ICPC Templates

Komorebie

September 14, 2024

Contents

1 杂项

1.1 __int128 输出流自定义

```
std::ostream &operator<<(std::ostream &os, __int128 n) {
    std::string s;
    while (n) {
        s += '0' + n % 10;
        n /= 10;
    }
    std::reverse(s.begin(), s.end());
    return os << s;
}
```

1.2 unordered_map 使用 pair 作为 key

对于任意结构体使用哈希，要先重载等于号（冲突时），然后在哈希函数中将所有的哈希值异或起来返回即可。

```
struct pair_hash {
    template <class T1, class T2>
    size_t operator () (const pair<T1,T2> &p) const {
        auto h1 = hash<T1>{}(p.first);
        auto h2 = hash<T2>{}(p.second);
        return h1 ^ h2; // 哈希组合
    }
};

unordered_map<pair<int, int>, int, pair_hash>mp; // 使用方法
```

1.3 cout 设置精度

```
// 该函数在头文件<iomanip>中
// 保留小数点后12位
cout << fixed << setprecision(12);
// 控制输出流显示浮点数的有效数字个数，会四舍五入
cout << setprecision(12);
```

1.4 库函数

1.4.1 位运算函数

```
// 返回x二进制下含1的数量
cout << __builtin_popcount(x); // 例如x=15时答案为4

// 返回x二进制下最后一个1的位置（从1开始计算）
cout << __builtin_ffs(x); // 例如x=1答案为1，x=8答案为4

// 返回x二进制下后导0的个数
cout << __builtin_ctz(x); // 例如x=1答案为0，x=8答案为3
```

```

// 返回x二进制下第一个1的位置（原函数是求前导0的个数）
cout << 31 - __builtin_clz(x); // x = 9(1001) 返回 3
cout << 63 - __builtin_clzll(x);
```

1.4.2 批量递增赋值函数

```
//将a数组[start,end)区间复制成“x, x+1, x+2, …”
iota(a + start, a + end, x);
```

1.4.3 数组随机打乱

```
mt19937 rnd(chrono::steady_clock::now().time_since_epoch().count());
shuffle(ver.begin(), ver.end(), rnd); gt
```

1.5 字符串转化

1.5.1 数字转字符串

to_string 函数会直接将你的各种类型的数字转换为字符串。【不建议使用】itoa 允许你将整数转换成任意进制的字符串，参数为待转换整数、目标字符数组、进制。但是其不是标准的 C 函数，且为 Windows 独有，且不支持 long long ，建议手写。

```
// string to_string(T val);
double val = 12.12;
cout << to_string(val);
// char* itoa(int value, char* string, int radix);
char ans[10] = {};
itoa(12, ans, 2);
cout << ans << endl; /*1100*/

// 长整型函数名ltoa，最高支持到int型上限2^31。ultoa同理。
```

1.5.2 字符串转数字

```
// stoi直接使用
cout << stoi("12") << endl;

// 【不建议使用】stoi转换进制，参数为待转换字符串、起始位置、进制。
// int stoi(string value, int st, int radix);
cout << stoi("1010", 0, 2) << endl; /*10*/
cout << stoi("c", 0, 16) << endl; /*12*/
cout << stoi("0x3f3f3f3f", 0, 0) << endl; /*1061109567*/

// 长整型函数名stoll，最高支持到long long型上限2^63。stoull、stod、stold同理。
```

2 数据结构

2.1 并查集

并查集究极版（支持维护 size 和到祖先的距离 dis）

```
struct DSU {
    vector<int> p, sz, d;
    DSU(int n)
    {
        p.resize(n + 1);
        sz.resize(n + 1, 1);
        d.resize(n + 1, 0);
        iota(p.begin(), p.end(), 0);
    }

    int find(int x)
    {
        if (p[x] != x) {
            int u = find(p[x]);
            d[x] += d[p[x]];
            p[x] = u;
        }
        return p[x];
    }

    bool same(int a, int b) { return find(a) == find(b); }

    void merge(int a, int b)
    {
        p[find(a)] = find(b);
        sz[b] += sz[a];
    }

    void merge(int a, int b, int dis)
    {
        p[find(a)] = find(b);
        sz[b] += sz[a];
        d[find(a)] = dis; // 根据具体问题，初始化find(a)的偏移量
    }

    int get_sz(int a) { return sz[find(a)]; }
};
```

2.2 树状数组

```
int tr[N];

int lowbit(int x) { return x & -x; }

void add(int x, int c) // 第x位加上c
{
```

```cpp
    for (int i = x; i <= n; i += lowbit(i)) {
        tr[i] += c;
    }
}

int sum(int x) // 求前x位的和
{
    int res = 0;
    for (int i = x; i; i -= lowbit(i)) {
        res += tr[i];
    }
    return res;
}
```

2.3 ST 表

```cpp
template <typename T> struct ST {
    const int B = 30;
    int n;
    vector<vector<T>> a;
    ST(int n) : n(n) { a.resize(n + 1, vector<T>(B)); }
    ST(vector<T> nums)
    {
        this->n = nums.size() - 1;
        a.resize(n + 1, vector<T>(B));
        for (int i = 1; i <= n; i++) a[i][0] = nums[i];
        build();
    }
    ST(T nums[], int n)
    {
        this->n = n;
        a.resize(n + 1, vector<T>(B));
        for (int i = 1; i <= n; i++) a[i][0] = nums[i];
        build();
    }
    T calc(T x, T y) { return max(x, y); }
    void build()
    {
        for (int j = 1; (1 << j) <= n; j++)
            for (int i = 1; i + (1 << j) - 1 <= n; i++) a[i][j] = calc(a[i][j - 1], a[
                i + (1 << (j - 1))][j - 1]);
    }
    T query(int l, int r)
    {
        if (r < l) return 0;
        int k = log2(r - l + 1);
        return calc(a[l][k], a[r - (1 << k) + 1][k]);
    }
};
```

2.4 线段树

支持区间修改和查询区间和（懒标记）。

```
struct Seg_tree {
    struct node {
        int l, r;
        ll sum, add;
    };
    vector<node> tr;
    vector<int> a;
    Seg_tree(int n)
    {
        tr.resize((n << 2) + 1);
        a.resize(n + 1);
        build(1, 1, n);
    }
    Seg_tree(vector<int> nums)
    {
        int n = nums.size() - 1;
        a.assign(nums.begin(), nums.end());
        tr.resize((n << 2) + 1);
        build(1, 1, n);
    }

    void pushup(int u) { tr[u].sum = tr[u << 1].sum + tr[u << 1 | 1].sum; }

    void pushdown(int u)
    {
        auto &root = tr[u], &left = tr[u << 1], &right = tr[u << 1 | 1];
        if (root.add) {
            left.add += root.add, right.add += root.add;
            left.sum += (left.r - left.l + 1) * root.add, right.sum += (right.r -
                right.l + 1) * root.add;
            root.add = 0;
        }
    }

    void build(int u, int l, int r)
    {
        if (l == r)
            tr[u] = {l, r, a[l], 0};
        else {
            tr[u] = {l, r};
            int mid = l + r >> 1;
            build(u << 1, l, mid);
            build(u << 1 | 1, mid + 1, r);
            pushup(u);
        }
    }

    void modify(int u, int l, int r, ll v)
    {
```

```
        if (tr[u].l >= l && tr[u].r <= r) {
            tr[u].sum += 1ll * (tr[u].r - tr[u].l + 1) * v;
            tr[u].add += v;
        }
        else {
            pushdown(u);
            int mid = tr[u].l + tr[u].r >> 1;
            if (l <= mid) modify(u << 1, l, r, v);
            if (r > mid) modify(u << 1 | 1, l, r, v);
            pushup(u);
        }
    }

    ll query(int u, int l, int r)
    {
        if (tr[u].l >= l && tr[u].r <= r)
            return tr[u].sum;
        else {
            pushdown(u);
            int mid = tr[u].l + tr[u].r >> 1;
            ll sum = 0;
            if (l <= mid) sum += query(u << 1, l, r);
            if (r > mid) sum += query(u << 1 | 1, l, r);
            return sum;
        }
    }
};
```

2.5 主席树

注意先离散化再将离散化后的值逐个插入 (modify)。

```
struct PresidentTree {
    static constexpr int N = 2e5 + 10;
    int ls[N * 25], rs[N * 25], idx = 0, cnt[N * 25], root[N * 25];

    void modify(int& cur, int past, int x, int l, int r)
    {
        cur = ++idx;
        ls[cur] = ls[past];
        rs[cur] = rs[past];
        cnt[cur] = cnt[past] + 1;
        if (l == r) return;
        int mid = l + r >> 1;
        if (x <= mid)
            modify(ls[cur], ls[past], x, l, mid);
        else
            modify(rs[cur], rs[past], x, mid + 1, r);
    }

    int query(int lx, int rx, int l, int r, double v) // [l, r] 中第一个出现次数严格大
        于 v 的数，不存在则输出-1
```

```
    {
        if (l == r) return l - 1;
        int mid = l + r >> 1;
        int res = -1;
        if ((double)(cnt[ls[rx]]) - (double)(cnt[ls[lx]]) > v) res = query(ls[lx], ls
            [rx], l, mid, v);
        if (res == -1 && (double)(cnt[rs[rx]]) - (double)(cnt[rs[lx]]) > v) res =
            query(rs[lx], rs[rx], mid + 1, r, v);
        return res;
    }

    int kth(int lx, int rx, int l, int r, int k) // [l, r] 中第k小的数
    {
        if (l == r) return l - 1;
        int res = cnt[ls[rx]] - cnt[ls[lx]];
        int mid = l + r >> 1;
        if (k <= res)
            return kth(ls[lx], ls[rx], l, mid, k);
        else
            return kth(rs[lx], rs[rx], mid + 1, r, k - res);
    }
} T;
```

