目录

基础算法

常用函数

```
int mypow(int n, int k, int p = MOD) { // 复杂度是 log N
    int r = 1;
    for (; k; k >>= 1, n = n * n % p) {
        if (k & 1) r = r * n % p;
    }
    return r;
}
i64 mysqrt(i64 n) { // 针对 sqrt 无法精确计算 ll 型
    i64 ans = sqrt(n);
```

```
    while ((ans + 1) * (ans + 1) <= n) ans++;
    while (ans * ans > n) ans--;
    return ans;
}
int mylcm(int x, int y) {
    return x / gcd(x, y) * y;
}

template<typename T> int log2floor(T n) { // 针对 log2 无法精确计算 ll 型; 向下取整
    assert(n > 0);
    for (T i = 0, chk = 1;; i++, chk *= 2) {
        if (chk <= n && n < chk * 2) {
            return i;
        }
    }
}
template<typename T> int log2ceil(T n) { // 向上取整
    assert(n > 0);
    for (T i = 0, chk = 1;; i++, chk *= 2) {
        if (n <= chk) {
            return i;
        }
    }
}
int log2floor(int x) {
    return 31 - __builtin_clz(x);
}
int log2ceil(int x) { // 向上取整
    return log2floor(x) + (__builtin_popcount(x) != 1);
}

template<typename T> T sign(const T &a) {
    return a == 0 ? 0 : (a < 0 ? -1 : 1);
}
```

```cpp
template<typename T> T floor(const T &a, const T &b) { // 注意大数据计算时会丢失精度
    T A = abs(a), B = abs(b);
    assert(B != 0);
    return sign(a) * sign(b) > 0 ? A / B : -(A + B - 1) / B;
}
template<typename T> T ceil(const T &a, const T &b) { // 注意大数据计算时会丢失精度
    T A = abs(a), B = abs(b);
    assert(b != 0);
    return sign(a) * sign(b) > 0 ? (A + B - 1) / B : -A / B;
}
```

最大公约数 gcd

欧几里得算法 速度不如内置函数！以 $\mathcal{O}(\log(a+b))$ 的复杂度求解最大公约数。与内置函数 __gcd 功能基本相同（支持 $a, b \leq 0$ ）。

```cpp
inline int mygcd(int a, int b) { return b ? gcd(b, a % b) : a; }
```

位运算优化 略快于内置函数，用于卡常。

```cpp
LL gcd(LL a, LL b) { // 卡常 gcd!!
    #define tz __builtin_ctzll
    if (!a || !b) return a | b;
    int t = tz(a | b);
    a >>= tz(a);
    while (b) {
        b >>= tz(b);
        if (a > b) swap(a, b);
        b -= a;
    }
    return a << t;
    #undef tz
}
```

整数域二分

by flyx

```
auto l = l, r = r;
auto check = [&](auto x)->bool {

    };
while (l < r) {
    auto mid = l + r >> 1;
    if (check(mid))r = mid;
    else l = mid + 1;
}
```

旧版（无法处理负数情况）

- 在递增序列 a 中查找 $\geq x$ 数中最小的一个（即 x 或 x 的后继）

```
while (l < r) {
    int mid = (l + r) / 2;
    if (a[mid] >= x) {
        r = mid;
    } else {
        l = mid + 1;
    }
}
return a[l];
```

- 在递增序列 a 中查找 $\leq x$ 数中最大的一个（即 x 或 x 的前驱）

```
while (l < r) {
    int mid = (l + r + 1) / 2;
    if (a[mid] <= x) {
        l = mid;
    } else {
```

```
        r = mid - 1;
    }
}
return a[l];
```

新版

- x 或 x 的后继

```
int l = 0, r = 1E8, ans = r;
while (l <= r) {
    int mid = (l + r) / 2;
    if (judge(mid)) {
        r = mid - 1;
        ans = mid;
    } else {
        l = mid + 1;
    }
}
return ans;
```

- x 或 x 的前驱

```
int l = 0, r = 1E8, ans = l;
while (l <= r) {
    int mid = (l + r) / 2;
    if (judge(mid)) {
        l = mid + 1;
        ans = mid;
    } else {
        r = mid - 1;
    }
}
return ans;
```

整体二分

1. 询问的答案具有可二分性
2. **修改对判定答案的贡献互相独立**，修改之间互不影响效果
3. 修改如果对判定答案有贡献，则贡献为一确定的与判定标准无关的值
4. 贡献满足交换律，结合律，具有可加性
5. 题目允许使用离线算法

——许昊然《浅谈数据结构题几个非经典解法》

```
int cal(auto x) {
    // todo
}

void solve(int ql, int qr, int l, int r) {
    if (ql > qr)return;
    if (l > r)return;
    if (l == r) {
        for (int q = ql;q <= qr;++q)ans[q] = l;
        return;
    }
    int mid = l + r + 1 >> 1;
    int cnt = cal(mid);
    solve(std::max(ql, cnt + 1), qr, l, mid - 1);
    solve(ql, std::min(qr, cnt), mid, r);
}

void solve() {
    //input
    solve(ql, qr, 0, n, zf);
    //todo
}
```

实数域二分

目前主流的写法是限制二分次数。

```
for (int t = 1; t <= 100; t++) {
    ld mid = (l + r) / 2;
    if (judge(mid)) r = mid;
    else l = mid;
}
cout << l << endl;
```

整数域三分

```
while (l < r) {
    int mid = (l + r) / 2;
    if (check(mid) <= check(mid + 1)) r = mid;
    else l = mid + 1;
}
cout << check(l) << endl;
```

实数域三分

限制次数实现。

```
ld l = -1E9, r = 1E9;
for (int t = 1; t <= 100; t++) {
    ld mid1 = (l * 2 + r) / 3;
    ld mid2 = (l + r * 2) / 3;
    if (judge(mid1) < judge(mid2)) {
        r = mid2;
    } else {
        l = mid1;
    }
```

```
}
cout << l << endl;
```

数据结构 A

笛卡尔树

小根笛卡尔树

```
cin >> n;
for (int i = 0;i < n;++i)cin >> nums[i];
for (int i = 0;i < n;++i)rs[i] = -1;
for (int i = 0;i < n;++i)ls[i] = -1;
top = 0;
for (int i = 0; i < n; i++) {
    int k = top;
    while (k > 0 && nums[stk[k - 1]] > nums[i]) k--;
    if (k) rs[stk[k - 1]] = i;  // rs 代表笛卡尔树每个节点的右儿子
    if (k < top) ls[i] = stk[k];  // ls 代表笛卡尔树每个节点的左儿子
    stk[k++] = i;
    top = k;
}
```

dsu 并查集

路径优化 (普遍)

```
struct dsu {
    std::vector<int> d;
    dsu(int n) { d.resize(n + 1); iota(d.begin(), d.end(), 0); }
```

```
    int get_root(int x) { return d[x] = (x == d[x] ? x : get_root(d[x])); };
    bool merge(int u, int v) {
        if (get_root(u) != get_root(v)) {
            d[get_root(u)] = get_root(v);
            return true;
        }
        else return false;
    }
};
```

根据集合的大小优化

```
//左移位数根据节点个数定
#define UFLIMIT (2<<17)
int unicnt[UFLIMIT];
void ufinit(int n) {
    for (int i = 0;i < n;i++)unicnt[i] = 1;
}
int ufroot(int x) { return unicnt[x] <= 0 ? -(unicnt[x] = -ufroot(-unicnt[x])) : x; }
int ufsame(int x, int y) { return ufroot(x) == ufroot(y); }
void uni(int x, int y) {
    if ((x = ufroot(x)) == (y = ufroot(y)))return;
    if (unicnt[x] < unicnt[y])std::swap(x, y);
    unicnt[x] += unicnt[y];
    unicnt[y] = -x;
}
```

按秩合并优化

```
class UnionFind {
private:
    std::vector<int> parent;
    std::vector<int> rank;
```

```
public:
    UnionFind(int n) {
        parent.resize(n, 0);
        rank.resize(n, 0);
        iota(parent.begin(), parent.end(), 0);
    }
    int find(int x) {
        if (parent[x] == x)
            return x;
        return parent[x] = find(parent[x]);
    }
    void merge(int x, int y) {
        int rootX = find(x);
        int rootY = find(y);
        if (rootX == rootY) return;
        if (rank[rootX] > rank[rootY])
            std::swap(rootX, rootY);
        parent[rootX] = rootY;
        if (rank[rootX] == rank[rootY]) {
            rank[rootY]++;
        }
    }
    bool isConnect(int x, int y) {
        return find(x) == find(y);
    }
};
```

常用操作

```
struct DSU {
    vector<int> fa, p, e, f;

    DSU(int n) {
```

```
        fa.resize(n + 1);
        iota(fa.begin(), fa.end(), 0);
        p.resize(n + 1, 1);
        e.resize(n + 1);
        f.resize(n + 1);
    }
    int get(int x) {
        while (x != fa[x]) {
            x = fa[x] = fa[fa[x]];
        }
        return x;
    }
    bool merge(int x, int y) { // 设 x 是 y 的祖先
        if (x == y) f[get(x)] = 1;
        x = get(x), y = get(y);
        e[x]++;
        if (x == y) return false;
        if (x < y) swap(x, y); // 将编号小的合并到大的上
        fa[y] = x;
        f[x] |= f[y], p[x] += p[y], e[x] += e[y];
        return true;
    }
    bool same(int x, int y) {
        return get(x) == get(y);
    }
    bool F(int x) { // 判断连通块内是否存在自环
        return f[get(x)];
    }
    int size(int x) { // 输出连通块中点的数量
        return p[get(x)];
    }
    int E(int x) { // 输出连通块中边的数量
        return e[get(x)];
```

```
    }
};
```

ST 表

用于解决区间可重复贡献问题, 需要满足 x 运算符$x = x$ (如区间最大值:max$(x,x) = x$, 区间 gcd:gcd$(x,x) = x$ 等), 但是不支持修改操作.$\mathcal{O}(N \log N)$ 预处理,$\mathcal{O}(1)$ 查询.

```
template<typename T>
struct sparse_table
{
    std::vector<std::vector<T>> vt;
    sparse_table(std::vector<T> a) {
        int n = a.size();
        vt.assign(n, std::vector<T>(30));
        for (int i = 0;i < n;++i)
            vt[i][0] = a[i];
        for (int s = 1;s < 30;++s) {
            for (int i = 0;i < n;++i) {
                int j = i + (1 << s - 1);
                if (j < n) {
                    vt[i][s] = vt[i][s - 1] + vt[i + (1 << s - 1)][s - 1];
                }
                else vt[i][s] = vt[i][s - 1];
            }
        }
    }
    T query(int l, int r) {//[l,r)
        if (l == r) return T(0);
        int len = r - l;
        int x = std::__lg(len);
        return vt[l][x] + vt[r - (1 << x)][x];
```

```
    }
};

struct Info
{
    i64 a;
    Info operator+(Info x) {
        return Info(std::max(a, x.a));
    }
};
```

Fenwick Tree 树状数组

```
template<typename T> struct BIT {
    int n;
    vector<T> w;
    BIT(int n, auto &in) : n(n), w(n + 1) { // 预处理填值
        for (int i = 1; i <= n; i++) {
            add(i, in[i]);
        }
    }
    void add(int x, T v) {
        for (; x <= n; x += x & -x) {
            w[x] += v;
        }
    }
    T ask(int x) { // 前缀和查询
        T ans = 0;
        for (; x; x -= x & -x) {
            ans += w[x];
        }
        return ans;
    }
```

```
    T ask(int l, int r) { // 差分实现区间和查询
        return ask(r) - ask(l - 1);
    }
};
```

逆序对扩展

```
struct BIT {
    int n;
    vector<int> w, chk; // chk 为传入的待处理数组
    BIT(int n, auto &in) : n(n), w(n + 1), chk(in) {}
    /* 需要全部常规封装 */
    int get() {
        vector<array<int, 2>> alls;
        for (int i = 1; i <= n; i++) {
            alls.push_back({chk[i], i});
        }
        sort(alls.begin(), alls.end());
        int ans = 0;
        for (auto [val, idx] : alls) {
            ans += ask(idx + 1, n);
            add(idx, 1);
        }
        return ans;
    }
};
```

前驱后继扩展 (常规 + 第 k 小值查询 + 元素排名查询 + 元素前驱后继查询) 注意, 被查询的值都应该小于等于 N , 否则会越界; 如果离散化不可使用, 则需要使用平衡树替代.

```
struct BIT {
    int n;
```

```
    vector<int> w;
    BIT(int n) : n(n), w(n + 1) {}
    void add(int x, int v) {
        for (; x <= n; x += x & -x) {
            w[x] += v;
        }
    }
    int kth(int x) { // 查找第 k 小的值
        int ans = 0;
        for (int i = __lg(n); i >= 0; i--) {
            int val = ans + (1 << i);
            if (val < n && w[val] < x) {
                x -= w[val];
                ans = val;
            }
        }
        return ans + 1;
    }
    int get(int x) { // 查找 x 的排名
        int ans = 1;
        for (x--; x; x -= x & -x) {
            ans += w[x];
        }
        return ans;
    }
    int pre(int x) { return kth(get(x) - 1); } // 查找 x 的前驱
    int suf(int x) { return kth(get(x + 1)); } // 查找 x 的后继
};
const int N = 10000000; // 可以用于在线处理平衡二叉树的全部要求
signed main() {
    BIT bit(N + 1); // 在线处理不能够离散化，一定要开到比最大值更大
    int n;
    cin >> n;
```

```
    for (int i = 1; i <= n; i++) {
        int op, x;
        cin >> op >> x;
        if (op == 1) bit.add(x, 1); // 插入 x
        else if (op == 2) bit.add(x, -1); // 删除任意一个 x
        else if (op == 3) cout << bit.get(x) << "\n"; // 查询 x 的排名
        else if (op == 4) cout << bit.kth(x) << "\n"; // 查询排名为 x 的数
        else if (op == 5) cout << bit.pre(x) << "\n"; // 求小于 x 的最大值 (前驱)
        else if (op == 6) cout << bit.suf(x) << "\n"; // 求大于 x 的最小值 (后继)
    }
}
```

最值查询扩展 (常规 + 区间最值查询 + 单点赋值) 以 $\mathcal{O}(\log\log N)$ 的复杂度运行, 但是即便如此依然略优于线段树 (后者常数较大).

```
template<typename T> struct BIT {
    int n;
    vector<T> w, base;
    #define low(x) (x & -x)
    BIT(int n, auto &in) : n(n), w(n + 1), base(n + 1) {
        for (int i = 1; i <= n; i++) {
            update(i, in[i]);
        }
    } /* 可以增加并使用常规封装中的几个函数 */
    void update(int x, int v) { // 单点赋值
        base[x] = max(base[x], v);
        for (; x <= n; x += low(x)) {
            w[x] = max(w[x], v);
        }
    }
    T getMax(int l, int r) { // 最值查询
        T ans = T();
        while (r >= l) {
```

```
            ans = max(base[r], ans);
            for (r--; r - low(r) >= l; r -= low(r)) {
                ans = max(w[r], ans);
            }
        }
        return ans;
    }
};
```

二维树状数组

封装一: 该版本不能同时进行区间修改 + 区间查询. 无离散化版本的空间占用为 $\mathcal{O}(NM)$, 建树复杂度为 $\mathcal{O}(NM)$, 单次查询复杂度为 $\mathcal{O}(\log N \cdot \log M)$.

```
struct BIT_2D {
    int n, m;
    vector<vector<int>> w;

    BIT_2D(int n, int m) : n(n), m(m) {
        w.resize(n + 1, vector<int>(m + 1));
    }
    void add(int x, int y, int k) {
        for (int i = x; i <= n; i += i & -i) {
            for (int j = y; j <= m; j += j & -j) {
                w[i][j] += k;
            }
        }
    }
    void add(int x, int y, int X, int Y, int k) { // 区块修改: 二维差分
        X++, Y++;
        add(x, y, k), add(X, y, -k);
        add(X, Y, k), add(x, Y, -k);
```

```
    }
    int ask(int x, int y) { // 单点查询
        int ans = 0;
        for (int i = x; i; i -= i & -i) {
            for (int j = y; j; j -= j & -j) {
                ans += w[i][j];
            }
        }
        return ans;
    }
    int ask(int x, int y, int X, int Y) { // 区块查询：二维前缀和
        x--, y--;
        return ask(X, Y) - ask(x, Y) - ask(X, y) + ask(x, y);
    }
};
```

封装二: 该版本支持全部操作. 但是时空复杂度均比上一个版本多 4 倍.

```
struct BIT_2D {
    int n, m;
    vector<vector<int>> b1, b2, b3, b4;

    BIT_2D(int n, int m) : n(n), m(m) {
        b1.resize(n + 1, vector<int>(m + 1));
        b2.resize(n + 1, vector<int>(m + 1));
        b3.resize(n + 1, vector<int>(m + 1));
        b4.resize(n + 1, vector<int>(m + 1));
    }
    void add(auto &w, int x, int y, int k) { // 单点修改
        for (int i = x; i <= n; i += i & -i) {
            for (int j = y; j <= m; j += j & -j) {
                w[i][j] += k;
            }
        }
```

```
}
void add(int x, int y, int k) { // 多了一步计算
    add(b1, x, y, k);
    add(b2, x, y, k * (x - 1));
    add(b3, x, y, k * (y - 1));
    add(b4, x, y, k * (x - 1) * (y - 1));
}
void add(int x, int y, int X, int Y, int k) { // 区块修改：二维差分
    X++, Y++;
    add(x, y, k), add(X, y, -k);
    add(X, Y, k), add(x, Y, -k);
}
int ask(auto &w, int x, int y) { // 单点查询
    int ans = 0;
    for (int i = x; i; i -= i & -i) {
        for (int j = y; j; j -= j & -j) {
            ans += w[i][j];
        }
    }
    return ans;
}
int ask(int x, int y) { // 多了一步计算
    int ans = 0;
    ans += x * y * ask(b1, x, y);
    ans -= y * ask(b2, x, y);
    ans -= x * ask(b3, x, y);
    ans += ask(b4, x, y);
    return ans;
}
int ask(int x, int y, int X, int Y) { // 区块查询：二维前缀和
    x--, y--;
    return ask(X, Y) - ask(x, Y) - ask(X, y) + ask(x, y);
}
```

```
};
```

线段树

LazyInfoTag 线段树

```
template<typename Info, typename Tag>
struct LazySegmentTree {
    int n;
    std::vector<Info> info;
    std::vector<Tag> tag;
    LazySegmentTree() : n(0) {}
    LazySegmentTree(int n_, Info v_ = Info()) {
        init(n_, v_);
    }
    template<typename T>
    LazySegmentTree(std::vector<T> init_) {
        init(init_);
    }
    void init(int n_, Info v_ = Info()) {
        init(std::vector(n_, v_));
    }
    template<typename T>
    void init(std::vector<T> init_) {
        n = init_.size();
        info.assign(4 << std::__lg(n), Info());
        tag.assign(4 << std::__lg(n), Tag());
        std::function<void(int, int, int)> build = [&](int p, int l, int r) {
            if (r - l == 1) {
                info[p] = init_[l];
                return;
            }
            int m = (l + r) / 2;
```

```
            build(2 * p, l, m);
            build(2 * p + 1, m, r);
            pull(p);
            };
        build(1, 0, n);
    }
    void pull(int p) {
        info[p] = info[2 * p] + info[2 * p + 1];
    }
    void apply(int p, const Tag& v) {
        info[p].apply(v);
        tag[p].apply(v);
    }
    void push(int p) {
        apply(2 * p, tag[p]);
        apply(2 * p + 1, tag[p]);
        tag[p] = Tag();
    }
    void modify(int p, int l, int r, int x, const Info& v) {
        if (r - l == 1) {
            info[p] = v;
            return;
        }
        int m = (l + r) / 2;
        push(p);
        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);
        return rangeQuery(2 * p, l, m, x, y) + rangeQuery(2 * p + 1, m, r, x, y);
    }
    Info rangeQuery(int l, int r) {
        return rangeQuery(1, 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, v);
            return;
        }
        int m = (l + r) / 2;
        push(p);
        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);
```

```
}
template<typename F>
int findFirst(int p, int l, int r, int x, int y, F pred) {
    if (l >= y || r <= x || !pred(info[p])) {
        return -1;
    }
    if (r - l == 1) {
        return l;
    }
    int m = (l + r) / 2;
    push(p);
    int res = findFirst(2 * p, l, m, x, y, pred);
    if (res == -1) {
        res = findFirst(2 * p + 1, m, r, x, y, pred);
    }
    return res;
}
template<typename F>
int findFirst(int l, int r, F pred) {
    return findFirst(1, 0, n, l, r, pred);
}
template<typename F>
int findLast(int p, int l, int r, int x, int y, F pred) {
    if (l >= y || r <= x || !pred(info[p])) {
        return -1;
    }
    if (r - l == 1) {
        return l;
    }
    int m = (l + r) / 2;
    push(p);
    int res = findLast(2 * p + 1, m, r, x, y, pred);
    if (res == -1) {
```

```
            res = findLast(2 * p, l, m, x, y, pred);
        }
        return res;
    }
    template<typename F>
    int findLast(int l, int r, F pred) {
        return findLast(1, 0, n, l, r, pred);
    }
};

struct Tag {
    i64 x = 0;
    void apply(Tag t) {
    }
};

struct Info {
    i64 x = 0;
    void apply(Tag t) {
    }
};
Info operator+(Info a, Info b) {
    return { a.x + b.x };
}
```

快速线段树 (单点修改 + 区间最值)

```
struct Segt {
    vector<int> w;
    int n;
    Segt(int n) : w(2 * n, (int)-2E9), n(n) {}

    void modify(int pos, int val) {
```

```
        for (w[pos += n] = val; pos > 1; pos /= 2) {
            w[pos / 2] = max(w[pos], w[pos ^ 1]);
        }
    }

    int ask(int l, int r) {
        int res = -2E9;
        for (l += n, r += n; l < r; l /= 2, r /= 2) {
            if (l % 2) res = max(res, w[l++]);
            if (r % 2) res = max(res, w[--r]);
        }
        return res;
    }
};
```

区间取模 原题需要进行“单点赋值 + 区间取模 + 区间求和”See . 该操作不需要懒标记.

需要额外维护一个区间最大值, 当模数大于区间最大值时剪枝, 否则进行单点取模. 由于单点 MOD $<$ x 时 x mod MOD $<$ $\frac{x}{2}$, 故单点取模至 0 最劣只需要 $\log x$ 次.

拆位运算 原题同上. 使用若干棵线段树维护每一位的值, 区间异或转变为区间翻转.

树套树

线段树套平衡树

```
#include <bits/stdc++.h>
#include <ext/pb_ds/assoc_container.hpp>
using namespace __gnu_pbds;
using pii = std::pair<int, int>;
```

```
const int inf = 2147483647;

tree<pii, null_type, std::less<pii>, rb_tree_tag, tree_order_statistics_node_update> ve

template<typename Info>
struct SegmentTree {
    int n;
    std::vector<Info> info;

    SegmentTree() : n(0) {}

    void init(int n_) {
        n = n_;
        info.assign(4 << std::__lg(n), Info());
    }

public:
    // --- 初始化与构建 ---
    void build(const std::vector<int>& a) {
        _build(1, 0, n, a);
    }

private:
    void _build(int p, int l, int r, const std::vector<int>& a) {
        // 从叶子节点开始，向上启发式合并来构建整棵树
        if (r - l == 1) {
            if (l < a.size()) info[p].ver.insert({ a[l], l });
            return;
        }
        int m = (l + r) / 2;
        _build(2 * p, l, m, a);
        _build(2 * p + 1, m, r, a);
```

```
        // 启发式合并（小树合并到大树）
        if (info[2 * p].ver.size() > info[2 * p + 1].ver.size()) {
            info[p].ver = info[2 * p].ver;
            for (const auto& item : info[2 * p + 1].ver) info[p].ver.insert(item);
        }
        else {
            info[p].ver = info[2 * p + 1].ver;
            for (const auto& item : info[2 * p].ver) info[p].ver.insert(item);
        }
    }

public:
    // 单点修改
    void update(int pos, int old_val, int new_val) {
        _update(1, 0, n, pos, old_val, new_val);
    }

    // 查询排名（返回比 k 小的数的个数）
    int query_rank_count(int l, int r, int k) {
        return _query_rank_count(1, 0, n, l, r, k);
    }

    // 查询前驱
    int query_pred(int l, int r, int k) {
        return _query_pred(1, 0, n, l, r, k);
    }

    // 查询后继
    int query_succ(int l, int r, int k) {
        return _query_succ(1, 0, n, l, r, k);
    }

private:
```

```
    void _update(int p, int l, int r, int pos, int old_val, int new_val) {
        info[p].ver.erase({ old_val, pos });
        info[p].ver.insert({ new_val, pos });
        if (r - l == 1) return;
        int m = (l + r) / 2;
        if (pos < m) _update(2 * p, l, m, pos, old_val, new_val);
        else _update(2 * p + 1, m, r, pos, old_val, new_val);
    }

    int _query_rank_count(int p, int l, int r, int x, int y, int k) {
        if (l >= y || r <= x) return 0;
        if (l >= x && r <= y) {
            return info[p].ver.order_of_key({ k, -1 });
        }
        int m = (l + r) / 2;
        return _query_rank_count(2 * p, l, m, x, y, k) + _query_rank_count(2 * p + 1, m
    }

    int _query_pred(int p, int l, int r, int x, int y, int k) {
        if (l >= y || r <= x) return -inf;
        if (l >= x && r <= y) {
            auto it = info[p].ver.lower_bound({ k, -1 });
            if (it == info[p].ver.begin()) return -inf;
            return (--it)->first;
        }
        int m = (l + r) / 2;
        return std::max(_query_pred(2 * p, l, m, x, y, k), _query_pred(2 * p + 1, m, r,
    }

    int _query_succ(int p, int l, int r, int x, int y, int k) {
        if (l >= y || r <= x) return inf;
        if (l >= x && r <= y) {
            auto it = info[p].ver.upper_bound({ k, inf });
```

```
            if (it == info[p].ver.end()) return inf;
            return it->first;
        }
        int m = (l + r) / 2;
        return std::min(_query_succ(2 * p, l, m, x, y, k), _query_succ(2 * p + 1, m, r,
    }
};

struct Info {
    tree<pii, null_type, std::less<pii>, rb_tree_tag, tree_order_statistics_node_update
};
```

小波矩阵树: 高效静态区间第 K 大查询

手写 bitset 压位, 以 $\mathcal{O}(N \log N)$ 的时间复杂度和 $\mathcal{O}(N + \frac{N \log N}{64})$ 的空间建树后, 实现单次 $\mathcal{O}(\log N)$ 复杂度的区间第 k 大值询问. 建议使用 0-idx 计数法, 但是经测试 1-idx 也有效, 但需要更多的检验.

```
#define __count(x) __builtin_popcountll(x)
struct Wavelet {
    vector<int> val, sum;
    vector<u64> bit;
    int t, n;

    int getSum(int i) {
        return sum[i >> 6] + __count(bit[i >> 6] & ((1ULL << (i & 63)) - 1));
    }

    Wavelet(vector<int> v) : val(v), n(v.size()) {
        sort(val.begin(), val.end());
        val.erase(unique(val.begin(), val.end()), val.end());

        int n_ = val.size();
```

```
        t = __lg(2 * n_ - 1);
        bit.resize((t * n + 64) >> 6);
        sum.resize(bit.size());
        vector<int> cnt(n_ + 1);

        for (int &x : v) {
            x = lower_bound(val.begin(), val.end(), x) - val.begin();
            cnt[x + 1]++;
        }
        for (int i = 1; i < n_; ++i) {
            cnt[i] += cnt[i - 1];
        }
        for (int j = 0; j < t; ++j) {
            for (int i : v) {
                int tmp = i >> (t - 1 - j);
                int pos = (tmp >> 1) << (t - j);
                auto setBit = [&](int i, u64 v) {
                    bit[i >> 6] |= (v << (i & 63));
                };
                setBit(j * n + cnt[pos], tmp & 1);
                cnt[pos]++;
            }
            for (int i : v) {
                cnt[(i >> (t - j)) << (t - j)]--;
            }
        }
        for (int i = 1; i < sum.size(); ++i) {
            sum[i] = sum[i - 1] + __count(bit[i - 1]);
        }
    }

    int small(int l, int r, int k) {
        r++;
```

```
        for (int j = 0, x = 0, y = n, res = 0;; ++j) {
            if (j == t) return val[res];
            int A = getSum(n * j + x), B = getSum(n * j + l);
            int C = getSum(n * j + r), D = getSum(n * j + y);
            int ab_zeros = r - l - C + B;
            if (ab_zeros > k) {
                res = res << 1;
                y -= D - A;
                l -= B - A;
                r -= C - A;
            } else {
                res = (res << 1) | 1;
                k -= ab_zeros;
                x += y - x - D + A;
                l += y - l - D + B;
                r += y - r - D + C;
            }
        }
    }
    int large(int l, int r, int k) {
        return small(l, r, r - l - k);
    }
};
```

主席树 (可持久化线段树)

以 $\mathcal{O}(N \log N)$ 的时间复杂度建树, 查询, 修改.

```
struct PreesidentTree {
    static constexpr int N = 2e5 + 10;
    int cntNodes, root[N];

    struct node {
```

```
        int l, r;
        int cnt;
    }tr[4 * N + 17 * N];

    //u 是新节点,v 是旧节点
    void modify(int& u, int v, int l, int r, int x) {
        u = ++cntNodes;
        tr[u] = tr[v];
        tr[u].cnt++;
        if (l == r) return;
        int mid = (l + r) / 2;
        if (x <= mid) modify(tr[u].l, tr[v].l, l, mid, x);
        else modify(tr[u].r, tr[v].r, mid + 1, r, x);
    }

    //u 是新节点,v 是旧节点
    int kth(int u, int v, int l, int r, int k) {
        if (l == r) return l;
        int res = tr[tr[u].l].cnt - tr[tr[v].l].cnt;
        int mid = (l + r) / 2;
        if (k <= res) return kth(tr[u].l, tr[v].l, l, mid, k);
        else return kth(tr[u].r, tr[v].r, mid + 1, r, k - res);
    }
};
```

普通莫队

以 $\mathcal{O}(N\sqrt{N})$ 的复杂度完成 Q 次询问的离线查询, 其中每个分块的大小取 $\sqrt{N} = \sqrt{10^5} = 317$, 也可以使用 `n / min<int>(n, sqrt(q))` , `ceil((double)n / (int)sqrt(n))` 或者 `sqrt(n)` 划分.

```
signed main() {
    int n;
```

```
    cin >> n;
    vector<int> w(n + 1);
    for (int i = 1; i <= n; i++) {
        cin >> w[i];
    }

    int q;
    cin >> q;
    vector<array<int, 3>> query(q + 1);
    for (int i = 1; i <= q; i++) {
        int l, r;
        cin >> l >> r;
        query[i] = {l, r, i};
    }

    int Knum = n / min<int>(n, sqrt(q)); // 计算块长
    vector<int> K(n + 1);
    for (int i = 1; i <= n; i++) { // 固定块长
        K[i] = (i - 1) / Knum + 1;
    }
    sort(query.begin() + 1, query.end(), [&](auto x, auto y) {
        if (K[x[0]] != K[y[0]]) return x[0] < y[0];
        if (K[x[0]] & 1) return x[1] < y[1];
        return x[1] > y[1];
    });

    int l = 1, r = 0, val = 0;
    vector<int> ans(q + 1);
    for (int i = 1; i <= q; i++) {
        auto [ql, qr, id] = query[i];
        auto add = [&](int x) -> void {};
        auto del = [&](int x) -> void {};
        while (l > ql) add(w[--l]);
```

```
        while (r < qr) add(w[++r]);
        while (l < ql) del(w[l++]);
        while (r > qr) del(w[r--]);
        ans[id] = val;
    }
    for (int i = 1; i <= q; i++) {
        cout << ans[i] << endl;
    }
}
```

需要注意的是, 在普通莫队中,K 数组的作用是根据左边界的值进行排序, 当询问次数很少时 ($q \ll n$), 可以直接合并到 query 数组中.

带修改的莫队 (带时间维度的莫队)

以 $\mathcal{O}(N^{\frac{5}{3}})$ 的复杂度完成 Q 次询问的离线查询, 其中每个分块的大小取 $N^{\frac{2}{3}} = \sqrt[3]{100000^2} = 2154$ (直接取会略快), 也可以使用 pow(n, 0.6666) 划分.

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

    std::vector<a4> q{ {} };        // {左区间, 右区间, 累计修改次数, 下标}
    std::vector<a2> upd{ {} };      // {修改位置, 修改的值}
    for (int i = 1;i <= m;i++) {
        char op;    std::cin >> op;
        if (op == 'Q') {
            int l, r;   std::cin >> l >> r;
            q.push_back(a4{ l,r,(int)upd.size() - 1 ,(int)q.size() });
        }
        else {
            int idx, val;   std::cin >> idx >> val;
            upd.push_back({ idx,val });
```

```
    }
}

int block = 2610;   //n ^ (2 / 3)
std::vector<int> b(n + 1);
for (int i = 1;i <= n;i++) b[i] = (i - 1) / block + 1;
std::sort(q.begin() + 1, q.end(), [&](auto x, auto y) {
    if (b[x[0]] != b[y[0]]) return x[0] < y[0];
    if (b[x[1]] != b[y[1]]) return x[1] < y[1];
    return x[3] < y[3];
    });

n = q.size() - 1;
int l = 1, r = 0, t = 0;
std::vector<int> ans(n + 1);
for (int i = 1;i <= n;i++) {
    auto [ql, qr, qt, id] = q[i];

    auto add = [&](int x) {};
    auto del = [&](int x) {};
    auto time = [&](int t, int l, int r) {
        int pos = upd[t][0];
        int& val = upd[t][1];
        if (pos >= l && pos <= r) {
            del(a[pos]);
            add(val);
        }
        std::swap(a[pos], val);
        };

    while (l > ql) add(a[--l]);
    while (r < qr) add(a[++r]);
    while (l < ql) del(a[l++]);
```

```
        while (r > qr) del(a[r--]);
        while (t < qt) time(++t, ql, qr);
        while (t > qt) time(t--, ql, qr);

        ans[id] = cnt;
    }
    for (int i = 1;i <= n;i++)    std::cout << ans[i] << '\n';
```

回滚莫队

```
    std::vector<a3> q(m + 1);
    for (int i = 1;i <= m;i++) {
        int l, r;   std::cin >> l >> r;
        q[i] = { l,r,i };
    }
    int block = n / std::min<int>(n, sqrt(m));
    std::vector<int> b(n + 1);
    for (int i = 1;i <= n;i++) b[i] = (i - 1) / block + 1;
    std::sort(q.begin() + 1, q.end(), [&](auto x, auto y) {
        if (b[x[0]] != b[y[0]]) return x[0] < y[0];
        return x[1] < y[1];
        });

    int l = 1, r = 0, cur_block = 0, tmpl;
    int res = 0;
    std::vector<i64> ans(m + 1);
    for (int i = 1;i <= m;i++) {
        auto [ql, qr, id] = q[i];

        if (b[ql] == b[qr]) {
            //暴力
            for (int j = ql;j <= qr;j++);
            //遍历答案
```

```
            for (int j = ql;j <= qr;j++);
            //撤销
            for (int j = ql;j <= qr;j++);
            continue;
        }

        auto add = [&](int x, i64& res) {};
        auto del = [&](int x) {};

        //若当前更新到了一个新的块
        if (b[ql] != cur_block) {
            while (r > b[ql] * block) del(w[r--]);
            while (l < b[ql] * block + 1) del(w[l++]);
            res = 0;
            cur_block = b[ql];
        }
        //先移动右指针
        while (r < qr) add(w[++r], res);
        tmpl = l;
        i64 tmpres = res;
        //查询答案
        while (tmpl > ql) add(w[--tmpl], tmpres);
        ans[id] = tmpres;
        //回滚
        while (tmpl < l) del(w[tmpl++]);
    }

    for (int i = 1;i <= m;i++)   std::cout << ans[i] << '\n';
```

对顶堆

```
namespace Set {
    const int kInf = 1e9 + 2077;
```

```
std::multiset<int> less, greater;
void init() {
    less.clear(), greater.clear();
    less.insert(-kInf), greater.insert(kInf);
}
void adjust() {
    while (less.size() > greater.size() + 1) {
        std::multiset<int>::iterator it = (--less.end());
        greater.insert(*it);
        less.erase(it);
    }
    while (greater.size() > less.size()) {
        std::multiset<int>::iterator it = greater.begin();
        less.insert(*it);
        greater.erase(it);
    }
}
void add(int val_) {
    if (val_ <= *greater.begin()) less.insert(val_);
    else greater.insert(val_);
    adjust();
}
void del(int val_) {
    std::multiset<int>::iterator it = less.lower_bound(val_);
    if (it != less.end()) {
        less.erase(it);
    }
    else {
        it = greater.lower_bound(val_);
        greater.erase(it);
    }
    adjust();
}
```

```
    int get_middle() {
        return *less.rbegin();
    }
}
```

KD Tree

在第 k 维上的单次查询复杂度最坏为 $\mathcal{O}(n^{1-k^{-1}})$.

```
struct KDT {
    constexpr static int N = 1e5 + 10, K = 2;
    double alpha = 0.725;
    struct node {
        int info[K];
        int mn[K], mx[K];
    } tr[N];
    int ls[N], rs[N], siz[N], id[N], d[N];
    int idx, rt, cur;
    int ans;
    KDT() {
        rt = 0;
        cur = 0;
        memset(ls, 0, sizeof ls);
        memset(rs, 0, sizeof rs);
        memset(d, 0, sizeof d);
    }
    void apply(int p, int son) {
        if (son) {
            for (int i = 0; i < K; i++) {
                tr[p].mn[i] = min(tr[p].mn[i], tr[son].mn[i]);
                tr[p].mx[i] = max(tr[p].mx[i], tr[son].mx[i]);
            }
            siz[p] += siz[son];
```

```
        }
    }
    void maintain(int p) {
        for (int i = 0; i < K; i++) {
            tr[p].mn[i] = tr[p].info[i];
            tr[p].mx[i] = tr[p].info[i];
        }
        siz[p] = 1;
        apply(p, ls[p]);
        apply(p, rs[p]);
    }
    int build(int l, int r) {
        if (l > r) return 0;
        vector<double> avg(K);
        for (int i = 0; i < K; i++) {
            for (int j = l; j <= r; j++) {
                avg[i] += tr[id[j]].info[i];
            }
            avg[i] /= (r - l + 1);
        }
        vector<double> var(K);
        for (int i = 0; i < K; i++) {
            for (int j = l; j <= r; j++) {
                var[i] += (tr[id[j]].info[i] - avg[i]) * (tr[id[j]].info[i] - avg[i]);
            }
        }
        int mid = (l + r) / 2;
        int x = max_element(var.begin(), var.end()) - var.begin();
        nth_element(id + l, id + mid, id + r + 1, [&](int a, int b) {
            return tr[a].info[x] < tr[b].info[x];
        });
        d[id[mid]] = x;
        ls[id[mid]] = build(l, mid - 1);
```

```
        rs[id[mid]] = build(mid + 1, r);
        maintain(id[mid]);
        return id[mid];
    }
    void print(int p) {
        if (!p) return;
        print(ls[p]);
        id[++idx] = p;
        print(rs[p]);
    }
    void rebuild(int &p) {
        idx = 0;
        print(p);
        p = build(1, idx);
    }
    bool bad(int p) {
        return alpha * siz[p] <= max(siz[ls[p]], siz[rs[p]]);
    }
    void insert(int &p, int cur) {
        if (!p) {
            p = cur;
            maintain(p);
            return;
        }
        if (tr[p].info[d[p]] > tr[cur].info[d[p]]) insert(ls[p], cur);
        else insert(rs[p], cur);
        maintain(p);
        if (bad(p)) rebuild(p);
    }
    void insert(vector<int> &a) {
        cur++;
        for (int i = 0; i < K; i++) {
            tr[cur].info[i] = a[i];
```

```
        }
        insert(rt, cur);
    }
    bool out(int p, vector<int> &a) {
        for (int i = 0; i < K; i++) {
            if (a[i] < tr[p].mn[i]) {
                return true;
            }
        }
        return false;
    }
    bool in(int p, vector<int> &a) {
        for (int i = 0; i < K; i++) {
            if (a[i] < tr[p].info[i]) {
                return false;
            }
        }
        return true;
    }
    bool all(int p, vector<int> &a) {
        for (int i = 0; i < K; i++) {
            if (a[i] < tr[p].mx[i]) {
                return false;
            }
        }
        return true;
    }
    void query(int p, vector<int> &a) {
        if (!p) return;
        if (out(p, a)) return;
        if (all(p, a)) {
            ans += siz[p];
            return;
```

```
        }
        if (in(p, a)) ans++;
        query(ls[p], a);
        query(rs[p], a);
    }
    int query(vector<int> &a) {
        ans = 0;
        query(rt, a);
        return ans;
    }
};
```

/END/

数据结构 B

基于状压的线性 RMQ 算法

严格 $\mathcal{O}(N)$ 预处理,$\mathcal{O}(1)$ 查询.

```
template<typename T, typename Cmp = less<T>> struct RMQ {
    const Cmp cmp = Cmp();
    static constexpr unsigned B = 64;
    using u64 = unsigned long long;
    int n;
    vector<vector<T>> a;
    vector<T> pre, suf, ini;
    vector<u64> stk;
    RMQ() {}
    RMQ(const vector<T> &v) {
        init(v);
    }
    void init(const vector<T> &v) {
        n = v.size();
```

```
    pre = suf = ini = v;
    stk.resize(n);
    if (!n) {
        return;
    }
    const int M = (n - 1) / B + 1;
    const int lg = __lg(M);
    a.assign(lg + 1, vector<T>(M));
    for (int i = 0; i < M; i++) {
        a[0][i] = v[i * B];
        for (int j = 1; j < B && i * B + j < n; j++) {
            a[0][i] = min(a[0][i], v[i * B + j], cmp);
        }
    }
    for (int i = 1; i < n; i++) {
        if (i % B) {
            pre[i] = min(pre[i], pre[i - 1], cmp);
        }
    }
    for (int i = n - 2; i >= 0; i--) {
        if (i % B != B - 1) {
            suf[i] = min(suf[i], suf[i + 1], cmp);
        }
    }
    for (int j = 0; j < lg; j++) {
        for (int i = 0; i + (2 << j) <= M; i++) {
            a[j + 1][i] = min(a[j][i], a[j][i + (1 << j)], cmp);
        }
    }
    for (int i = 0; i < M; i++) {
        const int l = i * B;
        const int r = min(1U * n, l + B);
        u64 s = 0;
```

```
            for (int j = l; j < r; j++) {
                while (s && cmp(v[j], v[__lg(s) + l])) {
                    s ^= 1ULL << __lg(s);
                }
                s |= 1ULL << (j - l);
                stk[j] = s;
            }
        }
    }
    T operator()(int l, int r) {
        if (l / B != (r - 1) / B) {
            T ans = min(suf[l], pre[r - 1], cmp);
            l = l / B + 1;
            r = r / B;
            if (l < r) {
                int k = __lg(r - l);
                ans = min({ans, a[k][l], a[k][r - (1 << k)]}, cmp);
            }
            return ans;
        } else {
            int x = B * (l / B);
            return ini[__builtin_ctzll(stk[r - 1] >> (l - x)) + l];
        }
    }
};
```

珂朵莉树 (OD Tree)

区间赋值的数据结构都可以骗分, 在数据随机的情况下, 复杂度可以保证, 时间复杂度:$\mathcal{O}(N \log \log N)$.

```
struct ODT {
    struct node {
```

```
        int l, r;
        mutable LL v;
        node(int l, int r = -1, LL v = 0) : l(l), r(r), v(v) {}
        bool operator<(const node &o) const {
            return l < o.l;
        }
    };
    set<node> s;
    ODT() {
        s.clear();
    }
    auto split(int pos) {
        auto it = s.lower_bound(node(pos));
        if (it != s.end() && it->l == pos) return it;
        it--;
        int l = it->l, r = it->r;
        LL v = it->v;
        s.erase(it);
        s.insert(node(l, pos - 1, v));
        return s.insert(node(pos, r, v)).first;
    }
    void assign(int l, int r, LL x) {
        auto itr = split(r + 1), itl = split(l);
        s.erase(itl, itr);
        s.insert(node(l, r, x));
    }
    void add(int l, int r, LL x) {
        auto itr = split(r + 1), itl = split(l);
        for (auto it = itl; it != itr; it++) {
            it->v += x;
        }
    }
    LL kth(int l, int r, int k) {
```

```
        vector<pair<LL, int>> a;
        auto itr = split(r + 1), itl = split(l);
        for (auto it = itl; it != itr; it++) {
            a.push_back(pair<LL, int>(it->v, it->r - it->l + 1));
        }
        sort(a.begin(), a.end());
        for (auto [val, len] : a) {
            k -= len;
            if (k <= 0) return val;
        }
    }
    LL power(LL a, int b, int mod) {
        a %= mod;
        LL res = 1;
        for (; b; b /= 2, a = a * a % mod) {
            if (b % 2) {
                res = res * a % mod;
            }
        }
        return res;
    }
    LL powersum(int l, int r, int x, int mod) {
        auto itr = split(r + 1), itl = split(l);
        LL ans = 0;
        for (auto it = itl; it != itr; it++) {
            ans = (ans + power(it->v, x, mod) * (it->r - it->l + 1) % mod) % mod;
        }
        return ans;
    }
};
```

pbds 扩展库实现平衡二叉树

记得加上相应的头文件, 同时需要注意定义时的参数, 一般只需要修改第三个参数: 即定义的是大根堆还是小根堆.

附常见成员函数:

```
empty() / size()
insert(x) // 插入元素 x
erase(x) // 删除元素/迭代器 x
order_of_key(x) // 返回元素 x 的排名
find_by_order(x) // 返回排名为 x 的元素迭代器
lower_bound(x) / upper_bound(x) // 返回迭代器
join(Tree) // 将 Tree 树的全部元素并入当前的树
split(x, Tree) // 将大于 x 的元素放入 Tree 树
```

```cpp
#include <ext/pb_ds/assoc_container.hpp>
using namespace __gnu_pbds;
using V = pair<int, int>;
tree<V, null_type, less<V>, rb_tree_tag, tree_order_statistics_node_update> ver;
map<int, int> dic;

int n; cin >> n;
for (int i = 1, op, x; i <= n; i++) {
    cin >> op >> x;
    if (op == 1) { // 插入一个元素 x, 允许重复
        ver.insert({x, ++dic[x]});
    } else if (op == 2) { // 删除元素 x, 若有重复, 则任意删除一个
        ver.erase({x, dic[x]--});
    } else if (op == 3) { // 查询元素 x 的排名 (排名定义为比当前数小的数的个数 +
        cout << ver.order_of_key({x, 1}) + 1 << endl;
    } else if (op == 4) { // 查询排名为 x 的元素
        cout << ver.find_by_order(--x)->first << endl;
    } else if (op == 5) { // 查询元素 x 的前驱
        int idx = ver.order_of_key({x, 1}) - 1; // 无论 x 存不存在,idx 都代表 x 的位
```

```
        cout << ver.find_by_order(idx)->first << endl;
    } else if (op == 6) { // 查询元素 x 的后继
        int idx = ver.order_of_key( {x, dic[x]}); // 如果 x 不存在，那么 idx 就是 x
        if (ver.find({x, 1}) != ver.end()) idx++; // 如果 x 存在，那么 idx 是 x 的位
        cout << ver.find_by_order(idx)->first << endl;
    }
}
```

vector 模拟实现平衡二叉树

```
#define ALL(x) x.begin(), x.end()
#define pre lower_bound
#define suf upper_bound
int n; cin >> n;
vector<int> ver;
for (int i = 1, op, x; i <= n; i++) {
    cin >> op >> x;
    if (op == 1) ver.insert(pre(ALL(ver), x), x);
    if (op == 2) ver.erase(pre(ALL(ver), x));
    if (op == 3) cout << pre(ALL(ver), x) - ver.begin() + 1 << endl;
    if (op == 4) cout << ver[x - 1] << endl;
    if (op == 5) cout << ver[pre(ALL(ver), x) - ver.begin() - 1] << endl;
    if (op == 6) cout << ver[suf(ALL(ver), x) - ver.begin()] << endl;
}
```

常见结论

题意:(区间移位问题) 要求将整个序列左移/右移若干个位置，例如，原序列为 $A = (a_1, a_2, \dots, a_n)$，右移 x 位后变为 $A = (a_{x+1}, a_{x+2}, \dots, a_n, a_1, a_2, \dots, a_x)$.

区间的端点只是一个数字, 即使被改变了, 通过一定的转换也能够还原, 所以我们可以 $\mathcal{O}(1)$ 解决这一问题. 为了方便计算, 我们规定下标从 0 开始, 即整

个线段的区间为 $[0,n)$, 随后, 使用一个偏移量 `shift` 记录. 使用 `shift = (shift + x) % n;` 更新偏移量; 此后的区间查询/修改前, 再将坐标偏移回去即可, 下方代码使用区间修改作为示例.

```
cin >> l >> r >> x;
l--; // 坐标修改为 0 开始
r--;
l = (l + shift) % n; // 偏移
r = (r + shift) % n;
if (l > r) { // 区间分离则分别操作
    segt.modify(l, n - 1, x);
    segt.modify(0, r, x);
} else {
    segt.modify(l, r, x);
}
```

常见例题

题意:(带修莫队 - 维护队列) 要求能够处理以下操作:

- `'Q' l r` : 询问区间 $[l,r]$ 有几个颜色;
- `'R' idx w` : 将下标 `idx` 的颜色修改为 `w` .

输入格式为: 第一行 n 和 q $(1 \leq n,q \leq 133333)$ 分别代表区间长度和操作数量; 第二行 n 个整数 $a_1,a_2 \ldots,a_n$ $(1 \leq a_i \leq 10^6)$ 代表初始颜色; 随后 q 行为具体操作.

```
const int N = 1e6 + 7;
signed main() {
    int n, q;
    cin >> n >> q;
    vector<int> w(n + 1);
    for (int i = 1; i <= n; i++) {
        cin >> w[i];
    }
```

```
vector<array<int, 4>> query = {{}}; // {左区间，右区间，累计修改次数，下标}
vector<array<int, 2>> modify = {{}}; // {修改的值，修改的元素下标}
for (int i = 1; i <= q; i++) {
    char op;
    cin >> op;
    if (op == 'Q') {
        int l, r;
        cin >> l >> r;
        query.push_back({l, r, (int)modify.size() - 1, (int)query.size()});
    } else {
        int idx, w;
        cin >> idx >> w;
        modify.push_back({w, idx});
    }
}

int Knum = 2154; // 计算块长
vector<int> K(n + 1);
for (int i = 1; i <= n; i++) { // 固定块长
    K[i] = (i - 1) / Knum + 1;
}
sort(query.begin() + 1, query.end(), [&](auto x, auto y) {
    if (K[x[0]] != K[y[0]]) return x[0] < y[0];
    if (K[x[1]] != K[y[1]]) return x[1] < y[1];
    return x[3] < y[3];
});

int l = 1, r = 0, val = 0;
int t = 0; // 累计修改次数
vector<int> ans(query.size()), cnt(N);
for (int i = 1; i < query.size(); i++) {
    auto [ql, qr, qt, id] = query[i];
```

```
        auto add = [&](int x) -> void {
            if (cnt[x] == 0) ++ val;
            ++ cnt[x];
        };
        auto del = [&](int x) -> void {
            -- cnt[x];
            if (cnt[x] == 0) -- val;
        };
        auto time = [&](int x, int l, int r) -> void {
            if (l <= modify[x][1] && modify[x][1] <= r) { //当修改的位置在询问期间内
                del(w[modify[x][1]]);
                add(modify[x][0]);
            }
            swap(w[modify[x][1]], modify[x][0]); //直接交换修改数组的值与原始值，减
        };
        while (l > ql) add(w[--l]);
        while (r < qr) add(w[++r]);
        while (l < ql) del(w[l++]);
        while (r > qr) del(w[r--]);
        while (t < qt) time(++t, ql, qr);
        while (t > qt) time(t--, ql, qr);
        ans[id] = val;
    }
    for (int i = 1; i < ans.size(); i++) {
        cout << ans[i] << endl;
    }
}
```

/END/

STL 与库函数

数组打乱 shuffle

```
mt19937_64 rng(chrono::steady_clock::now().time_since_epoch().count());
shuffle(ver.begin(), ver.end(), rng);
```

bit 库与位运算函数 __builtin_

```
__builtin_popcount(x) // 返回 x 二进制下含 1 的数量，例如 x=15=(1111) 时答案为 4
__builtin_ffs(x) // 返回 x 右数第一个 1 的位置 (1-idx),1(1) 返回 1,8(1000) 返回 4,
__builtin_ctz(x) // 返回 x 二进制下后导 0 的个数,1(1) 返回 0,8(1000) 返回 3
bit_width(x) // 返回 x 二进制下的位数,9(1001) 返回 4,26(11010) 返回 5
```

注: 以上函数的 long long 版本只需要在函数后面加上 ll 即可 (例如 __builtin_popcountll(x)), unsigned long long 加上 ull .

数字转字符串函数

itoa 虽然能将整数转换成任意进制的字符串, 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.

```
// to_string 函数会直接将你的各种类型的数字转换为字符串.
// string to_string(T val);
double val = 12.12;
cout << to_string(val);
// 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
// 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.
```

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.
// atoi 直接使用, 空字符返回 0, 允许正负符号, 数字字符前有其他字符返回 0, 数字
cout << atoi("12") << endl;
cout << atoi("   12") << endl; /*12*/
cout << atoi("-12abc") << endl; /*-12*/
cout << atoi("abc12") << endl; /*0*/

// 长整型函数名 atoll, 最高支持到 long long 型上限 2^63.
```

xxxxxxxxxx2 1p=(a+b+c)/2;2sum=sqrt(p(p-a)(p-b)*(p-c));cpp

在提及这个函数时, 我们先需要补充几点字典序相关的知识.

> 对于三个字符所组成的序列 {a,b,c}, 其按照字典序的 6 种排列分别为: {abc},{acb},{bac},{bca},{cab},{cba} 其排序原理是: 先固定 a (序列内最小元素), 再对之后的元素排列. 而 b < c , 所

以 abc < acb . 同理, 先固定 b (序列内次小元素), 再对之后的元素排列. 即可得出以上序列.

next_permutation 算法, 即是按照**字典序顺序**输出的全排列; 相对应的, prev_permutation 则是按照**逆字典序顺序**输出的全排列. 可以是数字, 亦可以是其他类型元素. 其直接在序列上进行更新, 故直接输出序列即可.

```
int n;
cin >> n;
vector<int> a(n);
// iota(a.begin(), a.end(), 1);
for (auto &it : a) cin >> it;
sort(a.begin(), a.end());

do {
    for (auto it : a) cout << it << " ";
    cout << endl;
} while (next_permutation(a.begin(), a.end()));
```

字符串转换为数值函数 sto

可以快捷的将**一串字符串**转换为**指定进制的数字**.

使用方法

- stoi(字符串, 0, x 进制) : 将一串 x 进制的字符串转换为 int 型数字.

```cpp
void Solve() {
    cout << stoi("1010", 0, 2) << endl;
    cout << stoi("c", 0, 16) << endl;
    cout << stoi("0x3f3f3f3f", 0, 0) << endl;
    cout << stoi("10", 0, 8) << endl;
    cout << stoll("aaaaaaaaaaa", 0, 16) << endl;
}
```

```
C:\Users\26099\Desktop\万能头文件.exe
10
12
1061109567
8
11728124029610
```

- `stoll(字符串, 0, x 进制)`：将一串 `x` 进制的字符串转换为 `long long` 型数字.
- `stoull`, `stod`, `stold` 同理.

数值转换为字符串函数 to_string

允许将**各种数值类型**转换为字符串类型.

```cpp
//将数值 num 转换为字符串 s
string s = to_string(num);
```

判断非递减 is_sorted

```cpp
//a 数组 [start,end) 区间是否是非递减的，返回 bool 型
cout << is_sorted(a + start, a + end);
```

累加 accumulate

```cpp
//将 a 数组 [start,end) 区间的元素进行累加，并输出累加和 +x 的值
cout << accumulate(a + start, a + end, x);
```

迭代器 iterator

```
//构建一个 UUU 容器的正向迭代器，名字叫 it
UUU::iterator it;

vector<int>::iterator it; //创建一个正向迭代器,++ 操作时指向下一个
vector<int>::reverse_iterator it; //创建一个反向迭代器,++ 操作时指向上一个
```

特殊函数 next 和 prev 详解:

```
auto it = s.find(x); // 建立一个迭代器
prev(it) / next(it); // 默认返回迭代器 it 的前/后一个迭代器
prev(it, 2) / next(it, 2); // 可选参数可以控制返回前/后任意个迭代器

/* 以下是一些应用 */
auto pre = prev(s.lower_bound(x)); // 返回第一个 <x 的迭代器
int ed = *prev(S.end(), 1); // 返回最后一个元素
```

其他函数

`exp2(x)`：返回 2^x

`log2(x)`：返回 $\log_2(x)$

`gcd(x, y)` / `lcm(x, y)` ：以 log 的复杂度返回 $\gcd(|x|,|y|)$ 与 `lcm`$(|x|,|y|)$，且返回值符号也为正数.

容器与成员函数

优先队列 priority_queue

默认升序 (大根堆), 自定义排序需要重载 < .

```
//没有 clear 函数
priority_queue<int, vector<int>, greater<int> > p; //重定义为降序 (小根堆)
push(x); //向栈顶插入 x
top(); //获取栈顶元素
pop(); //弹出栈顶元素

//重载运算符注意, 符号相反!
struct Node {
```

```
    int x; string s;
    friend bool operator < (const Node &a, const Node &b) {
        if (a.x != b.x) return a.x > b.x;
        return a.s > b.s;
    }
};
```

bitset

将数据转换为二进制, 从高位到低位排序, 以 0 为最低位. 当位数相同时支持全部的位运算.

```
// 如果输入的是 01 字符串，可以直接使用">>" 读入
bitset<10> s;
cin >> s;

//使用只含 01 的字符串构造--bitset< 容器长度 >B (字符串)
string S; cin >> S;
bitset<32> B (S);

//使用整数构造 (两种方式)
int x; cin >> x;
bitset<32> B1 (x);
bitset<32> B2 = x;

// 构造时，尖括号里的数字不能是变量
int x; cin >> x;
bitset<x> ans; // 错误构造

[] //随机访问
set(x) //将第 x 位置 1,x 省略时默认全部位置 1
reset(x) //将第 x 位置 0,x 省略时默认全部位置 0
flip(x) //将第 x 位取反,x 省略时默认全部位取反
```

```
to_ullong() //重转换为 ULL 类型
to_string() //重转换为 ULL 类型
count() //返回 1 的个数
any() //判断是否至少有一个 1
none() //判断是否全为 0

_Find_fisrt() // 找到从低位到高位第一个 1 的位置
_Find_next(x) // 找到当前位置 x 的下一个 1 的位置，复杂度 O(n/w + count)

bitset<23> B1("11101001"), B2("11101000");
cout << (B1 ^ B2) << "\n";  //按位异或
cout << (B1 | B2) << "\n";  //按位或
cout << (B1 & B2) << "\n";  //按位与
cout << (B1 == B2) << "\n"; //比较是否相等
cout << B1 << " " << B2 << "\n"; //你可以直接使用 cout 输出
```

哈希系列 unordered

通常指代 unordered_map,unordered_set,unordered_multimap,unordered_multiset, 与原版相比不进行排序.

如果将不支持哈希的类型作为 key 值代入, 编译器就无法正常运行, 这时需要我们为其手写哈希函数. 而我们写的这个哈希函数的正确性其实并不是特别重要 (但是不可以没有), 当发生冲突时编译器会调用 key 的 operator == 函数进行进一步判断.参考

对 pair,tuple 定义哈希

```
struct hash_pair {
    template<typename T1, typename T2>
    size_t operator()(const pair<T1, T2> &p) const {
        return hash<T1>()(p.fi) ^ hash<T2>()(p.se);
    }
```

```
};
unordered_set<pair<int, int>, int, hash_pair> S;
unordered_map<tuple<int, int, int>, int, hash_pair> M;
```

对结构体定义哈希 需要两个条件, 一个是在结构体中重载等于号 (区别于非哈希容器需要重载小于号, 如上所述, 当冲突时编译器需要根据重载的等于号判断), 第二是写一个哈希函数. 注意 `hash<>()` 的尖括号中的类型匹配.

```
struct fff {
    string x, y;
    int z;
    friend bool operator == (const fff &a, const fff &b) {
        return a.x == b.x || a.y == b.y || a.z == b.z;
    }
};
struct hash_fff {
    size_t operator()(const fff &p) const {
        return hash<string>()(p.x) ^ hash<string>()(p.y) ^ hash<int>()(p.z);
    }
};
unordered_map<fff, int, hash_fff> mp;
```

对 vector 定义哈希 以下两个方法均可. 注意 `hash<>()` 的尖括号中的类型匹配.

```
struct hash_vector {
    size_t operator()(const vector<int> &p) const {
        size_t seed = 0;
        for (auto it : p) {
            seed ^= hash<int>()(it);
        }
        return seed;
    }
```

```
};
unordered_map<vector<int>, int, hash_vector> mp;

namespace std {
    template<> struct hash<vector<int>> {
        size_t operator()(const vector<int> &p) const {
            size_t seed = 0;
            for (int i : p) {
                seed ^= hash<int>()(i) + 0x9e3779b9 + (seed << 6) + (seed >> 2);
            }
            return seed;
        }
    };
}
unordered_set<vector<int> > S;
```

/END/

图论

常见概念

oriented graph: 有向图

bidirectional edges: 双向边

平面图: 若能将无向图 $G=(V,E)$ 画在平面上使得任意两条无重合顶点的边不相交, 则称 G 是平面图.

无向正权图上某一点的偏心距: 记为 $ecc(u)=\max\{dist(u,v)\}$, 即以这个点为源, 到其他点的**所有最短路的最大值**. 如下图 A 点,$ecc(A)$ 即为 12 .

图的直径: 定义为 $d=\max\{ecc(u)\}$, 即**最大的偏心距**, 亦可以简化为图中最远的一对点的距离.

图的中心: 定义为 $arg = \min\{ecc(u)\}$, 即**偏心距最小的点**. 如下图, 图的中心即为 B 点.

图的绝对中心: 可以定义在边上的图的中心.

