Standard Code Library

Your TeamName

Your School

October 14, 2025

Contents

一切的开始

宏定义

- 需要 C++11

```cpp
#include <bits/stdc++.h>
using namespace std;
using LL = long long;
#define FOR(i, x, y) for (decay<decltype(y)>::type i = (x), _##i = (y); i < _##i; ++i)
#define FORD(i, x, y) for (decay<decltype(x)>::type i = (x), _##i = (y); i > _##i; --i)
#ifdef DEBUG
#ifndef ONLINE_JUDGE
#define zerol
#endif
#endif
#ifdef zerol
#define dbg(x...) do { cout << "\033[32;1m" << #x << " -> "; err(x); } while (0)
void err() { cout << "\033[39;0m" << endl; }
template<template<typename...> class T, typename t, typename... A>
void err(T<t> a, A... x) { for (auto v: a) cout << v << ' '; err(x...); }
template<typename T, typename... A>
void err(T a, A... x) { cout << a << ' '; err(x...); }
#else
#define dbg(...)
#define err(...)
#endif
// -----------------------------------------------------------------------------
```

- 调试时添加编译选项 `-DDEBUG`，提交时注释
- 注意检查判题系统编译选项，修改 `#ifndef ONLINE_JUDGE`
- `FOR` ++ 循环 `FOR(循环变量名称, 循环变量起始值, 循环变量结束值（不含）)`
- `FORD` –循环
- `err()` 调试时输出（支持单层迭代）
- `dbg()` 变色输出变量名和变量值（支持单层迭代）
- 黄色 `33`，蓝色 `34`，橙色 `31`

对拍

- Linux

```bash
#!/usr/bin/env bash
g++ -o r main.cpp -O2 -std=c++11
g++ -o std std.cpp -O2 -std=c++11
while true; do
    python gen.py > in
    ./std < in > stdout
    ./r < in > out
    if test $? -ne 0; then
        exit 0
    fi
    if diff stdout out; then
        printf "AC\n"
    else
        printf "GG\n"
        exit 0
    fi
done
```

- Windows

```bat
@echo off
setlocal enabledelayedexpansion

g++ -o r main.cpp -O2 -std=c++11
g++ -o std std.cpp -O2 -std=c++11

:loop
python gen.py > in
if !errorlevel! neq 0 exit /b
```

```
std.exe < in > stdout
if !errorlevel! neq 0 exit /b

r.exe < in > out
if !errorlevel! neq 0 exit /b

fc /b stdout out > nul
if !errorlevel! equ 0 (
    echo AC
) else (
    echo GG
    exit /b
)

goto loop
```

快速编译运行（配合无插件 VSC）

- Linux

```bash
#!/bin/bash
g++ $1.cpp -o $1 -O2 -std=c++14 -Wall -Dzerol -g
if $? -eq 0; then
    ./$1
fi
```

- Windows

```
@echo off
:: 参数为文件名（不含.cpp后缀）
g++ %1.cpp -o %1 -O2 -std=c++14 -Wall -Dzerol -g
if %errorlevel% equ 0 (
    %1.exe
)
```

数据结构

ST 表

- 一维

```cpp
#define M 10

struct RMQ {
    int f[22][M];
    inline int highbit(int x) { return 31 - __builtin_clz(x); }
    void init(int* v, int n) {
        FOR (i, 0, n) f[0][i] = v[i];
        FOR (x, 1, highbit(n) + 1)
        FOR (i, 0, n - (1 << x) + 1)
        f[x][i] = min(f[x - 1][i], f[x - 1][i + (1 << (x - 1))]);
    }
    int get_min(int l, int r) {
        assert(l <= r);
        int t = highbit(r - l + 1);
        return min(f[t][l], f[t][r - (1 << t) + 1]);
    }
};
```

- 二维

```cpp
#define maxn 10
LL n, m, a[maxn][maxn];

struct RMQ2D{
    int f[maxn][maxn][10][10];
    inline int highbit(int x) { return 31 - __builtin_clz(x); }
```

```cpp
    inline int calc(int x, int y, int xx, int yy, int p, int q) {
        return max(
            max(f[x][y][p][q], f[xx - (1 << p) + 1][yy - (1 << q) + 1][p][q]),
            max(f[xx - (1 << p) + 1][y][p][q], f[x][yy - (1 << q) + 1][p][q])
            );
    }
    void init() {
        FOR (x, 0, highbit(n) + 1)
        FOR (y, 0, highbit(m) + 1)
        FOR (i, 0, n - (1 << x) + 1)
        FOR (j, 0, m - (1 << y) + 1) {
            if (!x && !y) { f[i][j][x][y] = a[i][j]; continue; }
            f[i][j][x][y] = calc(
                i, j,
                i + (1 << x) - 1, j + (1 << y) - 1,
                max(x - 1, 0), max(y - 1, 0)
                );
        }
    }
    inline int get_max(int x, int y, int xx, int yy) {
        return calc(x, y, xx, yy, highbit(xx - x + 1), highbit(yy - y + 1));
    }
};
```

线段树

朴素线段树

- 默认为最大值，可自行修改 `struct Q` `struct P` `P operator &`
- 注意建树时的下标问题 (1-based)

```cpp
const LL INF = LONG_LONG_MAX;
#define maxn 10
LL n;

namespace SGT {
    struct Q {
        LL setv;
        explicit Q(LL setv = -1): setv(setv) {}
        void operator += (const Q& q) { if (q.setv != -1) setv = q.setv; }
    };
    struct P {
        LL max;
        explicit P(LL max = -INF): max(max) {}
        void up(Q& q) { if (q.setv != -1) max = q.setv; }
    };
    template<typename T>
    P operator & (T&& a, T&& b) {
        return P(max(a.max, b.max));
    }
    P p[maxn << 2];
    Q q[maxn << 2];
#define lson o * 2, l, (l + r) / 2
#define rson o * 2 + 1, (l + r) / 2 + 1, r
    void up(int o, int l, int r) {
        if (l == r) p[o] = P();
        else p[o] = p[o * 2] & p[o * 2 + 1];
        p[o].up(q[o]);
    }
    void down(int o, int l, int r) {
        q[o * 2] += q[o]; q[o * 2 + 1] += q[o];
        q[o] = Q();
        up(lson); up(rson);
    }
    template<typename T>
    void build(T&& f, int o = 1, int l = 1, int r = n) {
        if (l == r) q[o] = f(l);
        else { build(f, lson); build(f, rson); q[o] = Q(); }
        up(o, l, r);
    }
    P query(int ql, int qr, int o = 1, int l = 1, int r = n) {
```

```cpp
        if (ql > r || l > qr) return P();
        if (ql <= l && r <= qr) return p[o];
        down(o, l, r);
        return query(ql, qr, lson) & query(ql, qr, rson);
    }
    void update(int ql, int qr, const Q& v, int o = 1, int l = 1, int r = n) {
        if (ql > r || l > qr) return;
        if (ql <= l && r <= qr) q[o] += v;
        else {
            down(o, l, r);
            update(ql, qr, v, lson); update(ql, qr, v, rson);
        }
        up(o, l, r);
    }
}

// ----------------------------------------------------------------
void solve(){
    vector<LL> arr = {1, 5, 7, 4, 2, 8 ,3, 6, 10, 9};
    n = arr.size();
    SGT::build([&](int idx){
        return SGT::Q(arr[idx-1]);
    });
    for(LL i=1; i<=n; i++){
        dbg(SGT::query(1, i).max);
    }
    SGT::update(2, 4, SGT::Q(-3));
    cout << "MODIFIED\n";
    for(LL i=1; i<=n; i++){
        dbg(SGT::query(1, i).max);
    }
}
```

- 区间修改，区间累加，查询区间和、最大值、最小值。

```cpp
#define maxn 100005
#define INF LONG_LONG_MAX
LL a[maxn];

struct IntervalTree {
#define ls o * 2, l, m
#define rs o * 2 + 1, m + 1, r
    static const LL M = maxn * 4, RS = 1E18 - 1;
    LL addv[M], setv[M], minv[M], maxv[M], sumv[M];
    int n;
    void init() {
        memset(addv, 0, sizeof addv);
        fill(setv, setv + M, RS);
        memset(minv, 0, sizeof minv);
        memset(maxv, 0, sizeof maxv);
        memset(sumv, 0, sizeof sumv);
    }
    void maintain(LL o, LL l, LL r) {
        if (l < r) {
            LL lc = o * 2, rc = o * 2 + 1;
            sumv[o] = sumv[lc] + sumv[rc];
            minv[o] = min(minv[lc], minv[rc]);
            maxv[o] = max(maxv[lc], maxv[rc]);
        } else sumv[o] = minv[o] = maxv[o] = 0;
        if (setv[o] != RS) { minv[o] = maxv[o] = setv[o]; sumv[o] = setv[o] * (r - l + 1); }
        if (addv[o]) { minv[o] += addv[o]; maxv[o] += addv[o]; sumv[o] += addv[o] * (r - l + 1); }
    }
    void build(LL o, LL l, LL r) {
        if (l == r) addv[o] = a[l];
        else {
            LL m = (l + r) / 2;
            build(ls); build(rs);
        }
        maintain(o, l, r);
    }
    void pushdown(LL o) {
        LL lc = o * 2, rc = o * 2 + 1;
```

```cpp
        if (setv[o] != RS) {
            setv[lc] = setv[rc] = setv[o];
            addv[lc] = addv[rc] = 0;
            setv[o] = RS;
        }
        if (addv[o]) {
            addv[lc] += addv[o]; addv[rc] += addv[o];
            addv[o] = 0;
        }
    }
    void update(LL p, LL q, LL o, LL l, LL r, LL v, LL op) {
        if (p <= r && l <= q){
            if (p <= l && r <= q) {
                if (op == 2) { setv[o] = v; addv[o] = 0; }
                else addv[o] += v;
            } else {
                pushdown(o);
                LL m = (l + r) / 2;
                update(p, q, ls, v, op); update(p, q, rs, v, op);
            }
        }
        maintain(o, l, r);
    }
    void query(LL p, LL q, LL o, LL l, LL r, LL add, LL& ssum, LL& smin, LL& smax) {
        if (p > r || l > q) return;
        if (setv[o] != RS) {
            LL v = setv[o] + add + addv[o];
            ssum += v * (min(r, q) - max(l, p) + 1);
            smin = min(smin, v);
            smax = max(smax, v);
        } else if (p <= l && r <= q) {
            ssum += sumv[o] + add * (r - l + 1);
            smin = min(smin, minv[o] + add);
            smax = max(smax, maxv[o] + add);
        } else {
            LL m = (l + r) / 2;
            query(p, q, ls, add + addv[o], ssum, smin, smax);
            query(p, q, rs, add + addv[o], ssum, smin, smax);
        }
    }
    // 简化接口
    void build(int _n) {
        n = _n;
        build(1, 1, n);
    }

    void range_add(int l, int r, int val) {
        update(l, r, 1, 1, n, val, 1);
    }

    void range_set(int l, int r, int val) {
        update(l, r, 1, 1, n, val, 2);
    }

    void range_query(int l, int r, LL& sum, LL& min_val, LL& max_val) {
        sum = 0;
        min_val = INF;
        max_val = -INF;
        query(l, r, 1, 1, n, 0, sum, min_val, max_val);
    }
} IT;
// ------------------------------------------------------------------------
void solve(){
    IT.init();

    LL n = 5;
    vector<int> data = {1, 3, 5, 7, 9};
    for (int i = 0; i < n; i++) {
        a[i + 1] = data[i];  // 注意：线段树从 1 开始索引
    }

```

```
    IT.build(n);

    LL sum, min_val, max_val;
    IT.range_query(1, 5, sum, min_val, max_val);
    cout << " " << sum << " " << min_val << " " << max_val << endl;

    IT.range_add(2, 4, 2);
    IT.range_query(1, 5, sum, min_val, max_val);
    cout << " " << sum << " " << min_val << " " << max_val << endl;

    IT.range_set(3, 5, 10);
    IT.range_query(1, 5, sum, min_val, max_val);
    cout << " " << sum << " " << min_val << " " << max_val << endl;

    IT.range_query(2, 4, sum, min_val, max_val);
    cout << " " << sum << " " << min_val << " " << max_val << endl;
}
```

动态开点

```
namespace SGT{
    const LL N = 3e5 + 10, INF = LONG_LONG_MAX;
    LL sum[N << 2], lazy[N << 2];
//  LL minn[N << 2], lazy2[N << 2];
    LL lson[N << 2], rson[N << 2],  tot = 0, root = 0;

    inline void push_up(LL rt){
        sum[rt] = sum[lson[rt]] + sum[rson[rt]];
//      minn[rt] = min(minn[lson[rt]], minn[rson[rt]]);
    }
    inline void push_down(LL rt, LL m){
        if(!lazy[rt]) return;
        if(!lson[rt]){
            lson[rt] = ++tot;
//          minn[lson[rt]] = INF;
        }
        if(!rson[rt]){
            rson[rt] = ++tot;
//          minn[rson[rt]] = INF;
        }
        lazy[lson[rt]] += lazy[rt], lazy[rson[rt]] += lazy[rt];
        sum[lson[rt]] += lazy[rt] * (m - (m >> 1));
        sum[rson[rt]] += lazy[rt] * (m >> 1);
        lazy[rt] = 0;

//      lazy2[lson[rt]] = min(lazy2[lson[rt]], lazy2[rt]);
//      lazy2[rson[rt]] = min(lazy2[rson[rt]], lazy2[rt]);
//      minn[lson[rt]] = min(minn[lson[rt]], lazy2[rt]);
//      minn[rson[rt]] = min(minn[rson[rt]], lazy2[rt]);
//      lazy2[rt] = INF;
    }

    static void add_range(LL &rt ,LL l, LL r, LL L, LL R, LL val){
        if(!rt){
            rt = ++tot;
//          minn[rt] = INF;
//          lazy2[rt] = INF;
        }
        if(l >= L && r <= R){
            lazy[rt] += val;
            sum[rt] += val * (r - l + 1);
//          minn[rt] = min(minn[rt], val);
//          lazy2[rt] = min(lazy2[rt], val);
            return;
        }
        push_down(rt, r - l + 1);
        LL mid = l + r >> 1;
        if(mid >= L) add_range(lson[rt], l, mid, L, R, val);
        if(mid < R) add_range(rson[rt], mid + 1, r, L, R, val);
        push_up(rt);
    }
```

```
    static void add_point(LL &rt, LL l, LL r, LL pos, LL val){
        if(!rt){
            rt = ++tot;
//          minn[rt] = INF;
        }
        if(l == r){
            sum[rt] += val;
//          minn[rt] = min(minn[rt], val);
            return;
        }
        LL mid = l + r >> 1;
        if(mid >= pos) add_point(lson[rt], l, mid, pos, val);
        else add_point(rson[rt], mid + 1, r, pos, val);
        push_up(rt);
    }

    static LL query(LL rt, LL l, LL r, LL L, LL R){
        if(!rt) return 0;
//      if(!rt) return INF;
        if(l >= L && r <= R) return sum[rt];
//      if(l >= L && r <= R) return minn[rt];
        push_down(rt, r - l + 1);
        LL mid = l + r >> 1;
        LL ans = 0;
//      LL ans = INF;
        if(mid >= L) ans += query(lson[rt], l, mid, L, R);
        if(mid < R) ans += query(rson[rt], mid + 1, r, L, R);
//      if(mid >= L) ans = min(ans, query(lson[rt], l, mid, L, R));
//      if(mid < R) ans = min(ans, query(rson[rt], mid + 1, r, L, R));
        return ans;
    }
};
// --------------------------- Template End ----------------------------
// !!!!!!!!!!!!!!!!!!!!!!!!!!! Tester Start !!!!!!!!!!!!!!!!!!!!!!!!!!!!

void solve(){
    LL rt = 0, l = -1e9, r = 1e9; // 根（必需为 0）和值域（操作范围）
    SGT::root = rt;
    SGT::add_point(rt, l, r, 1e4, 1e4);
    SGT::add_range(rt, l, r, 100000, 100010, 100);
    cout << SGT::query(rt, l, r, -1e9, 0) << '\n';
    cout << SGT::query(rt, l, r, 0, 1e5) << '\n';
    cout << SGT::query(rt, l, r, 1e5, 1e6) << '\n';
    cout << SGT::query(rt, l, r, 1e6, 1e10) << '\n';
}
```