2.6 树上启发式合并

```
void dfs(int u, int f) // 与重链剖分相同的写法找重儿子
{
    siz[u] = 1;
    for (int i = Head[u]; ~i; i = Edge[i].next) {
        int v = Edge[i].to;
        if (v == f) continue;
        dfs(v, u);
        siz[u] += siz[v];
        if (siz[v] > siz[son[u]]) son[u] = v;
    }
}
int col[maxn], cnt[maxn]; // col存放某结点的颜色，cnt存放某颜色在“当前”子树中的数量
long long ans[maxn], sum; // ans是答案数组，sum用于累加计算出“当前”子树的答案
int flag, maxc; // flag用于标记重儿子，maxc用于更新最大值
// TODO 统计某结点及其所有轻儿子的贡献
void count(int u, int f, int val)
{
    cnt[col[u]] += val; // val为正为负可以控制是增加贡献还是删除贡献
    if (cnt[col[u]] > maxc) // 找最大值，基操吧
    {
        maxc = cnt[col[u]];
        sum = col[u];
    }
    else if (cnt[col[u]] == maxc) // 这样做是因为如果两个颜色数量相同那都得算
        sum += col[u];
    for (int i = Head[u]; ~i; i = Edge[i].next) // 排除被标记的重儿子，统计其它儿子子树
        信息
```

```
    {
        int v = Edge[i].to;
        if (v == f || v == flag) continue; // 不能写if(v==f||v==son[u]) continue;
        count(v, u, val);
    }
}
// dsu on tree的板子
void dfs(int u, int f, bool keep)
{
    // 第一步: 搞轻儿子及其子树算其答案删贡献
    for (int i = Head[u]; ~i; i = Edge[i].next) {
        int v = Edge[i].to;
        if (v == f || v == son[u]) continue;
        dfs(v, u, false);
    }
    // 第二步: 搞重儿子及其子树算其答案不删贡献
    if (son[u]) {
        dfs(son[u], u, true);
        flag = son[u];
    }
    // 第三步: 暴力统计u及其所有轻儿子的贡献合并到刚算出的重儿子信息里
    count(u, f, 1);
    flag = 0;
    ans[u] = sum;
    // 把需要删除贡献的删一删
    if (!keep) {
        count(u, f, -1);
        sum = maxc = 0; // 这是因为count函数中会改变这两个变量值
    }
}
/********************* 另一种写法 *********************/
#include <bits/stdc++.h>
using namespace std;
#define endl '\n'
#define ll long long
#define PII pair<int, int>
#define pi acos(-1.0)
const int N = 1e5 + 10;
int n;
int a[N];
ll ans = 0;
vector<int> G[N];
vector<int> path;
int son[N], sz[N];
int cnt[22][2][(int)1e6 + 10];
int lca;

void dfs(int u, int fa)
{
    sz[u] = 1;
    for (int v : G[u]) {
        if (v == fa) continue;
        dfs(v, u);
```

```
        sz[u] += sz[v];
        if (sz[v] > sz[son[u]]) son[u] = v;
    }
}

void calc(int u, int fa)
{
    path.push_back(u);
    if ((a[u] ^ lca) < (int)1e6 + 10)
        for (int j = 0; j <= 20; j++) ans += 1ll * cnt[j][((u >> j) & 1) ^ 1][a[u] ^
            lca] * (1 << j);
    for (int v : G[u]) {
        if (v == fa) continue;
        calc(v, u);
    }
}

void insert(int u, int fa)
{
    for (int j = 0; j <= 20; j++) cnt[j][(u >> j) & 1][a[u]]++;
    for (int v : G[u]) {
        if (v == fa) continue;
        insert(v, u);
    }
}

void del(int u, int fa)
{
    for (int j = 0; j <= 20; j++) cnt[j][(u >> j) & 1][a[u]]--;
    for (int v : G[u]) {
        if (v == fa) continue;
        del(v, u);
    }
}

void dfs(int u, int fa, bool keep)
{
    for (int v : G[u]) {
        if (v == fa || v == son[u]) continue;
        dfs(v, u, false);
    }
    if (son[u]) {
        dfs(son[u], u, true);
    }
    lca = a[u];
    for (int v : G[u]) {
        if (v == fa || v == son[u]) continue;
        calc(v, u);
        for (auto vv : path)
            for (int j = 0; j <= 20; j++) cnt[j][(vv >> j) & 1][a[vv]]++;
        path.clear();
    }
    for (int j = 0; j <= 20; j++) cnt[j][(u >> j) & 1][a[u]]++;
```

```cpp
    if (!keep) {
        del(u, fa);
    }
}

int main()
{
    // ios::sync_with_stdio(false), cin.tie(nullptr);
    scanf("%d", &n);
    for (int i = 1; i <= n; i++) scanf("%d", a + i);
    for (int i = 1; i <= n - 1; i++) {
        int u, v;
        scanf("%d%d", &u, &v);
        G[u].push_back(v), G[v].push_back(u);
    }
    dfs(1, 0);
    dfs(1, 0, true);
    printf("%lld\n", ans);
    return 0;
}
```

2.7 莫队算法

一定要先拓展区间再收缩区间，防止区间减到 0 及以下。

```cpp
struct query {
    int l, r, pos, id;
    bool operator<(query a)
    {
        if (a.pos == this->pos) return a.pos & 1 ? this->r < a.r : this->r > a.r; // 右端点奇偶波浪排序
        return this->pos < a.pos;
    }
} q[N];

int len = sqrt(n);
for (int i = 1; i <= m; i++)
{
    cin >> q[i].l >> q[i].r;
    q[i].id = i;
    q[i].pos = q[i].l / len;
}
sort(q + 1, q + 1 + m, cmp1);
for (int i = 1, l = 1, r = 0; i <= m; i++)
{
    while (l > q[i].l) add(--l);
    while (r < q[i].r) add(++r);
    while (l < q[i].l) sub(l++);
    while (r > q[i].r) sub(r--);
    ans[q[i].id] = res;
}
```

3 字符串

3.1 KMP

```
// 求Next数组：
// s[]是模式串，p[]是模板串, n是s的长度，m是p的长度
for (int i = 2, j = 0; i <= m; i ++ )
{
    while (j && p[i] != p[j + 1]) j = ne[j];
    if (p[i] == p[j + 1]) j ++ ;
    ne[i] = j;
}
// 匹配
for (int i = 1, j = 0; i <= n; i ++ )
{
    while (j && s[i] != p[j + 1]) j = ne[j];
    if (s[i] == p[j + 1]) j ++ ;
    if (j == m)
    {
        j = ne[j];
        // 匹配成功后的逻辑
    }
}
```

3.2 Manacher

```
std::vector<int> manacher(std::string s) // 最后减一才是半径
{
    std::string t = "#";
    for (auto c : s) {
        t += c, t += '#';
    }
    int n = t.size();
    std::vector<int> r(n);
    for (int i = 0, j = 0; i < n; i++) {
        if (2 * j - i >= 0 && j + r[j] > i) {
            r[i] = std::min(r[2 * j - i], j + r[j] - i);
        }
        while (i - r[i] >= 0 && i + r[i] < n && t[i - r[i]] == t[i + r[i]]) {
            r[i] += 1;
        }
        if (i + r[i] > j + r[j]) {
            j = i;
        }
    }
    return r;
}
```

3.3 trie 树

```
int son[N][26], cnt[N], idx;
```

```
// 0号点既是根节点，又是空节点
// son[][]存储树中每个节点的子节点
// cnt[]存储以每个节点结尾的单词数量

// 插入一个字符串
void insert(char* str)
{
    int p = 0;
    for (int i = 0; str[i]; i++) {
        int u = str[i] - 'a';
        if (!son[p][u]) son[p][u] = ++idx;
        p = son[p][u];
    }
    cnt[p]++;
}

// 查询字符串出现的次数
int query(char* str)
{
    int p = 0;
    for (int i = 0; str[i]; i++) {
        int u = str[i] - 'a';
        if (!son[p][u]) return 0;
        p = son[p][u];
    }
    return cnt[p];
}
```

3.4 Z 函数

$z[i] = lcp(s[1:n-1], s[i:n-1])$

```
vector<int> get_z(string s)
{
    int n = s.size();
    vector<int> z(n);
    for (int i = 1, l = 0; i < n; i++)
    {
        if (i <= l + z[l] - 1) z[i] = min(z[i - l], l + z[l] - i);
        while (i + z[i] < n && s[i + z[i]] == s[z[i]]) z[i]++;
        if (i + z[i] > l + z[l]) l = i;
    }
    return z;
}//最后需要修改z[0] = n;
```

3.5 AC 自动机

```
struct ACAM {
    int n;
    int trie_size = 10;
    vector<string> T;
```

```cpp
    vector<vector<int>> trie;
    vector<int> pos, fail, cnt, id, q, end;

    void init()
    {
        trie.resize(trie_size, vector<int>(26, 0));
        pos.resize(n, 0);
        fail.resize(trie_size, 0);
        cnt.resize(trie_size, 0);
        id.resize(trie_size, -1);
        end.resize(trie_size, 0);
    }
    void build()
    {
        int idx = 1; // trie树
        for (int i = 0; i < n; i++) {
            int p = 1;
            for (auto c : T[i]) {
                int cur = c - 'a';
                if (!trie[p][cur]) trie[p][cur] = ++idx;
                p = trie[p][cur];
            }
            pos[i] = p;
            id[p] = i;
            end[p]++; // 统计以该节点结尾的模式串个数
        }
        auto& q = this->q; // 处理根节点的回跳
        int ql = 0;
        for (auto& c : trie[1]) {
            if (c) {
                fail[c] = 1;
                q.push_back(c);
            }
            else
                c = 1;
        }
        while (ql < q.size()) // BFS
        {
            int u = q[ql++];
            for (int c = 0; c < 26; c++) {
                if (trie[u][c]) // 有儿子存在时
                {
                    fail[trie[u][c]] = trie[fail[u]][c]; // 回跳边 (四边形)
                    q.push_back(trie[u][c]);
                }
                else {
                    trie[u][c] = trie[fail[u]][c]; // 转移边 (三角形)
                }
            }
            end[u] += end[fail[u]];
        }
    }

```

```cpp
    void count(string S) // 统计每个结点在文本串中出现次数
    {
        auto& q = this->q;
        for (int cur = 1, i = 0; i < S.size(); i++) {
            int nxt = trie[cur][S[i] - 'a'];
            cnt[cur = nxt]++;
        }

        reverse(q.begin(), q.end());
        for (auto cur : q) {
            cnt[fail[cur]] += cnt[cur];
        }
    }
};
```

3.6 回文自动机

应用

- 本质不同回文子串个数
- 回文子串出现次数

每个节点代表一个回文串，每个节点的 len 表示回文串的长度，num 表示回文串出现次数。0 号节点表示长度为 0 的回文串，1 号节点表示长度为-1 的回文串。

```cpp
struct PAM {
    int len[N], num[N], fail[N], trie[N][26], tot = 1;
    int getfail(int x, int i, string s)
    {
        while (i - len[x] - 1 < 0 || s[i] != s[i - len[x] - 1]) x = fail[x];
        return x;
    }
    void build(string s)
    {
        int cur = 0;
        fail[0] = 1, len[1] = -1;
        for (int i = 0; i < s.size(); i++) {
            int u = s[i] - 'A';
            int pos = getfail(cur, i, s);
            if (!trie[pos][u]) {
                trie[pos][s[i]] = ++tot;
                fail[tot] = trie[getfail(fail[pos], i, s)][u];
                len[tot] = len[pos] + 2;
            }
            cur = trie[pos][u];
            num[cur]++;
        }
        for (int i = tot; i >= 2; i--) num[fail[i]] += num[i];
    }
}

```

```cpp
// 用法
void dfs(int u1, int u2)
{
    if (u1 > 1 && u2 > 1) ans += 1ll * A.num[u1] * B.num[u2];
    for (int i = 0; i < 26; i++) {
        if (A.trie[u1][i] && B.trie[u2][i]) dfs(A.trie[u1][i], B.trie[u2][i]);
    }
}
int main()
{
    dfs(0, 0);
    dfs(1, 1);
}
```

3.7 后缀排序

sa 表示排第 i 的后缀的前一半是第几个后缀, $sa2$ 表示排第 i 的后缀的后一半是第几个后缀，rk 表示第 i 个后缀排第几, 有 $sa[rk[i]] = i, rk[sa[i]] = i$.