图的半径: 图的半径不同于圆的半径, 其不等于直径的一半 (但对于绝对中心定义上的直径而言是一半). 定义为 $r = \min\{ecc(u)\}$, 即**中心的偏心距**. 计算方式: 使用全源最短路, 计算出所有点的偏心距, 再加以计算.

平面图性质

一, 定义 $G = (V, E)$ 是一个无向图.

1. **图 G 可嵌入平面:** 如果可以把图 G 的所有结点和边都画在平面上, 同时除断点外连线之间没有交点, 就称图 G 可嵌入平面. 画出的无边相交的 G 称 G 的平面嵌入.
2. **可平面化:** 如果图 G 可以嵌入平面, 就称图 G 可平面化.
3. **面:** G 中边所包含的区域称作一个面. 有界区域称为内部面, 无界区域称为外部面, 常记作 R_0, 包围面的长度最短的闭链称为该面的边界, 面 R 的边界的长度称为该面的度数, 记作 $\deg(R)$.
4. **面的度数计算:** 含有割边和桥的度数为 2, 其余为 1.

二, 性质

1. 性质 1. $K_1, K_2, K_3, K_4, K_5 - e$ 均为极大可平面图.
2. 性质 2. 极大平面图必是连通图.
3. 性质 3. 当图阶数 $n \geq 3$ 时, 有割点或者桥的平面图不是极大平面图.

三, 定理

1. **定理 1:** 图 G 可嵌入球面当且仅当图 G 可嵌入平面.
2. **定理 2:** G 中各面的度数之和等于图 G 边数的两倍. **证明:** 设 e 为图 G 的两个面的公共边, 再计算两个面的度数时候边数各提供 1, 当 e 不是公共边时候, 也就是 e 为桥或者割边时候提供度数为 2. 因此, 面的度数之和为边的两倍.

3. **定理 3:** 设 R 是图 G 的某个平面嵌入的一个内部面, 则存在图 G 的一个平面嵌入使 R 为外部面.
4. **定理 4:** 设图 G 是简单的可平面图, 如果 G 中任意两个不相邻的结点加边后所得到的为非可平面图. 则称 G 是极大可平面图, 极大可平面图的任何平面嵌入都称为极大平面图. 极大平面图必是连通图.
5. **定理 5:** 图 G 为 n 阶简单的连通的平面图,G 为极大平面图当且仅当 G 的每一个面的度数为 3. 定理说明: 结点数大于等于 3 的极大平面图的任何面都是由三角形组成.
6. **定理 6: 欧拉公式:** 设图 G 是有 n 个结点,m 条边和 r 个面的连通平面图, 则它们满足:

$$n - m + r = 2$$

四, 结论

1. **结论 1:** $K_1, K_2, K_3, K_4, K_5 - e$ (K_5 任意删去一条边) 均为极大可平面图, 它们的任何平面嵌入都是极大平面图; 当阶数等于 3 时候, 有割边或桥的平面图不可能是极大平面图.
2. **结论 2:** 无向完全图 K_5 和无向完全二部图 $K_{3,3}$ 都是极小非可平面图 (去掉一条边就成为可平面图).
3. **结论 3:** 一个图是可平面图, 那么它的子图也是可平面图; 一个图的子图是非可平面图, 那么图本身也是非可平面图.
4. **结论 4:** 同一个图的平面嵌入中, 外部面和内部面的度数可以不同.

五, 推论

1. **推论 1:** 设图 G 是有 n 个结点,m 条边的连通平面简单图, 其中 $n \geq 3$, 则有:

$$m \leq 3n - 6$$

证明: 由图 G 的面度数之和为边数的二倍, 即 $2m$. 又因为 G 是平面简单图每一个面的度数至少为 3, 则 $2m \geq 3r$, 由欧拉公式有:$m \leq 3n - 6$

2. **推论 2:** 设图 G 是有 n 个结点,m 条边的连通平面简单图, 其中 $n \geq 3$ 且没有长度为 3 的圈, 则有:

$$m \leq 2n - 4$$

证明: G 没有长度为 3 的圈也就没有度为 3 的面,G 的每一个面的度数至少为 4. 所以 $2m \geq 4r$, 由欧拉公式有:$m \leq 2n - 4$ 提示: 对于推论 1 和推论 2 我们可以用定理进行判定它不是平面图. **例 1:** 证明 K_5 和 $K_{3,3}$ 是非平面图. **证明:**

- 在 K_5 中,m 应该小于等于 $3n-6$, 即 $m \leq 9$. 而完全图 K_5 具有 10 条边. 所以是非平面图.
- 在 $K_{3,3}$ 中, 没有长度大于 3 的圈, 根据推论 2 可知,$m \leq 2n-4$, 也就是 $m \leq 8$, 而 $K_{3,3}$ 含有 9 条边, 所以是非平面图.

3. **推论 3:** 设 G 是连通的平面图, 且每个面的度数至少为 $l(l \geq 3)$, 则

$$m \leq \frac{l}{l-2}(n-2).$$

证明: 同理, 有 $2m \geq r \times l$, 根据欧拉公式化简得:

$$2m \geq l(m-n+2)$$

4. **推论 4:** 设 G 是平面图, 有 ω 个连通分支, n 个结点, m 条边, r 个面, 则公式

$$n - m + r = \omega + 1$$

成立.

5. **推论 5:** 设 G 是有 n 个结点,m 条边和 r 个面,ω 个连通分支的平面图, 且 G 的各个面的度数至少为 l, ($l \geq 4$), 则

$$m \leq \frac{(n-\omega-1)l}{l-2}.$$

证明: 证明过程与推论 3 类似, 用到推论 4 的结论.

6. **推论 6:** 设 G 是任意平面简单图, 则

$$\delta(G) \leq 5.$$

证明: 设 G 有 n 个顶点 m 条边. 若 $m \leq 6$, 结论显然成立; 若 $m > 6$, 假设 G 的每个顶点的度数 > 6, 则由推论 1, 有

$$6n \leq \sum d(v) = 2m \leq 2(3n-6) = 6n - 12$$

与定理矛盾, 故 $\delta(G) \leq 5$.

六, 判别定理

1. **极大平面图的判别定理:** $n(n \geq 3)$ 阶连通的简单平面图 G. 则以下四个条件等价:
   1. G 是极大平面图;
   2. G 中每个面的度数都是 3;
   3. G 中有 m 条边 r 个面, 则
   $$3r = 2m;$$
   4. 设 G 带有 n 个顶点, m 条边, r 个面则
   $$m = 3n - 6;$$

单源最短路径 (SSSP 问题)

(正权稀疏图) 动态数组存图 +Djikstra 算法 使用优先队列优化, 以 $\mathcal{O}(M \log N)$ 的复杂度计算.

(负权图)Bellman ford 算法 使用结构体存边 (该算法无需存图), 以 $\mathcal{O}(NM)$ 的复杂度计算, 注意, 当所求点的路径上存在负环时, 所求点的答案无法得到, 但是会比 INF 小 (因为负环之后到所求点之间的边权会将 `d[end]` 的值更新), 该性质可以用于判断路径上是否存在负环: 在 $N-1$ 轮后仍无法得到答案 (一般与 `INF`$/2$ 进行比较) 的点, 到达其的路径上存在负环.

下方代码例题: 求解从 1 到 n 号节点的, 最多经过 k 条边的最短距离.

```
const int N = 550, M = 1e5 + 7;
int n, m, k;
struct node { int x, y, w; } ver[M];
int d[N], backup[N];

void bf() {
    memset(d, 0x3f, sizeof d); d[1] = 0;
```

```
    for (int i = 1; i <= k; ++ i) {
        memcpy(backup, d, sizeof d);
        for (int j = 1; j <= m; ++ j) {
            int x = ver[j].x, y = ver[j].y, w = ver[j].w;
            d[y] = min(d[y], backup[x] + w);
        }
    }
}
int main() {
    cin >> n >> m >> k;
    for (int i = 1; i <= m; ++ i) {
        int x, y, w; cin >> x >> y >> w;
        ver[i] = {x, y, w};
    }
    bf();
    for (int i = 1; i <= n; ++ i) {
        if (d[i] > INF / 2) cout << "N" << endl;
        else cout << d[n] << endl;
    }
}
```

(负权图)SPFA 算法 以 $\mathcal{O}(KM)$ 的复杂度计算, 其中 K 虽然为常数, 但是可以通过特殊的构造退化成接近 N , 需要注意被卡.

```
const int N = 1e5 + 7, M = 1e6 + 7;
int n, m;
int ver[M], ne[M], h[N], edge[M], tot;
int d[N], v[N];

void add(int x, int y, int w) {
    ver[++ tot] = y, ne[tot] = h[x], h[x] = tot;
    edge[tot] = w;
}
```

```
void spfa() {
    ms(d, 0x3f); d[1] = 0;
    queue<int> q; q.push(1);
    v[1] = 1;
    while(!q.empty()) {
        int x = q.front(); q.pop(); v[x] = 0;
        for (int i = h[x]; i; i = ne[i]) {
            int y = ver[i];
            if(d[y] > d[x] + edge[i]) {
                d[y] = d[x] + edge[i];
                if(v[y] == 0) q.push(y), v[y] = 1;
            }
        }
    }
}
int main() {
    cin >> n >> m;
    for (int i = 1; i <= m; ++ i) {
        int x, y, w; cin >> x >> y >> w;
        add(x, y, w);
    }
    spfa();
    for (int i = 1; i <= n; ++ i) {
        if (d[i] == INF) cout << "N" << endl;
        else cout << d[n] << endl;
    }
}
```

多源汇最短路 (Floyd)

使用邻接矩阵存图, 可以处理负权边, 以 $\mathcal{O}(N^3)$ 的复杂度计算. **注意, 这里建立的是单向边, 计算双向边需要额外加边.**

```
void floyd() {
    for (int k = 1; k <= n; k ++)
        for (int i = 1; i <= n; i ++)
            for (int j = 1; j <= n; j ++)
                d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
}
```

平面图最短路 (对偶图)

对于矩阵图, 建立对偶图的过程如下 (注释部分为建立原图), 其中数据的给出顺序依次为: 各 $n(n+1)$ 个数字分别代表从左向右, 从上向下, 从右向左, 从下向上的边.

```
for (int i = 1; i <= n + 1; i++) {
    for (int j = 1, w; j <= n; j++) {
        cin >> w;
        int pre = Hash(i - 1, j), now = Hash(i, j);
        if (i == 1) {
            add(s, now, w);
        } else if (i == n + 1) {
            add(pre, t, w);
        } else {
            add(pre, now, w);
        }
        // flow.add(Hash(i, j), Hash(i, j + 1), w);
    }
}
for (int i = 1; i <= n; i++) {
    for (int j = 1, w; j <= n + 1; j++) {
        cin >> w;
        int now = Hash(i, j), net = Hash(i, j - 1);
        if (j == 1) {
            add(now, t, w);
```

```
        } else if (j == n + 1) {
            add(s, net, w);
        } else {
            add(now, net, w);
        }
        // flow.add(Hash(i, j), Hash(i + 1, j), w);
    }
}
for (int i = 1; i <= n + 1; i++) {
    for (int j = 1, w; j <= n; j++) {
        cin >> w;
        int now = Hash(i, j), net = Hash(i - 1, j);
        if (i == 1) {
            add(now, s, w);
        } else if (i == n + 1) {
            add(t, net, w);
        } else {
            add(now, net, w);
        }
        // flow.add(Hash(i, j), Hash(i, j - 1), w);
    }
}
for (int i = 1; i <= n; i++) {
    for (int j = 1, w; j <= n + 1; j++) {
        cin >> w;
        int pre = Hash(i, j - 1), now = Hash(i, j);
        if (j == 1) {
            add(t, now, w);
        } else if (j == n + 1) {
            add(pre, s, w);
        } else {
            add(pre, now, w);
        }
```

```
        // flow.add(Hash(i, j), Hash(i - 1, j), w);
    }
}
```

最小生成树 (MST 问题)

(稀疏图)Prim 算法 使用邻接矩阵存图, 以 $\mathcal{O}(N^2 + M)$ 的复杂度计算, 思想与 djikstra 基本一致.

```
const int N = 550, INF = 0x3f3f3f3f;
int n, m, g[N][N];
int d[N], v[N];
int prim() {
    ms(d, 0x3f); //这里的 d 表示到" 最小生成树集合的距离
    int ans = 0;
    for (int i = 0; i < n; ++ i) { //遍历 n 轮
        int t = -1;
        for (int j = 1; j <= n; ++ j)
            if (v[j] == 0 && (t == -1 || d[j] < d[t])) //如果这个点不在集合内且当前
                t = j;
        v[t] = 1; //将 t 加入" 最小生成树集合
        if (i && d[t] == INF) return INF; //如果发现不连通, 直接返回
        if (i) ans += d[t];
        for (int j = 1; j <= n; ++ j) d[j] = min(d[j], g[t][j]); //用 t 更新其他点到
    }
    return ans;
}
int main() {
    ms(g, 0x3f); cin >> n >> m;
    while (m -- ) {
        int x, y, w; cin >> x >> y >> w;
        g[x][y] = g[y][x] = min(g[x][y], w);
    }
```

```
    int t = prim();
    if (t == INF) cout << "impossible" << endl;
    else cout << t << endl;
} //22.03.19 已测试
```

(稠密图)Kruskal 算法 平均时间复杂度为 $\mathcal{O}(M \log M)$, 简化了并查集.

```
struct DSU {
    vector<int> fa;
    DSU(int n) : fa(n + 1) {
        iota(fa.begin(), fa.end(), 0);
    }
    int get(int x) {
        while (x != fa[x]) {
            x = fa[x] = fa[fa[x]];
        }
        return x;
    }
    bool merge(int x, int y) { // 设 x 是 y 的祖先
        x = get(x), y = get(y);
        if (x == y) return false;
        fa[y] = x;
        return true;
    }
    bool same(int x, int y) {
        return get(x) == get(y);
    }
};
struct Tree {
    using TII = tuple<int, int, int>;
    int n;
    priority_queue<TII, vector<TII>, greater<TII>> ver;
```

```
    Tree(int n) {
        this->n = n;
    }
    void add(int x, int y, int w) {
        ver.emplace(w, x, y); // 注意顺序
    }
    int kruskal() {
        DSU dsu(n);
        int ans = 0, cnt = 0;
        while (ver.size()) {
            auto [w, x, y] = ver.top();
            ver.pop();
            if (dsu.same(x, y)) continue;
            dsu.merge(x, y);
            ans += w;
            cnt++;
        }
        assert(cnt < n - 1); // 输入有误，建树失败
        return ans;
    }
};
```

缩点 (Tarjan 算法)

(有向图) 强连通分量缩点 强连通分量缩点后的图称为 SCC. 以 $\mathcal{O}(N+M)$ 的复杂度完成上述全部操作.

> 性质: 缩点后的图拥有拓扑序 $color_{cnt}, color_{cnt-1}, , 1$, 可以不需再另跑一遍 `topsort` ; 缩点后的图是一张有向无环图 (`DAG` , 拓扑图).

```
struct SCC {
    int n, now, cnt;
    vector<vector<int>> ver;
```

```
    vector<int> dfn, low, col, S;

    SCC(int n) : n(n), ver(n + 1), low(n + 1) {
        dfn.resize(n + 1, -1);
        col.resize(n + 1, -1);
        now = cnt = 0;
    }
    void add(int x, int y) {
        ver[x].push_back(y);
    }
    void tarjan(int x) {
        dfn[x] = low[x] = now++;
        S.push_back(x);
        for (auto y : ver[x]) {
            if (dfn[y] == -1) {
                tarjan(y);
                low[x] = min(low[x], low[y]);
            } else if (col[y] == -1) {
                low[x] = min(low[x], dfn[y]);
            }
        }
        if (dfn[x] == low[x]) {
            int pre;
            cnt++;
            do {
                pre = S.back();
                col[pre] = cnt;
                S.pop_back();
            } while (pre != x);
        }
    }
    auto work() { // [cnt 新图的顶点数量]
        for (int i = 1; i <= n; i++) { // 避免图不连通
```

```
            if (dfn[i] == -1) {
                tarjan(i);
            }
        }

        vector<int> siz(cnt + 1); // siz 每个 scc 中点的数量
        vector<vector<int>> adj(cnt + 1);
        for (int i = 1; i <= n; i++) {
            siz[col[i]]++;
            for (auto j : ver[i]) {
                int x = col[i], y = col[j];
                if (x != y) {
                    adj[x].push_back(y);
                }
            }
        }
        return {cnt, adj, col, siz};
    }
};
```

(无向图) 割边缩点 割边缩点后的图称为边双连通图 (E-DCC), 该模板可以在 $\mathcal{O}(N+M)$ 复杂度内求解图中全部割边, 划分边双 (颜色相同的点位于同一个边双连通分量中).

> 割边 (桥): 将某边 e 删去后, 原图分成两个以上不相连的子图, 称 e 为图的割边.
>
> 边双连通: 在一张连通的无向图中, 对于两个点 u 和 v, 删去任何一条边 (只能删去一条) 它们依旧连通, 则称 u 和 v 边双连通. 一个图如果不存在割边, 则它是一个边双连通图.
>
> 性质补充: 对于一个边双, 删去任意边后依旧联通; 对于边双中的任意两点, 一定存在两条不相交的路径连接这两个点 (路径上可以有公共点, 但是没有公共边).

```
struct EDCC {
    int n, m, now, cnt;
    vector<vector<array<int, 2>>> ver;
    vector<int> dfn, low, col, S;
    set<array<int, 2>> bridge, direct; // 如果不需要，删除这一部分可以得到一些时间

    EDCC(int n) : n(n), low(n + 1), ver(n + 1), dfn(n + 1), col(n + 1) {
        m = now = cnt = 0;
    }
    void add(int x, int y) { // 和 scc 相比多了一条连边
        ver[x].push_back({y, m});
        ver[y].push_back({x, m++});
    }
    void tarjan(int x, int fa) {
        dfn[x] = low[x] = ++now;
        S.push_back(x);
        for (auto &[y, id] : ver[x]) {
            if (!dfn[y]) {
                direct.insert({x, y});
                tarjan(y, id);
                low[x] = min(low[x], low[y]);
                if (dfn[x] < low[y]) {
                    bridge.insert({x, y});
                }
            } else if (id != fa && dfn[y] < dfn[x]) {
                direct.insert({x, y});
                low[x] = min(low[x], dfn[y]);
            }
        }
        if (dfn[x] == low[x]) {
            int pre;
            cnt++;
            do {
```

```
                pre = S.back();
                col[pre] = cnt;
                S.pop_back();
            } while (pre != x);
        }
    }
    auto work() {
        for (int i = 1; i <= n; i++) { // 避免图不连通
            if (!dfn[i]) {
                tarjan(i, 0);
            }
        }
        /**
         * @param cnt 新图的顶点数量, adj 新图, col 旧图节点对应的新图节点
         * @param siz 旧图每一个边双中点的数量
         * @param bridge 全部割边, direct 非割边定向
         */
        vector<int> siz(cnt + 1);
        vector<vector<int>> adj(cnt + 1);
        for (int i = 1; i <= n; i++) {
            siz[col[i]]++;
            for (auto &[j, id] : ver[i]) {
                int x = col[i], y = col[j];
                if (x != y) {
                    adj[x].push_back(y);
                }
            }
        }
        return tuple{cnt, adj, col, siz};
    }
};
```

(无向图) 割点缩点 割点缩点后的图称为点双连通图 (V-DCC), 该模板可以在 $\mathcal{O}(N+M)$ 复杂度内求解图中全部割点, 划分点双 (颜色相同的点位于同一个点双连通分量中).

> 割点 (割顶): 将与某点 i 连接的所有边删去后, 原图分成两个以上不相连的子图, 称 i 为图的割点.
>
> 点双连通: 在一张连通的无向图中, 对于两个点 u 和 v, 删去任何一个点 (只能删去一个, 且不能删 u 和 v 自己) 它们依旧连通, 则称 u 和 v 边双连通. 如果一个图不存在割点, 那么它是一个点双连通图.
>
> 性质补充: 每一个割点至少属于两个点双.

```cpp
struct V_DCC {
    int n;
    vector<vector<int>> ver, col;
    vector<int> dfn, low, S;
    int now, cnt;
    vector<bool> point; // 记录是否为割点

    V_DCC(int n) : n(n) {
        ver.resize(n + 1);
        dfn.resize(n + 1);
        low.resize(n + 1);
        col.resize(2 * n + 1);
        point.resize(n + 1);
        S.clear();
        cnt = now = 0;
    }
    void add(int x, int y) {
        if (x == y) return; // 手动去除重边
        ver[x].push_back(y);
        ver[y].push_back(x);
    }
```

```
void tarjan(int x, int root) {
    low[x] = dfn[x] = ++now;
    S.push_back(x);
    if (x == root && !ver[x].size()) { // 特判孤立点
        ++cnt;
        col[cnt].push_back(x);
        return;
    }

    int flag = 0;
    for (auto y : ver[x]) {
        if (!dfn[y]) {
            tarjan(y, root);
            low[x] = min(low[x], low[y]);
            if (dfn[x] <= low[y]) {
                flag++;
                if (x != root || flag > 1) {
                    point[x] = true; // 标记为割点
                }
                int pre = 0;
                cnt++;
                do {
                    pre = S.back();
                    col[cnt].push_back(pre);
                    S.pop_back();
                } while (pre != y);
                col[cnt].push_back(x);
            }
        } else {
            low[x] = min(low[x], dfn[y]);
        }
    }
}
```

```
    pair<int, vector<vector<int>>> rebuild() { // [新图的顶点数量, 新图]
        work();
        vector<vector<int>> adj(cnt + 1);
        for (int i = 1; i <= cnt; i++) {
            if (!col[i].size()) { // 注意, 孤立点也是 V-DCC
                continue;
            }
            for (auto j : col[i]) {
                if (point[j]) { // 如果 j 是割点
                    adj[i].push_back(point[j]);
                    adj[point[j]].push_back(i);
                }
            }
        }
        return {cnt, adj};
    }
    void work() {
        for (int i = 1; i <= n; ++i) { // 避免图不连通
            if (!dfn[i]) {
                tarjan(i, i);
            }
        }
    }
};
```

链式前向星建图与搜索

很少使用这种建图法.`dfs`：标准复杂度为 $\mathcal{O}(N+M)$. 节点子节点的数量包含它自己 (至少为 1), 深度从 0 开始 (根节点深度为 0).`bfs`：深度从 1 开始 (根节点深度为 1).`topsort`：有向无环图 (包括非联通) 才拥有完整的拓扑序列 (故该算法也可用于判断图中是否存在环). 每次找到入度为 0 的点并将其放入待查找队列.

```
namespace Graph {
    const int N = 1e5 + 7;
    const int M = 1e6 + 7;
    int tot, h[N], ver[M], ne[M];
    int deg[N], vis[M];

    void clear(int n) {
        tot = 0; //多组样例清空
        for (int i = 1; i <= n; ++i) {
            h[i] = 0;
            deg[i] = vis[i] = 0;
        }
    }
    void add(int x, int y) {
        ver[++tot] = y, ne[tot] = h[x], h[x] = tot;
        ++deg[y];
    }
    void dfs(int x) {
        a.push_back(x); // DFS 序
        siz[x] = vis[x] = 1;
        for (int i = h[x]; i; i = ne[i]) {
            int y = ver[i];
            if (vis[y]) continue;
            dis[y] = dis[x] + 1;
            dfs(y);
            siz[x] += siz[y];
        }
        a.push_back(x);
    }
    void bfs(int s) {
        queue<int> q;
        q.push(s);
        dis[s] = 1;
```

```
        while (!q.empty()) {
            int x = q.front();
            q.pop();
            for (int i = h[x]; i; i = ne[i]) {
                int y = ver[i];
                if (dis[y]) continue;
                d[y] = d[x] + 1;
                q.push(y);
            }
        }
    }
    bool topsort() {
        queue<int> q;
        vector<int> ans;
        for (int i = 1; i <= n; ++i)
            if (deg[i] == 0) q.push(i);
        while (!q.empty()) {
            int x = q.front();
            q.pop();
            ans.push_back(x);
            for (int i = h[x]; i; i = ne[i]) {
                int y = ver[i];
                --deg[y];
                if (deg[y] == 0) q.push(y);
            }
        }
        return ans.size() == n; //判断是否存在拓扑排序
    }
} // namespace Graph
```

一般图最大匹配 (带花树算法)

与二分图匹配的差别在于图中可能存在奇环, 时间复杂度与边的数量无关, 为 $\mathcal{O}(N^3)$. 下方模板编号从 0 开始, 例题为 UOJ #79. 一般图最大匹配 .

```
struct Graph {
    int n;
    vector<vector<int>> e;
    Graph(int n) : n(n), e(n) {}
    void add(int u, int v) {
        e[u].push_back(v);
        e[v].push_back(u);
    }
    pair<int, vector<int>> work() {
        vector<int> match(n, -1), vis(n), link(n), f(n), dep(n);
        auto find = [&](int u) {
            while (f[u] != u) u = f[u] = f[f[u]];
            return u;
        };
        auto lca = [&](int u, int v) {
            u = find(u), v = find(v);
            while (u != v) {
                if (dep[u] < dep[v]) swap(u, v);
                u = find(link[match[u]]);
            }
            return u;
        };
        queue<int> q;
        auto blossom = [&](int u, int v, int p) {
            while (find(u) != p) {
                link[u] = v;
                v = match[u];
                if (vis[v] == 0) {
                    vis[v] = 1;
```

```
                q.push(v);
            }
            f[u] = f[v] = p;
            u = link[v];
        }
    };
    auto augment = [&](int u) {
        while (!q.empty()) q.pop();
        iota(f.begin(), f.end(), 0);
        fill(vis.begin(), vis.end(), -1);
        q.push(u);
        vis[u] = 1;
        dep[u] = 0;
        while (!q.empty()) {
            int u = q.front();
            q.pop();
            for (auto v : e[u]) {
                if (vis[v] == -1) {
                    vis[v] = 0;
                    link[v] = u;
                    dep[v] = dep[u] + 1;
                    if (match[v] == -1) {
                        for (int x = v, y = u, temp; y != -1;
                             x = temp, y = x == -1 ? -1 : link[x]) {
                            temp = match[y];
                            match[x] = y;
                            match[y] = x;
                        }
                        return;
                    }
                    vis[match[v]] = 1;
                    dep[match[v]] = dep[u] + 2;
                    q.push(match[v]);
```

```
            } else if (vis[v] == 1 && find(v) != find(u)) {
                int p = lca(u, v);
                blossom(u, v, p);
                blossom(v, u, p);
            }
        }
    }
};
auto greedy = [&]() {
    for (int u = 0; u < n; ++u) {
        if (match[u] != -1) continue;
        for (auto v : e[u]) {
            if (match[v] == -1) {
                match[u] = v;
                match[v] = u;
                break;
            }
        }
    }
};
greedy();
for (int u = 0; u < n; u++) {
    if (match[u] == -1) {
        augment(u);
    }
}
int ans = 0;
for (int u = 0; u < n; u++) {
    if (match[u] != -1) {
        ans++;
    }
}
return {ans / 2, match};
```

```
    }
};

signed main() {
    int n, m;
    cin >> n >> m;

    Graph graph(n);
    for (int i = 1; i <= m; i++) {
        int x, y;
        cin >> x >> y;
        graph.add(x - 1, y - 1);
    }
    auto [ans, match] = graph.work();
    cout << ans << endl;
    for (auto it : match) {
        cout << it + 1 << " ";
    }
}
```

一般图最大权匹配 (带权带花树算法)

下方模板编号从 1 开始, 复杂度为 $\mathcal{O}(N^3)$.

```
namespace Graph {
    const int N = 403 * 2; //两倍点数
    typedef int T; //权值大小
    const T inf = numeric_limits<int>::max() >> 1;
    struct Q { int u, v; T w; } e[N][N];
    T lab[N];
    int n, m = 0, id, h, t, lk[N], sl[N], st[N], f[N], b[N][N], s[N], ed[N], q[N];
    vector<int> p[N];
#define dvd(x) (lab[x.u] + lab[x.v] - e[x.u][x.v].w * 2)
```

```
#define FOR(i, b) for (int i = 1; i <= (int)(b); i++)
#define ALL(x) (x).begin(), (x).end()
#define ms(x, i) memset(x + 1, i, m * sizeof x[0])
    void upd(int u, int v) {
        if (!sl[v] || dvd(e[u][v]) < dvd(e[sl[v]][v])) {
            sl[v] = u;
        }
    }
    void ss(int v) {
        sl[v] = 0;
        FOR(u, n) {
            if (e[u][v].w > 0 && st[u] != v && !s[st[u]]) {
                upd(u, v);
            }
        }
    }
    void ins(int u) {
        if (u <= n) { q[++t] = u; }
        else {
            for (int v : p[u]) ins(v);
        }
    }
    void mdf(int u, int w) {
        st[u] = w;
        if (u > n) {
            for (int v : p[u]) mdf(v, w);
        }
    }
    int gr(int u, int v) {
        v = find(ALL(p[u]), v) - p[u].begin();
        if (v & 1) {
            reverse(1 + ALL(p[u]));
            return (int)p[u].size() - v;
```

```
        }
        return v;
    }
    void stm(int u, int v) {
        lk[u] = e[u][v].v;
        if (u <= n) return;
        Q w = e[u][v];
        int x = b[u][w.u], y = gr(u, x);
        for (int i = 0; i < y; i++) {
            stm(p[u][i], p[u][i ^ 1]);
        }
        stm(x, v);
        rotate(p[u].begin(), y + ALL(p[u]));
    }
    void aug(int u, int v) {
        int w = st[lk[u]];
        stm(u, v);
        if (!w) return;
        stm(w, st[f[w]]), aug(st[f[w]], w);
    }
    int lca(int u, int v) {
        for (++id; u | v; swap(u, v)) {
            if (!u) continue;
            if (ed[u] == id) return u;
            ed[u] = id;
            if (u = st[lk[u]]) u = st[f[u]];
        }
        return 0;
    }
    void add(int u, int a, int v) {
        int x = n + 1, i, j;
        while (x <= m && st[x]) ++x;
        if (x > m) ++m;
```

```
        lab[x] = s[x] = st[x] = 0;
        lk[x] = lk[a];
        p[x].clear();
        p[x].push_back(a);
        for (i = u; i != a; i = st[f[j]]) {
            p[x].push_back(i);
            p[x].push_back(j = st[lk[i]]);
            ins(j);
        }
        reverse(1 + ALL(p[x]));
        for (i = v; i != a; i = st[f[j]]) { // 复制，只需改循环
            p[x].push_back(i);
            p[x].push_back(j = st[lk[i]]);
            ins(j);
        }
        mdf(x, x);
        FOR(i, m) {
            e[x][i].w = e[i][x].w = 0;
        }
        memset(b[x] + 1, 0, n * sizeof b[0][0]);
        for (int u : p[x]) {
            FOR(v, m) {
                if (!e[x][v].w || dvd(e[u][v]) < dvd(e[x][v])) {
                    e[x][v] = e[u][v], e[v][x] = e[v][u];
                }
            }
            FOR(v, n) {
                if (b[u][v]) { b[x][v] = u; }
            }
        }
        ss(x);
    }
    void ex(int u) {
```

```
        for (int x : p[u]) mdf(x, x);
        int a = b[u][e[u][f[u]].u], r = gr(u, a);
        for (int i = 0; i < r; i += 2) {
            int x = p[u][i], y = p[u][i + 1];
            f[x] = e[y][x].u;
            s[x] = 1;
            s[y] = sl[x] = 0;
            ss(y), ins(y);
        }
        s[a] = 1, f[a] = f[u];
        for (int i = r + 1; i < p[u].size(); i++) {
            s[p[u][i]] = -1;
            ss(p[u][i]);
        }
        st[u] = 0;
    }
    bool on(const Q &e) {
        int u = st[e.u], v = st[e.v];
        if (s[v] == -1) {
            f[v] = e.u, s[v] = 1;
            int a = st[lk[v]];
            sl[v] = sl[a] = s[a] = 0;
            ins(a);
        } else if (!s[v]) {
            int a = lca(u, v);
            if (!a) {
                return aug(u, v), aug(v, u), 1;
            } else {
                add(u, a, v);
            }
        }
        return 0;
    }
```

```
bool bfs() {
    ms(s, -1), ms(sl, 0);
    h = 1, t = 0;
    FOR(i, m) {
        if (st[i] == i && !lk[i]) {
            f[i] = s[i] = 0;
            ins(i);
        }
    }
    if (h > t) return 0;
    while (1) {
        while (h <= t) {
            int u = q[h++];
            if (s[st[u]] == 1) continue;
            FOR(v, n) {
                if (e[u][v].w > 0 && st[u] != st[v]) {
                    if (dvd(e[u][v])) upd(u, st[v]);
                    else if (on(e[u][v])) return 1;
                }
            }
        }
        T x = inf;
        for (int i = n + 1; i <= m; i++) {
            if (st[i] == i && s[i] == 1) {
                x = min(x, lab[i] >> 1);
            }
        }
        FOR(i, m) {
            if (st[i] == i && sl[i] && s[i] != 1) {
                x = min(x, dvd(e[sl[i]][i]) >> s[i] + 1);
            }
        }
        FOR(i, n) {
```

```
            if (~s[st[i]]) {
                if ((lab[i] += (s[st[i]] * 2 - 1) * x) <= 0) return 0;
            }
        }
        for (int i = n + 1; i <= m; i++) {
            if (st[i] == i && ~s[st[i]]) {
                lab[i] += (2 - s[st[i]] * 4) * x;
            }
        }
        h = 1, t = 0;
        FOR(i, m) {
            if (st[i] == i && sl[i] && st[sl[i]] != i && !dvd(e[sl[i]][i]) && on(e[
                return 1;
            }
        }
        for (int i = n + 1; i <= m; i++) {
            if (st[i] == i && s[i] == 1 && !lab[i]) ex(i);
        }
    }
    return 0;
}
template<typename TT> i64 work(int N, const vector<tuple<int, int, TT>> &edges) {
    ms(ed, 0), ms(lk, 0);
    n = m = N; id = 0;
    iota(st + 1, st + n + 1, 1);
    T wm = 0; i64 r = 0;
    FOR(i, n) FOR(j, n) {
        e[i][j] = {i, j, 0};
    }
    for (auto [u, v, w] : edges) {
        wm = max(wm, e[v][u].w = e[u][v].w = max(e[u][v].w, (T)w));
    }
    FOR(i, n) { p[i].clear(); }
```

```
        FOR(i, n) FOR(j, n) {
            b[i][j] = i * (i == j);
        }
        fill_n(lab + 1, n, wm);
        while (bfs()) {};
        FOR(i, n) if (lk[i]) {
            r += e[i][lk[i]].w;
        }
        return r / 2;
    }
    auto match() {
        vector<array<int, 2>> ans;
        FOR(i, n) if (lk[i]) {
            ans.push_back({i, lk[i]});
        }
        return ans;
    }
} // namespace Graph
using Graph::work, Graph::match;

signed main() {
    int n, m;
    cin >> n >> m;
    vector<tuple<int, int, i64>> ver(m);
    for (auto &[u, v, w] : ver) {
        cin >> u >> v >> w;
    }
    cout << work(n, ver) << "\n";
    auto ans = match();
}
```

二分图最大匹配

二分图: 一个图能被分为左右两部分, 任何一条边的两个端点都不在同一部分中.

匹配 (独立边集): 一个边的集合, 这些边没有公共顶点.

二分图最大匹配即找到边的数量最多的那个匹配.

一般我们规定, 左半部包含 n_1 个点 (编号 $1-n_1$), 右半部包含 n_2 个点 (编号 $1-n_2$), 保证任意一条边的两个端点都不可能在同一部分中.

匈牙利算法 (KM 算法) 解 $\mathcal{O}(NM)$.

```
signed main() {
    int n1, n2, m;
    cin >> n1 >> n2 >> m;

    vector<vector<int>> ver(n1 + 1);
    for (int i = 1; i <= m; ++i) {
        int x, y;
        cin >> x >> y;
        ver[x].push_back(y); //只需要建立单向边
    }

    int ans = 0;
    vector<int> match(n2 + 1);
    for (int i = 1; i <= n1; ++i) {
        vector<int> vis(n2 + 1);
        auto dfs = [&](auto self, int x) -> bool {
            for (auto y : ver[x]) {
                if (vis[y]) continue;
                vis[y] = 1;
                if (!match[y] || self(self, match[y])) {
```

```
                    match[y] = x;
                    return true;
                }
            }
            return false;
        };
        if (dfs(dfs, i)) {
            ans++;
        }
    }
    cout << ans << endl;
}
```

HopcroftKarp 算法 (基于最大流) 解 该算法基于最大流, 常数极小, 且引入随机化, 几乎卡不掉. 最坏时间复杂度为 $\mathcal{O}(\sqrt{N}M)$, 经测试, 在 N, M 均为 2×10^5 的情况下能在 60ms 内跑完.

```
struct HopcroftKarp {
    int n, m;
    vector<array<int, 2>> ver;
    vector<int> l, r;

    HopcroftKarp(int n, int m) : n(n), m(m) { // 左右半部
        l.assign(n, -1);
        r.assign(m, -1);
    }
    void add(int x, int y) {
        x--, y--; // 这个板子是 0-idx 的
        ver.push_back({x, y});
    }
    int work() {
        vector<int> adj(ver.size());
```

```
mt19937 rgen(chrono::steady_clock::now().time_since_epoch().count());
shuffle(ver.begin(), ver.end(), rgen); // 随机化防卡

vector<int> deg(n + 1);
for (auto &[u, v] : ver) {
    deg[u]++;
}
for (int i = 1; i <= n; i++) {
    deg[i] += deg[i - 1];
}
for (auto &[u, v] : ver) {
    adj[--deg[u]] = v;
}

int ans = 0;
vector<int> a, p, q(n);
while (true) {
    a.assign(n, -1), p.assign(n, -1);

    int t = 0;
    for (int i = 0; i < n; i++) {
        if (l[i] == -1) {
            q[t++] = a[i] = p[i] = i;
        }
    }

    bool match = false;
    for (int i = 0; i < t; i++) {
        int x = q[i];
        if (~l[a[x]]) continue;

        for (int j = deg[x]; j < deg[x + 1]; j++) {
            int y = adj[j];
```

```
                    if (r[y] == -1) {
                        while (~y) {
                            r[y] = x;
                            swap(l[x], y);
                            x = p[x];
                        }
                        match = true;
                        ++ans;
                        break;
                    }
                    if (p[r[y]] == -1) {
                        q[t++] = y = r[y];
                        p[y] = x;
                        a[y] = a[x];
                    }
                }
            }
            if (!match) break;
        }
        return ans;
    }
    vector<array<int, 2>> answer() {
        vector<array<int, 2>> ans;
        for (int i = 0; i < n; i++) {
            if (~l[i]) {
                ans.push_back({i, l[i]});
            }
        }
        return ans;
    }
};

signed main() {
```

```
    int n1, n2, m;
    cin >> n1 >> n2 >> m;
    HopcroftKarp flow(n1, n2);
    while (m--) {
        int x, y;
        cin >> x >> y;
        flow.add(x, y);
    }

    cout << flow.work() << "\n";

    auto match = flow.answer();
    for (auto [u, v] : match) {
        cout << u << " " << v << "\n";
    }
}
```

二分图最大权匹配 (二分图完美匹配)

> 定义: 找到边权和最大的那个匹配.
>
> 一般我们规定, 左半部包含 n_1 个点 (编号 $1-n_1$), 右半部包含 n_2 个点 (编号 $1-n_2$).

使用匈牙利算法 (KM 算法) 解, 时间复杂度为 $\mathcal{O}(N^3)$. 下方模板用于求解最大权值, 且可以输出其中一种可行方案, 例题为 UOJ #80. 二分图最大权匹配 .

```
struct MaxCostMatch {
    vector<int> ansl, ansr, pre;
    vector<int> lx, ly;
    vector<vector<int>> ver;
    int n;
```

```
    MaxCostMatch(int n) : n(n) {
        ver.resize(n + 1, vector<int>(n + 1));
        ansl.resize(n + 1, -1);
        ansr.resize(n + 1, -1);
        lx.resize(n + 1);
        ly.resize(n + 1, -1E18);
        pre.resize(n + 1);
    }
    void add(int x, int y, int w) {
        ver[x][y] = w;
    }
    void bfs(int x) {
        vector<bool> visl(n + 1), visr(n + 1);
        vector<int> slack(n + 1, 1E18);
        queue<int> q;
        function<bool(int)> check = [&](int x) {
            visr[x] = 1;
            if (~ansr[x]) {
                q.push(ansr[x]);
                visl[ansr[x]] = 1;
                return false;
            }
            while (~x) {
                ansr[x] = pre[x];
                swap(x, ansl[pre[x]]);
            }
            return true;
        };
        q.push(x);
        visl[x] = 1;
        while (1) {
            while (!q.empty()) {
                int x = q.front();
```

```
                q.pop();
                for (int y = 1; y <= n; ++y) {
                    if (visr[y]) continue;
                    int del = lx[x] + ly[y] - ver[x][y];
                    if (del < slack[y]) {
                        pre[y] = x;
                        slack[y] = del;
                        if (!slack[y] && check(y)) return;
                    }
                }
            }
            int val = 1E18;
            for (int i = 1; i <= n; ++i) {
                if (!visr[i]) {
                    val = min(val, slack[i]);
                }
            }
            for (int i = 1; i <= n; ++i) {
                if (visl[i]) lx[i] -= val;
                if (visr[i]) {
                    ly[i] += val;
                } else {
                    slack[i] -= val;
                }
            }
            for (int i = 1; i <= n; ++i) {
                if (!visr[i] && !slack[i] && check(i)) {
                    return;
                }
            }
        }
    }
    int work() {
```

```
        for (int i = 1; i <= n; ++i) {
            for (int j = 1; j <= n; ++j) {
                ly[i] = max(ly[i], ver[j][i]);
            }
        }
        for (int i = 1; i <= n; ++i) bfs(i);
        int res = 0;
        for (int i = 1; i <= n; ++i) {
            res += ver[i][ansl[i]];
        }
        return res;
    }
    void getMatch(int x, int y) { // 获取方案 (0 代表无匹配)
        for (int i = 1; i <= x; ++i) {
            cout << (ver[i][ansl[i]] ? ansl[i] : 0) << " ";
        }
        cout << endl;
        for (int i = 1; i <= y; ++i) {
            cout << (ver[i][ansr[i]] ? ansr[i] : 0) << " ";
        }
        cout << endl;
    }
};

signed main() {
    int n1, n2, m;
    cin >> n1 >> n2 >> m;

    MaxCostMatch match(max(n1, n2));
    for (int i = 1; i <= m; i++) {
        int x, y, w;
        cin >> x >> y >> w;
        match.add(x, y, w);
```

```
    }
    cout << match.work() << '\n';
    match.getMatch(n1, n2);
}
```

二分图最大独立点集 (Konig 定理)

给出一张二分图, 要求选择一些点使得它们两两没有边直接连接. 最小点覆盖等价于最大匹配数, 转换为最小割模板, 答案即为总点数减去最大流得到的值.

```
cout << n - flow.work(s, t) << endl;
```

最长路 (topsort+DP 算法)

计算一张 DAG 中的最长路径, 在执行前可能需要使用 tarjan 重构一张正确的 DAG , 复杂度 $\mathcal{O}(N+M)$.

```
struct DAG {
    int n;
    vector<vector<pair<int, int>>> ver;
    vector<int> deg, dis;
    DAG(int n) : n(n) {
        ver.resize(n + 1);
        deg.resize(n + 1);
        dis.assign(n + 1, -1E18);
    }
    void add(int x, int y, int w) {
        ver[x].push_back({y, w});
        ++deg[y];
    }
    int topsort(int s, int t) {
        queue<int> q;
```

```
        for (int i = 1; i <= n; i++) {
            if (deg[i] == 0) {
                q.push(i);
            }
        }
        dis[s] = 0;
        while (!q.empty()) {
            int x = q.front();
            q.pop();
            for (auto [y, w] : ver[x]) {
                dis[y] = max(dis[y], dis[x] + w);
                --deg[y];
                if (deg[y] == 0) {
                    q.push(y);
                }
            }
        }
        return dis[t];
    }
};

signed main() {
    int n, m;
    cin >> n >> m;
    DAG dag(n);
    for (int i = 1; i <= m; i++) {
        int x, y, w;
        cin >> x >> y >> w;
        dag.add(x, y, w);
    }

    int s, t;
    cin >> s >> t;
```

```
    cout << dag.topsort(s, t) << "\n";
}
```

最短路径树 (SPT 问题)

> 定义: 在一张无向带权联通图中, 有这样一棵**生成树**: 满足从根节点到任意点的路径都为原图中根到任意点的最短路径.
>
> 性质: 记根节点 $Root$ 到某一结点 x 的最短距离 $dis_{Root,x}$, 在 SPT 上这两点之间的距离为 $len_{Root,x}$ -则两者长度相等.

该算法与最小生成树无关, 基于最短路 Djikstra 算法完成 (但多了个等于号). 下方代码实现的功能为: 读入图后, 输出以 1 为根的 SPT 所使用的各条边的编号, 边权和.

```
map<pair<int, int>, int> id;
namespace G {
    vector<pair<int, int> > ver[N];
    map<pair<int, int>, int> edge;
    int v[N], d[N], pre[N], vis[N];
    int ans = 0;

    void add(int x, int y, int w) {
        ver[x].push_back({y, w});
        edge[{x, y}] = edge[{y, x}] = w;
    }
    void djikstra(int s) { // ! 注意, 该 djikstra 并非原版, 多加了一个等于号
        priority_queue<PII, vector<PII>, greater<PII> > q; q.push({0, s});
        memset(d, 0x3f, sizeof d); d[s] = 0;
        while (!q.empty()) {
            int x = q.top().second; q.pop();
            if (v[x]) continue; v[x] = 1;
            for (auto [y, w] : ver[x]) {
                if (d[y] >= d[x] + w) { // ! 注意,SPT 这里修改为 >= 号
```

```
                    d[y] = d[x] + w;
                    pre[y] = x; // 记录前驱结点
                    q.push({d[y], y});
                }
            }
        }
    }
    void dfs(int x) {
        vis[x] = 1;
        for (auto [y, w] : ver[x]) {
            if (vis[y]) continue;
            if (pre[y] == x) {
                cout << id[{x, y}] << " "; // 输出 SPT 所使用的边编号
                ans += edge[{x, y}];
                dfs(y);
            }
        }
    }
    void solve(int n) {
        djikstra(1); // 以 1 为根
        dfs(1); // 以 1 为根
        cout << endl << ans; // 输出 SPT 的边权和
    }
}
bool Solve() {
    int n, m; cin >> n >> m;
    for (int i = 1; i <= m; ++ i) {
        int x, y, w; cin >> x >> y >> w;
        G::add(x, y, w), G::add(y, x, w);
        id[{x, y}] = id[{y, x}] = i;
    }
    G::solve(n);
    return 0;
```

```
}
```

无源汇点的最小割问题 StoerWagner

> 也称为全局最小割. 定义补充 (与网络流中的定义不同):
>
> **割**: 是一个边集, 去掉其中所有边能使一张网络流图不再连通 (即分成两个子图).

通过**递归**的方式来解决**无向正权图**上的全局最小割问题, 算法复杂度 $\mathcal{O}(VE + V^2 \log V)$, 一般可近似看作 $\mathcal{O}(V^3)$.

```
signed main() {
    int n, m;
    cin >> n >> m;

    DSU dsu(n); // 这里引入 DSU 判断图是否联通, 如题目有保证, 则不需要此步骤
    vector<vector<int>> edge(n + 1, vector<int>(n + 1));
    for (int i = 1; i <= m; i++) {
        int x, y, w;
        cin >> x >> y >> w;
        dsu.merge(x, y);
        edge[x][y] += w;
        edge[y][x] += w;
    }

    if (dsu.Poi(1) != n || m < n - 1) { // 图不联通
        cout << 0 << endl;
        return 0;
    }

    int MinCut = INF, S = 1, T = 1; // 虚拟源汇点
    vector<int> bin(n + 1);
    auto contract = [&]() -> int { // 求解 S 到 T 的最小割, 定义为 cut of phase
```

```
        vector<int> dis(n + 1), vis(n + 1);
        int Min = 0;
        for (int i = 1; i <= n; i++) {
            int k = -1, maxc = -1;
            for (int j = 1; j <= n; j++) {
                if (!bin[j] && !vis[j] && dis[j] > maxc) {
                    k = j;
                    maxc = dis[j];
                }
            }
            if (k == -1) return Min;
            S = T, T = k, Min = maxc;
            vis[k] = 1;
            for (int j = 1; j <= n; j++) {
                if (!bin[j] && !vis[j]) {
                    dis[j] += edge[k][j];
                }
            }
        }
        return Min;
    };
    for (int i = 1; i < n; i++) { // 这里取不到等号
        int val = contract();
        bin[T] = 1;
        MinCut = min(MinCut, val);
        if (!MinCut) {
            cout << 0 << endl;
            return 0;
        }
        for (int j = 1; j <= n; j++) {
            if (!bin[j]) {
                edge[S][j] += edge[j][T];
                edge[j][S] += edge[j][T];
```

```
            }
        }
    }
    cout << MinCut << endl;
}
```

欧拉路径/欧拉回路 Hierholzers

> 欧拉路径: 一笔画完图中全部边, 画的顺序就是一个可行解; 当起点终点相同时称欧拉回路.

有向图欧拉路径存在判定 有向图欧拉路径存在:[1] 恰有一个点出度比入度多 1 (为起点);[2] 恰有一个点入度比出度多 1 (为终点);[3] 恰有 $N-2$ 个点入度均等于出度. 如果是欧拉回路, 则上方起点与终点的条件不存在, 全部点均要满足最后一个条件.

```
signed main() {
    int n, m;
    cin >> n >> m;

    DSU dsu(n + 1); // 如果保证连通，则不需要 DSU
    vector<unordered_multiset<int>> ver(n + 1); // 如果对于字典序有要求，则不能使用
    vector<int> degI(n + 1), degO(n + 1);
    for (int i = 1; i <= m; i++) {
        int x, y;
        cin >> x >> y;
        ver[x].insert(y);
        degI[y]++;
        degO[x]++;
        dsu.merge(x, y); // 直接当无向图
    }
    int s = 1, t = 1, cnt = 0;
    for (int i = 1; i <= n; i++) {
```

```
        if (degI[i] == degO[i]) {
            cnt++;
        } else if (degI[i] + 1 == degO[i]) {
            s = i;
        } else if (degI[i] == degO[i] + 1) {
            t = i;
        }
    }
    if (dsu.size(1) != n || (cnt != n - 2 && cnt != n)) {
        cout << "No\n";
    } else {
        cout << "Yes\n";
    }
}
```

无向图欧拉路径存在判定 无向图欧拉路径存在:[1] 恰有两个点度数为奇数(为起点与终点);[2] 恰有 $N-2$ 个点度数为偶数.

```
signed main() {
    int n, m;
    cin >> n >> m;

    DSU dsu(n + 1); // 如果保证连通，则不需要 DSU
    vector<unordered_multiset<int>> ver(n + 1); // 如果对于字典序有要求，则不能使用
    vector<int> deg(n + 1);
    for (int i = 1; i <= m; i++) {
        int x, y;
        cin >> x >> y;
        ver[x].insert(y);
        ver[y].insert(x);
        deg[y]++;
        deg[x]++;
        dsu.merge(x, y); // 直接当无向图
```

```
    }
    int s = -1, t = -1, cnt = 0;
    for (int i = 1; i <= n; i++) {
        if (deg[i] % 2 == 0) {
            cnt++;
        } else if (s == -1) {
            s = i;
        } else {
            t = i;
        }
    }
    if (dsu.size(1) != n || (cnt != n - 2 && cnt != n)) {
        cout << "No\n";
    } else {
        cout << "Yes\n";
    }
}
```

有向图欧拉路径求解 (字典序最小)

```
vector<int> ans;
auto dfs = [&](auto self, int x) -> void {
    while (ver[x].size()) {
        int net = *ver[x].begin();
        ver[x].erase(ver[x].begin());
        self(self, net);
    }
    ans.push_back(x);
};
dfs(dfs, s);
reverse(ans.begin(), ans.end());
for (auto it : ans) {
    cout << it << " ";
```

```
}
```

无向图欧拉路径求解

```
auto dfs = [&](auto self, int x) -> void {
    while (ver[x].size()) {
        int net = *ver[x].begin();
        ver[x].erase(ver[x].find(net));
        ver[net].erase(ver[net].find(x));
        cout << x << " " << net << endl;
        self(self, net);
    }
};
dfs(dfs, s);
```

差分约束

给出一组包含 m 个不等式, 有 n 个未知数的形如:

$$\begin{cases} x*u_1 - x*v*1 \leq w_1 \\ x*u*2 - x*v*2 \leq w_2 \\ \cdots \\ x*u*m - x*v_m \leq w_m \end{cases}$$

的不等式组, 求任意一组满足这个不等式组的解.SPFA 解,$\mathcal{O}(nm)$.参考

```
signed main() {
    int n, m;
    cin >> n >> m;

    vector<array<int, 3>> e(m + 1);
    for (int i = 1; i <= m; i++) {
        int u, v, w;
        cin >> u >> v >> w;
```

```
        e[i] = {v, u, w};
    }

    vector<int> d(n + 1, 1E9);
    d[1] = 0;
    for (int i = 1; i < n; i++) {
        for (int j = 1; j <= m; j++) {
            auto [u, v, w] = e[j];
            d[v] = min(d[v], d[u] + w);
        }
    }
    for (int i = 1; i <= m; i++) {
        auto [u, v, w] = e[i];
        if (d[v] > d[u] + w) {
            cout << "NO\n";
            return 0;
        }
    }
    for (int i = 1; i <= n; i++) {
        cout << d[i] << " \n"[i == n];
    }
    return 0;
}
```

2-Sat

基础封装 基于 tarjan 缩点, 时间复杂度为 $\mathcal{O}(N+M)$. 注意下标从 0 开始, 答案输出为字典序最小的一个可行解.

```
struct TwoSat {
    int n;
    vector<vector<int>> e;
    vector<bool> ans;
```

```
    TwoSat(int n) : n(n), e(2 * n), ans(n) {}
    void add(int u, bool f, int v, bool g) {
        e[2 * u + !f].push_back(2 * v + g);
        e[2 * v + !g].push_back(2 * u + f);
    }
    bool work() {
        vector<int> id(2 * n, -1), dfn(2 * n, -1), low(2 * n, -1);
        vector<int> stk;
        int now = 0, cnt = 0;
        auto tarjan = [&](auto self, int u) -> void {
            stk.push_back(u);
            dfn[u] = low[u] = now++;
            for (auto v : e[u]) {
                if (dfn[v] == -1) {
                    self(self, v);
                    low[u] = min(low[u], low[v]);
                } else if (id[v] == -1) {
                    low[u] = min(low[u], dfn[v]);
                }
            }
            if (dfn[u] == low[u]) {
                int v;
                do {
                    v = stk.back();
                    stk.pop_back();
                    id[v] = cnt;
                } while (v != u);
                ++cnt;
            }
        };
        for (int i = 0; i < 2 * n; ++i) {
            if (dfn[i] == -1) {
                tarjan(tarjan, i);
```

```
            }
        }
        for (int i = 0; i < n; ++i) {
            if (id[2 * i] == id[2 * i + 1]) return false;
            ans[i] = id[2 * i] > id[2 * i + 1];
        }
        return true;
    }
    vector<bool> answer() {
        return ans;
    }
};
```

答案不唯一时不输出 在运行后针对每一个点进行一次 dfs, 时间复杂度为 $\mathcal{O}(N^2)$, 当且仅当答案唯一时才输出, 否则输出 ? 替代.

2-Sat 方案计数为 NPC 问题.

```
// 结构体中增加
int check(int x, int y) {
    vector<int> vis(2 * n);
    auto dfs = [&](auto self, int x) -> void {
        vis[x] = 1;
        for (auto y : e[x]) {
            if (vis[y]) continue;
            self(self, y);
        }
    };
    dfs(dfs, x);
    return vis[y];
}
// 主函数中增加
for (int i = 0; i < n; i++) {
    if (sat.check(2 * i, 2 * i + 1)) {
```

```
            cout << 1 << " ";
        } else if (sat.check(2 * i + 1, 2 * i)) {
            cout << 0 << " ";
        } else {
            cout << "?" << " ";
        }
}
```

图论常见结论及例题

常见结论

1. 要在有向图上求一个最大点集, 使得任意两个点 (i, j) 之间至少存在一条路径 (可以是从 i 到 j , 也可以反过来, 这两种有一个就行), **即求解最长路**;
2. 要求出连通图上的任意一棵生成树, 只需要跑一遍 **bfs** ;
3. 给出一棵树, 要求添加尽可能多的边, 使得其是二分图: 对树进行二分染色, 显然, 相同颜色的点之间连边不会破坏二分图的性质, 故可添加的最多的边数即为 $cnt_{\texttt{Black}} * cnt_{\texttt{White}} - (n-1)$;
4. 当一棵树可以被黑白染色时, 所有染黑节点的度之和等于所有染白节点的度之和;
5. 在竞赛图中, 入度小的点, 必定能到达出度小 (入度大) 的点 See .
6. 在竞赛图中, 将所有点按入度从小到大排序, 随后依次遍历, 若对于某一点 i 满足前 i 个点的入度之和恰好等于 $\left\lfloor \dfrac{n \cdot (n+1)}{2} \right\rfloor$, 那么对于上一次满足这一条件的点 p ,$p+1$ 到 i 点构成一个新的强连通分量 See .

   > 举例说明, 设满足上方条件的点为 p_1, p_2 $(p_1 + 1 < p_2)$, 那么点 1 到 p_1 构成一个强连通分量, 点 $p_1 + 1$ 到 p_2 构成一个强连通分量.

7. 选择图中最少数量的边删除, 使得图不连通, 即求最小割; 如果是删除点, 那么拆点后求最小割 See.
8. 如果一张图是**平面图**, 那么其边数一定小于等于 $3n-6$ See .
9. 若一张有向完全图存在环, 则一定存在三元环.
10. 竞赛图三元环计数:See .
11. 有向图判是否存在环直接用 topsort; 无向图判是否存在环直接用 dsu, 也可以使用 topsort, 条件变为 `deg[i] <= 1` 时入队.

常见例题

杂 题意: 给出一棵节点数为 $2n$ 的树, 要求将点分割为 n 个点对, 使得点对的点之间的距离和最大.

可以转化为边上问题: 对于每一条边, 其被利用的次数即为 min $\{$其左边的点的数量, 其右边的点的数量$\}$, 使用树形 dp 计算一遍即可. 如下图样例, 答案为 10 .

```
vector<int> val(n + 1, 1);
int ans = 0;
function<void(int, int)> dfs = [&](int x, int fa) {
    for (auto y : ver[x]) {
        if (y == fa) continue;
        dfs(y, x);
        val[x] += val[y];
        ans += min(val[y], k - val[y]);
    }
};
dfs(1, 0);
cout << ans << endl;
```

题意: 以哪些点为起点可以无限的在有向图上绕

概括一下这些点可以发现, 一类是环上的点, 另一类是可以到达环的点. 建反图跑一遍 topsort 板子, 根据容斥, 未被移除的点都是答案 See .

题意: 添加最少的边, 使得有向图变成一个 SCC

将原图的 SCC 缩点, 统计缩点后的新图上入度为 0 和出度为 0 的点的数量 $cnt_{\text{in}}, cnt_{\text{out}}$, 答案即为 $\max(cnt_{\text{in}}, cnt_{\text{out}})$. 过程大致是先将一个出度为 0 的点和一个入度为 0 的点相连, 剩下的点随便连 See .

题意: 添加最少的边, 使得无向图变成一个 E-DCC

将原图的 E-DCC 缩点, 统计缩点后的新图上入度为 1 的点 (叶子结点) 的数量 cnt , 答案即为 $\left\lceil \frac{cnt}{2} \right\rceil$. 过程大致是每次找两个叶子结点 (但是还有一些条件限制) 相连, 若最后余下一个点随便连 See .

题意: 在树上找到一个最大的连通块, 使得这个联通内点权和边权之和最大, 输出这个值, 数据中存在负数的情况.

使用 dfs 即可解决.

```
LL n, point[N];
LL ver[N], head[N], nex[N], tot; bool v[N];
map<pair<LL, LL>, LL> edge;
// void add(LL x, LL y) {}
void dfs(LL x) {
    for (LL i = head[x]; i; i = nex[i]) {
        LL y = ver[i];
        if (v[y]) continue;
        v[y] = true; dfs(y); v[y] = false;
    }
    for (LL i = head[x]; i; i = nex[i]) {
        LL y = ver[i];
```

```
        if (v[y]) continue;
        point[x] += max(point[y] + edge[{x, y}], 0LL);
    }
}
void Solve() {
    cin >> n;
    FOR(i, 1, n) cin >> point[i];
    FOR(i, 2, n) {
        LL x, y, w; cin >> x >> y >> w;
        edge[{x, y}] = edge[{y, x}] = w;
        add(x, y), add(y, x);
    }
    v[1] = true; dfs(1); LL ans = -MAX18;
    FOR(i, 1, n) ans = max(ans, point[i]);
    cout << ans << endl;
}
```

Prfer 序列: 凯莱公式 题意: 给定 n 个顶点, 可以构建出多少棵标记树

$n \leq 4$ 时的样例如上, 通项公式为 n^{n-2} .

Prfer 序列扩展 一个 n 个点 m 条边的带标号无向图有 k 个连通块. 我们希望添加 $k-1$ 条边使得整个图连通, 求方案数量 See .

设 s_i 表示每个连通块的数量, 通项公式为 $n^{k-2} \cdot \prod_{i=1}^{k} s_i$, 当 $k < 2$ 时答案为 1 .