树状数组

- 单点修改，区间查询
- 频次统计下的 k 小值
- 维护差分数组时的区间修改，单点查询

```
#define M 100005

namespace BIT {
    LL c[M]; // 注意初始化开销
    inline int lowbit(int x) { return x & -x; }
    void add(int x, LL v) { // 单点加
        for (int i = x; i < M; i += lowbit(i))
            c[i] += v;
    }
    LL sum(int x) { // 前缀和
        LL ret = 0;
        for (int i = x; i > 0; i -= lowbit(i))
            ret += c[i];
        return ret;
    }
    int kth(LL k) { // 频次统计下从小到大第 k 个，详见应用
        int p = 0;
```

```
        for (int lim = 1 << 20; lim; lim /= 2)
            if (p + lim < M && c[p + lim] < k) {
                p += lim;
                k -= c[p];
            }
        return p + 1;
    }
    LL sum(int l, int r) { return sum(r) - sum(l - 1); } // 区间和
    // 区间加（此时树状数组为差分数组，sum(x) 为第 x 个数的值）
    void add(int l, int r, LL v) { add(l, v); add(r + 1, -v); }
}
// -----------------------------------------------------------------
void solve(){
    vector<LL> a={9, 9, 9, 9, 5, 3, 3, 3, 1, 1};
    LL n = a.size(), i;
    for(i=1; i<=n; i++) BIT::add(a[i-1], 1);
    // 1 1 3 3 3 5 9 9 9 9
    for(i=1; i<=n; i++) cout << BIT::kth(i) << ' ';
}
```

- 区间修改、区间查询

```
#define maxn 100005

namespace BIT {
    int n;
    int c[maxn], cc[maxn];
    inline int lowbit(int x) { return x & -x; }
    void init(int siz){ // 初始化
        n = siz;
        for(LL i=0; i<=n; i++){
            c[i] = cc[i] = 0;
        }
    }
    void add(int x, int v) { // 不要用这个
        for (int i = x; i <= n; i += lowbit(i)) {
            c[i] += v; cc[i] += x * v;
        }
    }
    void add(int l, int r, int v) { add(l, v); add(r + 1, -v); } // 区间修改
    int sum(int x) { // 前缀和
        int ret = 0;
        for (int i = x; i > 0; i -= lowbit(i))
            ret += (x + 1) * c[i] - cc[i];
        return ret;
    }
    int sum(int l, int r) { return sum(r) - sum(l - 1); } // 区间和
}
// -----------------------------------------------------------------
void solve(){
    LL i, n=8;
    BIT::init(n);
    BIT::add(2, 4, 2);
    for(i=1; i<=n; i++) cout << BIT::sum(i, i) << ' ';
    cout << '\n';
    cout << BIT::sum(5) << '\n';
    cout << BIT::sum(2, 3) << '\n';
}
```

- 三维

```
#define maxn 105

namespace BIT{
    int n;
    LL c[maxn][maxn][maxn];
    inline int lowbit(int x) { return x & -x; }
    void init(int siz){
        n = siz;
        for(int i=0; i<=n; i++){
            for(int j=0; j<=n; j++){
                for(int k=0; k<=n; k++){
```

```
                    c[i][j][k] = 0;
                }
            }
        }
    }
    void update(int x, int y, int z, int d) {
        for (int i = x; i <= n; i += lowbit(i))
            for (int j = y; j <= n; j += lowbit(j))
                for (int k = z; k <= n; k += lowbit(k))
                    c[i][j][k] += d;
    }
    LL query(int x, int y, int z) {
        LL ret = 0;
        for (int i = x; i > 0; i -= lowbit(i))
            for (int j = y; j > 0; j -= lowbit(j))
                for (int k = z; k > 0; k -= lowbit(k))
                    ret += c[i][j][k];
        return ret;
    }
    LL solve(int x, int y, int z, int xx, int yy, int zz) {
        return    query(xx, yy, zz)
        - query(xx, yy, z - 1)
        - query(xx, y - 1, zz)
        - query(x - 1, yy, zz)
        + query(xx, y - 1, z - 1)
        + query(x - 1, yy, z - 1)
        + query(x - 1, y - 1, zz)
        - query(x - 1, y - 1, z - 1);
    }
}
```

数学

快速乘

```
LL mul(LL a, LL b, LL m) {
    LL ret = 0;
    while (b) {
        if (b & 1) {
            ret += a;
            if (ret >= m) ret -= m;
        }
        a += a;
        if (a >= m) a -= m;
        b >>= 1;
    }
    return ret;
}
```

- O(1)

```
LL mul(LL u, LL v, LL p) {
    return (u * v - LL((long double) u * v / p) * p + p) % p;
}
LL mul(LL u, LL v, LL p) { // 卡常
    LL t = u * v - LL((long double) u * v / p) * p;
    return t < 0 ? t + p : t;
}
```

高斯消元

- n 是方程个数，m 是未知量个数，a[n][m+1] 是增广矩阵
- x[m] 是每个未知量的解（如果有），free_x[m] 是每个未知量是否为自由变量。

```
typedef double LD;
const LD eps = 1E-10;
const int maxn = 2000 + 10;

int n, m;
```

```
LD a[maxn][maxn], x[maxn];
bool free_x[maxn];

inline int sgn(LD x) { return (x > eps) - (x < -eps); }

int gauss(LD a[maxn][maxn], int n, int m) {
//int gauss() {
    memset(free_x, 1, sizeof free_x); memset(x, 0, sizeof x);
    int r = 0, c = 0;
    while (r < n && c < m) {
        int m_r = r;
        FOR (i, r + 1, n)
        if (fabs(a[i][c]) > fabs(a[m_r][c])) m_r = i;
        if (m_r != r)
            FOR (j, c, m + 1)
            swap(a[r][j], a[m_r][j]);
        if (!sgn(a[r][c])) {
            a[r][c] = 0;
            ++c;
            continue;
        }
        FOR (i, r + 1, n)
        if (a[i][c]) {
            LD t = a[i][c] / a[r][c];
            FOR (j, c, m + 1) a[i][j] -= a[r][j] * t;
        }
        ++r; ++c;
    }
    FOR (i, r, n)
    if (sgn(a[i][m])) return -1;
    if (r < m) {
        FORD (i, r - 1, -1) {
            int f_cnt = 0, k = -1;
            FOR (j, 0, m)
            if (sgn(a[i][j]) && free_x[j]) {
                ++f_cnt;
                k = j;
            }
            if(f_cnt > 0) continue;
            LD s = a[i][m];
            FOR (j, 0, m)
            if (j != k) s -= a[i][j] * x[j];
            x[k] = s / a[i][k];
            free_x[k] = 0;
        }
        return m - r;
    }
    FORD (i, m - 1, -1) {
        LD s = a[i][m];
        FOR (j, i + 1, m)
        s -= a[i][j] * x[j];
        x[i] = s / a[i][i];
    }
    return 0;
}
```

快速幂

- 如果模数是素数，则可在函数体内加上 `n %= MOD - 1;`（费马小定理）。

```
LL bin(LL x, LL n, LL MOD) {
    LL ret = MOD != 1;
    for (x %= MOD; n; n >>= 1, x = x * x % MOD)
        if (n & 1) ret = ret * x % MOD;
    return ret;
}
```

- 防爆 LL
- 前置模板：快速乘

```
LL bin(LL x, LL n, LL MOD) {
```

```cpp
    LL ret = MOD != 1;
    for (x %= MOD; n; n >>= 1, x = mul(x, x, MOD))
        if (n & 1) ret = mul(ret, x, MOD);
    return ret;
}
```

高精度

- https://github.com/Baobaobear/MiniBigInteger/blob/main/bigint_tiny.h，带有压位优化
- 按需实现

```cpp
#include <algorithm>
#include <cstdio>
#include <string>
#include <vector>

struct BigIntTiny {
    int sign;
    std::vector<int> v;

    BigIntTiny() : sign(1) {}
    BigIntTiny(const std::string &s) { *this = s; }
    BigIntTiny(int v) {
        char buf[21];
        sprintf(buf, "%d", v);
        *this = buf;
    }
    void zip(int unzip) {
        if (unzip == 0) {
            for (int i = 0; i < (int)v.size(); i++)
              v[i] = get_pos(i * 4) + get_pos(i * 4 + 1) * 10 + get_pos(i * 4 + 2) * 100 + get_pos(i * 4 + 3) * 1000;
        } else
            for (int i = (v.resize(v.size() * 4), (int)v.size() - 1), a; i >= 0; i--)
                a = (i % 4 >= 2) ? v[i / 4] / 100 : v[i / 4] % 100, v[i] = (i & 1) ? a / 10 : a % 10;
        setsign(1, 1);
    }
    int get_pos(unsigned pos) const { return pos >= v.size() ? 0 : v[pos]; }
    BigIntTiny &setsign(int newsign, int rev) {
        for (int i = (int)v.size() - 1; i > 0 && v[i] == 0; i--)
            v.erase(v.begin() + i);
        sign = (v.size() == 0 || (v.size() == 1 && v[0] == 0)) ? 1 : (rev ? newsign * sign : newsign);
        return *this;
    }
    std::string to_str() const {
        BigIntTiny b = *this;
        std::string s;
        for (int i = (b.zip(1), 0); i < (int)b.v.size(); ++i)
            s += char(*(b.v.rbegin() + i) + '0');
        return (sign < 0 ? "-" : "") + (s.empty() ? std::string("0") : s);
    }
    bool absless(const BigIntTiny &b) const {
        if (v.size() != b.v.size()) return v.size() < b.v.size();
        for (int i = (int)v.size() - 1; i >= 0; i--)
            if (v[i] != b.v[i]) return v[i] < b.v[i];
        return false;
    }
    BigIntTiny operator-() const {
        BigIntTiny c = *this;
        c.sign = (v.size() > 1 || v[0]) ? -c.sign : 1;
        return c;
    }
    BigIntTiny &operator=(const std::string &s) {
        if (s[0] == '-')
            *this = s.substr(1);
        else {
            for (int i = (v.clear(), 0); i < (int)s.size(); ++i)
                v.push_back(*(s.rbegin() + i) - '0');
            zip(0);
        }
        return setsign(s[0] == '-' ? -1 : 1, sign = 1);
```

```
    }
    bool operator<(const BigIntTiny &b) const {
        return sign != b.sign ? sign < b.sign : (sign == 1 ? absless(b) : b.absless(*this));
    }
    bool operator==(const BigIntTiny &b) const { return v == b.v && sign == b.sign; }
    BigIntTiny &operator+=(const BigIntTiny &b) {
        if (sign != b.sign) return *this = (*this) - -b;
        v.resize(std::max(v.size(), b.v.size()) + 1);
        for (int i = 0, carry = 0; i < (int)b.v.size() || carry; i++) {
            carry += v[i] + b.get_pos(i);
            v[i] = carry % 10000, carry /= 10000;
        }
        return setsign(sign, 0);
    }
    BigIntTiny operator+(const BigIntTiny &b) const {
        BigIntTiny c = *this;
        return c += b;
    }
    void add_mul(const BigIntTiny &b, int mul) {
        v.resize(std::max(v.size(), b.v.size()) + 2);
        for (int i = 0, carry = 0; i < (int)b.v.size() || carry; i++) {
            carry += v[i] + b.get_pos(i) * mul;
            v[i] = carry % 10000, carry /= 10000;
        }
    }
    BigIntTiny operator-(const BigIntTiny &b) const {
        if (b.v.empty() || b.v.size() == 1 && b.v[0] == 0) return *this;
        if (sign != b.sign) return (*this) + -b;
        if (absless(b)) return -(b - *this);
        BigIntTiny c;
        for (int i = 0, borrow = 0; i < (int)v.size(); i++) {
            borrow += v[i] - b.get_pos(i);
            c.v.push_back(borrow);
            c.v.back() -= 10000 * (borrow >>= 31);
        }
        return c.setsign(sign, 0);
    }
    BigIntTiny operator*(const BigIntTiny &b) const {
        if (b < *this) return b * *this;
        BigIntTiny c, d = b;
        for (int i = 0; i < (int)v.size(); i++, d.v.insert(d.v.begin(), 0))
            c.add_mul(d, v[i]);
        return c.setsign(sign * b.sign, 0);
    }
    BigIntTiny operator/(const BigIntTiny &b) const {
        BigIntTiny c, d;
        BigIntTiny e=b;
        e.sign=1;

        d.v.resize(v.size());
        double db = 1.0 / (b.v.back() + (b.get_pos((unsigned)b.v.size() - 2) / 1e4) +
                           (b.get_pos((unsigned)b.v.size() - 3) + 1) / 1e8);
        for (int i = (int)v.size() - 1; i >= 0; i--) {
            c.v.insert(c.v.begin(), v[i]);
            int m = (int)((c.get_pos((int)e.v.size()) * 10000 + c.get_pos((int)e.v.size() - 1)) * db);
            c = c - e * m, c.setsign(c.sign, 0), d.v[i] += m;
            while (!(c < e))
                c = c - e, d.v[i] += 1;
        }
        return d.setsign(sign * b.sign, 0);
    }
    BigIntTiny operator%(const BigIntTiny &b) const { return *this - *this / b * b; }
    bool operator>(const BigIntTiny &b) const { return b < *this; }
    bool operator<=(const BigIntTiny &b) const { return !(b < *this); }
    bool operator>=(const BigIntTiny &b) const { return !(*this < b); }
    bool operator!=(const BigIntTiny &b) const { return !(*this == b); }
};
```