$height[i] : lcp(sa[i], sa[i-1])$，即排名为 i 的后缀与排名为 $i-1$ 的后缀的最长公共前缀。

$height[rak[i]]$，即 i 号后缀与它前一名的后缀的最长公共前缀。

经典应用：

- 两个后缀的最大公共前缀：$lcp(x, y) = min(heigh[x-y])$，用 RMQ 维护，$O(1)$ 查询。
- 可重叠最长重复子串：$height$ 数组中最大值
- 本质不同的字串的数量：枚举每一个后缀，第 i 个后缀对答案的贡献为 $len - sa[i] + 1 - height[i]$

```cpp
int sa[N], rk_base[2][N], *rk = rk_base[0], *rk2 = rk_base[1], sa2[N], cnt[N],
     height[N];

void get_sa(const char* s, int n)
{
    int m = 122;
    for (int i = 0; i <= m; ++i) cnt[i] = 0;
    for (int i = 1; i <= n; ++i) cnt[rk[i] = s[i]] += 1;
    for (int i = 1; i <= m; ++i) cnt[i] += cnt[i - 1];
    for (int i = 1; i <= n; ++i) sa[cnt[rk[i]]--] = i;
    for (int d = 1; d <= n; d <<= 1)
    {
        // 按第二关键字排序
        int p = 0;
        for (int i = n - d + 1; i <= n; i++) sa2[++p] = i;
        for (int i = 1; i <= n; ++i)
            if (sa[i] > d) sa2[++p] = sa[i] - d;
        // 按第一关键字排序
        for (int i = 0; i <= m; ++i) cnt[i] = 0;
        for (int i = 1; i <= n; ++i) cnt[rk[i]] += 1;
        for (int i = 1; i <= m; ++i) cnt[i] += cnt[i - 1];
```

```
        for (int i = n; i; --i) sa[cnt[rk[sa2[i]]]--] = sa2[i];

        rk2[sa[1]] = 1;
        for (int i = 2; i <= n; ++i) rk2[sa[i]] = rk2[sa[i - 1]] + (rk[sa[i]] != rk[
            sa[i - 1]] || rk[sa[i] + d] != rk[sa[i - 1] + d]);

        std::swap(rk, rk2);

        m = rk[sa[n]];
        if (m == n) break;
    }
}

void get_height()
{
    for (int i = 1; i <= n; i++) rk[sa[i]] = i;
    for (int i = 1, k = 0; i <= n; i++)
    {
        if (rk[i] == 1) continue;
        if (k) k--;
        int j = sa[rk[i] - 1];
        while (s[i + k] == s[j + k]) k++;
        height[rk[i]] = k;
    }
}
```

4 图论

4.1 最短路

4.1.1 Dijkstra 朴素版

时间复杂度: $O(n^2 + m)$

```
int g[N][N]; // 存储每条边
int dist[N]; // 存储1号点到每个点的最短距离
bool st[N]; // 存储每个点的最短路是否已经确定

// 求1号点到n号点的最短路，如果不存在则返回-1
int dijkstra()
{
    memset(dist, 0x3f, sizeof dist);
    dist[1] = 0;

    for (int i = 0; i < n - 1; i ++ )
    {
        int t = -1; // 在还未确定最短路的点中，寻找距离最小的点
        for (int j = 1; j <= n; j ++ )
            if (!st[j] && (t == -1 || dist[t] > dist[j]))
                t = j;

        // 用t更新其他点的距离
```

```
        for (int j = 1; j <= n; j ++ )
            dist[j] = min(dist[j], dist[t] + g[t][j]);

        st[t] = true;
    }

    if (dist[n] == 0x3f3f3f3f) return -1;
    return dist[n];
}
```

4.1.2 Dijkstra 堆优化版

时间复杂度: $O(m \log n)$

```
int dist[N]; // 存储所有点到1号点的距离
bool st[N]; // 存储每个点的最短距离是否已确定
// 求1号点到n号点的最短距离，如果不存在，则返回-1
int dijkstra()
{
    memset(dist, 0x3f, sizeof dist);
    dist[1] = 0;
    priority_queue<PII, vector<PII>, greater<PII>> heap;
    heap.push({0, 1}); // first存储距离，second存储节点编号
    while (heap.size())
    {
        auto [d, u] = heap.top();
        heap.pop();
        if (st[u]) continue;
        st[u] = true;

        for (auto [v, w] : G[u])
        {
            if (dist[v] > distance + w)
            {
                dist[v] = distance + w;
                heap.push({dist[v], v});
            }
        }
    }

    if (dist[n] == 0x3f3f3f3f) return -1;
    return dist[n];
}
```

4.1.3 SPFA

最坏时间复杂度 $O(nm)$

```
int vis[N], cnt[N];
vector<PII> G[N];

bool spfa(int n, int s)
```

```
{
    memset(dis, 0x3f, sizeof(dis));
    dis[s] = 0;
    queue<int> q;
    while (q.size()) {
        int u = q.front();
        q.pop();
        vis[u] = 0;
        for (auto [v, w] : G[u]) {
            if (dis[v] >= dis[u] + w) {
                dis[v] = dis[u] + w;
                cnt[v] = cnt[u] + 1; // 记录经过了多少点
                if (cnt[v] >= n) // 存在负环
                    return false;
                if (!vis[v]) {
                    q.push(v);
                    vis[v] = 1;
                }
            }
        }
    }
    return true;
}
```

4.1.4 Bellman-Ford

```
struct EDGE // for bellman-ford
{
    int u, int v, int w;
};

vector<EDGE> edges;

bool Bellman_ford(int n, int s)
{
    memset(dis, 0x3f, sizeof(dis));
    dis[s] = 0;
    bool flag;
    for (int i = 1; i <= n; i++) {
        flag = false;
        for (auto [u, v, w] : edges) {
            if (dis[u] == 0x3f3f3f3f) continue;
            if (dis[v] > dis[u] + w) {
                dis[v] = dis[u] + w;
                flag = true;
            }
        }
        if (!flag) break;
    }
    return flag; // 第n轮循环仍可以松弛说明存在负环
}
```

4.2 网络流

4.2.1 EK

```
memset(h, -1, sizeof(h)); // 将h初始化为-1，因为边的标号从0开始。

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

bool bfs()
{
    queue<int> q;
    memset(st, false, sizeof(st));
    q.push(S);
    st[S] = true;
    d[S] = inf;
    while (q.size()) {
        int u = q.front();
        q.pop();
        for (int i = h[u]; i != -1; i = ne[i]) {
            int v = e[i];
            if (!st[v] && f[i]) {
                st[v] = true;
                d[v] = min(d[u], f[i]);
                pre[v] = i;
                if (v == T) return true;
                q.push(v);
            }
        }
    }
    return false;
}

int EK()
{
    int r = 0;
    while (bfs()) {
        r += d[T];
        for (int i = T; i != S; i = e[pre[i ^ 1]]) {
            f[pre[i]] -= d[T];
            f[pre[i ^ 1]] += d[T];
        }
    }
    return r;
}

struct EK {
    const int n, S, T;
    vector<int> h, e, ne, cur, f, d, pre, st;

```

```cpp
    EK(int n, int S, int T) : n(n), S(S), T(T) // 点数，源点，汇点
    {
        h.resize(n + 1, -1);
        d.resize(n + 1);
        cur.resize(n + 1);
        pre.resize(n + 1);
        st.resize(n + 1);
    }

    void addedge(int a, int b, int c)
    {
        e.push_back(b), ne.push_back(h[a]), f.push_back(c), h[a] = e.size() - 1;
        e.push_back(a), ne.push_back(h[b]), f.push_back(0), h[b] = e.size() - 1;
    }

    bool bfs()
    {
        queue<int> q;
        st.assign(n + 1, false);
        q.push(S);
        st[S] = true;
        d[S] = 0x3f3f3f3f;
        while (q.size()) {
            int u = q.front();
            q.pop();
            for (int i = h[u]; i != -1; i = ne[i]) {
                int v = e[i];
                if (!st[v] && f[i]) {
                    st[v] = true;
                    d[v] = min(d[u], f[i]);
                    pre[v] = i;
                    if (v == T) return true;
                    q.push(v);
                }
            }
        }
        return false;
    }

    int maxflow()
    {
        int r = 0;
        while (bfs()) {
            r += d[T];
            for (int i = T; i != S; i = e[pre[i ^ 1]]) {
                f[pre[i]] -= d[T];
                f[pre[i ^ 1]] += d[T];
            }
        }
        return r;
    }
};
```

4.2.2 Dinic

```cpp
struct Dinic {
    const int n, S, T;
    vector<int> h, e, ne, cur, f, d;

    Dinic(int n, int S, int T) : n(n), S(S), T(T) // 点数，源点，汇点
    {
        h.resize(n + 1, -1);
        d.resize(n + 1);
        cur.resize(n + 1);
    }

    void addedge(int a, int b, int c)
    {
        e.push_back(b), ne.push_back(h[a]), f.push_back(c), h[a] = e.size() - 1;
        e.push_back(a), ne.push_back(h[b]), f.push_back(0), h[b] = e.size() - 1;
    }

    bool bfs()
    {
        d.assign(n + 1, -1);
        queue<int> q;
        q.push(S);
        d[S] = 0;
        cur[S] = h[S];
        while (q.size()) {
            int u = q.front();
            q.pop();
            for (int i = h[u]; ~i; i = ne[i]) {
                int v = e[i];
                if (d[v] == -1 && f[i] > 0) {
                    d[v] = d[u] + 1;
                    cur[v] = h[v];
                    if (v == T) return true;
                    q.push(v);
                }
            }
        }
        return false;
    }

    int find(int u, int limit)
    {
        if (u == T) return limit;
        int flow = 0;
        for (int i = cur[u]; ~i; i = ne[i]) {
            cur[u] = i;
            int v = e[i];
            if (d[v] == d[u] + 1 && f[i]) {
                int t = find(v, min(limit - flow, f[i]));
                if (!t) d[v] = -1; // important!!!!!!!!!!!!!!!!!!!!!!!!!
                f[i] -= t, f[i ^ 1] += t;
```

```
                flow += t;
            }
        }
        return flow;
    }

    int dinic()
    {
        int r = 0, flow;
        while (bfs()) {
            while (flow = find(S, 0x3f3f3f3f)) {
                r += flow;
            }
        }
        return r;
    }
};
```