单源最短/次短路计数

```
const int N = 2e5 + 7, M = 1e6 + 7;
int n, m, s, e; int d[N][2], v[N][2]; // 0 代表最短路, 1 代表次短路
```

```
Z num[N][2];

void Clear() {
    for (int i = 1; i <= n; ++ i) h[i] = edge[i] = 0;
    tot = 0;
    for (int i = 1; i <= n; ++ i) num[i][0] = num[i][1] = v[i][0] = v[i][1] = 0;
    for (int i = 1; i <= n; ++ i) d[i][0] = d[i][1] = INF;
}

int ver[M], ne[M], h[N], edge[M], tot;
void add(int x, int y, int w) {
    ver[++ tot] = y, ne[tot] = h[x], h[x] = tot;
    edge[tot] = w;
}

void dji() {
    priority_queue<PIII, vector<PIII>, greater<PIII> > q; q.push({0, s, 0});
    num[s][0] = 1; d[s][0] = 0;
    while (!q.empty()) {
        auto [dis, x, type] = q.top(); q.pop();
        if (v[x][type]) continue; v[x][type] = 1;
        for (int i = h[x]; i; i = ne[i]) {
            int y = ver[i], w = dis + edge[i];
            if (d[y][0] > w) {
                d[y][1] = d[y][0], num[y][1] = num[y][0];
                    // 如果找到新的最短路，将原有的最短路数据转化为次短路
                q.push({d[y][1], y, 1});
                d[y][0] = w, num[y][0] = num[x][type];
                q.push({d[y][0], y, 0});
            }
            else if (d[y][0] == w) num[y][0] += num[x][type];
            else if (d[y][1] > w) {
                d[y][1] = w, num[y][1] = num[x][type];
```

```
                q.push({d[y][1], y, 1});
            }
            else if (d[y][1] == w) num[y][1] += num[x][type];
        }
    }
}
void Solve() {
    cin >> n >> m >> s >> e;
    Clear(); //多组样例务必完全清空
    for (int i = 1; i <= m; ++ i) {
        int x, y, w; cin >> x >> y; w = 1;
        add(x, y, w), add(y, x, w);
    }
    dji();
    Z ans = num[e][0];
    if (d[e][1] == d[e][0] + 1) {
        ans += num[e][1]; // 只有在次短路满足条件时才计算 (距离恰好比最短路大 1
    }
    cout << ans.val() << endl;
}
```

判定图中是否存在负环 使用 SPFA，复杂度为 $\mathcal{O}(KM)$，其中常数 K 相较裸的 SPFA 更高.

```
const int N = 1e5 + 7, M = 1e6 + 7;
int n, m;
int ver[M], ne[M], h[N], edge[M], tot;
int d[N], v[N], num[N];

void add(int x, int y, int w)  {
    ver[++ tot] = y, ne[tot] = h[x], h[x] = tot;
    edge[tot] = w;
}
```

```
bool spfa() {
    queue<int> q;
    for (int i = 1; i <= n; ++ i) q.push(i), v[i] = 1; //全部入队
    while(!q.empty()) {
        int x = q.front(); q.pop();
        v[x] = 0;
        for (int i = h[x]; i; i = ne[i]) {
            int y = ver[i];
            if(d[y] > d[x] + edge[i]) {
                num[y] = num[x] + 1;
                if (num[y] >= n) return true;
                d[y] = d[x] + edge[i];
                if(!v[y]) q.push(y), v[y] = 1;
            }
        }
    }
    return false;
}
int main() {
    cin >> n >> m;
    for (int i = 1; i <= m; ++ i) {
        int x, y, w; cin >> x >> y >> w;
        add(x, y, w);
    }
    if(spfa() == true) cout << "Yes" << endl;
    else cout << "No" << endl;
}
```

输出任意一个三元环 原题: 给出一张有向完全图, 输出任意一个三元环上的全部元素 See . 使用 dfs, 复杂度 $\mathcal{O}(N+M)$, 可以扩展到非完全图和无向图.

```
int n;
cin >> n;
vector<vector<int>> a(n + 1, vector<int>(n + 1));
for (int i = 1; i <= n; ++i) {
    for (int j = 1; j <= n; ++j) {
        char x;
        cin >> x;
        if (x == '1') a[i][j] = 1;
    }
}

vector<int> vis(n + 1);
function<void(int, int)> dfs = [&](int x, int fa) {
    vis[x] = 1;
    for (int y = 1; y <= n; ++y) {
        if (a[x][y] == 0) continue;
        if (a[y][fa] == 1) {
            cout << fa << " " << x << " " << y;
            exit(0);
        }
        if (!vis[y]) dfs(y, x); // 这一步的 if 判断很关键
    }
};
for (int i = 1; i <= n; ++i) {
    if (!vis[i]) dfs(i, -1);
}
cout << -1;
```

带权最小环大小与计数 原题: 给出一张有向带权图, 求解图上最小环的长度, 有多少个这样的最小环 See . 使用 floyd, 复杂度为 $\mathcal{O}(N^3)$, 可以扩展到无向图.

```
LL Min = 1e18, ans = 0;
for (int k = 1; k <= n; k++) {
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
            if (dis[i][j] > dis[i][k] + dis[k][j]) {
                dis[i][j] = dis[i][k] + dis[k][j];
                cnt[i][j] = cnt[i][k] * cnt[k][j] % mod;
            } else if (dis[i][j] == dis[i][k] + dis[k][j]) {
                cnt[i][j] = (cnt[i][j] + cnt[i][k] * cnt[k][j] % mod) % mod;
            }
        }
    }
    for (int i = 1; i < k; i++) {
        if (a[k][i]) {
            if (a[k][i] + dis[i][k] < Min) {
                Min = a[k][i] + dis[i][k];
                ans = cnt[i][k];
            } else if (a[k][i] + dis[i][k] == Min) {
                ans = (ans + cnt[i][k]) % mod;
            }
        }
    }
}
```

最小环大小 原题: 给出一张无向图, 求解图上最小环的长度, 有多少个这样的最小环 See . 使用 floyd, 可以扩展到有向图.

```
int flody(int n) {
    for (int i = 1; i <= n; ++ i) {
        for (int j = 1; j <= n; ++ j) {
            val[i][j] = dis[i][j]; // 记录最初的边权值
        }
    }
```

```
    int ans = 0x3f3f3f3f;
    for (int k = 1; k <= n; ++ k) {
        for (int i = 1; i < k; ++ i) { // 注意这里是没有等于号的
            for (int j = 1; j < i; ++ j) {
                ans = min(ans, dis[i][j] + val[i][k] + val[k][j]);
            }
        }
    for (int i = 1; i <= n; ++ i) { // 往下是标准的 flody
        for (int j = 1; j <= n; ++ j) {
                dis[i][j] = min(dis[i][j], dis[i][k] + dis[k][j]);
            }
        }
    }
    return ans;
}
```

使用 bfs, 复杂度为 $\mathcal{O}(N^2)$.

```
auto bfs = [&] (int s) {
    queue<int> q; q.push(s);
    dis[s] = 0;
    fa[s] = -1;
    while (q.size()) {
        auto x = q.front(); q.pop();
        for (auto y : ver[x]) {
            if (y == fa[x]) continue;
            if (dis[y] == -1) {
                dis[y] = dis[x] + 1;
                fa[y] = x;
                q.push(y);
            }
            else ans = min(ans, dis[x] + dis[y] + 1);
        }
    }
```

```
};
for (int i = 1; i <= n; ++ i) {
    fill(dis + 1, dis + 1 + n, -1);
    bfs(i);
}
cout << ans;
```

本质不同简单环计数 原题: 给出一张无向图, 输出简单环的数量 See . 注意这里环套环需要分别多次统计, 下图答案应当为 7. 使用状压 dp, 复杂度为 $\mathcal{O}(M \cdot 2^N)$, 可以扩展到有向图.

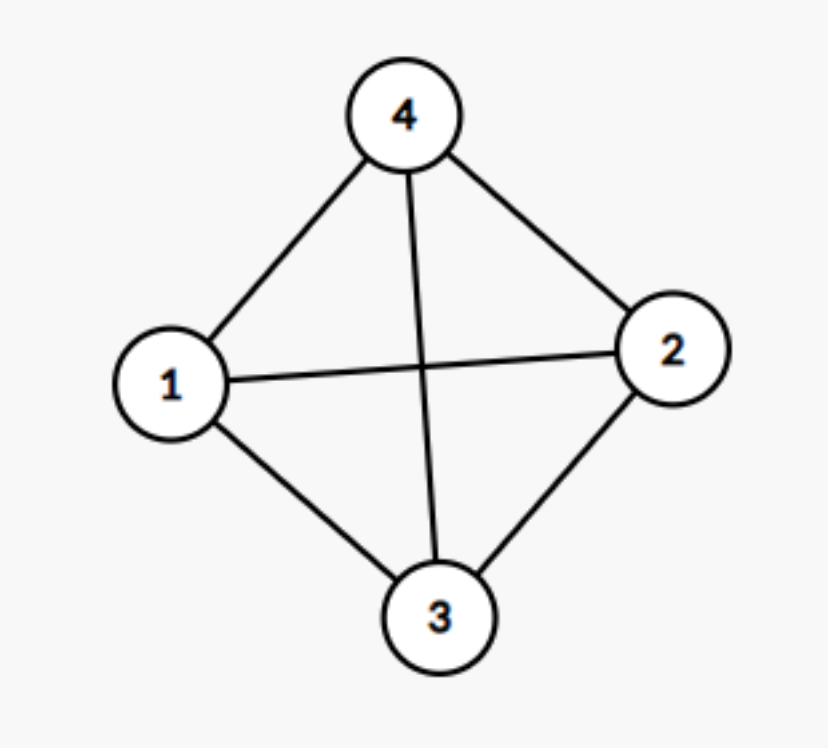


图 1: image.png

```
int n, m;
cin >> n >> m;
vector<vector<int>> G(n);
```

```
for (int i = 0; i < m; i++) {
    int u, v;
    cin >> u >> v;
    u--, v--;
    G[u].push_back(v);
    G[v].push_back(u);
}
vector<vector<LL>> dp(1 << n, vector<LL>(n));
for (int i = 0; i < n; i++) dp[1 << i][i] = 1;
LL ans = 0;
for (int st = 1; st < (1 << n); st++) {
    for (int u = 0; u < n; u++) {
        if (!dp[st][u]) continue;
        int start = st & -st;
        for (auto v : G[u]) {
            if ((1 << v) < start) continue;
            if ((1 << v) & st) {
                if ((1 << v) == start) {
                    ans += dp[st][u];
                }
            } else {
                dp[st | (1 << v)][v] += dp[st][u];
            }
        }
    }
}
cout << (ans - m) / 2 << "\n";
```

输出任意一个非二元简单环 原题: 给出一张无向图, 不含自环与重边, 输出任意一个简单环的大小以及其上面的全部元素 See . 注意输出的环的大小是随机的, **不等价于最小环**.

由于不含重边与自环, 所以环的大小至少为 3 , 使用 dfs 处理出 dfs 序, 复杂

度为 $\mathcal{O}(N+M)$, 可以扩展到有向图; 如果有向图中二元环也允许计入答案, 则需要删除下方标注行.

```
vector<int> dis(n + 1, -1), fa(n + 1);
auto dfs = [&](auto self, int x) -> void {
    for (auto y : ver[x]) {
        if (y == fa[x]) continue; // 二元环需删去该行
        if (dis[y] == -1) {
            dis[y] = dis[x] + 1;
            fa[y] = x;
            self(self, y);
        } else if (dis[y] < dis[x]) {
            cout << dis[x] - dis[y] + 1 << endl;
            int pre = x;
            cout << pre << " ";
            while (pre != y) {
                pre = fa[pre];
                cout << pre << " ";
            }
            cout << endl;
            exit(0);
        }
    }
};
for (int i = 1; i <= n; i++) {
    if (dis[i] == -1) {
        dis[i] = 0;
        dfs(dfs, 1);
    }
}
```

有向图环计数 原题: 给出一张有向图, 输出环的数量. 注意这里环套环仅需要计算一次, 数据包括二元环和自环, 下图例应当输出 3 个环. 使用 dfs 染色

法, 复杂度为 $\mathcal{O}(N+M)$.

```
int ans = 0;
vector<int> vis(n + 1);
auto dfs = [&](auto self, int x) -> void {
    vis[x] = 1;
    for (auto y : ver[x]) {
        if (vis[y] == 0) {
            self(self, y);
        } else if (vis[y] == 1) {
            ans++;
        }
    }
    vis[x] = 2;
};
for (int i = 1; i <= n; i++) {
    if (!vis[i]) {
        dfs(dfs, i);
    }
}
cout << ans << endl;
```

输出有向图任意一个环 原题: 给出一张有向图, 输出任意一个环, 数据包括二元环和自环. 使用 dfs 染色法.

```
vector<int> dis(n + 1), vis(n + 1), fa(n + 1);
auto dfs = [&](auto self, int x) -> void {
    vis[x] = 1;
    for (auto y : ver[x]) {
        if (vis[y] == 0) {
            dis[y] = dis[x] + 1;
            fa[y] = x;
            self(self, y);
        } else if (vis[y] == 1) {
```

```
            cout << dis[x] - dis[y] + 1 << endl;
            int pre = x;
            cout << pre << " ";
            while (pre != y) {
                pre = fa[pre];
                cout << pre << " ";
            }
            cout << endl;
            exit(0);
        }
    }
    vis[x] = 2;
};
for (int i = 1; i <= n; i++) {
    if (!vis[i]) {
        dfs(dfs, i);
    }
}
```

判定带环图是否是平面图 原题: 给定一个环以一些额外边, 对于每一条额外边判定其位于环外还是环内, 使得任意两条无重合顶点的额外边都不相交(即这张图构成平面图)See1, See2 .

使用 2-sat. 考虑全部边都位于环内, 那么 “一条边完全包含另一条边,” 两条边完全没有交集这两种情况都不会相交, 可以直接跳过这两种情况的讨论.

```
signed main() {
    int n, m;
    cin >> n >> m;
    vector<pair<int, int>> in(m);
    for (int i = 0, x, y; i < m; i++) {
        cin >> x >> y;
        in[i] = minmax(x, y);
    }
```

```
    TwoSat sat(m);
    for (int i = 0; i < m; i++) {
        auto [s, e] = in[i];
        for (int j = i + 1; j < m; j++) {
            auto [S, E] = in[j];
            if (s < S && S < e && e < E || S < s && s < E && E < e) {
                sat.add(i, 0, j, 0);
                sat.add(i, 1, j, 1);
            }
        }
    }
    if (!sat.work()) {
        cout << "Impossible\n";
        return 0;
    }
    auto ans = sat.answer();
    for (auto it : ans) {
        cout << (it ? "out" : "in") << " ";
    }
}
```

数论

常见数列

调和级数 满足调和级数 $\mathcal{O}\left(\frac{N}{1}+\frac{N}{2}+\frac{N}{3}+\cdots+\frac{N}{N}\right)$，可以用 $\approx N\ln N$ 来拟合，但是会略小，误差量级在 10% 左右. 本地可以在 500ms 内完成 10^8 量级的预处理计算.

N 的量级	1	2	3	4	5	6	7	8	9
累加和	27	482	7069	93668	1166750	13970034	162725364	1857511568	20877

下方示例为求解 1 到 N 中各个数字的因数值.

```
const int N = 1E5;
vector<vector<int>> dic(N + 1);
for (int i = 1; i <= N; i++) {
    for (int j = i; j <= N; j += i) {
        dic[j].push_back(i);
    }
}
```

素数密度与分布

N 的量级	1	2	3	4	5	6	7	8	9
素数数量	4	25	168	1229	9592	78498	664579	5761455	50847534

除此之外，对于任意两个相邻的素数 $p_1, p_2 \leq 10^9$，有 $|p_1 - p_2| < 300$ 成立，更具体的说，最大的差值为 282.

因数最多数字与其因数数量

N 的量级	1	2	3	4	5	6
因数最多数字的因数数量	4	25	32	64	128	240
因数最多的数字	-	-	-	7560, 9240	83160, 98280	720720, 831600, 942480, 98

欧拉筛 (线性筛)

时间复杂度为 $\mathcal{O}(N \log \log N)$.

```
vector<int> prime; // 这里储存筛出来的全部质数
auto euler_Prime = [&](int n) -> void {
    vector<int> v(n + 1);
    for (int i = 2; i <= n; ++i) {
        if (!v[i]) {
            v[i] = i;
            prime.push_back(i);
        }
        for (int j = 0; j < prime.size(); ++j) {
            if (prime[j] > v[i] || prime[j] > n / i) break;
            v[i * prime[j]] = prime[j];
        }
    }
};
```

最小质因数

```
std::vector<int> minp, primes;

void sieve(int n) {
    minp.assign(n + 1, 0);
    primes.clear();
```

```
    for (int i = 2; i <= n; i++) {
        if (minp[i] == 0) {
            minp[i] = i;
            primes.push_back(i);
        }

        for (auto p : primes) {
            if (i * p > n) {
                break;
            }
            minp[i * p] = p;
            if (p == minp[i]) {
                break;
            }
        }
    }
}
```

防爆模乘

借助浮点数实现 以 $\mathcal{O}(1)$ 计算 $a \cdot b \bmod p$, 由于不取模, 常数比 int128 法小很多. 其中 $1 \leq n, k, p \leq 10^{18}$.

```
int mul(int a, int b, int m) {
    int r = a * b - m * (int)(1.L / m * a * b);
    return r - m * (r >= m) + m * (r < 0);
}
```

借助 int128 实现

```
int mul(int a, int b, int m) {
    return (__int128)a * b % m;
}
```

威尔逊定理

1. 当且仅当 p 为素数时,$(p-1)!-1(\mod p)$
2. 当且仅当 p 为素数时,$(p-1)!p-1(\mod p)$
3. 若 p 为质数, 则 p 能被 $(p-1)!+1$ 整除
4. 当且仅当 p 为素数时,$p(p1)!+1$

裴蜀定理

> $ax+by=c\,(x\in Z,y\in Z)$ 成立的充要条件是 $gcd(a,b)c($ Z^* 表示正整数集).

逆定理 设 a,b 是不全为零的整数, 若 $d>0$ 是 a,b 的公因数, 且存在整数 x,y, 使得 $ax+by=d$, 则 $d=\gcd(a,b)$.

特殊地, 设 a,b 是不全为零的整数, 若存在整数 x,y, 使得 $ax+by=1$, 则 a,b 互质.

多个整数 裴蜀定理可以推广到 n 个整数的情形: 设 $a_1,a_2,\dots,a_n$ 是不全为零的整数, 则存在整数 $x_1,x_2,\dots,x_n$, 使得 $a_1x_1+a_2x_2+\cdots+a_nx_n=\gcd(a_1,a_2,\dots,a_n)$. 其逆定理也成立: 设 $a_1,a_2,\dots,a_n$ 是不全为零的整数,$d>0$ 是 $a_1,a_2,\dots,a_n$ 的公因数, 若存在整数 $x_1,x_2,\dots,x_n$, 使得 $a_1x_1+a_2x_2+\cdots+a_nx_n=d$, 则 $d=\gcd(a_1,a_2,\dots,a_n)$.

例题: 给定一个序列 a, 找到一个序列 x, 使得 $\sum_{i=1}^{n}a_ix_i$ 最小.

```
LL n, a, ans;
LL gcd(LL a, LL b){
    return b ? gcd(b, a % b) : a;
}
int main(){
    cin >> n;
```

```
    for (int i = 0; i < n; i ++ ){
        cin >> a;
        if (a < 0) a = -a;
        ans = gcd(ans, a);
    }
    cout << ans << "\n";
    return 0;
}
```

逆元

费马小定理解 (借助快速幂) 若 p 为素数,$\gcd(a,p)=1$, 则 $a^{p-1}\equiv 1 \pmod p$.

另一个形式: 对于任意整数 a, 有 $a^p \equiv a \pmod p$.

单次计算的复杂度即为快速幂的复杂度 $\mathcal{O}(\log X)$. 限制:MOD 必须是质数, 且需要满足 x 与 MOD 互质.

```
LL inv(LL x) { return mypow(x, mod - 2, mod);}
```

扩展欧几里得解 此方法的 MOD 没有限制, 复杂度为 $\mathcal{O}(\log X)$, 但是比快速幂法常数大一些.

```
int x, y;
int exgcd(int a, int b, int &x, int &y) { //扩展欧几里得算法
    if (b == 0) {
        x = 1, y = 0;
        return a; //到达递归边界开始向上一层返回
    }
    int r = exgcd(b, a % b, x, y);
    int temp = y; //把 x y 变成上一层的
    y = x - (a / b) * y;
    x = temp;
```

```
    return r; //得到 a b 的最大公因数
}
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;
}
```

离线求解: 线性递推解 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;
```

扩展欧几里得 exgcd

求解形如 $a \cdot x + b \cdot y = \gcd(a, b)$ 的不定方程的任意一组解.

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

例题: 求解二元一次不定方程 $A \cdot x + B \cdot y = C$.

```
auto clac = [&](int a, int b, int c) {
    int u = 1, v = 1;
    if (a < 0) { // 负数特判，但是没用经过例题测试
        a = -a;
```

```
        u = -1;
    }
    if (b < 0) {
        b = -b;
        v = -1;
    }

    int x, y, d = exgcd(a, b, x, y), ans;
    if (c % d != 0) { // 无整数解
        cout << -1 << "\n";
        return;
    }
    a /= d, b /= d, c /= d;
    x *= c, y *= c; // 得到可行解

    ans = (x % b + b - 1) % b + 1;
    auto [A, B] = pair{u * ans, v * (c - ans * a) / b}; // x 最小正整数 特解

    ans = (y % a + a - 1) % a + 1;
    auto [C, D] = pair{u * (c - ans * b) / a, v * ans}; // y 最小正整数 特解

    int num = (C - A) / b + 1; // xy 均为正整数 的 解的组数
};
```

离散对数 bsgs 与 exbsgs

以 $\mathcal{O}(\sqrt{P})$ 的复杂度求解 $a^x \equiv b(\mathrm{mod}P)$. 其中标准 BSGS 算法不能计算 a 与 MOD 互质的情况, 而 exbsgs 则可以.

```
namespace BSGS {
    LL a, b, p;
    map<LL, LL> f;
    inline LL gcd(LL a, LL b) { return b > 0 ? gcd(b, a % b) : a; }
```

```
inline LL ps(LL n, LL k, int p) {
    LL r = 1;
    for (; k; k >>= 1) {
        if (k & 1) r = r * n % p;
        n = n * n % p;
    }
    return r;
}
void exgcd(LL a, LL b, LL &x, LL &y) {
    if (!b) {
        x = 1, y = 0;
    } else {
        exgcd(b, a % b, x, y);
        LL t = x;
        x = y;
        y = t - a / b * y;
    }
}
LL inv(LL a, LL b) {
    LL x, y;
    exgcd(a, b, x, y);
    return (x % b + b) % b;
}
LL bsgs(LL a, LL b, LL p) {
    f.clear();
    int m = ceil(sqrt(p));
    b %= p;
    for (int i = 1; i <= m; i++) {
        b = b * a % p;
        f[b] = i;
    }
    LL tmp = ps(a, m, p);
    b = 1;
```

```
        for (int i = 1; i <= m; i++) {
            b = b * tmp % p;
            if (f.count(b) > 0) return (i * m - f[b] + p) % p;
        }
        return -1;
    }
    LL exbsgs(LL a, LL b, LL p) {
        if (b == 1 || p == 1) return 0;
        LL g = gcd(a, p), k = 0, na = 1;
        while (g > 1) {
            if (b % g != 0) return -1;
            k++;
            b /= g;
            p /= g;
            na = na * (a / g) % p;
            if (na == b) return k;
            g = gcd(a, p);
        }
        LL f = bsgs(a, b * inv(na, p) % p, p);
        if (f == -1) return -1;
        return f + k;
    }
} // namespace BSGS

using namespace BSGS;

int main() {
    IOS;
    cin >> p >> a >> b;
    a %= p, b %= p;
    LL ans = exbsgs(a, b, p);
    if (ans == -1) cout << "no solution\n";
    else cout << ans << "\n";
```

```
    return 0;
}
```

欧拉函数

直接求解单个数的欧拉函数 1 到 N 中与 N 互质数的个数称为欧拉函数, 记作 $\varphi(N)$. 求解欧拉函数的过程即为分解质因数的过程, 复杂度 $\mathcal{O}(\sqrt{n})$.

```
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
        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;
}
```

求解 1 到 N 所有数的欧拉函数 利用上述性质, 我们可以快速递推出 $2-N$ 中每个数的欧拉函数, 复杂度 $\mathcal{O}(N)$, 而该算法**即是线性筛的算法**.

$$\varphi(n) = (1-1/p_1)(1-1/p_2)(1-1/p_3)(1-1/p_4)\cdots(1-1/p_n);$$

```
const int N = 1e5 + 7;
int v[N], prime[N], phi[N];
void euler(int n) {
    ms(v, 0); //最小质因子
    int m = 0; //质数数量
    for (int i = 2; i <= n; ++ i) {
```

```
        if (v[i] == 0) { // i 是质数
            v[i] = i, prime[++ m] = i;
            phi[i] = i - 1;
        }
         //为当前的数 i 乘上一个质因子
        for (int j = 1; j <= m; ++ j) {
             //如 i 有比 prime[j] 更小的质因子，或超出 n ，停止
            if(prime[j] > v[i] || prime[j] > n / i) break;
             // prime[j] 是合数 i * prime[j] 的最小质因子
            v[i * prime[j]] = prime[j];
            phi[i * prime[j]] = phi[i] * (i % prime[j] ? prime[j] - 1 : prime[j]);
        }
    }
}
int main() {
    int n; cin >> n; euler(n);
    for (int i = 1; i <= n; ++ i) cout << phi[i] << endl;
    return 0;
}

std::vector<int> pri, not_prime, phi;

void init(int n) {
    not_prime.assign(n + 1, 0);
    phi.assign(n + 1, 0);
    phi[1] = 1;
    for (int i = 2; i <= n; i++) {
        if (!not_prime[i]) {
            pri.push_back(i);
            phi[i] = i - 1;
        }
        for (int pri_j : pri) {
            if (i * pri_j > n) break;
            not_prime[i * pri_j] = true;
```

```
            if (i % pri_j == 0) {
                phi[i * pri_j] = phi[i] * pri_j;
                break;
            }
            phi[i * pri_j] = phi[i] * phi[pri_j];
        }
    }
}
```

使用莫比乌斯反演求解欧拉函数

```
int phi[N];
vector<int> fac[N];
void get_eulers() {
    for (int i = 1; i <= N - 10; i++) {
        for (int j = i; j <= N - 10; j += i) {
            fac[j].push_back(i);
        }
    }
    phi[1] = 1;
    for (int i = 2; i <= N - 10; i++) {
        phi[i] = i;
        for (auto j : fac[i]) {
            if (j == i) continue;
            phi[i] -= phi[j];
        }
    }
}
```

扩展欧拉定理

若正整数 a 与 m 互质, 则

$$a^{\varphi(m)} \equiv 1 (\bmod m)$$

推论:

$$a^b \equiv a^{b \bmod \varphi(m)} (\bmod m)$$

当 a, m 不互质时, 扩展 Euler 定理表述如下:

$$a^b \equiv a^{b \bmod \varphi(m) + \varphi(m)} (\bmod m)$$

式子仅在 $\varphi(m) \leq b$ 时成立

```cpp
#include <bits/stdc++.h>
using namespace std;
bool large_enough = false; // 判断是否有 b >= phi(m)
inline int read(int MOD = 1e9 + 7) // 快速读入稍加修改即可以边读入边取模, 不取模
{
    int ans = 0;
    char c = getchar();
```

```
    while (!isdigit(c))
        c = getchar();
    while (isdigit(c))
    {
        ans = ans * 10 + c - '0';
        if (ans >= MOD)
        {
            ans %= MOD;
            large_enough = true;
        }
        c = getchar();
    }
    return ans;
}
int phi(int n) // 求欧拉函数
{
    int res = n;
    for (int i = 2; i * i <= n; i++)
    {
        if (n % i == 0)
            res = res / i * (i - 1);
        while (n % i == 0)
            n /= i;
    }
    if (n > 1)
        res = res / n * (n - 1);
    return res;
}
int qpow(int a, int n, int MOD) // 快速幂
{
    int ans = 1;
    while (n)
    {
```

```
        if (n & 1)
            ans = 1LL * ans * a % MOD; // 注意防止溢出
        n >>= 1;
        a = 1LL * a * a % MOD;
    }
    return ans;
}
int main()
{
    int a = read(), m = read(), phiM = phi(m), b = read(phiM);
    cout << qpow(a, b + (large_enough ? phiM : 0), m);
    return 0;
}
```

求解连续数字的正约数集合-倍数法

使用规律递推优化, 时间复杂度为 $\mathcal{O}(N \log N)$, 如果不需要详细的输出集合, 则直接将 vector 换为普通数组即可 (时间更快) .

```
#include <bits/stdc++.h>
using namespace std;
const int N = 1e5 + 7;
vector<int> f[N];

void divide(int n) {
    for (int i = 1; i <= n; ++ i)
        for (int j = 1; j <= n / i; ++ j)
            f[i * j].push_back(i);
    for (int i = 1; i <= n; ++ i) {
        for (auto it : f[i]) cout << it << " ";
        cout << endl;
    }
}
```

```
int main() {
    int x; cin >> x; divide(x);
    return 0;
}
```

试除法判是否是质数

标准解 $\mathcal{O}(\sqrt{N})$.

```
bool is_prime(int n) {
    if (n < 2) return false;
    for (int i = 2; i <= x / i; i++) {
        if (n % i == 0) return false;
    }
    return true;
}
```

常数优化法 常数优化, 达到 $\mathcal{O}(\frac{\sqrt{N}}{3})$.

```
bool is_prime(int n) {
    if (n < 2) return false;
    if (n == 2 || n == 3) return true;
    if (n % 6 != 1 && n % 6 != 5) return false;
    for (int i = 5, j = n / i; i <= j; i += 6) {
        if (n % i == 0 || n % (i + 2) == 0) {
            return false;
        }
    }
    return true;
}
```

同余方程组, 拓展中国剩余定理 excrt

公式:$x \equiv b_i (\mathrm{mod}\ a_i)$, 即 $(x - b_i) \mid a_i$.

```
int n; LL ai[maxn], bi[maxn];
inline int mypow(int n, int k, int p) {
    int r = 1;
    for (; k; k >>= 1, n = n * n % p)
        if (k & 1) r = r * n % p;
    return r;
}
LL exgcd(LL a, LL b, LL &x, LL &y) {
    if (b == 0) { x = 1, y = 0; return a; }
    LL gcd = exgcd(b, a % b, x, y), tp = x;
    x = y, y = tp - a / b * y;
    return gcd;
}
LL excrt() {
    LL x, y, k;
    LL M = bi[1], ans = ai[1];
    for (int i = 2; i <= n; ++ i) {
        LL a = M, b = bi[i], c = (ai[i] - ans % b + b) % b;
        LL gcd = exgcd(a, b, x, y), bg = b / gcd;
        if (c % gcd != 0) return -1;
        x = mul(x, c / gcd, bg);
        ans += x * M;
        M *= bg;
        ans = (ans % M + M) % M;
    }
    return (ans % M + M) % M;
}
int main() {
    cin >> n;
    for (int i = 1; i <= n; ++ i) cin >> bi[i] >> ai[i];
```

```
    cout << excrt() << endl;
    return 0;
}
```

求解连续按位异或

以 $\mathcal{O}(1)$ 复杂度计算 $0 \oplus 1 \oplus \cdots \oplus n$.

```
unsigned xor_n(unsigned n) {
    unsigned t = n & 3;
    if (t & 1) return t / 2u ^ 1;
    return t / 2u ^ n;
}
```

```
i64 xor_n(i64 n) {
    if (n % 4 == 1) return 1;
    else if (n % 4 == 2) return n + 1;
    else if (n % 4 == 3) return 0;
    else return n;
}
```

高斯消元求解线性方程组

题目大意: 输入一个包含 N 个方程 N 个未知数的线性方程组, 系数与常数均为实数 (两位小数). 求解这个方程组. 如果存在唯一解, 则输出所有 N 个未知数的解, 结果保留两位小数. 如果无数解, 则输出 `X` , 如果无解, 则输出 `N` .

```
const int N = 110;
const double eps = 1e-8;
LL n;
double a[N][N];
LL gauss(){
    LL c, r;
```

```
    for (c = 0, r = 0; c < n; c ++ ){
        LL t = r;
        for (int i = r; i < n; i ++ )    //找到绝对值最大的行
            if (fabs(a[i][c]) > fabs(a[t][c]))
                t = i;
        if (fabs(a[t][c]) < eps) continue;
        for (int j = c; j < n + 1; j ++ ) swap(a[t][j], a[r][j]);    //将绝对值最大的
        for (int j = n; j >= c; j -- ) a[r][j] /= a[r][c];    //将当前行首位变成 1
        for (int i = r + 1; i < n; i ++ )    //将下面列消成 0
            if (fabs(a[i][c]) > eps)
                for (int j = n; j >= c; j -- )
                    a[i][j] -= a[r][j] * a[i][c];
        r ++ ;
    }
    if (r < n){
        for (int i = r; i < n; i ++ )
            if (fabs(a[i][n]) > eps)
                return 2;
        return 1;
    }
    for (int i = n - 1; i >= 0; i -- )
        for (int j = i + 1; j < n; j ++ )
            a[i][n] -= a[i][j] * a[j][n];
    return 0;
}
int main(){
    cin >> n;
    for (int i = 0; i < n; i ++ )
        for (int j = 0; j < n + 1; j ++ )
            cin >> a[i][j];
    LL t = gauss();
    if (t == 0){
        for (int i = 0; i < n; i ++ ){
```

```
            if (fabs(a[i][n]) < eps) a[i][n] = abs(a[i][n]);
            printf("%.2lf\n", a[i][n]);
        }
    }
    else if (t == 1) cout << "Infinite group solutions\n";
    else cout << "No solution\n";
    return 0;
}
```

Min25 筛

求解 $1-N$ 的质数和, 其中 $N \leq 10^{10}$.

```
namespace min25{
    const int N = 1000000 + 10;
    int prime[N], id1[N], id2[N], flag[N], ncnt, m;
    LL g[N], sum[N], a[N], T;
    LL n;
    LL mod;
    inline LL ps(LL n,LL k) {LL r=1;for(;k;k>>=1){if(k&1)r=r*n%mod;n=n*n%mod;}return r;
    void finit(){ // 最开始清 0
        memset(g, 0, sizeof(g));
        memset(a, 0, sizeof(a));
        memset(sum, 0, sizeof(sum));
        memset(prime, 0, sizeof(prime));
        memset(id1, 0, sizeof(id1));
        memset(id2, 0, sizeof(id2));
        memset(flag, 0, sizeof(flag));
        ncnt = m = 0;
    }
    int ID(LL x) {
        return x <= T ? id1[x] : id2[n / x];
    }
```

```
    LL calc(LL x) {
        return x * (x + 1) / 2 - 1;
    }

    LL init(LL x) {
        T = sqrt(x + 0.5);
        for (int i = 2; i <= T; i++) {
            if (!flag[i]) prime[++ncnt] = i, sum[ncnt] = sum[ncnt - 1] + i;
            for (int j = 1; j <= ncnt && i * prime[j] <= T; j++) {
                flag[i * prime[j]] = 1;
                if (i % prime[j] == 0) break;
            }
        }
        for (LL l = 1; l <= x; l = x / (x / l) + 1) {
            a[++m] = x / l;
            if (a[m] <= T) id1[a[m]] = m; else id2[x / a[m]] = m;
            g[m] = calc(a[m]);
        }
        for (int i = 1; i <= ncnt; i++)
            for (int j = 1; j <= m && (LL) prime[i] * prime[i] <= a[j]; j++)
                g[j] = g[j] - (LL) prime[i] * (g[ID(a[j] / prime[i])] - sum[i - 1]);
    }
    LL solve(LL x) {
        if (x <= 1) return x;
        return n = x, init(n), g[ID(n)];
    }
}

using namespace min25;

int main() {
    // while (1) {
```

```
    int tt;
    scanf("%d",&tt);
    while(tt--){
        finit();
        scanf("%lld%lld", &n, &mod);
        LL ans = (n + 3) % mod * n % mod  * ps(2 , mod - 2) % mod + solve(n + 1) - 4;
        // cout << solve(n) << endl;
        // ans = (ans + mod) % mod;
        ans = (ans + mod) % mod;
        printf("%lld\n", ans);
    }

    // }
}
```

矩阵四则运算

封装来自 . 矩阵乘法复杂度 $\mathcal{O}(N^3)$.

```
const int SIZE = 2;
struct Matrix {
    ll M[SIZE + 5][SIZE + 5];
    void clear() { memset(M, 0, sizeof(M)); }
    void reset() { //初始化
        clear();
        for (int i = 1; i <= SIZE; ++i) M[i][i] = 1;
    }
    Matrix friend operator*(const Matrix &A, const Matrix &B) {
        Matrix Ans;
        Ans.clear();
        for (int i = 1; i <= SIZE; ++i)
            for (int j = 1; j <= SIZE; ++j)
                for (int k = 1; k <= SIZE; ++k)
```

```
                    Ans.M[i][j] = (Ans.M[i][j] + A.M[i][k] * B.M[k][j]) % mod;
        return Ans;
    }
    Matrix friend operator+(const Matrix &A, const Matrix &B) {
        Matrix Ans;
        Ans.clear();
        for (int i = 1; i <= SIZE; ++i)
            for (int j = 1; j <= SIZE; ++j)
                Ans.M[i][j] = (A.M[i][j] + B.M[i][j]) % mod;
        return Ans;
    }
};

inline int mypow(LL n, LL k, int p = MOD) {
    LL r = 1;
    for (; k; k >>= 1, n = n * n % p) {
        if (k & 1) r = r * n % p;
    }
    return r;
}
bool ok = 1;
Matrix getinv(Matrix a) { //矩阵求逆
    int n = SIZE, m = SIZE * 2;
    for (int i = 1; i <= n; i++) a.M[i][i + n] = 1;
    for (int i = 1; i <= n; i++) {
        int pos = i;
        for (int j = i + 1; j <= n; j++)
            if (abs(a.M[j][i]) > abs(a.M[pos][i])) pos = j;
        if (i != pos) swap(a.M[i], a.M[pos]);
        if (!a.M[i][i]) {
            puts("No Solution");
            ok = 0;
        }
```

```
        ll inv = q_pow(a.M[i][i], mod - 2);
        for (int j = 1; j <= n; j++)
            if (j != i) {
                ll mul = a.M[j][i] * inv % mod;
                for (int k = i; k <= m; k++)
                    a.M[j][k] = ((a.M[j][k] - a.M[i][k] * mul) % mod + mod) % mod;
            }
        for (int j = 1; j <= m; j++) a.M[i][j] = a.M[i][j] * inv % mod;
    }
    Matrix res;
    res.clear();
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= m; j++)
            res.M[i][j] = a.M[i][n + j];
    return res;
}
```

矩阵快速幂

以 $\mathcal{O}(N^3 \log M)$ 的复杂度计算.

```
const int N = 40;
using mat = std::array<std::array<i64, N + 1>, N + 1>;
mat operator*(const mat& a, const mat& b) {
    mat ans{};
    for (int i = 1; i <= N; i++) {
        for (int j = 1; j <= N; j++) {
            for (int k = 1; k <= N; k++)
                ans[i][j] = (ans[i][j] + a[i][k] * b[k][j]) % mod;
        }
    }
    return ans;
}
```

```
mat MatPow(mat a, i64 b) {
    mat ans{};
    for (int i = 1;i <= N;i++) ans[i][i] = 1;
    while (b) {
        if (b & 1) ans = ans * a;
        b >>= 1;
        a = a * a;
    }
    return ans;
}
```

矩阵加速

```
const int mod = 1e9 + 7;
LL T, n, t[5][5], a[5][5], b[5][5];
void matrixQp(LL y){
    while (y){
        if (y & 1){
            memset(t, 0, sizeof t);
            for (int i = 1; i <= 3; i ++ )
                for (int j = 1; j <= 1; j ++ )
                    for (int k = 1; k <= 3; k ++ )
                        t[i][j] = ( t[i][j] + (a[i][k] * b[k][j]) % mod ) % mod;
            memcpy(b, t, sizeof t);
        }
        y >>= 1;
        memset(t, 0, sizeof t);
        for (int i = 1; i <= 3; i ++ )
            for (int j = 1; j <= 3; j ++ )
                for (int k = 1; k <= 3; k ++ )
                    t[i][j] = ( t[i][j] + (a[i][k] * a[k][j]) % mod ) % mod;
        memcpy(a, t, sizeof t);
```

```
    }
}
void init(){
    b[1][1] = b[2][1] = b[3][1] = 1;
    memset(a, 0, sizeof a);
    a[1][1] = a[2][1] = a[1][3] = a[3][2] = 1;
}
void solve(){
    cin >> n;
    if (n <= 3) cout << "1\n";
    else{
        init();
        matrixQp(n - 3);
        cout << b[1][1] << "\n";
    }
}
int main(){
    cin >> T;
    while ( T -- )
        solve();
    return 0;
}
```

莫比乌斯函数/反演

莫比乌斯函数定义:$\mu(n) = \begin{cases} 1 & n = 1 \\ (-1)^k & n = \prod_{i=1}^{k} p_i \text{ 且} p_i \text{ 互质} \\ 0 & else \end{cases}$.

莫比乌斯函数性质: 对于任意正整数 n 满足

$$\sum_{d|n} \mu(d) = \begin{cases} 1 & n = 1 \\ 0 & n \neq 1 \end{cases}$$

$$;\sum_{d|n}\frac{\mu(d)}{d}=\frac{\varphi(n)}{n}\,.$$

莫比乌斯反演定义: 定义:$F(n)$ 和 $f(n)$ 是定义在非负整数集合上的两个函数, 并且满足 $F(n)=\sum_{d|n}f(d)$, 可得 $f(n)=\sum_{d|n}\mu(d)F(\left\lfloor\frac{n}{d}\right\rfloor)$.

```
const int N = 5e4 + 10;
bool st[N];
int mu[N], prime[N], cnt, sum[N];
void getMu() {
    mu[1] = 1;
    for (int i = 2; i <= N - 10; i++) {
        if (!st[i]) {
            prime[++cnt] = i;
            mu[i] = -1;
        }
        for (int j = 1; j <= cnt && i * prime[j] <= N - 10; j++) {
            st[i * prime[j]] = true;
            if (i % prime[j] == 0) {
                mu[i * prime[j]] = 0;
                break;
            }
            mu[i * prime[j]] = -mu[i];
        }
    }
    for (int i = 1; i <= N - 10; i++) {
        sum[i] = sum[i - 1] + mu[i];
    }
}
void solve() {
    int n, m, k; cin >> n >> m >> k;
    n = n / k, m = m / k;
    if (n < m) swap(n, m);
```

```
    LL ans = 0;
    for (int i = 1, j = 0; i <= m; i = j + 1) {
        j = min(n / (n / i), m / (m / i));
        ans += (LL)(sum[j] - sum[i - 1]) * (n / i) * (m / i);
    }
    cout << ans << "\n";
}
int main() {
    getMu();
    int T; cin >> T;
    while (T--) solve();
}
```

整除 (数论) 分块

$\left\lfloor \frac{n}{l} \right\rfloor = \left\lfloor \frac{n}{l+1} \right\rfloor = \ldots = \left\lfloor \frac{n}{r} \right\rfloor \iff \left\lfloor \frac{n}{l} \right\rfloor \le \frac{n}{r} < \left\lfloor \frac{n}{l} \right\rfloor + 1$, 根据不等式左侧, 得到 $r \le \left\lfloor \frac{n}{\lfloor \frac{n}{l} \rfloor} \right\rfloor$.

```
void solve() {
    LL n; cin >> n;
    LL ans = 0;
    for (LL i = 1, j; i <= n; i = j + 1) {
        j = n / (n / i);
        ans += (LL)(j - i + 1) * (n / i);
    }
    cout << ans << "\n";
}
int main() {
    int T; cin >> T;
    while (T--) solve();
}
```

Miller - Rabin 素数测试

以平均 $\mathcal{O}(4 \cdot \log^3 X)$ 的复杂度判定数字 X 是否是素数, 这里记录的版本复杂度非常优秀, 基本可以看作是 $\mathcal{O}(1)$.

```
int mul(int a, int b, int m) {
    int r = a * b - m * (int)(1.L / m * a * b);
    return r - m * (r >= m) + m * (r < 0);
}
int mypow(int a, int b, int m) {
    int res = 1 % m;
    for (; b; b >>= 1, a = mul(a, a, m)) {
        if (b & 1) {
            res = mul(res, a, m);
        }
    }
    return res;
}

int B[] = {2, 3, 5, 7, 11, 13, 17, 19, 23};
bool MR(int n) {
    if (n <= 1) return 0;
    for (int p : B) {
        if (n == p) return 1;
        if (n % p == 0) return 0;
    }
    int m = (n - 1) >> __builtin_ctz(n - 1);
    for (int p : B) {
        int t = m, a = mypow(p, m, n);
        while (t != n - 1 && a != 1 && a != n - 1) {
            a = mul(a, a, n);
            t *= 2;
        }
        if (a != n - 1 && t % 2 == 0) return 0;
```

```
    }
    return 1;
}
```

Pollard - Rho 因式分解

以单个因子 $\mathcal{O}(\log X)$ 的复杂度输出数字 X 的全部质因数, 由于需要结合素数测试, 总复杂度会略高一些. 如果遇到超时的情况, 可能需要考虑进一步优化, 例如检查题目是否强制要求枚举全部质因数等等. 此外, 还有一个较长的模板可供参考, 比这里记录的版本常数小约五倍.

```
int PR(int n) {
    for (int p : B) {
        if (n % p == 0) return p;
    }
    auto f = [&](int x) -> int {
        x = mul(x, x, n) + 1;
        return x >= n ? x - n : x;
    };
    int x = 0, y = 0, tot = 0, p = 1, q, g;
    for (int i = 0; (i & 255) || (g = gcd(p, n)) == 1; i++, x = f(x), y = f(f(y))) {
        if (x == y) {
            x = tot++;
            y = f(x);
        }
        q = mul(p, abs(x - y), n);
        if (q) p = q;
    }
    return g;
}
vector<int> fac(int n) {
    #define pb emplace_back
    if (n == 1) return {};
```

```
    if (MR(n)) return {n};
    int d = PR(n);
    auto v1 = fac(d), v2 = fac(n / d);
    auto i1 = v1.begin(), i2 = v2.begin();
    vector<int> ans;
    while (i1 != v1.end() || i2 != v2.end()) {
        if (i1 == v1.end()) {
            ans.pb(*i2++);
        } else if (i2 == v2.end()) {
            ans.pb(*i1++);
        } else {
            if (*i1 < *i2) {
                ans.pb(*i1++);
            } else {
                ans.pb(*i2++);
            }
        }
    }
    return ans;
}
```

常见结论和定理

麦乐鸡定理 给定两个互质的数 n, m , 定义 $x = a * n + b * m (a \geq 0, b \geq 0)$, 当 $x > n * m - n - m$ 时, 该式子恒成立.

抽屉原理 (鸽巢原理) 将 $n + 1$ 个物体, 划分为 n 组, 那么有至少一组有两个 (或以上) 的物体.

哥德巴赫猜想 任何一个大于 5 的整数都可写成三个质数之和; 任何一个大于 2 的偶数都可写成两个素数之和.

除法, 取模运算的本质 有公式:$x \div i = \left\lfloor \frac{x}{i} \right\rfloor + x - i \cdot \left\lfloor \frac{x}{i} \right\rfloor$,$x \mod i = x - i \cdot \left\lfloor \frac{x}{i} \right\rfloor$.

与, 或, 异或

运算	运算符, 数学符号表示	解释
与	&,and	同 1 出 1
或	|,or	有 1 出 1
异或	^,$\bigoplus$,xor	不同出 1

一些结论:

> 对于给定的 X 和序列 $[a_1, a_2, , a_n]$, 有:$\boldsymbol{X = (X \& a_1) or (X \& a_2) or or (X \& a_n)}$. 原理是 and 意味着取交集,or 意味着取子集.来源 - 牛客小白月赛 49C

调和级数近似公式

```
log(n) + 0.5772156649 + 1.0 / (2 * n)
```

欧拉函数常见性质

- $1 - n$ 中与 n 互质的数之和为 $n * \varphi(n)/2$.
- 若 a, b 互质, 则 $\varphi(a * b) = \varphi(a) * \varphi(b)$. 实际上, 所有满足这一条件的函数统称为积性函数.
- 若 f 是积性函数, 且有 $n = \prod_{i=1}^{m} p_i^{c_i}$, 那么 $f(n) = \prod_{i=1}^{m} f(p_i^{c_i})$.
- 若 p 为质数, 且满足 $p \mid n$,
  - $p^2 \mid n$, 那么 $\varphi(n) = \varphi(n/p) * p$.

– $p^2 \nmid n$, 那么 $\varphi(n) = \varphi(n/p) * (p-1)$.

- $\sum_{d|n} \varphi(d) = n$.

  如 $n = 10$, 则 $d = 10/5/2/1$, 那么 $10 = \varphi(10) + \varphi(5) + \varphi(2) + \varphi(1)$.

- $\sum_{i=1}^{n} \mathsf{gcd}(i,n) = \sum_{d|n} \left\lfloor \frac{n}{d} \right\rfloor \varphi(d)$ (欧拉反演).

狄利克雷卷积 $\sum_{d|n} \varphi(d) = n$,$\sum_{d|n} \mu(d)\frac{n}{d} = \varphi(n)$.

斐波那契数列 通项公式:$F_n = \frac{1}{\sqrt{5}} * \left[\left(\frac{1+\sqrt{5}}{2} \right)^n - \left(\frac{1-\sqrt{5}}{2} \right)^n \right]$.

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 $\leq 6m$ ($m = 2 \times 5^k$ 时取到等号);
- 既是斐波那契数又是平方数的有且仅有 $1, 144$.

杂

- 负数取模得到的是负数, 如果要用 $0/1$ 判断的话请取绝对值;
- 辗转相除法原式为 $\gcd(x, y) = \gcd(x, y - x)$, 推广到 N 项为 $\gcd(a_1, a_2, \dots, a_N) = \gcd(a_1, a_2 - a_1, \dots, a_N - a_{N-1})$,
  - 该推论在" 四则运算后 gcd 这类题中有特殊意义, 如求解 $\gcd(a_1 + X, a_2 + X, \dots, a_N + X)$ 时See;
- 以下式子成立: $\gcd(a, m) = \gcd(a + x, m) \Leftrightarrow \gcd(a, m) = \gcd(x, m)$. 求解上式满足条件的 x 的数量即为求比 $\dfrac{m}{\gcd(a, m)}$ 小且与其互质的数的个数, 即用欧拉函数求解 $\varphi\left(\dfrac{m}{\gcd(a, m)}\right)$.
- 已知序列 a , 定义集合 $S = \{a_i \cdot a_j \mid i < j\}$, 现在要求解 $\gcd(S)$, 即为求解 $\gcd(a_j, \gcd(a_i \mid i < j))$, 换句话说, 即为求解后缀 gcd .
- 连续四个数互质的情况如下, 当 n 为奇数时,$n, n-1, n-2$ 一定互质; 而当 n 为偶数时,$\begin{cases} n, n-1, n-3\text{互质} & \gcd(n, n-3) = 1\text{时} \\ n-1, n-2, n-3\text{互质} & \gcd(n, n-3) \neq 1\text{时} \end{cases}$ See;

- 由 $a \mod b = (b+a) \mod b = (2 \cdot b + a) \mod b = \cdots = (K \cdot b + a) \mod b$ 可以推广得到 $(a \mod b) \mod c = ((K \cdot bc + a) \mod b) \mod c$, 由此可以得到一个 bc 的答案周期See;

- 对于长度为 $2 \cdot N$ 的数列 a , 将其任意均分为两个长度为 N 的数列 p, q , 随后对 p 非递减排序, 对 q 非递增排序, 定义 $f(p,q) = \sum_{i=1}^{n} |p_i - q_i|$, 那么答案为 a 数列前 N 大的数之和减去前 N 小的数之和See.

- 令 $\begin{cases} X = a + b \\ Y = a \oplus b \end{cases}$, **如果**该式子**有解**, 那么存在前提条件 $\begin{cases} X \geq Y \\ X, Y\text{同奇偶} \end{cases}$; 进一步, 此时最小的 a 的取值为 $\dfrac{X-Y}{2}$ See.

  然而, 上方方程并不总是有解的, 只有当变量增加到三个时, 才**一定有解**, 即: **在保证上方前提条件成立的情况下**, 求解 $\begin{cases} X = a + b + c \\ Y = a \oplus b \oplus c \end{cases}$, 则一定存在一组解 $\{\dfrac{X-Y}{2}, \dfrac{X-Y}{2}, Y\}$ See.

- 已知序列 p 是由序列 a_1 , 序列 a_2 „ 序列 a_n 合并而成, 且合并过程中各序列内元素相对顺序不变, 记 $T(p)$ 是 p 序列的最大前缀和, 则 $T(p) = \sum_{i=1}^{n} T(a_i)$ See .

- $x + y = x|y + x\&y$, 对于两个数字 x 和 y , 如果将 x 变为 $x|y$, 同时将 y 变为 $x\&y$, 那么在本质上即将 x 二进制模式下的全部 1 移动到了 y 的对应的位置上 See .

- 一个正整数 x 异或, 加上另一个正整数 y 后奇偶性不发生变化:$a+b \equiv a \oplus b (\mathrm{mod} 2)$ See .

常见例题

题意: 将 1 至 N 的每个数字分组, 使得每一组的数字之和均为质数. 输出每一个数字所在的组别, 且要求分出的组数最少 See .

考察哥德巴赫猜想, 记全部数字之和为 S , 分类讨论如下:

- 为 S 质数时, 只需要分入同一组;
- 当 S 为偶数时, 由猜想可知一定能分成两个质数, 可以证明其中较小的那个一定小于 N , 暴力枚举分组;
- 当 $S-2$ 为质数时, 特殊判断出答案;
- 其余情况一定能被分成三组, 其中 3 单独成组,$S-3$ 后成为偶数, 重复讨论二的过程即可.

题意: 给定一个长度为 n 的数组, 定义这个数组是 BAD 的, 当且仅当可以把数组分成两个子序列, 这两个子序列的元素之和相等. 现在你需要删除**最少的**元素, 使得删除后的数组不是 BAD 的.

最少删除一个元素-如果原数组存在奇数, 则直接删除这个奇数即可; 反之, 我们发现, 对数列同除以一个数不影响计算, 故我们只需要找到最大的满足 $2^k \mid a_i$ 成立的 2^k , 随后将全部的 a_i 变为 $\frac{a_i}{2^k}$, 此时一定有一个奇数 (换句话说, 我们可以对原数列的每一个元素不断的除以 2 直到出现奇数为止), 删除这个奇数即可 See .

题意: 设当前有一个数字为 x , 减去, 加上最少的数字使得其能被 k 整除.

最少减去 $x \bmod k$ 这个很好想; 最少加上 $\left(\left\lceil\frac{x}{k}\right\rceil * k\right) \bmod k$ 也比较好想, 但是更简便的方法为加上 $k-x \bmod k$, 这个式子等价于前面这一坨.

题意: 给定一个整数 n , 用恰好 k 个 2 的幂次数之和表示它. 例如:$n=9, k=4$, 答案为 $1+2+2+4$.

结论 1:k 合法当且仅当 `__builtin_popcountll(n) <= k && k <= n` , 显然.

结论 2:$2^{k+1} = 2 \cdot 2^k$, 所以我们可以将二进制位看作是数组, 然后从高位向低位推, 一个高位等于两个低位, 直到数组之和恰好等于 k , 随后依次输出即可. 举例说明,$\{1,0,0,1\} \to \{0,2,0,1\} \to \{0,1,2,1\}$, 即答案为 0 个 2^3 ,1 个 2^2 ,.

```
signed main() {
    int n, k;
    cin >> n >> k;

    int cnt = __builtin_popcountll(n);

    if (k < cnt || n < k) {
        cout << "NO\n";
        return 0;
    }
    cout << "YES\n";

    vector<int> num;
    while (n) {
        num.push_back(n % 2);
        n /= 2;
    }

    for (int i = num.size() - 1; i > 0; i--) {
        int p = min(k - cnt, num[i]);
        num[i] -= p;
        num[i - 1] += 2 * p;
        cnt += p;
    }

    for (int i = 0; i < num.size(); i++) {
```

```
        for (int j = 1; j <= num[i]; j++) {
            cout << (1LL << i) << " ";
        }
    }
}
```

题意:n 个取值在 $[0,k)$ 之间的数之和为 m 的方案数

答案为 $\sum_{i=0}^{n} -1^i \cdot \binom{n}{i} \cdot \binom{m-i\cdot k+n-1}{n-1}$ See1 See2.

```
Z clac(int n, int k, int m) {
    Z ans = 0;
    for(int i = 0; i <= n; ++i) {
        ans += C(n, i) * C(m - i * k + n - 1, n - 1) * pow(-1, i);
    }
    return ans;
}
```

[1] 先考虑没有 k 的限制, 那么即球盒模型:m 个球放入 n 个盒子, 球同, 盒子不同, 能空. 使用隔板法得到公式:`C(m + n - 1, n - 1)`;[2] 下面加上取值范围后进一步考虑: 假设现在 n 个数之和为 $m-k$, 运用上述隔板法可得公式:`C(m - k + n - 1, n - 1)`;[3] 随后, 选择任意一个数字, 将其加上 k, 这样, 这个数字一定不满足条件, 选法为:`C(n, 1)`;[4] 此时, 至少有一个数字是不满足条件的, 按照一般流程, 到这里,`C(m + n - 1, n - 1) - C(n, 1) * C(m - k + n - 1, n - 1)` 即是答案; 但是, 这样的操作会导致重复的部分, 所以这里要使用容斥原理将重复部分去除 (关于为什么会重复, 试比较概率论中的加法公式).

约瑟夫问题

n 个人编号 $0,1,2,n-1$, 每次数到 k 出局, 求最后剩下的人的编号.

$\mathcal{O}(N)$.

```
int jos(int n,int k){
    int res=0;
    repeat(i,1,n+1)res=(res+k)%i;
    return res; // res+1, 如果编号从 1 开始
}
```

$\mathcal{O}(K \log N)$, 适用于 K 较小的情况.

```
int jos(int n,int k){
    if(n==1 || k==1)return n-1;
    if(k>n)return (jos(n-1,k)+k)%n; // 线性算法
    int res=jos(n-n/k,k)-n%k;
    if(res<0)res+=n; // mod n
    else res+=res/(k-1); // 还原位置
    return res; // res+1, 如果编号从 1 开始
}
```

$\mathcal{O}(\sqrt{N})$

```
void jos(){
    int64_t n, k, a{}, b{ 1 }; cin >> n >> k; --k;
    while (b < n) {
        auto s = a / k + 1, u = b / k + 1, v = min(k - a / s, (min(u * k, n) - b + u -
        a += s * v, b += u * v;
    }
    cout << a + 1 << '\n';
}
```

线性代数

线性基

求 n 个数的值为 k 的方案数. 如果 k 不能被早已出来, 那么为 0 . 否则为 2^{n-m}(m 为线性基中的元素个数). 证明: 考虑不是线性基中的 n 个数, 对于每一个子集, 都可以在线性基中找到唯一一种方案使得异或和为 k .

```
std::vector<i64> get_linear_basis(std::vector<i64>& nums, int N = 63) {
    std::vector<i64> p(N + 1);
    auto insert = [&](i64 x) {
        for (int s = N;s >= 0;--s)if (x >> s & 1) {
            if (!p[s]) {
                p[s] = x;
                break;
            }
            x ^= p[s];
        }
        };
    for (auto& x : nums) insert(x);
    return p;
}

signed main() {
```

```
    std::ios::sync_with_stdio(false);
    std::cin.tie(0), std::cout.tie(0);
    int n;std::cin >> n;
    std::vector<i64> nums(n);
    for (auto& x : nums)std::cin >> x;
    auto p = get_linear_basis(nums, 63);
    i64 ans = 0;
    for (int s = N;s >= 0;--s)
        ans = std::max(ans, ans ^ p[s]);
    std::cout << ans;
    return 0;
}
```

高斯消元法 设向量长度为 N(一般取 63), 总数为 M, 时间复杂度为 $\mathcal{O}(NM)$.