中位数

```
struct Median{
    const LL inf = LONG_LONG_MAX;
    multiset<LL> smaller,bigger;
    LL mid = -inf;
    int sz = 2;

    void init(){
        smaller.clear();bigger.clear();
        mid = -inf;
        smaller.insert(-inf);
        bigger.insert(inf);
        sz = 2;
    }

    LL query(){
        return mid;
    }

    LL query_size(){
        return sz - 2;
    }

    void add(LL val){
        if(sz & 1){ // odd add
            if(val >= mid){
                smaller.insert(mid);
                bigger.insert(val);
            }else{
                smaller.insert(val);
                bigger.insert(mid);
            }
            mid = *smaller.rbegin();
        }
        else{ // add even
            if(val >= *smaller.rbegin() && val <= *bigger.begin()){
                mid = val;
            }else if(val < *smaller.rbegin()){
                smaller.insert(val);
                mid = *smaller.rbegin();
                smaller.erase(smaller.find(mid));
            }else{
                bigger.insert(val);
                mid = *bigger.begin();
                bigger.erase(bigger.find(mid));
            }
        }
        sz++;
    }

    void erase(LL val){
        if(sz & 1){
            if(val == mid){
                mid = *smaller.rbegin();
                sz --;
                return;
            }
            if(val <= *smaller.rbegin())
            {
                smaller.erase(smaller.find(val) );
                smaller.insert(mid);
                mid = *smaller.rbegin();
            }else if(val >= *bigger.begin()){
                bigger.erase(bigger.find(val) );
                bigger.insert(mid);
                mid = *smaller.rbegin();
            }
        }else{ //erase even
            if(val <= *smaller.rbegin()){
                smaller.erase(smaller.find(val) );
```

```
                mid = *bigger.begin();
                bigger.erase(bigger.find(mid));
            }else if(val >= *bigger.begin()){
                bigger.erase(bigger.find(val));
                mid = *smaller.rbegin();
                smaller.erase(smaller.find(mid));
            }
        }
        sz --;
    }
};
```

矩阵运算

```
#define MOD 998244353
#define M 10

struct Mat {
    LL m;
    LL v[M][M];
    Mat(int siz=2) {
        m = siz;
        for(int i=0; i<=m; i++){
            for(int j=0; j<=m; j++){
                v[i][j] = 0;
            }
        }
    }
    void eye() { FOR (i, 0, m) v[i][i] = 1; }
    LL* operator [] (LL x) { return v[x]; }
    const LL* operator [] (LL x) const { return v[x]; }
    Mat operator * (const Mat& B) {
        const Mat& A = *this;
        Mat ret;
        FOR (k, 0, m)
        FOR (i, 0, m) if (A[i][k])
            FOR (j, 0, m)
            ret[i][j] = (ret[i][j] + A[i][k] * B[k][j]) % MOD;
        return ret;
    }
    Mat pow(LL n) const {
        Mat A = *this, ret; ret.eye();
        for (; n; n >>= 1, A = A * A)
            if (n & 1) ret = ret * A;
        return ret;
    }
    Mat operator + (const Mat& B) {
        const Mat& A = *this;
        Mat ret;
        FOR (i, 0, m)
        FOR (j, 0, m)
        ret[i][j] = (A[i][j] + B[i][j]) % MOD;
        return ret;
    }
    void pprint() const {
        FOR (i, 0, m)
        FOR (j, 0, m)
        printf("%lld%c", (*this)[i][j], j == m - 1 ? '\n' : ' ');
    }
};
// ------------------------------------------------------------------
void solve(){
    Mat mat1, mat2;
    mat1.eye();
    mat1[1][0] = 2; // 0-based
    mat2.eye();
    mat2[1][1] = 4;
    Mat mat3 = mat1 * mat2;
    mat3.pprint();
}
```

数论分块

$f(i) = \lfloor \frac{n}{i} \rfloor = v$ 时 i 的取值范围是 $[l, r]$。

```
void sqrt_decomposition(LL n){
    for (LL l = 1, v, r; l <= n; l = r + 1) {
        v = n / l; r = n / v;
        printf("%lld / [%lld, %lld] = %lld\n", n, l, r, v);
    }
}
```

质数筛

- $\mathcal{O}(n)$

```
const LL p_max = 1E6 + 100;
LL pr[p_max], p_sz;
void get_prime() {
    static bool vis[p_max];
    FOR (i, 2, p_max) {
        if (!vis[i]) pr[p_sz++] = i;
        FOR (j, 0, p_sz) {
            if (pr[j] * i >= p_max) break;
            vis[pr[j] * i] = 1;
            if (i % pr[j] == 0) break;
        }
    }
}
```

欧拉函数

朴素

```
int phi(int x)
{
    int res = x;
    for (int i = 2; i <= x / i; i ++ )
        if (x % i == 0)
        {
            res = res / i * (i - 1);
            while (x % i == 0) x /= i;
        }
    if (x > 1) res = res / x * (x - 1);

    return res;
}
```

筛法求欧拉函数

- 前置模板：质数筛

```
const LL p_max = 1E5 + 100;
LL phi[p_max];
void get_phi() {
    phi[1] = 1;
    static bool vis[p_max];
    static LL prime[p_max], p_sz, d;
    FOR (i, 2, p_max) {
        if (!vis[i]) {
            prime[p_sz++] = i;
            phi[i] = i - 1;
        }
        for (LL j = 0; j < p_sz && (d = i * prime[j]) < p_max; ++j) {
            vis[d] = 1;
            if (i % prime[j] == 0) {
                phi[d] = phi[i] * prime[j];
                break;
            }
            else phi[d] = phi[i] * (prime[j] - 1);
        }
```

```
    }
}
```

素性测试

试除法

- $\mathcal{O}(\sqrt{n})$

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

Miller–Rabin

- 前置：快速幂
- $\mathcal{O}(k \times \log^3 n)$

```
bool miller_rabin(LL n) {
    static vector<LL> tester = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37};
    if (n < 3 || n % 2 == 0) return n == 2;
    if (n % 3 == 0) return n == 3;
    LL u = n - 1, t = 0;
    while (u % 2 == 0) u /= 2, ++t;
    for (auto nt: tester) {
        if(nt >= n) continue;
        LL v = bin(nt, u, n);
        if (v == 1) continue;
        LL s;
        for (s = 0; s < t; ++s) {
            if (v == n - 1) break;
            v = v * v % n;
        }
        if (s == t) return false;
    }
    return true;
}
```

质因数分解

朴素质因数分解

- 前置模板：素数筛
- 带指数
- $\mathcal{O}(\frac{\sqrt{N}}{\ln N})$

```
LL factor[30], f_sz, factor_exp[30];
void get_factor(LL x) {
    f_sz = 0;
    LL t = sqrt(x + 0.5);
    for (LL i = 0; pr[i] <= t; ++i)
        if (x % pr[i] == 0) {
            factor_exp[f_sz] = 0;
            while (x % pr[i] == 0) {
                x /= pr[i];
                ++factor_exp[f_sz];
            }
            factor[f_sz++] = pr[i];
        }
    if (x > 1) {
        factor_exp[f_sz] = 1;
        factor[f_sz++] = x;
    }
}
```

- 不带指数

```
LL factor[30], f_sz;
void get_factor(LL x) {
    f_sz = 0;
    LL t = sqrt(x + 0.5);
    for (LL i = 0; pr[i] <= t; ++i)
        if (x % pr[i] == 0) {
            factor[f_sz++] = pr[i];
            while (x % pr[i] == 0) x /= pr[i];
        }
    if (x > 1) factor[f_sz++] = x;
}
```

Pollard-Rho

- 前置：素数测试

```
mt19937 mt(time(0));
LL pollard_rho(LL n, LL c) {
    LL x = uniform_int_distribution<LL>(1, n - 1)(mt), y = x;
    auto f = [&](LL v) { LL t = mul(v, v, n) + c; return t < n ? t : t - n; };
    while (1) {
        x = f(x); y = f(f(y));
        if (x == y) return n;
        LL d = gcd(abs(x - y), n);
        if (d != 1) return d;
    }
}

LL fac[100], fcnt;
void get_fac(LL n, LL cc = 19260817) {
    if (n == 4) { fac[fcnt++] = 2; fac[fcnt++] = 2; return; }
    if (miller_rabin(n)) { fac[fcnt++] = n; return; }
    LL p = n;
    while (p == n) p = pollard_rho(n, --cc);
    get_fac(p); get_fac(n / p);
}

void go_fac(LL n) { fcnt = 0; if (n > 1) get_fac(n); }
```

原根

- 前置模板：质因数分解、快速幂
- 要求 p 为质数
- 别忘了调用质因数分解的函数

```
LL find_smallest_primitive_root(LL p) {
    get_factor(p - 1);
    FOR (i, 2, p) {
        bool flag = true;
        FOR (j, 0, f_sz)
        if (bin(i, (p - 1) / factor[j], p) == 1) {
            flag = false;
            break;
        }
        if (flag) return i;
    }
//  assert(0);
    return -1;
}
```

欧几里得

- 朴素

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

- 卡常

```cpp
inline int ctz(LL x) { return __builtin_ctzll(x); }
LL gcd(LL a, LL b) {
    if (!a) return b; if (!b) return a;
    int t = ctz(a | b);
    a >>= ctz(a);
    do {
        b >>= ctz(b);
        if (a > b) swap(a, b);
        b -= a;
    } while (b);
    return a << t;
}
```

扩展欧几里得

- 求 $ax + by = gcd(a, b)$ 的一组解
- 如果 a 和 b 互素，那么 x 是 a 在模 b 下的逆元
- 注意 x 和 y 可能是负数

```cpp
LL ex_gcd(LL a, LL b, LL &x, LL &y) {
    if (b == 0) { x = 1; y = 0; return a; }
    LL ret = ex_gcd(b, a % b, y, x);
    y -= a / b * x;
    return ret;
}
```

二次剩余

- 求解二次同余方程
- 给定 a, p，求一组 x 满足 $x^2 \equiv a \pmod p$
- 前置模板：快速幂

```cpp
LL q1, q2, w;
struct P { // x + y * sqrt(w)
    LL x, y;
};

P pmul(const P& a, const P& b, LL p) {
    P res;
    res.x = (a.x * b.x + a.y * b.y % p * w) % p;
    res.y = (a.x * b.y + a.y * b.x) % p;
    return res;
}

P bin(P x, LL n, LL MOD) {
    P ret = {1, 0};
    for (; n; n >>= 1, x = pmul(x, x, MOD))
        if (n & 1) ret = pmul(ret, x, MOD);
    return ret;
}
LL Legendre(LL a, LL p) { return bin(a, (p - 1) >> 1, p); }

LL equation_solve(LL b, LL p) {
    if (p == 2) return 1;
    if ((Legendre(b, p) + 1) % p == 0)
        return -1;
    LL a;
    while (true) {
        a = rand() % p;
        w = ((a * a - b) % p + p) % p;
        if ((Legendre(w, p) + 1) % p == 0)
            break;
    }
    return bin({a, 1}, (p + 1) >> 1, p).x;
}
// !!!!!!!!!!!!!!!!!!!!!!!!!!! Tester Start !!!!!!!!!!!!!!!!!!!!!!!!!!!!
void solve(){
    LL a, p; cin >> a >> p;
```

```
    a = a % p;
    LL x = equation_solve(a, p);
    if (x == -1) {
        puts("No root");
    } else {
        LL y = p - x;
        if (x == y){
            cout << x << endl;
        }else{
            LL tx = min(x, y), ty = max(x, y);
            cout << tx << " " << ty << endl;
        }
    }
}
// !!!!!!!!!!!!!!!!!!!!!!!!!!!! Tester End !!!!!!!!!!!!!!!!!!!!!!!!!!!!!