4.2.3 MCMF

```
struct MCMF {
    vector<long long> ne, h, e, f, w, dis, incf, vis, pre;
    int idx, n, S, T;
    MCMF(int n, int m, int s, int t) // 点数，边数
    {
        this->S = s, this->T = t, this->n = n, this->idx = 0;
        ne.resize(m, 0);
        e.resize(m, 0);
        f.resize(m, 0);
        w.resize(m, 0);
        pre.resize(m, 0);
        vis.resize(n + 1, 0);
        h.resize(n + 1, -1);
    };

    void addedge(int a, int b, int c, int d)
    {
        int& idx = this->idx;
        ne[idx] = h[a], e[idx] = b, f[idx] = c, w[idx] = d, h[a] = idx++;
        ne[idx] = h[b], e[idx] = a, f[idx] = 0, w[idx] = -d, h[b] = idx++;
    }

    bool spfa()
    {
        queue<int> q;
        dis.assign(n + 1, 0x3f3f3f3f);
        incf.assign(n + 1, 0);
        q.push(S);
        dis[S] = 0, incf[S] = 0x3f3f3f3f;
        while (q.size()) {
            int u = q.front();
            q.pop(), vis[u] = 0;
```

```
            for (int i = h[u]; ~i; i = ne[i]) {
                int v = e[i];
                if (f[i] && dis[v] > dis[u] + w[i]) {
                    dis[v] = dis[u] + w[i];
                    pre[v] = i;
                    incf[v] = min(f[i], incf[u]);
                    if (vis[v] == 0) {
                        q.push(v), vis[v] = 1;

                    }
                }
            }
        }
        return incf[T] > 0;
    }

    pair<long long, long long> EK()
    {
        long long flow = 0, cost = 0;
        while (spfa()) {
            int t = incf[T];
            flow += t, cost += t * dis[T];
            for (int i = T; i != S; i = e[pre[i] ^ 1]) {
                f[pre[i]] -= t;
                f[pre[i] ^ 1] += t;
            }
        }
        return make_pair(flow, cost);
    }
};
```

4.3 tarjan

4.3.1 tarjan 求割点

low：最多经过一条后向边能追溯到的最小树中结点编号。

一个顶点 u 是割点，当且仅当满足 (1) 或 (2)：

1. u 为树根，且 u 有多于一个子树。因为无向图 DFS 搜索树中不存在横叉边，所以若有多个子树，这些子树间不会有边相连。

2. u 不为树根，且满足存在 (u, v) 为树枝边（即 u 为 v 在搜索树中的父亲），并使得 $DFN(u) <= Low(v)$.（因为删去 u 后 v 以及 v 的子树不能到达 u 的其他子树以及祖先）

```
int dfn[N], low[N], tim, vis[N], flag[N];
int ans;
vector<int> G[N];

void dfs(int u, int fa)
{
    dfn[u] = low[u] = ++tim;
    vis[u] = 1;
    int children = 0;
```

```cpp
    for (int v : G[u]) {
        if (!vis[v]) {
            children++;
            dfs(v, u);
            low[u] = min(low[u], low[v]);
            if (u != fa && low[v] >= dfn[u] && !flag[u]) {
                flag[u] = 1;
                ans++;
            }
        }
        else if (v != fa) {
            low[u] = min(low[u], dfn[v]);
        }
    }
    if (u == fa && children >= 2 && !flag[u]) {
        flag[u] = 1;
        ans++;
    }
}
```

4.3.2 tarjan 求强连通分量

low：最多经过一条后向边或栈中横插边所能到达的栈中的最小编号。

```cpp
stack<int> st;
int in_stack[N];

void tarjan(int u)
{
    low[u] = dfn[u] = ++idx;
    st.push(u);
    in_stack[u] = 1;
    for (int v : G[u]) {
        if (!dfn[v]) {
            tarjan(v);
            low[u] = min(low[u], low[v]);
        }
        else if (in_stack[v]) {
            low[u] = min(low[u], dfn[v]);
        }
    }
    if (low[u] == dfn[u]) {
        ++sc;
        while (st.top() != u) {
            scc[st.top()] = sc;
            in_stack[st.top()] = 0;
            st.pop();
            sz[sc]++;
        }
        scc[st.top()] = sc;
        in_stack[st.top()] = 0;
        st.pop();
        sz[sc]++;
```

```
    }
}
```

4.3.3 tarjan 求点双连通分量（圆方树）

```
vector<int> G[N]; // 原图
vector<int> T[N]; // 新图（圆方树）
void tarjan(int u, int fa)
{
    int son = 0;
    dfn[u] = low[u] = ++tim;
    st.push(u);
    for (int v : G[u]) {
        if (!dfn[v]) {
            son++;
            tarjan(v, u);
            low[u] = min(low[u], low[v]);
            if (low[v] >= dfn[u]) {
                scc++;
                while (st.top() != v) {
                    ans[scc].push_back(st.top());
                    T[scc].push_back(st.top());
                    T[st.top()].push_back(scc);
                    st.pop();
                }
                ans[scc].push_back(st.top());
                T[scc].push_back(st.top());
                T[st.top()].push_back(scc);
                st.pop();
                ans[scc].push_back(u);
                T[scc].push_back(u);
                T[u].push_back(scc);
            }
        }
        else if (v != fa) // 返祖边
        {
            low[u] = min(low[u], dfn[v]);
        }
    }
    if (fa == 0 && son == 0) ans[++scc].push_back(u); // 特判孤立点
}
```

4.3.4 tarjan 求边双连通分量

```
void tarjan(int u, int fa)
{
    dfn[u] = low[u] = ++tim;
    int son = 0;
    for (int v : G[u]) {
        if (!dfn[v]) {
            son++;
```

```cpp
            tarjan2(v, u);
            low[u] = min(low[u], low[v]);
            if (low[v] > dfn[u]) // 找割边
            {
                cnt_bridge++;
                es[mp[hh(u, v)]].tag = 1;
            }
        }
        else if (v != fa) {
            low[u] = min(low[u], dfn[v]);
        }
    }
}
```

4.4 树上问题

4.4.1 树的直径

```cpp
void dfs(int u, int fa) // 树形dp法 一个数组实现 存的是到子树中的最长路径
{
    for (int v : G[u]) {
        if (v == fa) continue;
        dfs(v, u);
        diameter = max(diameter, d1[u] + d1[v] + 1);
        d1[u] = max(d1[u], d1[v] + 1);
    }
}
```

4.4.2 树的重心

```cpp
vector<int> centroid;
int sz[N], weight[N]; // weight记录子树大小最大值

void dfs(int u, int fa)
{
    sz[u] = 1, weight[u] = 0;
    for (auto v : G[u]) {
        if (v == fa) continue;
        dfs(v, u);
        sz[u] += sz[v];
        weight[u] = max(weight[u], sz[v]);
    }
    weight[u] = max(n - weight[u], weight[u]);
    if (weight[u] <= n / 2) // 所有子树大小都不超过n/2
    {
        centroid.push_back(u);
    }
}
```

4.4.3 最近公共祖先（倍增）

```cpp
struct LCA {
    vector<vector<int>> G;
    vector<vector<int>> fa;
    vector<int> dep;
    LCA(int n)
    {
        G.resize(n + 1);
        fa.resize(n + 1, vector<int>(31, 0));
        dep.resize(n + 1, 0);
    }

    void dfs(int u, int f)
    {
        fa[u][0] = f;
        dep[u] = dep[f] + 1;
        for (int i = 1; i < 31; i++) {
            fa[u][i] = fa[fa[u][i - 1]][i - 1];
        }
        for (int v : e[u]) {
            if (v != f) {
                dfs(v, u);
            }
        }
    }

    int lca(int x, int y)
    {
        if (dep[x] < dep[y]) {
            swap(x, y);
        }
        int d = dep[x] - dep[y];
        for (int i = 0; (1 << i) <= d; i++) {
            if ((d >> i) & 1) x = fa[x][i];
        }
        if (x == y) return x;
        for (int i = log2(dep[y]); i >= 0; i--) {
            if (fa[x][i] != fa[y][i]) {
                x = fa[x][i];
                y = fa[y][i];
            }
        }
        return fa[x][0];
    }

    int dist(int x, int y) { return dep[x] + dep[y] - 2 * dep[lca(x, y)]; }
};
```

4.4.4 树链剖分

```cpp
struct HLD {
    int n, idx;
    vector<vector<int>> G;
```

```cpp
    vector<int> sz, dep, top, son, parent;
    HLD(int n)
    {
        this->n = n;
        G.resize(n + 1);
        sz.resize(n + 1);
        dep.resize(n + 1);
        top.resize(n + 1);
        son.resize(n + 1);
        parent.resize(n + 1);
    }
    void dfs1(int u) // 处理出深度和重儿子
    {
        sz[u] = 1;
        dep[u] = dep[parent[u]] + 1;
        for (auto v : G[u]) {
            if (v == parent[u]) continue;
            parent[v] = u;
            dfs1(v);
            sz[u] += sz[v];
            if (sz[v] > sz[son[u]]) son[u] = v;
        }
    }
    void dfs2(int u, int up)
    {
        top[u] = up;
        if (son[u]) dfs2(son[u], up);
        for (int v : G[u]) {
            if (v == parent[u] || v == son[u]) continue;
            dfs2(v, v);
        }
    }
    int lca(int x, int y)
    {
        while (top[x] != top[y]) {
            if (dep[top[x]] > dep[top[y]]) {
                x = parent[top[x]];
            }
            else {
                y = parent[top[y]];
            }
        }
        return dep[x] < dep[y] ? x : y;
    }
    int calc(int x, int y) { return dep[x] + dep[y] - 2 * dep[lca(x, y)]; } // 查询两点距离
};
```

4.5 拓扑排序

```cpp
bool topsort()
{
```

```cpp
    int hh = 0, tt = -1;
    // d[i] 存储点i的入度
    for (int i = 1; i <= n; i++)
        if (!d[i]) q[++tt] = i;
    while (hh <= tt) {
        int u = q[hh++];

        for (int v : G[u]) {
            if (--d[v] == 0) q[++tt] = v;
        }
    }
    // 如果所有点都入队了，说明存在拓扑序列；否则不存在拓扑序列。
    return tt == n - 1;
}
```

