int _cost = 0) : v(_v), cost(_cost) {}
};

vector<Edge>E[maxn], revE[maxn];

void Dijkstra(int n, int s) {
    memset(vis, false, sizeof(vis));
    for (int i = 1; i <= n; i++) dist[i] = INF;
    priority_queue<Node>que;
    dist[s] = 0;
    que.push(Node(s, 0));
    while (!que.empty()) {
        Node tep = que.top(); que.pop();
        int u = tep.v;
        if (vis[u]) continue;
        vis[u] = true;
        for (int i = 0; i < (int)E[u].size(); i++) {
            int v = E[u][i].v;
            int cost = E[u][i].cost;
            if (!vis[v] && dist[v] > dist[u] + cost) {
                dist[v] = dist[u] + cost;
                que.push(Node(v, dist[v]));
            }
        }
    }
}

int astar(int s) {
    priority_queue<Node> que;
    que.push(Node(s, 0)); k--;
    while (!que.empty()) {
        Node pre = que.top(); que.pop();
        int u = pre.v;
        if (u == t) {
            if (k) k--;
            else return pre.c;
        }
        for (int i = 0; i < (int)revE[u].size(); i++) {
            int v = revE[u][i].v;
            int c = revE[u][i].cost;
            que.push(Node(v, pre.c + c));
        }
    }
    return -1;
}

void addedge(int u, int v, int w) {
    revE[u].push_back(Edge(v, w));
    E[v].push_back(Edge(u, w));
}

int main() {
    int n, m, u, v, w;
    while (scanf("%d%d", &n, &m) != EOF) {
        for (int i = 0; i <= n; i++) {
            E[i].clear();
            revE[i].clear();
        }
        int aaa;
        scanf("%d%d%d%d", &s, &t, &k, &aaa);
        for (int i = 0; i < m; i++) {
            scanf("%d%d%d", &u, &v, &w);
            addedge(u, v, w);
        }
        Dijkstra(n, t);
        if (dist[s] == INF) {
            printf("No Solution\n");
            continue;
```

```
        }
        if (s == t) k++;
        ans = astar(s);
    }
    return 0;
}
```

4.6 强连通分量

tarjan:

```
/*
* Author: Simon
* 功能: 求强连通分量
*/
int dfn[maxn],low[maxn],Stack[maxn],Belong[maxn];
int index=0,scnt=0,bcnt=0;
void Tarjan(int u){
    dfn[u]=low[u]=++index;
    Stack[++scnt]=u;vis[u]=1;
    for(int i=head[u];~i;i=g[i].next){
        int &v=g[i].v;
        if(!dfn[v]){
            Tarjan(v);
            low[u]=min(low[u],low[v]);
        }
        else if(vis[v]) low[u]=min(low[u],dfn[v]);
    }
    if(dfn[u]==low[u]){
        bcnt++;int t;
        do{
            t=Stack[scnt--];vis[t]=0;
            Belong[t]=bcnt;
        }while(t!=u);
    }
}
void solve(int n){
    memset(dfn,0,sizeof(dfn));
    index=bcnt=scnt=0;
    for(int i=1;i<=n;i++){
        if(!dfn[i]) Tarjan(i);
    }
}
/*
* Author: Simon
* 功能: 缩点建立新图
*/
for(int u=1;u<=n;u++){
    for(int i=head1[u];~i;i=g[i].next){
        int &v=g[i].v;
        if(Belong[u]!=Belong[v]){
            addedge(g2,head2,cnt2,Belong[v],Belong[u]);
            rdu[Belong[u]]++;
        }
    }
}
/*
* Author: Simon
* 功能: 双联通算法
*/
int dfn[maxn],low[maxn],Stack[maxn],Belong[maxn];
int index=0,scnt=0,bcnt=0;
int tarjan(int u,int fa)
{
    dfn[u]=low[u]++index;
    for(int i=head[u];~i;i=g[i].next){
        int &v=g[i].v;
        if(!dfn[v])
        {
            Stack[++scnt]=v;/*搜索到的点入栈  */
            tarjan(edges[i].to,u);
            low[u]=min(low[u],low[v]);
            if(low[v]>=dfn[u])/*是割点或根  */
            {
                bcnt++;int t;
                do{
                    t=Stack[scnt--];
                    Belong[t]=bcnt;
                }while(t!=v);
                while(stack[top]!=edges[i].to)/*将点出栈直到目标点  */
                bcc[num].push_back(stack[top--]);
                bcc[num].push_back(stack[top--]);/*目标点出栈  */
                bcc[num].push_back(u);/*不要忘了将当前点存入bcc  */
            }
        }
        else if(edges[i].to!=fa)
        lowu=min(lowu,dfn[edges[i].to]);
    }
}
```

4.7 次小生成树

Kruscal:

```
//1-indexed
struct edge {
    int s, t, w;    //从s到t 权值w
    bool vis;
    edge() {}
    edge(int s, int t, int w) :s(s), t(t), w(w) {}
    bool operator < (const edge e) const {
        return w < e.w;
    }
}e[maxm];

int pre[maxn];
int Max[maxn][maxn];    // Max[i][j]表示从i到j路径上的最大边权

int find(int x) {
    int r = x, i = x, j;
    while (pre[r] != r)
    r = pre[r];
    while (i != r) {    // 状态压缩
        j = pre[i];
        pre[i] = r;
        i = j;
    }
    return r;
}

```

```
int kruskal(int n, int m) { // n为边数 m为点数
    int lef = m - 1, ans = 0;
    memset(Max, 0, sizeof(Max));
    vector<int>v[maxn];
    for (int i = 1; i <= m; i++) {
        pre[i] = i;
        v[i].push_back(i);
    }
    sort(e + 1, e + n + 1);
    for (int i = 1; i <= n; i++) {
        int fs = find(e[i].s), ft = find(e[i].t)
            , len1, len2;
        if (fs != ft) {
            pre[fs] = ft;
            ans += e[i].w;
            lef--; e[i].vis = true;
            len1 = v[fs].size(), len2 = v[ft].
                size();
            for (int j = 0; j < len1; j++)
            for (int k = 0; k < len2; k++)
            Max[v[fs][j]][v[ft][k]] = Max[v[ft][
                k]][v[fs][j]] = e[i].w;
            int tmp[maxn];
            for (int j = 0; j < len1; j++)
            tmp[j] = v[fs][j];
            for (int j = 0; j < len2; j++)
            v[fs].push_back(v[ft][j]);
            for (int j = 0; j < len1; j++)
            v[ft].push_back(tmp[j]);
        }
        if (!lef)break;
    }
    if (lef) ans = -1;  // 图不连通
    return ans;
}

int SMST(int n, int ans) {  // n为边数,ans为最小
    生成树权值
    int ret = INF;
    for (int i = 1; i <= n; i++)
    if (!e[i].vis)
    ret = min(ret, ans + e[i].w - Max[e[i].s][e[
        i].t]);
    if (ret == INF) return -1;
    return ret;
}
```

Prim:

```
//1-indexed
struct edge {
    int s, t, w;    //从s到t 权值w
    bool vis;
    edge() {}
    edge(int s, int t, int w) :s(s), t(t), w(w)
        {}
    bool operator < (const edge e) const {
        return w < e.w;
    }
}e[maxm];

int pre[maxn];
int Max[maxn][maxn];    // Max[i][j]表示从i到j路
    径上的最大边权

int find(int x) {
    int r = x, i = x, j;
    while (pre[r] != r)
    r = pre[r];
    while (i != r) {    // 状态压缩
        j = pre[i];
        pre[i] = r;
        i = j;
    }
    return r;
}

int kruskal(int n, int m) { // n为边数 m为点数
    int lef = m - 1, ans = 0;
    memset(Max, 0, sizeof(Max));
    vector<int>v[maxn];
    for (int i = 1; i <= m; i++) {
        pre[i] = i;
        v[i].push_back(i);
    }
    sort(e + 1, e + n + 1);
    for (int i = 1; i <= n; i++) {
        int fs = find(e[i].s), ft = find(e[i].t)
            , len1, len2;
        if (fs != ft) {
            pre[fs] = ft;
            ans += e[i].w;
            lef--; e[i].vis = true;
            len1 = v[fs].size(), len2 = v[ft].
                size();
            for (int j = 0; j < len1; j++)
            for (int k = 0; k < len2; k++)
            Max[v[fs][j]][v[ft][k]] = Max[v[ft][
                k]][v[fs][j]] = e[i].w;
            int tmp[maxn];
            for (int j = 0; j < len1; j++)
            tmp[j] = v[fs][j];
            for (int j = 0; j < len2; j++)
            v[fs].push_back(v[ft][j]);
            for (int j = 0; j < len1; j++)
            v[ft].push_back(tmp[j]);
        }
        if (!lef)break;
    }
    if (lef) ans = -1;  // 图不连通
    return ans;
}

int SMST(int n, int ans) {  // n为边数,ans为最小
    生成树权值
    int ret = INF;
    for (int i = 1; i <= n; i++)
    if (!e[i].vis)
    ret = min(ret, ans + e[i].w - Max[e[i].s][e[
        i].t]);
    if (ret == INF) return -1;
    return ret;
}
```

4.8 最大流

给定一个包含 n 个点 m 条边的有向图，并给定每条边的容量，边的容量非负。

图中可能存在重边和自环。求从点 S 到点 T 的最大流。

输入格式第一行包含四个整数 n,m,S,T。

接下来 m 行，每行三个整数 u,v,c，表示从点 u 到点 v 存在一条有向边，容量为 c。
点的编号从 1 到 n。
输出格式输出点 S 到点 T 的最大流。
如果从点 S 无法到达点 T 则输出 0。

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

using namespace std;

#define int long long
#define endl '\n'

const int N = 1e4 + 10, M = 2e5 + 10, inf = 0x3f3f3f3f;

int n, m, st, ed;
int h[N], e[M], w[M], ne[M], idx;
int d[N], cu[N];

void add(int a, int b, int c) {
    e[idx] = b, w[idx] = c, ne[idx] = h[a], h[a] = idx++;
    e[idx] = a, w[idx] = 0, ne[idx] = h[b], h[b] = idx++;
}

bool bfs() {
    memset(d, -1, sizeof d); d[st] = 0;
    queue<int> q; q.push(st);
    cu[st] = h[st];
    while (q.size()) {
        int ver = q.front(); q.pop();
        for (int i = h[ver]; ~i; i = ne[i]) {
            int to = e[i];
            if (d[to] == -1 && w[i]) {
                d[to] = d[ver] + 1;
                cu[to] = h[to];
                if (to == ed) return 1;
                q.push(to);
            }
        }
    }
    return 0;
}

int Find(int pre, int limit) {
    if (pre == ed) return limit;
    int flow = 0;
    for (int i = cu[pre]; ~i && flow < limit; i = ne[i]) {
        cu[pre] = i;
        int to = e[i];
        if (d[to] == d[pre] + 1 && w[i]) {
            int tmp = Find(to, min(w[i], limit - flow));
            if (!tmp) d[to] = -1;
            w[i] -= tmp, w[i ^ 1] += tmp, flow += tmp;
        }
    }
    return flow;
}

int dinic() {
    int maxf = 0, flow;
    while (bfs()) while (flow = Find(st, inf)) maxf += flow;
    return maxf;
}

void oper() {
    cin >> n >> m >> st >> ed;
    memset(h, -1, sizeof h); idx = 0;
    for (int i = 1; i <= m; i++) {
        int a, b, c; cin >> a >> b >> c;
        add(a, b, c);
    }
    cout << dinic() << endl;
}

signed main() {
    ios::sync_with_stdio(0), cin.tie(0), cout.tie(0);
    int t = 1; //cin >> t;
    while (t--) oper();
    return 0;
}
```

```cpp
/*
* Author: Simon,Ttmq36
* 复杂度: O(V^2·E)
*/
namespace Dinic{
    struct node{
        int v,w,next;
        node(){}
        node(int v,int w,int next):v(v),w(w),next(next){}
    }g[maxn<<1];
    int head[maxn],cur[maxn],cnt=0;
    int dep[maxn];
    void init(){
        memset(head,-1,sizeof(head));cnt=0;
    }
    void addedge(int u,int v,int w){
        g[cnt]=node(v,w,head[u]);
        head[u]=cnt++;
        g[cnt]=node(u,0,head[v]);
        head[v]=cnt++;
    }
    bool bfs(int s,int t){
        memset(dep,0,sizeof(dep));
        queue<int>q;q.push(s);
        dep[s]=1; cur[s]=head[s];
        while(!q.empty()){
            int u=q.front();q.pop();
            for(int i=head[u];~i;i=g[i].next){
                int &v=g[i].v,&w=g[i].w;
                if(!dep[v]&&w>0){
                    dep[v] = dep[u] + 1;
                    cur[v] = head[v];
                    q.push(v);
                }
            }
        }
        if(!dep[t]) return 0;
        return 1;
    }
    int dfs(int s,int t,int f){
        if(s==t) return f;
```

```
        int flow=0,u=s;
        for(int &i=cur[u];~i;i=g[i].next){
            int &v=g[i].v,&w=g[i].w;
            if(dep[u]+1==dep[v]&&w>0){
                int d=dfs(v,t,min(f,w));
                if(d>0){
                    flow+=d; f-=d;
                    g[i].w-=d,g[i^1].w+=d;
                    if(f<=0) break;
                }
            }
        }
        if(!flow) dep[u]=-INF;
        return flow;
    }
    int maxFlow(int s,int t){
        int ans=0;
        while(bfs(s,t)){
            ans+=dfs(s,t,INF);
        }
        return ans;
    }
}
```

4.9 启发式合并

一天，小徐的好友邀请他去吃布丁，于是小徐高高兴兴的来到好友家。

哇，这么多五彩缤纷的布丁！

好友说：“在我们开吃前先玩会儿游戏吧。”

于是他将布丁摆成一行，接着说：“我可以把某种颜色的布丁全部变成另一种颜色，我还会在某些时刻问你当前一共有多少段颜色。例如：颜色分别为 1,2,2,1 的四个布丁一共有 3 段颜色。”