```

中国剩余定理

- 求解线性同余方程组
- $$\begin{cases} x & \equiv r_1 \pmod{m_1} \\ x & \equiv r_2 \pmod{m_2} \\ & \vdots \\ x & \equiv r_k \pmod{m_k} \end{cases}$$
- 无解返回 -1
- 前置模板：扩展欧几里得

```
LL CRT(LL *m, LL *r, LL n) {
    if (!n) return 0;
    LL M = m[0], R = r[0], x, y, d;
    FOR (i, 1, n) {
        d = ex_gcd(M, m[i], x, y);
        if ((r[i] - R) % d) return -1;
        x = (r[i] - R) / d * x % (m[i] / d);
        // 防爆 LL
        // x = mul((r[i] - R) / d, x, m[i] / d);
        R += x * M;
        M = M / d * m[i];
        R %= M;
    }
    return R >= 0 ? R : R + M;
}
```

逆元

- 如果 p 是素数，使用快速幂（费马小定理）
- 前置模板：快速幂

```
inline LL get_inv(LL x, LL p) { return bin(x, p - 2, p); }
```

- 如果 p 不是素数，使用拓展欧几里得
- 前置模板：扩展欧几里得

```
LL get_inv(LL a, LL M) {
    static LL x, y;
    assert(exgcd(a, M, x, y) == 1);
    return (x % M + M) % M;
}
```

- 预处理 1~n 的逆元

```
LL inv[N];
void inv_init(LL n, LL p) {
    inv[1] = 1;
    FOR (i, 2, n)
```

```
        inv[i] = (p - p / i) * inv[p % i] % p;
}
```

- 预处理阶乘及其逆元

```
LL invf[M], fac[M] = {1};
void fac_inv_init(LL n, LL p) {
    FOR (i, 1, n)
        fac[i] = i * fac[i - 1] % p;
    invf[n - 1] = bin(fac[n - 1], p - 2, p);
    FORD (i, n - 2, -1)
        invf[i] = invf[i + 1] * (i + 1) % p;
}
```

组合数

组合数预处理（递推法）

```
LL C[M][M];
void init_C(int n) {
    FOR (i, 0, n) {
        C[i][0] = C[i][i] = 1;
        FOR (j, 1, i)
            C[i][j] = (C[i - 1][j] + C[i - 1][j - 1]) % MOD;
    }
}
```

预处理逆元法

- 如果数较小，模较大时使用逆元
- 前置模板：逆元-预处理阶乘及其逆元

```
inline LL C(LL n, LL m) { // n >= m >= 0
    return n < m || m < 0 ? 0 : fac[n] * invf[m] % MOD * invf[n - m] % MOD;
}
```

Lucas 定理

- 如果模数较小，数字较大，使用 Lucas 定理
- 前置模板可选 1：求组合数（如果使用阶乘逆元，需 `fac_inv_init(MOD, MOD);`）

```
LL C(LL n, LL m) { // m >= n >= 0
    if (m - n < n) n = m - n;
    if (n < 0) return 0;
    LL ret = 1;
    FOR (i, 1, n + 1)
        ret = ret * (m - n + i) % MOD * bin(i, MOD - 2, MOD) % MOD;
    return ret;
}
```

- 前置模板可选 2：模数不固定下使用，无法单独使用。

```
LL Lucas(LL n, LL m) { // m >= n >= 0
    return m ? C(n % MOD, m % MOD) * Lucas(n / MOD, m / MOD) % MOD : 1;
}
```

求具体值

- 分解质因数法

```
int primes[N], cnt;     // 存储所有质数
int sum[N];     // 存储每个质数的次数
bool st[N];     // 存储每个数是否已被筛掉

void get_primes(int n)      // 线性筛法求素数
{
    for (int i = 2; i <= n; i ++ )
    {
        if (!st[i]) primes[cnt ++ ] = i;
        for (int j = 0; primes[j] <= n / i; j ++ )
```

```
        {
            st[primes[j] * i] = true;
            if (i % primes[j] == 0) break;
        }
    }
}


int get(int n, int p)       // 求 n! 中的次数
{
    int res = 0;
    while (n)
    {
        res += n / p;
        n /= p;
    }
    return res;
}


vector<int> mul(vector<int> a, int b)       // 高精度乘低精度模板
{
    vector<int> c;
    int t = 0;
    for (int i = 0; i < a.size(); i ++ )
    {
        t += a[i] * b;
        c.push_back(t % 10);
        t /= 10;
    }

    while (t)
    {
        c.push_back(t % 10);
        t /= 10;
    }

    return c;
}

get_primes(a);  // 预处理范围内的所有质数

for (int i = 0; i < cnt; i ++ )     // 求每个质因数的次数
{
    int p = primes[i];
    sum[i] = get(a, p) - get(b, p) - get(a - b, p);
}

vector<int> res;
res.push_back(1);

for (int i = 0; i < cnt; i ++ )     // 用高精度乘法将所有质因子相乘
    for (int j = 0; j < sum[i]; j ++ )
        res = mul(res, primes[i]);
```

FFT & NTT & FWT

FFT

- 计算多项式乘法，可用于高精度乘法
- $\mathcal{O}(n \log n)$

```
typedef double LD;
const LD PI = acos(-1.0);

struct Complex {
    LD r, i;
    Complex(LD r = 0, LD i = 0) : r(r), i(i) {}
    Complex operator + (const Complex& other) const {
        return Complex(r + other.r, i + other.i);
    }
```

```cpp
    Complex operator - (const Complex& other) const {
        return Complex(r - other.r, i - other.i);
    }
    Complex operator * (const Complex& other) const {
        return Complex(r * other.r - i * other.i, r * other.i + i * other.r);
    }
};

// 快速傅里叶变换，p=1 为正向，p=-1 为反向
void FFT(vector<Complex>& x, int p) {
    int n = x.size();
    for (int i = 0, t = 0; i < n; ++i) {
        if (i > t) swap(x[i], x[t]);
        for (int j = n >> 1; (t ^= j) < j; j >>= 1);
    }
    for (int h = 2; h <= n; h <<= 1) {
        Complex wn(cos(p * 2 * PI / h), sin(p * 2 * PI / h));
        for (int i = 0; i < n; i += h) {
            Complex w(1, 0);
            for (int j = 0; j < h / 2; ++j) {
                Complex u = x[i + j];
                Complex v = x[i + j + h/2] * w;
                x[i + j] = u + v;
                x[i + j + h/2] = u - v;
                w = w * wn;
            }
        }
    }
    if (p == -1) {
        for (int i = 0; i < n; ++i) {
            x[i].r /= n;
        }
    }
}

// 计算两个多项式的卷积，返回结果多项式的系数向量
vector<LD> convolution(const vector<LD>& a, const vector<LD>& b) {
    int len = 1;
    int n = a.size(), m = b.size();
    while (len < n + m - 1) len <<= 1;
    vector<Complex> fa(len), fb(len);
    for (int i = 0; i < n; ++i) fa[i] = Complex(a[i], 0);
    for (int i = 0; i < m; ++i) fb[i] = Complex(b[i], 0);
    FFT(fa, 1);
    FFT(fb, 1);
    for (int i = 0; i < len; ++i) {
        fa[i] = fa[i] * fb[i];
    }
    FFT(fa, -1);
    vector<LD> res(n + m - 1);
    for (int i = 0; i < n + m - 1; ++i) {
        res[i] = fa[i].r;
    }
    return res;
}
```

NTT

- 用于大整数乘法时，位数不宜过高（在 MOD=998244353 的情况下，总位数不超过 $12324004(3510^2)$）
- 前置模板：快速幂、逆元

```cpp
const int N = 1e5+10;
const int MOD = 998244353; // 模数
const int G = 3; // 原根

LL wn[N << 2], rev[N << 2];
int NTT_init(int n_) {
    int step = 0; int n = 1;
    for ( ; n < n_; n <<= 1) ++step;
    FOR (i, 1, n)
    rev[i] = (rev[i >> 1] >> 1) | ((i & 1) << (step - 1));
```

```
    int g = bin(G, (MOD - 1) / n, MOD);
    wn[0] = 1;
    for (int i = 1; i <= n; ++i)
        wn[i] = wn[i - 1] * g % MOD;
    return n;
}

void NTT(vector<LL> &a, int n, int f) {
    FOR (i, 0, n) if (i < rev[i])
        std::swap(a[i], a[rev[i]]);
    for (int k = 1; k < n; k <<= 1) {
        for (int i = 0; i < n; i += (k << 1)) {
            int t = n / (k << 1);
            FOR (j, 0, k) {
                LL w = f == 1 ? wn[t * j] : wn[n - t * j];
                LL x = a[i + j];
                LL y = a[i + j + k] * w % MOD;
                a[i + j] = (x + y) % MOD;
                a[i + j + k] = (x - y + MOD) % MOD;
            }
        }
    }
    if (f == -1) {
        LL ninv = get_inv(n, MOD);
        FOR (i, 0, n)
        a[i] = a[i] * ninv % MOD;
    }
}

vector<LL> conv(vector<LL> a, vector<LL> b){
    int len_a = a.size(), len_b = b.size();
    int len = len_a + len_b - 1;
    int n = NTT_init(len);
    a.resize(n);
    b.resize(n);
    NTT(a, n, 1);
    NTT(b, n, 1);
    vector<LL> c(n);
    for (int i = 0; i < n; ++i) {
        c[i] = a[i] * b[i] % MOD;
    }
    NTT(c, n, -1);
    vector<LL> res(len);
    for (int i = 0; i < len; ++i) {
        res[i] = c[i];
    }
    return res;
}
```

FWT

```
const LL MOD = 998244353;

template<typename T>
void fwt(vector<LL> &a, int n, T f) {
    for (int d = 1; d < n; d *= 2)
        for (int i = 0, t = d * 2; i < n; i += t)
            FOR (j, 0, d)
            f(a[i + j], a[i + j + d]);
}

void AND(LL& a, LL& b) { a += b; }
void OR(LL& a, LL& b) { b += a; }
void XOR (LL& a, LL& b) {
    LL x = a, y = b;
    a = (x + y) % MOD;
    b = (x - y + MOD) % MOD;
}
void rAND(LL& a, LL& b) { a -= b; }
void rOR(LL& a, LL& b) { b -= a; }
void rXOR(LL& a, LL& b) {
```

```cpp
    static LL INV2 = (MOD + 1) / 2;
    LL x = a, y = b;
    a = (x + y) * INV2 % MOD;
    b = (x - y + MOD) * INV2 % MOD;
}

int next_power_of_two(int n) {
    if (n <= 0) return 1;
    // __lg(n-1) 返回 n-1 的最高位所在位置（0-based）
    return 1 << (__lg(n - 1) + 1);
}

template<typename T, typename F>
vector<LL> conv(vector<LL> a, vector<LL> b, T f, F inv_f){
    LL len_a = a.size(), len_b = b.size(), len = max(len_a, len_b), n = next_power_of_two(len);
    a.resize(n), b.resize(n);
    fwt(a, n, f), fwt(b, n, f);
    vector<LL> c(n);
    for (int i = 0; i < n; i++) {
        c[i] = a[i] * b[i] % MOD;
    }
    fwt(c, n, inv_f);
    // 提取结果 (可选)
    c.resize(len);
    return c;
}
```

线性基

贪心法

可查询最大异或和

```cpp
struct BasisGreedy{
    ULL p[64];
    BasisGreedy(){memset(p, 0, sizeof p);}
    void insert(ULL x) {
        for (int i = 63; ~i; --i) {
            if (!(x >> i))  // x 的第 i 位是 0
                continue;
            if (!p[i]) {
                p[i] = x;
                break;
            }
            x ^= p[i];
        }
    }
    ULL query_max(){
        ULL ans = 0;
        for (int i = 63; ~i; --i) {
            ans = std::max(ans, ans ^ p[i]);
        }
        return ans;
    }
};
```

高斯消元法

可查询任意大异或和

```cpp
struct BasisGauss{
    vector<ULL> a;
    LL n, tmp, cnt;

    BasisGauss(){a = {0};}

    void insert(ULL x){
        a.push_back(x);
    }

```

```
    void init(){
        n = (LL)a.size() - 1;
        LL k=1;
        for(int i=63;i>=0;i--){
            int t=0;
            for(LL j=k;j<=n;j++){
                if((a[j]>>i)&1){
                    t=j;
                    break;
                }
            }
            if(t){
                swap(a[k],a[t]);
                for(LL j=1;j<=n;j++){
                    if(j!=k&&(a[j]>>i)&1) a[j]^=a[k];
                }
                k++;
            }
        }
        cnt = k-1;
        tmp = 1LL << cnt;
        if(cnt==n) tmp--;
    }

    LL query_xth(LL x){ // 从小到大，若 x 为负数，则查询倒数第几个
        if(x<0) x = tmp + x + 1;
        if(x>tmp) return -1;
        else{
            if(n>cnt) x--;
            LL ans=0;
            for(LL i=0; i<cnt; i++){
                if((x>>i)&1) ans^=a[cnt-i];
            }
            return ans;
        }
    }
};
```

性质与公式

低阶等幂求和

- $\sum_{i=1}^{n} i^1 = \frac{n(n+1)}{2} = \frac{1}{2}n^2 + \frac{1}{2}n$
- $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} = \frac{1}{3}n^3 + \frac{1}{2}n^2 + \frac{1}{6}n$
- $\sum_{i=1}^{n} i^3 = \left[\frac{n(n+1)}{2}\right]^2 = \frac{1}{4}n^4 + \frac{1}{2}n^3 + \frac{1}{4}n^2$
- $\sum_{i=1}^{n} i^4 = \frac{n(n+1)(2n+1)(3n^2+3n-1)}{30} = \frac{1}{5}n^5 + \frac{1}{2}n^4 + \frac{1}{3}n^3 - \frac{1}{30}n$
- $\sum_{i=1}^{n} i^5 = \frac{n^2(n+1)^2(2n^2+2n-1)}{12} = \frac{1}{6}n^6 + \frac{1}{2}n^5 + \frac{5}{12}n^4 - \frac{1}{12}n^2$

一些组合公式

- 错排公式（对于 $1 \sim n$ 的排列 P，满足 $P_i \neq i$）：$D_1 = 0, D_2 = 1, D_n = (n-1)(D_{n-1} + D_{n-2}) = n!(\frac{1}{2!} - \frac{1}{3!} + \cdots + (-1)^n \frac{1}{n!}) = \lfloor \frac{n!}{e} + 0.5 \rfloor$
- 卡塔兰数（n 对括号合法方案数，n 个结点二叉树个数，$n \times n$ 方格中对角线下方的单调路径数，凸 $n+2$ 边形的三角形划分数，n 个元素的合法出栈序列数）：$C_n = \frac{1}{n+1}\binom{2n}{n} = \frac{(2n)!}{(n+1)!n!}$

互质

若整数 a 与 m 互质（即 $\gcd(a, m) = 1$）

- 对于整数 $k = 0, 1, 2, \ldots, m-1$，$ak \mod m$ 的结果恰好是 $0, 1, 2, \ldots, m-1$ 的一个**排列**（每个数出现且仅出现一次）。
- 存在唯一的整数 b（$1 \leq b < m$），使得 $ab \equiv 1 \mod m$，此时 b 称为 a 在模 m 下的**乘法逆元**（记为 $a^{-1} \mod m$）。

图论

最短路

朴素 djikstra 算法

- 无负权边、稠密图

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

// 求 1 号点到 n 号点的最短路，如果不存在则返回-1
int dijkstra(){
    memset(dist, 0x3f, sizeof dist);
    dist[1] = 0;
    for (int i = 0; i < n - 1; i ++ ){
        int t = -1;     // 在还未确定最短路的点中，寻找距离最小的点
        for (int j = 1; j <= n; j ++ )
            if (!st[j] && (t == -1 || dist[t] > dist[j]))
                t = j;
        // 用 t 更新其他点的距离
        for (int j = 1; j <= n; j ++ )
            dist[j] = min(dist[j], dist[t] + g[t][j]);
        st[t] = true;
    }
    if (dist[n] == 0x3f3f3f3f) return -1;
    return dist[n];
}
```

堆优化的 djikstra

- 无负权边、稀疏图

```
typedef pair<int, int> PII;

int n;      // 点的数量
int h[N], w[N], e[N], ne[N], idx;       // 邻接表存储所有边
int dist[N];        // 存储所有点到 1 号点的距离
bool st[N];     // 存储每个点的最短距离是否已确定