4.6 染色法判断二分图

```cpp
int n; // n表示点数
int color[N]; // 表示每个点的颜色，-1表示为染色，0表示白色，1表示黑色

// 参数：u表示当前节点，c表示当前点的颜色
bool dfs(int u, int c)
{
    color[u] = c;
    for (int v : G[u]) {
        if (color[v] == -1) {
            if (!dfs(v, !c)) return false;
        }
        else if (color[v] == c)
            return false;
    }

    return true;
}

bool check()
{
    memset(color, -1, sizeof color);
    bool flag = true;
    for (int i = 1; i <= n; i++)
        if (color[i] == -1)
            if (!dfs(i, 0)) {
                flag = false;
                break;
            }
    return flag;
}
```

4.7 匈牙利算法求最大匹配

```cpp
int n1, n2; // n1表示第一个集合中的点数，n2表示第二个集合中的点数
```

```cpp
vector<int> G[N]; // 匈牙利算法中只会用到从第二个集合指向第一个集合的边，所以这里只用存一个方向的边
int match[N]; // 存储第二个集合中的每个点当前匹配的第一个集合中的点是哪个
bool st[N]; // 表示第二个集合中的每个点是否已经被遍历过

bool find(int u)
{
    for (int v : G[u]) {
        if (!st[v]) {
            st[v] = true;
            if (match[v] == 0 || find(match[v])) {
                match[v] = u;
                return true;
            }
        }
    }
    return false;
}

int res = 0;
for (int i = 1; i <= n1; i++) // 求最大匹配数，依次枚举第一个集合中的每个点能否匹配第二个集合中的点
{
    memset(st, false, sizeof st);
    if (find(i)) res++;
}
```

4.8 差分约束

求解差分约束系统，有 m 条约束条件，每条都为形如 $x_a - x_b \geq c_k$，$x_a - x_b \leq c_k$ 或 $x_a = x_b$ 的形式，判断该差分约束系统有没有解，如果有解，求出一组解。

题意	转化	连边
$x_a - x_b \geq c$	$x_b - x_a \leq -c$	add(a, b, -c);
$x_a - x_b \leq c$	$x_a - x_b \leq c$	add(b, a, c);
$x_a = x_b$	$x_a - x_b \leq 0$, $x_b - x_a \leq 0$	add(b, a, 0), add(a, b, 0);

若要求出一组解，则每个点到源点的最短路即为解。具体过程见参考代码

```cpp
#include <bits/stdc++.h>
using namespace std;
const int N = 1e6 + 10;
int n, m;
int h[N], e[N], ne[N], we[N], idx;
int dis[N], cnt[N], vis[N];

struct node {
    int v, w;
};

vector<node> G[N];

```

```cpp
void add(int u, int v, int w) { ne[++idx] = h[u], h[u] = idx, e[idx] = v, we[idx] =
    w; }

bool spfa(int num, int s)
{
    memset(dis, 0x3f, sizeof(dis));
    dis[s] = 0;
    vis[s] = 1;
    queue<int> q;
    q.push(s);
    while (q.size()) {
        int u = q.front();
        q.pop();
        vis[u] = 0;
        for (auto it : G[u]) {
            int v = it.v, w = it.w;
            if (dis[v] > dis[u] + w) {
                dis[v] = dis[u] + w;
                cnt[v] = cnt[u] + 1; // 记录经过了多少点
                if (cnt[v] > num) // 存在负环
                    return false;
                if (!vis[v]) {
                    q.push(v);
                    vis[v] = 1;
                }
            }
        }
    }
    return true;
}

int main()
{
    ios::sync_with_stdio(false);
    cin.tie(0);
    cin >> n >> m;
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            char c;
            cin >> c;
            if (c == '>') {
                G[i].push_back({n + j, -1});
            }
            else if (c == '<') {
                G[n + j].push_back({i, -1});
            }
            else {
                G[i].push_back({n + j, 0});
                G[n + j].push_back({i, 0});
            }
        }
    }
    for (int i = 1; i <= n + m; i++) {
```

```
        G[0].push_back({i, 0});
    }
    if (spfa(n + m + 1, 0)) {
        cout << "Yes" << endl;
        int minn = 0x3f3f3f3f, d = 0;
        for (int i = 1; i <= n + m; i++) {
            minn = min(minn, dis[i]);
        }
        if (minn <= 0) {
            d = -minn + 1;
        }
        for (int i = 1; i <= n; i++) {
            cout << dis[i] + d << " \n"[i == n];
        }
        cout << dis[n] + d << endl;
        for (int i = 1; i <= m; i++) {
            cout << dis[n + i] + d << " \n"[i == m];
        }
        cout << dis[n + m] + d;
    }
    else
        cout << "No";
    return 0;
}
```

5 数学

5.1 试除法分解质因数

```
void divide(int x)
{
    for (int i = 2; i <= x / i; i ++ )
        if (x % i == 0)
        {
            int s = 0;
            while (x % i == 0) x /= i, s ++ ;
            cout << i << ' ' << s << endl;
        }
    if (x > 1) cout << x << ' ' << 1 << endl;
    cout << endl;
}
```

5.2 欧拉筛

```
vector<ll> primes; // primes[]存储所有素数

void get_primes(int n)
{
    vector<bool> st(n + 1); // st[x]存储x是否被筛掉
    for (int i = 2; i <= n; i++) {
        if (!st[i]) primes.push_back(i);
```

```
        for (int j = 0; j < primes.size() && primes[j] <= n / i; j++) {
            st[primes[j] * i] = true;
            if (i % primes[j] == 0) break;
        }
    }
}
```

5.3 欧拉函数和欧拉定理

5.3.1 欧拉函数

欧拉函数定义：1 到 N 中与 N 互质数的个数称为欧拉函数，即 $\varphi(n) = \sum_{i=1}^{n}[\gcd(n,i)=1]$，记作 $\varphi(N)$ 。

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

求解单个数的欧拉函数直接质因数分解

欧拉定理：若 a 与 m 互质，则 $a^{\varphi(m)} \equiv 1 \pmod m$，变式 a^b $a^{b\%\varphi(m)}(mod\ m)$ 。

扩展欧拉定理：若 a 与 m 不互质，则 $a^b \equiv a^{b\%\varphi(m)+\varphi(m)} \pmod m$ 。

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

```
int phi(int n) // 求解 phi(n)
{
    int ans = n;
    for (int i = 2; i <= n / i; i++) { // 注意，这里要写 n / i ，以防止 int 型溢出风险和 sqrt 超时风险
        if (n % i == 0) {
            ans = ans / i * (i - 1);
            while (n % i == 0) n /= i;
        }
    }
    if (n > 1) ans = ans / n * (n - 1); // 特判 n 为质数的情况
    return ans;
}
```

5.3.2 线性筛欧拉函数

```
int phi[N], st[N];
vector<int>primes;
void sieve(int n)
{
    st[1] = phi[1] = 1;
    for (int i = 2; i <= n; i++) {
        if (!st[i]) primes.push_back(i), phi[i] = i - 1;
        for (int j = 1; j < primes.size() && i * primes[j] <= n; j++) {
```

```
            st[i * primes[j]] = 1;
            if (i % primes[j])
                phi[i * primes[j]] = phi[i] * phi[primes[j]];
            else {
                phi[i * primes[j]] = phi[i] * primes[j];
                break;
            }
        }
    }
}
```

5.4 莫比乌斯函数

设 $n=\prod_{i=1}^{m} p_i^{c_i}$，则

$$\mu(n)=\begin{cases}1 & n=1\\ (-1)^m & \prod_{i=1}^{m} c_i=1(\text{则}c_1=c_2=\cdots=c_m=1)\\ 0 & \text{otherwise}\end{cases}$$

```
int mu[N], st[N];
vector<int>primes;
void sieve(int n)
{
    st[1] = mu[1] = 1;
    for (int i = 2; i <= n; i++) {
        if (!st[i]) primes.push_back(i), mu[i] = -1;
        for (int j = 1; j < primes.size() && i * primes[j] <= n; j++) {
            st[i * primes[j]] = 1;
            if (i % primes[j])
                mu[i * primes[j]] = -mu[i];
            else {
                mu[i * primes[j]] = 0;
                break;
            }
        }
    }
}
```

5.5 线性筛约数个数

记 d_i 为 i 的约数个数，$d(i)=\prod_{k=1}^{i}(a_i+1)$ 维护每一个数的最小值因子出现的次数（即 a_1 ）即可。

```
int d[N], a[N], st[N];
vector<int>primes;
void sieve(int n)
{
    st[1] = d[1] = 1;
```

```cpp
    for (int i = 2; i <= n; i++) {
        if (!st[i]) primes.push_back(i), d[i] = 2, a[i] = 1;
        for (int j = 1; j < primes.size() && i * primes[j] <= n; j++) {
            st[i * primes[j]] = 1;
            if (i % primes[j])
                d[i * primes[j]] = d[i] * d[primes[j]], a[i * primes[j]] = 1;
            else {
                d[i * primes[j]] = d[i] / (a[i] + 1) * (a[i] + 2);
                a[i * primes[j]] = a[i] + 1;
                break;
            }
        }
    }
}
```

5.6 线性筛约数和

记 $\sigma(i)$ 表示 i 的约数和

$$\sigma(i) = \prod_{k=1}^{i} \left(\sum_{a_i=0}^{p_i} p_i^i \right)$$

维护 $\mathrm{low}(i)$ 表示 i 的最小质因子的指数次幂，即 $p_1^{a_1}$，$\mathrm{sum}(i)$ 表示 i 的最小质因子对答案的贡献，即 $\sum_{a_1=0}^{p_1} p_1^1$。可能会爆 int。

```cpp
int low[N], sum[N], sigma[N], st[N];
vector<int> primes;
void sieve(int n)
{
    st[1] = low[1] = sum[1] = sigma[1] = 1;
    for (int i = 2; i <= n; i++) {
        if (!st[i]) primes.push_back(i), low[i] = i, sum[i] = sigma[i] = i + 1;
        for (int j = 1; j < primes.size() && i * primes[j] <= n; j++) {
            st[i * primes[j]] = 1;
            if (i % primes[j] == 0) {
                low[i * primes[j]] = low[i] * primes[j];
                sum[i * primes[j]] = sum[i] + low[i * primes[j]];
                sigma[i * primes[j]] = sigma[i] / sum[i] * sum[i * primes[j]];
                break;
            }
            low[i * primes[j]] = primes[j];
            sum[i * primes[j]] = primes[j] + 1;
            sigma[i * primes[j]] = sigma[i] * sigma[primes[j]];
        }
    }
}
```