输入格式第一行包含整数 n 和 m，分别表示布丁的个数和好友的操作次数。

第二行包含 n 个空格隔开的整数 A1,A2,…,An，其中 Ai 表示第 i 个布丁的颜色。

从第三行起的 m 行，依次描述 m 个操作。

对每个操作，若第一个数是 1，则表示好友要改变颜色，这时后跟两个整数 x 和 y（可能相等），表示执行该操作后所有颜色为 x 的布丁被变成颜色 y。

若第一个数是 2，则表示好友要询问目前有多少段颜色，这时应该输出一个整数回答。

输出格式对于每个询问，在一行中输出一个整数作为回答。

数据范围 0<n,m<100001, 0<Ai,x,y<106

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

using namespace std;

#define int long long

const int N = 1e5 + 10;

int n, m;
int rt, idx;

struct type {
    int s[2], v, p;
    int sz, flag;
    int& l = s[0], & r = s[1];
    void init(int vv, int pp) {
        v = vv, p = pp;
        sz = 1;
    }
}tree[N];

void pushup(int x) {
    tree[x].sz = tree[tree[x].l].sz + tree[tree[
        x].r].sz + 1;
}

void pushdown(int x) {
    if (tree[x].flag) {
        swap(tree[x].l, tree[x].r);
        tree[tree[x].l].flag ^= 1;
        tree[tree[x].r].flag ^= 1;
        tree[x].flag = 0;
    }
}

void rotate(int x) {
    int y = tree[x].p, z = tree[y].p, k = tree[y
        ].r == x;
    int& b = tree[x].s[k ^ 1];
    tree[b].p = y; tree[y].p = x; tree[x].p = z;
    tree[y].s[k] = b;
    b = y;
    tree[z].s[tree[z].r == y] = x;
    pushup(y); pushup(x);
}

void splay(int x, int k) {
    while (tree[x].p != k) {
        int y = tree[x].p, z = tree[y].p;
        if (z != k) {
            if ((tree[y].r == x) ^ tree[z].r ==
                y) rotate(x);
            else rotate(y);
        }
        rotate(x);
    }
    if (!k) rt = x;
}

void insert(int v) {
    int x = rt, p = 0;
    while (x) p = x, x = tree[x].s[v > tree[x].v
        ];
    x = ++idx;
    if (p) tree[p].s[v > tree[p].v] = x;
    tree[x].init(v, p);
    splay(x, 0);
}

int getp(int k) {
    int x = rt;
    while (1) {
        pushdown(x);
        int sz = tree[tree[x].l].sz;
        if (sz >= k) x = tree[x].l;
        else if (sz + 1 == k) return x;
        else k -= sz + 1, x = tree[x].r;
    }
}

void print(int x) {
    pushdown(x);
    if (tree[x].l) print(tree[x].l);
    if (tree[x].v >= 1 && tree[x].v <= n) cout
```

```
        << tree[x].v << " ";
    if (tree[x].r) print(tree[x].r);
}

void oper() {
    cin >> n >> m;
    for (int i = 0; i <= n + 1; i++) insert(i);
    for (int i = 1; i <= m; i++) {
        int l, r; cin >> l >> r;
        l = getp(l), r = getp(r + 2);
        splay(l, 0), splay(r, l);
        tree[tree[r].l].flag ^= 1;
    }
    print(rt);
}

signed main() {
    ios::sync_with_stdio(0), cin.tie(0), cout.
        tie(0);
    int t = 1; //cin >> t;
    while (t--) oper();
    return 0;
}
```

5 动态规划

5.1 背包

```
/*01背包 */
void ZeroOnepark(int V /*背包容量*/, int val /*
    物品价值*/, int vol /*物品体积*/) {
    for (int j = V; j >= vol; j--) {
        dp[j] = max(dp[j], dp[j - vol] + val);
    }
}

/*完全背包 */
void Completepark(int V /*背包容量*/, int val /*
    物品价值*/, int vol /*物品体积*/) {
    for (int j = vol; j <= V; j++) {
        dp[j] = max(dp[j], dp[j - vol] + val);
    }
}

/*多重背包 */
void Multiplepark(int val /*物品价值*/, int vol
    /*物品体积*/, int amount /*物品数量*/) {
    if (vol * amount >= v) {
        Completepark(val, vol);
    } else {
        int k = 1;
        while (k < amount) {
            ZeroOnepark(k * val, k * vol);
            amount -= k;
            k <<= 1;
        }
        if (amount > 0) {
            ZeroOnepark(amount * val, amount *
                vol);
        }
    }
}
int main() {
    memset(dp, 0, sizeof(dp));
    for (int i = 1; i <= n; i++) {
        Multiplepark(value[i], volume[i], num[i
            ]);
    }
}
```

5.2 最长上升子序列

```
int arr[maxn], n;

template<class Cmp>
int LIS (Cmp cmp) {
    static int m, end[maxn];
    m = 0;
    for (int i=0; i<n; i++) {
        int pos = lower_bound(end, end+m, arr[i
            ], cmp)-end;
        end[pos] = arr[i], m += pos==m;
    }
    return m;
}

bool greater1(int value) {
    return value >=1;
}

/*********
std::cout << LIS(std::less<int>()) << std::endl;
          //严格上升
std::cout << LIS(std::less_equal<int>()) << std
    ::endl;   //非严格上升
std::cout << LIS(std::greater<int>()) << std::
    endl;      //严格下降
std::cout << LIS(std::greater_equal<int>()) <<
    std::endl;//非严格下降
std::cout << count_if(a,a+7,std::greater1) <<
    std::endl; //计数
**********/
```

5.3 最长公共子序列

```
int dp[maxn][maxn];

void LCS(int n1, int n2, int A[], int B[]) {
    for(int i=1; i<=n1; i++) {
        for(int j=1; j<=n2; j++) {
            dp[i][j] = dp[i-1][j];
            if (dp[i][j-1] > dp[i][j]) {
                dp[i][j] = dp[i][j-1];
            }
            if (A[i] == B[j] && dp[i-1][j-1] + 1
                 > dp[i][j]) {
                dp[i][j] = dp[i-1][j-1] + 1;
            }
        }
    }
}
```

5.4 数位 dp

```cpp
typedef long long ll;
int a[20];
ll dp[20][state];//不同题目状态不同
ll dfs(int pos,/*state变量*/,bool lead/*前导零*/
    ,bool limit/*数位上界变量*/)//不是每个题都要
    判断前导零
{
    //递归边界，既然是按位枚举，最低位是0，那么
        pos==-1说明这个数我枚举完了
    if(pos==-1) return 1;/*这里一般返回1，表示你
        枚举的这个数是合法的，那么这里就需要你在
        枚举时必须每一位都要满足题目条件，也就是
        说当前枚举到pos位，一定要保证前面已经枚举
        的数位是合法的。不过具体题目不同或者写法
        不同的话不一定要返回1 */
    //第二个就是记忆化(在此前可能不同题目还能有
        一些剪枝)
    if(!limit && !lead && dp[pos][state]!=-1)
        return dp[pos][state];
    /*常规写法都是在没有限制的条件记忆化，这里与
        下面记录状态是对应，具体为什么是有条件的
        记忆化后面会讲*/
    int up=limit?a[pos]:9;//根据limit判断枚举的
        上界up;这个的例子前面用213讲过了
    ll ans=0;
    //开始计数
    for(int i=0;i<=up;i++)//枚举，然后把不同情况
        的个数加到ans就可以了
    {
        if() ...
        else if()...
        ans+=dfs(pos-1,/*状态转移*/,lead && i
            ==0,limit && i==a[pos]) //最后两个变
            量传参都是这样写的
        /*这里还算比较灵活，不过做几个题就觉得这
            里也是套路了
        大概就是说，我当前数位枚举的数是i，然后
            根据题目的约束条件分类讨论
        去计算不同情况下的个数，还有要根据state
            变量来保证i的合法性，比如题目
        要求数位上不能有62连续出现,那么就是state
            就是要保存前一位pre,然后分类，
        前一位如果是6那么这意味就不能是2，这里一
            定要保存枚举的这个数是合法*/
    }
    //计算完，记录状态
    if(!limit && !lead) dp[pos][state]=ans;
    /*这里对应上面的记忆化，在一定条件下时记录，
        保证一致性，当然如果约束条件不需要考虑
        lead，这里就是lead就完全不用考虑了*/
    return ans;
}