// 求 1 号点到 n 号点的最短距离，如果不存在，则返回-1
int dijkstra(){
    memset(dist, 0x3f, sizeof dist);
    dist[1] = 0;
    priority_queue<PII, vector<PII>, greater<PII>> heap;
    heap.push({0, 1});      // first 存储距离，second 存储节点编号
    while (heap.size()){
        auto t = heap.top();
        heap.pop();
        int ver = t.second, distance = t.first;
        if (st[ver]) continue;
        st[ver] = true;
        for (int i = h[ver]; i != -1; i = ne[i]){
            int j = e[i];
            if (dist[j] > distance + w[i]){
                dist[j] = distance + w[i];
                heap.push({dist[j], j});
            }
        }
    }
    if (dist[n] == 0x3f3f3f3f) return -1;
    return dist[n];
}
```

Bellman-Ford 算法

- 有负权边、可以处理负环

```
int n, m;       // n 表示点数，m 表示边数
int dist[N];        // dist[x] 存储 1 到 x 的最短路距离
```

```
struct Edge{     // 边，a 表示出点，b 表示入点，w 表示边的权重
    int a, b, w;
}edges[M];

// 求 1 到 n 的最短路距离，如果无法从 1 走到 n，则返回-1。
int bellman_ford(){
    memset(dist, 0x3f, sizeof dist);
    dist[1] = 0;

    // 如果第 n 次迭代仍然会松弛三角不等式，就说明存在一条长度是 n+1 的最短路径，由抽屉原理，路径中至少存在两个相同的点，说明图中存在负权回路。
    for (int i = 0; i < n; i ++ ){
        for (int j = 0; j < m; j ++ ){
            int a = edges[j].a, b = edges[j].b, w = edges[j].w;
            if (dist[b] > dist[a] + w)
                dist[b] = dist[a] + w;
        }
    }

    if (dist[n] > 0x3f3f3f3f / 2) return -1;
    return dist[n];
}
```

spfa 算法

- 有负权边、不能有负环，快

```
int n;      // 总点数
int h[N], w[N], e[N], ne[N], idx;       // 邻接表存储所有边
int dist[N];        // 存储每个点到 1 号点的最短距离
bool st[N];     // 存储每个点是否在队列中

// 求 1 号点到 n 号点的最短路距离，如果从 1 号点无法走到 n 号点则返回-1
int spfa(){
    memset(dist, 0x3f, sizeof dist);
    dist[1] = 0;
    queue<int> q;
    q.push(1);
    st[1] = true;
    while (q.size()){
        auto t = q.front();
        q.pop();
        st[t] = false;
        for (int i = h[t]; i != -1; i = ne[i]){
            int j = e[i];
            if (dist[j] > dist[t] + w[i]){
                dist[j] = dist[t] + w[i];
                if (!st[j]){     // 如果队列中已存在 j，则不需要将 j 重复插入
                    q.push(j);
                    st[j] = true;
                }
            }
        }
    }
    if (dist[n] == 0x3f3f3f3f) return -1;
    return dist[n];
}
```

spfa 判断负环

```
int n;      // 总点数
int h[N], w[N], e[N], ne[N], idx;       // 邻接表存储所有边
int dist[N], cnt[N];        // dist[x] 存储 1 号点到 x 的最短距离，cnt[x] 存储 1 到 x 的最短路中经过的点数
bool st[N];     // 存储每个点是否在队列中

// 如果存在负环，则返回 true，否则返回 false。
bool spfa(){
    // 不需要初始化 dist 数组
    // 原理：如果某条最短路径上有 n 个点（除了自己），那么加上自己之后一共有 n+1 个点，由抽屉原理一定有两个点相同，所以存在环。
    queue<int> q;
    for (int i = 1; i <= n; i ++ ){
```

```
            q.push(i);
            st[i] = true;
        }
    while (q.size()){
        auto t = q.front();
        q.pop();
        st[t] = false;
        for (int i = h[t]; i != -1; i = ne[i]){
            int j = e[i];
            if (dist[j] > dist[t] + w[i]){
                dist[j] = dist[t] + w[i];
                cnt[j] = cnt[t] + 1;
                if (cnt[j] >= n) return true;       // 如果从 1 号点到 x 的最短路中包含至少 n 个点（不包括自己），则说明存在环
                if (!st[j]){
                    q.push(j);
                    st[j] = true;
                }
            }
        }
    }
    return false;
}
```

floyd 算法

```
初始化:
    for (int i = 1; i <= n; i ++ )
        for (int j = 1; j <= n; j ++ )
            if (i == j) d[i][j] = 0;
            else d[i][j] = INF;

// 算法结束后，d[a][b] 表示 a 到 b 的最短距离
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]);
}
```

最小生成树

朴素 Prim 算法

- 稠密图（m 接近于 n^2 ）

```
int n;      // n 表示点数
int g[N][N];        // 邻接矩阵，存储所有边
int dist[N];        // 存储其他点到当前最小生成树的距离
bool st[N];     // 存储每个点是否已经在生成树中
// 如果图不连通，则返回 INF(值是 0x3f3f3f3f), 否则返回最小生成树的树边权重之和
int prim(){
    memset(dist, 0x3f, sizeof dist);
    int res = 0;
    for (int i = 0; i < n; i ++ ){
        int t = -1;
        for (int j = 1; j <= n; j ++ )
            if (!st[j] && (t == -1 || dist[t] > dist[j]))
                t = j;
        if (i && dist[t] == INF) return INF;
        if (i) res += dist[t];
        st[t] = true;
        for (int j = 1; j <= n; j ++ ) dist[j] = min(dist[j], g[t][j]);
    }
    return res;
}
```

Kruskal 算法

- 实现简单，稀疏图（m 接近 n ）

```cpp
int n, m;       // n 是点数, m 是边数
int p[N];       // 并查集的父节点数组
struct Edge{    // 存储边
    int a, b, w;
    bool operator< (const Edge &W)const{
        return w < W.w;
    }
}edges[M];

int find(int x){    // 并查集核心操作
    if (p[x] != x) p[x] = find(p[x]);
    return p[x];
}

int kruskal(){
    sort(edges, edges + m);
    for (int i = 1; i <= n; i ++ ) p[i] = i;    // 初始化并查集
    int res = 0, cnt = 0;
    for (int i = 0; i < m; i ++ ){
        int a = edges[i].a, b = edges[i].b, w = edges[i].w;
        a = find(a), b = find(b);
        if (a != b){    // 如果两个连通块不连通, 则将这两个连通块合并
            p[a] = b;
            res += w;
            cnt ++ ;
        }
    }
    if (cnt < n - 1) return INF;
    return res;
}
```

拓扑排序

- 有向图
- 别忘了存储入度
- 当 `toporder(int n)` 返回值的长度不等于 n 时，不存在拓扑排序。

```cpp
const int N = 1e5+10;
vector<int> G[N];
int deg[N]; // 入度

vector<int> toporder(int n) {
    vector<int> orders;
    queue<int> q;
    for (int i = 1; i <= n; i++)
        if (!deg[i]) {
            q.push(i);
            orders.push_back(i);
        }
    while (!q.empty()) {
        int u = q.front(); q.pop();
        for (int v: G[u])
            if (!--deg[v]) {
                q.push(v);
                orders.push_back(v);
            }
    }
    return orders;
}
```

差分约束

一个系统 n 个变量和 m 个约束条件组成，每个约束条件形如 $x_j - x_i \leq b_k$。可以发现每个约束条件都形如最短路中的三角不等式 $d_u - d_v \leq w_{u,v}$。因此连一条边 (i, j, b_k) 建图。

若要判断解的存在性，使用 spfa 判断是否存在负环，有则无解。

若要使得所有量两两的值最接近，源点到各点的距离初始成 0，跑最远路。

若要使得某一变量与其他变量的差尽可能大，则源点到各点距离初始化成 ∞，跑最短路。

最近公共祖先

```
const LL N = 5e5+10, SP = log2(N)+1;
vector<int> G[N];
int pa[N][SP], dep[N];

void dfs(int u, int fa) {
    pa[u][0] = fa; dep[u] = dep[fa] + 1;
    FOR (i, 1, SP) pa[u][i] = pa[pa[u][i - 1]][i - 1];
    for (int& v: G[u]) {
        if (v == fa) continue;
        dfs(v, u);
    }
}

int lca(int u, int v) {
    if (dep[u] < dep[v]) swap(u, v);
    int t = dep[u] - dep[v];
    FOR (i, 0, SP) if (t & (1 << i)) u = pa[u][i];
    FORD (i, SP - 1, -1) {
        int uu = pa[u][i], vv = pa[v][i];
        if (uu != vv) { u = uu; v = vv; }
    }
    return u == v ? u : pa[u][0];
}
```

树链剖分

- 将树上操作转化为区间操作，套用区间数据结构
- 别忘了选一种方法（取消注释）
- `fa[N]`：存储每个节点的父节点
- `dep[N]`：存储每个节点的深度
- `idx[N]`：存储每个节点在线段树中的索引（DFS 序）
- `out[N]`：存储每个节点子树在 DFS 序中的结束位置
- `ridx[N]`：存储 DFS 序到节点的反向映射
- `sz[N]`：存储每个节点的子树大小
- `son[N]`：存储每个节点的重儿子（子树最大的儿子）
- `top[N]`：存储每个节点所在重链的顶端节点
- `clk`：DFS 序计数器
- `init()`：初始化（先建图再调用）
- `go(u, v, f)`：`f` 是一个形如 `f(int l, int r)` 的函数。对树上节点 u 到节点 v 的简单路径，分解为 dfs 序中的区间 $[l, r]$，调用函数 `f`
- 子树操作：u 的子树的 dfs 序区间为 $[idx[u], out[u]]$

```
const int N = 3e4+10;

vector<int> G[N];
int fa[N], dep[N], idx[N], out[N], ridx[N];
namespace hld {
    int sz[N], son[N], top[N], clk;
    void predfs(int u, int d) {
        dep[u] = d; sz[u] = 1;
        int& maxs = son[u] = -1;
        for (int& v: G[u]) {
            if (v == fa[u]) continue;
            fa[v] = u;
            predfs(v, d + 1);
            sz[u] += sz[v];
            if (maxs == -1 || sz[v] > sz[maxs]) maxs = v;
        }
    }
    void dfs(int u, int tp) {
        top[u] = tp; idx[u] = ++clk; ridx[clk] = u;
        if (son[u] != -1) dfs(son[u], tp);
```

```
        for (int& v: G[u])
            if (v != fa[u] && v != son[u]) dfs(v, v);
        out[u] = clk;
    }
    void init(){
        clk = 0;
        predfs(1, 1);
        dfs(1, 1);
    }
    template<typename T>
    int go(int u, int v, T&& f = [](int, int) {}) {
        int uu = top[u], vv = top[v];
        while (uu != vv) {
            if (dep[uu] < dep[vv]) { swap(uu, vv); swap(u, v); }
            f(idx[uu], idx[u]);
            u = fa[uu]; uu = top[u];
        }
        if (dep[u] < dep[v]) swap(u, v);
        // 下面两行代码选择一个
        // f(idx[v], idx[u]); // 包含 lca(u, v)
        // if (u != v) f(idx[v] + 1, idx[u]); // 不包含 lca(u, v)
        return v;
    }
    int up(int u, int d) { // 查询 u 节点向上走 d 步的节点编号
        while (d) {
            if (dep[u] - dep[top[u]] < d) {
                d -= dep[u] - dep[top[u]];
                u = top[u];
            } else return ridx[idx[u] - d];
            u = fa[u]; --d;
        }
        return u;
    }
    int finds(int u, int rt) { // 找 u 在 rt 的哪个儿子的子树中
        while (top[u] != top[rt]) {
            u = top[u];
            if (fa[u] == rt) return u;
            u = fa[u];
        }
        return ridx[idx[rt] + 1];
    }
}
```

网络流

- 最大流

```
const LL INF = LONG_LONG_MAX;

struct E {
    LL to, cp;
    E(LL to, LL cp): to(to), cp(cp) {}
};

struct Dinic {
    static const LL M = 1E5 * 5;
    LL m, s, t;
    vector<E> edges;
    vector<LL> G[M];
    LL d[M];
    LL cur[M];

    void init(LL n, LL s, LL t) {
        this->s = s; this->t = t;
        for (LL i = 0; i <= n; i++) G[i].clear();
        edges.clear(); m = 0;
    }

    void addedge(LL u, LL v, LL cap) {
        edges.emplace_back(v, cap);
        edges.emplace_back(u, 0);
```

```cpp
        G[u].push_back(m++);
        G[v].push_back(m++);
    }

    bool BFS() {
        memset(d, 0, sizeof d);
        queue<LL> Q;
        Q.push(s); d[s] = 1;
        while (!Q.empty()) {
            LL x = Q.front(); Q.pop();
            for (LL& i: G[x]) {
                E &e = edges[i];
                if (!d[e.to] && e.cp > 0) {
                    d[e.to] = d[x] + 1;
                    Q.push(e.to);
                }
            }
        }
        return d[t];
    }

    LL DFS(LL u, LL cp) {
        if (u == t || !cp) return cp;
        LL tmp = cp, f;
        for (LL& i = cur[u]; i < G[u].size(); i++) {
            E& e = edges[G[u][i]];
            if (d[u] + 1 == d[e.to]) {
                f = DFS(e.to, min(cp, e.cp));
                e.cp -= f;
                edges[G[u][i] ^ 1].cp += f;
                cp -= f;
                if (!cp) break;
            }
        }
        return tmp - cp;
    }

    LL go() {
        LL flow = 0;
        while (BFS()) {
            memset(cur, 0, sizeof cur);
            flow += DFS(s, INF);
        }
        return flow;
    }
} DC;
```

- 最小费用最大流

```cpp
const LL M = 5e4+10;
const int INF = INT_MAX;

struct E {
    int from, to, cp, v;
    E() {}
    E(int f, int t, int cp, int v) : from(f), to(t), cp(cp), v(v) {}
};

struct MCMF {
    int n, m, s, t;
    vector<E> edges;
    vector<int> G[M];
    bool inq[M];
    int d[M], p[M], a[M];

    void init(int _n, int _s, int _t) {
        n = _n; s = _s; t = _t;
        FOR (i, 0, n + 1) G[i].clear();
        edges.clear(); m = 0;
    }

    void addedge(int from, int to, int cap, int cost) {
```

```
        edges.emplace_back(from, to, cap, cost);
        edges.emplace_back(to, from, 0, -cost);
        G[from].push_back(m++);
        G[to].push_back(m++);
    }

    bool BellmanFord(int &flow, int &cost) {
        FOR (i, 0, n + 1) d[i] = INF;
        memset(inq, 0, sizeof inq);
        d[s] = 0, a[s] = INF, inq[s] = true;
        queue<int> Q; Q.push(s);
        while (!Q.empty()) {
            int u = Q.front(); Q.pop();
            inq[u] = false;
            for (int& idx: G[u]) {
                E &e = edges[idx];
                if (e.cp && d[e.to] > d[u] + e.v) {
                    d[e.to] = d[u] + e.v;
                    p[e.to] = idx;
                    a[e.to] = min(a[u], e.cp);
                    if (!inq[e.to]) {
                        Q.push(e.to);
                        inq[e.to] = true;
                    }
                }
            }
        }
        if (d[t] == INF) return false;
        flow += a[t];
        cost += a[t] * d[t];
        int u = t;
        while (u != s) {
            edges[p[u]].cp -= a[t];
            edges[p[u] ^ 1].cp += a[t];
            u = edges[p[u]].from;
        }
        return true;
    }

    pair<int, int> go() {
        int flow = 0, cost = 0;
        while (BellmanFord(flow, cost));
        return {flow, cost};
    }
} MM;
```

树上路径交

- 前置模板：最近公共祖先