```
struct LB { // Linear Basis
    using i64 = long long;
    const int BASE = 63;
    std::vector<i64> d, p;
    int cnt, flag;

    LB() {
        d.resize(BASE + 1);
        p.resize(BASE + 1);
        cnt = flag = 0;
    }
    bool insert(i64 val) {
        for (int i = BASE - 1; i >= 0; i--) {
            if (val & (1ll << i)) {
                if (!d[i]) {
                    d[i] = val;
                    return true;
```

```
            }
            val ^= d[i];
        }
    }
    flag = 1; //可以异或出 0
    return false;
}
bool check(i64 val) { // 判断 val 是否能被异或得到
    for (int i = BASE - 1; i >= 0; i--) {
        if (val & (1ll << i)) {
            if (!d[i]) {
                return false;
            }
            val ^= d[i];
        }
    }
    return true;
}
i64 ask_max() {
    i64 res = 0;
    for (int i = BASE - 1; i >= 0; i--) {
        if ((res ^ d[i]) > res) res ^= d[i];
    }
    return res;
}
i64 ask_min() {
    if (flag) return 0; // 特判 0
    for (int i = 0; i <= BASE - 1; i++) {
        if (d[i]) return d[i];
    }
}
void rebuild() { // 第 k 小值独立预处理
    for (int i = BASE - 1; i >= 0; i--) {
```

```
            for (int j = i - 1; j >= 0; j--) {
                if (d[i] & (1ll << j)) d[i] ^= d[j];
            }
        }
        for (int i = 0; i <= BASE - 1; i++) {
            if (d[i]) p[cnt++] = d[i];
        }
    }
    i64 kthquery(i64 k) { // 查询能被异或得到的第 k 小值，如不存在则返回 -1
        if (flag) k--; // 特判 0，如果不需要 0，直接删去
        if (!k) return 0;
        i64 res = 0;
        if (k >= (1ll << cnt)) return -1;
        for (int i = BASE - 1; i >= 0; i--) {
            if (k & (1LL << i)) res ^= p[i];
        }
        return res;
    }
    void Merge(const LB &b) { // 合并两个线性基
        for (int i = BASE - 1; i >= 0; i--) {
            if (b.d[i]) {
                insert(b.d[i]);
            }
        }
    }
};
```

三角形面积

行列式求面积

$$S = \frac{1}{2}\begin{vmatrix} 1 & 1 & 1 \\ x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \end{vmatrix}$$

```
int main(){
    float num[6];
    for(int i = 0; i < 6; i++)
        cin >> num[i];
    float sum = 0.0;
    sum = 0.5*(num[0]*num[3]+num[2]*num[5]+num[4]*num[1]-num[0]*num[5]-num[2]*num[1]-nu
    cout << " 三角形的面积为: ";
    sum == 0 ? cout << "Impossible" : cout <<sum;
    return 0;
}
```

海伦公式

$$S = \frac{1}{4}\sqrt{(a+b+c)(a+b-c)(a+c-b)(b+c-a)}$$

```
p=(a+b+c)/2;
sum=sqrt(p*(p-a)*(p-b)*(p-c));
```

多项式

线性凸包

```
struct Line {
  i64 a, b, r;
  bool operator<(Line l) { return pair(a, b) > pair(l.a, l.b); }
  bool operator<(i64 x) { return r < x; }
};
struct Lines : vector<Line> {
  static constexpr i64 inf = numeric_limits<i64>::max();
  Lines(i64 a, i64 b) : vector<Line>{{a, b, inf}} {}
  Lines(vector<Line>& lines) {
    if (not ranges::is_sorted(lines, less())) ranges::sort(lines, less());
    for (auto [a, b, _] : lines) {
      for (; not empty(); pop_back()) {
        if (back().a == a) continue;
        i64 da = back().a - a, db = b - back().b;
        back().r = db / da - (db < 0 and db % da);
        if (size() == 1 or back().r > end()[-2].r) break;
      }
      emplace_back(a, b, inf);
    }
  }
  Lines operator+(Lines& lines) {
    vector<Line> res(size() + lines.size());
    ranges::merge(*this, lines, res.begin(), less());
    return Lines(res);
```

```
  }
  i64 min(i64 x) {
    auto [a, b, _] = *lower_bound(begin(), end(), x, less());
    return a * x + b;
  }
};
```

多项式封装

```
template<int P = 998244353> struct Poly : public vector<MInt<P>> {
    using Value = MInt<P>;

    Poly() : vector<Value>() {}
    explicit constexpr Poly(int n) : vector<Value>(n) {}

    explicit constexpr Poly(const vector<Value> &a) : vector<Value>(a) {}
    constexpr Poly(const initializer_list<Value> &a) : vector<Value>(a) {}

    template<typename InputIt, typename = _RequireInputIter<InputIt>>
    explicit constexpr Poly(InputIt first, InputIt last) : vector<Value>(first, last) {

    template<typename F> F>plicit constexpr Poly(int n, F f) : vector<Value>(n) {
        for (int i = 0; i < n; i++) {
            (*this)[i] = f(i);
        }
    }

    constexpr Poly shift(int k) const {
        if (k >= 0) {
            auto b = *this;
            b.insert(b.begin(), k, 0);
            return b;
        } else if (this->size() <= -k) {
```

```
            return Poly();
        } else {
            return Poly(this->begin() + (-k), this->end());
        }
    }
    constexpr Poly trunc(int k) const {
        Poly f = *this;
        f.resize(k);
        return f;
    }
    constexpr friend Poly operator+(const Poly &a, const Poly &b) {
        Poly res(max(a.size(), b.size()));
        for (int i = 0; i < a.size(); i++) {
            res[i] += a[i];
        }
        for (int i = 0; i < b.size(); i++) {
            res[i] += b[i];
        }
        return res;
    }
    constexpr friend Poly operator-(const Poly &a, const Poly &b) {
        Poly res(max(a.size(), b.size()));
        for (int i = 0; i < a.size(); i++) {
            res[i] += a[i];
        }
        for (int i = 0; i < b.size(); i++) {
            res[i] -= b[i];
        }
        return res;
    }
    constexpr friend Poly operator-(const Poly &a) {
        vector<Value> res(a.size());
        for (int i = 0; i < int(res.size()); i++) {
```

```
            res[i] = -a[i];
        }
        return Poly(res);
    }
    constexpr friend Poly operator*(Poly a, Poly b) {
        if (a.size() == 0 || b.size() == 0) {
            return Poly();
        }
        if (a.size() < b.size()) {
            swap(a, b);
        }
        int n = 1, tot = a.size() + b.size() - 1;
        while (n < tot) {
            n *= 2;
        }
        if (((P - 1) & (n - 1)) != 0 || b.size() < 128) {
            Poly c(a.size() + b.size() - 1);
            for (int i = 0; i < a.size(); i++) {
                for (int j = 0; j < b.size(); j++) {
                    c[i + j] += a[i] * b[j];
                }
            }
            return c;
        }
        a.resize(n);
        b.resize(n);
        dft(a);
        dft(b);
        for (int i = 0; i < n; ++i) {
            a[i] *= b[i];
        }
        idft(a);
        a.resize(tot);
```

```
        return a;
    }
    constexpr friend Poly operator*(Value a, Poly b) {
        for (int i = 0; i < int(b.size()); i++) {
            b[i] *= a;
        }
        return b;
    }
    constexpr friend Poly operator*(Poly a, Value b) {
        for (int i = 0; i < int(a.size()); i++) {
            a[i] *= b;
        }
        return a;
    }
    constexpr friend Poly operator/(Poly a, Value b) {
        for (int i = 0; i < int(a.size()); i++) {
            a[i] /= b;
        }
        return a;
    }
    constexpr Poly &operator+=(Poly b) {
        return (*this) = (*this) + b;
    }
    constexpr Poly &operator-=(Poly b) {
        return (*this) = (*this) - b;
    }
    constexpr Poly &operator*=(Poly b) {
        return (*this) = (*this) * b;
    }
    constexpr Poly &operator*=(Value b) {
        return (*this) = (*this) * b;
    }
    constexpr Poly &operator/=(Value b) {
```

```
        return (*this) = (*this) / b;
    }
    constexpr Poly deriv() const {
        if (this->empty()) {
            return Poly();
        }
        Poly res(this->size() - 1);
        for (int i = 0; i < this->size() - 1; ++i) {
            res[i] = (i + 1) * (*this)[i + 1];
        }
        return res;
    }
    constexpr Poly integr() const {
        Poly res(this->size() + 1);
        for (int i = 0; i < this->size(); ++i) {
            res[i + 1] = (*this)[i] / (i + 1);
        }
        return res;
    }
    constexpr Poly inv(int m) const {
        Poly x{(*this)[0].inv()};
        int k = 1;
        while (k < m) {
            k *= 2;
            x = (x * (Poly{2} - trunc(k) * x)).trunc(k);
        }
        return x.trunc(m);
    }
    constexpr Poly log(int m) const {
        return (deriv() * inv(m)).integr().trunc(m);
    }
    constexpr Poly exp(int m) const {
        Poly x{1};
```

```
        int k = 1;
        while (k < m) {
            k *= 2;
            x = (x * (Poly{1} - x.log(k) + trunc(k))).trunc(k);
        }
        return x.trunc(m);
    }
    constexpr Poly pow(int k, int m) const {
        int i = 0;
        while (i < this->size() && (*this)[i] == 0) {
            i++;
        }
        if (i == this->size() || 1LL * i * k >= m) {
            return Poly(m);
        }
        Value v = (*this)[i];
        auto f = shift(-i) * v.inv();
        return (f.log(m - i * k) * k).exp(m - i * k).shift(i * k) * power(v, k);
    }
    constexpr Poly sqrt(int m) const {
        Poly x{1};
        int k = 1;
        while (k < m) {
            k *= 2;
            x = (x + (trunc(k) * x.inv(k)).trunc(k)) * CInv<2, P>;
        }
        return x.trunc(m);
    }
    constexpr Poly mulT(Poly b) const {
        if (b.size() == 0) {
            return Poly();
        }
        int n = b.size();
```

```
        reverse(b.begin(), b.end());
        return ((*this) * b).shift(-(n - 1));
    }
    constexpr vector<Value> eval(vector<Value> x) const {
        if (this->size() == 0) {
            return vector<Value>(x.size(), 0);
        }
        const int n = max(x.size(), this->size());
        vector<Poly> q(4 * n);
        vector<Value> ans(x.size());
        x.resize(n);
        function<void(int, int, int)> build = [&](int p, int l, int r) {
            if (r - l == 1) {
                q[p] = Poly{1, -x[l]};
            } else {
                int m = (l + r) / 2;
                build(2 * p, l, m);
                build(2 * p + 1, m, r);
                q[p] = q[2 * p] * q[2 * p + 1];
            }
        };
        build(1, 0, n);
        function<void(int, int, int, const Poly &)> work = [&](int p, int l, int r,
                                                               const Poly &num) {
            if (r - l == 1) {
                if (l < int(ans.size())) {
                    ans[l] = num[0];
                }
            } else {
                int m = (l + r) / 2;
                work(2 * p, l, m, num.mulT(q[2 * p + 1]).resize(m - l));
                work(2 * p + 1, m, r, num.mulT(q[2 * p]).resize(r - m));
            }
```

```
        };
        work(1, 0, n, mulT(q[1].inv(n)));
        return ans;
    }
};
```

离散傅里叶变换 dft 与其逆变换 idft

```
vector<int> rev;
template<int P> vector<MInt<P>> roots{0, 1};

template<int P> constexpr MInt<P> findPrimitiveRoot() {
    MInt<P> i = 2;
    int k = __builtin_ctz(P - 1);
    while (true) {
        if (power(i, (P - 1) / 2) != 1) {
            break;
        }
        i += 1;
    }
    return power(i, (P - 1) >> k);
}

template<int P> constexpr MInt<P> primitiveRoot = findPrimitiveRoot<P>();
template<> constexpr MInt<998244353> primitiveRoot<998244353>{31};

template<int P> constexpr void dft(vector<MInt<P>> &a) { // 离散傅里叶变换
    int n = a.size();

    if (int(rev.size()) != n) {
        int k = __builtin_ctz(n) - 1;
        rev.resize(n);
        for (int i = 0; i < n; i++) {
```

```
            rev[i] = rev[i >> 1] >> 1 | (i & 1) << k;
        }
    }

    for (int i = 0; i < n; i++) {
        if (rev[i] < i) {
            swap(a[i], a[rev[i]]);
        }
    }
    if (roots<P>.size() < n) {
        int k = __builtin_ctz(roots<P>.size());
        roots<P>.resize(n);
        while ((1 << k) < n) {
            auto e = power(primitiveRoot<P>, 1 << (__builtin_ctz(P - 1) - k - 1));
            for (int i = 1 << (k - 1); i < (1 << k); i++) {
                roots<P>[2 * i] = roots<P>[i];
                roots<P>[2 * i + 1] = roots<P>[i] * e;
            }
            k++;
        }
    }
    for (int k = 1; k < n; k *= 2) {
        for (int i = 0; i < n; i += 2 * k) {
            for (int j = 0; j < k; j++) {
                MInt<P> u = a[i + j];
                MInt<P> v = a[i + j + k] * roots<P>[k + j];
                a[i + j] = u + v;
                a[i + j + k] = u - v;
            }
        }
    }
}
template<int P> constexpr void idft(vector<MInt<P>> &a) { // 逆变换
```

```
    int n = a.size();
    reverse(a.begin() + 1, a.end());
    dft(a);
    MInt<P> inv = (1 - P) / n;
    for (int i = 0; i < n; i++) {
        a[i] *= inv;
    }
}
```

Berlekamp-Massey 算法 (杜教筛)

求解数列的最短线性递推式, 最坏复杂度为 $\mathcal{O}(NM)$, 其中 N 为数列长度,M 为它的最短递推式的阶数.

```
template<int P = 998244353> Poly<P> berlekampMassey(const Poly<P> &s) {
    Poly<P> c;
    Poly<P> oldC;
    int f = -1;
    for (int i = 0; i < s.size(); i++) {
        auto delta = s[i];
        for (int j = 1; j <= c.size(); j++) {
            delta -= c[j - 1] * s[i - j];
        }
        if (delta == 0) {
            continue;
        }
        if (f == -1) {
            c.resize(i + 1);
            f = i;
        } else {
            auto d = oldC;
            d *= -1;
            d.insert(d.begin(), 1);
```

```
            MInt<P> df1 = 0;
            for (int j = 1; j <= d.size(); j++) {
                df1 += d[j - 1] * s[f + 1 - j];
            }
            assert(df1 != 0);
            auto coef = delta / df1;
            d *= coef;
            Poly<P> zeros(i - f - 1);
            zeros.insert(zeros.end(), d.begin(), d.end());
            d = zeros;
            auto temp = c;
            c += d;
            if (i - temp.size() > f - oldC.size()) {
                oldC = temp;
                f = i;
            }
        }
    }
    c *= -1;
    c.insert(c.begin(), 1);
    return c;
}
```

Linear-Recurrence 算法

```
template<int P = 998244353> MInt<P> linearRecurrence(Poly<P> p, Poly<P> q, i64 n) {
    int m = q.size() - 1;
    while (n > 0) {
        auto newq = q;
        for (int i = 1; i <= m; i += 2) {
            newq[i] *= -1;
        }
        auto newp = p * newq;
```

```
        newq = q * newq;
        for (int i = 0; i < m; i++) {
            p[i] = newp[i * 2 + n % 2];
        }
        for (int i = 0; i <= m; i++) {
            q[i] = newq[i * 2];
        }
        n /= 2;
    }
    return p[0] / q[0];
}
```

快速傅里叶变换 FFT

$\mathcal{O}(N \log N)$.

```
struct Polynomial {
    constexpr static double PI = acos(-1);
    struct Complex {
        double x, y;
        Complex(double _x = 0.0, double _y = 0.0) {
            x = _x;
            y = _y;
        }
        Complex operator-(const Complex &rhs) const {
            return Complex(x - rhs.x, y - rhs.y);
        }
        Complex operator+(const Complex &rhs) const {
            return Complex(x + rhs.x, y + rhs.y);
        }
        Complex operator*(const Complex &rhs) const {
            return Complex(x * rhs.x - y * rhs.y, x * rhs.y + y * rhs.x);
        }
```

```
};
vector<Complex> c;
Polynomial(vector<int> &a) {
    int n = a.size();
    c.resize(n);
    for (int i = 0; i < n; i++) {
        c[i] = Complex(a[i], 0);
    }
    fft(c, n, 1);
}
void change(vector<Complex> &a, int n) {
    for (int i = 1, j = n / 2; i < n - 1; i++) {
        if (i < j) swap(a[i], a[j]);
        int k = n / 2;
        while (j >= k) {
            j -= k;
            k /= 2;
        }
        if (j < k) j += k;
    }
}
void fft(vector<Complex> &a, int n, int opt) {
    change(a, n);
    for (int h = 2; h <= n; h *= 2) {
        Complex wn(cos(2 * PI / h), sin(opt * 2 * PI / h));
        for (int j = 0; j < n; j += h) {
            Complex w(1, 0);
            for (int k = j; k < j + h / 2; k++) {
                Complex u = a[k];
                Complex t = w * a[k + h / 2];
                a[k] = u + t;
                a[k + h / 2] = u - t;
                w = w * wn;
```

```
                }
            }
        }
        if (opt == -1) {
            for (int i = 0; i < n; i++) {
                a[i].x /= n;
            }
        }
    }
};
```

快速数论变换 NTT

$\mathcal{O}(N \log N)$.

```
struct Polynomial {
    vector<Z> z;
    vector<int> r;
    Polynomial(vector<int> &a) {
        int n = a.size();
        z.resize(n);
        r.resize(n);
        for (int i = 0; i < n; i++) {
            z[i] = a[i];
            r[i] = (i & 1) * (n / 2) + r[i / 2] / 2;
        }
        ntt(z, n, 1);
    }
    LL power(LL a, int b) {
        LL res = 1;
        for (; b; b /= 2, a = a * a % mod) {
            if (b % 2) {
                res = res * a % mod;
```

```
            }
        }
        return res;
    }
    void ntt(vector<Z> &a, int n, int opt) {
        for (int i = 0; i < n; i++) {
            if (r[i] < i) {
                swap(a[i], a[r[i]]);
            }
        }
        for (int k = 2; k <= n; k *= 2) {
            Z gn = power(3, (mod - 1) / k);
            for (int i = 0; i < n; i += k) {
                Z g = 1;
                for (int j = 0; j < k / 2; j++, g *= gn) {
                    Z t = a[i + j + k / 2] * g;
                    a[i + j + k / 2] = a[i + j] - t;
                    a[i + j] = a[i + j] + t;
                }
            }
        }
        if (opt == -1) {
            reverse(a.begin() + 1, a.end());
            Z inv = power(n, mod - 2);
            for (int i = 0; i < n; i++) {
                a[i] *= inv;
            }
        }
    }
};
```

需要注意的是, 最后答案要除以做 DFT/IDFT 的长度, 而且做 DFT/IDFT 的长度要一样且是 2 的整数次幂. 还有就是做高精度乘法的时候要记得把数组反向. 如果 TLE 了可以考虑一些常数优化.

```
constexpr i64 mod = 998244353, g = 3;
i64 qpow(i64 a, i64 b) {
    i64 r = 1;
    for (; b; b >>= 1, a = a * a % mod) {
        if (b & 1) r = r * a % mod;
    }
    return r;
}
std::vector<i64> mul(std::vector<i64> a, std::vector<i64> b) {
    int M = a.size() + b.size() - 1u, N = 1;
    while (N < M) N <<= 1;
    std::vector<int> r(N);
    for(int i = 1;i <= N;i++)
        r[i] = r[i / 2] / 2 | (i % 2 ? N / 2 : 0);

    Z ntt = [&](std::vector<i64> &a, bool inv) -> void {
        a.resize(N);
        for(int i = 0;i < N;i++) if (i < r[i]) std::swap(a[i], a[r[i]]);
        for (int sz = 1; sz < N; sz <<= 1) {
            i64 wm = qpow(inv ? g : qpow(g, mod - 2), (mod - 1) / sz / 2);
            for (int i = 0; i < N; i += sz * 2) {
                for (int k = 0, w = 1; k < sz; ++k, w = w * wm % mod) {
                    i64 &x = a[i + k + sz], &y = a[i + k], t = w * x % mod;
                    std::tie(x, y) = pair((y + mod - t) % mod, (y + t) % mod);
                }
            }
        }
        if (i64 in = qpow(N, mod - 2); inv) for(int i = 0;i < N;i++) a[i] = a[i] * in %
    };

    ntt(a, 0);
    ntt(b, 0);
    for(int i = 0;i < N;i++) a[i] = a[i] * b[i] % mod;
```

```
    ntt(a, 1);
    a.resize(M);
    return a;
}
```

拉格朗日插值

$n+1$ 个点可以唯一确定一个最高为 n 次的多项式. 普通情况:$f(k) = \sum_{i=1}^{n+1} y_i \prod_{i \neq j} \frac{k - x[j]}{x[i] - x[j]}$.

```
struct Lagrange {
    int n;
    vector<Z> x, y, fac, invfac;
    Lagrange(int n) {
        this->n = n;
        x.resize(n + 3);
        y.resize(n + 3);
        fac.resize(n + 3);
        invfac.resize(n + 3);
        init(n);
    }
    void init(int n) {
        iota(x.begin(), x.end(), 0);
        for (int i = 1; i <= n + 2; i++) {
            Z t;
            y[i] = y[i - 1] + t.power(i, n);
        }
        fac[0] = 1;
        for (int i = 1; i <= n + 2; i++) {
            fac[i] = fac[i - 1] * i;
        }
        invfac[n + 2] = fac[n + 2].inv();
```

```
        for (int i = n + 1; i >= 0; i--) {
            invfac[i] = invfac[i + 1] * (i + 1);
        }
    }
    Z solve(LL k) {
        if (k <= n + 2) {
            return y[k];
        }
        vector<Z> sub(n + 3);
        for (int i = 1; i <= n + 2; i++) {
            sub[i] = k - x[i];
        }
        vector<Z> mul(n + 3);
        mul[0] = 1;
        for (int i = 1; i <= n + 2; i++) {
            mul[i] = mul[i - 1] * sub[i];
        }
        Z ans = 0;
        for (int i = 1; i <= n + 2; i++) {
            ans = ans + y[i] * mul[n + 2] * sub[i].inv() * pow(-1, n + 2 - i) * invfac[
                            invfac[n + 2 - i];
        }
        return ans;
    }
};
```

结论 **from LuanXR**

1. 序列 a 的**普通生成函数**: $F(x) = \sum a_n x^n$
2. 序列 a 的**指数生成函数**: $F(x) = \sum a_n \frac{x^n}{n!}$

泰勒展开式

1. $\frac{1}{1-x} = 1 + x + x^2 + x^3 + \cdots = \sum_{n=0}^{\infty} x^n$

2. $\frac{1}{1-x^2} = 1 + x^2 + x^4 + \cdots$
3. $\frac{1}{1-x^3} = 1 + x^3 + x^6 + \cdots$
4. $\frac{1}{(1-x)^2} = 1 + 2x + 3x^2 + \cdots$
5. $e^x = 1 + \frac{x^1}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots = \sum_{n=0}^{\infty} \frac{x^n}{n!}$
6. $e^{-x} = 1 - \frac{x^1}{1!} + \frac{x^2}{2!} - \frac{x^3}{3!} + \cdots$
7. $\frac{e^x + e^{-x}}{2} = 1 + \frac{x^2}{2!} + \frac{x^4}{4!} + \cdots$
8. $\frac{e^x - e^{-x}}{2} = x + \frac{x^3}{3!} + \frac{x^5}{5!} + \cdots$

有穷序列的生成函数

1. $1 + x + x^2 = \frac{1-x^3}{1-x}$
2. $1 + x + x^2 + x^3 = \frac{1-x^4}{1-x}$

广义二项式定理

$$\frac{1}{(1-x)^n} = \sum_{i=0}^{\infty} \binom{n+i-1}{i} x^i$$

证明

1. 扩展域 $(1+x)^n = \sum_{i=0}^{n} \binom{n}{i} x^i$, 因 $i > n, \binom{n}{i} = 0$.
2. 扩展指数为负数 $\binom{-n}{i} = \frac{(-n)(-n-1)\cdots(-n-i+1)}{i!} = (-1)^i \times \frac{n(n+1)\cdots(n+i-1)}{i!} = (-1)^i \binom{n+i-1}{i}$
3. 括号内的加号变减号 $(1-x)^{-n} = \sum_{i=0}^{\infty} (-1)^i \binom{n+i-1}{i} (-x)^i = \sum_{i=0}^{\infty} \binom{n+i-1}{i} x^i$

常用结论

杂

- 求 $B_i = \sum_{k=i}^{n} C_k^i A_{k'}$ 即 $B_i = \frac{1}{i!} \sum_{k=i}^{n} \frac{1}{(k-i)!} \cdot k! A_{k'}$ 反转后卷积.

- NTT 中,ω_n = `qpow(G,(mod-1)/n)`.
- 遇到 $\sum_{i=0}^{n}[i\%k=0]f(i)$ 可以转换为 $\sum_{i=0}^{n}\frac{1}{k}\sum_{j=0}^{k-1}(\omega_k^i)^j f(i)$.(单位根卷积)
- 广义二项式定理 $(1+x)^\alpha=\sum_{i=0}^{\infty}\binom{n}{\alpha}x^i$.

普通生成函数 / OGF

- 普通生成函数:$A(x)=a_0+a_1x+a_2x^2+...=\langle a_0,a_1,a_2,...\rangle$;
- $1+x^k+x^{2k}+...=\frac{1}{1-x^k}$;
- 取对数后 $=-\ln(1-x^k)=\sum_{i=1}^{\infty}\frac{1}{i}x^{ki}$ 即 $\sum_{i=1}^{\infty}\frac{1}{i}x^i\otimes x^k$(polymul_special);
- $x+\frac{x^2}{2}+\frac{x^3}{3}+...=-\ln(1-x)$;
- $1+x+x^2+...+x^{m-1}=\frac{1-x^m}{1-x}$;
- $1+2x+3x^2+...=\frac{1}{(1-x)^2}$(借用导数,$nx^{n-1}=(x^n)'$);
- $C_m^0+C_m^1x+C_m^2x^2+...+C_m^mx^m=(1+x)^m$(二项式定理);
- $C_m^0+C_{m+1}^1x^1+C_{m+2}^2x^2+...=\frac{1}{(1-x)^{m+1}}$(归纳法证明);
- $\sum_{n=0}^{\infty}F_nx^n=\frac{(F_1-F_0)x+F_0}{1-x-x^2}$(F 为斐波那契数列, 列方程 $G(x)=xG(x)+x^2G(x)+(F_1-F_0)x+F_0$);
- $\sum_{n=0}^{\infty}H_nx^n=\frac{1-\sqrt{n-4x}}{2x}$(H 为卡特兰数);
- 前缀和 $\sum_{n=0}^{\infty}s_nx^n=\frac{1}{1-x}f(x)$;

- 五边形数定理:$\prod_{i=1}^{\infty}(1-x^i)=\sum_{k=0}^{\infty}(-1)^k x^{\frac{1}{2}k(3k\pm1)}$.

指数生成函数 / EGF

- 指数生成函数:$A(x)=a_0+a_1x+a_2\frac{x^2}{2!}+a_3\frac{x^3}{3!}+...=\langle a_0,a_1,a_2,a_3,...\rangle$;
- 普通生成函数转换为指数生成函数: 系数乘以 $n!$;
- $1+x+\frac{x^2}{2!}+\frac{x^3}{3!}+...=\exp x$;
- 长度为 n 的循环置换数为 $P(x)=-\ln(1-x)$, 长度为 n 的置换数为 $\exp P(x)=\frac{1}{1-x}$(注意是**指数**生成函数)
  - n 个点的生成树个数是 $P(x)=\sum_{n=1}^{\infty}n^{n-2}\frac{x^n}{n!}$,n 个点的生成森林个数是 $\exp P(x)$;
  - n 个点的无向连通图个数是 $P(x)$,n 个点的无向图个数是 $\exp P(x)=\sum_{n=0}^{\infty}2^{\frac{1}{2}n(n-1)}\frac{x^n}{n!}$;
  - 长度为 $n(n\geq2)$ 的循环置换数是 $P(x)=-\ln(1-x)-x$, 长度为 n 的错排数是 $\exp P(x)$.

/END/

串

子串与子序列

中文名称	常见英文名称	解释
子串	substring	连续的选择一段字符 (可以全选, 可以不选) 组成的新字符串
子序列	subsequence	从左到右取出若干个字符 (可以不取, 可以全取, 可以不连续) 组成的新字符串

kmp

应用:

1. 在字符串中查找子串;
2. 最小周期: 字符串长度-整个字符串的 border ;
3. 最小循环节: 区别于周期, 当字符串长度 n mod $(n - nxt[n]) = 0$ 时, 等于最小周期, 否则为 n .

以最坏 $\mathcal{O}(N + M)$ 的时间计算 t 在 s 中出现的全部位置.

```
std::vector<int> get_next(std::string& t) {
    std::vector<int> next(t.size());
    next[0] = -1;
    for (int i = 0, j = -1; i < (int)t.size();) {
        if (j == -1 || t[i] == t[j]) {
            ++i, ++j;
            next[i] = j;
        }
        else
            j = next[j];
    }
    return next;
}

bool kmp(std::string& s, std::string& t) {
    if (t.length() > s.length())return false;
    auto next = get_next(t);

    for (int i = 0, j = 0; i < (int)s.size() && j < (int)t.size();) {
        if (j == -1 || s[i] == t[j]) {
            ++i, ++j;
        }
        else
            j = next[j];
```

```
        if (j == (int)t.size())return true;
    }
    return false;
}
```

zfunction

获取字符串 s 和 $s[i, n-1]$ (即以 $s[i]$ 开头的后缀) 的最长公共前缀 (LCP) 的长度, 总复杂度 $\mathcal{O}(N)$.

```
std::vector<int> z_function(std::string s) {
    int n = (int)s.length();
    std::vector<int> z(n);
    for (int i = 1, l = 0, r = 0; i < n; ++i) {
        if (i <= r && z[i - l] < r - i + 1) {
            z[i] = z[i - l];
        }
        else {
            z[i] = std::max(0, r - i + 1);
            while (i + z[i] < n && s[z[i]] == s[i + z[i]]) ++z[i];
        }
        if (i + z[i] - 1 > r) l = i, r = i + z[i] - 1;
    }
    return z;
}
```

最长公共子序列 LCS

求解两个串的最长公共子序列的长度.

小数据解 针对 10^3 以内的数据.

```
const int N = 1e3 + 10;
char a[N], b[N];
int n, m, f[N][N];
void solve(){
    cin >> n >> m >> a + 1 >> b + 1;
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= m; j++){
            f[i][j] = max(f[i - 1][j], f[i][j - 1]);
            if (a[i] == b[j]) f[i][j] = max(f[i][j], f[i - 1][j - 1] + 1);
        }
    cout << f[n][m] << "\n";
}
int main(){
    solve();
    return 0;
}
```

大数据解 针对 10^5 以内的数据.

```
const int INF = 0x7fffffff;
int n, a[maxn], b[maxn], f[maxn], p[maxn];
int main(){
    cin >> n;
    for (int i = 1; i <= n; i++){
        scanf("%d", &a[i]);
        p[a[i]] = i;  //将第二个序列中的元素映射到第一个中
    }
    for (int i = 1; i <= n; i++){
        scanf("%d", &b[i]);
        f[i] = INF;
    }
    int len = 0;
    f[0] = 0;
```

```
    for (int i = 1; i <= n; i++){
        if (p[b[i]] > f[len]) f[++len] = p[b[i]];
        else {
            int l = 0, r = len;
            while (l < r){
                int mid = (l + r) >> 1;
                if (f[mid] > p[b[i]]) r = mid;
                else l = mid + 1;
            }
            f[l] = min(f[l], p[b[i]]);
        }
    }
    cout << len << "\n";
    return 0;
}
```

字符串哈希

双哈希封装　随机质数列表:1111111121,1211111123,1311111119.

```
const int N = 1 << 21;
static const int mod1 = 1E9 + 7, base1 = 127;
static const int mod2 = 1E9 + 9, base2 = 131;
using U = Zmod<mod1>;
using V = Zmod<mod2>;
vector<U> val1;
vector<V> val2;
void init(int n = N) {
    val1.resize(n + 1), val2.resize(n + 2);
    val1[0] = 1, val2[0] = 1;
    for (int i = 1; i <= n; i++) {
        val1[i] = val1[i - 1] * base1;
        val2[i] = val2[i - 1] * base2;
```

```
    }
}
struct String {
    vector<U> hash1;
    vector<V> hash2;
    string s;

    String(string s_) : s(s_), hash1{1}, hash2{1} {
        for (auto it : s) {
            hash1.push_back(hash1.back() * base1 + it);
            hash2.push_back(hash2.back() * base2 + it);
        }
    }
    pair<U, V> get() { // 输出整串的哈希值
        return {hash1.back(), hash2.back()};
    }
    pair<U, V> substring(int l, int r) { // 输出子串的哈希值
        if (l > r) swap(l, r);
        U ans1 = hash1[r + 1] - hash1[l] * val1[r - l + 1];
        V ans2 = hash2[r + 1] - hash2[l] * val2[r - l + 1];
        return {ans1, ans2};
    }
    pair<U, V> modify(int idx, char x) { // 修改 idx 位为 x
        int n = s.size() - 1;
        U ans1 = hash1.back() + val1[n - idx] * (x - s[idx]);
        V ans2 = hash2.back() + val2[n - idx] * (x - s[idx]);
        return {ans1, ans2};
    }
};
```

前后缀去重 sample please ease 去重后得到 samplease.

```
string compress(vector<string> in) { // 前后缀压缩
    vector<U> hash1{1};
    vector<V> hash2{1};
    string ans = "#";
    for (auto s : in) {
        s = "#" + s;
        int st = 0;
        U chk1 = 0;
        V chk2 = 0;
        for (int j = 1; j < s.size() && j < ans.size(); j++) {
            chk1 = chk1 * base1 + s[j];
            chk2 = chk2 * base2 + s[j];
            if ((hash1.back() == hash1[ans.size() - 1 - j] * val1[j] + chk1) &&
                (hash2.back() == hash2[ans.size() - 1 - j] * val2[j] + chk2)) {
                st = j;
            }
        }
        for (int j = st + 1; j < s.size(); j++) {
            ans += s[j];
            hash1.push_back(hash1.back() * base1 + s[j]);
            hash2.push_back(hash2.back() * base2 + s[j]);
        }
    }
    return ans.substr(1);
}
```

马拉车

```
struct Manachar {
    std::vector<int> d1, d2;
    Manachar(std::string s) {
        int n = s.length();
        d1.assign(n, 0);
```

```
        d2.assign(n, 0);
        for (int i = 0, l = 0, r = -1;i < n;++i) {
            int k = (i > r) ? 1 : std::min(d1[l + r - i], r - i + 1);
            while (i + k < n && i - k >= 0 && s[i + k] == s[i - k])k++;
            d1[i] = k--;
            if (i + k > r) {
                r = i + k;
                l = i - k;
            }
        }
        for (int i = 0, l = 0, r = -1;i < n;++i) {
            int k = (i > r) ? 0 : std::min(d2[l + r - i + 1], r - i + 1);
            while (i + k < n && i - k - 1 >= 0 && s[i + k] == s[i - k - 1])k++;
            d2[i] = k--;
            if (i + k > r) {
                r = i + k;
                l = i - k - 1;
            }
        }
    }
    bool check(int l, int r) {
        if (r < l)return false;
        int len = r - l + 1;
        if (len % 2) {
            return d1[l + len / 2] * 2 - 1 < len;
        }
        else {
            return d2[l + len / 2] * 2 < len;
        }
    }
};
```

字典树 trie

基础封装

```
struct Trie {
    int ch[N][63], cnt[N], idx = 0;
    map<char, int> mp;
    void init() {
        LL id = 0;
        for (char c = 'a'; c <= 'z'; c++) mp[c] = ++id;
        for (char c = 'A'; c <= 'Z'; c++) mp[c] = ++id;
        for (char c = '0'; c <= '9'; c++) mp[c] = ++id;
    }
    void insert(string s) {
        int u = 0;
        for (int i = 0; i < s.size(); i++) {
            int v = mp[s[i]];
            if (!ch[u][v]) ch[u][v] = ++idx;
            u = ch[u][v];
            cnt[u]++;
        }
    }
    LL query(string s) {
        int u = 0;
        for (int i = 0; i < s.size(); i++) {
            int v = mp[s[i]];
            if (!ch[u][v]) return 0;
            u = ch[u][v];
        }
        return cnt[u];
    }
    void Clear() {
        for (int i = 0; i <= idx; i++) {
            cnt[i] = 0;
```

```
            for (int j = 0; j <= 62; j++) {
                ch[i][j] = 0;
            }
        }
        idx = 0;
    }
} trie;
```

01 字典树

```
struct Trie {
    int n, idx;
    vector<vector<int>> ch;
    Trie(int n) {
        this->n = n;
        idx = 0;
        ch.resize(30 * (n + 1), vector<int>(2));
    }
    void insert(int x) {
        int u = 0;
        for (int i = 30; ~i; i--) {
            int &v = ch[u][x >> i & 1];
            if (!v) v = ++idx;
            u = v;
        }
    }
    int query(int x) {
        int u = 0, res = 0;
        for (int i = 30; ~i; i--) {
            int v = x >> i & 1;
            if (ch[u][!v]) {
                res += (1 << i);
                u = ch[u][!v];
```

```
            } else {
                u = ch[u][v];
            }
        }
        return res;
    }
};
```

后缀数组 SA

以 $\mathcal{O}(N)$ 的复杂度求解.

```
struct SuffixArray {
    int n;
    vector<int> sa, rk, lc;
    SuffixArray(const string &s) {
        n = s.length();
        sa.resize(n);
        lc.resize(n - 1);
        rk.resize(n);
        iota(sa.begin(), sa.end(), 0);
        sort(sa.begin(), sa.end(), [&](int a, int b) { return s[a] < s[b]; });
        rk[sa[0]] = 0;
        for (int i = 1; i < n; ++i) {
            rk[sa[i]] = rk[sa[i - 1]] + (s[sa[i]] != s[sa[i - 1]]);
        }
        int k = 1;
        vector<int> tmp, cnt(n);
        tmp.reserve(n);
        while (rk[sa[n - 1]] < n - 1) {
            tmp.clear();
            for (int i = 0; i < k; ++i) {
                tmp.push_back(n - k + i);
```

```
        }
        for (auto i : sa) {
            if (i >= k) {
                tmp.push_back(i - k);
            }
        }
        fill(cnt.begin(), cnt.end(), 0);
        for (int i = 0; i < n; ++i) {
            ++cnt[rk[i]];
        }
        for (int i = 1; i < n; ++i) {
            cnt[i] += cnt[i - 1];
        }
        for (int i = n - 1; i >= 0; --i) {
            sa[--cnt[rk[tmp[i]]]] = tmp[i];
        }
        swap(rk, tmp);
        rk[sa[0]] = 0;
        for (int i = 1; i < n; ++i) {
            rk[sa[i]] = rk[sa[i - 1]] + (tmp[sa[i - 1]] < tmp[sa[i]] || sa[i - 1] +
                                         tmp[sa[i - 1] + k] < tmp[sa[i] + k]);
        }
        k *= 2;
    }
    for (int i = 0, j = 0; i < n; ++i) {
        if (rk[i] == 0) {
            j = 0;
            continue;
        }
        for (j -= j > 0;
             i + j < n && sa[rk[i] - 1] + j < n && s[i + j] == s[sa[rk[i] - 1] + j]
            ++j;
        }
```

```
            lc[rk[i] - 1] = j;
        }
    }
};
```

AC 自动机

定义 $|s_i|$ 是模板串的长度,$|S|$ 是文本串的长度,$|\Sigma|$ 是字符集的大小 (常数, 一般为 26), 时间复杂度为 $\mathcal{O}(\sum |s_i| + |S|)$.

```
// Trie+Kmp, 多模式串匹配
struct ACAutomaton {
    static constexpr int N = 1e6 + 10;
    int ch[N][26], fail[N], cntNodes;
    int cnt[N];
    ACAutomaton() {
        cntNodes = 1;
    }
    void insert(string s) {
        int u = 1;
        for (auto c : s) {
            int &v = ch[u][c - 'a'];
            if (!v) v = ++cntNodes;
            u = v;
        }
        cnt[u]++;
    }
    void build() {
        fill(ch[0], ch[0] + 26, 1);
        queue<int> q;
        q.push(1);
        while (!q.empty()) {
            int u = q.front();
```

```
            q.pop();
            for (int i = 0; i < 26; i++) {
                int &v = ch[u][i];
                if (!v)
                    v = ch[fail[u]][i];
                else {
                    fail[v] = ch[fail[u]][i];
                    q.push(v);
                }
            }
        }
    }
    LL query(string t) {
        LL ans = 0;
        int u = 1;
        for (auto c : t) {
            u = ch[u][c - 'a'];
            for (int v = u; v && ~cnt[v]; v = fail[v]) {
                ans += cnt[v];
                cnt[v] = -1;
            }
        }
        return ans;
    }
};

struct AhoCorasick {
    static constexpr int ALPHABET = 26;
    struct Node {
        int len;
        int link;
        std::array<int, ALPHABET> next;
        Node() : len{ 0 }, link{ 0 }, next{} {}
    };
```

```
    std::vector<Node> t;

    AhoCorasick() {
        init();
    }

    void init() {
        t.assign(2, Node());
        t[0].next.fill(1);
        t[0].len = -1;
    }

    int newNode() {
        t.emplace_back();
        return t.size() - 1;
    }

    int add(const std::string& a) {
        int p = 1;
        for (auto c : a) {
            int x = c - 'a';
            if (t[p].next[x] == 0) {
                t[p].next[x] = newNode();
                t[t[p].next[x]].len = t[p].len + 1;
            }
            p = t[p].next[x];
        }
        return p;
    }

    void get_fail() {
        std::queue<int> q;
```

```
        q.push(1);

        while (!q.empty()) {
            int x = q.front();
            q.pop();

            for (int i = 0; i < ALPHABET; i++) {
                if (t[x].next[i] == 0) {
                    t[x].next[i] = t[t[x].link].next[i];
                }
                else {
                    t[t[x].next[i]].link = t[t[x].link].next[i];
                    q.push(t[x].next[i]);
                }
            }
        }
    }

    std::vector<int> work(std::string s) {
        get_fail();
        int p = 1;
        std::vector<int> f(t.size());
        for (auto c : s) {
            p = next(p, c - 'a');
            f[p]++;
        }

        std::vector<std::vector<int>> adj(t.size());
        for (int i = 2; i < t.size(); i++) {
            adj[link(i)].push_back(i);
        }

        std::function<void(int)> dfs = [&](int x) -> void {
```

```
            for (auto y : adj[x]) {
                dfs(y);
                f[x] += f[y];
            }
            };
        dfs(1);
        return f;
    }

    int next(int p, int x) {
        return t[p].next[x];
    }

    int link(int p) {
        return t[p].link;
    }

    int len(int p) {
        return t[p].len;
    }

    int size() {
        return t.size();
    }
};
```

回文自动机 PAM(回文树)

```
struct PalindromeAutomaton {
    constexpr static int N = 5e5 + 10;
    int tr[N][26], fail[N], len[N];
    int cntNodes, last;
    int dep[N]; //记录深度
```

```
    int cnt[N]; //记录出现次数
    std::string s;
    PalindromeAutomaton(std::string s) {
        memset(tr, 0, sizeof tr);
        memset(fail, 0, sizeof fail);
        memset(dep, 0, sizeof dep);
        memset(cnt, 0, sizeof cnt);
        len[0] = 0, fail[0] = 1;
        len[1] = -1, fail[1] = 0;
        cntNodes = 1;
        last = 0;
        this->s = s;
    }
    void insert(char c, int i) {
        int u = get_fail(last, i);
        if (!tr[u][c - 'a']) {
            int v = ++cntNodes;
            fail[v] = tr[get_fail(fail[u], i)][c - 'a'];
            tr[u][c - 'a'] = v;
            len[v] = len[u] + 2;
            dep[v] = dep[fail[v]] + 1;
        }
        last = tr[u][c - 'a'];
        cnt[last] += 1;
    }
    void countAll() {
        for (int i = cntNodes;i >= 0;i--)
            cnt[fail[i]] += cnt[i];
    }
    int get_fail(int u, int i) {
        while (i - len[u] - 1 <= -1 || s[i - len[u] - 1] != s[i]) {
            u = fail[u];
        }
```

```
        return u;
    }
};

const int TL = 10;
const char BC = '0';
struct PAM {
    struct node
    {
        int len, link, cnt;
        std::array<int, TL> next;
    };
    std::vector<node> nodes;
    int last;
    std::string s;
    int n;
    PAM(std::string& s) {
        n = s.length();
        nodes.reserve(n);
        this->s = s;
        nodes.assign(2, node());
        last = 0;
        nodes[0].len = 0;
        nodes[0].link = 1;
        nodes[1].len = -1;
        nodes[1].link = 0;
        for (int i = 0;i < n;++i) {
            extend(s[i], i);
        }
    }
    void extend(char ch, int p) {
        int u = get_fail(last, p);
        int c = ch - BC;
        if (!nodes[u].next[c]) {
```

```
            int v = nodes.size();
            nodes.emplace_back();
            nodes[v].link = nodes[get_fail(nodes[u].link, p)].next[c];
            nodes[u].next[c] = v;
            nodes[v].len = nodes[u].len + 2;
        }
        last = nodes[u].next[c];
        nodes[last].cnt++;
    }
    int get_fail(int u, int i) {
        while (i - nodes[u].len - 1 < 0 || s[i - nodes[u].len - 1] != s[i]) {
            u = nodes[u].link;
        }
        return u;
    }
    void debug() {
        for (auto node : nodes) {
            std::cerr << node.cnt << ' ';
            std::cerr << node.len << ' ';
            std::cerr << node.link << '\n';
        }
    }
};
```

后缀自动机 SAM

定义 $|\Sigma|$ 是字符集的大小, 复杂度为 $\mathcal{O}(N \log |\Sigma|)$.

后缀自动机 (Suffix Automaton, SAM) 虽然也能用于匹配, 但它的强大之处在于对**单个字符串**的**所有子串**进行深度分析. 除去 AC 自动机也能做的简单子串匹配外, 后缀自动机还能高效完成以下任务:

1. 统计不同子串的数量:

- 后缀自动机可以在线性时间内计算出一个字符串中本质不同的子串有多少个. 这是 AC 自动机完全无法做到的.

2. **计算最长公共子串 (LCS)**:
   - 通过构建两个或多个字符串的广义后缀自动机, 可以高效地找到它们的最长公共子串. 这虽然也涉及多字符串, 但其解决问题的角度和 AC 自动机完全不同.
3. **查找字典序第 k 小子串**:
   - 后缀自动机的图结构天然支持按字典序遍历, 因此可以高效地找出所有不同子串中, 按字典序排序后的第 k 个.
4. **计算任意子串的出现次数**:
   - 对于给定的字符串 S 的任意一个子串 P, 后缀自动机可以快速计算出 P 在 S 中出现了多少次. 而 AC 自动机只能计算" 词典中的串的出现次数.
5. **寻找最小循环移位**:
   - 这是一个经典应用, 可以利用后缀自动机在线性时间内找到一个字符串的最小字典序循环移位.

如果把 AC 自动机看作是"**在一篇文章里找特定的几个关键词**的专家, 那么后缀自动机就是"**给你一篇文章, 然后问关于这篇文章任何片段 (子串) 的任何刁钻问题**的全能专家. 它的应用深度和广度远超多模式匹配.

```
// 有向无环图
struct SuffixAutomaton {
    static constexpr int N = 1e6;
    struct node {
        int len, link, nxt[26];
        int siz;
    } t[N << 1];
    int cntNodes;
    SuffixAutomaton() {
        cntNodes = 1;
        fill(t[0].nxt, t[0].nxt + 26, 1);
        t[0].len = -1;
    }
```

```
    int extend(int p, int c) {
        if (t[p].nxt[c]) {
            int q = t[p].nxt[c];
            if (t[q].len == t[p].len + 1) {
                return q;
            }
            int r = ++cntNodes;
            t[r].siz = 0;
            t[r].len = t[p].len + 1;
            t[r].link = t[q].link;
            copy(t[q].nxt, t[q].nxt + 26, t[r].nxt);
            t[q].link = r;
            while (t[p].nxt[c] == q) {
                t[p].nxt[c] = r;
                p = t[p].link;
            }
            return r;
        }
        int cur = ++cntNodes;
        t[cur].len = t[p].len + 1;
        t[cur].siz = 1;
        while (!t[p].nxt[c]) {
            t[p].nxt[c] = cur;
            p = t[p].link;
        }
        t[cur].link = extend(p, c);
        return cur;
    }
};
```

endpos, size 按需 link 构造后缀树

```
struct SAM {
    struct node {
```

```
        int len, link, endpos, size;
        std::map<char, int> next;
        node() {
            len = link = endpos = -1;
            size = 0;
            next = std::map<char, int>();
        }
    };
    std::vector<node> nodes;
    int last;
    int n;

    SAM(std::string& s) {
        n = s.length();
        nodes.reserve(2 * n);
        nodes.assign(1, node());
        nodes[0].len = 0;
        nodes[0].link = -1;
        last = 0;
        for (int i = 0;i < n;++i) {
            extend(s[i], i + 1);
        }
    }

    void extend(char c, int pos) {
        int cur = nodes.size();
        nodes.emplace_back();
        nodes[cur].len = nodes[last].len + 1;
        int p = last;
        while (p != -1 && !nodes[p].next.count(c)) {
            nodes[p].next[c] = cur;
            p = nodes[p].link;
        }
```

```
        if (p == -1) {
            nodes[cur].link = 0;
        }
        else {
            int q = nodes[p].next[c];
            if (nodes[p].len + 1 == nodes[q].len) {
                nodes[cur].link = q;
            }
            else {
                int clone = nodes.size();
                nodes.emplace_back();
                nodes[clone].len = nodes[p].len + 1;
                nodes[clone].link = nodes[q].link;
                nodes[clone].next = nodes[q].next;
                while (p != -1 && nodes[p].next[c] == q) {
                    nodes[p].next[c] = clone;
                    p = nodes[p].link;
                }
                nodes[q].link = nodes[cur].link = clone;
            }
        }
        nodes[cur].endpos = pos;
        nodes[cur].size = 1;
        last = cur;
    }

    void debug() {
        for (auto x : nodes) {
            std::cerr << x.len << ' ';
            std::cerr << x.link << ' ';
            std::cerr << x.endpos << ' ';
            std::cerr << x.size << ' ';
            std::cerr << '\n';
```

```
        }
    }
};
```

子序列自动机

对于给定的长度为 n 的主串 s , 以 $\mathcal{O}(n)$ 的时间复杂度预处理,$\mathcal{O}(m + \log \text{size:} s)$ 的复杂度判定长度为 m 的询问串是否是主串的子序列.

好的, 我们来谈谈子序列自动机 (Subsequence Automaton).

相比于后缀自动机 (SAM) 和回文自动机 (PAM), 子序列自动机在结构上要简单得多, 但它同样是处理特定字符串问题的有效工具. 它的主要应用领域集中在与**子序列**相关的匹配和统计问题上.

什么是子序列自动机

对于一个给定的字符串 S(长度为 n), 它的子序列自动机是一个能够识别 S 所有子序列的自动机. 其构造非常直观和简单:

它通常被实现为一个二维数组 `next[i][c]`, 表示在字符串的第 `i` 个位置之后 (不包括 `i`), 字符 `c` 第一次出现的位置. 这个数组可以在 O(n) 的时间内预处理出来, 其中是字符集的大小 (例如, 对于小写字母是 26).

举例:

对于字符串 S = “banana”

next[0][‘b’] 是 1 (第一个‘b’的位置)

next[1][‘n’] 是 3 (位置 1 ‘a’之后, 下一个‘n’在位置 3)

next[3][‘n’] 是 5 (位置 3 ‘n’之后, 下一个‘n’在位置 5)

主要用途

子序列自动机的主要用途可以归结为以下几点:

1. **判断一个字符串是否为子序列 (子序列匹配)**:
    - 这是最核心和最常见的用途. 给定一个模式串 T, 要判断它是否是主串 S 的子序列, 只需利用预处理好的 `next` 数组进行贪心匹配. 从位置 0 开始, 依次为 T 的每个字符在 S 中寻找下一个最近的匹配位置. 这个过程的效率极高, 时间复杂度为 O(T).
2. **解决" 公共子序列相关问题**:
    - 虽然寻找" 最长公共子序列 (LCS) 通常使用动态规划, 但在某些特定场景下, 子序列自动机可以提供不同的解题思路.
    - 例如, 在多个字符串上构建各自的子序列自动机, 然后通过在这些自动机上同步转移 (类似 DP), 可以用来寻找多个字符串的" 最短的公共超序列 (Shortest Common Supersequence) 或解决其他相关的公共子序列变种问题.
3. **计算不同子序列的数量**:
    - 可以通过在子序列自动机上进行动态规划 (DP) 来计算一个字符串本质不同的子序列有多少个.DP 状态通常定义为 `dp[i]` 表示从位置 `i` 开始的子序列个数.
4. **寻找字典序第 k 小子序列**:
    - 与计算数量类似, 通过在自动机上进行 DP, 预先计算出从每个位置出发能产生多少不同的子序列, 然后就可以按位确定第 k 小的子序列应该选择哪个字符作为开头, 并跳转到相应的位置.

- **结构简单**: 相比 SAM 和 PAM, 它的概念和实现都非常简单, 就是一个 `next` 数组.
- **构建快速**: 预处理速度很快, 尤其适用于字符集较小的情况.
- **匹配高效**: 对于子序列匹配问题, 查询效率是线性的, 与主串长度无关.

子序列自动机是专门用于高效处理字符串" 子序列相关问题的简单数据结构. 当题目需要反复, 快速地判断一个或多个字符串是否为某个主串的子序列, 或者需要对子序列进行统计和计数时, 它就是非常有用的工具.

自动离散化, 自动类型匹配封装

```
template<typename T> struct SequenceAutomaton {
    vector<T> alls;
    vector<vector<int>> ver;

    SequenceAutomaton(auto in) {
        for (auto &i : in) {
            alls.push_back(i);
        }
        sort(alls.begin(), alls.end());
        alls.erase(unique(alls.begin(), alls.end()), alls.end());

        ver.resize(alls.size() + 1);
        for (int i = 0; i < in.size(); i++) {
            ver[get(in[i])].push_back(i + 1);
        }
    }
    bool count(T x) {
        return binary_search(alls.begin(), alls.end(), x);
    }
    int get(T x) {
        return lower_bound(alls.begin(), alls.end(), x) - alls.begin();
    }
    bool contains(auto in) {
        int at = 0;
        for (auto &i : in) {
            if (!count(i)) {
                return false;
            }

            auto j = get(i);
            auto it = lower_bound(ver[j].begin(), ver[j].end(), at + 1);
            if (it == ver[j].end()) {
```

```
                return false;
            }
            at = *it;
        }
        return true;
    }
};
```

朴素封装 原时间复杂度中的 size:s 需要手动设置. 类型需要手动设置.

```
struct SequenceAutomaton {
    vector<vector<int>> ver;

    SequenceAutomaton(vector<int> &in, int size) : ver(size + 1) {
        for (int i = 0; i < in.size(); i++) {
            ver[in[i]].push_back(i + 1);
        }
    }
    bool contains(vector<int> &in) {
        int at = 0;
        for (auto &i : in) {
            auto it = lower_bound(ver[i].begin(), ver[i].end(), at + 1);
            if (it == ver[i].end()) {
                return false;
            }
            at = *it;
        }
        return true;
    }
};
```

/END/

二维几何

format 格式化输出小数点

```
cout << format("{:.2f}", 114514.1919810) << endl;
// output: 114514.19
```

库实数类实现 (双精度)

```
using Real = int;
using Point = complex<Real>;

Real cross(const Point &a, const Point &b) {
    return (conj(a) * b).imag();
}
Real dot(const Point &a, const Point &b) {
    return (conj(a) * b).real();
}
```

平面几何必要初始化

字符串读入浮点数

```
const int Knum = 4;
int read(int k = Knum) {
    string s;
    cin >> s;

    int num = 0;
    int it = s.find('.');
    if (it != -1) { // 存在小数点
        num = s.size() - it - 1; // 计算小数位数
        s.erase(s.begin() + it); // 删除小数点
```

```
    }
    for (int i = 1; i <= k - num; i++) { // 补全小数位数
        s += '0';
    }
    return stoi(s);
}
```

预置函数

```
using ld = long double;
const ld PI = acos(-1);
const ld EPS = 1e-7;
const ld INF = numeric_limits<ld>::max();
#define cc(x) cout << fixed << setprecision(x);

ld fgcd(ld x, ld y) { // 实数域 gcd
    return abs(y) < EPS ? abs(x) : fgcd(y, fmod(x, y));
}
template<typename T, typename S>
bool equal(T x, S y) {
    return -EPS < x - y && x - y < EPS;
}
template<typename T>
int sign(T x) {
    if (-EPS < x && x < EPS) return 0;
    return x < 0 ? -1 : 1;
}
```

点线封装

```
template<typename T>
struct Point {  // 在 C++17 下使用 emplace_back 绑定可能会导致 CE!
    T x, y;
```

```
    Point(T x_ = 0, T y_ = 0) : x(x_), y(y_) {}

    template<typename U>
    operator Point<U>() {  // 自动类型匹配
        return Point<U>(U(x), U(y));
    }

    Point operator-() const { return {-x, -y}; }
    Point operator+(const Point &b) const { return {x + b.x, y + b.y}; }
    Point operator-(const Point &b) const { return {x - b.x, y - b.y}; }

    Point operator+(const T &b) const { return {x + b, y + b}; }
    Point operator-(const T &b) const { return {x - b, y - b}; }
    Point operator*(const T &b) const { return {x * b, y * b}; }
    Point operator/(const T &b) const { return {x / b, y / b}; }

    T operator*(const Point &b) const { return x * b.x + y * b.y; }  // 点积
    T operator^(const Point &b) const { return x * b.y - y * b.x; }  // 叉积

    Point &operator+=(const Point &p) { x += p.x; y += p.y; return *this; }
    Point &operator-=(const Point &p) { x -= p.x; y -= p.y; return *this; }

    Point &operator+=(const T t) { x += t; y += t; return *this; }
    Point &operator-=(const T t) { x -= t; y -= t; return *this; }
    Point &operator*=(const T t) { x *= t; y *= t; return *this; }
    Point &operator/=(const T t) { x /= t; y /= t; return *this; }

    bool operator<(const Point &b) const {
        return equal(x, b.x) ? y < b.y - EPS : x < b.x - EPS;
    }
    bool operator>(const Point &b) const { return b < *this; }
    bool operator==(const Point &b) const { return !(b < *this) && !(*this < b); }
    bool operator!=(const Point &b) const { return *this < b || b < *this; }
```

```
    friend istream &operator>>(istream &is, Point &p) {
        return is >> p.x >> p.y;
    }
    friend ostream &operator<<(ostream &os, const Point &p) {
        return os << format("({},{})", p.x, p.y);     // C++20
        return os << '(' << p.x << ',' << p.y << ')'; // C++17
    }
};

template<typename T>
struct Line {
    Point<T> a, b;
    Line(Point<T> a_ = Point<T>(), Point<T> b_ = Point<T>()) : a(a_), b(b_) {}
    template<typename U>
    operator Line<U>() {  // 自动类型匹配
        return Line<U>(Point<U>(a), Point<U>(b));
    }
    friend ostream &operator<<(ostream &os, const Line &l) {
        return os << '<' << l.a << ',' << l.b << '>';
    }
};
```

叉乘 定义公式 $a \times b = |a||b| \sin\theta$.

```
template<typename T>  // 叉乘
T cross(Point<T> a, Point<T> b) { return a.x * b.y - a.y * b.x; }
template<typename T>  // 叉乘 (p1 - p0) x (p2 - p0);
T cross(Point<T> p1, Point<T> p2, Point<T> p0) { return cross(p1 - p0, p2 - p0); }
```

点乘 定义公式 $a \times b = |a||b| \cos\theta$.

```
template<typename T>  // 点乘
T dot(Point<T> a, Point<T> b) { return a.x * b.x + a.y * b.y; }
template<typename T>  // 点乘 (p1 - p0) * (p2 - p0);
T dot(Point<T> p1, Point<T> p2, Point<T> p0) { return dot(p1 - p0, p2 - p0); }
```

欧几里得距离公式 最常用的距离公式. **需要注意**, 开根号会丢失精度, 如无强制要求, 先不要开根号, 留到最后一步一起开.

```
template<typename T> ld dis(T x1, T y1, T x2, T y2) {
    ld val = (x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 - y2);
    return sqrt(val);
}
template<typename T> ld dis(Point<T> a, Point<T> b) {
    return dis(a.x, a.y, b.x, b.y);
}
```

曼哈顿距离公式

```
template<typename T> T dis1(Point<T> p1, Point<T> p2) { // 曼哈顿距离公式
    return abs(p1.x - p2.x) + abs(p1.y - p2.y);
}
```

将向量转换为单位向量

```
Point<ld> standardize(Point<ld> vec) { // 转换为单位向量
    return vec / sqrt(vec.x * vec.x + vec.y * vec.y);
}
```

向量旋转 将当前向量移动至原点后顺时针旋转 90° , 即获取垂直于当前向量的, 起点为原点的向量. 在计算垂线时非常有用. 例如, 要想获取点 a 绕点 o 顺时针旋转 90° 后的点, 可以这样书写代码:`auto ans = o + rotate(o, a);` ; 如果是逆时针旋转, 那么只需更改符号即可:`auto ans = o - rotate(o, a);` .

```cpp
template<typename T> Point<T> rotate(Point<T> p1, Point<T> p2) { // 旋转
    Point<T> vec = p1 - p2;
    return {-vec.y, vec.x};
}
```

平面角度与弧度

弧度角度相互转换

```cpp
ld toDeg(ld x) { // 弧度转角度
    return x * 180 / PI;
}
ld toArc(ld x) { // 角度转弧度
    return PI / 180 * x;
}
```

正弦定理 $\frac{a}{\sin A} = \frac{b}{\sin B} = \frac{c}{\sin C} = 2R$, 其中 R 为三角形外接圆半径;

余弦定理 (已知三角形三边, 求角) $\cos C = \frac{a^2+b^2-c^2}{2ab}, \cos B = \frac{a^2+c^2-b^2}{2ac}, \cos A = \frac{b^2+c^2-a^2}{2bc}$. 可以借此推导出三角形面积公式 $S_{\triangle ABC} = \frac{ab\cdot\sin C}{2} = \frac{bc\cdot\sin A}{2} = \frac{ac\cdot\sin B}{2}$.

注意, 计算格式是: 由 b, c, a 三边求 $\angle A$; 由 a, c, b 三边求 $\angle B$; 由 a, b, c 三边求 $\angle C$.

```cpp
ld angle(ld a, ld b, ld c) { // 余弦定理
    ld val = acos((a * a + b * b - c * c) / (2.0 * a * b)); // 计算弧度
    return val;
}
```

求两向量的夹角 能够计算 $[0°, 180°]$ 区间的角度.

```
ld angle(Point<ld> a, Point<ld> b) {
    ld val = abs(cross(a, b));
    return abs(atan2(val, a.x * b.x + a.y * b.y));
}
```

向量旋转任意角度 逆时针旋转, 转换公式: $\begin{cases} x' = x\cos\theta - y\sin\theta \\ y' = x\sin\theta + y\cos\theta \end{cases}$

```
Point<ld> rotate(Point<ld> p, ld rad) {
    return {p.x * cos(rad) - p.y * sin(rad), p.x * sin(rad) + p.y * cos(rad)};
}
```

点绕点旋转任意角度 逆时针旋转, 转换公式: $\begin{cases} x' = (x_0 - x_1)\cos\theta + (y_0 - y_1)\sin\theta + x_1 \\ y' = (x_1 - x_0)\sin\theta + (y_0 - y_1)\cos\theta + y_1 \end{cases}$

```
Point<ld> rotate(Point<ld> a, Point<ld> b, ld rad) {
    ld x = (a.x - b.x) * cos(rad) + (a.y - b.y) * sin(rad) + b.x;
    ld y = (b.x - a.x) * sin(rad) + (a.y - b.y) * cos(rad) + b.y;
    return {x, y};
}
```

平面点线相关

点是否在直线上 (三点是否共线)

```
template<typename T> bool onLine(Point<T> a, Point<T> b, Point<T> c) {
    return sign(cross(b, a, c)) == 0;
}
template<typename T> bool onLine(Point<T> p, Line<T> l) {
    return onLine(p, l.a, l.b);
}
```

点是否在向量 (直线) 左侧 需要注意, 向量的方向会影响答案; 点在向量上时不视为在左侧.

```
template<typename T> bool pointOnLineLeft(Pt p, Lt l) {
    return cross(l.b, p, l.a) > 0;
}
```

两点是否在直线同侧/异侧

```
template<typename T> bool pointOnLineSide(Pt p1, Pt p2, Lt vec) {
    T val = cross(p1, vec.a, vec.b) * cross(p2, vec.a, vec.b);
    return sign(val) == 1;
}
template<typename T> bool pointNotOnLineSide(Pt p1, Pt p2, Lt vec) {
    T val = cross(p1, vec.a, vec.b) * cross(p2, vec.a, vec.b);
    return sign(val) == -1;
}
```

两直线相交交点 在使用前需要先判断直线是否平行.