5.7 裴蜀定理

如果 a, b 均为整数，一定存在整数 x, y 使得 $ax + by = gcd(a, b)$ 成立。

推论：对于方程 $ax + by = c$，如果 $gcd(a, b) \mid c$，则方程一定有解，反之一定无解。

5.8 扩展欧几里得算法

求得的是 $ax + by = gcd(a, b)$ 的一组特解，该方程的通解可以表示为

$$\begin{cases} x' = x + k\frac{b}{gcd(a,b)} \\ y' = y - k\frac{a}{gcd(a,b)} \end{cases} \quad (k \in Z)$$

```
int exgcd(int a, int b, int &x, int &y) {
    if(b == 0) {
        x = 1, y = 0;
        return a;
    }
    int gcd = exgcd(b, a % b, y, x);
    y -= (a / b) * x;
    return gcd;
}
```

5.9 快速幂

```
int qpow(int x, int k, int Mod) {
    int res = 1;
    while(k) {
        if(k & 1)
            res = 1ll * res * x % Mod;
        x = 1ll * x * x % Mod;
        k >>= 1;
    }
    return res;
}
```

5.10 BSGS 离散对数

```
ll BSGS(ll x, ll y, ll mod) // y是x的多少次方（模意义下）
{
    x %= mod, y %= mod;
    if (x == 0) return y == 0 ? 1 : -1;
    if (y == 1) return 0;
    ll m = ceil(sqrt(mod));
    ll t = y;
    unordered_map<ll, ll> mp;
    mp[y] = 0;
    for (int i = 1; i < m; i++) {
        t = (t * x) % mod;
        mp[t] = i;
    }
    ll res = 1;
    t = 1;
    for (int i = 1; i <= m; i++) res = (res * x) % mod;
    for (int i = 1; i <= m; i++) {
```

```
        t = (res * t) % mod;
        if (mp.count(t)) {
            return (m * i - mp[t]) % mod;
        }
    }
    return -1;
}
```

5.11 乘法逆元

- 当 mod 为质数时, $ax \equiv 1 \pmod b$ 由费马小定理有 $ax \equiv a^{b-1} \pmod b$, $\therefore x \equiv a^{b-2} \pmod b$, 快速幂求 a^{b-2} 即可
- 扩展欧几里得方法（要求 $\gcd(a,b)=1$ ）：等价于求 $ax \equiv 1 \pmod p$ 的解，可以写为 $ax+pk=1$，求解 x,k 即可。

```
void exgcd(ll a, ll b, ll& x, ll& y)
{
    if (b == 0) {
        x = 1;
        y = 0;
        return;
    }
    exgcd(b, a % b, x, y);
    ll tmp = x;
    x = y;
    y = tmp - a / b * y;
}
ll getinv(int a, int mod) // 求a在mod下的逆元，不存在逆元返回-1
{
    ll x, y, d = exgcd(a, mod, x, y);
    return d == 1 ? (x % mod + mod) % mod : -1;
}
```

5.12 快速递推求逆元

以 $\mathcal{O}(N)$ 的复杂度完成 $1-N$ 中全部逆元的计算。

```
inv[1] = 1;
for (int i = 2; i <= n; i ++ )
    inv[i] = (p - p / i) * inv[p % i] % p;
```

5.13 中国剩余定理

$x \mod a_i = b_i$

```
ll CRT(int n, ll a[], ll b[]) // a为模数，b为余数
{
    ll ans = 0;
    ll s = 1;
```

```
    for (int i = 1; i <= n; i++) {
        s *= a[i];
    }
    for (int i = 1; i <= n; i++) {
        ll m = s / a[i];
        ll x, y;
        exgcd(m, a[i], x, y); // x为m在模a[i]下的逆元
        ans = (ans + (m * x * b[i]) % s + s) % s; // m*x不要对a[i]取模
    }
    return ans;
}
```

5.14 高斯消元

求解模意义下 n 元线性方程组

```
std::cin >> n >> p;
for (int i = 1; i <= n; i++) { // 读入增广矩阵
    for (int j = 1; j <= n + 1; j++) {
        std::cin >> a[i][j];
        a[i][j] %= p;
    }
}
int cnt = 1;
for (int i = 1; i <= n; i++) {
    int r = cnt;
    for (int j = cnt; j <= n; j++) {
        if (abs(a[j][i]) > abs(a[r][i])) {
            r = j;
        }
    }
    if (a[r][i] == 0) continue;
    if (r != cnt) std::swap(a[cnt], a[r]);
    ll inv = qpow(a[cnt][i], p - 2);
    for (int j = n + 1; j >= i; j--) {
        a[cnt][j] = (a[cnt][j] * inv) % p;
    }
    for (int j = cnt + 1; j <= n; j++) {
        if (a[j][i]) {
            for (int k = n + 1; k >= i; k--) {
                a[j][k] = (a[j][k] - (a[j][i] * a[cnt][k]) % p + p) % p;
            }
        }
    }
    cnt++;
}
if (cnt < n + 1) {
    for (int i = cnt; i <= n; i++) {
        if (a[i][n + 1]) {
            std::cout << -1 << "\n"; // 无解
            return 0;
        }
```

```
    }
    std::cout << 0 << "\n"; // 多解
    return 0;
}
for (int i = n; i >= 1; i--) {
    for (int j = i + 1; j <= n; j++) {
        a[i][n + 1] = (a[i][n + 1] - (a[i][j] * a[j][n + 1]) % p + p) % p;
    }
}
for (int i = 1; i <= n; i++) {
    std::cout << "x" << i << "=" << a[i][n + 1] << "\n";
}
return 0;
```

5.15 FFT

5.15.1 FFT 递归版

```
void FFT(complex<double> *A, int limit, int op)
{
    if (limit == 1)
        return;
    complex<double> A1[limit / 2], A2[limit / 2];
    for (int i = 0; i < limit / 2; i++)
    {
        A1[i] = A[i * 2], A2[i] = A[i * 2 + 1];
    }
    FFT(A1, limit / 2, op), FFT(A2, limit / 2, op);
    complex<double> w1 ({cos(2 * pi / limit), sin(2 * pi / limit) * op});
    complex<double> wk({1, 0});
    for (int i = 0; i < limit / 2; i++)
    {
        A[i] = A1[i] + A2[i] * wk;
        A[i + limit / 2] = A1[i] - A2[i] * wk;
        wk = wk * w1;
    }
}
```

5.15.2 FFT 迭代版

```
void change(complex<double> *A, int len)
{
    for (int i = 0; i < len; i++)
        R[i] = R[i / 2] / 2 + ((i & 1) ? len / 2 : 0);
    for (int i = 0; i < len; i++)
        if (i < R[i]) swap(A[i], A[R[i]]);
}

void FFT(complex<double> *A, int limit, int op)
{
    change(A, limit);
    for (int k = 2; k <= limit; k <<= 1)
```

```
    {
        complex<double> w1 ({cos(2 * pi / k), sin(2 * pi / k) * op});
        for (int i = 0; i < limit; i += k)
        {
            complex<double> wk({1, 0});
            for (int j = 0; j < k / 2; j++)
            {
                complex<double> x = A[i + j], y = A[i + j + k / 2] * wk;
                A[i + j] = x + y;
                A[i + j + k / 2] = x - y;
                wk = wk * w1;
            }
        }
    }
}
```

5.16 线性基

5.16.1 高斯消元法求线性基

高斯消元法构造线性基特点：

- 从大到小排列
- 各个基的高位没有重复的 1

```
int n, k;
ll a[N];

void gauss()
{
    for (int i = 63; i >= 0; i--) {
        for (int j = k; j < n; j++) {
            if (a[i] >> i & 1) {
                swap(a[i], a[k]);
                break;
            }
        }
        if ((a[k] >> i & 1) == 0) continue;
        for (int j = 0; j < n; j++) {
            if (j != k && (a[j] >> i & 1)) a[j] ^= a[k];
        }
        k++;
        if (k == n) break;
    }
}
```

5.16.2 贪心法求线性基

贪心法构造线性基的特点：

- 从大到小排列

- 各个基的高位可能存在重复的 1
- 线性基不是唯一的，与原集合即插入顺序有关

```
int n;
ll a[N];

void insert(ll x)
{
    for (int i = 63; i >= 0; i--) {
        if ((x >> i) & 1) {
            if (a[i])
                x ^= a[i];
            else {
                a[i] = x;
                break;
            }
        }
    }
}
```

5.17 数学常见结论

5.17.1 插板法

给定 n 个小球 m 个盒子。

- 球同，盒不同、不能空，隔板法：N 个小球即一共 $N-1$ 个空，分成 M 堆即 $M-1$ 个隔板，答案为 $\binom{n-1}{m-1}$ 。
- 球同，盒不同、能空，隔板法：多出 $M-1$ 个虚空球，答案为 $\binom{m-1+n}{n}$ 。
- 球同，盒同、能空：$\frac{1}{(1-x)(1-x^2)\dots(1-x^m)}$ 的 x^n 项的系数。动态规划，答案为

$$dp[i][j] = \begin{cases} dp[i][j-1] + dp[i-j][j] & \text{if } i \geq j \\ dp[i][j-1] & \text{if } i < j \\ 1 & \text{if } j = 1 \text{ or } i \leq 1 \end{cases}$$

- 球同，盒同、不能空：$\frac{x^m}{(1-x)(1-x^2)\dots(1-x^m)}$ 的 x^n 项的系数。动态规划，答案为

$$dp[n][m] = \begin{cases} dp[n-m][m] & \text{if } n \geq m \\ 0 & \text{if } n < m \end{cases}$$

- 球不同，盒同、不能空：第二类斯特林数 `Stirling2`(n, m) ，答案为

$$dp[n][m] = \begin{cases} m \cdot dp[n-1][m] + dp[n-1][m-1] & \text{if } 1 \le m < n \\ 1 & \text{if } 0 \le n = m \\ 0 & \text{if } m = 0 \text{ and } 1 \le n \end{cases}$$

- 球不同，盒同、能空：第二类斯特林数之和 $\sum_{i=1}^{m}$ `Stirling2`(n, m) ，答案为 $\sum_{i=0}^{m} dp[n][i]$ 。
- 球不同，盒不同、不能空：第二类斯特林数乘上 m 的阶乘 $m! \cdot$ `Stirling2`(n, m) ，答案为 $dp[n][m] * m!$ 。
- 球不同，盒不同、能空：答案为 m^n 。