```
int intersection(int x1, int y1, int x2, int y2) {
    int t[4] = {lca(x1, x2), lca(x1, y2), lca(y1, x2), lca(y1, y2)};
    int p1 = 0,p2 = 0;
    FOR(j,0,4)
    if(dep[t[j]] > dep[p1]) p2 = p1, p1 = t[j];
    else if(dep[t[j]] > dep[p2]) p2 = t[j];
    int h1 = lca(x1,y1), h2 = lca(x2,y2);
    if(p1 == p2){
        if(dep[p1] < dep[h1] || dep[p1] < dep[h2]) return 0;
        else return 1;
    }
    else{
        int ans = dep[p1]+dep[p2]-2*dep[lca(p1,p2)]+1;
        return ans;
    }
}
```

树上点分治（树的重心）

```cpp
const LL N = 2e4+10, N2 = N * 2;

int h[N], e[N2], ne[N2], idx;

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

vector<bool> vis;

// 获取子树的重心（自动处理父子关系）(如果有两个重心，输出编号小的那个)
// 若重心为 u，则 mx[u] 为以 u 为重心子树大小的最大值
int q[N], fa[N], sz[N], mx[N];
int get_rt(int u) {
    int p = 0, cur = -1;
    q[p++] = u; fa[u] = -1;
    while (++cur < p) {
        u = q[cur]; mx[u] = 0; sz[u] = 1;
        for (int i = h[u]; i!=-1; i=ne[i]){
            int j = e[i];
            if(vis[j] or j == fa[u]) continue;
            fa[q[p++] = j] = u;
        }
    }
    FORD (i, p - 1, -1) {
        u = q[i];
        mx[u] = max(mx[u], p - sz[u]);
        if (mx[u] * 2 <= p) return u;
        sz[fa[u]] += sz[u];
        mx[fa[u]] = max(mx[fa[u]], sz[u]);
    }
//  assert(0);
}

// 分治 dfs（起点任意）
void dfs(int u) {
    cout << "u: " << u;
    u = get_rt(u);
    vis[u] = true;
    // 处理子树逻辑
    cout << " centroid: " << u << '\n';
    // 如果在此处 DFS，会遍历整棵子树（if(vis[u]) return）
    // ...

    for(int i=h[u]; i!=-1; i=ne[i]){
        int j = e[i];
        if(vis[j]) continue;
        dfs(j);
    }
}
```

二分图

最大匹配

- 最小覆盖数 = 最大匹配数
- 最大独立集 = 顶点数 - 二分图匹配数
- DAG 最小路径覆盖数 = 结点数 - 拆点后二分图最大匹配数

```cpp
const int N = 500+10;

struct MaxMatch {
    int n;
    vector<int> G[N];
    int vis[N], left[N], clk;

    void init(int n) {
        this->n = n;
        FOR (i, 0, n + 1) G[i].clear();
```

```
        memset(left, -1, sizeof left);
        memset(vis, -1, sizeof vis);
    }

    bool dfs(int u) {
        for (int v: G[u])
            if (vis[v] != clk) {
                vis[v] = clk;
                if (left[v] == -1 || dfs(left[v])) {
                    left[v] = u;
                    return true;
                }
            }
        return false;
    }

    int match() {
        int ret = 0;
        for (clk = 0; clk <= n; ++clk)
            if (dfs(clk)) ++ret;
        return ret;
    }
} MM;
// !!!!!!!!!!!!!!!!!!!!!!!!!!! Tester Start !!!!!!!!!!!!!!!!!!!!!!!!!!!!
void solve(){
    LL n1, n2, m, n, i, t1, t2;
    cin >> n1 >> n2 >> m;
    n = n1 + n2;
    MM.init(n);
    for(i=0; i<m; i++){
        cin >> t1 >> t2;
        MM.G[t1].push_back(n1+t2);
    }
    cout << MM.match() << '\n';
}
```

最大权匹配

- `py[j] = i` 表示右侧顶点 j 与左侧顶点 i 匹配

```
namespace R {
    const int M = 400 + 5;
    const int INF = 2E9;
    int n;
    int w[M][M], kx[M], ky[M], py[M], vy[M], slk[M], pre[M];

    LL KM() {
        FOR (i, 1, n + 1)
        FOR (j, 1, n + 1)
        kx[i] = max(kx[i], w[i][j]);
        FOR (i, 1, n + 1) {
            fill(vy, vy + n + 1, 0);
            fill(slk, slk + n + 1, INF);
            fill(pre, pre + n + 1, 0);
            int k = 0, p = -1;
            for (py[k = 0] = i; py[k]; k = p) {
                int d = INF;
                vy[k] = 1;
                int x = py[k];
                FOR (j, 1, n + 1)
                if (!vy[j]) {
                    int t = kx[x] + ky[j] - w[x][j];
                    if (t < slk[j]) { slk[j] = t; pre[j] = k; }
                    if (slk[j] < d) { d = slk[j]; p = j; }
                }
                FOR (j, 0, n + 1)
                if (vy[j]) { kx[py[j]] -= d; ky[j] += d; }
                else slk[j] -= d;
            }
            for (; k; k = pre[k]) py[k] = py[pre[k]];
        }
```

```
        LL ans = 0;
        FOR (i, 1, n + 1) ans += kx[i] + ky[i];
        return ans;
    }
}
// !!!!!!!!!!!!!!!!!!!!!!!!!!! Tester Start !!!!!!!!!!!!!!!!!!!!!!!!!!!!
void solve(){
    LL n1, n2, i, t1, t2, t3, m, n, j;
    cin >> n1 >> n2 >> m;
    // 初始化
    n = max(n1, n2);
    R::n = n;
    for(i=0; i<=n; i++){
        for(j=0; j<=n; j++){
            R::w[i][j] = 0;
        }
    }
    // 读数据
    for(i=0; i<m; i++){
        cin >> t1 >> t2 >> t3;
        R::w[t1][t2] = t3;
    }
    // 计算
    LL maxx = R::KM();
    cout << maxx << '\n';
    // 结果转换
    vector<pair<LL, LL>> anss;
    for(i=1; i<=n; i++){ // 注意遍历最大范围
        if(R::w[R::py[i]][i]){
            anss.push_back({R::py[i], i});
        }else{
            // 未匹配
            anss.push_back({R::py[i], 0});
        }
    }
    sort(anss.begin(), anss.end());
    for(i=0; i<n1; i++){
        cout << anss[i].second << ' ';
    }
}
```

Tarjan

割点

- 判断割点（无向图）
- 注意原图可能不连通

```
int dfn[N], low[N], clk;
void init() { clk = 0; memset(dfn, 0, sizeof dfn); }
void tarjan(int u, int fa) {
    low[u] = dfn[u] = ++clk;
    int cc = fa != -1;
    for (int& v: G[u]) {
        if (v == fa) continue;
        if (!dfn[v]) {
            tarjan(v, u);
            low[u] = min(low[u], low[v]);
            cc += low[v] >= dfn[u];
        } else low[u] = min(low[u], dfn[v]);
    }
    if (cc > 1) // u 是割点
}
```

桥

- 无向图
- 注意原图不连通和重边

```cpp
int dfn[N], low[N], clk;
void init() { memset(dfn, 0, sizeof dfn); clk = 0; }
void tarjan(int u, int fa) {
    low[u] = dfn[u] = ++clk;
    int _fst = 0;
    for (E& e: G[u]) {
        int v = e.to; if (v == fa && ++_fst == 1) continue;
        if (!dfn[v]) {
            tarjan(v, u);
            if (low[v] > dfn[u]) // (u, v) 是桥
            low[u] = min(low[u], low[v]);
        } else low[u] = min(low[u], dfn[v]);
    }
}
```

强连通分量缩点

- 有向图
- B：强连通分量的数量计数器
- bl[N]：记录每个顶点所属的强连通分量编号
- bcc[N]：存储每个强连通分量包含的顶点列表

```cpp
int low[N], dfn[N], clk, B, bl[N];
vector<int> bcc[N];
void init() { B = clk = 0; memset(dfn, 0, sizeof dfn); }
void tarjan(int u) {
    static int st[N], p;
    static bool in[N];
    dfn[u] = low[u] = ++clk;
    st[p++] = u; in[u] = true;
    for (int& v: G[u]) {
        if (!dfn[v]) {
            tarjan(v);
            low[u] = min(low[u], low[v]);
        } else if (in[v]) low[u] = min(low[u], dfn[v]);
    }
    if (dfn[u] == low[u]) {
        while (1) {
            int x = st[--p]; in[x] = false;
            bl[x] = B; bcc[B].push_back(x);
            if (x == u) break;
        }
        ++B;
    }
}
```

点双连通分量 / 广义圆方树

- 数组开两倍
- 一条边也被计入点双了（适合拿来建圆方树），可以用点数 <= 边数过滤
- B：双连通分量的数量（编号从 0 开始）。
- bc[B]：存储第 B 个双连通分量包含的节点。
- be[B]：存储第 B 个双连通分量包含的边（索引）。
- bno[x]：标记节点 x 属于哪个双连通分量（用于去重）。

```cpp
struct E { int to, nxt; } e[N];
int hd[N], ecnt;
void addedge(int u, int v) {
    e[ecnt] = {v, hd[u]};
    hd[u] = ecnt++;
}
int low[N], dfn[N], clk, B, bno[N];
vector<int> bc[N], be[N];
bool vise[N];
void init() {
    memset(vise, 0, sizeof vise);
    memset(hd, -1, sizeof hd);
    memset(dfn, 0, sizeof dfn);
    memset(bno, -1, sizeof bno);
```

```
    B = clk = ecnt = 0;
}

void tarjan(int u, int feid) {
    static int st[N], p;
    static auto add = [&](int x) {
        if (bno[x] != B) { bno[x] = B; bc[B].push_back(x); }
    };
    low[u] = dfn[u] = ++clk;
    for (int i = hd[u]; ~i; i = e[i].nxt) {
        if ((feid ^ i) == 1) continue;
        if (!vise[i]) { st[p++] = i; vise[i] = vise[i ^ 1] = true; }
        int v = e[i].to;
        if (!dfn[v]) {
            tarjan(v, i);
            low[u] = min(low[u], low[v]);
            if (low[v] >= dfn[u]) {
                bc[B].clear(); be[B].clear();
                while (1) {
                    int eid = st[--p];
                    add(e[eid].to); add(e[eid ^ 1].to);
                    be[B].push_back(eid);
                    if ((eid ^ i) <= 1) break;
                }
                ++B;
            }
        } else low[u] = min(low[u], dfn[v]);
    }
}
```

计算几何

二维几何：点与向量

```
#define y1 yy1
#define nxt(i) ((i + 1) % s.size())
typedef double LD;
const LD PI = 3.14159265358979323846;
const LD eps = 1E-10;
int sgn(LD x) { return fabs(x) < eps ? 0 : (x > 0 ? 1 : -1); }
struct L;
struct P;
typedef P V;
struct P {
    LD x, y;
    explicit P(LD x = 0, LD y = 0): x(x), y(y) {}
    explicit P(const L& l);
};
struct L {
    P s, t;
    L() {}
    L(P s, P t): s(s), t(t) {}
};

P operator + (const P& a, const P& b) { return P(a.x + b.x, a.y + b.y); }
P operator - (const P& a, const P& b) { return P(a.x - b.x, a.y - b.y); }
P operator * (const P& a, LD k) { return P(a.x * k, a.y * k); }
P operator / (const P& a, LD k) { return P(a.x / k, a.y / k); }
inline bool operator < (const P& a, const P& b) {
    return sgn(a.x - b.x) < 0 || (sgn(a.x - b.x) == 0 && sgn(a.y - b.y) < 0);
}
bool operator == (const P& a, const P& b) { return !sgn(a.x - b.x) && !sgn(a.y - b.y); }
P::P(const L& l) { *this = l.t - l.s; }
ostream &operator << (ostream &os, const P &p) {
    return (os << "(" << p.x << "," << p.y << ")");
}
istream &operator >> (istream &is, P &p) {
    return (is >> p.x >> p.y);
}

```

```
LD dist(const P& p) { return sqrt(p.x * p.x + p.y * p.y); }
LD dot(const V& a, const V& b) { return a.x * b.x + a.y * b.y; }
LD det(const V& a, const V& b) { return a.x * b.y - a.y * b.x; }
LD cross(const P& s, const P& t, const P& o = P()) { return det(s - o, t - o); }
// --------------------------------------------
```

象限

```
// 象限
int quad(P p) {
    int x = sgn(p.x), y = sgn(p.y);
    if (x > 0 && y >= 0) return 1;
    if (x <= 0 && y > 0) return 2;
    if (x < 0 && y <= 0) return 3;
    if (x >= 0 && y < 0) return 4;
    assert(0);
}

// 仅适用于参照点在所有点一侧的情况
struct cmp_angle {
    P p;
    bool operator () (const P& a, const P& b) {
//        int qa = quad(a - p), qb = quad(b - p);
//        if (qa != qb) return qa < qb;
        int d = sgn(cross(a, b, p));
        if (d) return d > 0;
        return dist(a - p) < dist(b - p);
    }
};
```

线

```
// 是否平行
bool parallel(const L& a, const L& b) {
    return !sgn(det(P(a), P(b)));
}
// 直线是否相等
bool l_eq(const L& a, const L& b) {
    return parallel(a, b) && parallel(L(a.s, b.t), L(b.s, a.t));
}
// 逆时针旋转 r 弧度
P rotation(const P& p, const LD& r) { return P(p.x * cos(r) - p.y * sin(r), p.x * sin(r) + p.y * cos(r)); }
P RotateCCW90(const P& p) { return P(-p.y, p.x); }
P RotateCW90(const P& p) { return P(p.y, -p.x); }
// 单位法向量
V normal(const V& v) { return V(-v.y, v.x) / dist(v); }
```

点与线