```
Pd lineIntersection(Ld l1, Ld l2) {
    ld val = cross(l2.b - l2.a, l1.a - l2.a) / cross(l2.b - l2.a, l1.a - l1.b);
    return l1.a + (l1.b - l1.a) * val;
}
```

两直线是否平行/垂直/相同

```
template<typename T> bool lineParallel(Lt p1, Lt p2) {
    return sign(cross(p1.a - p1.b, p2.a - p2.b)) == 0;
}
template<typename T> bool lineVertical(Lt p1, Lt p2) {
    return sign(dot(p1.a - p1.b, p2.a - p2.b)) == 0;
}
template<typename T> bool same(Line<T> l1, Line<T> l2) {
```

```
    return lineParallel(Line{l1.a, l2.b}, {l1.b, l2.a}) &&
           lineParallel(Line{l1.a, l2.a}, {l1.b, l2.b}) && lineParallel(l1, l2);
}
```

点到直线的最近距离与最近点

```
pair<Pd, ld> pointToLine(Pd p, Ld l) {
    Pd ans = lineIntersection({p, p + rotate(l.a, l.b)}, l);
    return {ans, dis(p, ans)};
}
```

如果只需要计算最近距离, 下方的写法可以减少书写的代码量, 效果一致.

```
template<typename T> ld disPointToLine(Pt p, Lt l) {
    ld ans = cross(p, l.a, l.b);
    return abs(ans) / dis(l.a, l.b); // 面积除以底边长
}
```

点是否在线段上

```
template<typename T> bool pointOnSegment(Pt p, Lt l) { // 端点也算作在直线上
    return sign(cross(p, l.a, l.b)) == 0 && min(l.a.x, l.b.x) <= p.x && p.x <= max(l.a.
           min(l.a.y, l.b.y) <= p.y && p.y <= max(l.a.y, l.b.y);
}
template<typename T> bool pointOnSegment(Pt p, Lt l) { // 端点不算
    return pointOnSegment(p, l) && min(l.a.x, l.b.x) < p.x && p.x < max(l.a.x, l.b.x) &
           min(l.a.y, l.b.y) < p.y && p.y < max(l.a.y, l.b.y);
}
```

点到线段的最近距离与最近点

```
pair<Pd, ld> pointToSegment(Pd p, Ld l) {
    if (sign(dot(p, l.b, l.a)) == -1) { // 特判到两端点的距离
        return {l.a, dis(p, l.a)};
```

```
    } else if (sign(dot(p, l.a, l.b)) == -1) {
        return {l.b, dis(p, l.b)};
    }
    return pointToLine(p, l);
}
```

点在直线上的投影点 (垂足)

```
Pd project(Pd p, Ld l) { // 投影
    Pd vec = l.b - l.a;
    ld r = dot(vec, p - l.a) / (vec.x * vec.x + vec.y * vec.y);
    return l.a + vec * r;
}
```

线段的中垂线

```
template<typename T> Lt midSegment(Lt l) {
    Pt mid = (l.a + l.b) / 2; // 线段中点
    return {mid, mid + rotate(l.a, l.b)};
}
```

两线段是否相交及交点 该扩展版可以同时返回相交状态和交点, 分为四种情况:0 代表不相交;1 代表普通相交;2 代表重叠 (交于两个点);3 代表相交于端点. **需要注意**, 部分运算可能会使用到直线求交点, 此时务必保证变量类型为浮点数!

```
template<typename T> tuple<int, Pt, Pt> segmentIntersection(Lt l1, Lt l2) {
    auto [s1, e1] = l1;
    auto [s2, e2] = l2;
    auto A = max(s1.x, e1.x), AA = min(s1.x, e1.x);
    auto B = max(s1.y, e1.y), BB = min(s1.y, e1.y);
    auto C = max(s2.x, e2.x), CC = min(s2.x, e2.x);
    auto D = max(s2.y, e2.y), DD = min(s2.y, e2.y);
```

```
    if (A < CC || C < AA || B < DD || D < BB) {
        return {0, {}, {}};
    }
    if (sign(cross(e1 - s1, e2 - s2)) == 0) {
        if (sign(cross(s2, e1, s1)) != 0) {
            return {0, {}, {}};
        }
        Pt p1(max(AA, CC), max(BB, DD));
        Pt p2(min(A, C), min(B, D));
        if (!pointOnSegment(p1, l1)) {
            swap(p1.y, p2.y);
        }
        if (p1 == p2) {
            return {3, p1, p2};
        } else {
            return {2, p1, p2};
        }
    }
    auto cp1 = cross(s2 - s1, e2 - s1);
    auto cp2 = cross(s2 - e1, e2 - e1);
    auto cp3 = cross(s1 - s2, e1 - s2);
    auto cp4 = cross(s1 - e2, e1 - e2);
    if (sign(cp1 * cp2) == 1 || sign(cp3 * cp4) == 1) {
        return {0, {}, {}};
    }
    // 使用下方函数时请使用浮点数
    Pd p = lineIntersection(l1, l2);
    if (sign(cp1) != 0 && sign(cp2) != 0 && sign(cp3) != 0 && sign(cp4) != 0) {
        return {1, p, p};
    } else {
        return {3, p, p};
    }
}
```

如果不需要求交点, 那么使用快速排斥 + 跨立实验即可, 其中重叠, 相交于端点均视为相交.

```
template<typename T> bool segmentIntersection(Lt l1, Lt l2) {
    auto [s1, e1] = l1;
    auto [s2, e2] = l2;
    auto A = max(s1.x, e1.x), AA = min(s1.x, e1.x);
    auto B = max(s1.y, e1.y), BB = min(s1.y, e1.y);
    auto C = max(s2.x, e2.x), CC = min(s2.x, e2.x);
    auto D = max(s2.y, e2.y), DD = min(s2.y, e2.y);
    return A >= CC && B >= DD && C >= AA && D >= BB &&
           sign(cross(s1, s2, e1) * cross(s1, e1, e2)) == 1 &&
           sign(cross(s2, s1, e2) * cross(s2, e2, e1)) == 1;
}
```

平面圆相关 (浮点数处理)

点到圆的最近点 同时返回最近点与最近距离. **需要注意**, 当点为圆心时, 这样的点有无数个, 此时我们视作输入错误, 直接返回圆心.

```
pair<Pd, ld> pointToCircle(Pd p, Pd o, ld r) {
    Pd U = o, V = o;
    ld d = dis(p, o);
    if (sign(d) == 0) { // p 为圆心时返回圆心本身
        return {o, 0};
    }
    ld val1 = r * abs(o.x - p.x) / d;
    ld val2 = r * abs(o.y - p.y) / d * ((o.x - p.x) * (o.y - p.y) < 0 ? -1 : 1);
    U.x += val1, U.y += val2;
    V.x -= val1, V.y -= val2;
    if (dis(U, p) < dis(V, p)) {
        return {U, dis(U, p)};
    } else {
        return {V, dis(V, p)};
```

```
    }
}
```

根据圆心角获取圆上某点 将圆上最右侧的点以圆心为旋转中心, 逆时针旋转 rad 度.

```
Point<ld> getPoint(Point<ld> p, ld r, ld rad) {
    return {p.x + cos(rad) * r, p.y + sin(rad) * r};
}
```

直线是否与圆相交及交点 0 代表不相交;1 代表相切;2 代表相交.

```
tuple<int, Pd, Pd> lineCircleCross(Ld l, Pd o, ld r) {
    Pd P = project(o, l);
    ld d = dis(P, o), tmp = r * r - d * d;
    if (sign(tmp) == -1) {
        return {0, {}, {}};
    } else if (sign(tmp) == 0) {
        return {1, P, {}};
    }
    Pd vec = standardize(l.b - l.a) * sqrt(tmp);
    return {2, P + vec, P - vec};
}
```

线段是否与圆相交及交点 0 代表不相交;1 代表相切;2 代表相交于一个点;3 代表相交于两个点.

```
tuple<int, Pd, Pd> segmentCircleCross(Ld l, Pd o, ld r) {
    auto [type, U, V] = lineCircleCross(l, o, r);
    bool f1 = pointOnSegment(U, l), f2 = pointOnSegment(V, l);
    if (type == 1 && f1) {
        return {1, U, {}};
    } else if (type == 2 && f1 && f2) {
```

```
        return {3, U, V};
    } else if (type == 2 && f1) {
        return {2, U, {}};
    } else if (type == 2 && f2) {
        return {2, V, {}};
    } else {
        return {0, {}, {}};
    }
}
```

两圆是否相交及交点 0 代表内含;1 代表相离;2 代表相切;3 代表相交.

```
tuple<int, Pd, Pd> circleIntersection(Pd p1, ld r1, Pd p2, ld r2) {
    ld x1 = p1.x, x2 = p2.x, y1 = p1.y, y2 = p2.y, d = dis(p1, p2);
    if (sign(abs(r1 - r2) - d) == 1) {
        return {0, {}, {}};
    } else if (sign(r1 + r2 - d) == -1) {
        return {1, {}, {}};
    }
    ld a = r1 * (x1 - x2) * 2, b = r1 * (y1 - y2) * 2, c = r2 * r2 - r1 * r1 - d * d;
    ld p = a * a + b * b, q = -a * c * 2, r = c * c - b * b;
    ld cosa, sina, cosb, sinb;
    if (sign(d - (r1 + r2)) == 0 || sign(d - abs(r1 - r2)) == 0) {
        cosa = -q / p / 2;
        sina = sqrt(1 - cosa * cosa);
        Point<ld> p0 = {x1 + r1 * cosa, y1 + r1 * sina};
        if (sign(dis(p0, p2) - r2)) {
            p0.y = y1 - r1 * sina;
        }
        return {2, p0, p0};
    } else {
        ld delta = sqrt(q * q - p * r * 4);
        cosa = (delta - q) / p / 2;
```

```
        cosb = (-delta - q) / p / 2;
        sina = sqrt(1 - cosa * cosa);
        sinb = sqrt(1 - cosb * cosb);
        Pd ans1 = {x1 + r1 * cosa, y1 + r1 * sina};
        Pd ans2 = {x1 + r1 * cosb, y1 + r1 * sinb};
        if (sign(dis(ans1, p1) - r2)) ans1.y = y1 - r1 * sina;
        if (sign(dis(ans2, p2) - r2)) ans2.y = y1 - r1 * sinb;
        if (ans1 == ans2) ans1.y = y1 - r1 * sina;
        return {3, ans1, ans2};
    }
}
```

两圆相交面积 上述所言四种相交情况均可计算, 之所以不使用三角形面积计算公式是因为在计算过程中会出现" 负数面积 (扇形面积与三角形面积的符号关系会随圆的位置关系发生变化), 故公式全部重新推导, 这里采用的是扇形面积减去扇形内部的那个三角形的面积.

```
ld circleIntersectionArea(Pd p1, ld r1, Pd p2, ld r2) {
    ld x1 = p1.x, x2 = p2.x, y1 = p1.y, y2 = p2.y, d = dis(p1, p2);
    if (sign(abs(r1 - r2) - d) >= 0) {
        return PI * min(r1 * r1, r2 * r2);
    } else if (sign(r1 + r2 - d) == -1) {
        return 0;
    }
    ld theta1 = angle(r1, dis(p1, p2), r2);
    ld area1 = r1 * r1 * (theta1 - sin(theta1 * 2) / 2);
    ld theta2 = angle(r2, dis(p1, p2), r1);
    ld area2 = r2 * r2 * (theta2 - sin(theta2 * 2) / 2);
    return area1 + area2;
}
```

三点确定一圆

```
tuple<int, Pd, ld> getCircle(Pd A, Pd B, Pd C) {
    if (onLine(A, B, C)) { // 特判三点共线
        return {0, {}, 0};
    }
    Ld l1 = midSegment(Line{A, B});
    Ld l2 = midSegment(Line{A, C});
    Pd O = lineIntersection(l1, l2);
    return {1, O, dis(A, O)};
}
```

求解点到圆的切线数量与切点

```
pair<int, vector<Point<ld>>> tangent(Point<ld> p, Point<ld> A, ld r) {
    vector<Point<ld>> ans; // 储存切点
    Point<ld> u = A - p;
    ld d = sqrt(dot(u, u));
    if (d < r) {
        return {0, {}};
    } else if (sign(d - r) == 0) { // 点在圆上
        ans.push_back(u);
        return {1, ans};
    } else {
        ld ang = asin(r / d);
        ans.push_back(getPoint(A, r, -ang));
        ans.push_back(getPoint(A, r, ang));
        return {2, ans};
    }
}
```

求解两圆的内公, 外公切线数量与切点 同时返回公切线数量以及每个圆的切点.

```
tuple<int, vector<Point<ld>>, vector<Point<ld>>> tangent(Point<ld> A, ld Ar, Point<ld>
    vector<Point<ld>> a, b; // 储存切点
    if (Ar < Br) {
        swap(Ar, Br);
        swap(A, B);
        swap(a, b);
    }
    int d = disEx(A, B), dif = Ar - Br, sum = Ar + Br;
    if (d < dif * dif) { // 内含，无
        return {0, {}, {}};
    }
    ld base = atan2(B.y - A.y, B.x - A.x);
    if (d == 0 && Ar == Br) { // 完全重合，无数条外公切线
        return {-1, {}, {}};
    }
    if (d == dif * dif) { // 内切,1 条外公切线
        a.push_back(getPoint(A, Ar, base));
        b.push_back(getPoint(B, Br, base));
        return {1, a, b};
    }
    ld ang = acos(dif / sqrt(d));
    a.push_back(getPoint(A, Ar, base + ang)); // 保底 2 条外公切线
    a.push_back(getPoint(A, Ar, base - ang));
    b.push_back(getPoint(B, Br, base + ang));
    b.push_back(getPoint(B, Br, base - ang));
    if (d == sum * sum) { // 外切，多 1 条内公切线
        a.push_back(getPoint(A, Ar, base));
        b.push_back(getPoint(B, Br, base + PI));
    } else if (d > sum * sum) { // 相离，多 2 条内公切线
        ang = acos(sum / sqrt(d));
        a.push_back(getPoint(A, Ar, base + ang));
        a.push_back(getPoint(A, Ar, base - ang));
        b.push_back(getPoint(B, Br, base + ang + PI));
```

```
        b.push_back(getPoint(B, Br, base - ang + PI));
    }
    return {a.size(), a, b};
}
```

平面三角形相关 (浮点数处理)

三角形面积

```
ld area(Point<ld> a, Point<ld> b, Point<ld> c) {
    return abs(cross(b, c, a)) / 2;
}
```

三角形外心 三角形外接圆的圆心, 即三角形三边垂直平分线的交点.

```
template<typename T> Pt center1(Pt p1, Pt p2, Pt p3) { // 外心
    return lineIntersection(midSegment({p1, p2}), midSegment({p2, p3}));
}
```

三角形内心 三角形内切圆的圆心, 也是三角形三个内角的角平分线的交点.
其到三角形三边的距离相等.

```
Pd center2(Pd p1, Pd p2, Pd p3) { // 内心
    #define atan2(p) atan2(p.y, p.x) // 注意先后顺序
    Line<ld> U = {p1, {}}, V = {p2, {}};
    ld m, n, alpha;
    m = atan2((p2 - p1));
    n = atan2((p3 - p1));
    alpha = (m + n) / 2;
    U.b = {p1.x + cos(alpha), p1.y + sin(alpha)};
    m = atan2((p1 - p2));
    n = atan2((p3 - p2));
    alpha = (m + n) / 2;
```

```
    V.b = {p2.x + cos(alpha), p2.y + sin(alpha)};
    return lineIntersection(U, V);
}
```

三角形垂心 三角形的三条高线所在直线的交点. 锐角三角形的垂心在三角形内; 直角三角形的垂心在直角顶点上; 钝角三角形的垂心在三角形外.

```
Pd center3(Pd p1, Pd p2, Pd p3) { // 垂心
    Ld U = {p1, p1 + rotate(p2, p3)}; // 垂线
    Ld V = {p2, p2 + rotate(p1, p3)};
    return lineIntersection(U, V);
}
```

平面直线方程转换

浮点数计算直线的斜率 一般很少使用到这个函数, 因为斜率的取值不可控 (例如接近平行于 x, y 轴时). **需要注意**, 当直线平行于 y 轴时斜率为 `inf` .

```
template<typename T> ld slope(Pt p1, Pt p2) { // 斜率, 注意 inf 的情况
    return (p1.y - p2.y) / (p1.x - p2.x);
}
template<typename T> ld slope(Lt l) {
    return slope(l.a, l.b);
}
```

分数精确计算直线的斜率 调用分数四则运算精确计算斜率, 返回最简分数, 只适用于整数计算.

```
template<typename T> Frac<T> slopeEx(Pt p1, Pt p2) {
    Frac<T> U = p1.y - p2.y;
    Frac<T> V = p1.x - p2.x;
    return U / V; // 调用分数精确计算
}
```

两点式转一般式 返回由三个整数构成的方程, 在输入较大时可能找不到较小的满足题意的一组整数解. 可以处理平行于 x, y 轴, 两点共点的情况.

```
template<typename T> tuple<int, int, int> getfun(Lt p) {
    T A = p.a.y - p.b.y, B = p.b.x - p.a.x, C = p.a.x * A + p.a.y * B;
    if (A < 0) { // 符号调整
        A = -A, B = -B, C = -C;
    } else if (A == 0) {
        if (B < 0) {
            B = -B, C = -C;
        } else if (B == 0 && C < 0) {
            C = -C;
        }
    }
    if (A == 0) { // 数值计算
        if (B == 0) {
            C = 0; // 共点特判
        } else {
            T g = fgcd(abs(B), abs(C));
            B /= g, C /= g;
        }
    } else if (B == 0) {
        T g = fgcd(abs(A), abs(C));
        A /= g, C /= g;
    } else {
        T g = fgcd(fgcd(abs(A), abs(B)), abs(C));
        A /= g, B /= g, C /= g;
    }
    return tuple{A, B, C}; // Ax + By = C
}
```

一般式转两点式 由于整数点可能很大或者不存在, 故直接采用浮点数; 如果与 x, y 轴有交点则取交点. 可以处理平行于 x, y 轴的情况.

```
Line<ld> getfun(int A, int B, int C) { // Ax + By = C
    ld x1 = 0, y1 = 0, x2 = 0, y2 = 0;
    if (A && B) { // 正常
        if (C) {
            x1 = 0, y1 = 1. * C / B;
            y2 = 0, x2 = 1. * C / A;
        } else { // 过原点
            x1 = 1, y1 = 1. * -A / B;
            x2 = 0, y2 = 0;
        }
    } else if (A && !B) { // 垂直
        if (C) {
            y1 = 0, x1 = 1. * C / A;
            y2 = 1, x2 = x1;
        } else {
            x1 = 0, y1 = 1;
            x2 = 0, y2 = 0;
        }
    } else if (!A && B) { // 水平
        if (C) {
            x1 = 0, y1 = 1. * C / B;
            x2 = 1, y2 = y1;
        } else {
            x1 = 1, y1 = 0;
            x2 = 0, y2 = 0;
        }
    } else { // 不合法，请特判
        assert(false);
    }
    return {{x1, y1}, {x2, y2}};
}
```

抛物线与 x 轴是否相交及交点 0 代表没有交点;1 代表相切;2 代表有两个交点.

```
tuple<int, ld, ld> getAns(ld a, ld b, ld c) {
    ld delta = b * b - a * c * 4;
    if (delta < 0.) {
        return {0, 0, 0};
    }
    delta = sqrt(delta);
    ld ans1 = -(delta + b) / 2 / a;
    ld ans2 = (delta - b) / 2 / a;
    if (ans1 > ans2) {
        swap(ans1, ans2);
    }
    if (sign(delta) == 0) {
        return {1, ans2, 0};
    }
    return {2, ans1, ans2};
}
```

SMU_inch

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

#define endl '\n'
#define append push_back
#define pop pop_back
#define list vector
//#include <bits/extc++.h>
using namespace std;
//using namespace __gnu_pbds;
typedef long long ll;
typedef unsigned long long ull;
```

```
typedef pair<int, int> pii;
typedef pair<ll, ll> pll;
const int N = 2e5 + 5, inf = 0x3f3f3f3f, MOD = 998244353, mod = 1e9 + 7;
const ll llinf = 0x3f3f3f3f3f3f3f3f;
//const double PI=acos(-1);
typedef double db;
const db EPS = 1e-9;

// long double 的区分精度大约为 2^-64,1e-15~1e-18
// double 的区分精度大约为 2^-53,1e-12~1e-15
//精度问题，求两个 1e9 内的点的斜率，误差为 1e-18

inline int sign(db a) { return a < -EPS ? -1 : a > EPS; }

inline int cmp(db a, db b) { return sign(a - b); }

struct P {
    db x, y;

    P() {}

    P(db _x, db _y) : x(_x), y(_y) {}

    P operator+(P p) { return {x + p.x, y + p.y}; }

    P operator-(P p) { return {x - p.x, y - p.y}; }

    P operator*(db d) { return {x * d, y * d}; }

    P operator/(db d) { return {x / d, y / d}; }

    bool operator<(P p) const {
        int c = cmp(x, p.x);
```

```
        if (c)return c == -1;
        return cmp(y, p.y) == -1;
    }

    bool operator==(P o) const {
        //没有传递性
        return cmp(x, o.x) == 0 && cmp(y, o.y) == 0;
    }


    db dot(P p) { return x * p.x + y * p.y; }//点积, |a|*|b|*cos(an) 结果 大于 0, 两
    db det(P p) {
        return x * p.y - y * p.x;
    }//叉积, |a|*|b|*sin(an) an 为有向角, an 为 a 逆时针旋转多少度到 b, a x b = -

    db disTo(P p) { return (*this - p).abs(); }//两点距离
    db disTo2(P p) { return (*this - p).abs2(); }//两点距离的平方
    db alpha() { return atan2(y, x); }//求极角
    void readint() {
        int x_, y_;
        cin >> x_ >> y_;
        x = x_, y = y_;
    }//输入整数
    void readdb() { cin >> x >> y; }

    void write() { cout << "(" << x << ", " << y << ")" << endl; }//输出
    db abs() { return sqrt(abs2()); }//原点距离
    db abs2() { return x * x + y * y; }//原点距离的平方
    P rot90() { return P(-y, x); }//原点旋转 90
    int quad() const { return sign(y) == 1 || (sign(y) == 0 && sign(x) >= 0); }//判断点
    P unit() { return *this / abs(); }//单位向量

    P rot(db an) {
```

```
        return {x * cos(an) - y * sin(an), x * sin(an) + y * cos(an)};
    }// 绕原点旋转 an 度表示: (x+yi)(cos(an)+sin(an)i)

};

#define cross(p1, p2, p3)((p2.x-p1.x)*(p3.y-p1.y)-(p3.x-p1.x)*(p2.y-p1.y))
#define crossOp(p1, p2, p3) sign(cross(p1,p2,p3))

//如果 crossop 大于 0, 表示 p1,p2,p3 为逆时针关系, 小于 0 表示为顺时针关系, 等于
//也可以解释为 p3 在 p1,p2 的上方还是下方, 还是 p3 在直线 p1,p2 上
int cmp2(P A, P B) { return A.det(B) > 0 || (A.det(B) == 0 && A.abs2() < B.abs2()); }

bool chkLL(P p1, P p2, P q1, P q2) {
    ////两个线段是否平行
    db a1 = cross(q1, q2, p1);
    db a2 = -cross(q1, q2, p2);
    return sign(a1 + a2) != 0;
}

P isLL(P p1, P p2, P q1, P q2) {
    ////求出交点
    db a1 = cross(q1, q2, p1);
    db a2 = -cross(q1, q2, p2);
    return (p1 * a2 + p2 * a1) / (a1 + a2);
}

bool intersect(db l1, db r1, db l2, db r2) {
    ////判断 [l1,r1],[l2,r2] 是否相交
    if (l1 > r1) swap(l1, r1);
    if (l2 > r2) swap(l2, r2);
    return !(cmp(r1, l2) == -1 || cmp(r2, l1) == -1);
}
```

```
bool isSS(P p1, P p2, P q1, P q2) {
    ////线段是否相交
    return intersect(p1.x, p2.x, q1.x, q2.x) && intersect(p1.y, p2.y, q1.y, q2.y) &&
           crossOp(p1, p2, q1) * crossOp(p1, p2, q2) <= 0 && crossOp(q1, q2, p1) * cros
}

bool isSS_strict(P p1, P p2, P q1, P q2) {
    ////线段是否严格相交
    ////严格相交指：只有一个公共点，且不能端点相交，就是一个 x 的形状
    return crossOp(p1, p2, q1) * crossOp(p1, p2, q2) < 0 && crossOp(q1, q2, p1) * cross
}

bool isMiddle(db a, db b, db m) {
    ////点 m 在不在区间 [a,b] 上
    if (a > b)swap(a, b);
    return cmp(a, m) <= 0 && cmp(m, b) <= 0;
}

bool isMiddle(P a, P b, P m) {
    ////判断直线 q1q2 和直线 p1p2 的交点在不在线段 p1,p2 上，可以调用 isMiddle,
    return isMiddle(a.x, b.x, m.x) && isMiddle(a.y, b.y, m.y);
}

bool onSeg(P p1, P p2, P q) {
    ////p 在不在线段 p1,p2 上
    //可能精度有点问题
    return crossOp(p1, p2, q) == 0 && isMiddle(p1, p2, q);
}

bool onSeg_strict(P p1, P p2, P q) {
    ////p 是不是严格在线段 p1,p2 上
    return crossOp(p1, p2, q) == 0 && sign((q - p1).dot(p1 - p2)) * sign((q - p2).dot(p
}
```

```
P proj(P p1, P p2, P q) {
    ////求 q 到 p1p2 的垂足，且 p1!=p2
    if (p1 == p2)return p1;
    P dir = p2 - p1;
    return p1 + dir * (dir.dot(q - p1) / dir.abs2());
}

P reflect(P p1, P p2, P q) {
    ////求 q 关于 p1p2 的反射
    return proj(p1, p2, q) * 2 - q;
}

db nearest(P p1, P p2, P q) {
    ////求 q 到线段 p1p2 的最小距离
    if (p1 == p2)return p1.disTo(q);
    P h = proj(p1, p2, q);
    if (isMiddle(p1, p2, h))return q.disTo(h);
    return min(p1.disTo(q), p2.disTo(q));
}

db disSS(P p1, P p2, P q1, P q2) {
    ////求线段 p1p2 到 q1q2 的距离
    if (isSS(p1, p2, q1, q2))return 0;
    return min(min(nearest(p1, p2, q1), nearest(p1, p2, q1)), min(nearest(q1, q2, p1),
}
//极角排序
/*
sort(p,p+n,[&](const P &a,const P &b){
    int qa = a.quad(),qb=b.quad();
    if(qa!=qb) return qa<qb;
    return sign(a.det(b)) > 0;
});
```

```
*/
bool cmp1(P a, const P &b) {
    int qa = a.quad(), qb = b.quad();
    if (qa != qb) return qa < qb;
    return sign(a.det(b)) > 0;
}

int type(P o1, db r1, P o2, db r2) {
    ///求两个圆的关系
    /// 4 : 相离
    /// 3 : 外切
    /// 2 : 相交
    /// 1 : 内切
    /// 0 : 内含
    db d = o1.disTo(o2);
    if (cmp(d, r1 + r2) == 1) return 4;
    if (cmp(d, r1 + r2) == 0) return 3;
    if (cmp(d, abs(r1 - r2)) == 1) return 2;
    if (cmp(d, abs(r1 - r2)) == 0) return 1;
    return 0;
}

vector<P> isCL(P o, db r, P p1, P p2) {
    ///求圆和直线的交点，返回的两个点属于 p1->p2 方向
    if (cmp(abs((o - p1).det(p2 - p1) / p1.disTo(p2)), r) > 0) return {};
    db x = (p1 - o).dot(p2 - p1), y = (p2 - p1).abs2(), d = x * x - y * ((p1 - o).abs2(
    d = max(d, (db) 0.0);
    P m = p1 - (p2 - p1) * (x / y), dr = (p2 - p1) * (sqrt(d) / y);
    return {m - dr, m + dr};
}

vector<P> isCC(P o1, db r1, P o2, db r2) {
    ///两个圆的交点，需要判断两个圆是否全等
```

```
    ///返回的交点沿着第一个圆的逆时针方向
    db d = o1.disTo(o2);
    if (cmp(d, r1 + r2) == 1)return {};
    if (cmp(d, abs(r1 - r2)) == -1)return {};
    d = min(d, r1 + r2);
    db y = (r1 * r1 + d * d - r2 * r2) / (2 * d), x = sqrt(r1 * r1 - y * y);
    P dr = (o2 - o1).unit();
    P q1 = o1 + dr * y, q2 = dr.rot90() * x;
    return {q1 - q2, q1 + q2};
}

vector<pair<P, P>> tancCC(P o1, db r1, P o2, db r2) {
    ///两个圆的外切线，如果需要内切线，把 r2 传入负值即可，如果需要点到圆的切
    P d = o2 - o1;
    db dr = r1 - r2, d2 = d.abs2(), h2 = d2 - dr * dr;
    if (sign(d2) == 0 || sign(h2) < 0)return {};
    h2 = max((db) 0.0, h2);
    vector<pair<P, P>> ret;
    for (db sign: {-1, 1}) {
        P v = (d * dr + d.rot90() * sqrt(h2) * sign) / d2;
        ret.push_back({o1 + v * r1, o2 + v * r2});
    }
    if (sign(h2) == 0)ret.pop_back();
    return ret;
}

db rad(P p1, P p2) {
    ///求两个向量的夹角弧度
    return atan2l(p1.det(p2), p1.dot(p2));
}

db areaCT(P o, db r, P p1, P p2) {
    ///圆和其中一个顶点是圆心的三角形的面积交，返回有向面积
```

```
    p1 = p1 - o;
    p2 = p2 - o;
    vector<P> is = isCL(P(0, 0), r, p1, p2);
    if (is.empty()) return r * r * rad(p1, p2) / 2;
    bool b1 = cmp(p1.abs2(), r * r) == 1, b2 = cmp(p2.abs2(), r * r) == 1;
    if (b1 && b2) {
        P md = (is[0] + is[1]) / 2;
        if (sign((p1 - md).dot(p2 - md)) <= 0)
            return r * r * (rad(p1, is[0]) + rad(is[1], p2)) / 2 + is[0].det(is[1]) / 2
        else return r * r * rad(p1, p2) / 2;
    }
    if (b1) return (r * r * rad(p1, is[0]) + is[0].det(p2)) / 2;
    if (b2) return (p1.det(is[1]) + r * r * rad(is[1], p2)) / 2;
    return p1.det(p2) / 2;
}


P inCenter(P A, P B, P C) {
    ///三角形内心
    double a = (B - C).abs(), b = (C - A).abs(), c = (A - B).abs();
    return (A * a + B * b + C * c) / (a + b + c);
}

P circumCenter(P a, P b, P c) {
    ///三角形外心
    P bb = b - a, cc = c - a;
    double db = bb.abs2(), dc = cc.abs2(), d = 2 * bb.det(cc);
    return a - P(bb.y * dc - cc.y * db, cc.x * db - bb.x * dc) / d;
}

P othroCenter(P a, P b, P c) {
    ///三角形垂心
    P ba = b - a, ca = c - a, bc = b - c;
```

```
    double Y = ba.y * ca.y * bc.y,
           A = ca.x * ba.y - ba.x * ca.y,
           x0 = (Y + ca.x * ba.y * b.x - ba.x * ca.y * c.x) / A,
           y0 = -ba.x * (x0 - c.x) / ba.y + ca.y;
    return {x0, y0};
}

pair<P, db> min_circle(vector<P> ps) {
    ///最小圆覆盖，给定若干个点，求最小的一个圆能够覆盖这些点，复杂度为 O(n)
    random_shuffle(ps.begin(), ps.end());
    int n = ps.size();
    P o = ps[0];
    db r = 0;
    for (int i = 1; i < n; ++i) {
        if (o.disTo(ps[i]) > r + EPS)
            o = ps[i], r = 0;
        for (int j = 0; j < i; ++j)
            if (o.disTo(ps[j]) > r + EPS) {
                o = (ps[i] + ps[j]) / 2;
                r = o.disTo(ps[i]);
                for (int k = 0; k < j; ++k)
                    if (o.disTo(ps[k]) > r + EPS) {
                        o = circumCenter(ps[i], ps[j], ps[k]);
                        r = o.disTo(ps[i]);
                    }
            }
    }
    return {o, r};
}


db area(vector<P> ps) {
    ////计算多边形面积
```

```
    db ret = 0;
    int n = ps.size();
    for (int i = 0; i < ps.size(); ++i) {
        ret += ps[i].det(ps[(i + 1) % n]);
    }
    return ret / 2;
}


int containP(const vector<P> &ps, P p) {
    ////判断点是否在多边形内部
    ////如果返回 0: 不在内部;1: 在边界上;2: 在内部
    int n = ps.size(), ret = 0;
    for (int i = 0; i < n; ++i) {
        P u = ps[i], v = ps[(i + 1) % n];
        if (onSeg(u, v, p)) return 1;
        if (cmp(u.y, v.y) <= 0) swap(u, v);
        if (cmp(p.y, u.y) > 0 || cmp(p.y, v.y) <= 0)continue;
        ret ^= crossOp(p, u, v) > 0;
    }
    return ret * 2;
}


vector<P> convexHull(vector<P> ps) {
    ////求严格凸包
    int n = ps.size();
    if (n <= 1)return ps;
    sort(ps.begin(), ps.end());
    vector<P> qs(n * 2);
    int k = 0;
    for (int i = 0; i < n; qs[k++] = ps[i++]) {//求下凸壳
        while (k > 1 && crossOp(qs[k - 2], qs[k - 1], ps[i]) <= 0)--k;
```

```
    }
    for (int i = n - 2, t = k; i >= 0; qs[k++] = ps[i--]) {//求上凸壳
        while (k > t && crossOp(qs[k - 2], qs[k - 1], ps[i]) <= 0)--k;
    }
    qs.resize(k - 1);
    return qs;
}

vector<P> convexHullnonstrict(vector<P> ps) {
    ////求不严格凸包，需要先去重
    int n = ps.size();
    if (n <= 1)return ps;
    sort(ps.begin(), ps.end());
    vector<P> qs(n * 2);
    int k = 0;
    for (int i = 0; i < n; qs[k++] = ps[i++]) {//求下凸壳
        while (k > 1 && crossOp(qs[k - 2], qs[k - 1], ps[i]) <= 0)--k;
    }
    for (int i = n - 2, t = k; i >= 0; qs[k++] = ps[i--]) {//求上凸壳
        while (k > t && crossOp(qs[k - 2], qs[k - 1], ps[i]) <= 0)--k;
    }
    qs.resize(k - 1);
    return qs;
}

db convexDiamter(vector<P> ps) {
    ////求凸包最大直径
    int n = ps.size();
    if (n <= 1)return 0;
    int is = 0;
    int js = 0;
    for (int k = 1; k < n; ++k) {
        is = ps[k] < ps[is] ? k : is, js = ps[js] < ps[k] ? k : js;
```

```
    }
    int i = is, j = js;
    db ret = ps[i].disTo(ps[j]);
    do {
        if ((ps[(i + 1) % n] - ps[i]).det(ps[(j + 1) % n] - ps[j]) >= 0)
            (++j) %= n;
        else
            (++i) %= n;
        ret = max(ret, ps[i].disTo(ps[j]));
    } while (i != is || j != js);
    return ret;
}


vector<P> convexCut(const vector<P> &ps, P q1, P q2) {
    ////用直线切割 ps, 返回切线左边的点以及交点
    vector<P> qs;
    int n = ps.size();
    for (int i = 0; i < n; ++i) {
        P p1 = ps[i], p2 = ps[(i + 1) % n];
        int d1 = crossOp(q1, q2, p1), d2 = crossOp(q1, q2, p2);
        if (d1 >= 0) qs.push_back(p1);
        if (d1 * d2 < 0) qs.push_back(isLL(p1, p2, q1, q2));
    }
    return qs;
}

vector<P> isLD(const vector<P> &ps, P q1, P q2) {
    ////返回直线和多边形的所有交点
    int n = ps.size();
    vector<P> qs;
    for (int i = 0; i < n; ++i) {
        if (crossOp(q1, q2, ps[i]) == 0)qs.push_back(ps[i]);
```

```
        if (crossOp(q1, q2, ps[i]) * crossOp(q1, q2, ps[(i + 1) % n]) < 0)
            qs.push_back(isLL(q1, q2, ps[i], ps[(i + 1) % n]));
    }
    sort(qs.begin(), qs.end());
    qs.erase(unique(qs.begin(), qs.end()), qs.end());
    return qs;
}

vector<P> isSD(const vector<P> &ps, P q1, P q2) {
    ////返回直线和多边形的所有交点
    int n = ps.size();
    vector<P> qs;
    qs.push_back(q1);
    qs.push_back(q2);
    for (int i = 0; i < n; ++i) {
        if (crossOp(q1, q2, ps[i]) == 0)qs.push_back(ps[i]);
        if (crossOp(q1, q2, ps[i]) * crossOp(q1, q2, ps[(i + 1) % n]) < 0)
            qs.push_back(isLL(q1, q2, ps[i], ps[(i + 1) % n]));
    }
    sort(qs.begin(), qs.end());
    qs.erase(unique(qs.begin(), qs.end()), qs.end());
    int s = -1, t = -1;
    for (int i = 0; i < n; ++i) {
        if (q1 == qs[i])s = i;
        if (q2 == qs[i])t = i;
    }
    if (s > t)swap(s, t);
    vector<P> ks;
    for (int i = s; i < t; ++i) {
        ks.push_back(qs[i]);
    }
    return ks;
}
```

```
bool containSeg(vector<P> ps, P p1, P p2) {
    ////判断线段是否在内部
    vector<P> qs = isSD(ps, p1, p2);
    int n = qs.size();
    for (int i = 0; i < n - 1; ++i) {
        P m = (qs[i] + qs[i + 1]) / 2;
        if (containP(qs, m) == 0)return false;
    }
    return true;
}

vector<P> Minkowski(vector<P> A, vector<P> B) {
    vector<P> C(A.size() + B.size() + 1), v1(A.size()), v2(B.size());
    for (int i = 0; i < (int) A.size(); i++)v1[i] = A[(i + 1) % A.size()] - A[i];
    for (int i = 0; i < (int) B.size(); i++)v2[i] = B[(i + 1) % B.size()] - B[i];
    int cnt = 0;
    C[cnt] = (A[0] + B[0]);
    int p1 = 0, p2 = 0;
    while (p1 < (int) A.size() && p2 < (int) B.size()) {
        ++cnt;
        if (sign(v1[p1].det(v2[p2])) >= 0)
            C[cnt] = C[cnt - 1] + v1[p1++];
        else
            C[cnt] = C[cnt - 1] + v2[p2++];
    }
    while (p1 < (int) A.size()) {
        ++cnt;
        C[cnt] = C[cnt - 1] + v1[p1++];
    }
    while (p2 < (int) B.size()) {
        ++cnt;
```

```
        C[cnt] = C[cnt - 1] + v2[p2++];
    }
    return C;
}

bool containPs(const vector<P> &ts, P q) {
    ///判断点集是否在线段内，要保证 ps[0]={0,0};
    int ps = upper_bound(ts.begin(), ts.end(), q, cmp2) - ts.begin() - 1;
    return (crossOp(ts[ps], ts[(ps + 1) % ts.size()], q) >= 0);
}


void solve() {


}


int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
//    freopen(".\\Template\\CHECK\\data.in", "r", stdin);
//    freopen(".\\Template\\CHECK\\std.out", "w", stdout);
    int cases;
    cin >> cases;
    while (cases--)
        solve();
}
```

三维几何及常见例题

三维几何必要初始化

点线面封装

```
struct Point3 {
    ld x, y, z;
```

```
    Point3(ld x_ = 0, ld y_ = 0, ld z_ = 0) : x(x_), y(y_), z(z_) {}
    Point3 &operator+=(Point3 p) & {
        return x += p.x, y += p.y, z += p.z, *this;
    }
    Point3 &operator-=(Point3 p) & {
        return x -= p.x, y -= p.y, z -= p.z, *this;
    }
    Point3 &operator*=(Point3 p) & {
        return x *= p.x, y *= p.y, z *= p.z, *this;
    }
    Point3 &operator*=(ld t) & {
        return x *= t, y *= t, z *= t, *this;
    }
    Point3 &operator/=(ld t) & {
        return x /= t, y /= t, z /= t, *this;
    }
    friend Point3 operator+(Point3 a, Point3 b) { return a += b; }
    friend Point3 operator-(Point3 a, Point3 b) { return a -= b; }
    friend Point3 operator*(Point3 a, Point3 b) { return a *= b; }
    friend Point3 operator*(Point3 a, ld b) { return a *= b; }
    friend Point3 operator*(ld a, Point3 b) { return b *= a; }
    friend Point3 operator/(Point3 a, ld b) { return a /= b; }
    friend auto &operator>>(istream &is, Point3 &p) {
        return is >> p.x >> p.y >> p.z;
    }
    friend auto &operator<<(ostream &os, Point3 p) {
        return os << "(" << p.x << ", " << p.y << ", " << p.z << ")";
    }
};
struct Line3 {
    Point3 a, b;
};
struct Plane {
```

```
    Point3 u, v, w;
};
```

其他函数

```
ld len(P3 p) { // 原点到当前点的距离计算
    return sqrt(p.x * p.x + p.y * p.y + p.z * p.z);
}
P3 crossEx(P3 a, P3 b) { // 叉乘
    P3 ans;
    ans.x = a.y * b.z - a.z * b.y;
    ans.y = a.z * b.x - a.x * b.z;
    ans.z = a.x * b.y - a.y * b.x;
    return ans;
}
ld cross(P3 a, P3 b) {
    return len(crossEx(a, b));
}
ld dot(P3 a, P3 b) { // 点乘
    return a.x * b.x + a.y * b.y + a.z * b.z;
}
P3 getVec(Plane s) { // 获取平面法向量
    return crossEx(s.u - s.v, s.v - s.w);
}
ld dis(P3 a, P3 b) { // 三维欧几里得距离公式
    ld val = (a.x - b.x) * (a.x - b.x) + (a.y - b.y) * (a.y - b.y) + (a.z - b.z) * (a.z
    return sqrt(val);
}
P3 standardize(P3 vec) { // 将三维向量转换为单位向量
    return vec / len(vec);
}
```

三维点线面相关

空间三点是否共线 其中第二个函数是专门用来判断给定的三个点能否构成平面的, 因为不共线的三点才能构成平面.

```
bool onLine(P3 p1, P3 p2, P3 p3) { // 三点是否共线
    return sign(cross(p1 - p2, p3 - p2)) == 0;
}
bool onLine(Plane s) {
    return onLine(s.u, s.v, s.w);
}
```

四点是否共面

```
bool onPlane(P3 p1, P3 p2, P3 p3, P3 p4) { // 四点是否共面
    ld val = dot(getVec({p1, p2, p3}), p4 - p1);
    return sign(val) == 0;
}
```

空间点是否在线段上

```
bool pointOnSegment(P3 p, L3 l) {
    return sign(cross(p - l.a, p - l.b)) == 0 && min(l.a.x, l.b.x) <= p.x &&
           p.x <= max(l.a.x, l.b.x) && min(l.a.y, l.b.y) <= p.y && p.y <= max(l.a.y, l.
           min(l.a.z, l.b.z) <= p.z && p.z <= max(l.a.z, l.b.z);
}
bool pointOnSegmentEx(P3 p, L3 l) { // pointOnSegment 去除端点版
    return sign(cross(p - l.a, p - l.b)) == 0 && min(l.a.x, l.b.x) < p.x &&
           p.x < max(l.a.x, l.b.x) && min(l.a.y, l.b.y) < p.y && p.y < max(l.a.y, l.b.y
           min(l.a.z, l.b.z) < p.z && p.z < max(l.a.z, l.b.z);
}
```

空间两点是否在线段同侧 当给定的两点与线段不共面, 点在线段上时返回 $false$.

```
bool pointOnSegmentSide(P3 p1, P3 p2, L3 l) {
    if (!onPlane(p1, p2, l.a, l.b)) { // 特判不共面
        return 0;
    }
    ld val = dot(crossEx(l.a - l.b, p1 - l.b), crossEx(l.a - l.b, p2 - l.b));
    return sign(val) == 1;
}
```

两点是否在平面同侧 点在平面上时返回 $false$.

```
bool pointOnPlaneSide(P3 p1, P3 p2, Plane s) {
    ld val = dot(getVec(s), p1 - s.u) * dot(getVec(s), p2 - s.u);
    return sign(val) == 1;
}
```

空间两直线是否平行/垂直

```
bool lineParallel(L3 l1, L3 l2) {
    return sign(cross(l1.a - l1.b, l2.a - l2.b)) == 0;
}
bool lineVertical(L3 l1, L3 l2) {
    return sign(dot(l1.a - l1.b, l2.a - l2.b)) == 0;
}
```

两平面是否平行/垂直

```
bool planeParallel(Plane s1, Plane s2) {
    ld val = cross(getVec(s1), getVec(s2));
    return sign(val) == 0;
}
bool planeVertical(Plane s1, Plane s2) {
```

```
    ld val = dot(getVec(s1), getVec(s2));
    return sign(val) == 0;
}
```

空间两直线是否是同一条

```
bool same(L3 l1, L3 l2) {
    return lineParallel(l1, l2) && lineParallel({l1.a, l2.b}, {l1.b, l2.a});
}
```

两平面是否是同一个

```
bool same(Plane s1, Plane s2) {
    return onPlane(s1.u, s2.u, s2.v, s2.w) && onPlane(s1.v, s2.u, s2.v, s2.w) &&
           onPlane(s1.w, s2.u, s2.v, s2.w);
}
```

直线是否与平面平行

```
bool linePlaneParallel(L3 l, Plane s) {
    ld val = dot(l.a - l.b, getVec(s));
    return sign(val) == 0;
}
```

空间两线段是否相交

```
bool segmentIntersection(L3 l1, L3 l2) { // 重叠，相交于端点均视为相交
    if (!onPlane(l1.a, l1.b, l2.a, l2.b)) { // 特判不共面
        return 0;
    }
    if (!onLine(l1.a, l1.b, l2.a) || !onLine(l1.a, l1.b, l2.b)) {
        return !pointOnSegmentSide(l1.a, l1.b, l2) && !pointOnSegmentSide(l2.a, l2.b, l
    }
```

```
    return pointOnSegment(l1.a, l2) || pointOnSegment(l1.b, l2) || pointOnSegment(l2.a,
           pointOnSegment(l2.b, l2);
}
bool segmentIntersection1(L3 l1, L3 l2) { // 重叠，相交于端点不视为相交
    return onPlane(l1.a, l1.b, l2.a, l2.b) && !pointOnSegmentSide(l1.a, l1.b, l2) &&
           !pointOnSegmentSide(l2.a, l2.b, l1);
}
```

空间两直线是否相交及交点 当两直线不共面, 两直线平行时返回 $false$.

```
pair<bool, P3> lineIntersection(L3 l1, L3 l2) {
    if (!onPlane(l1.a, l1.b, l2.a, l2.b) || lineParallel(l1, l2)) {
        return {0, {}};
    }
    auto [s1, e1] = l1;
    auto [s2, e2] = l2;
    ld val = 0;
    if (!onPlane(l1.a, l1.b, {0, 0, 0}, {0, 0, 1})) {
        val = ((s1.x - s2.x) * (s2.y - e2.y) - (s1.y - s2.y) * (s2.x - e2.x)) /
              ((s1.x - e1.x) * (s2.y - e2.y) - (s1.y - e1.y) * (s2.x - e2.x));
    } else if (!onPlane(l1.a, l1.b, {0, 0, 0}, {0, 1, 0})) {
        val = ((s1.x - s2.x) * (s2.z - e2.z) - (s1.z - s2.z) * (s2.x - e2.x)) /
              ((s1.x - e1.x) * (s2.z - e2.z) - (s1.z - e1.z) * (s2.x - e2.x));
    } else {
        val = ((s1.y - s2.y) * (s2.z - e2.z) - (s1.z - s2.z) * (s2.y - e2.y)) /
              ((s1.y - e1.y) * (s2.z - e2.z) - (s1.z - e1.z) * (s2.y - e2.y));
    }
    return {1, s1 + (e1 - s1) * val};
}
```

直线与平面是否相交及交点 当直线与平面平行, 给定的点构不成平面时返回 $false$.

```
pair<bool, P3> linePlaneCross(L3 l, Plane s) {
    if (linePlaneParallel(l, s)) {
        return {0, {}};
    }
    P3 vec = getVec(s);
    P3 U = vec * (s.u - l.a), V = vec * (l.b - l.a);
    ld val = (U.x + U.y + U.z) / (V.x + V.y + V.z);
    return {1, l.a + (l.b - l.a) * val};
}
```

两平面是否相交及交线 当两平面平行, 两平面为同一个时返回 $false$.

```
pair<bool, L3> planeIntersection(Plane s1, Plane s2) {
    if (planeParallel(s1, s2) || same(s1, s2)) {
        return {0, {}};
    }
    P3 U = linePlaneParallel({s2.u, s2.v}, s1) ? linePlaneCross({s2.v, s2.w}, s1).secon
                                               : linePlaneCross({s2.u, s2.v}, s1).secon
    P3 V = linePlaneParallel({s2.w, s2.u}, s1) ? linePlaneCross({s2.v, s2.w}, s1).secon
                                               : linePlaneCross({s2.w, s2.u}, s1).secon
    return {1, {U, V}};
}
```

点到直线的最近点与最近距离

```
pair<ld, P3> pointToLine(P3 p, L3 l) {
    ld val = cross(p - l.a, l.a - l.b) / dis(l.a, l.b); // 面积除以底边长
    ld val1 = dot(p - l.a, l.a - l.b) / dis(l.a, l.b);
    return {val, l.a + val1 * standardize(l.a - l.b)};
}
```

点到平面的最近点与最近距离

```
pair<ld, P3> pointToPlane(P3 p, Plane s) {
    P3 vec = getVec(s);
    ld val = dot(vec, p - s.u);
    val = abs(val) / len(vec); // 面积除以底边长
    return {val, p - val * standardize(vec)};
}
```

空间两直线的最近距离与最近点对

```
tuple<ld, P3, P3> lineToLine(L3 l1, L3 l2) {
    P3 vec = crossEx(l1.a - l1.b, l2.a - l2.b); // 计算同时垂直于两直线的向量
    ld val = abs(dot(l1.a - l2.a, vec)) / len(vec);
    P3 U = l1.b - l1.a, V = l2.b - l2.a;
    vec = crossEx(U, V);
    ld p = dot(vec, vec);
    ld t1 = dot(crossEx(l2.a - l1.a, V), vec) / p;
    ld t2 = dot(crossEx(l2.a - l1.a, U), vec) / p;
    return {val, l1.a + (l1.b - l1.a) * t1, l2.a + (l2.b - l2.a) * t2};
}
```

三维角度与弧度

空间两直线夹角的 cos 值 任意位置的空间两直线.

```
ld lineCos(L3 l1, L3 l2) {
    return dot(l1.a - l1.b, l2.a - l2.b) / len(l1.a - l1.b) / len(l2.a - l2.b);
}
```

空间两平面夹角的 cos 值

```
ld planeCos(Plane s1, Plane s2) {
    P3 U = getVec(s1), V = getVec(s2);
```

```
    return dot(U, V) / len(U) / len(V);
}
```

直线与平面夹角的 sin 值

```
ld linePlaneSin(L3 l, Plane s) {
    P3 vec = getVec(s);
    return dot(l.a - l.b, vec) / len(l.a - l.b) / len(vec);
}
```

空间多边形

正 N 棱锥体积公式 棱锥通用体积公式 $V = \frac{1}{3}Sh$, 当其恰好是棱长为 l 的正 n 棱锥时, 有公式 $V = \frac{l^3 \cdot n}{12 \tan \frac{\pi}{n}} \cdot \sqrt{1 - \frac{1}{4 \cdot \sin^2 \frac{\pi}{n}}}$.

```
ld V(ld l, int n) { // 正 n 棱锥体积公式
    return l * l * l * n / (12 * tan(PI / n)) * sqrt(1 - 1 / (4 * sin(PI / n) * sin(PI
}
```

四面体体积

```
ld V(P3 a, P3 b, P3 c, P3 d) {
    return abs(dot(d - a, crossEx(b - a, c - a))) / 6;
}
```

点是否在空间三角形上 点位于边界上时返回 $false$.

```
bool pointOnTriangle(P3 p, P3 p1, P3 p2, P3 p3) {
    return pointOnSegmentSide(p, p1, {p2, p3}) && pointOnSegmentSide(p, p2, {p1, p3}) &
           pointOnSegmentSide(p, p3, {p1, p2});
}
```

线段是否与空间三角形相交及交点 只有交点在空间三角形内部时才视作相交.

```
pair<bool, P3> segmentOnTriangle(P3 l, P3 r, P3 p1, P3 p2, P3 p3) {
    P3 x = crossEx(p2 - p1, p3 - p1);
    if (sign(dot(x, r - l)) == 0) {
        return {0, {}};
    }
    ld t = dot(x, p1 - l) / dot(x, r - l);
    if (t < 0 || t - 1 > 0) { // 不在线段上
        return {0, {}};
    }
    bool type = pointOnTriangle(l + (r - l) * t, p1, p2, p3);
    if (type) {
        return {1, l + (r - l) * t};
    } else {
        return {0, {}};
    }
}
```

空间三角形是否相交 相交线段在空间三角形内部时才视作相交.

```
bool triangleIntersection(vector<P3> a, vector<P3> b) {
    for (int i = 0; i < 3; i++) {
        if (segmentOnTriangle(b[i], b[(i + 1) % 3], a[0], a[1], a[2]).first) {
            return 1;
        }
        if (segmentOnTriangle(a[i], a[(i + 1) % 3], b[0], b[1], b[2]).first) {
            return 1;
        }
    }
    return 0;
}
```

常用结论

平面几何结论归档

- `hypot` 函数可以直接计算直角三角形的斜边长;
- **边心距**是指正多边形的外接圆圆心到正多边形某一边的距离, 边长为 s 的正 n 角形的边心距公式为 $a = \dfrac{t}{2 \cdot \tan \frac{\pi}{n}}$, 外接圆半径为 R 的正 n 角形的边心距公式为 $a = R \cdot \cos \dfrac{\pi}{n}$;
- **三角形外接圆半径**为 $\dfrac{a}{2 \sin A} = \dfrac{abc}{4S}$, 其中 S 为三角形面积, 内切圆半径为 $\dfrac{2S}{a+b+c}$;
- 由小正三角形拼成的大正三角形, 耗费的小三角形数量即为构成一条边的小三角形数量的平方. 如下图, 总数量即为 4^2 See.
- 正 n 边形圆心角为 $\dfrac{360^\circ}{n}$, 圆周角为 $\dfrac{180^\circ}{n}$. 定义正 n 边形上的三个顶点 A, B 和 C(可以不相邻), 使得 $\angle ABC = \theta$, 当 $n \leq 360$ 时,θ 可以取 1° 到 179° 间的任何一个整数 See.
- 某一点 B 到直线 AC 的距离公式为 $\dfrac{|\vec{BA} \times \vec{BC}|}{|AC|}$, 等价于 $\dfrac{|aX + bY + c|}{\sqrt{a^2 + b^2}}$.
- `atan(y / x)` 函数仅用于计算第一, 四象限的值, 而 `atan2(y, x)` 则允许计算所有四个象限的正反切, 在使用这个函数时, 需要尽量保证 x 和 y 的类型为整数型, 如果使用浮点数, 实测会慢十倍.
- 在平面上有奇数个点 $A_0, A_1, \ldots, A_n$ 以及一个点 X_0 , 构造 X_1 使得 X_0, X_1 关于 A_0 对称, 构造 X_2 使得 X_1, X_2 关于 A_1 对称,„ 构造 X_j 使得 X_{j-1}, X_j 关于 $A_{(j-1) \bmod n}$ 对称. 那么周期为 $2n$, 即 A_0 与 A_{2n} 共点,A_1 与 A_{2n+1} 共点 See .
- 已知 $A\ (x_A, y_A)$ 和 $X\ (x_X, y_X)$ 两点及这两点的坐标, 构造 Y 使

得 X, Y 关于 A 对称, 那么 Y 的坐标为 $(2 \cdot x_A - x_X, 2 \cdot y_A - y_X)$.

- **海伦公式:** 已知三角形三边长 a, b 和 c , 定义 $p = \dfrac{a+b+c}{2}$, 则 $S_{\triangle} = \sqrt{p(p-a)(p-b)(p-c)}$, 在使用时需要注意越界问题, 本质是铅锤定理, 一般多使用叉乘计算三角形面积而不使用该公式.
- 棱台体积 $V = \frac{1}{3}(S_1 + S_2 + \sqrt{S_1 S_2}) \cdot h$, 其中 S_1, S_2 为上下底面积.
- 正棱台侧面积 $\frac{1}{2}(C_1 + C_2) \cdot L$, 其中 C_1, C_2 为上下底周长,L 为斜高 (上下底对应的平行边的距离).
- 球面积 $4\pi r^2$, 体积 $\frac{4}{3}\pi r^3$.
- 正三角形面积 $\dfrac{\sqrt{3}a^2}{4}$, 正四面体面积 $\dfrac{\sqrt{2}a^3}{12}$.
- 设扇形对应的圆心角弧度为 θ , 则面积为 $S = \frac{\theta}{2} \cdot R^2$.

立体几何结论归档

- 已知向量 $\vec{r} = \{x, y, z\}$, 则该向量的三个方向余弦为 $\cos\alpha = \dfrac{x}{|\vec{r}|} = \dfrac{x}{\sqrt{x^2+y^2+z^2}}$; $\cos\beta = \dfrac{y}{|\vec{r}|}$; $\cos\gamma = \dfrac{z}{|\vec{r}|}$. 其中 $\alpha, \beta, \gamma \in [0, \pi]$,$\cos^2\alpha + \cos^2\beta + \cos^2\gamma = 1$.

常用例题

将平面某点旋转任意角度 题意: 给定平面上一点 (a, b) , 输出将其逆时针旋转 d 度之后的坐标.

```
signed main() {
    int a, b, d;
    cin >> a >> b >> d;
```

```
    ld l = hypot(a, b); // 库函数，求直角三角形的斜边
    ld alpha = atan2(b, a) + toArc(d);

    cout << l * cos(alpha) << " " << l * sin(alpha) << endl;
}
```

平面最近点对 (set 解) 借助 set，在严格 $\mathcal{O}(N \log N)$ 复杂度内求解，比常见的分治法稍快.

```
template<typename T> T sqr(T x) {
    return x * x;
}

using V = Point<int>;
signed main() {
    int n;
    cin >> n;

    vector<V> in(n);
    for (auto &it : in) {
        cin >> it;
    }

    int dis = disEx(in[0], in[1]); // 设定阈值
    sort(in.begin(), in.end());

    set<V> S;
    for (int i = 0, h = 0; i < n; i++) {
        V now = {in[i].y, in[i].x};
        while (dis && dis <= sqr(in[i].x - in[h].x)) { // 删除超过阈值的点
            S.erase({in[h].y, in[h].x});
            h++;
        }
```

```
        auto it = S.lower_bound(now);
        for (auto k = it; k != S.end() && sqr(k->x - now.x) < dis; k++) {
            dis = min(dis, disEx(*k, now));
        }
        if (it != S.begin()) {
            for (auto k = prev(it); sqr(k->x - now.x) < dis; k--) {
                dis = min(dis, disEx(*k, now));
                if (k == S.begin()) break;
            }
        }
        S.insert(now);
    }
    cout << sqrt(dis) << endl;
}
```

平面若干点能构成的最大四边形的面积 (简单版, 暴力枚举) 题意: 平面上存在若干个点, 保证没有两点重合, 没有三点共线, 你需要从中选出四个点, 使得它们构成的四边形面积是最大的, 注意这里能组成的四边形可以不是凸四边形.

暴力枚举其中一条对角线后枚举剩余两个点,$\mathcal{O}(N^3)$.

```
signed main() {
    int n;
    cin >> n;
    vector<Pi> in(n);
    for (auto &it : in) {
        cin >> it;
    }
    ld ans = 0;
    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) { // 枚举对角线
            ld l = 0, r = 0;
            for (int k = 0; k < n; k++) { // 枚举第三点
```

```
                if (k == i || k == j) continue;
                if (pointOnLineLeft(in[k], {in[i], in[j]})) {
                    l = max(l, triangleS(in[k], in[j], in[i]));
                } else {
                    r = max(r, triangleS(in[k], in[j], in[i]));
                }
            }
            if (l * r != 0) { // 确保构成的是四边形
                ans = max(ans, l + r);
            }
        }
    }
    cout << ans << endl;
}
```

平面若干点能构成的最大四边形的面积 (困难版, 分类讨论 + 旋转卡壳) 题意: 平面上存在若干个点, 可能存在多点重合, 共线的情况, 你需要从中选出四个点, 使得它们构成的四边形面积是最大的, 注意这里能组成的四边形可以不是凸四边形, 可以是退化的四边形.

当凸包大小 ≤ 2 时, 说明是退化的四边形, 答案直接为 0 ; 大小恰好为 3 时, 说明是凹四边形, 我们枚举不在凸包上的那一点, 将两个三角形面积相减既可得到答案; 大小恰好为 4 时, 说明是凸四边形, 使用旋转卡壳求解.

```
using V = Point<int>;
signed main() {
    int Task = 1;
    for (cin >> Task; Task; Task--) {
        int n;
        cin >> n;

        vector<V> in_(n);
        for (auto &it : in_) {
            cin >> it;
```

```
        }
        auto in = staticConvexHull(in_, 0);
        n = in.size();

        int ans = 0;
        if (n > 3) {
            ans = rotatingCalipers(in);
        } else if (n == 3) {
            int area = triangleAreaEx(in[0], in[1], in[2]);
            for (auto it : in_) {
                if (it == in[0] || it == in[1] || it == in[2]) continue;
                int Min = min({triangleAreaEx(it, in[0], in[1]), triangleAreaEx(it, in[
                ans = max(ans, area - Min);
            }
        }

        cout << ans / 2;
        if (ans % 2) {
            cout << ".5";
        }
        cout << endl;
    }
}
```

线段将多边形切割为几个部分 题意: 给定平面上一线段与一个任意多边形, 求解线段将多边形切割为几个部分; 保证线段的端点不在多边形内, 多边形边上, 多边形顶点不位于线段上, 多边形的边不与线段重叠; 多边形端点按逆时针顺序给出. 下方的几个样例均合法, 答案均为 3 .