5.17.2 组合数学常见性质

- $k * C_n^k = n * C_{n-1}^{k-1}$ ；
- $C_k^n * C_m^k = C_m^n * C_{m-n}^{m-k}$ ；
- $C_n^k + C_n^{k+1} = C_{n+1}^{k+1}$ ；
- $\sum_{i=0}^{n} C_n^i = 2^n$ ；
- $\sum_{k=0}^{n} (-1)^k * C_n^k = 0$ 。
- 二项式反演：$\begin{cases} f_n = \sum\limits_{i=0}^{n} \binom{n}{i} g_i \Leftrightarrow g_n = \sum\limits_{i=0}^{n} (-1)^{n-i} \binom{n}{i} f_i \\ f_k = \sum\limits_{i=k}^{n} \binom{i}{k} g_i \Leftrightarrow g_k = \sum\limits_{i=k}^{n} (-1)^{i-k} \binom{i}{k} f_i \end{cases}$ ；
- $\sum\limits_{i=1}^{n} i \binom{n}{i} = n * 2^{n-1}$ ；
- $\sum\limits_{i=1}^{n} i^2 \binom{n}{i} = n * (n+1) * 2^{n-2}$ ；
- $\sum\limits_{i=1}^{n} \frac{1}{i} \binom{n}{i} = \sum\limits_{i=1}^{n} \frac{1}{i}$ ；
- $\sum\limits_{i=0}^{n} \binom{n}{i}^2 = \binom{2n}{n}$ ；
- 拉格朗日恒等式：$\sum\limits_{i=1}^{n} \sum\limits_{j=i+1}^{n} (a_i b_j - a_j b_i)^2 = (\sum\limits_{i=1}^{n} a_i)^2 (\sum\limits_{i=1}^{n} b_i)^2 - (\sum\limits_{i=1}^{n} a_i b_i)^2$ 。

5.17.3 斯特林数

第二类斯特林数（斯特林子集数） $\left\{ n \atop k \right\}$，也可记做 $S(n,k)$，表示将 n 个两两不同的元素，划分为 k 个互不区分的非空子集的方案数。递推式

$$\left\{ n \atop k \right\} = \left\{ n-1 \atop k-1 \right\} + k \left\{ n-1 \atop k \right\}$$

边界是 $\left\{ n \atop 0 \right\} = [n = 0]$

考虑用组合意义来证明。我们插入一个新元素时，有两种方案：

- 将新元素单独放入一个子集，有 $\left\{ n-1 \atop k-1 \right\}$ 种方案；
- 将新元素放入一个现有的非空子集，有 $k \left\{ n-1 \atop k \right\}$ 种方案。

根据加法原理，将两式相加即可得到递推式。

第一类斯特林数（斯特林轮换数） $\left[n \atop k \right]$，也可记做 $s(n,k)$，表示将 n 个两两不同的元素，划分为 k 个互不区分的非空轮换的方案数。

一个轮换就是一个首尾相接的环形排列。我们可以写出一个轮换 $[A,B,C,D]$，并且我们认为 $[A,B,C,D] = [B,C,D,A] = [C,D,A,B] = [D,A,B,C]$，即，两个可以通过旋转而互相得到的轮换是等价的。注意，我们不认为两个可以通过翻转而相互得到的轮换等价，即 $[A,B,C,D] \neq [D,C,B,A]$。递推式

$$\left[n \atop k \right] = \left[n-1 \atop k-1 \right] + (n-1) \left[n-1 \atop k \right]$$

边界是 $\left[n \atop 0 \right] = [n = 0]$。

该递推式的证明可以考虑其组合意义。我们插入一个新元素时，有两种方案：

- 将该新元素置于一个单独的轮换中，共有 $\left[n-1 \atop k-1 \right]$ 种方案；
- 将该元素插入到任何一个现有的轮换中，共有 $(n-1) \left[n-1 \atop k \right]$ 种方案。

根据加法原理，将两式相加即可得到递推式。

5.17.4 卡特兰数

是一类奇特的组合数，前几项为 $1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862$ 。如遇到以下问题，则直接套用即可。

- 【括号匹配问题】n 个左括号和 n 个右括号组成的合法括号序列的数量，为 Cat_n 。
- 【进出栈问题】$1, 2, ..., n$ 经过一个栈，形成的合法出栈序列的数量，为 Cat_n 。
- 【二叉树生成问题】n 个节点构成的不同二叉树的数量，为 Cat_n 。
- 【路径数量问题】在平面直角坐标系上，每一步只能 ** 向上 ** 或 ** 向右 ** 走，从 $(0, 0)$ 走到 (n, n) ，并且除两个端点外不接触直线 $y = x$ 的路线数量，为 $2Cat_{n-1}$ 。

计算公式：$Cat_n = \dfrac{C_{2n}^n}{n+1}$ ，$C_n = \dfrac{C_{n-1} * (4n-2)}{n+1}$ 。

5.17.5 斐波那契数列

通项公式：$F_n = \dfrac{1}{\sqrt{5}} * \left[\left(\dfrac{1+\sqrt{5}}{2}\right)^n - \left(\dfrac{1-\sqrt{5}}{2}\right)^n\right]$ 。

直接结论：

- 卡西尼性质：$F_{n-1} * F_{n+1} - F_n^2 = (-1)^n$ ；
- $F_n^2 + F_{n+1}^2 = F_{2n+1}$ ；
- $F_{n+1}^2 - F_{n-1}^2 = F_{2n}$ （由上一条写两遍相减得到）；
- 若存在序列 $a_0 = 1, a_n = a_{n-1} + a_{n-3} + a_{n-5} + ...(n \geq 1)$ 则 $a_n = F_n(n \geq 1)$ ；
- 齐肯多夫定理：任何正整数都可以表示成若干个不连续的斐波那契数（F_2 开始）可以用贪心实现。

求和公式结论：

- 奇数项求和：$F_1 + F_3 + F_5 + ... + F_{2n-1} = F_{2n}$ ；
- 偶数项求和：$F_2 + F_4 + F_6 + ... + F_{2n} = F_{2n+1} - 1$ ；
- 平方和：$F_1^2 + F_2^2 + F_3^2 + ... + F_n^2 = F_n * F_{n+1}$ ；
- $F_1 + 2F_2 + 3F_3 + ... + nF_n = nF_{n+2} - F_{n+3} + 2$ ；
- $-F_1 + F_2 - F_3 + ... + (-1)^n F_n = (-1)^n(F_{n+1} - F_n) + 1$ ；
- $F_{2n-2m-2}(F_{2n} + F_{2n+2}) = F_{2m+2} + F_{4n-2m}$ 。

数论结论：

- $F_a \mid F_b \Leftrightarrow a \mid b$ ；
- $\gcd(F_a, F_b) = F_{\gcd(a,b)}$ ；

- 当 p 为 $5k \pm 1$ 型素数时，$\begin{cases} F_{p-1} \equiv 0 \pmod p \\ F_p \equiv 1 \pmod p \\ F_{p+1} \equiv 1 \pmod p \end{cases}$；

- 当 p 为 $5k \pm 2$ 型素数时，$\begin{cases} F_{p-1} \equiv 1 \pmod p \\ F_p \equiv -1 \pmod p \\ F_{p+1} \equiv 0 \pmod p \end{cases}$；

- $F(n)\%m$ 的周期 $\le 6m$ （$m = 2 \times 5^k$ 时取到等号）；

- 既是斐波那契数又是平方数的有且仅有 $1, 144$ 。

6 博弈论

6.1 巴什博弈

6.1.1 朴素巴什博弈

有 N 个石子，两名玩家轮流行动，按以下规则取石子：

规定：每人每次可以取走 $X(1 \le X \le M)$ 个石子，拿到最后一颗石子的一方获胜。

双方均采用最优策略，询问谁会获胜。

两名玩家轮流报数。

规定：第一个报数的人可以报 $X(1 \le X \le M)$ ，后报数的人需要比前者所报数大 $Y(1 \le Y \le M)$ ，率先报到 N 的人获胜。

双方均采用最优策略，询问谁会获胜。

- $N = K \cdot (M+1)$ （其中 $K \in \mathbb{N}^+$ ），后手必胜（后手可以控制每一回合结束时双方恰好取走 $M+1$ 个，重复 K 轮后即胜利）；

- $N = K \cdot (M+1) + R$ （其中 $K \in \mathbb{N}^+, 0 < R < M+1$ ），先手必胜（先手先取走 R 个，之后控制每一回合结束时双方恰好取走 $M+1$ 个，重复 K 轮后即胜利）。

6.1.2 扩展巴什博弈

有 N 颗石子，两名玩家轮流行动，按以下规则取石子：。

规定：每人每次可以取走 $X(a \le X \le b)$ 个石子，如果最后剩余物品的数量小于 a 个，则不能再取，拿到最后一颗石子的一方获胜。

双方均采用最优策略，询问谁会获胜。

- $N = K \cdot (a+b)$ 时，后手必胜；

- $N = K \cdot (a+b) + R_1$ （其中 $K \in \mathbb{N}^{+}, 0 < R_1 < a$ ）时，后手必胜（这些数量不够再取一次，先手无法逆转局面）；
- $N = K \cdot (a+b) + R_2$ （其中 $K \in \mathbb{N}^{+}, a \le R_2 \le b$ ）时，先手必胜；
- $N = K \cdot (a+b) + R_3$ （其中 $K \in \mathbb{N}^{+}, b < R_3 < a+b$ ）时，先手必胜（这些数量不够再取一次，后手无法逆转局面）。

6.2 Nim 博弈

6.2.1 Nim 博弈

有 N 堆石子，给出每一堆的石子数量，两名玩家轮流行动，按以下规则取石子：

规定：每人每次任选一堆，取走正整数颗石子，拿到最后一颗石子的一方获胜（注：几个特点是**不能跨堆、不能不拿**）。

双方均采用最优策略，询问谁会获胜。

记初始时各堆石子的数量 $(A_1, A_2, \ldots, A_n)$ ，定义尼姆和 $Sum_N = A_1 \oplus A_2 \oplus \cdots \oplus A_n$ 。

当 $\boldsymbol{Sum_N = 0}$ 时先手必败，反之先手必胜。

具体取法

先计算出尼姆和，再对每一堆石子计算 $A_i \oplus Sum_N$ ，记为 X_i 。

若得到的值 $X_i < A_i$ ， X_i 即为一个可行解，即剩下 $\boldsymbol{X}_i$ 颗石头，取走 $\boldsymbol{A_i - X_i}$ 颗石头（这里取小于号是因为至少要取走 1 颗石子）。