ll solve(ll x)
{
    int pos=0;
    while(x)//把数位都分解出来
    {
        a[pos++]=x%10;//个人老是喜欢编号为[0,pos
            ),看不惯的就按自己习惯来，反正注意数
            位边界就行
        x/=10;
    }
    return dfs(pos-1/*从最高位开始枚举*/,/*一系
        列状态 */,true,true);//刚开始最高位都是有
        限制并且有前导零的，显然比最高位还要高的
        一位视为0嘛
}

int main()
{
    ll le,ri;
    while(~scanf("%lld%lld",&le,&ri))
    {
        //初始化dp数组为-1,这里还有更加优美的优
            化,后面讲
        printf("%lld\n",solve(ri)-solve(le-1));
    }
}
```

5.5 区间 dp

```cpp
for (int x = 0; x < n; x++){//枚举长度
    for (int i = 1; i + x <= n; i++){//枚举起点
        dp[i][i] = 1;
        int j = x + i;//终点
        dp[i][j] = dp[i + 1][j] + 1;
        for (int k = i + 1; k <= j; k++) {
            if (a[i] == a[k])
            dp[i][j] = min(dp[i][j], dp[i][k -
                1] + dp[k + 1][j]);
        }
    }
}
```

6 搜索

6.1 双向搜索

双向同时搜索：

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

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

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

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

Meet in the middle

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

它适用于输入数据较小，但还没小到能直接使用暴力搜索的情况。

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

例题：

有 n 盏灯，每盏灯与若干盏灯相连，每盏灯上都有一个开关，如果按下一盏灯上的开关，这盏灯以及与之相连的所有灯的开关状态

都会改变。一开始所有灯都是关着的，你需要将所有灯打开，求最小的按开关次数。

$1 \le n \le 35$。

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

using namespace std;

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

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

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

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

    for (int i = 0; i < (1 << (n - n / 2)); ++i) {  // 对后一半进行搜索
        long long t = 0;
        int cnt = 0;
        for (int j = 0; j < (n - n / 2); ++j) {
            if ((i >> j) & 1) {
                t ^= a[n / 2 + j];
                ++cnt;
            }
        }
        if (f.count((((long long)1 << n) - 1) ^ t))
        ans = min(ans, cnt + f[(((long long)1 << n) - 1) ^ t]);
    }

    cout << ans;

    return 0;
}
```

6.2 启发式搜索

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

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

「NOIP2005 普及组」采药

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

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

估价函数 f 的运行过程如下：

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

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

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

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

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

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

int main() {
    scanf("%d %d", &m, &n);
    for (int i = 1; i <= n; i++) {
        scanf("%d %d", &node[i].a, &node[i].b);
        node[i].f = 1.0 * node[i].b / node[i].a;
            // f为性价比
    }
```

```
    sort(node + 1, node + n + 1);  // 根据性价比排序
    work(1, m, 0);
    printf("%d\n", ans);
    return 0;
}
```

7 数学

7.1 exgcd

```
/*
* Author: Simon
* 复杂度: O(log(max(a,b)))
* 功能: 求解a*x+b*y=c
*/

/*解a*x+b*y=gcd(a,b)*/
int exgcd(int a, int b, int &x, int &y) {
    if (b == 0) {
        x = 1, y = 0;
        return a;
    }
    int g = exgcd(b, a % b, y, x);
    y -= a / b * x;
    return g;
}
/*
* 解a*x+b*y=c
* 假设有一对特解x=n,y=m
* 则其通解为: x=n-b*t or y=m+a*t
*/
void solve(int n, int A, int B, int C) {
    int x, y;
    int g = exgcd(A, B, x, y);
    if (C % g != 0) {
        cout << "-1" << endl;
        return;
    }
    A /= g, B /= g, C /= g;
    x *= C % B;
    x = (x % B + B) % B; y = (C - A * x) / B; /*求最小非负整数x, 则y=(c-a*x)/b */
    // y *= C % A;
    // y = (y % A + A) % A; x = (C - B * y) / A; /*求最小非负整数y, 则x=(c-b*y)/a */
    /*
    具体题目
    */
}
```

7.2 欧拉函数

```
int Phi(int n) {
    int m = sqrt(n + 0.5);
    int ans = n;
    for (int i = 2; i <= m; ++i) {
        if (n % i == 0) {
            ans = ans - ans / i;
            while (n % i == 0) n /= i;
        }
    }
    if (n > 1) ans = ans - ans / n;
    return ans;
}
/*
* Author: Simon
* 复杂度: O(n)
* 功能: 求[1,n]小于i且与i互质的数的个数phi(i)
*/
int prime[maxn], phi[maxn], cnt = 0;
bool vis[maxn];
void Euler(int n) {
    phi[1] = 1;
    for (int i = 2; i <= n; i++) {
        if (!vis[i]) {
            prime[++cnt] = i;
            phi[i] = i - 1;
        }
        for (int j = 1; j <= cnt && i * prime[j] <= n; j++) {
            vis[i * prime[j]] = 1;
            if (i % prime[j] == 0) {
                phi[i * prime[j]] = phi[i] * prime[j];
                break;
            }
            phi[i * prime[j]] = phi[i] * (prime[j] - 1);
        }
    }
}
```

7.3 逆元

```
LL Inv(LL a, LL n){
    return PowMod(a, EulerPhi(n) - 1, n);
    //return PowMod(a,n-2,n); //n为素数
}

int Inv(int a, int n) {
    int d, x, y;
    d = extended_euclid(a, n, x, y);
    if(d == 1)  return (x%n + n) % n;
    else     return -1; // no solution
}
// by Simon
int inv[maxn];
int Inv(int n,int p){ //线性求1~n的逆元
    inv[1] = 1;
    for (int i = 2; i <= n; ++i) inv[i] = (long long)(p - p / i) * inv[p % i] % p;
}
//by Simon
int a[maxn]/*n个数 */,s[maxn]/*前缀积 */,inv[maxn]/*第i个数的逆元 */;
int Inv(int n,int p){ //线性求任意n个数的逆元
    s[0] = 1;
    for (int i = 1; i <= n; ++i) s[i] = s[i - 1] * a[i] % p; //前缀积
    sv[n] = fpow(s[n], p - 2);      // 当然这里也可以用 exgcd 来求逆元,视个人喜好而定.
    for (int i = n; i >= 1; --i) sv[i - 1] = sv[i] * a[i] % p; //前缀积的逆元
    for (int i = 1; i <= n; ++i) inv[i] = sv[i] * s[i - 1] % p;
```