当线段切割多边形时, 本质是与多边形的边交于两个点, 或者说是与多边形的两条边相交, 设交点数目为 x , 那么答案即为 $\frac{x}{2}+1$. 于是, 我们只需要计算交点数量即可, 先判断某一条边是否与线段相交, 再判断边的两个端点是否位于线段两侧.

```
signed main() {
    Pi s, e;
    cin >> s >> e; // 读入线段

    int n;
    cin >> n;
    vector<Pi> in(n);
    for (auto &it : in) {
        cin >> it; // 读入多边形端点
    }

    int cnt = 0;
    for (int i = 0; i < n; i++) {
        Pi x = in[i], y = in[(i + 1) % n];
        cnt += (pointNotOnLineSide(x, y, {s, e}) && segmentIntersection(Line{x, y}, {s,
    }
    cout << cnt / 2 + 1 << endl;
}
```

平面若干点能否构成凸包 (暴力枚举) 题意: 给定平面上若干个点, 判断其是否构成凸包 See .

可以直接使用凸包模板, 但是代码较长; 在这里我们使用暴力枚举试点, 也能以 $\mathcal{O}(N)$ 的复杂度通过. 当两个向量的叉乘 ≤ 0 时说明其夹角大于等于 180 度, 使用这一点即可判定.

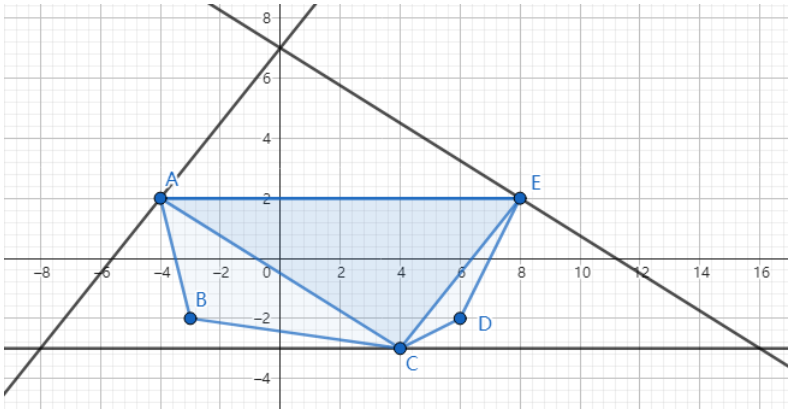
```
signed main() {
    int n;
    cin >> n;

    vector<Point<ld>> in(n);
    for (auto &it : in) {
        cin >> it;
    }
```

```
    for (int i = 0; i < n; i++) {
        auto A = in[(i - 1 + n) % n];
        auto B = in[i];
        auto C = in[(i + 1) % n];
        if (cross(A - B, C - B) > 0) {
            cout << "No\n";
            return 0;
        }
    }
    cout << "Yes\n";
}
```

凸包上的点能构成的最大三角形 (暴力枚举) 可以直接使用凸包模板, 但是代码较长; 在这里我们使用暴力枚举试点, 也能以 $\mathcal{O}(N)$ 的复杂度通过.

> 另外补充一点性质: 所求三角形的反互补三角形一定包含了凸包上的所有点 (可以在边界). 通俗的说, 构成的三角形是这个反互补三角形的中点三角形. 如下图所示, 点 A 不在 $\triangle BCE$ 的反互补三角形内部, 故 $\triangle BCE$ 不是最大三角形;$\triangle ACE$ 才是.




```
signed main() {
    int n;
    cin >> n;
```

```
    vector<Point<int>> in(n);
    for (auto &it : in) {
        cin >> it;
    }

    #define S(x, y, z) triangleAreaEx(in[x], in[y], in[z])

    int i = 0, j = 1, k = 2;
    while (true) {
        int val = S(i, j, k);
        if (S((i + 1) % n, j, k) > val) {
            i = (i + 1) % n;
        } else if (S((i - 1 + n) % n, j, k) > val) {
            i = (i - 1 + n) % n;
        } else if (S(i, (j + 1) % n, k) > val) {
            j = (j + 1) % n;
        } else if (S(i, (j - 1 + n) % n, k) > val) {
            j = (j - 1 + n) % n;
        } else if (S(i, j, (k + 1) % n) > val) {
            k = (k + 1) % n;
        } else if (S(i, j, (k - 1 + n) % n) > val) {
            k = (k - 1 + n) % n;
        } else {
            break;
        }
    }
    cout << i + 1 << " " << j + 1 << " " << k + 1 << endl;
}
```

凸包上的点能构成的最大四角形的面积 (旋转卡壳) 由于是凸包上的点, 所以保证了四边形一定是凸四边形, 时间复杂度 $\mathcal{O}(N^2)$.

```cpp
template<typename T> T rotatingCalipers(vector<Point<T>> &p) {
    #define S(x, y, z) triangleAreaEx(p[x], p[y], p[z])
    int n = p.size();
    T ans = 0;
    auto nxt = [&](int i) -> int {
        return i == n - 1 ? 0 : i + 1;
    };
    for (int i = 0; i < n; i++) {
        int p1 = nxt(i), p2 = nxt(nxt(nxt(i)));
        for (int j = nxt(nxt(i)); nxt(j) != i; j = nxt(j)) {
            while (nxt(p1) != j && S(i, j, nxt(p1)) > S(i, j, p1)) {
                p1 = nxt(p1);
            }
            if (p2 == j) {
                p2 = nxt(p2);
            }
            while (nxt(p2) != i && S(i, j, nxt(p2)) > S(i, j, p2)) {
                p2 = nxt(p2);
            }
            ans = max(ans, S(i, j, p1) + S(i, j, p2));
        }
    }
    return ans;
    #undef S
}
```

判断一个凸包是否完全在另一个凸包内 题意: 给定一个凸多边形 A 和一个凸多边形 B , 询问 B 是否被 A 包含, 分别判断严格/不严格包含.例题.

考虑严格包含, 使用 A 点集计算出凸包 T_1 , 使用 A, B 两个点集计算出不严格凸包 T_2 , 如果包含, 那么 T_1 应该与 T_2 完全相等; 考虑不严格包含, 在计算凸包 T_2 时严格即可. 最终以 $\mathcal{O}(N)$ 复杂度求解, 且代码不算很长.

/END/

多边形相关

平面多边形

两向量构成的平面四边形有向面积

```
template<typename T> T areaEx(Point<T> p1, Point<T> p2, Point<T> p3) {
    return cross(b, c, a);
}
```

判断四个点能否组成矩形/正方形 可以处理浮点数, 共点的情况. 返回分为三种情况:2 代表构成正方形;1 代表构成矩形;0 代表其他情况.

```
template<typename T> int isSquare(vector<Pt> x) {
    sort(x.begin(), x.end());
    if (equal(dis(x[0], x[1]), dis(x[2], x[3])) && sign(dis(x[0], x[1])) &&
        equal(dis(x[0], x[2]), dis(x[1], x[3])) && sign(dis(x[0], x[2])) &&
        lineParallel(Lt{x[0], x[1]}, Lt{x[2], x[3]}) &&
        lineParallel(Lt{x[0], x[2]}, Lt{x[1], x[3]}) &&
        lineVertical(Lt{x[0], x[1]}, Lt{x[0], x[2]})) {
        return equal(dis(x[0], x[1]), dis(x[0], x[2])) ? 2 : 1;
    }
    return 0;
}
```

点是否在任意多边形内 射线法判定,t 为穿越次数, 当其为奇数时即代表点在多边形内部; 返回 2 代表点在多边形边界上.

```
template<typename T> int pointInPolygon(Point<T> a, vector<Point<T>> p) {
    int n = p.size();
    for (int i = 0; i < n; i++) {
        if (pointOnSegment(a, Line{p[i], p[(i + 1) % n]})) {
            return 2;
        }
```

```
    }
    int t = 0;
    for (int i = 0; i < n; i++) {
        auto u = p[i], v = p[(i + 1) % n];
        if (u.x < a.x && v.x >= a.x && pointOnLineLeft(a, Line{v, u})) {
            t ^= 1;
        }
        if (u.x >= a.x && v.x < a.x && pointOnLineLeft(a, Line{u, v})) {
            t ^= 1;
        }
    }
    return t == 1;
}
```

线段是否在任意多边形内部

```
template<typename T>
bool segmentInPolygon(Line<T> l, vector<Point<T>> p) {
// 线段与多边形边界不相交且两端点都在多边形内部
#define L(x, y) pointOnLineLeft(x, y)
    int n = p.size();
    if (!pointInPolygon(l.a, p)) return false;
    if (!pointInPolygon(l.b, p)) return false;
    for (int i = 0; i < n; i++) {
        auto u = p[i];
        auto v = p[(i + 1) % n];
        auto w = p[(i + 2) % n];
        auto [t, p1, p2] = segmentIntersection(l, Line(u, v));
        if (t == 1) return false;
        if (t == 0) continue;
        if (t == 2) {
            if (pointOnSegment(v, l) && v != l.a && v != l.b) {
                if (cross(v - u, w - v) > 0) {
```

```
                return false;
            }
        }
    } else {
        if (p1 != u && p1 != v) {
            if (L(l.a, Line(v, u)) || L(l.b, Line(v, u))) {
                return false;
            }
        } else if (p1 == v) {
            if (l.a == v) {
                if (L(u, l)) {
                    if (L(w, l) && L(w, Line(u, v))) {
                        return false;
                    }
                } else {
                    if (L(w, l) || L(w, Line(u, v))) {
                        return false;
                    }
                }
            } else if (l.b == v) {
                if (L(u, Line(l.b, l.a))) {
                    if (L(w, Line(l.b, l.a)) && L(w, Line(u, v))) {
                        return false;
                    }
                } else {
                    if (L(w, Line(l.b, l.a)) || L(w, Line(u, v))) {
                        return false;
                    }
                }
            } else {
                if (L(u, l)) {
                    if (L(w, Line(l.b, l.a)) || L(w, Line(u, v))) {
                        return false;
```

```
                    }
                } else {
                    if (L(w, l) || L(w, Line(u, v))) {
                        return false;
                    }
                }
            }
        }
    }
    return true;
}
```

任意多边形的面积

```
template<typename T> ld area(vector<Point<T>> P) {
    int n = P.size();
    ld ans = 0;
    for (int i = 0; i < n; i++) {
        ans += cross(P[i], P[(i + 1) % n]);
    }
    return ans / 2.0;
}
```

皮克定理 绘制在方格纸上的多边形面积公式可以表示为 $S = n + \frac{s}{2} - 1$, 其中 n 表示多边形内部的点数,s 表示多边形边界上的点数. 一条线段上的点数为 $\gcd(|x_1 - x_2|, |y_1 - y_2|) + 1$.

任意多边形上/内的网格点个数 (仅能处理整数) 皮克定理用.

```
int onPolygonGrid(vector<Point<int>> p) { // 多边形上
    int n = p.size(), ans = 0;
```

```
    for (int i = 0; i < n; i++) {
        auto a = p[i], b = p[(i + 1) % n];
        ans += gcd(abs(a.x - b.x), abs(a.y - b.y));
    }
    return ans;
}
int inPolygonGrid(vector<Point<int>> p) { // 多边形内
    int n = p.size(), ans = 0;
    for (int i = 0; i < n; i++) {
        auto a = p[i], b = p[(i + 1) % n], c = p[(i + 2) % n];
        ans += b.y * (a.x - c.x);
    }
    ans = abs(ans);
    return (ans - onPolygonGrid(p)) / 2 + 1;
}
```

二维凸包

获取二维静态凸包 (Andrew 算法) `flag` 用于判定凸包边上的点, 重复的顶点是否要加入到凸包中, 为 0 时代表加入凸包 (不严格); 为 1 时不加入凸包 (严格). 时间复杂度为 $\mathcal{O}(N \log N)$.

```
template<typename T> vector<Point<T>> staticConvexHull(vector<Point<T>> A, int flag = 1
    int n = A.size();
    if (n <= 2) { // 特判
        return A;
    }
    vector<Point<T>> ans(n * 2);
    sort(A.begin(), A.end());
    int now = -1;
    for (int i = 0; i < n; i++) { // 维护下凸包
        while (now > 0 && cross(A[i], ans[now], ans[now - 1]) <= 0) {
            now--;
```

```
        }
        ans[++now] = A[i];
    }
    int pre = now;
    for (int i = n - 2; i >= 0; i--) { // 维护上凸包
        while (now > pre && cross(A[i], ans[now], ans[now - 1]) <= 0) {
            now--;
        }
        ans[++now] = A[i];
    }
    ans.resize(now);
    return ans;
}
```

二维动态凸包 固定为 int 型, 需要重新书写 Line 函数,cmp 用于判定边界情况. 可以处理如下两个要求:

- 动态插入点 (x, y) 到当前凸包中;
- 判断点 (x, y) 是否在凸包上或是在内部 (包括边界).

```
template<typename T> bool turnRight(Pt a, Pt b) {
    return cross(a, b) < 0 || (cross(a, b) == 0 && dot(a, b) < 0);
}
struct Line {
    static int cmp;
    mutable Point<int> a, b;
    friend bool operator<(Line x, Line y) {
        return cmp ? x.a < y.a : turnRight(x.b, y.b);
    }
    friend auto &operator<<(ostream &os, Line l) {
        return os << "<" << l.a << ", " << l.b << ">";
    }
};
```

```cpp
int Line::cmp = 1;
struct UpperConvexHull : set<Line> {
    bool contains(const Point<int> &p) const {
        auto it = lower_bound({p, 0});
        if (it != end() && it->a == p) return true;
        if (it != begin() && it != end() && cross(prev(it)->b, p - prev(it)->a) <= 0) {
            return true;
        }
        return false;
    }
    void add(const Point<int> &p) {
        if (contains(p)) return;
        auto it = lower_bound({p, 0});
        for (; it != end(); it = erase(it)) {
            if (turnRight(it->a - p, it->b)) {
                break;
            }
        }
        for (; it != begin() && prev(it) != begin(); erase(prev(it))) {
            if (turnRight(prev(prev(it))->b, p - prev(prev(it))->a)) {
                break;
            }
        }
        if (it != begin()) {
            prev(it)->b = p - prev(it)->a;
        }
        if (it == end()) {
            insert({p, {0, -1}});
        } else {
            insert({p, it->a - p});
        }
    }
};
```

```
struct ConvexHull {
    UpperConvexHull up, low;
    bool empty() const {
        return up.empty();
    }
    bool contains(const Point<int> &p) const {
        Line::cmp = 1;
        return up.contains(p) && low.contains(-p);
    }
    void add(const Point<int> &p) {
        Line::cmp = 1;
        up.add(p);
        low.add(-p);
    }
    bool isIntersect(int A, int B, int C) const {
        Line::cmp = 0;
        if (empty()) return false;
        Point<int> k = {-B, A};
        if (k.x < 0) k = -k;
        if (k.x == 0 && k.y < 0) k.y = -k.y;
        Point<int> P = up.upper_bound({{0, 0}, k})->a;
        Point<int> Q = -low.upper_bound({{0, 0}, k})->a;
        return sign(A * P.x + B * P.y - C) * sign(A * Q.x + B * Q.y - C) > 0;
    }
    friend ostream &operator<<(ostream &out, const ConvexHull &ch) {
        for (const auto &line : ch.up) out << "(" << line.a.x << "," << line.a.y << ")"
        cout << "/";
        for (const auto &line : ch.low) out << "(" << -line.a.x << "," << -line.a.y <<
        return out;
    }
};
```

点与凸包的位置关系 0 代表点在凸包外面;1 代表在凸壳上;2 代表在凸包内部.

```
template<typename T> int contains(Point<T> p, vector<Point<T>> A) {
    int n = A.size();
    bool in = false;
    for (int i = 0; i < n; i++) {
        Point<T> a = A[i] - p, b = A[(i + 1) % n] - p;
        if (a.y > b.y) {
            swap(a, b);
        }
        if (a.y <= 0 && 0 < b.y && cross(a, b) < 0) {
            in = !in;
        }
        if (cross(a, b) == 0 && dot(a, b) <= 0) {
            return 1;
        }
    }
    return in ? 2 : 0;
}
```

闵可夫斯基和 计算两个凸包合成的大凸包.

```
template<typename T> vector<Point<T>> mincowski(vector<Point<T>> P1, vector<Point<T>> P
    int n = P1.size(), m = P2.size();
    vector<Point<T>> V1(n), V2(m);
    for (int i = 0; i < n; i++) {
        V1[i] = P1[(i + 1) % n] - P1[i];
    }
    for (int i = 0; i < m; i++) {
        V2[i] = P2[(i + 1) % m] - P2[i];
    }
    vector<Point<T>> ans = {P1.front() + P2.front()};
    int t = 0, i = 0, j = 0;
```

```
    while (i < n && j < m) {
        Point<T> val = sign(cross(V1[i], V2[j])) > 0 ? V1[i++] : V2[j++];
        ans.push_back(ans.back() + val);
    }
    while (i < n) ans.push_back(ans.back() + V1[i++]);
    while (j < m) ans.push_back(ans.back() + V2[j++]);
    return ans;
}
```

半平面交 计算多条直线左边平面部分的交集.

```
template<typename T> vector<Point<T>> halfcut(vector<Line<T>> lines) {
    sort(lines.begin(), lines.end(), [&](auto l1, auto l2) {
        auto d1 = l1.b - l1.a;
        auto d2 = l2.b - l2.a;
        if (sign(d1) != sign(d2)) {
            return sign(d1) == 1;
        }
        return cross(d1, d2) > 0;
    });
    deque<Line<T>> ls;
    deque<Point<T>> ps;
    for (auto l : lines) {
        if (ls.empty()) {
            ls.push_back(l);
            continue;
        }
        while (!ps.empty() && !pointOnLineLeft(ps.back(), l)) {
            ps.pop_back();
            ls.pop_back();
        }
        while (!ps.empty() && !pointOnLineLeft(ps[0], l)) {
            ps.pop_front();
```

```
            ls.pop_front();
        }
        if (cross(l.b - l.a, ls.back().b - ls.back().a) == 0) {
            if (dot(l.b - l.a, ls.back().b - ls.back().a) > 0) {
                if (!pointOnLineLeft(ls.back().a, l)) {
                    assert(ls.size() == 1);
                    ls[0] = l;
                }
                continue;
            }
            return {};
        }
        ps.push_back(lineIntersection(ls.back(), l));
        ls.push_back(l);
    }
    while (!ps.empty() && !pointOnLineLeft(ps.back(), ls[0])) {
        ps.pop_back();
        ls.pop_back();
    }
    if (ls.size() <= 2) {
        return {};
    }
    ps.push_back(lineIntersection(ls[0], ls.back()));
    return vector(ps.begin(), ps.end());
}
```

博弈论

巴什博奕

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 $X(1 \leq X \leq M)$ 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 $X(1 \leq X \leq M)$, 后报数的人需要比前者所报数大 $Y(1 \leq Y \leq M)$, 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).

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 $X(a \leq X \leq b)$ 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 $K \in \mathbb{N}^+, a \leq R_2 \leq b$) 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);

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 $Sum_N = 0$ 时先手必败, 反之先手必胜.

Nim 游戏具体取法

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 X_i 颗石头, 取走 $A_i - X_i$ 颗石头** (这里取小于号是因为至少要取走 1 颗石子).

Moores Nim 游戏 (Nim-K 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, $One_1, One_2, \dots, One_N$ 均不为 $K+1$ 的倍数.

Anti-Nim 游戏 (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 $Sum_N = 0$ (也可看作" 且有偶数堆);
- 至少有一堆的石头数量大于 1 , 且 $Sum_N \neq 0$.

阶梯 - Nim 博弈

> 有 N 级台阶, 每一级台阶上均有一定数量的石子, 给出每一级石子的数量, 两名玩家轮流行动, 按以下规则操作石子:
>
> 规定: 每人每次任选一级台阶, 拿走正整数颗石子放到下一级台阶中, 已经拿到地面上的石子不能再拿, 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.

对奇数台阶做传统 `Nim` 博弈, 当 $Sum_N = 0$ 时先手必败, 反之先手必胜.**

SG 游戏 (有向图游戏)

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 n, m, a[N], num[N];
int sg(int x) {
    if (num[x] != -1) return num[x];

    unordered_set<int> S;
    for (int i = 1; i <= m; ++ i)
        if(x >= a[i])
            S.insert(sg(x - a[i]));
```

```
    for (int i = 0; ; ++ i)
        if (S.count(i) == 0)
            return num[x] = i;
}
void Solve() {
    cin >> m;
    for (int i = 1; i <= m; ++ i) cin >> a[i];
    cin >> n;

    int ans = 0; memset(num, -1, sizeof num);
    for (int i = 1; i <= n; ++ i) {
        int x; cin >> x;
        ans ^= sg(x);
    }

    if (ans == 0) no;
    else yes;
}
```

Anti-SG 游戏 (反 SG 游戏)

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.

Laskers-Nim 游戏 (Multi-SG 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, 每人每次任选以下规定的一种操作石子:

- 任选一堆, 取走正整数颗石子;
- 任选数量大于 2 的一堆, 分成两堆非空石子.

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.

本题使用 SG 函数求解,SG 值定义为:

$$SG(x) = \begin{cases} x-1 & ,x \mod 4 = 0 \\ x & ,x \mod 4 = 1 \\ x & ,x \mod 4 = 2 \\ x+1 & ,x \mod 4 = 3 \end{cases}$$

Every-SG 游戏

给出一个有向无环图, 其中 K 个顶点上放置了石子, 两名玩家轮流行动, 按以下规则操作石子:

移动图上所有还能够移动的石子;

无法移动石子的一方出局. 双方均采用最优策略, 询问谁会获胜.

定义 $step$ 为某一局游戏至多需要经过的回合数.

以下局面先手必胜:$step$ 为奇数.

威佐夫博弈

有两堆石子, 给出每一堆的石子数量, 两名玩家轮流行动, 每人每次任选以下规定的一种操作石子:

- 任选一堆, 取走正整数颗石子;
- 从两队中同时取走正整数颗石子.

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.

以下局面先手必败:

$(1,2),(3,5),(4,7),(6,10),$ 具体而言, 每一对的第一个数为此前没出现过的最小整数, 第二个数为第一个数加上 $1,2,3,4,$.

更一般地, 对于第 k 对数, 第一个数为 $First_k = \left\lfloor \frac{k*(1+\sqrt{5})}{2} \right\rfloor$, 第二个数为 $Second_k = First_k + k$.

其中, 在两堆石子的数量均大于 10^9 次时, 由于需要使用高精度计算, 我们需要人为定义 $\frac{1+\sqrt{5}}{2}$ 的取值为 $lorry = 1.6180339887498948482045868 34$.

```
const double lorry = (sqrt(5.0) + 1.0) / 2.0;
//const double lorry = 1.6180339887498948482045868 34;
void Solve() {
    int n, m; cin >> n >> m;
    if (n < m) swap(n, m);
    double x = n - m;
    if ((int)(lorry * x) == m) cout << "lose\n";
    else cout << "win\n";
}
```

斐波那契博弈

有一堆石子, 数量为 N , 两名玩家轮流行动, 按以下规则取石子:

先手第 1 次可以取任意多颗, 但不能全部取完, 此后每人取的石子数不能超过上个人的两倍, 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.

当且仅当 N 为斐波那契数时先手必败.

```
int fib[100] = {1, 2};
map<int, bool> mp;
void Force() {
  for (int i = 2; i <= 86; ++ i) fib[i] = fib[i - 1] + fib[i - 2];
    for (int i = 0; i <= 86; ++ i) mp[fib[i]] = 1;
}
void Solve() {
    int n; cin >> n;
    if (mp[n] == 1) cout << "lose\n";
    else cout << "win\n";
}
```

树上删边游戏

> 给出一棵 N 个节点的有根树, 两名玩家轮流行动, 按以下规则操作:
>
> 选择任意一棵子树并删除 (即删去任意一条边, 不与根相连的部分会同步被删去);
>
> 删掉最后一棵子树的一方获胜 (换句话说, 删去根节点的一方失败). 双方均采用最优策略, 询问谁会获胜.

结论: 当根节点 SG 值非 1 时先手必胜.

相较于传统 SG 值的定义, 本题的 SG 函数值定义为:

- 叶子节点的 SG 值为 0 .
- 非叶子节点的 SG 值为其所有孩子节点 SG 值 $+1$ 的异或和.

```
auto dfs = [&](auto self, int x, int fa) -> int {
    int x = 0;
    for (auto y : ver[x]) {
        if (y == fa) continue;
```

```
        x ^= self(self, y, x);
    }
    return x + 1;
};
cout << (dfs(dfs, 1, 0) == 1 ? "Bob\n" : "Alice\n");
```

无向图删边游戏 (Fusion Principle 定理)

> 给出一张 N 个节点的无向联通图, 有一个点作为图的根, 两名玩家轮流行动, 按以下规则操作:
>
> 选择任意一条边删除, 不与根相连的部分会同步被删去;
>
> 删掉最后一条边的一方获胜. 双方均采用最优策略, 询问谁会获胜.

- **对于奇环, 我们将其缩成一个新点 + 一条新边;**
- **对于偶环, 我们将其缩成一个新点;**
- **所有连接到原来环上的边全部与新点相连.**

此时, 本模型转化为" 树上删边游戏.

/END/

网络流

最大流

Dinic 解 使用 Dinic 算法, 理论最坏复杂度为 $\mathcal{O}(N^2M)$, 例题范围:$N = 1200,\ m = 5 \times 10^3$. 一般步骤:BFS 建立分层图, 无回溯 DFS 寻找所有可行的增广路径. 封装: 求从点 S 到点 T 的最大流.

```
template<typename T> struct Flow_ {
    const int n;
    const T inf = numeric_limits<T>::max();
```

```
    struct Edge {
        int to;
        T w;
        Edge(int to, T w) : to(to), w(w) {}
    };
    vector<Edge> ver;
    vector<vector<int>> h;
    vector<int> cur, d;

    Flow_(int n) : n(n + 1), h(n + 1) {}
    void add(int u, int v, T c) {
        h[u].push_back(ver.size());
        ver.emplace_back(v, c);
        h[v].push_back(ver.size());
        ver.emplace_back(u, 0);
    }
    bool bfs(int s, int t) {
        d.assign(n, -1);
        d[s] = 0;
        queue<int> q;
        q.push(s);
        while (!q.empty()) {
            auto x = q.front();
            q.pop();
            for (auto it : h[x]) {
                auto [y, w] = ver[it];
                if (w && d[y] == -1) {
                    d[y] = d[x] + 1;
                    if (y == t) return true;
                    q.push(y);
                }
            }
        }
```

```
        return false;
    }
    T dfs(int u, int t, T f) {
        if (u == t) return f;
        auto r = f;
        for (int &i = cur[u]; i < h[u].size(); i++) {
            auto j = h[u][i];
            auto &[v, c] = ver[j];
            auto &[u, rc] = ver[j ^ 1];
            if (c && d[v] == d[u] + 1) {
                auto a = dfs(v, t, std::min(r, c));
                c -= a;
                rc += a;
                r -= a;
                if (!r) return f;
            }
        }
        return f - r;
    }
    T work(int s, int t) {
        T ans = 0;
        while (bfs(s, t)) {
            cur.assign(n, 0);
            ans += dfs(s, t, inf);
        }
        return ans;
    }
};
using Flow = Flow_<int>;
```

预流推进 HLPP 理论最坏复杂度为 $\mathcal{O}(N^2\sqrt{M})$，例题范围:$N = 1200$, $m = 1.2 \times 10^5$.

```cpp
template<typename T> struct PushRelabel {
    const int inf = 0x3f3f3f3f;
    const T INF = 0x3f3f3f3f3f3f3f3f;
    struct Edge {
        int to, cap, flow, anti;
        Edge(int v = 0, int w = 0, int id = 0) : to(v), cap(w), flow(0), anti(id) {}
    };
    vector<vector<Edge>> e;
    vector<vector<int>> gap;
    vector<T> ex; // 超额流
    vector<bool> ingap;
    vector<int> h;
    int n, gobalcnt, maxH = 0;
    T maxflow = 0;

    PushRelabel(int n) : n(n), e(n + 1), ex(n + 1), gap(n + 1) {}
    void addedge(int u, int v, int w) {
        e[u].push_back({v, w, (int)e[v].size()});
        e[v].push_back({u, 0, (int)e[u].size() - 1});
    }
    void PushEdge(int u, Edge &edge) {
        int v = edge.to, d = min(ex[u], 1LL * edge.cap - edge.flow);
        ex[u] -= d;
        ex[v] += d;
        edge.flow += d;
        e[v][edge.anti].flow -= d;
        if (h[v] != inf && d > 0 && ex[v] == d && !ingap[v]) {
            ++gobalcnt;
            gap[h[v]].push_back(v);
            ingap[v] = 1;
        }
    }
    void PushPoint(int u) {
```

```
        for (auto k = e[u].begin(); k != e[u].end(); k++) {
            if (h[k->to] + 1 == h[u] && k->cap > k->flow) {
                PushEdge(u, *k);
                if (!ex[u]) break;
            }
        }
        if (!ex[u]) return;
        if (gap[h[u]].empty()) {
            for (int i = h[u] + 1; i <= min(maxH, n); i++) {
                for (auto v : gap[i]) {
                    ingap[v] = 0;
                }
                gap[i].clear();
            }
        }
        h[u] = inf;
        for (auto [to, cap, flow, anti] : e[u]) {
            if (cap > flow) {
                h[u] = min(h[u], h[to] + 1);
            }
        }
        if (h[u] >= n) return;
        maxH = max(maxH, h[u]);
        if (!ingap[u]) {
            gap[h[u]].push_back(u);
            ingap[u] = 1;
        }
    }
    void init(int t, bool f = 1) {
        ingap.assign(n + 1, 0);
        for (int i = 1; i <= maxH; i++) {
            gap[i].clear();
        }
```

```
        gobalcnt = 0, maxH = 0;
        queue<int> q;
        h.assign(n + 1, inf);
        h[t] = 0, q.push(t);
        while (q.size()) {
            int u = q.front();
            q.pop(), maxH = h[u];
            for (auto &[v, cap, flow, anti] : e[u]) {
                if (h[v] == inf && e[v][anti].cap > e[v][anti].flow) {
                    h[v] = h[u] + 1;
                    q.push(v);
                    if (f) {
                        gap[h[v]].push_back(v);
                        ingap[v] = 1;
                    }
                }
            }
        }
    }
    T work(int s, int t) {
        init(t, 0);
        if (h[s] == inf) return maxflow;
        h[s] = n;
        ex[s] = INF;
        ex[t] = -INF;
        for (auto k = e[s].begin(); k != e[s].end(); k++) {
            PushEdge(s, *k);
        }
        while (maxH > 0) {
            if (gap[maxH].empty()) {
                maxH--;
                continue;
            }
```

```
            int u = gap[maxH].back();
            gap[maxH].pop_back();
            ingap[u] = 0;
            PushPoint(u);
            if (gobalcnt >= 10 * n) {
                init(t);
            }
        }
        ex[s] -= INF;
        ex[t] += INF;
        return maxflow = ex[t];
    }
};
```

最小割

基础模型: 构筑二分图, 左半部 n 个点代表盈利项目, 右半部 m 个点代表材料成本, 收益为盈利之和减去成本之和, 求最大收益.

建图: 建立源点 S 向左半部连边, 建立汇点 T 向右半部连边, 如果某个项目需要某个材料, 则新增一条容量 $+\infty$ 的跨部边.

割边: 放弃某个项目则断开 S 至该项目的边, 购买某个原料则断开该原料至 T 的边, 最终的图一定不存在从 S 到 T 的路径, 此时我们得到二分图的一个 $S-T$ 割. 此时最小割即为求解最大流, 边权之和减去最大流即为最大收益.

```
signed main() {
    int n, m;
    cin >> n >> m;

    int S = n + m + 1, T = n + m + 2;
    Flow flow(T);
    for (int i = 1; i <= n; i++) {
```

```
        int w;
        cin >> w;
        flow.add(S, i, w);
    }

    int sum = 0;
    for (int i = 1; i <= m; i++) {
        int x, y, w;
        cin >> x >> y >> w;
        flow.add(x, n + i, 1E18);
        flow.add(y, n + i, 1E18);
        flow.add(n + i, T, w);
        sum += w;
    }
    cout << sum - flow.work(S, T) << endl;
}
```

最小割树 Gomory-Hu Tree

无向连通图抽象出的一棵树, 满足任意两点间的距离是他们的最小割. 一共需要跑 n 轮最小割, 总复杂度 $\mathcal{O}(N^3M)$, 预处理最小割树上任意两点的距离 $\mathcal{O}(N^2)$.

过程: 分治 n 轮, 每一轮在图上随机选点, 跑一轮最小割后连接树边; 这一网络的残留网络会将剩余的点分为两组, 根据分组分治.

```
void reset() { // struct 需要额外封装退流
    for (int i = 0; i < ver.size(); i += 2) {
        ver[i].w += ver[i ^ 1].w;
        ver[i ^ 1].w = 0;
    }
}

signed main() { // Gomory-Hu Tree
```

```
int n, m;
cin >> n >> m;

Flow<int> flow(n);
for (int i = 1; i <= m; i++) {
    int u, v, w;
    cin >> u >> v >> w;
    flow.add(u, v, w);
    flow.add(v, u, w);
}

vector<int> vis(n + 1), fa(n + 1);
vector ans(n + 1, vector<int>(n + 1, 1E9)); // N^2 枚举出全部答案
vector<vector<pair<int, int>>> adj(n + 1);
for (int i = 1; i <= n; i++) { // 分治 n 轮
    int s = 0; // 本质是在树上随机选点，跑最小割后连边
    for (; s <= n; s++) {
        if (fa[s] != s) break;
    }
    int t = fa[s];

    int ans = flow.work(s, t); // 残留网络将点集分为两组，分治
    adj[s].push_back({t, ans});
    adj[t].push_back({s, ans});

    vis.assign(n + 1, 0);
    auto dfs = [&](auto dfs, int u) -> void {
        vis[u] = 1;
        for (auto it : flow.h[u]) {
            auto [v, c] = flow.ver[it];
            if (c && !vis[v]) {
                dfs(dfs, v);
            }
```

```
            }
        };
        dfs(dfs, s);
        for (int j = 0; j <= n; j++) {
            if (vis[j] && fa[j] == t) {
                fa[j] = s;
            }
        }
    }

    for (int i = 0; i <= n; i++) {
        auto dfs = [&](auto dfs, int u, int fa, int c) -> void {
            ans[i][u] = c;
            for (auto [v, w] : adj[u]) {
                if (v == fa) continue;
                dfs(dfs, v, u, min(c, w));
            }
        };
        dfs(dfs, i, -1, 1E9);
    }

    int q;
    cin >> q;
    while (q--) {
        int u, v;
        cin >> u >> v;
        cout << ans[u][v] << "\n"; // 预处理答数组
    }
}
```

费用流

给定一个带费用的网络, 规定 (u, v) 间的费用为 $f(u, v) \times w(u, v)$, 求解该网络中总花费最小的最大流称之为**最小费用最大流**. 总时间复杂度为 $\mathcal{O}(NMf)$, 其中 f 代表最大流.

```cpp
struct MinCostFlow {
    using LL = long long;
    using PII = pair<LL, int>;
    const LL INF = numeric_limits<LL>::max();
    struct Edge {
        int v, c, f;
        Edge(int v, int c, int f) : v(v), c(c), f(f) {}
    };
    const int n;
    vector<Edge> e;
    vector<vector<int>> g;
    vector<LL> h, dis;
    vector<int> pre;

    MinCostFlow(int n) : n(n), g(n) {}
    void add(int u, int v, int c, int f) { // c 流量, f 费用
        // if (f < 0) {
        //     g[u].push_back(e.size());
        //     e.emplace_back(v, 0, f);
        //     g[v].push_back(e.size());
        //     e.emplace_back(u, c, -f);
        // } else {
            g[u].push_back(e.size());
            e.emplace_back(v, c, f);
            g[v].push_back(e.size());
            e.emplace_back(u, 0, -f);
        // }
    }
```

```
    bool dijkstra(int s, int t) {
        dis.assign(n, INF);
        pre.assign(n, -1);
        priority_queue<PII, vector<PII>, greater<PII>> que;
        dis[s] = 0;
        que.emplace(0, s);
        while (!que.empty()) {
            auto [d, u] = que.top();
            que.pop();
            if (dis[u] < d) continue;
            for (int i : g[u]) {
                auto [v, c, f] = e[i];
                if (c > 0 && dis[v] > d + h[u] - h[v] + f) {
                    dis[v] = d + h[u] - h[v] + f;
                    pre[v] = i;
                    que.emplace(dis[v], v);
                }
            }
        }
        return dis[t] != INF;
    }
    pair<int, LL> flow(int s, int t) {
        int flow = 0;
        LL cost = 0;
        h.assign(n, 0);
        while (dijkstra(s, t)) {
            for (int i = 0; i < n; ++i) h[i] += dis[i];
            int aug = numeric_limits<int>::max();
            for (int i = t; i != s; i = e[pre[i] ^ 1].v) aug = min(aug, e[pre[i]].c);
            for (int i = t; i != s; i = e[pre[i] ^ 1].v) {
                e[pre[i]].c -= aug;
                e[pre[i] ^ 1].c += aug;
            }
```

```
            flow += aug;
            cost += LL(aug) * h[t];
        }
        return {flow, cost};
    }
};
```

/END/

常用例题

逆序对 (归并排序解)

> 性质: 交换序列的任意两元素, 序列的逆序数的奇偶性必定发生改变.

```
LL a[N], tmp[N], n, ans = 0;
void mergeSort(LL l, LL r){
    if (l >= r) return;
    LL mid = (l + r) >> 1, i = l, j = mid + 1, cnt = 0;
    mergeSort(l, mid);
    mergeSort(mid + 1, r);
    while (i <= mid || j <= r)
        if (j > r || (i <= mid && a[i] <= a[j]))
            tmp[cnt++] = a[i++];
        else
            tmp[cnt++] = a[j++], ans += mid - i + 1;
    for (LL k = 0; k < r - l + 1; k++)
        a[l + k] = tmp[k];
}
int main(){
    cin >> n;
    for (int i = 1; i <= n; i++)
        scanf("%lld", &a[i]);
```

```
    mergeSort(1, n);
    cout << ans << "\n";
    return 0;
}
```

统计区间不同数字的数量 (离线查询)

核心在于使用 pre 数组滚动维护每一个数字出现的最后位置, 配以树状数组统计数量. 由于滚动维护具有后效性, 所以需要离线操作, 从前往后更新. 时间复杂度 $\mathcal{O}(N \log N)$, 常数瓶颈在于 map, 用手造哈希或者离散化可以优化到理想区间; 同时也有莫队做法, 复杂度稍劣.例题链接 .

```
signed main() {
    int n;
    cin >> n;
    vector<int> in(n + 1);
    for (int i = 1; i <= n; i++) {
        cin >> in[i];
    }

    int q;
    cin >> q;
    vector<array<int, 3>> query;
    for (int i = 0; i < q; i++) {
        int l, r;
        cin >> l >> r;
        query.push_back({r, l, i});
    }
    sort(query.begin(), query.end());

    vector<pair<int, int>> ans;
    map<int, int> pre;
    int st = 1;
```

```
    BIT bit(n);
    for (auto [r, l, id] : query) {
        for (int i = st; i <= r; i++, st++) {
            if (pre.count(in[i])) { // 消除此前操作的影响
                bit.add(pre[in[i]], -1);
            }
            bit.add(i, 1);
            pre[in[i]] = i; // 更新操作
        }
        ans.push_back({id, bit.ask(r) - bit.ask(l - 1)});
    }

    sort(ans.begin(), ans.end());
    for (auto [id, w] : ans) {
        cout << w << endl;
    }
}
```

选数 (DFS 解)

从 N 个整数中任选 K 个整数相加. 使用 DFS 求解.

```
int n, k; cin >> n >> k;
vector<int> in(n), now(n);
for (auto &it : in) { cin >> it; }
auto dfs = [&](auto self, int k, int bit, int idx) -> void {
    for (int i = idx; i < n; i++) {
        now[bit] = in[i];
        if (bit < k - 1) { self(self, k, bit + 1, i + 1); }
        if (bit == k - 1) {
            int add = 0;
            for (int j = 0; j < k; j++) {
                add += now[j];
```

```
            }
            cout << add << endl;
        }
    }
};
dfs(dfs, k, 0, 0);
```

选数 (位运算状压)

```
int n, k; cin >> n >> k;
vector<int> in(n);
for (auto &it : in) { cin >> it; }
int comb = (1 << k) - 1, U = 1 << n;
while (comb < U) {
    int add = 0;
    for (int i = 0; i < n; i++) {
        if (1 << i & comb) {
            add += in[i];
        }
    }
    cout << add << "\n";

    int x = comb & -comb;
    int y = comb + x;
    int z = comb & ~y;
    comb = (z / x >> 1) | y;
}
```

网格路径计数

从 $(0,0)$ 走到 (a,b), 规定每次只能从 (x,y) 走到左下或者右下, 方案数记为 $f(a,b)$.

- $f(a,b)=\binom{a}{\frac{a+b}{2}}$;
- 若路径和直线 $y=k, k\notin[0,b]$ 不能有交点, 则方案数为 $f(a,b)-f(a,2k-b)$;
- 若路径和两条直线 $y=k_1, y=k_2(k_1<0\le b<k_2)$ 不能有交点, 方案数记为 $g(a,b,k_1,k_2)$, 可以使用 $\mathcal{O}(N)$ 递归求解;
- 若路径必须碰到 $y=k_1$ 但是不能碰到 $y=k_2$, 方案数记为 $h(a,b,k_1,k_2)$, 可以使用 $\mathcal{O}(N)$ 递归求解 (递归过程中两条直线距离会越来越大).

从 $(0,0)$ 走到 $(a,0)$, 规定每次只能走到左下或者右下, 且必须有**恰好一次**传送 (向下 b 单位), 且不能走到 x 轴下方, 方案数为 $\binom{a+1}{\frac{a-b}{2}+k+1}$.

德州扑克

读入牌型, 并且支持两副牌之间的大小比较.代码参考

```
struct card {
      int suit, rank;
      friend bool operator < (const card &a, const card &b) {
        return a.rank < b.rank;
    }
    friend bool operator == (const card &a, const card &b) {
        return a.rank == b.rank;
    }
    friend bool operator != (const card &a, const card &b) {
        return a.rank != b.rank;
    }
    friend auto &operator>> (istream &it, card &C) {
        string S, T; it >> S;
        T = "__23456789TJQKA"; //点数
        FOR (i, 0, T.sz - 1) {
            if (T[i] == S[0]) C.rank = i;
```

```
        }
        T = "_SHCD"; //花色
        FOR (i, 0, T.sz - 1) {
            if (T[i] == S[1]) C.suit = i;
        }
        return it;
    }
};
struct game {
    int level;
    vector<card> peo;
    int a, b, c, d, e;
    int u, v, w, x, y;
    bool Rk10() { //Rk10: Royal Flush, 五张牌同花色, 且点数为 AKQJT(14,13,12,11,10)
        sort(ALL(peo));
        reverse(ALL(peo));
        a = peo[0].rank, b = peo[1].rank, c = peo[2].rank, d = peo[3].rank, e = peo[4].
        u = peo[0].suit, v = peo[1].suit, w = peo[2].suit, x = peo[3].suit, y = peo[4].

        if (u != v || v != w || w != x || x != y) return 0;
        if (a == 14 && b == 13 && c == 12 && d == 11 && e == 10) return 1;
        return 0;
    }
    bool Dif(vector<card> &peo) { //专门用于检查 A2345 这种顺子的情况 (这是最小的
        a = peo[0].rank, b = peo[1].rank, c = peo[2].rank, d = peo[3].rank, e = peo[4].
        u = peo[0].suit, v = peo[1].suit, w = peo[2].suit, x = peo[3].suit, y = peo[4].

        if (a != 14 || b != 5 || c != 4 || d != 3 || e != 2) return 0;
        vector<card> peo2 = {peo[1], peo[2], peo[3], peo[4], peo[0]}; //重新排序
        peo = peo2;
        return 1;
    }
    bool Rk9() { //Rk9: Straight Flush, 五张牌同花色, 且顺连 r1 > r2 > r3 > r4 > r5
```

```
    sort(ALL(peo));
    reverse(ALL(peo));
    a = peo[0].rank, b = peo[1].rank, c = peo[2].rank, d = peo[3].rank, e = peo[4].
    u = peo[0].suit, v = peo[1].suit, w = peo[2].suit, x = peo[3].suit, y = peo[4].

    if (u != v || v != w || w != x || x != y) return 0;
    if (Dif(peo)) return 1; //特判:A2345
    if (a == b + 1 && b == c + 1 && c == d + 1 && d == e + 1) return 1;
    return 0;
}
bool Rk8() { //Rk8: Four of a Kind, 四张牌点数一样 r1 = r2 = r3 = r4
    sort(ALL(peo));
    a = peo[0].rank, b = peo[1].rank, c = peo[2].rank, d = peo[3].rank, e = peo[4].
    u = peo[0].suit, v = peo[1].suit, w = peo[2].suit, x = peo[3].suit, y = peo[4].

    if (a == b && b == c && c == d) return 1;
    if (b == c && c == d && d == e) {
        reverse(ALL(peo));
        return 1;
    }
    return 0;
}
bool Rk7() { //Rk7: Fullhouse, 三张牌点数一样, 另外两张点数也一样 r1 = r2 = r
    sort(ALL(peo));
    a = peo[0].rank, b = peo[1].rank, c = peo[2].rank, d = peo[3].rank, e = peo[4].
    u = peo[0].suit, v = peo[1].suit, w = peo[2].suit, x = peo[3].suit, y = peo[4].

    if (a == b && b == c && d == e) return 1;
    if (a == b && c == d && d == e) {
        reverse(ALL(peo));
        return 1;
    }
    return 0;
```

```
}
bool Rk6() { //Rk6: Flush, 五张牌同花色 r1 > r2 > r3 > r4 > r5
    sort(ALL(peo));
    reverse(ALL(peo));
    a = peo[0].rank, b = peo[1].rank, c = peo[2].rank, d = peo[3].rank, e = peo[4].
    u = peo[0].suit, v = peo[1].suit, w = peo[2].suit, x = peo[3].suit, y = peo[4].

    if (u != v || v != w || w != x || x != y) return 0;
    return 1;
}
bool Rk5() { //Rk5: Straight, 五张牌顺连 r1 > r2 > r3 > r4 > r5
    sort(ALL(peo));
    reverse(ALL(peo));
    a = peo[0].rank, b = peo[1].rank, c = peo[2].rank, d = peo[3].rank, e = peo[4].
    u = peo[0].suit, v = peo[1].suit, w = peo[2].suit, x = peo[3].suit, y = peo[4].

    if (Dif(peo)) return 1; //特判:A2345
    if (a == b + 1 && b == c + 1 && c == d + 1 && d == e + 1) return 1;
    return 0;
}
bool Rk4() { //Rk4: Three of a kind, 三张牌点数一样 r1 = r2 = r3,r4 > r5
    sort(ALL(peo));
    reverse(ALL(peo));
    a = peo[0].rank, b = peo[1].rank, c = peo[2].rank, d = peo[3].rank, e = peo[4].
    u = peo[0].suit, v = peo[1].suit, w = peo[2].suit, x = peo[3].suit, y = peo[4].

    if (a == b && b == c) return 1;
    if (b == c && c == d) {
        swap(peo[3], peo[0]);
        return 1;
    }
    if (c == d && d == e) {
        swap(peo[3], peo[0]);
```

```
            swap(peo[4], peo[1]);
            return 1;
        }
        return 0;
    }
    bool Rk3() { //Rk3: Two Pairs, 两张牌点数一样, 另外有两张点数也一样 (两个对子
        sort(ALL(peo));
        reverse(ALL(peo));
        a = peo[0].rank, b = peo[1].rank, c = peo[2].rank, d = peo[3].rank, e = peo[4].
        u = peo[0].suit, v = peo[1].suit, w = peo[2].suit, x = peo[3].suit, y = peo[4].

        if (a == b && c == d) return 1;
        if (a == b && d == e) {
            swap(peo[2], peo[4]);
            return 1;
        }
        if (b == c && d == e) {
            swap(peo[0], peo[2]);
            swap(peo[2], peo[4]);
            return 1;
        }
        return 0;
    }
    bool Rk2() { //Rk2: One Pairs, 两张牌点数一样 (一个对子)r1 = r2,r3 > r4 > r5
        sort(ALL(peo));
        reverse(ALL(peo));
        a = peo[0].rank, b = peo[1].rank, c = peo[2].rank, d = peo[3].rank, e = peo[4].
        u = peo[0].suit, v = peo[1].suit, w = peo[2].suit, x = peo[3].suit, y = peo[4].

        vector<card> peo2;
        if (a == b) return 1;
        if (b == c) {
            peo2 = {peo[1], peo[2], peo[0], peo[3], peo[4]};
```

```
            peo = peo2;
            return 1;
        }
        if (c == d) {
            peo2 = {peo[2], peo[3], peo[0], peo[1], peo[4]};
            peo = peo2;
            return 1;
        }
        if (d == e) {
            peo2 = {peo[3], peo[4], peo[0], peo[1], peo[2]};
            peo = peo2;
            return 1;
        }
        return 0;
    }
    bool Rk1() { //Rk1: high card
        sort(ALL(peo));
        reverse(ALL(peo));
        return 1;
    }
    game (vector<card> New_peo) {
        peo = New_peo;
        if (Rk10()) { level = 10; return; }
        if (Rk9()) { level = 9; return; }
        if (Rk8()) { level = 8; return; }
        if (Rk7()) { level = 7; return; }
        if (Rk6()) { level = 6; return; }
        if (Rk5()) { level = 5; return; }
        if (Rk4()) { level = 4; return; }
        if (Rk3()) { level = 3; return; }
        if (Rk2()) { level = 2; return; }
        if (Rk1()) { level = 1; return; }
    }
```

```
    friend bool operator < (const game &a, const game &b) {
        if (a.level != b.level) return a.level < b.level;
        FOR (i, 0, 4) if (a.peo[i] != b.peo[i]) return a.peo[i] < b.peo[i];
        return 0;
    }
    friend bool operator == (const game &a, const game &b) {
        if (a.level != b.level) return 0;
        FOR (i, 0, 4) if (a.peo[i] != b.peo[i]) return 0;
        return 1;
    }
};
void debug(vector<card> peo) {
    for (auto it : peo) cout << it.rank << " " << it.suit << "  ";
    cout << "\n\n";
}
int clac(vector<card> Ali, vector<card> Bob) {
    game atype(Ali), btype(Bob);
    if (atype < btype) return -1;
    else if (atype == btype) return 0;
    return 1;
}
```

N*M 数独字典序最小方案

规则: 每个宫大小为 $2^N * 2^M$, 大图一共由 $M * N$ 个宫组成 (总大小即 $2^N 2^M * 2^N 2^M$), 要求每行, 每列, 每宫都要出现 1 到 $2^N * 2^M$ 的全部数字. 输出字典序最小方案.

下例为 $2, 1$ 和 $1, 2$ 时数独字典序最小的示意.

公式:(i, j) 格所填的内容为 $\left(i \bmod 2^N \oplus \left\lfloor \frac{j}{2^M} \right\rfloor\right) \cdot 2^M + \left(\left\lfloor \frac{i}{2^N} \right\rfloor \oplus j \bmod 2^M\right) + 1$, 注意 i, j 从 0 开始.

高精度进制转换

2 — 62 进制相互转换. 输入格式: “转换前进制转换后进制要转换的数据”.
注释: 进制排序为 0-9,A-Z,a-z.

```
#include<bits/stdc++.h>
using namespace std;
map<char, int> mp; //将字符转化为数字
map<int, char> mp2; //将数字转化为字符
int main(){
    for(int i = 0; i < 10; i++) mp[(char)i + 48] = i, mp2[i] = (char)i + 48;
    for(int i = 10; i < 36; i++) mp[(char)i + 55] = i, mp2[i] = (char)i + 55;
    for(int i = 36; i < 62; i++) mp[(char)i + 61] = i, mp2[i] = (char)i + 61;

    int tt = 1, a, b; cin >> tt;
    while(tt--){
        string s, sh;
        vector<int> nums, ans;
        cin >> a >> b >> s;
        for(auto c : s) nums.push_back(mp[c]);
        reverse(nums.begin(), nums.end());
        while(nums.size()){ //短除法, 将整个大数一直除 b , 取余数
            int remainder = 0;
            for(int i = nums.size() - 1; ~i; i--){
                nums[i] += remainder * a;
                remainder = nums[i] % b;
                nums[i] /= b;
            }
            ans.push_back(remainder); //得到余数
            while(nums.size() && nums.back() == 0) nums.pop_back(); //去掉前导 0
        }
        reverse(ans.begin(), ans.end());
        for(int i : ans) sh += mp2[i];
        cout << a << ' ' << s << endl;
```

```
        cout << b << ' ' << sh << endl << endl;
    }
    return 0;
}
```

物品装箱

有 N 个物品, 第 i 个物品为 $a[i]$, 有无限个容量为 C 的空箱子. 两种装箱方式, 输出需要多少个箱子才能装完所有物品.

从前往后装 (线段树解)

```
template<class T> vector<Point<T>> halfcut(vector<Line<T>> lines) {
    sort(lines.begin(), lines.end(), [&](auto l1, auto l2) {
        auto d1 = l1.b - l1.a;
        auto d2 = l2.b - l2.a;
        if (sign(d1) != sign(d2)) {
            return sign(d1) == 1;
        }
        return cross(d1, d2) > 0;
    });
    deque<Line<T>> ls;
    deque<Point<T>> ps;
    for (auto l : lines) {
        if (ls.empty()) {
            ls.push_back(l);
            continue;
        }
        while (!ps.empty() &&!pointOnLineLeft(ps.back(), l)) {
            ps.pop_back();
            ls.pop_back();
        }
        while (!ps.empty() && !pointOnLineLeft(ps[0], l)) {
```

```
            ps.pop_front();
            ls.pop_front();
        }
        if (cross(l.b - l.a, ls.back().b - ls.back().a) == 0) {
            if (dot(l.b - l.a, ls.back().b - ls.back().a) > 0) {
                if (!pointOnLineLeft(ls.back().a, l)) {
                    assert(ls.size() == 1);
                    ls[0] = l;
                }
                continue;
            }
            return {};
        }
        ps.push_back(lineIntersection(ls.back(), l));
        ls.push_back(l);
    }
    while (!ps.empty() && !pointOnLineLeft(ps.back(), ls[0])) {
        ps.pop_back();
        ls.pop_back();
    }
    if (ls.size() <= 2) {
        return {};
    }
    ps.push_back(lineIntersection(ls[0], ls.back()));
    return vector(ps.begin(), ps.end());
}

const int N = 1e6 + 10;
int T, n, a[N], c, tr[N << 2];
void pushup(int u){
    tr[u] = max(tr[u << 1], tr[u << 1 | 1]);
}
void build(int u, int l, int r){
    if (l == r) tr[u] = c;
```

```
    else {
        int mid = l + r >> 1;
        build(u << 1, l, mid);
        build(u << 1 | 1, mid + 1, r);
        pushup(u);
    }
}
void update(int u, int l, int r, int p, int k){
    if (l > p || r < p) return;
    if (l == r) tr[u] -= k;
    else {
        int mid = l + r >> 1;
        update(u << 1, l, mid, p, k);
        update(u << 1 | 1, mid + 1, r, p, k);
        pushup(u);
    }
}
int query(int u, int l, int r, int k){
    if (l == r){
        if (tr[u] >= k) return l;
        return n + 1;
    }
    int mid = l + r >> 1;
    if (tr[u << 1] >= k) return query(u << 1, l, mid, k);
    else return query(u << 1 | 1, mid + 1, r, k);
}
int main() {
    cin >> n >> c;
    for (int i = 1; i <= n; i++) cin >> a[i];
    build(1, 1, n);
    for (int i = 1; i <= n; i++)
        update(1, 1, n, query(1, 1, n, a[i]), a[i]);
    cout << query(1, 1, n, c) - 1 << " ";
```

```
}
```

选择最优的箱子装 (multiset 解) 选择能放下物品且剩余容量最小的箱子放物品

```
void solve(){
    cin >> n >> c;
    for (int i = 1; i <= n; i++) cin >> a[i];
    multiset <int> s;
    for (int i = 1; i <= n; i++){
        auto it = s.lower_bound(a[i]);
        if (it == s.end()) s.insert(c - a[i]);
        else {
            int x = *it;
            // multiset 可以存放重复数据，如果是删除某个值的话，会去掉多个箱子
            // 导致答案错误，所以直接删除对应位置的元素
            s.erase(it);
            s.insert(x - a[i]);
        }
    }
    cout << s.size() << "\n";
}
```

浮点数比较

比较下列浮点数的大小: $x^{yz}, x^{zy}, (x^y)^z, (x^z)^y, y^{xz}, y^{zx}, (y^x)^z, (y^z)^x, z^{xy}, z^{yx}, (z^x)^y$ 和 $(z^y)^x$.