```
// 点在线段上  <= 0 包含端点 < 0 则不包含
bool p_on_seg(const P& p, const L& seg) {
    P a = seg.s, b = seg.t;
    return !sgn(det(p - a, b - a)) && sgn(dot(p - a, p - b)) <= 0;
}
// 点到直线距离
LD dist_to_line(const P& p, const L& l) {
    return fabs(cross(l.s, l.t, p)) / dist(l);
}
// 点到线段距离（考虑端点和垂线）
LD dist_to_seg(const P& p, const L& l) {
    if (l.s == l.t) return dist(p - l);
    V vs = p - l.s, vt = p - l.t;
    if (sgn(dot(l, vs)) < 0) return dist(vs);
    else if (sgn(dot(l, vt)) > 0) return dist(vt);
    else return dist_to_line(p, l);
}
```

线与线

```
// 求直线交 需要事先保证有界 (需保证不平行)
P l_intersection(const L& a, const L& b) {
    LD s1 = det(P(a), b.s - a.s), s2 = det(P(a), b.t - a.s);
    return (b.s * s2 - b.t * s1) / (s2 - s1);
}
// 向量夹角的弧度
LD angle(const V& a, const V& b) {
    LD r = asin(fabs(det(a, b)) / dist(a) / dist(b));
    if (sgn(dot(a, b)) < 0) r = PI - r;
    return r;
}
// 线段和直线是否有交   1 = 规范 (十), 2 = 不规范 (L, T, I)
int s_l_cross(const L& seg, const L& line) {
    int d1 = sgn(cross(line.s, line.t, seg.s));
    int d2 = sgn(cross(line.s, line.t, seg.t));
    if ((d1 ^ d2) == -2) return 1; // proper
    if (d1 == 0 || d2 == 0) return 2;
    return 0;
}
// 线段的交   1 = 规范, 2 = 不规范
int s_cross(const L& a, const L& b, P& p) {
    int d1 = sgn(cross(a.t, b.s, a.s)), d2 = sgn(cross(a.t, b.t, a.s));
    int d3 = sgn(cross(b.t, a.s, b.s)), d4 = sgn(cross(b.t, a.t, b.s));
    if ((d1 ^ d2) == -2 && (d3 ^ d4) == -2) { p = l_intersection(a, b); return 1; }
    if (!d1 && p_on_seg(b.s, a)) { p = b.s; return 2; }
    if (!d2 && p_on_seg(b.t, a)) { p = b.t; return 2; }
    if (!d3 && p_on_seg(a.s, b)) { p = a.s; return 2; }
    if (!d4 && p_on_seg(a.t, b)) { p = a.t; return 2; }
    return 0;
}
```

多边形

面积、凸包

```
typedef vector<P> S;

// 点是否在多边形中 0 = 在外部 1 = 在内部 -1 = 在边界上
int inside(const S& s, const P& p) {
    int cnt = 0;
    FOR (i, 0, s.size()) {
        P a = s[i], b = s[nxt(i)];
        if (p_on_seg(p, L(a, b))) return -1;
        if (sgn(a.y - b.y) <= 0) swap(a, b);
        if (sgn(p.y - a.y) > 0) continue;
        if (sgn(p.y - b.y) <= 0) continue;
        cnt += sgn(cross(b, a, p)) > 0;
    }
    return bool(cnt & 1);
}
// 多边形面积，有向面积可能为负
LD polygon_area(const S& s) {
    LD ret = 0;
    FOR (i, 1, (LL)s.size() - 1)
        ret += cross(s[i], s[i + 1], s[0]);
    return ret / 2;
}
// 构建凸包 点不可以重复 < 0 边上可以有点， <= 0 则不能
// 会改变输入点的顺序
const int MAX_N = 1000;
S convex_hull(S& s) {
//    assert(s.size() >= 3);
    sort(s.begin(), s.end());
    S ret(MAX_N * 2);
    int sz = 0;
    FOR (i, 0, s.size()) {
        while (sz > 1 && sgn(cross(ret[sz - 1], s[i], ret[sz - 2])) < 0) --sz;
        ret[sz++] = s[i];
    }
```

```
    int k = sz;
    FORD (i, (LL)s.size() - 2, -1) {
        while (sz > k && sgn(cross(ret[sz - 1], s[i], ret[sz - 2])) < 0) --sz;
        ret[sz++] = s[i];
    }
    ret.resize(sz - (s.size() > 1));
    return ret;
}

P ComputeCentroid(const vector<P> &p) {
    P c(0, 0);
    LD scale = 6.0 * polygon_area(p);
    for (unsigned i = 0; i < p.size(); i++) {
        unsigned j = (i + 1) % p.size();
        c = c + (p[i] + p[j]) * (p[i].x * p[j].y - p[j].x * p[i].y);
    }
    return c / scale;
}
```

旋转卡壳

若干点中点对距离最大值

```
LD rotatingCalipers(vector<P>& qs) {
    int n = qs.size();
    if (n == 2)
        return dist(qs[0] - qs[1]);
    int i = 0, j = 0;
    FOR (k, 0, n) {
        if (!(qs[i] < qs[k])) i = k;
        if (qs[j] < qs[k]) j = k;
    }
    LD res = 0;
    int si = i, sj = j;
    while (i != sj || j != si) {
        res = max(res, dist(qs[i] - qs[j]));
        if (sgn(cross(qs[(i+1)%n] - qs[i], qs[(j+1)%n] - qs[j])) < 0)
            i = (i + 1) % n;
        else j = (j + 1) % n;
    }
    return res;
}

int main() {
    int n;
    while (cin >> n) {
        S v(n);
        FOR (i, 0, n) cin >> v[i].x >> v[i].y;
        convex_hull(v);
        printf("%.0f\n", rotatingCalipers(v));
    }
}
```

半平面交

左半平面交

```
struct LV {
    P p, v; LD ang;
    LV() {}
    LV(P s, P t): p(s), v(t - s) { ang = atan2(v.y, v.x); }
};  // 另一种向量表示

bool operator < (const LV &a, const LV& b) { return a.ang < b.ang; }
bool on_left(const LV& l, const P& p) { return sgn(cross(l.v, p - l.p)) >= 0; }
P l_intersection(const LV& a, const LV& b) {
    P u = a.p - b.p; LD t = cross(b.v, u) / cross(a.v, b.v);
    return a.p + a.v * t;
}

S half_plane_intersection(vector<LV>& L) {
```

```
    int n = L.size(), fi, la;
    sort(L.begin(), L.end());
    vector<P> p(n); vector<LV> q(n);
    q[fi = la = 0] = L[0];
    FOR (i, 1, n) {
        while (fi < la && !on_left(L[i], p[la - 1])) la--;
        while (fi < la && !on_left(L[i], p[fi])) fi++;
        q[++la] = L[i];
        if (sgn(cross(q[la].v, q[la - 1].v)) == 0) {
            la--;
            if (on_left(q[la], L[i].p)) q[la] = L[i];
        }
        if (fi < la) p[la - 1] = l_intersection(q[la - 1], q[la]);
    }
    while (fi < la && !on_left(q[fi], p[la - 1])) la--;
    if (la - fi <= 1) return vector<P>();
    p[la] = l_intersection(q[la], q[fi]);
    return vector<P>(p.begin() + fi, p.begin() + la + 1);
}

S convex_intersection(const vector<P> &v1, const vector<P> &v2) {
    vector<LV> h; int n = v1.size(), m = v2.size();
    FOR (i, 0, n) h.push_back(LV(v1[i], v1[(i + 1) % n]));
    FOR (i, 0, m) h.push_back(LV(v2[i], v2[(i + 1) % m]));
    return half_plane_intersection(h);
}
```

圆

```
struct C {
    P p; LD r;
    C(LD x = 0, LD y = 0, LD r = 0): p(x, y), r(r) {}
    C(P p, LD r): p(p), r(r) {}
};
```

三点求圆心

```
P compute_circle_center(P a, P b, P c) {
    b = (a + b) / 2;
    c = (a + c) / 2;
    return l_intersection({b, b + RotateCW90(a - b)}, {c , c + RotateCW90(a - c)});
}
```

圆线交点、圆圆交点

- 圆和线的交点关于圆心是顺时针的

```
vector<P> c_l_intersection(const L& l, const C& c) {
    vector<P> ret;
    P b(l), a = l.s - c.p;
    LD x = dot(b, b), y = dot(a, b), z = dot(a, a) - c.r * c.r;
    LD D = y * y - x * z;
    if (sgn(D) < 0) return ret;
    ret.push_back(c.p + a + b * (-y + sqrt(D + eps)) / x);
    if (sgn(D) > 0) ret.push_back(c.p + a + b * (-y - sqrt(D)) / x);
    return ret;
}

vector<P> c_c_intersection(C a, C b) {
    vector<P> ret;
    LD d = dist(a.p - b.p);
    if (sgn(d) == 0 || sgn(d - (a.r + b.r)) > 0 || sgn(d + min(a.r, b.r) - max(a.r, b.r)) < 0)
        return ret;
    LD x = (d * d - b.r * b.r + a.r * a.r) / (2 * d);
    LD y = sqrt(a.r * a.r - x * x);
    P v = (b.p - a.p) / d;
    ret.push_back(a.p + v * x + RotateCCW90(v) * y);
    if (sgn(y) > 0) ret.push_back(a.p + v * x - RotateCCW90(v) * y);
    return ret;
}
```

圆圆位置关系

```
// 1: 内含 2: 内切 3: 相交 4: 外切 5: 相离
int c_c_relation(const C& a, const C& v) {
    LD d = dist(a.p - v.p);
    if (sgn(d - a.r - v.r) > 0) return 5;
    if (sgn(d - a.r - v.r) == 0) return 4;
    LD l = fabs(a.r - v.r);
    if (sgn(d - l) > 0) return 3;
    if (sgn(d - l) == 0) return 2;
    if (sgn(d - l) < 0) return 1;
}
```

圆与多边形交

- HDU 5130
- 注意顺时针逆时针（可能要取绝对值）

```
LD sector_area(const P& a, const P& b, LD r) {
    LD th = atan2(a.y, a.x) - atan2(b.y, b.x);
    while (th <= 0) th += 2 * PI;
    while (th > 2 * PI) th -= 2 * PI;
    th = min(th, 2 * PI - th);
    return r * r * th / 2;
}

LD c_tri_area(P a, P b, P center, LD r) {
    a = a - center; b = b - center;
    int ina = sgn(dist(a) - r) < 0, inb = sgn(dist(b) - r) < 0;
    // dbg(a, b, ina, inb);
    if (ina && inb) {
        return fabs(cross(a, b)) / 2;
    } else {
        auto p = c_l_intersection(L(a, b), C(0, 0, r));
        if (ina ^ inb) {
            auto cr = p_on_seg(p[0], L(a, b)) ? p[0] : p[1];
            if (ina) return sector_area(b, cr, r) + fabs(cross(a, cr)) / 2;
            else return sector_area(a, cr, r) + fabs(cross(b, cr)) / 2;
        } else {
            if ((int) p.size() == 2 && p_on_seg(p[0], L(a, b))) {
                if (dist(p[0] - a) > dist(p[1] - a)) swap(p[0], p[1]);
                return sector_area(a, p[0], r) + sector_area(p[1], b, r)
                    + fabs(cross(p[0], p[1])) / 2;
            } else return sector_area(a, b, r);
        }
    }
}

typedef vector<P> S;
LD c_poly_area(S poly, const C& c) {
    LD ret = 0; int n = poly.size();
    FOR (i, 0, n) {
        int t = sgn(cross(poly[i] - c.p, poly[(i + 1) % n] - c.p));
        if (t) ret += t * c_tri_area(poly[i], poly[(i + 1) % n], c.p, c.r);
    }
    return ret;
}
```

圆的离散化、面积并

SPOJ: CIRU, EOJ: 284

- 版本 1：复杂度 $O(n^3 \log n)$。虽然常数小，但还是难以接受。
- 优点？想不出来。
- 原理上是用竖线进行切分，然后对每一个切片分别计算。
- 扫描线部分可以魔改，求各种东西。

```
inline LD rt(LD x) { return sgn(x) == 0 ? 0 : sqrt(x); }
inline LD sq(LD x) { return x * x; }

```

```
// 圆弧
// 如果按照 x 离散化，圆弧是 " 横着的"
// 记录圆弧的左端点、右端点、中点的坐标，和圆弧所在的圆
// 调用构造要保证 c.x - x.r <= xl < xr <= c.y + x.r
// t = 1 下圆弧 t = -1 上圆弧
struct CV {
    LD yl, yr, ym; C o; int type;
    CV() {}
    CV(LD yl, LD yr, LD ym, C c, int t)
        : yl(yl), yr(yr), ym(ym), type(t), o(c) {}
};

// 辅助函数 求圆上纵坐标
pair<LD, LD> c_point_eval(const C& c, LD x) {
    LD d = fabs(c.p.x - x), h = rt(sq(c.r) - sq(d));
    return {c.p.y - h, c.p.y + h};
}
// 构造上下圆弧
pair<CV, CV> pairwise_curves(const C& c, LD xl, LD xr) {
    LD yl1, yl2, yr1, yr2, ym1, ym2;
    tie(yl1, yl2) = c_point_eval(c, xl);
    tie(ym1, ym2) = c_point_eval(c, (xl + xr) / 2);
    tie(yr1, yr2) = c_point_eval(c, xr);
    return {CV(yl1, yr1, ym1, c, 1), CV(yl2, yr2, ym2, c, -1)};
}

// 离散化之后同一切片内的圆弧应该是不相交的
bool operator < (const CV& a, const CV& b) { return a.ym < b.ym; }
// 计算圆弧和连接圆弧端点的线段构成的封闭图形的面积
LD cv_area(const CV& v, LD xl, LD xr) {
    LD l = rt(sq(xr - xl) + sq(v.yr - v.yl));
    LD d = rt(sq(v.o.r) - sq(l / 2));
    LD ang = atan(l / d / 2);
    return ang * sq(v.o.r) - d * l / 2;
}