```
}
```

7.4 组合数

```
LL C(const LL &n, const LL &m, const int &pr) {
    LL ans = 1;
    for (int i = 1; i <= m; i++) {
        LL a = (n - m + i) % pr;
        LL b = i % pr;
        ans = (ans * (a * Inv(b, pr) % pr) % pr)
            % pr;
    }
    return ans;
}
/*
* Author: Simon
* 功能: 求组合数
* 复杂度: O(p+log(p))
*/
int bit[maxn]; //阶乘数组，对p取模，则只需初始化
    1~p的阶乘即可。
int C(const int n,const int m,const int p){
    if (n < m) return 0;
    return bit[n] * fpow(bit[m], p - 2, p) % p *
        fpow(bit[n - m], p - 2, p) % p;
}
```

Lucas:

```
//C(n, m) mod p(n 很大 p 较小(不知道能不能为非素
    数)
LL Lucas(LL n, LL m, const int &pr) {
    if (m == 0) return 1;
    return C(n % pr, m % pr, pr) * Lucas(n / pr,
        m / pr, pr) % pr;
}
```

计算从 C(n, 0) 到 C(n, p) 的值

```
//by Yuhao Du
int p;
std::vector<int> gao(int n) {
    std::vector<int> ret(p+1,0);
    if (n==0) {
        ret[0]=1;
    } else if (n%2==0) {
        std::vector<int> c = gao(n/2);
        for(int i = 0; i <= p+1; i++) {
            for(int j = 0; j <= p+1; j++) {
                if (i+j<=p) ret[i+j]+=c[i]*c[j];
            }
        }
    } else {
        std::vector<int> c = gao(n-1);
        for(int i = 0; i <= p+1; i++) {
            for(int j = 0; j <= 2; j++) {
                if (i+j<=p) ret[i+j]+=c[i];
            }
        }
    }
    return ret;
}
```

7.5 博弈论

1. Nim Game 最经典最基础的博弈. n 堆石子, 双方轮流从任意一堆石子中取出至少一个, 不能取的人输. 对于一堆 x 个石子的情况, 容易用归纳法得到 SG(x)=x. 所以所有石子个数的异或和为 0 是必败态, 否则为必胜态.
2. Bash Game 每人最多一次只能取 m 个石子, 其他规则同 Nim Game. 依旧数学归纳…SG(x)=xmod(m+1).
3. NimK Game 每人一次可以从最多 K 堆石子中取出任意多个, 其他规则同 Nim Game. 结论: 在二进制下各位上各堆石子的数字之和均为 (K+1) 的倍数的话则为必败态, 否则为必胜态. 这个证明要回到原始的方法上去. 补: 这个游戏还可以推广, 即一个由 n 个子游戏组成的游戏, 每次可以在最多 K 个子游戏中进行操作. 然后只要把结论中各堆石子的个数改为各个子游戏的 SG 值即可, 证明也还是一样的.
4. Anti-Nim Game 似乎又叫做 Misère Nim. 不能取的一方获胜, 其他规则同 Nim Game. 关于所谓的" Anti-SG 游戏" 及" SJ 定理" 贾志鹏的论文上有详细说明, 不过似乎遇到并不多. 结论是一个状态是必胜态当且仅当满足以下条件之一: SG 值不为 0 且至少有一堆石子数大于 1; SG 值为 0 且不存在石子数大于 1 的石子堆.
5. Fibonacci Nim 有一堆个数为 n(n>=2) 的石子，游戏双方轮流取石子，规则如下：1) 先手不能在第一次把所有的石子取完，至少取 1 颗；2) 之后每次可以取的石子数至少为 1，至多为对手刚取的石子数的 2 倍。约定取走最后一个石子的人为赢家。结论：当 n 为 Fibonacci 数的时候，必败。
6. Staircase Nim 每人一次可以从第一堆石子中取走若干个, 或者从其他石子堆的一堆中取出若干个放到左边一堆里 (没有石子的石子堆不会消失), 其他规则同 Nim Game. 这个游戏的结论比较神奇: 当且仅当奇数编号堆的石子数异或和为 0 时为必败态. 简单的理解是从偶数编号堆中取石子对手又可以放回到奇数编号堆中, 而且不会让对手不能移动. 比较意识流, 然而可以归纳证明.
7. Wythoff Game 有两堆石子, 双方轮流从某一堆取走若干石子或者从两堆中取走相同数目的石子, 不能取的人输. 容易推理得出对任意自然数 k, 都存在唯一的一个必败态使得两堆石子数差为 k, 设其为 Pk=(ak,bk), 表示石子数分别为 ak,bk(ak<=bk). 那么 ak 为在 Pk0(k0<k) 中未出现过的最小自然数,bk=ak+k. 数学班的说, 用 Betty 定理以及显然的单调性就可以推出神奇的结论: ak=floor(k*5√+12),bk=floor(k*5√+32).
8. Take & Break 有 n 堆石子, 双方轮流取出一堆石子, 然后新增两堆规模更小的石子堆 (可以没有石子), 无法操作者输. 这个游戏似乎只能暴力 SG, 知道一下就好.
9. 树上删边游戏给出一个有 n 个结点的树，有一个点作为树的根节点, 双方轮流从树中删去一条边边，之后不与根节点相连的部分将被移走, 无法操作者输. 结论是叶子结点的 SG 值为 0, 其他结点 SG 值为其每个儿子结点 SG 值加 1 后的异或和, 证明也并不复杂.
10. 翻硬币游戏 n 枚硬币排成一排，有的正面朝上，有的反面朝上。游戏者根据某些约束翻硬币（如：每次只能翻一或两枚，或者每次只能翻连续的几枚），但他所翻动的硬币中，最右边的必须是从正面翻到反面。谁不能翻谁输。
11. 需要先开动脑筋把游戏转化为其他的取石子游戏之类的, 然后用如下定理解决: 局面的 SG 值等于局面中每个正面朝上的棋子单一存在时的 SG 值的异或和。
12. 无向图删边游戏一个无向连通图，有一个点作为图的根。游戏者轮流从图中删去边，删去一条边后，不与根节点相连的部分将被移走。谁无路可走谁输。
13. 对于这个模型，有一个著名的定理——Fusion Principle: 我们可以对无向图做如下改动：将图中的任意一个偶环缩成一个新点，任意一个奇环缩成一个新点加一个新边；所有连到原先环上的边全部改为与新点相连。这样的改动不会影响图的 SG 值。

SG 函数:

```
/*
* author: Simon
* f[m]:可改变当前状态的方式，m为方式的种类，f[m]要在getSG之前先预处理
* sg[]:0~n的SG函数值
* mex[]:为x后继状态的集合
* 若sg值为正数，则先手必赢，否则若为0，则先手必输。
*/
int f[maxn],sg[maxn],mex[maxn];
void getSG(int n/*需要求多少个sg值*/,int m/*有多少种操作方式*/){
    memset(sg,0,sizeof(sg));
    for(int i = 1; i <= n; i++){ /*因为SG[0]始终等于0，所以i从1开始*/
        memset(mex,0,sizeof(mex)); /*每一次都要将上一状态的后继集合重置*/
        for(int j = 0; f[j] <= i && j < m; j++)
        mex[sg[i-f[j]]] = 1;  /*将后继状态的SG函数值进行标记*/
        for(int j = 0;; j++) if(!mex[j]){   /*查询当前后继状态SG值中最小的非零值*/
            sg[i] = j;
            break;
        }
    }
}
```

7.6 容斥