```
vector<pair<ld, int>> val = {
    {log(x) * pow(y, z), 0}, {log(x) * pow(z, y), 1}, {log(x) * y * z, 2},
    {log(x) * z * y, 3},     {log(y) * pow(x, z), 4}, {log(y) * pow(z, x), 5},
    {log(y) * x * z, 6},     {log(y) * z * x, 7},     {log(z) * pow(x, y), 8},
    {log(z) * pow(y, x), 9}, {log(z) * x * y, 10},    {log(z) * y * x, 11}};
```

```
sort(val.begin(), val.end(), [&](auto x, auto y) {
    if (equal(x.first, y.first)) return x.second < y.second; // queal 比较两个浮点数
    return x.first > y.first;
});
cout << ans[val.front().second] << endl;
```

/END/

杂项

单测多测

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

#define ranges std::ranges
#define views std::views

using u32 = unsigned;
using i64 = long long;
using u64 = unsigned long long;

using pii = std::pair<int, int>;
using a2 = std::array<int, 2>;
using a3 = std::array<int, 3>;
using a4 = std::array<int, 4>;

const int N = 1e6;
const int MAXN = 1e6 + 10;
const int inf = 1e9;
// const int mod = 1e9 + 7;
const int mod = 998244353;
```

```
std::mt19937_64 rng(std::chrono::steady_clock::now().time_since_epoch().count());

void solve() {

}

signed main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(0), std::cout.tie(0);
    int t = 1;//cin >> t;
    while (t--) {
        solve();
        std::cout << '\n';
    }
    return 0;
}
```

三路比较运算符

```
auto operator<=>(const V &) const = default;
```

取模类

集成了常见的取模四则运算, 运算速度与手动取模相差无几, 效率极高.

```
using i64 = long long;

template<typename T> constexpr T mypow(T n, i64 k) {
    T r = 1;
    for (; k; k /= 2, n *= n) {
        if (k % 2) {
            r *= n;
        }
```

```
    }
    return r;
}

template<typename T> constexpr T power(int n) {
    return mypow(T(2), n);
}

template<const int &MOD> struct Zmod {
    int x;
    Zmod(signed x = 0) : x(norm(x % MOD)) {}
    Zmod(i64 x) : x(norm(x % MOD)) {}

    constexpr int norm(int x) const noexcept {
        if (x < 0) [[unlikely]] {
            x += MOD;
        }
        if (x >= MOD) [[unlikely]] {
            x -= MOD;
        }
        return x;
    }
    explicit operator int() const {
        return x;
    }
    constexpr int val() const {
        return x;
    }
    constexpr Zmod operator-() const {
        Zmod val = norm(MOD - x);
        return val;
    }
    constexpr Zmod inv() const {
```

```
        assert(x != 0);
        return mypow(*this, MOD - 2);
    }
    friend constexpr auto &operator>>(istream &in, Zmod &j) {
        int v;
        in >> v;
        j = Zmod(v);
        return in;
    }
    friend constexpr auto &operator<<(ostream &o, const Zmod &j) {
        return o << j.val();
    }
    constexpr Zmod &operator++() {
        x = norm(x + 1);
        return *this;
    }
    constexpr Zmod &operator--() {
        x = norm(x - 1);
        return *this;
    }
    constexpr Zmod operator++(signed) {
        Zmod res = *this;
        ++*this;
        return res;
    }
    constexpr Zmod operator--(signed) {
        Zmod res = *this;
        --*this;
        return res;
    }
    constexpr Zmod &operator+=(const Zmod &i) {
        x = norm(x + i.x);
        return *this;
```

```
    }
    constexpr Zmod &operator-=(const Zmod &i) {
        x = norm(x - i.x);
        return *this;
    }
    constexpr Zmod &operator*=(const Zmod &i) {
        x = i64(x) * i.x % MOD;
        return *this;
    }
    constexpr Zmod &operator/=(const Zmod &i) {
        return *this *= i.inv();
    }
    constexpr Zmod &operator%=(const int &i) {
        return x %= i, *this;
    }
    friend constexpr Zmod operator+(const Zmod i, const Zmod j) {
        return Zmod(i) += j;
    }
    friend constexpr Zmod operator-(const Zmod i, const Zmod j) {
        return Zmod(i) -= j;
    }
    friend constexpr Zmod operator*(const Zmod i, const Zmod j) {
        return Zmod(i) *= j;
    }
    friend constexpr Zmod operator/(const Zmod i, const Zmod j) {
        return Zmod(i) /= j;
    }
    friend constexpr Zmod operator%(const Zmod i, const int j) {
        return Zmod(i) %= j;
    }
    friend constexpr bool operator==(const Zmod i, const Zmod j) {
        return i.val() == j.val();
    }
```

```
    friend constexpr bool operator!=(const Zmod i, const Zmod j) {
        return i.val() != j.val();
    }
    friend constexpr bool operator<(const Zmod i, const Zmod j) {
        return i.val() < j.val();
    }
    friend constexpr bool operator>(const Zmod i, const Zmod j) {
        return i.val() > j.val();
    }
};

int MOD[] = {998244353, 1000000007};
using Z = Zmod<MOD[1]>;
```

分数运算类

定义了分数的四则运算, 如果需要处理浮点数, 那么需要将函数中的 gcd 运算替换为 fgcd .

```
template<typename T> struct Frac {
    T x, y;
    Frac() : Frac(0, 1) {}
    Frac(T x_) : Frac(x_, 1) {}
    Frac(T x_, T y_) : x(x_), y(y_) {
        if (y < 0) {
            y = -y;
            x = -x;
        }
    }

    constexpr double val() const {
        return 1. * x / y;
    }
```

```
    constexpr Frac norm() const { // 调整符号，转化为最简形式
        T p = gcd(x, y);
        return {x / p, y / p};
    }
    friend constexpr auto &operator<<(ostream &o, const Frac &j) {
        T p = gcd(j.x, j.y);
        if (j.y == p) {
            return o << j.x / p;
        } else {
            return o << j.x / p << "/" << j.y / p;
        }
    }
    constexpr Frac &operator/=(const Frac &i) {
        x *= i.y;
        y *= i.x;
        if (y < 0) {
            x = -x;
            y = -y;
        }
        return *this;
    }
    constexpr Frac &operator+=(const Frac &i) { return x = x * i.y + y * i.x, y *= i.y,
    constexpr Frac &operator-=(const Frac &i) { return x = x * i.y - y * i.x, y *= i.y,
    constexpr Frac &operator*=(const Frac &i) { return x *= i.x, y *= i.y, *this; }
    friend constexpr Frac operator+(const Frac i, const Frac j) { return i += j; }
    friend constexpr Frac operator-(const Frac i, const Frac j) { return i -= j; }
    friend constexpr Frac operator*(const Frac i, const Frac j) { return i *= j; }
    friend constexpr Frac operator/(const Frac i, const Frac j) { return i /= j; }
    friend constexpr Frac operator-(const Frac i) { return Frac(-i.x, i.y); }
    friend constexpr bool operator<(const Frac i, const Frac j) { return i.x * j.y < i.
    friend constexpr bool operator>(const Frac i, const Frac j) { return i.x * j.y > i.
    friend constexpr bool operator==(const Frac i, const Frac j) { return i.x * j.y ==
    friend constexpr bool operator!=(const Frac i, const Frac j) { return i.x * j.y !=
```

```
};
```

大整数类 (高精度计算)

```
const int base = 1000000000;
const int base_digits = 9; // 分解为九个数位一个数字
struct bigint {
    vector<int> a;
    int sign;

    bigint() : sign(1) {}
    bigint operator-() const {
        bigint res = *this;
        res.sign = -sign;
        return res;
    }
    bigint(long long v) {
        *this = v;
    }
    bigint(const string &s) {
        read(s);
    }
    void operator=(const bigint &v) {
        sign = v.sign;
        a = v.a;
    }
    void operator=(long long v) {
        a.clear();
        sign = 1;
        if (v < 0) sign = -1, v = -v;
        for (; v > 0; v = v / base) {
            a.push_back(v % base);
        }
```

```
    }

    // 基础加减乘除
    bigint operator+(const bigint &v) const {
        if (sign == v.sign) {
            bigint res = v;
            for (int i = 0, carry = 0; i < (int)max(a.size(), v.a.size()) || carry; ++i
                if (i == (int)res.a.size()) {
                    res.a.push_back(0);
                }
                res.a[i] += carry + (i < (int)a.size() ? a[i] : 0);
                carry = res.a[i] >= base;
                if (carry) {
                    res.a[i] -= base;
                }
            }
            return res;
        }
        return *this - (-v);
    }
    bigint operator-(const bigint &v) const {
        if (sign == v.sign) {
            if (abs() >= v.abs()) {
                bigint res = *this;
                for (int i = 0, carry = 0; i < (int)v.a.size() || carry; ++i) {
                    res.a[i] -= carry + (i < (int)v.a.size() ? v.a[i] : 0);
                    carry = res.a[i] < 0;
                    if (carry) {
                        res.a[i] += base;
                    }
                }
                res.trim();
                return res;
```

```
            }
            return -(v - *this);
        }
        return *this + (-v);
    }
    void operator*=(int v) {
        check(v);
        for (int i = 0, carry = 0; i < (int)a.size() || carry; ++i) {
            if (i == (int)a.size()) {
                a.push_back(0);
            }
            long long cur = a[i] * (long long)v + carry;
            carry = (int)(cur / base);
            a[i] = (int)(cur % base);
        }
        trim();
    }
    void operator/=(int v) {
        check(v);
        for (int i = (int)a.size() - 1, rem = 0; i >= 0; --i) {
            long long cur = a[i] + rem * (long long)base;
            a[i] = (int)(cur / v);
            rem = (int)(cur % v);
        }
        trim();
    }
    int operator%(int v) const {
        if (v < 0) {
            v = -v;
        }
        int m = 0;
        for (int i = a.size() - 1; i >= 0; --i) {
            m = (a[i] + m * (long long)base) % v;
```

```
        }
        return m * sign;
    }

    void operator+=(const bigint &v) {
        *this = *this + v;
    }
    void operator-=(const bigint &v) {
        *this = *this - v;
    }
    bigint operator*(int v) const {
        bigint res = *this;
        res *= v;
        return res;
    }
    bigint operator/(int v) const {
        bigint res = *this;
        res /= v;
        return res;
    }
    void operator%=(const int &v) {
        *this = *this % v;
    }

    bool operator<(const bigint &v) const {
        if (sign != v.sign) return sign < v.sign;
        if (a.size() != v.a.size()) return a.size() * sign < v.a.size() * v.sign;
        for (int i = a.size() - 1; i >= 0; i--)
            if (a[i] != v.a[i]) return a[i] * sign < v.a[i] * sign;
        return false;
    }
    bool operator>(const bigint &v) const {
        return v < *this;
```

```
    }
    bool operator<=(const bigint &v) const {
        return !(v < *this);
    }
    bool operator>=(const bigint &v) const {
        return !(*this < v);
    }
    bool operator==(const bigint &v) const {
        return !(*this < v) && !(v < *this);
    }
    bool operator!=(const bigint &v) const {
        return *this < v || v < *this;
    }

    bigint abs() const {
        bigint res = *this;
        res.sign *= res.sign;
        return res;
    }
    void check(int v) { // 检查输入的是否为负数
        if (v < 0) {
            sign = -sign;
            v = -v;
        }
    }
    void trim() { // 去除前导零
        while (!a.empty() && !a.back()) a.pop_back();
        if (a.empty()) sign = 1;
    }
    bool isZero() const { // 判断是否等于零
        return a.empty() || (a.size() == 1 && !a[0]);
    }
    friend bigint gcd(const bigint &a, const bigint &b) {
```

```
        return b.isZero() ? a : gcd(b, a % b);
    }
    friend bigint lcm(const bigint &a, const bigint &b) {
        return a / gcd(a, b) * b;
    }
    void read(const string &s) {
        sign = 1;
        a.clear();
        int pos = 0;
        while (pos < (int)s.size() && (s[pos] == '-' || s[pos] == '+')) {
            if (s[pos] == '-') sign = -sign;
            ++pos;
        }
        for (int i = s.size() - 1; i >= pos; i -= base_digits) {
            int x = 0;
            for (int j = max(pos, i - base_digits + 1); j <= i; j++) x = x * 10 + s[j]
            a.push_back(x);
        }
        trim();
    }
    friend istream &operator>>(istream &stream, bigint &v) {
        string s;
        stream >> s;
        v.read(s);
        return stream;
    }
    friend ostream &operator<<(ostream &stream, const bigint &v) {
        if (v.sign == -1) stream << '-';
        stream << (v.a.empty() ? 0 : v.a.back());
        for (int i = (int)v.a.size() - 2; i >= 0; --i)
            stream << setw(base_digits) << setfill('0') << v.a[i];
        return stream;
    }
```

```
/* 大整数乘除大整数部分 */
typedef vector<long long> vll;
bigint operator*(const bigint &v) const { // 大整数乘大整数
    vector<int> a6 = convert_base(this->a, base_digits, 6);
    vector<int> b6 = convert_base(v.a, base_digits, 6);
    vll a(a6.begin(), a6.end());
    vll b(b6.begin(), b6.end());
    while (a.size() < b.size()) a.push_back(0);
    while (b.size() < a.size()) b.push_back(0);
    while (a.size() & (a.size() - 1)) a.push_back(0), b.push_back(0);
    vll c = karatsubaMultiply(a, b);
    bigint res;
    res.sign = sign * v.sign;
    for (int i = 0, carry = 0; i < (int)c.size(); i++) {
        long long cur = c[i] + carry;
        res.a.push_back((int)(cur % 1000000));
        carry = (int)(cur / 1000000);
    }
    res.a = convert_base(res.a, 6, base_digits);
    res.trim();
    return res;
}
friend pair<bigint, bigint> divmod(const bigint &a1,
                                   const bigint &b1) { // 大整数除大整数，同时返
    int norm = base / (b1.a.back() + 1);
    bigint a = a1.abs() * norm;
    bigint b = b1.abs() * norm;
    bigint q, r;
    q.a.resize(a.a.size());
    for (int i = a.a.size() - 1; i >= 0; i--) {
        r *= base;
        r += a.a[i];
```

```
            int s1 = r.a.size() <= b.a.size() ? 0 : r.a[b.a.size()];
            int s2 = r.a.size() <= b.a.size() - 1 ? 0 : r.a[b.a.size() - 1];
            int d = ((long long)base * s1 + s2) / b.a.back();
            r -= b * d;
            while (r < 0) r += b, --d;
            q.a[i] = d;
        }
        q.sign = a1.sign * b1.sign;
        r.sign = a1.sign;
        q.trim();
        r.trim();
        return make_pair(q, r / norm);
    }
    static vector<int> convert_base(const vector<int> &a, int old_digits, int new_digit
        vector<long long> p(max(old_digits, new_digits) + 1);
        p[0] = 1;
        for (int i = 1; i < (int)p.size(); i++) p[i] = p[i - 1] * 10;
        vector<int> res;
        long long cur = 0;
        int cur_digits = 0;
        for (int i = 0; i < (int)a.size(); i++) {
            cur += a[i] * p[cur_digits];
            cur_digits += old_digits;
            while (cur_digits >= new_digits) {
                res.push_back((int)(cur % p[new_digits]));
                cur /= p[new_digits];
                cur_digits -= new_digits;
            }
        }
        res.push_back((int)cur);
        while (!res.empty() && !res.back()) res.pop_back();
        return res;
    }
```

```
static vll karatsubaMultiply(const vll &a, const vll &b) {
    int n = a.size();
    vll res(n + n);
    if (n <= 32) {
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                res[i + j] += a[i] * b[j];
            }
        }
        return res;
    }

    int k = n >> 1;
    vll a1(a.begin(), a.begin() + k);
    vll a2(a.begin() + k, a.end());
    vll b1(b.begin(), b.begin() + k);
    vll b2(b.begin() + k, b.end());

    vll a1b1 = karatsubaMultiply(a1, b1);
    vll a2b2 = karatsubaMultiply(a2, b2);

    for (int i = 0; i < k; i++) a2[i] += a1[i];
    for (int i = 0; i < k; i++) b2[i] += b1[i];

    vll r = karatsubaMultiply(a2, b2);
    for (int i = 0; i < (int)a1b1.size(); i++) r[i] -= a1b1[i];
    for (int i = 0; i < (int)a2b2.size(); i++) r[i] -= a2b2[i];

    for (int i = 0; i < (int)r.size(); i++) res[i + k] += r[i];
    for (int i = 0; i < (int)a1b1.size(); i++) res[i] += a1b1[i];
    for (int i = 0; i < (int)a2b2.size(); i++) res[i + n] += a2b2[i];
    return res;
}
```

```
    void operator*=(const bigint &v) {
        *this = *this * v;
    }
    bigint operator/(const bigint &v) const {
        return divmod(*this, v).first;
    }
    void operator/=(const bigint &v) {
        *this = *this / v;
    }
    bigint operator%(const bigint &v) const {
        return divmod(*this, v).second;
    }
    void operator%=(const bigint &v) {
        *this = *this % v;
    }
};
```

阿达马矩阵 (Hadamard matrix)

构造题用, 其有一些性质: 将 0 看作 -1;1 看作 $+1$, 整个矩阵可以构成一个 2^k 维向量组, 任意两个行, 列向量的点积均为 0 See. 例如, 在 $k=2$ 时行向量 $\vec{2}$ 和行向量 $\vec{3}$ 的点积为 $1\cdot 1+(-1)\cdot 1+1\cdot(-1)+(-1)\cdot(-1)=0$.

xxxxxxxxx2 1p=(a+b+c)/2;2sum=sqrt(p(p-a)(p-b)*(p-c));cpp

```
int n;
cin >> n;
int N = pow(2, n);
vector ans(N, vector<int>(N));
ans[0][0] = 1;
for (int t = 0; t < n; t++) {
    int m = pow(2, t);
```

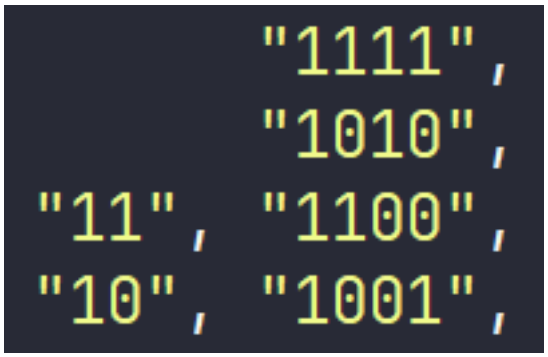


图 2: image.png

```
    for (int i = 0; i < m; i++) {
        for (int j = m; j < 2 * m; j++) {
            ans[i][j] = ans[i][j - m];
        }
    }
    for (int i = m; i < 2 * m; i++) {
        for (int j = 0; j < m; j++) {
            ans[i][j] = ans[i - m][j];
        }
    }
    for (int i = m; i < 2 * m; i++) {
        for (int j = m; j < 2 * m; j++) {
            ans[i][j] = 1 - ans[i - m][j - m];
        }
    }
}
```

幻方

构造题用, 其有一些性质 (保证 N 为奇数):1 到 N^2 每个数字恰好使用一次, 且每行, 每列及两条对角线上的数字之和都相同, 且为奇数 See .

构造方式: 将 1 写在第一行的中间, 随后不断向右上角位置填下一个数字, 直到填满.

```
int n;
cin >> n;
int x = 1, y = (n + 1) / 2;
vector ans(n + 1, vector<int>(n + 1));
for (int i = 1; i <= n * n; i++) {
    ans[x][y] = i;
    if (!ans[(x - 2 + n) % n + 1][y % n + 1]){
        x = (x - 2 + n) % n + 1;
        y = y % n + 1;
    } else {
        x = x % n + 1;
    }
}
```

最长严格/非严格递增子序列 (LIS)

一维 注意子序列是不连续的. 使用二分搜索, 以 $\mathcal{O}(N \log N)$ 复杂度通过, 另也有 $\mathcal{O}(N^2)$ 的 dp 解法.

> Dilworth: 对于任意有限偏序集, 其最大反链中元素的数目必等于最小链划分中链的数目. 将一个序列剖成若干个单调不升子序列的最小个数等于该序列最长上升子序列的个数

```
vector<int> val; // 堆数
for (int i = 1, x; i <= n; i++) {
    cin >> x;
    int it = upper_bound(val.begin(), val.end(), x) - val.begin(); // low/upp: 严格/非
    if (it >= val.size()) { // 新增一堆
        val.push_back(x);
    } else { // 更新对应位置元素
        val[it] = x;
```

```
    }
}
cout << val.size() << endl;
```

二维 + 输出方案

```
vector<array<int, 3>> in(n + 1);
for (int i = 1; i <= n; i++) {
    cin >> in[i][0] >> in[i][1];
    in[i][2] = i;
}
sort(in.begin() + 1, in.end(), [&](auto x, auto y) {
    if (x[0] != y[0]) return x[0] < y[0];
    return x[1] > y[1];
});

vector<int> val{0}, idx{0}, pre(n + 1);
for (int i = 1; i <= n; i++) {
    auto [x, y, z] = in[i];
    int it = lower_bound(val.begin(), val.end(), y) - val.begin(); // low/upp: 严格/非
    if (it >= val.size()) { // 新增一堆
        pre[z] = idx.back();
        val.push_back(y);
        idx.push_back(z);
    } else { // 更新对应位置元素
        pre[z] = idx[it - 1];
        val[it] = y;
        idx[it] = z;
    }
}

vector<int> ans;
for (int i = idx.back(); i != 0; i = pre[i]) {
```

```
    ans.push_back(i);
}
reverse(ans.begin(), ans.end());
cout << ans.size() << "\n";
for (auto it : ans) {
    cout << it << " ";
}
```

cout 输出流控制

设置字段宽度:`setw(x)`，该函数可以使得补全 x 位输出，默认用空格补全.

```
bool Solve() {
    cout << 12 << endl;
    cout << setw(12) << 12 << endl;
    return 0;
}
```

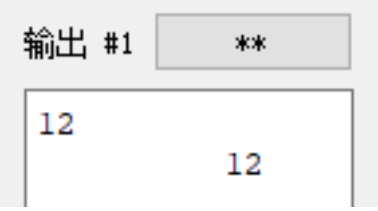


图 3: 67dce9cb83b4b4ede4f7eb453a7033e0.png

设置填充字符:`setfill(x)`，该函数可以设定补全类型，注意这里的 x 只能为 `char` 类型.

```
bool Solve() {
    cout << 12 << endl;
    cout << setw(12) << setfill('*') << 12 << endl;
    return 0;
}
```

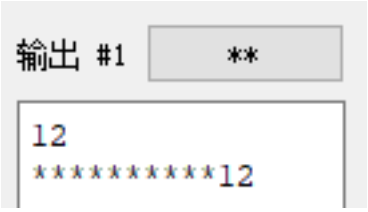


图 4: 761488b7b2fd4871c5cfba7b112fcc6e.png

读取一行数字, 个数未知

```
string s;
getline(cin, s);
stringstream ss;
ss << s;
while (ss >> s) {
    auto res = stoi(s);
    cout << res * 100 << endl;
}
```

约瑟夫问题

n 个人编号 $0, 1, 2, n-1$, 每次数到 k 出局, 求最后剩下的人的编号.

$\mathcal{O}(N)$.

```
int jos(int n,int k){
    int res=0;
    repeat(i,1,n+1)res=(res+k)%i;
    return res; // res+1, 如果编号从 1 开始
}
```

$\mathcal{O}(K \log N)$, 适用于 K 较小的情况.

```
int jos(int n,int k){
    if(n==1 || k==1)return n-1;
```

```
    if(k>n)return (jos(n-1,k)+k)%n; // 线性算法
    int res=jos(n-n/k,k)-n%k;
    if(res<0)res+=n; // mod n
    else res+=res/(k-1); // 还原位置
    return res; // res+1, 如果编号从 1 开始
}
```

$\mathcal{O}(\sqrt{N})$

```
void jos(){
    int64_t n, k, a{}, b{ 1 }; cin >> n >> k; --k;
    while (b < n) {
        auto s = a / k + 1, u = b / k + 1, v = min(k - a / s, (min(u * k, n) - b + u -
        a += s * v, b += u * v;
    }
    cout << a + 1 << '\n';
}
```

日期换算 (基姆拉尔森公式)

已知年月日, 求星期数.

```
int week(int y,int m,int d){
    if(m<=2)m+=12,y--;
    return (d+2*m+3*(m+1)/5+y+y/4-y/100+y/400)%7+1;
}
```

高精度快速幂

求解 $n^k \bmod p$, 其中 $0 \le n, k \le 10^{1000000}$, $1 \le p \le 10^9$. 容易发现 n 可以直接取模, 瓶颈在于 k See.

魔改十进制快速幂 (暴力计算) 该算法复杂度 $\mathcal{O}(\texttt{len}(k))$.

```cpp
int mypow10(int n, vector<int> k, int p) {
    int r = 1;
    for (int i = k.size() - 1; i >= 0; i--) {
        for (int j = 1; j <= k[i]; j++) {
            r = r * n % p;
        }
        int v = 1;
        for (int j = 0; j <= 9; j++) {
            v = v * n % p;
        }
        n = v;
    }
    return r;
}
signed main() {
    string n_, k_;
    int p;
    cin >> n_ >> k_ >> p;

    int n = 0; // 转化并计算 n % p
    for (auto it : n_) {
        n = n * 10 + it - '0';
        n %= p;
    }
    vector<int> k; // 转化 k
    for (auto it : k_) {
        k.push_back(it - '0');
    }
    cout << mypow10(n, k, p) << endl; // 暴力快速幂
}
```

扩展欧拉定理 (欧拉降幂公式)

$$n^k \equiv \begin{cases} n^{k \bmod \varphi(p)} & \gcd(n,p) = 1 \\ n^{k \bmod \varphi(p)+\varphi(p)} & \gcd(n,p) \neq 1 \wedge k \geq \varphi(p) \\ n^k & \gcd(n,p) \neq 1 \wedge k < \varphi(p) \end{cases}$$

最终我们可以将幂降到 $\varphi(p)$ 的级别, 使得能够直接使用快速幂解题, 复杂度瓶颈在求解欧拉函数 $\mathcal{O}(\sqrt{p})$.

```
int phi(int n) { //求解 phi(n)
    int ans = n;
    for (int i = 2; i <= n / i; i++) {
        if (n % i == 0) {
            ans = ans / i * (i - 1);
            while (n % i == 0) {
                n /= i;
            }
        }
    }
    if (n > 1) { //特判 n 为质数的情况
        ans = ans / n * (n - 1);
    }
    return ans;
}
signed main() {
    string n_, k_;
    int p;
    cin >> n_ >> k_ >> p;

    int n = 0; // 转化并计算 n % p
    for (auto it : n_) {
        n = n * 10 + it - '0';
        n %= p;
    }
    int mul = phi(p), type = 0, k = 0; // 转化 k
```

```
    for (auto it : k_) {
        k = k * 10 + it - '0';
        type |= (k >= mul);
        k %= mul;
    }
    if (type) {
        k += mul;
    }
    cout << mypow(n, k, p) << endl;
}
```

int128 输入输出流控制

int128 只在基于 $linux$ 系统的环境下可用, 需要 $C++20$ .38 位精度, 除输入输出外与普通数据类型无差别. 该封装支持负数读入, 需要注意 `write` 函数结尾不输出多余空格与换行.

```
using i128 = __int128;

std::istream& operator>>(std::istream& is, i128& n) {
    std::string s;is >> s;
    n = 0;
    for (char i : s) n = n * 10 + i - '0';
    return is;
}
std::ostream& operator<<(std::ostream& os, i128 n) {
    if (n == 0) {
        return os << 0;
    }
    std::string s;
    while (n) {
        s += '0' + n % 10;
        n /= 10;
```

```
    }
    std::reverse(s.begin(), s.end());
    return os << s;
}
```

对拍板子

- 文件控制

```
// BAD.cpp, 存放待寻找错误的代码
freopen("A.txt","r",stdin);
freopen("BAD.out","w",stdout);

// 1.cpp, 存放暴力或正确的代码
freopen("A.txt","r",stdin);
freopen("1.out","w",stdout);

// Ask.cpp
freopen("A.txt", "w", stdout);
```

- C++ 版 bat

```
int main() {
    int T = 1E5;
    while(T--) {
        system("BAD.exe");
        system("1.exe");
        system("A.exe");
        if (system("fc BAD.out 1.out")) {
            puts("WA");
            return 0;
        }
    }
}
```

在 $linux$ 中将 $diff$ 换成 fc

```cpp
#include <stdio.h>
#include <stdlib.h>

int main() {
    // For Windows
    // 对拍时不开文件输入输出
    // 当然，这段程序也可以改写成批处理的形式
    while (true) {
        system("data > test.in");  // 数据生成器将生成数据写入输入文件
        system("solve < test.in > solve.out");  // 获取程序 1 输出
        system("std < test.in > std.out");  // 获取程序 2 输出
        if (system("diff solve.out std.out")) {
            // 该行语句比对输入输出
            // fc 返回 0 时表示输出一致，否则表示有不同处
            system("pause");  // 方便查看不同处
            return 0;
            // 该输入数据已经存放在 test.in 文件中，可以直接利用进行调试
        }
    }
}
```

```python
import os
tc = 0
os.system("g++ ./std.cpp -o ./std")
os.system("g++ ./solve.cpp -o ./solve")

while True:
    os.system("python ./data.py > ./data.in")
    os.system("./std < ./data.in > ./std.out")
    os.system("./solve < ./data.in > ./solve.out");
    if(os.system("diff ./std.out ./solve.out")):
        print("WA")
```

```
        exit(0)
    else:
        tc += 1
        print("AC #%d" %(tc))
```

随机数生成与样例构造

```
mt19937 rnd(chrono::steady_clock::now().time_since_epoch().count());
int r(int a, int b) {
    return rnd() % (b - a + 1) + a;
}

void graph(int n, int root = -1, int m = -1) {
    vector<pair<int, int>> t;
    for (int i = 1; i < n; i++) { // 先建立一棵以 0 为根节点的树
        t.emplace_back(i, r(0, i - 1));
    }

    vector<pair<int, int>> edge;
    set<pair<int, int>> uni;
    if (root == -1) root = r(0, n - 1); // 确定根节点
    for (auto [x, y] : t) { // 偏移建树
        x = (x + root) % n + 1;
        y = (y + root) % n + 1;
        edge.emplace_back(x, y);
        uni.emplace(x, y);
    }

    if (m != -1) { // 如果是图，则在树的基础上继续加边
        for (int i = n; i <= m; i++) {
            while (true) {
                int x = r(1, n), y = r(1, n);
```

```
                if (x == y) continue; // 拒绝自环
                if (uni.count({x, y})) continue; // 拒绝重边
                edge.emplace_back(x, y);
                uni.emplace(x, y);
            }
        }
    }

    random_shuffle(edge.begin(), edge.end()); // 打乱节点
    for (auto [x, y] : edge) {
        cout << x << " " << y << endl;
    }
}
```

手工哈希

```
struct myhash {
    static uint64_t hash(uint64_t x) {
        x += 0x9e3779b97f4a7c15;
        x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9;
        x = (x ^ (x >> 27)) * 0x94d049bb133111eb;
        return x ^ (x >> 31);
    }
    size_t operator()(uint64_t x) const {
        static const uint64_t SEED = chrono::steady_clock::now().time_since_epoch().cou
        return hash(x + SEED);
    }
    size_t operator()(pair<uint64_t, uint64_t> x) const {
        static const uint64_t SEED = chrono::steady_clock::now().time_since_epoch().cou
        return hash(x.first + SEED) ^ (hash(x.second + SEED) >> 1);
    }
};
// unordered_map<int, int, myhash>
```

Python 常用语法

读入与定义

- 读入多个变量并转换类型:`X, Y = map(int, input().split())`
- 读入列表:`X = eval(input())`
- 多维数组定义:`X = [[0 for j in range(0, 100)] for i in range(0, 200)]`

格式化输出

- 保留小数输出:`print("{:.12f}".format(X))` 指保留 12 位小数
- 对齐与宽度:`print("{:<12f}".format(X))` 指左对齐, 保留 12 个宽度

排序

- 倒序排序: 使用 `reverse` 实现倒序 `X.sort(reverse=True)`
- 自定义排序: 下方代码实现了先按第一关键字降序, 再按第二关键字升序排序.

```
X.sort(key=lambda x: x[1])
X.sort(key=lambda x: x[0], reverse=True)
```

文件 IO

- 打开要读取的文件:`r = open('X.txt', 'r', encoding='utf-8')`
- 打开要写入的文件:`w = open('Y.txt', 'w', encoding='utf-8')`
- 按行写入:`w.write(XX)`

增加输出流长度, 递归深度

```
import sys
sys.set_int_max_str_digits(200000)
sys.setrecursionlimit(100000)
```

自定义结构体 自定义结构体并且自定义排序

```python
class node:
    def __init__(self, A, B, C):
        self.A = A
        self.B = B
        self.C = C

w = []
for i in range(1, 5):
    a, b, c = input().split()
    w.append(node(a, b, c))
w.sort(key=lambda x: x.C, reverse=True)
for i in w:
    print(i.A, i.B, i.C)
```

数据结构

- 模拟于 C_{++}^{map} , 定义:`dic = dict()`
- 模拟栈与队列: 使用常见的 `list` 即可完成,`list.insert(0, X)` 实现头部插入,`list.pop()` 实现尾部弹出,`list.pop(0)` 实现头部弹出

```cpp
// #pragma GCC optimize("Ofast", "unroll-loops")
#include <bits/stdc++.h>
using namespace std;

signed main() {
    int n = 4E3, cnt = 0;
    bitset<30> ans;
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j += 2) {
            for (int k = 1; k <= n; k += 4) {
                ans |= i | j | k;
                cnt++;
```

```
            }
        }
    }
    cout << cnt << "\n";
}
```

其他　获取 ASCII 码 ord(), 转换为 ASCII 字符 chr()

OJ 测试

对于一个未知属性的 OJ, 应当在正式赛前进行以下全部测试:

GNU C++ 版本测试

```
for (int i : {1, 2}) {} // GNU C++11 支持范围表达式

auto cc = [&](int x) { x++; }; // GNU C++11 支持 auto 与 lambda 表达式
cc(2);

tuple<string, int, int> V; // GNU C++11 引入
array<int, 3> C; // GNU C++11 引入

auto dfs = [&](auto self, int x) -> void { // GNU C++14 支持 auto 自递归
    if (x > 10) return;
    self(self, x + 1);
};
dfs(dfs, 1);

vector in(1, vector<int>(1)); // GNU C++17 支持 vector 模板类型缺失

map<int, int> dic;
for (auto [u, v] : dic) {} // GNU C++17 支持 auto 解绑
```

```
dic.contains(12); // GNU C++20 支持 contains 函数

constexpr double Pi = numbers::pi; // C++20 支持
```

评测器环境测试　Windows 系统输出 −1；反之则为一个随机数.

```
#define int long long
map<int, int> dic;
int x = dic.size() - 1;
cout << x << endl;
```

编译器位数测试

```
using i64 = __int128; // 64 位 GNU C++11 支持
```

运算速度测试

```
// #pragma GCC optimize("Ofast", "unroll-loops")

#include <bits/stdc++.h>
using namespace std;
mt19937 rnd(chrono::steady_clock::now().time_since_epoch().count());

signed main() {
    size_t n = 340000000, seed = 0;
    for (int i = 1; i <= n; i++) {
        seed ^= rnd();
    }

    return 0;
}
```

编译器设置

```
g++ -O2 -std=c++20 -pipe
-Wall -Wextra -Wconversion /* 这部分是警告相关，可能用不到 */
-fstack-protector
-Wl,--stack=268435456
```

树上问题

树的直径

```
struct Tree {
    int n;
    vector<vector<int>> ver;
    Tree(int n) {
        this->n = n;
        ver.resize(n + 1);
    }
    void add(int x, int y) {
        ver[x].push_back(y);
        ver[y].push_back(x);
    }
    int getlen(int root) { // 获取 x 所在树的直径
        map<int, int> dep; // map 用于优化输入为森林时的深度计算，亦可用 vector
        function<void(int, int)> dfs = [&](int x, int fa) -> void {
            for (auto y : ver[x]) {
                if (y == fa) continue;
                dep[y] = dep[x] + 1;
```

```
                dfs(y, x);
            }
            if (dep[x] > dep[root]) {
                root = x;
            }
        };
        dfs(root, 0);
        int st = root; // 记录直径端点

        dep.clear();
        dfs(root, 0);
        int ed = root; // 记录直径另一端点

        return dep[root];
    }
};
```

树论大封装 (直径 + 重心 + 中心)

```
struct Tree {
    int n;
    vector<vector<pair<int, int>>> e;
    vector<int> dep, parent, maxdep, d1, d2, s1, s2, up;
    Tree(int n) {
        this->n = n;
        e.resize(n + 1);
        dep.resize(n + 1);
        parent.resize(n + 1);
        maxdep.resize(n + 1);
        d1.resize(n + 1);
        d2.resize(n + 1);
        s1.resize(n + 1);
        s2.resize(n + 1);
```

```
        up.resize(n + 1);
    }
    void add(int u, int v, int w) {
        e[u].push_back({w, v});
        e[v].push_back({w, u});
    }
    void dfs(int u, int fa) {
        maxdep[u] = dep[u];
        for (auto [w, v] : e[u]) {
            if (v == fa) continue;
            dep[v] = dep[u] + 1;
            parent[v] = u;
            dfs(v, u);
            maxdep[u] = max(maxdep[u], maxdep[v]);
        }
    }

    void dfs1(int u, int fa) {
        for (auto [w, v] : e[u]) {
            if (v == fa) continue;
            dfs1(v, u);
            int x = d1[v] + w;
            if (x > d1[u]) {
                d2[u] = d1[u], s2[u] = s1[u];
                d1[u] = x, s1[u] = v;
            } else if (x > d2[u]) {
                d2[u] = x, s2[u] = v;
            }
        }
    }
    void dfs2(int u, int fa) {
        for (auto [w, v] : e[u]) {
            if (v == fa) continue;
```

```
            if (s1[u] == v) {
                up[v] = max(up[u], d2[u]) + w;
            } else {
                up[v] = max(up[u], d1[u]) + w;
            }
            dfs2(v, u);
        }
    }

    int radius, center, diam;
    void getCenter() {
        center = 1; //中心
        for (int i = 1; i <= n; i++) {
            if (max(d1[i], up[i]) < max(d1[center], up[center])) {
                center = i;
            }
        }
        radius = max(d1[center], up[center]); //距离最远点的距离的最小值
        diam = d1[center] + up[center] + 1; //直径
    }

    int rem; //删除重心后剩余连通块体积的最小值
    int cog; //重心
    vector<bool> vis;
    void getCog() {
        vis.resize(n);
        rem = INT_MAX;
        cog = 1;
        dfsCog(1);
    }
    int dfsCog(int u) {
        vis[u] = true;
        int s = 1, res = 0;
```

```
        for (auto [w, v] : e[u]) {
            if (vis[v]) continue;
            int t = dfsCog(v);
            res = max(res, t);
            s += t;
        }
        res = max(res, n - s);
        if (res < rem) {
            rem = res;
            cog = u;
        }
        return s;
    }
};
```

点分治 / 树的重心

重心的定义: 删除树上的某一个点, 会得到若干棵子树; 删除某点后, 得到的最大子树最小, 这个点称为重心. 我们假设某个点是重心, 记录此时最大子树的最小值, 遍历完所有点后取最大值即可.

重心的性质: 重心最多可能会有两个, 且此时两个重心相邻.

点分治的一般过程是: 取重心为新树的根, 随后使用 `dfs` 处理当前这棵树, 灵活运用 `child` 和 `pre` 两个数组分别计算通过根节点, 不通过根节点的路径信息, 根据需要进行答案的更新; 再对子树分治, 寻找子树的重心,. 时间复杂度降至 $\mathcal{O}(N \log N)$.

```
int root = 0, MaxTree = 1e18; //分别代表重心下标，最大子树大小
vector<int> vis(n + 1), siz(n + 1);
auto get = [&](auto self, int x, int fa, int n) -> void { // 获取树的重心
    siz[x] = 1;
    int val = 0;
    for (auto [y, w] : ver[x]) {
```

```
        if (y == fa || vis[y]) continue;
        self(self, y, x, n);
        siz[x] += siz[y];
        val = max(val, siz[y]);
    }
    val = max(val, n - siz[x]);
    if (val < MaxTree) {
        MaxTree = val;
        root = x;
    }
};

auto clac = [&](int x) -> void { // 以 x 为新的根，维护询问
    set<int> pre = {0}; // 记录到根节点 x 距离为 i 的路径是否存在
    vector<int> dis(n + 1);
    for (auto [y, w] : ver[x]) {
        if (vis[y]) continue;
        vector<int> child; // 记录 x 的子树节点的深度信息
        auto dfs = [&](auto self, int x, int fa) -> void {
            child.push_back(dis[x]);
            for (auto [y, w] : ver[x]) {
                if (y == fa || vis[y]) continue;
                dis[y] = dis[x] + w;
                self(self, y, x);
            }
        };
        dis[y] = w;
        dfs(dfs, y, x);

        for (auto it : child) {
            for (int i = 1; i <= m; i++) { // 根据询问更新值
                if (q[i] < it || !pre.count(q[i] - it)) continue;
                ans[i] = 1;
```

```
            }
        }
        pre.insert(child.begin(), child.end());
    }
};

auto dfz = [&](auto self, int x, int fa) -> void { // 点分治
    vis[x] = 1; // 标记已经被更新过的旧重心，确保只对子树分治
    clac(x);
    for (auto [y, w] : ver[x]) {
        if (y == fa || vis[y]) continue;
        MaxTree = 1e18;
        get(get, y, x, siz[y]);
        self(self, root, x);
    }
};

get(get, 1, 0, n);
dfz(dfz, root, 0);
```

最近公共祖先 LCA

树链剖分解法 预处理时间复杂度 $\mathcal{O}(N)$；单次查询 $\mathcal{O}(\log N)$，常数较小.

```
struct HLD {
    int n, idx;
    vector<vector<int>> ver;
    vector<int> siz, dep;
    vector<int> top, son, parent;

    HLD(int n) {
        this->n = n;
```

```
    ver.resize(n + 1);
    siz.resize(n + 1);
    dep.resize(n + 1);

    top.resize(n + 1);
    son.resize(n + 1);
    parent.resize(n + 1);
}
void add(int x, int y) { // 建立双向边
    ver[x].push_back(y);
    ver[y].push_back(x);
}
void dfs1(int x) {
    siz[x] = 1;
    dep[x] = dep[parent[x]] + 1;
    for (auto y : ver[x]) {
        if (y == parent[x]) continue;
        parent[y] = x;
        dfs1(y);
        siz[x] += siz[y];
        if (siz[y] > siz[son[x]]) {
            son[x] = y;
        }
    }
}
void dfs2(int x, int up) {
    top[x] = up;
    if (son[x]) dfs2(son[x], up);
    for (auto y : ver[x]) {
        if (y == parent[x] || y == son[x]) continue;
        dfs2(y, y);
    }
}
```

```
    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 clac(int x, int y) { // 查询两点间距离
        return dep[x] + dep[y] - 2 * dep[lca(x, y)];
    }
    void work(int root = 1) { // 在此初始化
        dfs1(root);
        dfs2(root, root);
    }
};
```

树上倍增解法 预处理时间复杂度 $\mathcal{O}(N \log N)$；单次查询 $\mathcal{O}(\log N)$，但是常数比树链剖分解法更大.

封装一: 基础封装, 针对无权图.

```
struct Tree {
    int n;
    vector<vector<int>> ver, val;
    vector<int> lg, dep;
    Tree(int n) {
        this->n = n;
        ver.resize(n + 1);
        val.resize(n + 1, vector<int>(30));
        lg.resize(n + 1);
        dep.resize(n + 1);
```

```
    for (int i = 1; i <= n; i++) { //预处理 log
        lg[i] = lg[i - 1] + (1 << lg[i - 1] == i);
    }
}
void add(int x, int y) { // 建立双向边
    ver[x].push_back(y);
    ver[y].push_back(x);
}
void dfs(int x, int fa) {
    val[x][0] = fa; // 储存 x 的父节点
    dep[x] = dep[fa] + 1;
    for (int i = 1; i <= lg[dep[x]]; i++) {
        val[x][i] = val[val[x][i - 1]][i - 1];
    }
    for (auto y : ver[x]) {
        if (y == fa) continue;
        dfs(y, x);
    }
}
int lca(int x, int y) {
    if (dep[x] < dep[y]) swap(x, y);
    while (dep[x] > dep[y]) {
        x = val[x][lg[dep[x] - dep[y]] - 1];
    }
    if (x == y) return x;
    for (int k = lg[dep[x]] - 1; k >= 0; k--) {
        if (val[x][k] == val[y][k]) continue;
        x = val[x][k];
        y = val[y][k];
    }
    return val[x][0];
}
int clac(int x, int y) { // 倍增查询两点间距离
```

```
        return dep[x] + dep[y] - 2 * dep[lca(x, y)];
    }
    void work(int root = 1) { // 在此初始化
        dfs(root, 0);
    }
};
```

封装二: 扩展封装, 针对有权图, 支持” 倍增查询两点路径上的最大边权功能.

```
struct Tree {
    int n;
    vector<vector<int>> val, Max;
    vector<vector<pair<int, int>>> ver;
    vector<int> lg, dep;
    Tree(int n) {
        this->n = n;
        ver.resize(n + 1);
        val.resize(n + 1, vector<int>(30));
        Max.resize(n + 1, vector<int>(30));
        lg.resize(n + 1);
        dep.resize(n + 1);
        for (int i = 1; i <= n; i++) { //预处理 log
            lg[i] = lg[i - 1] + (1 << lg[i - 1] == i);
        }
    }
    void add(int x, int y, int w) { // 建立双向边
        ver[x].push_back({y, w});
        ver[y].push_back({x, w});
    }
    void dfs(int x, int fa) {
        val[x][0] = fa;
        dep[x] = dep[fa] + 1;
        for (int i = 1; i <= lg[dep[x]]; i++) {
```

```
            val[x][i] = val[val[x][i - 1]][i - 1];
            Max[x][i] = max(Max[x][i - 1], Max[val[x][i - 1]][i - 1]);
        }
        for (auto [y, w] : ver[x]) {
            if (y == fa) continue;
            Max[y][0] = w;
            dfs(y, x);
        }
    }
    int lca(int x, int y) {
        if (dep[x] < dep[y]) swap(x, y);
        while (dep[x] > dep[y]) {
            x = val[x][lg[dep[x] - dep[y]] - 1];
        }
        if (x == y) return x;
        for (int k = lg[dep[x]] - 1; k >= 0; k--) {
            if (val[x][k] == val[y][k]) continue;
            x = val[x][k];
            y = val[y][k];
        }
        return val[x][0];
    }
    int clac(int x, int y) { // 倍增查询两点间距离
        return dep[x] + dep[y] - 2 * dep[lca(x, y)];
    }
    int query(int x, int y) { // 倍增查询两点路径上的最大边权（带权图）
        auto get = [&](int x, int y) -> int {
            int ans = 0;
            if (x == y) return ans;
            for (int i = lg[dep[x]]; i >= 0; i--) {
                if (dep[val[x][i]] > dep[y]) {
                    ans = max(ans, Max[x][i]);
                    x = val[x][i];
```

```
                }
            }
            ans = max(ans, Max[x][0]);
            return ans;
        };
        int fa = lca(x, y);
        return max(get(x, fa), get(y, fa));
    }
    void work(int root = 1) { // 在此初始化
        dfs(root, 0);
    }
};
```

树上路径交

计算两条路径的交点数量, 直接载入任意 LCA 封装即可.

```
int intersection(int x, int y, int X, int Y) {
    vector<int> t = {lca(x, X), lca(x, Y), lca(y, X), lca(y, Y)};
    sort(t.begin(), t.end());
    int r = lca(x, y), R = lca(X, Y);
    if (dep[t[0]] < min(dep[r], dep[R]) || dep[t[2]] < max(dep[r], dep[R])) {
        return 0;
    }
    return 1 + clac(t[2], t[3]);
}
```

树上启发式合并 (DSU on tree)

$\mathcal{O}(N \log N)$.

```
struct HLD {
    vector<vector<int>> e;
```

```
    vector<int> siz, son, cnt;
    vector<LL> ans;
    LL sum, Max;
    int hson;
    HLD(int n) {
        e.resize(n + 1);
        siz.resize(n + 1);
        son.resize(n + 1);
        ans.resize(n + 1);
        cnt.resize(n + 1);
        hson = 0;
        sum = 0;
        Max = 0;
    }
    void add(int u, int v) {
        e[u].push_back(v);
        e[v].push_back(u);
    }
    void dfs1(int u, int fa) {
        siz[u] = 1;
        for (auto v : e[u]) {
            if (v == fa) continue;
            dfs1(v, u);
            siz[u] += siz[v];
            if (siz[v] > siz[son[u]]) son[u] = v;
        }
    }
    void calc(int u, int fa, int val) {
        cnt[color[u]] += val;
        if (cnt[color[u]] > Max) {
            Max = cnt[color[u]];
            sum = color[u];
        } else if (cnt[color[u]] == Max) {
```

```
            sum += color[u];
        }
        for (auto v : e[u]) {
            if (v == fa || v == hson) continue;
            calc(v, u, val);
        }
    }
    void dfs2(int u, int fa, int opt) {
        for (auto v : e[u]) {
            if (v == fa || v == son[u]) continue;
            dfs2(v, u, 0);
        }
        if (son[u]) {
            dfs2(son[u], u, 1);
            hson = son[u]; //记录重链编号，计算的时候跳过
        }
        calc(u, fa, 1);
        hson = 0; //消除的时候所有儿子都清除
        ans[u] = sum;
        if (!opt) {
            calc(u, fa, -1);
            sum = 0;
            Max = 0;
        }
    }
};
```

prufur 序列

对树建立 Prfer 序列 Prfer 是这样建立的: 每次选择一个编号最小的叶结点并删掉它, 然后在序列中记录下它连接到的那个结点. 重复 $n-2$ 次后就只剩下两个结点, 算法结束.

显然使用堆可以做到 $O(n \log n)$ 的复杂度

```cpp
// 代码摘自原文，结点是从 0 标号的
vector<vector<int>> adj;

vector<int> pruefer_code() {
  int n = adj.size();
  set<int> leafs;
  vector<int> degree(n);
  vector<bool> killed(n, false);
  for (int i = 0; i < n; i++) {
    degree[i] = adj[i].size();
    if (degree[i] == 1) leafs.insert(i);
  }

  vector<int> code(n - 2);
  for (int i = 0; i < n - 2; i++) {
    int leaf = *leafs.begin();
    leafs.erase(leafs.begin());
    killed[leaf] = true;
    int v;
    for (int u : adj[leaf])
      if (!killed[u]) v = u;
    code[i] = v;
    if (--degree[v] == 1) leafs.insert(v);
  }
  return code;
}
```

```python
# 结点是从 0 标号的
adj = [[]]


def pruefer_code():
```

```
    n = len(adj)
    leafs = set()
    degree = [0] * n
    killed = [False] * n
    for i in range(1, n):
        degree[i] = len(adj[i])
        if degree[i] == 1:
            leafs.intersection(i)
    code = [0] * (n - 2)
    for i in range(1, n - 2):
        leaf = leafs[0]
        leafs.pop()
        killed[leaf] = True
        for u in adj[leaf]:
            if killed[u] == False:
                v = u
        code[i] = v
        if degree[v] == 1:
            degree[v] = degree[v] - 1
            leafs.intersection(v)
    return code
```

Cayley 公式 (Cayley's formula) 完全图 K_n 有 n^{n-2} 棵生成树.

怎么证明方法很多, 但是用 Prfer 序列证是很简单的. 任意一个长度为 $n-2$ 的值域 $[1,n]$ 的整数序列都可以通过 Prfer 序列双射对应一个生成树, 于是方案数就是 n^{n-2}.

图连通方案数 Prfer 序列可能比你想得还强大. 它能创造比凯莱公式 更通用的公式. 比如以下问题:

> 一个 n 个点 m 条边的带标号无向图有 k 个连通块. 我们希望添加 $k-1$ 条边使得整个图连通. 求方案数.

设 s_i 表示每个连通块的数量. 我们对 k 个连通块构造 Prfer 序列, 然后你发现这并不是普通的 Prfer 序列. 因为每个连通块的连接方法很多. 不能直接干就设啊. 于是设 d_i 为第 i 个连通块的度数. 由于度数之和是边数的两倍, 于是 $\sum_{i=1}^{k} d_i = 2k-2$. 则对于给定的 d 序列构造 Prufer 序列的方案数是

$$n^{k-2} \cdot \prod_{i=1}^{k} s_i$$

重链剖分

```
struct HPD_tree
{
    int tree_size;
    bool is_hpd_init = false;
    std::vector<std::vector<std::pair<int, i64>>> adj;
    std::vector<int> Fa, size, hson, top, rank, dfn, depth;
    HPD_tree(int n = 0) {
        tree_size = n;
        adj.resize(tree_size + 1);
    }
    void add_edge(int u, int v, i64 w = 1) {
        adj[u].push_back({ v,w });
        adj[v].push_back({ u,w });
    }
    void HPD_init() {
        is_hpd_init = true;
        Fa.assign(tree_size + 1, 0);
        size.assign(tree_size + 1, 0);
        hson.assign(tree_size + 1, 0);
        top.assign(tree_size + 1, 0);
        rank.assign(tree_size + 1, 0);
```

```
        dfn.assign(tree_size + 1, 0);
        depth.assign(tree_size + 1, 0);
        std::function<void(int, int, int)> dfs1 = [&](int u, int p, int d)->void {
            hson[u] = 0;
            size[hson[u]] = 0;
            size[u] = 1;
            depth[u] = d;
            for (auto [v, w] : adj[u])if (v != p) {
                dfs1(v, u, d + 1);
                size[u] += size[v];
                Fa[v] = u;
                if (size[v] > size[hson[u]]) {
                    hson[u] = v;
                }
            }
            };
        dfs1(1, 0, 0);
        int tot = 0;
        std::function<void(int, int, int)> dfs2 = [&](int u, int p, int t)->void {
            top[u] = t;
            dfn[u] = ++tot;
            rank[tot] = u;
            if (hson[u]) {
                dfs2(hson[u], u, t);
                for (auto [v, w] : adj[u])if (v != p && v != hson[u]) {
                    dfs2(v, u, v);
                }
            }
            };
        dfs2(1, 0, 1);
    }
    int lca(int u, int v) {
        if (!is_hpd_init)HPD_init();
```

```
        while (top[u] != top[v]) {
            if (depth[top[u]] > depth[top[v]])
                u = Fa[top[u]];
            else
                v = Fa[top[v]];
        }
        return depth[u] > depth[v] ? v : u;
    }
    i64 dist(int u, int v) {
        int w = lca(u, v);
        return depth[u] - depth[w] + depth[v] - depth[w] + 1;
    }
    a3 get_diam() {
        i64 cur; int pos;
        std::function<void(int, int, i64)> dfs = [&](int u, int p, i64 d) {
            if (d > cur) {
                cur = d;
                pos = u;
            }
            for (auto [v, dis] : adj[u])if (v != p) {
                dfs(v, u, d + dis);
            }
            };
        cur = 0, pos = 1;
        dfs(pos, 0, cur);
        int u = pos;
        cur = 0;
        dfs(pos, 0, cur);
        int v = pos;
        return { u,v,cur };
    }
};
```

轻重链剖分/树链剖分

将线段树处理的部分分离, 方便修改. 支持链上查询/修改, 子树查询/修改, 建树时间复杂度 $\mathcal{O}(N \log N)$, 单次查询时间复杂度 $\mathcal{O}(\log^2 N)$.

```
struct Segt {
    struct node {
        int l, r, w, lazy;
    };
    vector<int> w;
    vector<node> t;

    Segt() {}
    #define GL (k << 1)
    #define GR (k << 1 | 1)

    void init(vector<int> in) {
        int n = in.size() - 1;
        w.resize(n + 1);
        for (int i = 1; i <= n; i++) {
            w[i] = in[i];
        }
        t.resize(n * 4 + 1);
        auto build = [&](auto self, int l, int r, int k = 1) {
            if (l == r) {
                t[k] = {l, r, w[l], 0}; // 如果有赋值为 0 的操作, 则懒标记必须要
                return;
            }
            t[k] = {l, r};
            int mid = (l + r) / 2;
            self(self, l, mid, GL);
            self(self, mid + 1, r, GR);
            pushup(k);
        };
```

```
    build(build, 1, n);
}
void pushdown(node &p, int lazy) { /* 在此更新下递函数 */
    p.w += (p.r - p.l + 1) * lazy;
    p.lazy += lazy;
}
void pushdown(int k) { // 不需要动
    if (t[k].lazy == 0) return;
    pushdown(t[GL], t[k].lazy);
    pushdown(t[GR], t[k].lazy);
    t[k].lazy = 0;
}
void pushup(int k) { // 不需要动
    auto pushup = [&](node &p, node &l, node &r) { /* 在此更新上传函数 */
        p.w = l.w + r.w;
    };
    pushup(t[k], t[GL], t[GR]);
}
void modify(int l, int r, int val, int k = 1) {
    if (l <= t[k].l && t[k].r <= r) {
        pushdown(t[k], val);
        return;
    }
    pushdown(k);
    int mid = (t[k].l + t[k].r) / 2;
    if (l <= mid) modify(l, r, val, GL);
    if (mid < r) modify(l, r, val, GR);
    pushup(k);
}
int ask(int l, int r, int k = 1) {
    if (l <= t[k].l && t[k].r <= r) {
        return t[k].w;
    }
```

```
        pushdown(k);
        int mid = (t[k].l + t[k].r) / 2;
        int ans = 0;
        if (l <= mid) ans += ask(l, r, GL);
        if (mid < r) ans += ask(l, r, GR);
        return ans;
    }
};

struct HLD {
    int n, idx;
    vector<vector<int>> ver;
    vector<int> siz, dep;
    vector<int> top, son, parent;
    vector<int> in, id, val;
    Segt segt;

    HLD(int n) {
        this->n = n;
        ver.resize(n + 1);
        siz.resize(n + 1);
        dep.resize(n + 1);

        top.resize(n + 1);
        son.resize(n + 1);
        parent.resize(n + 1);

        idx = 0;
        in.resize(n + 1);
        id.resize(n + 1);
        val.resize(n + 1);
    }
    void add(int x, int y) { // 建立双向边
```

```
        ver[x].push_back(y);
        ver[y].push_back(x);
    }
    void dfs1(int x) {
        siz[x] = 1;
        dep[x] = dep[parent[x]] + 1;
        for (auto y : ver[x]) {
            if (y == parent[x]) continue;
            parent[y] = x;
            dfs1(y);
            siz[x] += siz[y];
            if (siz[y] > siz[son[x]]) {
                son[x] = y;
            }
        }
    }
    void dfs2(int x, int up) {
        id[x] = ++idx;
        val[idx] = in[x]; // 建立编号
        top[x] = up;
        if (son[x]) dfs2(son[x], up);
        for (auto y : ver[x]) {
            if (y == parent[x] || y == son[x]) continue;
            dfs2(y, y);
        }
    }
    void modify(int l, int r, int val) { // 链上修改
        while (top[l] != top[r]) {
            if (dep[top[l]] < dep[top[r]]) {
                swap(l, r);
            }
            segt.modify(id[top[l]], id[l], val);
            l = parent[top[l]];
```

```
        }
        if (dep[l] > dep[r]) {
            swap(l, r);
        }
        segt.modify(id[l], id[r], val);
    }
    void modify(int root, int val) { // 子树修改
        segt.modify(id[root], id[root] + siz[root] - 1, val);
    }
    int ask(int l, int r) { // 链上查询
        int ans = 0;
        while (top[l] != top[r]) {
            if (dep[top[l]] < dep[top[r]]) {
                swap(l, r);
            }
            ans += segt.ask(id[top[l]], id[l]);
            l = parent[top[l]];
        }
        if (dep[l] > dep[r]) {
            swap(l, r);
        }
        return ans + segt.ask(id[l], id[r]);
    }
    int ask(int root) { // 子树查询
        return segt.ask(id[root], id[root] + siz[root] - 1);
    }
    void work(auto in, int root = 1) { // 在此初始化
        assert(in.size() == n + 1);
        this->in = in;
        dfs1(root);
        dfs2(root, root);
        segt.init(val); // 建立线段树
    }
```

```
    void work(int root = 1) { // 在此初始化
        dfs1(root);
        dfs2(root, root);
        segt.init(val); // 建立线段树
    }
};
```

/END/

动态规划

01 背包

有 n 件物品和一个容量为 W 的背包, 第 i 件物品的体积为 $w[i]$, 价值为 $v[i]$, 求解将哪些物品装入背包中使总价值最大.

思路:

当放入一个价值为 $w[i]$ 的物品后, 价值增加了 $v[i]$, 于是我们可以构建一个二维的 $dp[i][j]$ 数组, 装入第 i 件物品时, 背包容量为 j 能实现的 **最大价值**, 可以得到 **转移方程** $dp[i][j] = max(dp[i-1][j], dp[i-1][j-w[i]] + v[i])$.

```
for (int i = 1; i <= n; i++)
    for (int j = 0; j <= W; j++){
        dp[i][j] = dp[i - 1][j];
        if (j >= w[i])
            dp[i][j] = max(dp[i][j], dp[i - 1][j - w[i]] + v[i]);
    }
```

我们可以发现, 第 i 个物品的状态是由第 $i-1$ 个物品转移过来的, 每次的 j 转移过来后, 第 $i-1$ 个方程的 j 已经没用了, 于是我们想到可以把二维方程压缩成 **一维**的, 用以 **优化空间复杂度**.

```
for (int i = 1; i <= n; i++)  //当前装第 i 件物品
    for (int j = W; j >= w[i]; j--)  //背包容量为 j
```

```
        dp[j] = max(dp[j], dp[j - w[i]] + v[i]);  //判断背包容量为 j 的情况下能是实
```

完全背包

有 n 件物品和一个容量为 W 的背包, 第 i 件物品的体积为 $w[i]$, 价值为 $v[i]$, 每件物品有**无限个**, 求解将哪些物品装入背包中使总价值最大.

思路:

思路和 **01 背包**差不多, 但是每一件物品有**无限个**, 其实就是从每 **种**物品中取 $0, 1, 2,... $ 件物品加入背包中

```
for (int i = 1; i <= n; i++)
    for (int j = 0; j <= W; j++)
        for (int k = 0; k * w[i] <= j; k++)    //选取几个物品
            dp[i][j] = max(dp[i][j], dp[i - 1][j - k * w[i]] + k * v[i]);
```

实际上, 我们可以发现, 取 k 件物品可以从取 $k-1$ 件转移过来, 那么我们就可以将 k 的循环优化掉

```
for (int i = 1; i <= n; i++)
    for (int j = 0; j <= W; j++){
        dp[i][j] = dp[i - 1][j];
        if (j >= w[i])
            dp[i][j] = max(dp[i][j], dp[i][j - w[i]] + v[i]);
    }
```

和 01 背包类似地压缩成一维

```
for (int i = 1; i <= n; i++)
    for (int j = w[i]; j <= W; j++)
        dp[j] = max(dp[j], dp[j - w[i]] + v[i]);
```

多重背包

有 n 种物品和一个容量为 W 的背包, 第 i 种物品的体积为 $w[i]$, 价值为 $v[i]$, 数量为 $s[i]$, 求解将哪些物品装入背包中使总价值最大.

思路:

对于每一种物品, 都有 $s[i]$ 种取法, 我们可以将其转化为 **01 背包**问题

```
for (int i = 1; i <= n; i++){
    for (int j = W; j >= 0; j--)
        for (int k = 0; k <= s[i]; k++){
            if (j - k * w[i] < 0) break;
            dp[j] = max(dp[j], dp[j - k * w[i]] + k * v[i]);
        }
```

上述方法的时间复杂度为 $O(n * m * s)$.

```
for (int i = 1; i <= n; i++){
    scanf("%lld%lld%lld", &x, &y, &s);  //x 为体积, y 为价值, s 为数量
    t = 1;
    while (s >= t){
        w[++num] = x * t;
        v[num] = y * t;
        s -= t;
        t *= 2;
    }
    w[++num] = x * s;
    v[num] = y * s;
}
for (int i = 1; i <= num; i++)
    for (int j = W; j >= w[i]; j--)
        dp[j] = max(dp[j], dp[j - w[i]] + v[i]);
```

尽管采用了 **二进制优化**, 时间复杂度还是太高, 采用 **单调队列优化**, 将时间复杂度优化至 $O(n * m)$

```cpp
#include <bits/stdc++.h>
using namespace std;
const int N = 2e5 + 10;
int n, W, w, v, s, f[N], g[N], q[N];
int main(){
    ios::sync_with_stdio(false);cin.tie(0);
    cin >> n >> W;
    for (int i = 0; i < n; i ++ ){
        memcpy ( g, f, sizeof f);
        cin >> w >> v >> s;
        for (int j = 0; j < w; j ++ ){
            int head = 0, tail = -1;
            for (int k = j; k <= W; k += w){
                if ( head <= tail && k - s * w > q[head] ) head ++ ;//保证队列长度 <=
                while ( head <= tail && g[q[tail]] - (q[tail] - j) / w * v <= g[k] - (k
                q[ ++ tail] = k;
                f[k] = g[q[head]] + (k - q[head]) / w * v;
            }
        }
    }
    cout << f[W] << "\n";
    return 0;
}
```

混合背包

放入背包的物品可能只有 **1** 件 (01 背包), 也可能有**无限**件 (完全背包), 也可能只有**可数的几件** (多重背包).

思路:

分类讨论即可, 哪一类就用哪种方法去 dp.

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

```cpp
using namespace std;
int n, W, w, v, s;
int main(){
    cin >> n >> W;
    vector <int> f(W + 1);
    for (int i = 0; i < n; i ++ ){
        cin >> w >> v >> s;
        if (s == -1){
            for (int j = W; j >= w; j -- )
                f[j] = max(f[j], f[j - w] + v);
        }
        else if (s == 0){
            for (int j = w; j <= W; j ++ )
                f[j] = max(f[j], f[j - w] + v);
        }
        else {
            int t = 1, cnt = 0;
            vector <int> x(s + 1), y(s + 1);
            while (s >= t){
                x[++cnt] = w * t;
                y[cnt] = v * t;
                s -= t;
                t *= 2;
            }
            x[++cnt] = w * s;
            y[cnt] = v * s;
            for (int i = 1; i <= cnt; i ++ )
                for (int j = W; j >= x[i]; j -- )
                    f[j] = max(f[j], f[j - x[i]] + y[i]);
        }
    }
    cout << f[W] << "\n";
    return 0;
```

```
}
```

二维费用的背包

有 n 件物品和一个容量为 W 的背包, 背包能承受的最大重量为 M, 每件物品只能用一次, 第 i 件物品的体积是 $w[i]$, 重量为 $m[i]$, 价值为 $v[i]$, 求解将哪些物品放入背包中使总体积不超过背包容量, 总重量不超过背包最大容量, 且总价值最大.

思路:

背包的限制条件由一个变成两个, 那么我们的循环再多一维即可.

```
for (int i = 1; i <= n; i++)
    for (int j = W; j >= w; j--)  //容量限制
        for (int k = M; k >= m; k--)  //重量限制
            dp[j][k] = max(dp[j][k], dp[j - w][k - m] + v);
```

分组背包

有 n 组物品, 一个容量为 W 的背包, 每组物品有若干, 同一组的物品最多选一个, 第 i 组第 j 件物品的体积为 $w[i][j]$, 价值为 $v[i][j]$, 求解将哪些物品装入背包, 可使物品总体积不超过背包容量, 且使总价值最大.