6.2.2 Nim_K

有 N 堆石子，给出每一堆的石子数量，两名玩家轮流行动，按以下规则取石子：

规定：每人每次任选不超过 K 堆，对每堆都取走不同的正整数颗石子，拿到最后一颗石子的一方获胜。

双方均采用最优策略，询问谁会获胜。

把每一堆石子的石子数用二进制表示，定义 One_i 为二进制第 i 位上 1 的个数。

以下局面先手必胜：

对于每一位，$\boldsymbol{One_1, One_2, \ldots, One_N}$ 均不为 $\boldsymbol{K+1}$ 的倍数。

6.2.3 反 Nim 游戏

有 N 堆石子，给出每一堆的石子数量，两名玩家轮流行动，按以下规则取石子：

规定：每人每次任选一堆，取走正整数颗石子，拿到最后一颗石子的一方 ** 出局 **。

双方均采用最优策略，询问谁会获胜。

- 所有堆的石头数量均不超过 1 ，且 $\boldsymbol{Sum_N = 0}$ （也可看作“且有偶数堆”）时；
- 至少有一堆的石头数量大于 1 ，且 $\boldsymbol{Sum_N \neq 0}$ 。

6.3 SG 游戏

6.3.1 SG 定理和 SG 函数

我们使用以下几条规则来定义暴力求解的过程:

- 使用数字来表示输赢情况，0 代表局面必败，非 0 代表 ** 存在必胜可能 **，我们称这个数字为这个局面的 SG 值；
- 找到最终态，根据题意人为定义最终态的输赢情况；
- 对于非最终态的某个节点，其 SG 值为所有子节点的 SG 值取 `mex`；
- 单个游戏的输赢态即对应根节点的 SG 值是否为 0 ，为 0 代表先手必败，非 0 代表先手必胜；
- 多个游戏的总 SG 值为单个游戏 SG 值的异或和。

使用哈希表，以 $\mathcal{O}(N+M)$ 的复杂度计算。

```
int getsg(int x) {
   if (sg[x] != -1) return sg[x];

   unordered_set<int> S;
   for (int v:G[x]) // 可能的转移
      if(x >= v)
         S.insert(sg(x - v));

   for (int i = 0; ; ++ i)
      if (S.count(i) == 0)
         return sg[x] = i;
}
```

6.3.2 反 SG 博弈

SG 游戏中最先不能行动的一方获胜。

以下局面先手必胜:

- 单局游戏的 SG 值均不超过 **1** ，且总 SG 值为 **0**;
- 至少有一局单局游戏的 SG 值大于 **1** ，且总 SG 值不为 **0** 。

在本质上，这与 Anti-Nim 游戏的结论一致。

7 计算几何

7.1 点和向量

```
#include <bits/stdc++.h>
using namespace std;
#define endl '\n'
#define ll long long
#define PII pair<int, int>
#define pi acos(-1.0)
const int N = 1e5 + 10;

#define eps 1e-8
inline int sgn(double x) { return fabs(x) < eps ? 0 : (x > 0 ? 1 : -1); } // 判断正负

struct Point {
   double x, y;
   Point(double nx = 0, double ny = 0) : x(nx), y(ny) {}
   void read() { cin >> this->x >> this->y; }
   inline Point operator+(Point A) { return Point(A.x + this->x, A.y + this->y); }
   inline Point operator-(Point A) { return Point(this->x - A.x, this->y - A.y); }
   inline Point operator*(double x) { return Point(this->x * x, this->y * x); }
   inline Point operator/(double x) { return Point(this->x / x, this->y / x); }
   inline bool operator==(Point A) { return this->x == A.x && this->y == A.y; }
   inline double operator*(Point A) { return this->x * A.x + this->y * A.y; } // 点乘
   inline double operator^(Point A) { return this->x * A.y - this->y * A.x; } // 叉积（注意先后顺序）
   inline double len() { return sqrt((*this) * (*this)); }
   inline Point rotate(double rad) { return Point(this->x * cos(rad) - this->y * sin(rad), this->x * sin(rad) + this->y * cos(rad)); }
};
Point projection(Point A, Point B, Point P) { return A + (B - A) / ((B - A).len()) * ((B - A) * (P - A)) / (B - A).len(); }// 求P在向量AB上的投影点

typedef Point Vector;
inline double Len(Vector A) { return sqrt(A * A); }
inline double Ang(Vector A, Vector B) { return acos(A * B) / A.len() / B.len(); } // 两个向量夹角
inline double Area(Vector A, Vector B) { return (A ^ B) / 2; } // 两个向量组成的三角形的面积
inline Vector Normal(Vector A) // 与A正交的单位向量
{
   double len = A.len();
   return Vector(-A.y / len, A.x / len);
}
inline int toleft(Vector A, Vector B) // toleft测试：B在A左边为1，右边为-1，方向相同为0
{
   double t = A ^ B;
   return (t > eps) - (t < -eps);
}

bool argcmp(Point a, Point b) // 极角排序
{
```

```
    auto quad = [](Point& a) {
        if (a.y < -eps) return 1;
        if (a.y > eps) return 4;
        if (a.x < -eps) return 5;
        if (a.x > eps) return 3;
        return 2;
    };
    int qa = quad(a), qb = quad(b);
    if (qa != qb) return qa < qb;
    auto t = a ^ b;
    // if (abs(t)<=eps) return a*a<b*b-eps; // 不同长度的向量需要分开
    return t > eps;
}
```

7.2 直线和线段

```
struct Line {
    Point A, B; //A为方向向量, B为直线上一点
    Line(Point x = Point(0, 0), Point y = Point(0, 0)) : A(x), B(y) {}
    void read() { cin >> this->A.x >> this->A.y >> this->B.x >> this->B.y; }
    double dis(Point p) { return fabs((p - this->A) ^ (this->A - this->B)) / Len(
        this->A - this->B); } // 点到直线的距离
};

struct Seg {
    Point A, B;
    Seg(Point x = Point(0, 0), Point y = Point(0, 0)) : A(x), B(y) {}
    void read() { cin >> this->A.x >> this->A.y >> this->B.x >> this->B.y; }
    double dis(Point p) // 点到线段距离
    {
        if ((p - this->A) * (this->B - this->A) < -eps || (p - this->B) * (this->A -
            this->B) < -eps) return min(Len(p - this->A), Len(p - this->B));
        return Line{A, B}.dis(p);
    }
    int on(Point p) // -1表示在端点, 1表示在线段上, 0表示不在线段上
    {
        if (p == this->A || p == this->B) return -1;
        return (toleft(p - this->A, p - this->B) == 0 && (p - this->A) * (p - this->B
            ) < -eps);
    }
};

inline bool IsIntersect(Seg S1, Seg S2) // 两个线段是否相交
{
    double f1 = (S1.B - S1.A) ^ (S2.A - S1.A), f2 = (S1.B - S1.A) ^ (S2.B - S1.A);
    double g1 = (S2.B - S2.A) ^ (S1.A - S2.A), g2 = (S2.B - S2.A) ^ (S1.B - S2.A);
    return ((f1 < 0) ^ (f2 < 0)) && ((g1 < 0) ^ (g2 < 0));
}
inline Point LineIntersection(Line L1, Line L2) { return L1.A + (L1.B - L1.A) * ((L2
    .B - L2.A) ^ (L1.A - L2.A)) / ((L1.B - L1.A) ^ (L2.B - L2.A)); } // 两条直线求交
    点
```

7.3 凸包

```
struct Convex {
    vector<Point> p;

    int in(Point a) // -1表示在边上，0在凸多边形外，1在凸多边形内
    {
        auto& p = this->p;
        if (p.size() == 1) return a == p[0] ? -1 : 0;
        if (p.size() == 2) return Seg{p[0], p[1]}.on(a) ? -1 : 0;
        if (a == p[0]) return -1;
        if (toleft(p[1] - p[0], a - p[0]) == -1 || toleft(p.back() - p[0], a - p[0])
             == 1) return 0;
        auto cmp = [&](Point u, Point v) { return toleft(u - p[0], v - p[0]) == 1; };
        int l = lower_bound(p.begin() + 1, p.end(), a, cmp) - p.begin();
        if (l == 1) return Seg{p[0], p[l]}.on(a) ? -1 : 0;
        if (l == p.size() - 1 && Seg{p[0], p[l]}.on(a)) return -1;
        if (Seg{p[l - 1], p[l]}.on(a)) return -1;
        return (toleft(p[l] - p[l - 1], a - p[l - 1]) > 0);
    }

    double circ() // 周长
    {
        double sum = 0;
        for (int i = 0; i < p.size(); i++) {
            sum += Len(p[i] - p[i + 1 == p.size() ? 0 : i + 1]);
        }
        return sum;
    }

    double area()
    {
        double sum = 0;
        for (int i = 0; i < p.size(); i++) {
            sum += p[i] ^ p[(i + 1) % p.size()];
        }
        return sum;
    }

    Convex operator+(Convex& c) // 闵可夫斯基和
    {
        auto& p = this->p;
        vector<Point> res;
        vector<Seg> e1(p.size()), e2(c.p.size()), edge(p.size() + c.p.size());
        res.reserve(p.size() + c.p.size());
        auto cmp = [](Seg& u, Seg& v) { return argcmp(u.B - u.A, v.B - v.A); };
        for (int i = 0; i < p.size(); i++) e1[i] = {p[i], p[i + 1 == p.size() ? 0 : i
             + 1]};
        for (int i = 0; i < c.p.size(); i++) e2[i] = {c.p[i], c.p[i + 1 == c.p.size()
             ? 0 : i + 1]};
        rotate(e1.begin(), min_element(e1.begin(), e1.end(), cmp), e1.end());
        rotate(e2.begin(), min_element(e2.begin(), e2.end(), cmp), e2.end());
        merge(e1.begin(), e1.end(), e2.begin(), e2.end(), edge.begin(), cmp);
```

```
        auto check = [&](Point& u) {
            auto last = res.back(), last2 = res[res.size() - 2];
            return toleft(last - last2, u - last) == 0 && (last - last2) * (u - last)
                >= -eps;
        };
        auto u = e1[0].A + e2[0].A;
        for (auto v : edge) {
            while (res.size() > 1 && check(u)) res.pop_back();
            res.push_back(u);
            u = u + v.B - v.A;
        }
        if (res.size() > 1 && check(res[0])) res.pop_back();
        return Convex{res};
    }
};

Convex Andrew(vector<Point> p) // Andrew求凸包O(n\log n)
{
    vector<Point> st;
    if (p.empty()) return Convex{st};
    sort(p.begin(), p.end(), [](Point& A, Point& B) { return fabs(A.x - B.x) > eps ?
         A.x < B.x : A.y < B.y; });

    auto check = [&](Point cur) {
        auto last = st.back(), last2 = st[st.size() - 2];
        return toleft(last - last2, cur - last) <= 0;
    };

    for (auto cur : p) {
        while (st.size() > 1 && check(cur)) st.pop_back();
        st.push_back(cur);
    }
    int t = st.size();
    p.pop_back();
    reverse(p.begin(), p.end());
    for (auto cur : p) {
        while (st.size() > t && check(cur)) st.pop_back();
        st.push_back(cur);
    }
    st.pop_back();
    return Convex{st};
}
```