```
/*
* Author: Simon
* 复杂度: O(2^k+sqrt(n))其中k为n中不同的质因数的个数，n<=10^5时，k最大为5
* 功能: 容斥求[1,n]中与p互质的数的和
*/
/*求n的质因数*/
void solve(int n){ /*需预处理欧拉筛*/
    fac.clear();
    for(int i=1;i<=cnt&&prime[i]*prime[i]<=n;i++){
        if(n%prime[i]==0){
            fac.push_back(prime[i]);
            while(n%prime[i]==0) n/=prime[i];
        }
    }
    if(n>1) fac.push_back(n);
}
/*求和公式*/
int sum_1(int n){
    return n*(n+1)/2;
}
/*容斥求[1,n]中与p互质的数的和*/
int cal(int n,int p){
    solve(p); /*求出p的质因数*/
    int ans=0;
    for(int i=1;i<(1<<fac.size());i++){ /*枚举子集*/
        int num=0,lcm=1;
        for(int j=0;j<fac.size();j++){
            if(i>>j&1){
                num++;lcm*=fac[j];
            }
        }
        if(num&1) ans+=sum_1(n)-sum_1(n/lcm)*lcm; /*总和减去不互质的数的和*/
        else ans-=sum_1(n)-sum_1(n/lcm)*lcm;
    }
    return ans;
}
```

7.7 素性检验

```
bool is_prime(int32_t n) {
    if (n <= 1) {
        return false;
    }
    if (n == 2 || n == 7 || n == 61) {
        return true;
    }
    if (n % 2 == 0) {
        return false;
    }
    int64_t d = n - 1;
    while (d % 2 == 0) {
        d /= 2;
    }
    constexpr int64_t bases[3] = {2, 7, 61};
    for (int64_t a : bases) {
        int64_t t = d;
        int64_t y = pow_mod(a, t, n);
        while (t != n - 1 && y != 1 && y != n - 1) {
            y = y * y % n;
            t <<= 1;
        }
        if (y != n - 1 && t % 2 == 0) {
            return false;
        }
    }
    return true;
}
```

Rabin-Miller:

```
bool millerRabin(int n) {
    if (n < 3 || n % 2 == 0) return n == 2;
    int a = n - 1, b = 0;
    while (a % 2 == 0) a /= 2, ++b;
    // test_time 为测试次数,建议设为不小于 8
    // 的整数以保证正确率,但也不宜过大,否则会影响效率
    for (int i = 1, j; i <= test_time; ++i) {
        int x = rand() % (n - 2) + 2, v = quickPow(x, a, n);
        if (v == 1) continue;
        for (j = 0; j < b; ++j) {
            if (v == n - 1) break;
            v = (long long)v * v % n;
        }
        if (j >= b) return 0;
    }
    return 1;
}
```

7.8 线性筛

```
struct prime {
    size_t max_num;
    std::vector<int> primes;
    std::deque<bool> visted;

    prime(int _max_num) : max_num(size_t(
        _max_num + 1)), primes(), visted(size_t(
        _max_num + 1), false) {
        for (size_t i = 2; i <= max_num; ++i) {
            if (!visted[i]) primes.emplace_back(
                i);
            for (size_t j = 0; j < primes.size()
                 && (__int128_t)(primes[j] * i)
                <= max_num; ++j) {
                visted[primes[j] * i] = 1;
                if (i % primes[j] == 0) break;
            }
        }
    }

    bool is_prime(int x) {
        if (x < 2 || x > int(max_num)) return
            false;
        return !visted[x];
    }
};
```

7.9 常用公式

1. 约数定理：若 $n=\prod_{i=1}^{k} p_i^{a_i}$，则
   (a) 约数个数 $f(n)=\prod_{i=1}^{k}(a_i+1)$
   (b) 约数和 $g(n)=\prod_{i=1}^{k}(\sum_{j=0}^{a_i} p_i^j)$
2. 小于 n 且互素的数之和为 $n\varphi(n)/2$
3. 若 $gcd(n,i)=1$，则 $gcd(n,n-i)=1(1\le i\le n)$
4. 错排公式：$D(n)=(n-1)(D(n-2)+D(n-1))=\sum_{i=2}^{n}\frac{(-1)^k n!}{k!}=[\frac{n!}{e}+0.5]$
5. 威尔逊定理：$p\ is\ prime\ \Rightarrow (p-1)!\equiv -1(mod\ p)$
6. 欧拉定理：$gcd(a,n)=1\Rightarrow a^{\varphi(n)}\equiv 1(mod\ n)$
7. 欧拉定理推广：$gcd(n,p)=1\Rightarrow a^n\equiv a^{n\%\varphi(p)}(mod\ p)$
8. 素数定理：对于不大于 n 的素数个数 $\pi(n)$，$\lim_{n\to\infty}\pi(n)=\frac{n}{\ln n}$
9. 位数公式：正整数 x 的位数 $N=log10(n)+1$
10. 斯特灵公式 $n!\approx\sqrt{2\pi n}(\frac{n}{e})^n$
11. 设 $a>1,m,n>0$，则 $gcd(a^m-1,a^n-1)=a^{gcd(m,n)}-1$
12. 设 $a>b,gcd(a,b)=1$，则 $gcd(a^m-b^m,a^n-b^n)=a^{gcd(m,n)}-b^{gcd(m,n)}$

$$G=gcd(C_n^1,C_n^2,...,C_n^{n-1})=\begin{cases}n, & n\ \text{is prime}\\ 1, & n\ \text{has multy prime factors}\\ p, & n\ \text{has single prime factor}\ p\end{cases}$$

$gcd(Fib(m),Fib(n))=Fib(gcd(m,n))$

13. 若 $gcd(m,n)=1$，则：
   (a) 最大不能组合的数为 $m*n-m-n$
   (b) 不能组合数个数 $N=\frac{(m-1)(n-1)}{2}$
14. $(n+1)lcm(C_n^0,C_n^1,...,C_n^{n-1},C_n^n)=lcm(1,2,...,n+1)$
15. 若 p 为素数，则 $(x+y+...+w)^p\equiv x^p+y^p+...+w^p(mod\ p)$
16. 卡特兰数：1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, 208012
    $h(0)=h(1)=1,h(n)=\frac{(4n-2)h(n-1)}{n+1}=\frac{C_{2n}^n}{n+1}=C_{2n}^n-C_{2n}^{n-1}$
17. 伯努利数：$B_n=-\frac{1}{n+1}\sum_{i=0}^{n-1}C_{n+1}^i B_i$

$$\sum_{i=1}^{n} i^k=\frac{1}{k+1}\sum_{i=1}^{k+1}C_{k+1}^i B_{k+1-i}(n+1)^i$$

18. 小于 n 的 i，j，gcd(i,j)=1 的 i,j 的对数与欧拉函数的关系 $\sum_{i=1}^{n}\sum_{j=1}^{n}[gcd(i,j)=1]=2\sum_{i=1}^{n}\sum_{j=1}^{i}[gcd(i,j)=1]-\sum_{i=1}^{n}[gcd(i,i)=1]=\left(2\sum_{i=1}^{n}\varphi(i)\right)-1$
19. 小于 n 的 i,j，且 gcd(i,j)=1 时，所有 i*j 的和与欧拉函数的关系 $\sum_{i=1}^{n} i\sum_{j=1}^{n}[gcd(i,j)=1]\cdot j=2\sum_{i=1}^{n} i\sum_{j=1}^{i}[gcd(i,j)=1]\cdot j-\sum_{i=1}^{n}[gcd(i,i)=1]\cdot i$
    $=\left(2\sum_{i=1}^{n} i\frac{i\cdot\varphi(i)+[i=1]}{2}\right)-1=\left(\sum_{i=1}^{n} i^2\cdot\varphi(i)+[i=1]\right)-1$
20. 约数，倍数之间重要的变换 $\sum_{k=1}^{n}\sum_{d|k} d\cdot k=\sum_{k=1}^{n}\sum_{d=1}^{\frac{n}{k}} d\cdot k\cdot d=\sum_{d=1}^{n}\sum_{k|d} d\cdot\frac{d}{k}=\sum_{d=1}^{n}\sum_{k=1}^{\frac{n}{d}} d\cdot k\cdot d=\sum_{d=1}^{n}\sum_{d|k} d\cdot k=\sum_{k=1}^{n}\sum_{k|d} d\cdot\frac{d}{k}$
21. $C_m^n=C_{m-1}^{n-1}+C_{m-1}^n$
22. $C_n^m=\frac{n!}{m!(n-m)!}$
23. $C_n^m=C_n^{n-m}=C_{n-1}^m+C_{n-1}^{m-1}$（杨辉三角）
24. $C_n^0+C_n^1+C_n^2+...+C_n^n=\sum_{i=0}^{n}C_n^i=2^n$
25. $C_n^0+C_n^2+C_n^4+...=C_n^1+C_n^3+C_n^5+...=2^{n-1}$
26. $C_n^m+C_{n+1}^m+C_{n+2}^m+...+C_{n+m}^m=\sum_{i=0}^{m}C_{n+i}^m=C_{n+m+1}^{m+1}$
27. $kC_n^k=nC_{n-1}^{k-1}$，$\frac{C_n^k}{k+1}=\frac{C_{n+1}^{k+1}}{n+1}$
28. $\sum_{k=0}^{n} k*C_m^k=\sum_{k=0}^{n-1} m*C_{m-1}^k$
29. $C_m^n\%p=C_{m/p}^{n/p}*C_{m\%p}^{n\%p}\%p$（Lucas 定理）
30. $C_{m+n}^k=\sum_{i=0}^{k}C_m^i C_n^{k-i}$（Vandermonde 恒等式）
31. $\frac{n!}{\prod_{i=1}^{k} a_i}$（有重复元素的全排列公式，$a_i$ 为第 i 种元素的个数，k 为元素种类数）

8 trick

8.1 离散化