思路:

考虑每**组**中的**某件**物品选不选, 可以选的话, 去下一组选下一个, 否则在这组继续寻找可以选的物品, 当这组遍历完后, 去下一组寻找.

```
#include <bits/stdc++.h>
using namespace std;
const int N = 110;
int n, W, s[N], w[N][N], v[N][N], dp[N];
int main(){
    cin >> n >> W;
```

```
    for (int i = 1; i <= n; i++){
        scanf("%d", &s[i]);
        for (int j = 1; j <= s[i]; j++)
            scanf("%d %d", &w[i][j], &v[i][j]);
    }
    for (int i = 1; i <= n; i++)
        for (int j = W; j >= 0; j--)
            for (int k = 1; k <= s[i]; k++)
                if (j - w[i][k] >= 0)
                    dp[j] = max(dp[j], dp[j - w[i][k]] + v[i][k]);
    cout << dp[W] << "\n";
    return 0;
}
```

有依赖的背包

有 n 个物品和一个容量为 W 的背包, 物品之间有依赖关系, 且之间的依赖关系组成一颗 **树**的形状, 如果选择一个物品, 则必须选择它的 **父节点**, 第 i 件物品的体积是 $w[i]$, 价值为 $v[i]$, 依赖的父节点的编号为 $p[i]$, 若 $p[i]$ 等于 -1, 则为 **根节点**. 求将哪些物品装入背包中, 使总体积不超过总容量, 且总价值最大.

思路:

定义 $f[i][j]$ 为以第 i 个节点为根, 容量为 j 的背包的最大价值. 那么结果就是 $f[root][W]$, 为了知道根节点的最大价值, 得通过其子节点来更新. 所以采用递归的方式. 对于每一个点, 先将这个节点装入背包, 然后找到剩余容量可以实现的最大价值, 最后更新父节点的最大价值即可.

```
#include <bits/stdc++.h>
using namespace std;
const int N = 110;
int n, W, w[N], v[N], p, f[N][N], root;
vector <int> g[N];
```

```
void dfs(int u){
    for (int i = w[u]; i <= W; i ++ )
        f[u][i] = v[u];
    for (auto v : g[u]){
        dfs(v);
        for (int j = W; j >= w[u]; j -- )
            for (int k = 0; k <= j - w[u]; k ++ )
                f[u][j] = max(f[u][j], f[u][j - k] + f[v][k]);
    }
}
int main(){
    cin >> n >> W;
    for (int i = 1; i <= n; i ++ ){
        cin >> w[i] >> v[i] >> p;
        if (p == -1) root = i;
        else g[p].push_back(i);
    }
    dfs(root);
    cout << f[root][W] << "\n";
    return 0;
}
```

背包问题求方案数

有 n 件物品和一个容量为 W 的背包, 每件物品只能用一次, 第 i 件物品的重量为 $w[i]$, 价值为 $v[i]$, 求解将哪些物品放入背包使总重量不超过背包容量, 且总价值最大, 输出 **最优选法的方案数**, 答案可能很大, 输出答案模 10^9+7 的结果.

思路:

开一个储存方案数的数组 cnt,$cnt[i]$ 表示容量为 i 时的 **方案数**, 先将 cnt 的每一个值都初始化为 1, 因为 **不装任何东西就是一种方案**, 如果装入这件物品使总的价值 **更大**, 那么装入后的方案数 **等于**装之前的方案数, 如果装入

后总价值 **相等**, 那么方案数就是 **二者之和**

```
#include <bits/stdc++.h>
using namespace std;
#define LL long long
const int mod = 1e9 + 7, N = 1010;
LL n, W, cnt[N], f[N], w, v;
int main(){
    cin >> n >> W;
    for (int i = 0; i <= W; i ++ )
        cnt[i] = 1;
    for (int i = 0; i < n; i ++ ){
        cin >> w >> v;
        for (int j = W; j >= w; j -- )
            if (f[j] < f[j - w] + v){
                f[j] = f[j - w] + v;
                cnt[j] = cnt[j - w];
            }
            else if (f[j] == f[j - w] + v){
                cnt[j] = (cnt[j] + cnt[j - w]) % mod;
            }
    }
    cout << cnt[W] << "\n";
    return 0;
}
```

背包问题求具体方案

有 n 件物品和一个容量为 W 的背包, 每件物品只能用一次, 第 i 件物品的重量为 $w[i]$, 价值为 $v[i]$, 求解将哪些物品放入背包使总重量不超过背包容量, 且总价值最大, 输出 **字典序最小的方案**

思路:

01 背包求解最优方案中 **字典序最小的方案**, **首先**我们先求 **01 背包**, 因为这道题需要输出方案, 所以我们 **不能压缩空间**, 得保留每一步的方案. **又**由于输出字典序最小的, 所以我们应该反着来, 从 n 到 1 求解最优解, 那么 $dp[1][W]$ 就是最优的解.

```
for (int i = n; i >= 1; i--)
    for (int j = 0; j <= W; j++){
        dp[i][j] = dp[i + 1][j];
        if (j >= w[i])
            dp[i][j] = max(dp[i][j], dp[i + 1][j - w[i]] + v[i]);
    }
```

接下来就是输出的问题, 如何判断这个物品**被选中**, 如果 $dp[i][k] = dp[i+1][k - w[i]] + v[i]$, 说明选择了第 i 个物品是最优的选择方案.

```
for (int i = 1; i <= n; i++)
    if (W - w[i] >= 0 && dp[i][W] == dp[i + 1][W - w[i]] + v[i]){
        cout << i << " ";
        W -= w[i];
    }
```

数位 DP

```
/* pos 表示当前枚举到第几位
sum 表示 d 出现的次数
limit 为 1 表示枚举的数字有限制
zero 为 1 表示有前导 0
d 表示要计算出现次数的数 */
const int N = 15;
LL dp[N][N];
int num[N];
LL dfs(int pos, LL sum, int limit, int zero, int d) {
    if (pos == 0) return sum;
    if (!limit && !zero && dp[pos][sum] != -1) return dp[pos][sum];
```

```
    LL ans = 0;
    int up = (limit ? num[pos] : 9);
    for (int i = 0; i <= up; i++) {
        ans += dfs(pos - 1, sum + ((!zero || i) && (i == d)), limit && (i == num[pos]),
                   zero && (i == 0), d);
    }
    if (!limit && !zero) dp[pos][sum] = ans;
    return ans;
}
LL solve(LL x, int d) {
    memset(dp, -1, sizeof dp);
    int len = 0;
    while (x) {
        num[++len] = x % 10;
        x /= 10;
    }
    return dfs(len, 0, 1, 1, d);
}
```

```
#include<bits/stdc++.h>
#define int long long
using namespace std;
constexpr int MAXN = 24 + 10;
int a[MAXN], mod, f[MAXN][MAXN * 10][MAXN * 10];

int dfs(int pos, int sum, int cur, bool lead0, bool lim) {
    if (!pos)return !lead0 && sum == mod && cur == 0;
    int& now = f[pos][cur][sum];
    if (!lead0 && !lim && ~now)return now;
    int up = lim ? a[pos] : 9, res = 0;
    for (int i = 0;i <= up;++i)
        res += dfs(pos - 1, sum + i, (cur * 10 + i) % mod, lead0 && !i, lim && i == up)
    if (!lead0 && !lim)now = res;
    return res;
```

```
}

signed main() {
    ios::sync_with_stdio(false);
    cin.tie(0), cout.tie(0);
    int n;cin >> n;
    int len = 0;
    while (n)a[++len] = n % 10, n /= 10;
    int res = 0;
    for (int i = 1;i <= len * 9;++i) {
        mod = i;memset(f, -1, sizeof f);
        res += dfs(len, 0, 0, 1, 1);
    }
    cout << res;
    return 0;
}
```

状压 DP

题意: 在 $n * n$ 的棋盘里面放 k 个国王, 使他们互不攻击, 共有多少种摆放方案. 国王能攻击到它上下左右, 以及左上左下右上右下八个方向上附近的各一个格子, 共 8 个格子.

```
#include <bits/stdc++.h>
using namespace std;
#define LL long long
const int N = 15, M = 150, K = 1500;
LL n, k;
LL cnt[K];    //每个状态的二进制中 1 的数量
LL tot;    //合法状态的数量
LL st[K];    //合法的状态
LL dp[N][M][K];    //第 i 行, 放置了 j 个国王, 状态为 k 的方案数
int main(){
```

```
    ios::sync_with_stdio(false);cin.tie(0);
    cin >> n >> k;
    for (int s = 0; s < (1 << n); s ++ ){  //找出合法状态
        LL sum = 0, t = s;
        while(t){  //计算 1 的数量
            sum += (t & 1);
            t >>= 1;
        }
        cnt[s] = sum;
        if ( (( (s << 1) | (s >> 1) ) & s) == 0 ){  //判断合法性
            st[ ++ tot] = s;
        }
    }
    dp[0][0][0] = 1;
    for (int i = 1; i <= n + 1; i ++ ){
        for (int j1 = 1; j1 <= tot; j1 ++ ){    //当前的状态
            LL s1 = st[j1];
            for (int j2 = 1; j2 <= tot; j2 ++ ){    //上一行的状态
                LL s2 = st[j2];
                if ( ( (s2 | (s2 << 1) | (s2 >> 1)) & s1 ) == 0 ){
                    for (int j = 0; j <= k; j ++ ){
                        if (j - cnt[s1] >= 0)
                            dp[i][j][s1] += dp[i - 1][j - cnt[s1]][s2];
                    }
                }
            }
        }
    }
    cout << dp[n + 1][k][0] << "\n";
    return 0;
}
```

最短 **Hamilton** 路径

```
using namespace std;

const int N = 20,M = 1 << N;

int n;
int w[N][N];
int f[M][N];//第一维表示是否访问到该点的压缩状态，第二维是走到点 j
            //f[i][j] 表示状态为 i 并且到 j 的最短路径

int main(){
    cin>>n;
    for (int i = 0; i < n; i ++ )
        for (int j = 0; j < n; j ++ )//读入 i 到 j 的距离
            cin>>w[i][j];
    memset(f, 0x3f, sizeof f);
    f[1][0]=0;
    for (int i = 0; i < 1 << n; i ++ )//枚举压缩的状态
        for (int j = 0; j < n; j ++ )//枚举到 0~j 的点
            if(i >> j & 1)//该状态存在 j 点
                for (int k = 0; k < n; k ++ )//枚举从 j 倒数第二个点 k
                    if(i >> k & 1)//倒数点 k 存在
                        f[i][j]=min(f[i][j],f[i-(1<<j)][k]+w[k][j]);//状态转移方程，在
    cout<<f[(1<<n)-1][n-1]<<endl;//输出状态全满也就是所有点都经过且到最后一个点的
    return 0;
}
```

状态转移方程:

```
f[i][j]=min(f[i][j],f[i-(1<<j)][k]+w[k][j]);
```

常用例题

题意: 在一篇文章 (包含大小写英文字母, 数字, 和空白字符 (制表/空格/回车) 中寻找 helloworld(任意一个字母的大小写都行) 的子序列出现了多少

次, 输出结果对 10^9+7 的余数.

字符串 DP , 构建一个二维 DP 数组,$dp[i][j]$ 的 i 表示文章中的第几个字符,j 表示寻找的字符串的第几个字符, 当字符串中的字符和文章中的字符相同时, 即找到符合条件的字符, `dp[i][j] = dp[i - 1][j] + dp[i - 1][j - 1]` , 因为字符串中的每个字符不会对后面的结果产生影响, 所以 DP 方程可以优化成一维的, 由于字符串中有重复的字符, 所以比较时应该从后往前.

```
#include <bits/stdc++.h>
using namespace std;
#define LL long long
const int mod = 1e9 + 7;
char c, s[20] = "!helloworld";
LL dp[20];
int main(){
    dp[0] = 1;
    while ((c = getchar()) != EOF)
        for (int i = 10; i >= 1; i--)
            if (c == s[i] || c == s[i] - 32)
                dp[i] = (dp[i] + dp[i - 1]) % mod;
    cout << dp[10] << "\n";
    return 0;
}
```

题意:(最长括号匹配) 给一个只包含 (,),[,] 的非空字符串, “() 和” [] 是匹配的, 寻找字符串中最长的括号匹配的子串, 若有两串长度相同, 输出靠前的一串.

设给定的字符串为 `s` , 可以定义数组 $dp[i], dp[i]$ 表示以 $s[i]$ 结尾的字符串里最长的括号匹配的字符. 显然, 从 $i-dp[i]+1$ 到 i 的字符串是括号匹配的, 当找到一个字符是) 或] 时, 再去判断第 $i-1-dp[i-1]$ 的字符和第 i 位的字符是否匹配, 如果是, 那么 `dp[i] = dp[i - 1] + 2 + dp[i - 2 - dp[i - 1]]` .

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

```
using namespace std;
const int maxn = 1e6 + 10;
string s;
int len, dp[maxn], ans, id;
int main(){
    cin >> s;
    len = s.length();
    for (int i = 1; i < len; i++){
        if ((s[i] == ')' && s[i - 1 - dp[i - 1]] == '(' ) || (s[i] == ']' && s[i - 1 -
            dp[i] = dp[i - 1] + 2 + dp[i - 2 - dp[i - 1]];
            if (dp[i] > ans) {
                ans = dp[i];  //记录长度
                id = i;  //记录位置
            }
        }
    }
    for (int i = id - ans + 1; i <= id; i++)
        cout << s[i];
    cout << "\n";
    return 0;
}
```

题意: 去掉区间内包含“4 和”62 的数字, 输出剩余的数字个数

```
int T,n,m,len,a[20];//a 数组用于判断每一位能取到的最大值
ll l,r,dp[20][15];
ll dfs(int pos,int pre,int limit){//记搜
    //pos 搜到的位置,pre 前一位数
    //limit 判断是否有最高位限制
    if(pos>len) return 1;//剪枝
    if(dp[pos][pre]!=-1 && !limit) return dp[pos][pre];//记录当前值
    ll ret=0;//暂时记录当前方案数
    int res=limit?a[len-pos+1]:9;//res 当前位能取到的最大值
```

```
    for(int i=0;i<=res;i++)
        if(!(i==4 || (pre==6 && i==2)))
            ret+=dfs(pos+1,i,i==res&&limit);
    if(!limit) dp[pos][pre]=ret;//当前状态方案数记录
    return ret;
}
ll part(ll x){//把数按位拆分
    len=0;
    while(x) a[++len]=x%10,x/=10;
    memset(dp,-1,sizeof dp);//初始化-1(因为有可能某些情况下的方案数是 0)
    return dfs(1,0,1);//进入记搜
}
int main(){
    cin>>n;
    while(n--){
        cin>>l>>r;
        if(l==0 && r==0)break;
        if(l) printf("%lld\n",part(r)-part(l-1));//[l,r](l!=0)
        else printf("%lld\n",part(r)-part(l));//从 0 开始要特判
    }
}
```

SOSdp 高维前缀和

子集向超集转移

```
for(int j = 0; j < n; j++)
    for(int i = 0; i < 1 << n; i++)
        if(i >> j & 1) f[i] += f[i ^ (1 << j)];
```

超集向子集转移

```
for(int j = 0; j < n; j++)
    for(int i = (1 << n) - 1; i >= 0 ; i--)
```

```
        if(!(i >> j & 1)) f[i] += f[i ^ (1 << j)]
```

汉明权重

```
for (int i = 0; (1<<i)-1 <= n; i++) {
    for (int x = (1<<i)-1, t; x <= n; t = x+(x&-x), x = x ? (t|((((t&-t)/(x&-x))>>1)-1)
        // todo
    }
}
```

组合数学

组合数

debug 提供一组测试数据:$\binom{132}{66} = 377$’ 389’ 666’ 165’ 540’ 953’ 244’ 592’ 352’ 291’ 892’ 721’ 700, 模数为 998244353 时为 $241'200'029$;10^9+7 时为 598375978.

质因数分解 此法适用于:$1 < n, m, \text{MOD} < 10^7$ 的情况.

```
int n,m,p,b[10000005],prime[1000005],t,min_prime[10000005];
void euler_Prime(int n){//用欧拉筛求出 1~n 中每个数的最小质因数的编号是多少，保
    for(int i=2;i<=n;i++){
        if(b[i]==0){
            prime[++t]=i;
            min_prime[i]=t;
        }
        for(int j=1;j<=t&&i*prime[j]<=n;j++){
            b[prime[j]*i]=1;
            min_prime[prime[j]*i]=j;
            if(i%prime[j]==0) break;
        }
    }
}
long long c(int n,int m,int p){//计算 C(n,m)%p 的值
    euler_Prime(n);
    int a[t+5];//t 代表 1~n 中质数的个数 ,a[i] 代表编号为 i 的质数在答案中出现的
    for(int i=1;i<=t;i++) a[i]=0;//注意清 0，一开始是随机数
```

```
    for(int i=n;i>=n-m+1;i--){//处理分子
        int x=i;
        while (x!=1){
            a[min_prime[x]]++;//注意 min_prime 中保存的是这个数的最小质因数的编号
            x/=prime[min_prime[x]];
        }
    }
    for(int i=1;i<=m;i++){//处理分母
        int x=i;
        while (x!=1){
            a[min_prime[x]]--;
            x/=prime[min_prime[x]];
        }
    }
    long long ans=1;
    for(int i=1;i<=t;i++){//枚举质数的编号，看它出现了几次
        while(a[i]>0){
            ans=ans*prime[i]%p;
            a[i]--;
        }
    }
    return ans;
}
int main(){
    cin>>n>>m;
    m=min(m,n-m);//小优化
    cout<<c(n,m,MOD);
}
```

杨辉三角 (精确计算) 60 以内 long long 可解,130 以内 __int128 可解.

```
vector C(n + 1, vector<int>(n + 1));
C[0][0] = 1;
```

```
for (int i = 1; i <= n; i++) {
    C[i][0] = 1;
    for (int j = 1; j <= n; j++) {
        C[i][j] = C[i - 1][j] + C[i - 1][j - 1];
    }
}
cout << C[n][m] << endl;
```

常用组合数公式

$$k * C_n^k = n * C_{n-1}^{k-1}$$

;

$$C_k^n * C_m^k = C_m^n * C_{m-n}^{m-k}$$

;

$$C_n^k + C_n^{k+1} = C_{n+1}^{k+1}$$

;

$$\sum_{i=0}^{n} C_n^i = 2^n$$

;

$$\sum_{k=0}^{n} (-1)^k * C_n^k = 0$$

.

$$\begin{cases} f * n = \sum *i = 0^n \binom{n}{i} g * i \Leftrightarrow g_n = \sum *i = 0^n (-1)^{n-i} \binom{n}{i} f * i \\ f_k = \sum *i = k^n \binom{i}{k} g * i \Leftrightarrow g_k = \sum *i = k^n (-1)^{i-k} \binom{i}{k} f_i \end{cases}$$

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

;

$$\sum_{i=1}^{n} i^2 \binom{n}{i} = n * (n+1) * 2^{n-2}$$

;

$$\sum_{i=1}^{n}\frac{1}{i}\binom{n}{i}=\sum_{i=1}^{n}\frac{1}{i}$$

;

$$\binom{\sum_{i=0}^{n}\binom{n}{i}^2=2n}{n}$$

;

$$\sum_{i=1}^{n}\sum_{j=i+1}^{n}(a_ib_j-a_jb_i)^2=(\sum_{i=1}^{n}a_i)^2(\sum_{i=1}^{n}b_i)^2-(\sum_{i=1}^{n}a_ib_i)^2$$

lucas 定理

Lucas 定理内容如下: 对于质数 p, 有

$$\binom{n}{m}\bmod p=\binom{\lfloor n/p\rfloor}{\lfloor m/p\rfloor}\cdot\binom{n\bmod p}{m\bmod p}\bmod p$$

观察上述表达式, 可知 $n \bmod p$ 和 $m \bmod p$ 一定是小于 p 的数, 可以直接求解, $\binom{\lfloor n/p\rfloor}{\lfloor m/p\rfloor}$ 可以继续用 Lucas 定理求解. 这也就要求 p 的范围不能够太大, 一般在 10^5 左右. 边界条件: 当 $m=0$ 的时候, 返回 1.

时间复杂度为 $O(f(p)+g(n)\log n)$, 其中 $f(n)$ 为预处理组合数的复杂度,$g(n)$ 为单次求组合数的复杂度.

```cpp
long long Lucas(long long n, long lm, long long p) {
  if (m == 0) return 1;
  return (C(n % p, m % p, p) * Lucas(n / p, m / p, p)) % p;
}
```

```python
def Lucas(n, m, p):
    if m == 0:
        return 1
    return (C(n % p, m % p, p) * Lucas(n // p, m // p, p)) % p
```

范德蒙德卷积公式

在数量为 $n+m$ 的堆中选 k 个元素, 和分别在数量为 n,m 的堆中选 $i,k-i$ 个元素的方案数是相同的, 即 $\sum_{i=0}^{k}\binom{n}{i}\binom{m}{k-i}=\binom{n+m}{k}$;

变体:

- $\sum_{i=0}^{k} C_{i+n}^{i}=C_{k+n+1}^{k}$;
- $\sum_{i=0}^{k} C_{n}^{i} * C_{m}^{i}=\sum_{i=0}^{k} C_{n}^{i} * C_{m}^{m-i}=C_{n+m}^{n}$.

推论 1 及证明

$$\sum_{i=-r}^{s}\binom{n}{r+i}\binom{m}{s-i}=\binom{n+m}{r+s}$$

推论 2 及证明

$$\sum_{i=1}^{n}\binom{n}{i}\binom{n}{i-1}=\sum_{i=0}^{n-1}\binom{n}{i+1}\binom{n}{i}=\sum_{i=0}^{n-1}\binom{n}{n-1-i}\binom{n}{i}=\binom{2n}{n-1}$$

推论 3 及证明

$$\sum_{i=0}^{n}\binom{n}{i}^{2}=\sum_{i=0}^{n}\binom{n}{i}\binom{n}{n-i}=\binom{2n}{n}$$

推论 4 及证明

$$\sum_{i=0}^{m}\binom{n}{i}\binom{m}{i}=\sum_{i=0}^{m}\binom{n}{i}\binom{m}{m-i}=\binom{n+m}{m}$$

其中 $\binom{n+m}{m}$ 是我们较为熟悉的网格图路径计数的方案数. 所以我们可以考虑其组合意义的证明.

在一张网格图中, 从 $(0,0)$ 走到 (n,m) 共走 $n+m$ 步. 规定 $(0,0)$ 位于网格图左上角, 其中向下走了 n 步, 向右走了 m 步, 方案数为 $\binom{n+m}{m}$.

换个视角, 我们将 $n+m$ 步拆成两部分走, 先走 n 步, 再走 m 步, 那么 n 步中若有 i 步向右, 则 m 步中就有 $m-i$ 步向右, 故得证.

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:$Cat_n = \dfrac{C_{2n}^n}{n+1}$,$Cat_n = \dfrac{Cat_{n-1} * (4n-2)}{n+1}$.

斯特林数

第一类斯特林数

- **递推公式** $[n,m] = [n-1,m-1] + (n-1)\cdot[n-1,m]$ 从组合意义上讲即自己作为一个新的轮换或者自己插入到已有的人的左边
- **递推边界 $ = 1$**

第二类斯特林数

- **递推公式** $\{n,m\} = \{n-1,m-1\} + m\cdot\{n-1,m\}$ 从组合意义上讲即自己作为一个新的子集或者自己插入已有的子集中

- 常用公式 $x^n=\sum_{k=0}^{n}\{n,k\}(x)_k$ 常用化简 n^m
- 递推边界 $\{0,0\}=1$
- 计算公式 $\{n,m\}=\frac{1}{m!}\sum_{i=0}^{m}(-1)^{m-i}\binom{m}{i}i^n$

普通幂下降幂上升幂互化 $x^n=\sum_{k=0}^{n}\{n,k\}x^{\underline{k}}$ 普通幂转下降幂 $x^{\underline{n}}=\sum_{k=0}^{n}[n,k](-1)^{n-k}x^k$ 下降幂转普通幂 $x^n=\sum_{k=0}^{n}\{n,k\}(-1)^{n-k}x^{\overline{k}}$ 普通幂转上升幂 $x^{\overline{n}}=\sum_{k=0}^{n}[n,k]x^k$ 上升幂转普通幂

斯特林数恒等式

$$n!=\sum_{k=0}^{n}[n,k]$$

$$\sum_{k=1}^{n+1}[n,k]\{k,m\}=\binom{n}{m}$$

$$\sum_{k=1}^{n+1}\{n,k\}[k,m]=\binom{n}{m}$$

$$\{n,n-1\}=\binom{n}{2}$$

- **第二类斯特林数行即求出** $\{n,0\},\{n,1\},\dots,\{n,n\}$ 显然公式就是卷积形式 NTT 即可
- **第二类斯特林数列即** $\{0,k\},\{1,k\},\dots,\{n,k\}$ 我们考虑设出其生成函数然后递推根据递推式我们有 $H(n)=\sum_i(\{i-1,k-1\}+k\{i-1,k\})x^i$ 那么有 $H(n)=\frac{xH(n-1)}{1-kx}$ 也即 $H(n)=\frac{x^k}{\prod_{i=0}^{k}(1-ix)}$ 十分需要注意的是我们需要在分治以后把多项式的 $size$ 重置为 k (开小会 RE) 之后再求逆

- **第二类斯特林数求和即** $\sum_{i=0}^{n}\sum_{j=0}^{i}\{i,j\}$ 考虑按公式展开并交换求和次序即 $\sum_{j=0}^{n}\frac{1}{j!}\sum_{k=0}^{j}(-1)^{j-k}\binom{j}{k}\sum_{i=0}^{n}k^i$ 发现后面是个卷积形式然后 $O(n\log n)$ 求得 $O(n)$ 遍历即可

莫茨金数

$M_n = M_{n-1} + \sum_{i=0}^{n-2} M_i M_{n-2-i} = \frac{(2n+1)M_{n-1}+3(n-1)M_{n-2}}{n+2}$
$M_n = \sum_{k=0}^{\lfloor n/2\rfloor}\binom{n}{2k}C_k$ (C 为卡特兰数) 1, 1, 2, 4, 9, 21, 51, 127, 323, 835, ...

经典应用场景 1. 一个圆上有 n 个点, 画若干条不相交弦的方案数 2. 在一个二维网格从 $(0,0)$ 走到 $(n,0)$,k_1 表示向右上走一步,k_0 表示向右走一步,k_{-1} 表示向右下走一步要求路径中始终不能低于 x 轴, 终点必须在 $(n,0)$, 问方案数.

球盒模型

n 个球全部放入 m 个盒子

I: 球互不相同, 盒子互不相同 每个球都有 m 种选择, 根据乘法原理, 答案是 m^n

II: 球互不相同, 盒子互不相同, 每个盒子至多装一个球 $n > m$, 放不下, 0 种可能。$n \le m$, $A(m,n)$。

III: 球互不相同, 盒子互不相同, 每个盒子至少装一个球 容斥枚举空盒个数:

$$\sum_{i=0}^{m}(-1)^i \times C(m,i) \times (m-i)^n$$

或第二类斯特林数乘上 m 的阶乘 $m!\cdot$ `Stirling2`(n,m) ，答案为 $dp[n][m]\times m!$。

IV: 球互不相同, 盒子全部相同 枚举几个盒子有球, 第二类斯特林数, 设 $f[k][b]$ 为将 k 个互异元素分为 b 个不为空的集合。

$$ans=\sum_{i=0}^{m}f[n][i]$$

V: 球互不相同, 盒子全部相同, 每个盒子至多装一个球 $n>m,0\ n\leq m,1$

VI: 球互不相同, 盒子全部相同, 每个盒子至少装一个球 正是第二类斯特林数的定义, 答案是 $f[n][m]$。

$$dp[n][m]=\begin{cases}m\times dp[n-1][m]+dp[n-1][m-1] & 1\leq m<n\\ 1 & 0\leq n=m\\ 0 & m=0\text{ 且}1\leq n\end{cases}$$

VII: 球全部相同, 盒子互不相同 插板法, $C(n+m-1,m-1)$。

VIII: 球全部相同, 盒子互不相同, 每个盒子至多装一个球 选 n 个盒子放球, $C(m,n)$。

IX: 球全部相同, 盒子互不相同, 每个盒子至少装一个球 先把每个盒子放一个球, 然后转化为第 VII 个问题, 插板法, $C(n-1,m-1)$。

X: 球全部相同, 盒子全部相同 解 1:

问题等价于将 $n+m$ 拆分为 m 个无序的正整数, 根据 Ferrers 图的理论, 等价于将 $n+m$ 拆分成若干个不超过 m 的正整数, 直接生成函数做. $T(n,m)=T(n,m-1)+T(n-m,m)$

$$[x^{n+m-m}]\prod_{i=1}^{m}\frac{1}{1-x^i}$$

解 2:

$\frac{1}{(1-x)(1-x^2)\ldots(1-x^m)}$ 的 x^n 项的系数. 动态规划, 答案为

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

XI: 球全部相同, 盒子全部相同, 每个盒子至多装一个球 同 V

XII: 球全部相同, 盒子全部相同, 每个盒子至少装一个球 解 1:

$$[x^{n-m}]\prod_{i=1}^{m}\frac{1}{1-x^i}$$

解 2:

$\frac{x^m}{(1-x)(1-x^2)\ldots(1-x^m)}$ 的 x^n 项的系数. 动态规划, 答案为

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

```
#include <algorithm>
#include <cstdio>
int n, m, fac, minv;
```

```
int const mod = 998244353, g = 3, gi = (mod + 1) / g;
int C(int x, int y)
{
    if (x < 0 || y < 0 || x < y)
        return 0;
    else
        return 1ll * fac[x] * minv[y] % mod * minv[x - y] % mod;
}
int pow(int x, int y)
{
    int res = 1;
    while (y) {
        if (y & 1)
            res = 1ll * res * x % mod;
        x = 1ll * x * x % mod;
        y >>= 1;
    }
    return res;
}
struct NTT {
    int r, lim;
    NTT()
        : r()
        , lim()
    {
    }
    void getr(int lm)
    {
        lim = lm;
        for (int i = 0; i < lim; i++)
            r[i] = (r[i >> 1] >> 1) | ((i & 1) * (lim >> 1));
    }
    void operator()(int* a, int type)
```

```
    {
        for (int i = 0; i < lim; i++)
            if (i < r[i])
                std::swap(a[i], a[r[i]]);
        for (int mid = 1; mid < lim; mid <<= 1) {
            int rt = pow(type == 1 ? g : gi, (mod - 1) / (mid << 1));
            for (int j = 0, r = mid << 1; j < lim; j += r) {
                int p = 1;
                for (int k = 0; k < mid; k++, p = 1ll * p * rt % mod) {
                    int x = a[j + k], y = 1ll * a[j + mid + k] * p % mod;
                    a[j + k] = (x + y) % mod, a[j + mid + k] = (x - y + mod) % mod;
                }
            }
        }
        if (type == -1)
            for (int i = 0, p = pow(lim, mod - 2); i < lim; i++)
                a[i] = 1ll * a[i] * p % mod;
    }
} ntt;
void inv(int const* a, int* ans, int n)
{
    static int tmp;
    for (int i = 0; i < n << 1; i++)
        tmp[i] = ans[i] = 0;
    ans = pow(a, mod - 2);
    for (int m = 2; m <= n; m <<= 1) {
        int lim = m << 1;
        ntt.getr(lim);
        for (int i = 0; i < m; i++)
            tmp[i] = a[i];
        ntt(tmp, 1), ntt(ans, 1);
        for (int i = 0; i < lim; i++)
            ans[i] = ans[i] * (2 - 1ll * ans[i] * tmp[i] % mod + mod) % mod, tmp[i] = 0
```

```
        ntt(ans, -1);
        for (int i = m; i < lim; i++)
            ans[i] = 0;
    }
}
void inte(int const* a, int* ans, int n)
{
    for (int i = n - 1; i; i--)
        ans[i] = 1ll * a[i - 1] * pow(i, mod - 2) % mod;
    ans = 0;
}
void der(int const* a, int* ans, int n)
{
    for (int i = 1; i < n; i++)
        ans[i - 1] = 1ll * i * a[i] % mod;
    ans[n - 1] = 0;
}
void ln(int const* a, int* ans, int n)
{
    static int b;
    for (int i = 0; i < n << 1; i++)
        ans[i] = b[i] = 0;
    inv(a, ans, n);
    der(a, b, n);
    int lim = n << 1;
    ntt.getr(lim);
    ntt(b, 1), ntt(ans, 1);
    for (int i = 0; i < lim; i++)
        b[i] = 1ll * ans[i] * b[i] % mod, ans[i] = 0;
    ntt(b, -1);
    for (int i = n; i < lim; i++)
        b[i] = 0;
    inte(b, ans, n);
```

```
}
void exp(int const* a, int* ans, int n)
{
    static int f;
    for (int i = 0; i < n << 1; i++)
        ans[i] = f[i] = 0;
    ans = 1;
    for (int m = 2; m <= n; m <<= 1) {
        int lim = m << 1;
        ln(ans, f, m);
        f = (a + 1 - f + mod) % mod;
        for (int i = 1; i < m; i++)
            f[i] = (a[i] - f[i] + mod) % mod;
        ntt.getr(lim);
        ntt(f, 1), ntt(ans, 1);
        for (int i = 0; i < lim; i++)
            ans[i] = 1ll * ans[i] * f[i] % mod, f[i] = 0;
        ntt(ans, -1);
        for (int i = m; i < lim; i++)
            ans[i] = 0;
    }
}
void solve1() { printf("%d\n", pow(m, n)); }
void solve2()
{
    if (m < n)
        puts("0");
    else
        printf("%lld\n", 1ll * fac[m] * minv[m - n] % mod);
}
void solve3()
{
    if (n < m)
```

```
        return puts("0"), void();
    int ans = 0;
    for (int i = 0; i <= m; i++)
        ans = (ans + 1ll * pow(mod - 1, i) * C(m, i) % mod * pow(m - i, n)) % mod;
    printf("%d\n", ans);
}
int s;
void solve4()
{
    static int tmp;
    for (int i = 0; i <= n; i++)
        tmp[i] = (i & 1 ? mod - 1ll : 1ll) * minv[i] % mod, s[i] = 1ll * pow(i, n) * mi
    int lim = 1;
    for (lim = 1; lim <= n + n; lim <<= 1)
        ;
    ntt.getr(lim);
    ntt(tmp, 1), ntt(s, 1);
    for (int i = 0; i < lim; i++)
        s[i] = 1ll * s[i] * tmp[i] % mod;
    ntt(s, -1);
    for (int i = n + 1; i < lim; i++)
        s[i] = 0;
    int ans = 0;
    for (int i = 0; i <= m; i++)
        ans = (ans + s[i]) % mod;
    printf("%d\n", ans);
}
void solve5() { printf("%d\n", int(m >= n)); }
void solve6() { printf("%d\n", s[m]); }
void solve7() { printf("%d\n", C(n + m - 1, m - 1)); }
void solve8() { printf("%d\n", C(m, n)); }
void solve9() { printf("%d\n", C(n - 1, m - 1)); }
int ans;
```

```
void solve10()
{
    static int tmp;
    for (int i = 1; i <= m; i++)
        for (int j = 1; j * i <= n; j++)
            ans[i * j] = (ans[i * j] - 1ll * minv[j] * fac[j - 1] % mod + mod) % mod;
    int lim = 1;
    for (; lim <= n; lim <<= 1)
        ;

    exp(ans, tmp, lim);
    for (int i = 0; i < lim; i++)
        ans[i] = 0;
    inv(tmp, ans, lim);
    printf("%d\n", ans[n]);
}
void solve11() { printf("%d\n", int(m >= n)); }
void solve12()
{
    printf("%d\n", n - m >= 0 ? ans[n - m] : 0);
}
int main()
{
    scanf("%d%d", &n, &m);
    fac = 1;
    for (int i = 1; i <= n + m; i++)
        fac[i] = 1ll * fac[i - 1] * i % mod;
    minv[n + m] = pow(fac[n + m], mod - 2);
    for (int i = n + m; i; i--)
        minv[i - 1] = 1ll * minv[i] * i % mod;
    solve1();
    solve2();
    solve3();
```

```
    solve4();
    solve5();
    solve6();
    solve7();
    solve8();
    solve9();
    solve10();
    solve11();
    solve12();
    return 0;
}
```

容斥原理

> 定义:$\big|S_1 \cup S_2 \cup S_3 \cup \cup S_n \big| =\sum_{i=1}^N|S_i| - \sum

例题:给定一个整数 n 和 m 个不同的质数 $p_1, p_2, ..., p_m$,请你求出 1 n 中

二进制枚举解

```cpp
int main(){
    ios::sync_with_stdio(false);cin.tie(0);
    LL n, m;
    cin >> n >> m;
    vector <LL> p(m);
    for (int i = 0; i < m; i ++ )
        cin >> p[i];
    LL ans = 0;
    for (int i = 1; i < (1 << m); i ++ ){
        LL t = 1, cnt = 0;
        for (int j = 0; j < m; j ++ ){
```

```
            if (i >> j & 1){
                cnt ++ ;
                t *= p[j];
                if (t > n){
                    t = -1;
                    break;
                }
            }
        }
        if (t != -1){
            if (cnt & 1) ans += n / t;
            else ans -= n / t;
        }
    }
    cout << ans << "\n";
    return 0;
}
```

dfs 解

```
int main(){
    ios::sync_with_stdio(false);cin.tie(0);
    LL n, m;
    cin >> n >> m;
    vector <LL> p(m);
    for (int i = 0; i < m; i ++ )
        cin >> p[i];
    LL ans = 0;
    function<void(LL, LL, LL)> dfs = [&](LL x, LL s, LL odd){
        if (x == m){
            if (s == 1) return;
            ans += odd * (n / s);
            return;
```

```
        }
        dfs(x + 1, s, odd);
        if (s <= n / p[x]) dfs(x + 1, s * p[x], -odd);
    };
    dfs(0, 1, -1);
    cout << ans << "\n";
    return 0;
}
```

群论计数

Burnside 引理 给定群 G 在集合 X 上的作用, 则所有不同的轨道的数目

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

这里,$X^g = \{x \in X : gx = x\}$ 是 $g \in G$ 的作用下的不动点集合.

Polya 引理 给定群 G 在集合 X 上的作用和颜色集合 C, 则不同的染色方案的数目

$$|C^X/G| = \frac{1}{|G|}\sum_{g\in G} m^{c(g)},$$

这里,m 是颜色数目,$c(g)$ 是元素 $g \in G$ 的置换表示的轮换分解中的轮换数目.

带权重形式的推广 无权重版本的 Plya 计数原理只能够给出所有的本质不同的染色问题的计数, 但是在处理更为精细的问题时就无能为力了. 比如说, 如果在上述染色问题中, 给定每种可以使用的颜色的数目, 就不能套用上面的 Plya 计数公式. 在实际求解这类问题时, 需要再次使用 Burnside 引理加

以推导; 而将这些结果总结为生成函数的形式, 就是带权重版本的 Plya 计数原理.

例子

项链染色 (带限制) 现在有一串共四个珠子的项链, 每个珠子可以是红色或者蓝色, 恰有两个红色珠子, 两个蓝色珠子可以使用, 计算共有几种本质不同的珠子.(如果两种染色的结果可以通过旋转项链重合, 就认为是相同的.)

解答和分析 考虑使用 Burnside 引理. 红色, 蓝色珠子各两个, 共计有 $\binom{4}{2} = 6$ 种染色方案. 空间对称群 $G = \{r_0, r_1, r_2, r_3\}$ 分别对应旋转 $0 \sim 3$ 次, 则它们对应的不动点集合分析如下:

- 旋转零次 $r_0 = (1)$, 全部 6 个染色方案都是不动点;
- 旋转一次 $r_1 = (1234)$, 不动点要求所有珠子染同样的颜色, 没有不动点;
- 旋转两次 $r_2 = (13)(24)$, 有两个可独立染色的区域, 大小都是 2, 它们要分别染成红色和蓝色, 则不动点集合的大小为 2;
- 旋转三次 $r_3 = (1432)$, 与旋转一次的情形相同, 没有不动点.

所以, 根据 Burnside 引理, 本质不同的染色数目为

$$\frac{6+0+2+0}{4} = 2.$$

从这个例子中可以总结出如下计算方法. 对于限制不同颜色个数的问题, 同样是要把空间对称群中各个置换的轮换分别染色, 但是需要让染色用到的颜色数目恰好等于给定的颜色个数. 这样的组合问题通常没有显式解, 除了可以通过 排列组合方法 计算的特殊情形外, 需要看做 背包问题 进行求解.

通过生成函数可以给出这类计数问题的答案. 给定置换 g, 如果它的 型 是 $1^{\alpha_1}2^{\alpha_2}\cdots n^{\alpha_n}$, 即它有 α_k 个长度为 k 的轮换, 且对于每个轮换可以染成 m 种颜色中的一种, 那么生成函数

$$\prod_{k=1}^{n}\left(\sum_{i=1}^{m}x_i^k\right)^{\alpha_k}$$

中单项式 $x_1^{\beta_1}x_2^{\beta_2}\cdots x_m^{\beta_m}$ 的系数就是第 i 种颜色用了 β_i 次的计数. 这里圆括号中的表达式 $\sum_{i=1}^{m}x_i^k$ 的组合意义是, 对于长度为 k 的轮换, 用到 k 次颜色 i 的染色方法的计数是 1, 对于其它情形, 计数是 0; 这正描述了同一轮换中各位置染色一致的要求.

给定置换 g 下染色计数的生成函数, 对各个单项式应用 Burnside 引理, 就得到各种颜色组合下的本质不同的计数. 因为生成函数对各个单项式是线性的, 所以本质不同染色方案的计数的生成函数是

$$\frac{1}{|G|}\sum_{g\in G}\prod_{k=1}^{n}\left(\sum_{i=1}^{m}x_i^k\right)^{\alpha_k}.$$

展开这个式子, 每个单项式的系数就给出了给定颜色组合下的本质不同染色的计数.

在上述过程中, 对每个轮换进行染色的生成函数 $\sum_{i=1}^{m}x_i^k$ 并无特殊之处, 可以替换成其它的生成函数. 因而, 有如下的一般版本的 Plya 计数原理.

置换群的轮换指标 给定置换群 G, 则群 G 的 **轮换指标** (cycle index), 定义为

$$Z_G(t_1,t_2,\cdots,t_n)=\frac{1}{|G|}\sum_{g\in G}t_1^{c_1(g)}t_2^{c_2(g)}\cdots t_n^{c_n(g)},$$

其中,$c_k(g)$ 是置换 g 的轮换分解中长度为 k 的轮换的个数, 即 $1^{c_1(g)}2^{c_2(g)}\cdots n^{c_n(g)}$ 是置换 g 的型.

Plya 计数原理 (带权重版本) 给定群 G 在集合 X 上的作用, 对每个点的染色方法由它的染色方案的计数的生成函数 $f(x_1, x_2, \cdots, x_m)$ 给出, 那么集合 X 的本质不同染色方案的计数的生成函数是

$$Z_G(f(x_1^1, x_2^1, \cdots, x_m^1), f(x_1^2, x_2^2, \cdots, x_m^2), \cdots, f(x_1^n, x_2^n, \cdots, x_m^n)),$$

这里,$Z_G(t_1, t_2, \cdots, t_n)$ 是群 G 的轮换指标.

这里, 如果单个位置的染色的生成函数是 $f(x_1, x_2, \cdots, x_m)$, 那么长度为 k 的轮换的染色的生成函数就是 $f(x_1^k, x_2^k, \cdots, x_m^k)$. 这反映了如果某一染色方案是给定置换的不动点, 那么同一轮换中的所有位置必须染相同的颜色. 如果将生成函数在 $x_i = 1$ 处取值, 就得到上文的无权重版本的 Plya 计数原理.

定理的叙述用到了置换群的轮换指标的概念. 它和具体的染色问题无关. 它描述了置换群的结构.

康拓展开

正向展开普通解法 将一个字典序排列转换成序号. 例如:12345->1,12354->2.

```
int f[20];
void jie_cheng(int n) { // 打出 1-n 的阶乘表
    f[0] = f[1] = 1; // 0 的阶乘为 1
    for (int i = 2; i <= n; i++) f[i] = f[i - 1] * i;
}
string str;
int kangtuo() {
    int ans = 1; // 注意，因为 12345 是算作 0 开始计算的，最后结果要把 12345 看作
    int len = str.length();
    for (int i = 0; i < len; i++) {
        int tmp = 0; // 用来计数的
```

```
        // 计算 str[i] 是第几大的数，或者说计算有几个比他小的数
        for (int j = i + 1; j < len; j++)
            if (str[i] > str[j]) tmp++;
        ans += tmp * f[len - i - 1];
    }
    return ans;
}
int main() {
    jie_cheng(10);
    string str = "52413";
    cout << kangtuo() << endl;
}
```

正向展开树状数组解 给定一个全排列, 求出它是 1 ~ n 所有全排列的第几个, 答案对 998244353 取模.

答案就是 $\sum_{i=1}^{n} res_{a_i}(n-i)!$ .res_x 表示剩下的比 x 小的数字的数量, 通过**树状数组**处理.

```
#include <bits/stdc++.h>
using namespace std;
#define LL long long
const int mod = 998244353, N = 1e6 + 10;
LL fact[N];
struct fwt{
    LL n;
    vector <LL> a;
    fwt(LL n) : n(n), a(n + 1) {}
    LL sum(LL x){
        LL res = 0;
        for (; x; x -= x & -x)
            res += a[x];
        return res;
    }
```

```
    void add(LL x, LL k){
        for (; x <= n; x += x & -x)
            a[x] += k;
    }
    LL query(LL x, LL y){
        return sum(y) - sum(x - 1);
    }
};
int main(){
    ios::sync_with_stdio(false);cin.tie(0);
    LL n;
    cin >> n;
    fwt a(n);
    fact[0] = 1;
    for (int i = 1; i <= n; i ++ ){
        fact[i] = fact[i - 1] * i % mod;
        a.add(i, 1);
    }
    LL ans = 0;
    for (int i = 1; i <= n; i ++ ){
        LL x;
        cin >> x;
        ans = (ans + a.query(1, x - 1) * fact[n - i] % mod ) % mod;
        a.add(x, -1);
    }
    cout << (ans + 1) % mod << "\n";
    return 0;
}
```

逆向还原

```
string str;
int kangtuo(){
```

```
    int ans = 1;  //注意，因为 12345 是算作 0 开始计算的，最后结果要把 12345 看作
    int len = str.length();
    for(int i = 0; i < len; i++){
        int tmp = 0;//用来计数的
        for(int j = i + 1; j < len; j++){
            if(str[i] > str[j]) tmp++;
            //计算 str[i] 是第几大的数，或者说计算有几个比他小的数
        }
        ans += tmp * f[len - i - 1];
    }
    return ans;
}
int main(){
    jie_cheng(10);
    string str = "52413";
    cout<<kangtuo()<<endl;
}
```

基本格路计数问题

问题描述: 在平面上有多少从 $(0,0)$ 到 $(m,n) \in N \times N$ 点的格路经, 其每一步都具有形式 $(1,0)$ 或 $(0,1)$(即每一步沿水平方向向右走或沿铅直方向向上走一个单位距离).

推导与结论: 从 $(0,0)$ 到 (m,n) 的路径, 记沿水平方向向右走一个单位距离为 E, 记沿竖直方向向上走一个单位距离为 N. 其与多重集 $\{m \cdot E, n \cdot N\}$ 的排列一一对应, 一条路径对应该多重集上的一个全排列. 所以共有:

$$\frac{(m+n)!}{m!n!} = \binom{m+n}{m}$$

种不同的走法.

Dyck 路计数问题

格路模型是信息学竞赛中一种常见的问题模型, 其中较为常见的一种是**Dyck 路计数**的相关问题.

定义

- **格路 (Lattice Path):** 在平面直角坐标系中, 横坐标和纵坐标都是整数的点称为格点, 平面格路是指从一个格点到另一格点只走格点的路, 格路的**长度**是指其所走的路的步数.
- **自由路 (Free Path):** 对于一条从 $(0,0)$ 到 (m,n) 的格路, 若其只使用了上步 $U=(0,1)$, 水平步 $L=(1,0)$, 则我们称其为 (m,n) **自由路**.
- **自由路计数:** 记 $F(n,m)$ 为 (n,m) 自由路的全集,$F(n,m)=\#F(n,m)$ 为 (n,m) 自由路的总个数, 即 $F(n,m)$ 的元素个数. 可以发现, 一条 $(0,0)$ 到 (n,m) 的自由路可以唯一地对应到一个含有 n 个水平步 L 和 m 个上步 U 的序列. 因此,(n,m) 自由路的条数等于从 $n+m$ 个位置中选出 n 个位置的方案数, 即 $\binom{n+m}{n}$.

$$F(n,m)=\binom{n+m}{n}$$

- **(n,m)-Dyck 路:** 对于一条从 $(0,0)$ 到 (n,m) 的自由路, 若其始终不经过对角线 $y=\frac{m}{n}x$ 下方, 则我们称之为 (n,m)**-Dyck 路**. 记 $D(n,m)$ 为 (n,m)-Dyck 路的集合,$D(n,m)=\#D(n,m)$ 为 (n,m)-Dyck 路的数量.
- **t-Dyck 路:** 特别地, 若 $m=t\cdot n$, 则我们称之为 n 阶 **t-Dyck 路**.

(n,m)-Dyck 路的计数　定理 2.2 (互质情况): 当 n,m 互质时,(n,m)-Dyck 路的数量为:

$$D(n,m)=\frac{1}{n+m}\binom{n+m}{n}$$

证明思路: 利用 **Raney 引理**的推广或通过更复杂的双射构造. 一个关键引理是, 当 n,m 互质时, 对任意一个 (n,m) 自由路, 通过循环移位 (将其看作

一个环), 恰好只有一个循环同构的路径是 (n,m)-Dyck 路. 总共有 $n+m$ 种循环移位, 因此 Dyck 路占总数的 $\frac{1}{n+m}$.

有 k 个峰的 (n,m)-Dyck 路计数

- **峰 (peak):** 对于一条从 $(0,0)$ 到 (n,m) 的自由路中的连续两步, 若其为 UL, 则我们称之为**峰**.
- **谷 (valley):** 若其为 LU, 则我们称之为一个**谷**.

定理 2.3: 记 $F(n,m;k)$ 为所有有恰好 k 个峰的 (n,m) 自由路的集合. 则:

$$F(n,m;k)=\binom{n}{k}\binom{m}{k}$$

证明: 将 UL 之间的格点叫做峰点. 一个有 k 个峰的路径, 其峰点坐标为 $(x_1,y_1),\dots,(x_k,y_k)$, 满足 $0\le x_1<x_2<\cdots<x_k\le n-1$ 和 $1\le y_1<y_2<\cdots<y_k\le m$. 选择 k 个 x 坐标和 k 个 y 坐标可以唯一确定一条有 k 个峰的路径, 反之亦然. 因此方案数为 $\binom{n}{k}\binom{m}{k}$.

定理 2.4: 记 $F^{UL}(n,m;k)$ 为所有有恰好 k 个峰, 且首步为 U, 末步为 L 的 (n,m) 自由路的集合. 则:

$$F^{UL}(n,m;k)=\binom{n-1}{k-1}\binom{m-1}{k-1}$$

定理 2.5: 记 $D(n,m;k)$ 为所有有恰好 k 个峰的 (n,m)-Dyck 路的集合. 当 n,m 互质时:

$$D(n,m;k)=\frac{1}{k}\binom{n-1}{k-1}\binom{m-1}{k-1}$$

证明思路: 对于任意一条 $F^{UL}(n,m;k)$ 中的路径, 其 k 个峰点可以循环映射到 $k-1$ 条不同的不合法路径上. 因此, 恰好有 $\frac{1}{k}$ 的路径是 Dyck 路.

定理 2.6 (t-Dyck 路): 有 k 个峰的 n 阶 t-Dyck 路的个数是:

$$D(n,tn;k)=\frac{1}{n}\binom{n}{k}\binom{tn}{k-1}$$

将 k 从 1 到 n 求和, 可以得到 n 阶 t-**Dyck 路的总数**:

$$D(n,tn)=\frac{1}{tn+1}\binom{tn+n}{n}$$

定理 2.8: 从 $(0,0)$ 到 (n,m) 的有 k 个峰的 t-Dyck 路 ($m=tn$) 的个数是:

$$D_t(n,m;k)=\frac{m-tn+1}{n}\binom{n}{k}\binom{m}{k-1}$$

定理 2.9: 从 $(0,0)$ 到 (n,m) 的 t-Dyck 路的总个数是:

$$D_t(n,m)=\frac{m-tn+1}{n+m+1}\binom{n+m+1}{n}$$

不相交格路问题

n 阶不交 Dyck 路计数　定义 3.1: 从 $(0,0)$ 到 (n,n) 的两条 Dyck 路 P,Q. 若 Q 始终不穿过 P (即 Q 在 P 的下方或与之重合), 则称 (P,Q) 是一对**不交 Dyck 路**.

定理 3.1: n 阶不交 Dyck 路对数为:

$$C*n+2-C*n+1^2$$

证明思路: 通过构造一个巧妙的双射 θ, 将一对不交的 Dyck 路 (P,Q) 映射到从 $(0,0)$ 到 $(n+1,n+1)$ 的一条 Dyck 路和从 $(1,1)$ 到 $(n+2,n+2)$ 的一条 Dyck 路. 利用这个双射关系, 结合卡特兰数的性质进行推导.

不交自由路计数　定义 3.2: 设 P,Q 是两条自由路, 如果 Q 始终不穿越 P, 则称 (P,Q) 是从 $(0,0)$ 到 (n,m) 的两条**不相交自由路对**.$F_{nc}(n,m)$ 表示不相交自由路对的个数.

定理 3.3: 从 $(0,0)$ 到 (n,m) 的不接触自由路对 (除了起点和终点) 的个数为:

$$F*nt(n,m)=\frac{1}{n+m-1}\binom{n+m-2}{n-1}\binom{n+m-2}{n-1}$$

定理 3.4: 从 $(0,0)$ 到 (n,m) 的不交自由路对个数是:

$$F*nc(n,m)=\frac{1}{n+m+1}\binom{n+m+1}{n}\binom{n+m+1}{n}$$

证明思路: 类似于不交 Dyck 路, 通过双射将问题转化. 将两条不交的 (n,m) 自由路 (P,Q) 构造成一条从 $(0,0)$ 到 $(n+1,m+1)$ 的不接触自由路对, 从而建立数量关系.

类 Dyck 路计数问题 (斜向行走)

从 $(0,0)$ 走到 (a,b), 规定每次只能从 (x,y) 走到左下或者右下, 方案数记为 $f(a,b)$.

- $f(a,b)=\binom{a}{\frac{a+b}{2}}$;
- 若路径和直线 $y=k,k\notin[0,b]$ 不能有交点, 则方案数为 $f(a,b)-f(a,2k-b)$;
- 若路径和两条直线 $y=k_1,y=k_2(k_1<0\le b<k_2)$ 不能有交点, 方案数记为 $g(a,b,k_1,k_2)$, 可以使用 $\mathcal{O}(N)$ 递归求解;
- 若路径必须碰到 $y=k_1$ 但是不能碰到 $y=k_2$, 方案数记为 $h(a,b,k_1,k_2)$, 可以使用 $\mathcal{O}(N)$ 递归求解 (递归过程中两条直线距离会越来越大).

从 $(0,0)$ 走到 $(a,0)$, 规定每次只能走到左下或者右下, 且必须有**恰好一次**传送 (向下 b 单位), 且不能走到 x 轴下方, 方案数为 $\binom{a+1}{\frac{a-b}{2}+k+1}$.

格路计数与经典分拆恒等式

格路计数方法可以用于建立和证明关于基本超几何函数和整数分拆的恒等式.

枚举函数与 q-升阶乘

- **q-升阶乘符号:** $(x;q)_n = \prod_{k=0}^{n-1}(1-xq^k)$
- 引理 **1:**
  - i) 纵向步骤 V 起始的格路枚举函数为 $\prod_{i=1}^{m}(1-tq^{i-1})^{-1}$.
  - ii) 纵向步骤 V 起始且终点高度为 n 的格路枚举函数为 $t^n/(q;q)_n$.
  - iii) 具有不等水平矢量表示的纵向步骤 V 起始的格路枚举函数为 $(-tq;q)_\infty$.

Gauss 二项式系数　定义: $\begin{bmatrix} n \\ k \end{bmatrix}_q = \frac{(q;q)_n}{(q;q)_k(q;q)_{n-k}}$

命题 3: 终点坐标为 (m,n) 的格路枚举函数为 Gauss 二项式系数 $\begin{bmatrix} m+n \\ n \end{bmatrix}_q$.

Durfee 矩与经典分拆恒等式　通过将格路按其所包容的最大 $(k+r)\times k$ **Durfee 矩形**进行分类, 可以推导出复杂的组合恒等式.

- 定理 **5:**

$$(tq)_\infty^{-1} = \sum_{k=0}^{m} \begin{bmatrix} m-r \\ k \end{bmatrix} t^k q^{k(k+r)}/(tq)\_k + r$$

- 系理 **6 (Kummer 定理的有限形式):**
  - i) $(tq)_\infty^{-1} = \sum_{k=0}^{m} \begin{bmatrix} m \\ k \end{bmatrix} t^k q^k/(tq)_k$
  - ii) $(tq)_\infty^{-1} = \sum_{k=0}^{\infty} t^k q^{k(k+r)}/(q)_k(tq)_{k+r}$

通过对这些公式取极限或特定值, 可以得到 **Euler 五角数定理**,**Cauchy 恒等式**,**Jacobi 三重积恒等式**等著名结论.

Euler 五角数定理的极限形式为:

$$(q)_\infty = \sum_{k=-\infty}^{\infty} (-1)^k q^{k(3k+1)/2}$$

格路计数的常用结论

反射原理 (Reflection Principle) 反射原理是一种巧妙的组合计数技巧, 用于处理带有**边界限制**的格路计数问题. 其核心思想如下:

1. **转化问题:** 将直接求解“合法路径数的难题, 转化为求解”总路径数减去” 不合法路径数. 总路径数通常很容易计算.
2. **建立一一对应:** 将**每一条不合法的路径** (即触碰或穿越了限制边界的路径), 通过一次**轴对称变换 (反射)**, **唯一地**映射到另一条从**新的起点**出发到**原终点**的无限制路径.
3. **计数替换:** 由于不合法路径与新起点路径之间存在一一对应关系, 因此“不合法路径的数量就等于”从新起点到原终点的路径总数. 后者是一个无限制的格路问题, 可以直接用组合数公式计算.

简单来说, 就是用**反射起点的无限制路径数**来等价替换**原起点的非法路径数**.

图示: 一条从 A 到 B 的不合法路径 (红色), 在第一个接触点 P 处, 将 A 到 P 段关于直线 L 反射, 得到一条从反射点 A’ 到 B 的路径 (绿色).

模型一: 标准格路模型

- **移动方式:** 每步只能向右 $(+1, 0)$ 或向上 $(0, +1)$.
- **总路径数:** 从 (a, b) 到 (c, d) 的总路径数为 $\binom{(c-a)+(d-b)}{c-a}$.

定理 1.1: 单条边界线 (不能触碰)

- **问题:** 从 (a, b) 到 (c, d), 路径不能触碰直线 $L : y = x + k$.
- **前提:** 起点和终点都在直线 L 的同一侧 (通常是下方).
- **反射起点:** 将起点 (a, b) 关于直线 $L : y = x + k$ 反射, 得到新起点 $A'(b - k, a + k)$.
- **公式:**

$$\text{合法路径数} = \binom{(c-a)+(d-b)}{c-a} - \binom{(c-(b-k))+(d-(a+k))}{c-(b-k)}$$

即: **(从原起点到终点的总路径数) - (从反射起点到终点的总路径数)**

定理 1.2: 卡特兰数 (Catalan Number)

- **问题:** 从 $(0,0)$ 到 (n,n), 路径不能越过对角线 $y=x$(即不能到达 $y=x$ 上方).
- **等价问题:** 路径不能触碰直线 $y=x+1$.
- **公式:**

$$C_n = \binom{2n}{n} - \binom{2n}{n+1} = \frac{1}{n+1}\binom{2n}{n}$$

模型二: 斜向格路模型

- **移动方式:** 每步只能右上 $(+1,+1)$ 或右下 $(+1,-1)$.
- **总路径数:** 从 $(0,0)$ 到 (a,b) 的总路径数为 $\binom{a}{\frac{a+b}{2}}$.(要求 $a+b$ 为偶数且 $a \geq |b|$)

定理 2.1: 单条水平边界线 (不能触碰)

- **问题:** 从 (a,b) 到 (c,d), 路径不能触碰水平直线 $L: y=k$.
- **前提:** 起点和终点都在直线 L 的同一侧.
- **反射终点:** 将终点 (c,d) 关于直线 $L: y=k$ 反射, 得到新终点 $B'(c,2k-d)$.
- **公式:**

$$\text{合法路径数} = \binom{c-a}{\frac{(c-a)+(d-b)}{2}} - \binom{c-a}{\frac{(c-a)+(2k-d-b)}{2}}$$

即: **(从起点到原终点的总路径数) - (从起点到反射终点的总路径数)**

说明:

- 在这两个模型中, 反射起点还是终点是等价的, 选择计算更方便的一个即可.

- **“不能越过”**和 **“不能触碰”**是有区别的. “不能越过 $y = x$” 意味着路径上的点 (x_p, y_p) 始终满足 $y_p \leq x_p$. 这等价于” 不能触碰 $y = x + 1$. 在比赛中要仔细审题.

杂项

1. 考虑不定方程 $X_1 + X_2 + \cdots + X_n = m, (X_i > 0)$ 的解的数量也就是把 m 分成 n 堆的方案数, 我们运用插空法为 $\binom{m-1}{n-1}$
2. 考虑不定方程 $X_1 + X_2 + \cdots + X_n = m, (X_i >= 0)$ 的解的数量我们把所有的 X_i 变为 $X_i + 1$ 那么套用结论 1, 方案数为 $C(m + n - 1, n - 1)$
3. 在 n 个红球中插入 m 个蓝球, 空格大小不限, 可以插入两边, 方案数为 $C(n + m, m)$ 证明: 考虑有 $n + 1$ 个空格可以插入, 那么问题转化为不定方程 $X_1 + \cdots + X_{n+1} = m, (X_i \geq 0)$ 的解的个数
4. 在 n 个红球中插入 m 个蓝球, 空格大小不限, 不可插入两边, 方案数为 $C(n + m - 2, m - 2)$ 证明: 考虑有 $n - 1$ 个空格可以插入, 那么问题转化为不定方程 $X_1 + \cdots + X_{n-1} = m, (X_i \geq 0)$ 的解的个数
5. 考虑在 $[1, n]$ 区间内选 k 个数, 满足 k 个数都不相邻问选法多少种考虑到不能相邻的原因, 我们不妨考虑把每个选的数字和它的下一个数字绑在一起, 特别地最后一个数字不用捆绑, 那么答案为 $\binom{n-k+1}{k}$