```
//数组离散化 含重复元素
std::sort(sub_a, sub_a + n);
int size = std::unique(sub_a, sub_a + n) - sub_a
    ;  //size为离散化后元素个数
for (i = 0; i < n; i++) {
    a[i] = std::lower_bound(sub_a, sub_a + size,
         a[i]) - sub_a + 1;  //k为b[i]经离散化后
        对应的值
}

//坐标离散化
int compress(int *x1, int *x2, int w) {
    std::vector<int> xs;
    for (int i = 0; i < N; i++) {
```

```
        for (int d = -1; d <= 1; d++) {
            int tx1 = x1[i] + d, tx2 = x2[i] + d;
            if (1 <= tx1 && tx1 <= w) xs.push_back(tx1);
            if (1 <= tx2 && tx2 <= w) xs.push_back(tx2);
        }
    }
    std::sort(xs.begin(), xs.end());
    xs.erase(unique(xs.begin(), xs.end()), xs.end());
    for (int i = 0; i < N; i++) {
        x1[i] = find(xs.begin(), xs.end(), x1[i]) - xs.begin();
        x2[i] = find(xs.begin(), xs.end(), x2[i]) - xs.begin();
    }
    return xs.size();
}
```

8.2 bitset

```
b.any()             b中是否存在置为1的二进制位？
b.none()            b中不存在置位1的二进制位吗？
b.count()           b中置为1的二进制位的个数
b.size()            b中二进制位的个数
b[pos]              访问b中在pos处的二进制位
b.test(pos)         b中在pos处的二进制位是否为1？
b.set()             把b中所有二进制位都置为1
b.set(pos)          把b中在pos处的二进制位置为1
b.reset()           把b中所有二进制位置为0
b.reset(pos)        把b中在pos处的二进制位置为0
b.flip()            把b中所有二进制位逐位取反
b.flip(pos)         把b中在pos处的二进制位取反
b.to\_ulong()       用b中同样的二进制位返回一个unsigned long值
os << b             把b中的位集中输出到os流
```

8.3 位运算

```
x >> 1                  //去掉最后一位
x << 1                  //在最后加一个0
x << 1 + 1              //在最后加一个1
x | 1                   //把最后一位变成1
x | 1 - 1               //把最后一位变成0
x ^ 1                   //最后一位取反
x | (1 <<(k-1))         //把右数第k位变成1
x & ~ (1 << (k-1))      //把右数第k位变成0
x ^ (1 << (k-1))        //右数第k位取反
x & 7                   //取末三位
x & (1 << k-1)          //取末k位
x >>(k-1) & 1           //取右数第k位
x |(1 << k-1)           //把末k位变成1
x ^ (1 << k-1)          //末k位取反
x & (x+1)               //把右边连续的1变成0
((1<<x-1)               //x个1
(x>>16)+(x&((1<<16)-1)) //二进制里1的数量
```

8.4 内置函数

```
— Built-in Function: int __builtin_ffs (unsigned int x)
Returns one plus the index of the least significant 1-bit of x, or if x is zero, returns zero.
返回右起第一个‘1’的位置。

— Built-in Function: int __builtin_clz (unsigned int x)
Returns the number of leading 0-bits in x, starting at the most significant bit position. If x is 0, the result is undefined.
返回左起第一个‘1’之前0的个数。

— Built-in Function: int __builtin_ctz (unsigned int x)
Returns the number of trailing 0-bits in x, starting at the least significant bit position. If x is 0, the result is undefined.
返回右起第一个‘1’之后的0的个数。

— Built-in Function: int __builtin_popcount (unsigned int x)
Returns the number of 1-bits in x.
返回‘1’的个数。

— Built-in Function: int __builtin_parity (unsigned int x)
Returns the parity of x, i.e. the number of 1-bits in x modulo 2.
返回‘1’的个数的奇偶性。

— Built-in Function: int __builtin_ffsl (unsigned long)
Similar to __builtin_ffs, except the argument type is unsigned long.

— Built-in Function: int __builtin_clzl (unsigned long)
Similar to __builtin_clz, except the argument type is unsigned long.

— Built-in Function: int __builtin_ctzl (unsigned long)
Similar to __builtin_ctz, except the argument type is unsigned long.

— Built-in Function: int __builtin_popcountl (unsigned long)
Similar to __builtin_popcount, except the argument type is unsigned long.

— Built-in Function: int __builtin_parityl (unsigned long)
Similar to __builtin_parity, except the argument type is unsigned long.

— Built-in Function: int __builtin_ffsll (unsigned long long)
Similar to __builtin_ffs, except the argument type is unsigned long long.

— Built-in Function: int __builtin_clzll (
```

```
    unsigned long long)
Similar to __builtin_clz, except the argument
    type is unsigned long long.

— Built-in Function: int __builtin_ctzll (
    unsigned long long)
Similar to __builtin_ctz, except the argument
    type is unsigned long long.

— Built-in Function: int __builtin_popcountll (
    unsigned long long)
Similar to __builtin_popcount, except the
    argument type is unsigned long long.

— Built-in Function: int __builtin_parityll (
    unsigned long long)
Similar to __builtin_parity, except the argument
     type is unsigned long long.



随机数函数:
default_random_engine: 随机非负数（不建议单独使
    用）。
uniform_int_distribution: 指定范围的随机非负数。
uniform_real_distribution: 指定范围的随机实数。
bernoulli_distribution: 指定概率的随机布尔值。

示例:default_random_engine e;
uniform_int_distribution<int>u(0,9);
cout<<u(e)<<endl;
```

8.5 模拟退火

```
#include <cmath>
#include <cstdio>
#include <cstdlib>
#include <ctime>

const int N = 10005;
int n, x[N], y[N], w[N];
double ansx, ansy, dis;

double Rand() { return (double)rand() / RAND_MAX
    ; }

double calc(double xx, double yy) {
    double res = 0;
    for (int i = 1; i <= n; ++i) {
        double dx = x[i] - xx, dy = y[i] - yy;
        res += sqrt(dx * dx + dy * dy) * w[i];
    }
    if (res < dis) dis = res, ansx = xx, ansy =
        yy;
    return res;
}

void simulateAnneal() {
    double t = 100000;
    double nowx = ansx, nowy = ansy;
    while (t > 0.001) {
        double nxtx = nowx + t * (Rand() * 2 -
            1);
        double nxty = nowy + t * (Rand() * 2 -
            1);
        double delta = calc(nxtx, nxty) - calc(
            nowx, nowy);
        if (exp(-delta / t) > Rand()) nowx =
            nxtx, nowy = nxty;
        t *= 0.97;
    }
    for (int i = 1; i <= 1000; ++i) {
        double nxtx = ansx + t * (Rand() * 2 -
            1);
        double nxty = ansy + t * (Rand() * 2 -
            1);
        calc(nxtx, nxty);
    }
}

int main() {
    srand(0);  // 注意，在实际使用中，不应使用固
        定的随机种子。
    scanf("%d", &n);
    for (int i = 1; i <= n; ++i) {
        scanf("%d%d%d", &x[i], &y[i], &w[i]);
        ansx += x[i], ansy += y[i];
    }
    ansx /= n, ansy /= n, dis = calc(ansx, ansy)
        ;
    simulateAnneal();
    printf("%.3lf %.3lf\n", ansx, ansy);
    return 0;
}
```

8.6 几何公式

```
三角形:
1. 半周长 P=(a+b+c)/2
2. 面积 S=aHa/2=absin(C)/2=sqrt(P(P-a)(P-b)(P-c)
    )
3. 中线 Ma=sqrt(2(b^2+c^2)-a^2)/2=sqrt(b^2+c^2+2
    bccos(A))/2
4. 角平分线 Ta=sqrt(bc((b+c)^2-a^2))/(b+c)=2
    bccos(A/2)/(b+c)
5. 高线 Ha=bsin(C)=csin(B)=sqrt(b^2-((a^2+b^2-c
    ^2)/(2a))^2)
6. 内切圆半径 r=S/P=asin(B/2)sin(C/2)/sin((B+C)
    /2)
=4Rsin(A/2)sin(B/2)sin(C/2)=sqrt((P-a)(P-b)(P-c)
    /P)
=Ptan(A/2)tan(B/2)tan(C/2)
7. 外接圆半径 R=abc/(4S)=a/(2sin(A))=b/(2sin(B))
    =c/(2sin(C))


四边形:
D1,D2为对角线,M对角线中点连线,A为对角线夹角
1. a^2+b^2+c^2+d^2=D1^2+D2^2+4M^2
2. S=D1D2sin(A)/2
(以下对圆的内接四边形)
3. ac+bd=D1D2
4. S=sqrt((P-a)(P-b)(P-c)(P-d)),P为半周长


正n边形:
R为外接圆半径,r为内切圆半径
```

```
1. 中心角 A=2PI/n
2. 内角 C=(n-2)PI/n
3. 边长 a=2sqrt(R^2-r^2)=2Rsin(A/2)=2rtan(A/2)
4. 面积 S=nar/2=nr^2tan(A/2)=nR^2sin(A)/2=na^2/(4tan(A/2))


圆:
1. 弧长 l=rA
2. 弦长 a=2sqrt(2hr-h^2)=2rsin(A/2)
3. 弓形高 h=r-sqrt(r^2-a^2/4)=r(1-cos(A/2))=atan(A/4)/2
4. 扇形面积 S1=rl/2=r^2A/2
5. 弓形面积 S2=(rl-a(r-h))/2=r^2(A-sin(A))/2


棱柱:
1. 体积 V=Ah,A为底面积,h为高
2. 侧面积 S=lp,l为棱长,p为直截面周长
3. 全面积 T=S+2A


棱锥:
1. 体积 V=Ah/3,A为底面积,h为高
(以下对正棱锥)
2. 侧面积 S=lp/2,l为斜高,p为底面周长
3. 全面积 T=S+A


棱台:
1. 体积 V=(A1+A2+sqrt(A1A2))h/3,A1.A2为上下底面积,h为高
(以下为正棱台)
2. 侧面积 S=(p1+p2)l/2,p1.p2为上下底面周长,l为斜高
3. 全面积 T=S+A1+A2


圆柱:
1. 侧面积 S=2PIrh
2. 全面积 T=2PIr(h+r)
3. 体积 V=PIr^2h


圆锥:
1. 母线 l=sqrt(h^2+r^2)
2. 侧面积 S=PIrl
3. 全面积 T=PIr(l+r)
4. 体积 V=PIr^2h/3


圆台:
1. 母线 l=sqrt(h^2+(r1-r2)^2)
2. 侧面积 S=PI(r1+r2)l
3. 全面积 T=PIr1(l+r1)+PIr2(l+r2)
4. 体积 V=PI(r1^2+r2^2+r1r2)h/3


球:
1. 全面积 T=4PIr^2
2. 体积 V=4PIr^3/3


球台:
1. 侧面积 S=2PIrh
2. 全面积 T=PI(2rh+r1^2+r2^2)
3. 体积 V=PIh(3(r1^2+r2^2)+h^2)/6


球扇形:
1. 全面积 T=PIr(2h+r0),h为球冠高,r0为球冠底面半径
2. 体积 V=2PIr^2h/3
```