LD circle_union(const vector<C>& cs) {
    int n = cs.size();
    vector<LD> xs;
    FOR (i, 0, n) {
        xs.push_back(cs[i].p.x - cs[i].r);
        xs.push_back(cs[i].p.x);
        xs.push_back(cs[i].p.x + cs[i].r);
        FOR (j, i + 1, n) {
            auto pts = c_c_intersection(cs[i], cs[j]);
            for (auto& p: pts) xs.push_back(p.x);
        }
    }
    sort(xs.begin(), xs.end());
    xs.erase(unique(xs.begin(), xs.end(), [](LD x, LD y) { return sgn(x - y) == 0; }), xs.end());
    LD ans = 0;
    FOR (i, 0, (int) xs.size() - 1) {
        LD xl = xs[i], xr = xs[i + 1];
        vector<CV> intv;
        FOR (k, 0, n) {
            auto& c = cs[k];
            if (sgn(c.p.x - c.r - xl) <= 0 && sgn(c.p.x + c.r - xr) >= 0) {
                auto t = pairwise_curves(c, xl, xr);
                intv.push_back(t.first); intv.push_back(t.second);
            }
        }
        sort(intv.begin(), intv.end());

        vector<LD> areas(intv.size());
        FOR (i, 0, intv.size()) areas[i] = cv_area(intv[i], xl, xr);

        int cc = 0;
        FOR (i, 0, intv.size()) {
            if (cc > 0) {
                ans += (intv[i].yl - intv[i - 1].yl + intv[i].yr - intv[i - 1].yr) * (xr - xl) / 2;
                ans += intv[i - 1].type * areas[i - 1];
```

```
                ans -= intv[i].type * areas[i];
            }
            cc += intv[i].type;
        }
    }
    return ans;
}
```

- 版本 2：复杂度 $O(n^2 \log n)$。
- 原理是：认为所求部分是一个奇怪的多边形 + 若干弓形。然后对于每个圆分别求贡献的弓形，并累加多边形有向面积。
- 同样可以魔改扫描线的部分，用于求周长、至少覆盖 k 次等等。
- 内含、内切、同一个圆的情况，通常需要特殊处理。
- 下面的代码是 k 圆覆盖。

```
inline LD angle(const P& p) { return atan2(p.y, p.x); }

// 圆弧上的点
// p 是相对于圆心的坐标
// a 是在圆上的 atan2 [-PI, PI]
struct CP {
    P p; LD a; int t;
    CP() {}
    CP(P p, LD a, int t): p(p), a(a), t(t) {}
};
bool operator < (const CP& u, const CP& v) { return u.a < v.a; }
LD cv_area(LD r, const CP& q1, const CP& q2) {
    return (r * r * (q2.a - q1.a) - cross(q1.p, q2.p)) / 2;
}

LD ans[N];
void circle_union(const vector<C>& cs) {
    int n = cs.size();
    FOR (i, 0, n) {
        // 有相同的圆的话只考虑第一次出现
        bool ok = true;
        FOR (j, 0, i)
            if (sgn(cs[i].r - cs[j].r) == 0 && cs[i].p == cs[j].p) {
                ok = false;
                break;
            }
        if (!ok) continue;
        auto& c = cs[i];
        vector<CP> ev;
        int belong_to = 0;
        P bound = c.p + P(-c.r, 0);
        ev.emplace_back(bound, -PI, 0);
        ev.emplace_back(bound, PI, 0);
        FOR (j, 0, n) {
            if (i == j) continue;
            if (c_c_relation(c, cs[j]) <= 2) {
                if (sgn(cs[j].r - c.r) >= 0) // 完全被另一个圆包含，等于说叠了一层
                    belong_to++;
                continue;
            }
            auto its = c_c_intersection(c, cs[j]);
            if (its.size() == 2) {
                P p = its[1] - c.p, q = its[0] - c.p;
                LD a = angle(p), b = angle(q);
                if (sgn(a - b) > 0) {
                    ev.emplace_back(p, a, 1);
                    ev.emplace_back(bound, PI, -1);
                    ev.emplace_back(bound, -PI, 1);
                    ev.emplace_back(q, b, -1);
                } else {
                    ev.emplace_back(p, a, 1);
                    ev.emplace_back(q, b, -1);
                }
            }
        }
        sort(ev.begin(), ev.end());
        int cc = ev[0].t;
```

```
        FOR (j, 1, ev.size()) {
            int t = cc + belong_to;
            ans[t] += cross(ev[j - 1].p + c.p, ev[j].p + c.p) / 2;
            ans[t] += cv_area(c.r, ev[j - 1], ev[j]);
            cc += ev[j].t;
        }
    }
}
```

最小圆覆盖

- 随机增量。期望复杂度 $O(n)$。

```
P compute_circle_center(P a, P b) { return (a + b) / 2; }
bool p_in_circle(const P& p, const C& c) {
    return sgn(dist(p - c.p) - c.r) <= 0;
}
C min_circle_cover(const vector<P> &in) {
    vector<P> a(in.begin(), in.end());
    dbg(a.size());
    random_shuffle(a.begin(), a.end());
    P c = a[0]; LD r = 0; int n = a.size();
    FOR (i, 1, n) if (!p_in_circle(a[i], {c, r})) {
        c = a[i]; r = 0;
        FOR (j, 0, i) if (!p_in_circle(a[j], {c, r})) {
            c = compute_circle_center(a[i], a[j]);
            r = dist(a[j] - c);
            FOR (k, 0, j) if (!p_in_circle(a[k], {c, r})) {
                c = compute_circle_center(a[i], a[j], a[k]);
                r = dist(a[k] - c);
            }
        }
    }
    return {c, r};
}
```

圆的反演

```
C inv(C c, const P& o) {
    LD d = dist(c.p - o);
    assert(sgn(d) != 0);
    LD a = 1 / (d - c.r);
    LD b = 1 / (d + c.r);
    c.r = (a - b) / 2 * R2;
    c.p = o + (c.p - o) * ((a + b) * R2 / 2 / d);
    return c;
}
```

三维计算几何

```
struct P;
struct L;
typedef P V;

struct P {
    LD x, y, z;
    explicit P(LD x = 0, LD y = 0, LD z = 0): x(x), y(y), z(z) {}
    explicit P(const L& l);
};

struct L {
    P s, t;
    L() {}
    L(P s, P t): s(s), t(t) {}
};

struct F {
    P a, b, c;
    F() {}
    F(P a, P b, P c): a(a), b(b), c(c) {}
```

```
};

P operator + (const P& a, const P& b) { return P(a.x + b.x, a.y + b.y, a.z + b.z); }
P operator - (const P& a, const P& b) { return P(a.x - b.x, a.y - b.y, a.z - b.z); }
P operator * (const P& a, LD k) { return P(a.x * k, a.y * k, a.z * k); }
P operator / (const P& a, LD k) { return P(a.x / k, a.y / k, a.z / k); }
inline int operator < (const P& a, const P& b) {
    return sgn(a.x - b.x) < 0 || (sgn(a.x - b.x) == 0 && (sgn(a.y - b.y) < 0 ||
                                  (sgn(a.y - b.y) == 0 && sgn(a.z - b.z) < 0)));
}
bool operator == (const P& a, const P& b) { return !sgn(a.x - b.x) && !sgn(a.y - b.y) && !sgn(a.z - b.z); }
P::P(const L& l) { *this = l.t - l.s; }
ostream &operator << (ostream &os, const P &p) {
    return (os << "(" << p.x << "," << p.y << "," << p.z << ")");
}
istream &operator >> (istream &is, P &p) {
    return (is >> p.x >> p.y >> p.z);
}

// --------------------------------------------
LD dist2(const P& p) { return p.x * p.x + p.y * p.y + p.z * p.z; }
LD dist(const P& p) { return sqrt(dist2(p)); }
LD dot(const V& a, const V& b) { return a.x * b.x + a.y * b.y + a.z * b.z; }
P cross(const P& v, const P& w) {
    return P(v.y * w.z - v.z * w.y, v.z * w.x - v.x * w.z, v.x * w.y - v.y * w.x);
}
LD mix(const V& a, const V& b, const V& c) { return dot(a, cross(b, c)); } // 混合积
```

旋转

```
// 逆时针旋转 r 弧度
// axis = 0 绕 x 轴
// axis = 1 绕 y 轴
// axis = 2 绕 z 轴
P rotation(const P& p, const LD& r, int axis = 0) {
    if (axis == 0)
        return P(p.x, p.y * cos(r) - p.z * sin(r), p.y * sin(r) + p.z * cos(r));
    else if (axis == 1)
        return P(p.z * cos(r) - p.x * sin(r), p.y, p.z * sin(r) + p.x * cos(r));
    else if (axis == 2)
        return P(p.x * cos(r) - p.y * sin(r), p.x * sin(r) + p.y * cos(r), p.z);
}
// n 是单位向量 表示旋转轴
// 模板是顺时针的
P rotation(const P& p, const LD& r, const P& n) {
    LD c = cos(r), s = sin(r), x = n.x, y = n.y, z = n.z;
    // dbg(c, s);
    return P((x * x * (1 - c) + c) * p.x + (x * y * (1 - c) + z * s) * p.y + (x * z * (1 - c) - y * s) * p.z,
             (x * y * (1 - c) - z * s) * p.x + (y * y * (1 - c) + c) * p.y + (y * z * (1 - c) + x * s) * p.z,
             (x * z * (1 - c) + y * s) * p.x + (y * z * (1 - c) - x * s) * p.y + (z * z * (1 - c) + c) * p.z);
}
```

线、面

函数相互依赖，所以交织在一起了。

```
// 点在线段上  <= 0 包含端点 < 0 则不包含
bool p_on_seg(const P& p, const L& seg) {
    P a = seg.s, b = seg.t;
    return !sgn(dist2(cross(p - a, b - a))) && sgn(dot(p - a, p - b)) <= 0;
}
// 点到直线距离
LD dist_to_line(const P& p, const L& l) {
    return dist(cross(l.s - p, l.t - p)) / dist(l);
}
// 点到线段距离
LD dist_to_seg(const P& p, const L& l) {
    if (l.s == l.t) return dist(p - l.s);
    V vs = p - l.s, vt = p - l.t;
    if (sgn(dot(l, vs)) < 0) return dist(vs);
    else if (sgn(dot(l, vt)) > 0) return dist(vt);
```

```
    else return dist_to_line(p, l);
}

P norm(const F& f) { return cross(f.a - f.b, f.b - f.c); }
int p_on_plane(const F& f, const P& p) { return sgn(dot(norm(f), p - f.a)) == 0; }

// 判两点在线段异侧 点在线段上返回 0 不共面无意义
int opposite_side(const P& u, const P& v, const L& l) {
    return sgn(dot(cross(P(l), u - l.s), cross(P(l), v - l.s))) < 0;
}

bool parallel(const L& a, const L& b) { return !sgn(dist2(cross(P(a), P(b)))); }
// 线段相交
int s_intersect(const L& u, const L& v) {
    return p_on_plane(F(u.s, u.t, v.s), v.t) &&
           opposite_side(u.s, u.t, v) &&
           opposite_side(v.s, v.t, u);
}
```

凸包

增量法。先将所有的点打乱顺序，然后选择四个不共面的点组成一个四面体，如果找不到说明凸包不存在。然后遍历剩余的点，不断更新凸包。对遍历到的点做如下处理。

1. 如果点在凸包内，则不更新。
2. 如果点在凸包外，那么找到所有原凸包上所有分隔了对于这个点可见面和不可见面的边，以这样的边的两个点和新的点创建新的面加入凸包中。

```

struct FT {
    int a, b, c;
    FT() { }
    FT(int a, int b, int c) : a(a), b(b), c(c) { }
};

bool p_on_line(const P& p, const L& l) {
    return !sgn(dist2(cross(p - l.s, P(l))));
}

vector<F> convex_hull(vector<P> &p) {
    sort(p.begin(), p.end());
    p.erase(unique(p.begin(), p.end()), p.end());
    random_shuffle(p.begin(), p.end());
    vector<FT> face;
    FOR (i, 2, p.size()) {
        if (p_on_line(p[i], L(p[0], p[1]))) continue;
        swap(p[i], p[2]);
        FOR (j, i + 1, p.size())
            if (sgn(mix(p[1] - p[0], p[2] - p[1], p[j] - p[0]))) {
                swap(p[j], p[3]);
                face.emplace_back(0, 1, 2);
                face.emplace_back(0, 2, 1);
                goto found;
            }
    }
found:
    vector<vector<int>> mk(p.size(), vector<int>(p.size()));
    FOR (v, 3, p.size()) {
        vector<FT> tmp;
        FOR (i, 0, face.size()) {
            int a = face[i].a, b = face[i].b, c = face[i].c;
            if (sgn(mix(p[a] - p[v], p[b] - p[v], p[c] - p[v])) < 0) {
                mk[a][b] = mk[b][a] = v;
                mk[b][c] = mk[c][b] = v;
                mk[c][a] = mk[a][c] = v;
            } else tmp.push_back(face[i]);
        }
        face = tmp;
        FOR (i, 0, tmp.size()) {
            int a = face[i].a, b = face[i].b, c = face[i].c;
```

```
            if (mk[a][b] == v) face.emplace_back(b, a, v);
            if (mk[b][c] == v) face.emplace_back(c, b, v);
            if (mk[c][a] == v) face.emplace_back(a, c, v);
        }
    }
    vector<F> out;
    FOR (i, 0, face.size())
        out.emplace_back(p[face[i].a], p[face[i].b], p[face[i].c]);
    return out;
}
```

距离

- 欧氏距离

$$|AB| = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

- 曼哈顿距离

$$d(A, B) = |x_1 - x_2| + |y_1 - y_2|$$

便于求一个点任意一点到其他所有点的距离之和。

- 切比雪夫距离

$$d(A, B) = \max(|x_1 - x_2|, |y_1 - y_2|)$$

便于求任意两点间距离的最值。

- 距离转化

假设 $A(x_1, y_1), B(x_2, y_2)$,

- A, B 两点的曼哈顿距离为 $(x_1 + y_1, x_1 - y_1), (x_2 + y_2, x_2 - y_2)$ 两点之间的切比雪夫距离。
- A, B 两点的切比雪夫距离为 $(\frac{x_1 + y_1}{2}, \frac{x_1 - y_1}{2}), (\frac{x_2 + y_2}{2}, \frac{x_2 - y_2}{2})$ 两点之间的曼哈顿距离。

- 距离之和

```
sumx[0] = 0;
sumy[0] = 0;
LL i, tx, ty;
cin >> n;
for(i=1; i<=n; i++){
    cin >> tx >> ty;
    // 求曼哈顿距离之和
    x[i] = hx[i] = tx;
    y[i] = hy[i] = ty;
    // 求切比雪夫距离之和
    x[i] = hx[i] = tx + ty;
    y[i] = hy[i] = tx - ty;
}
sort(hx+1, hx+1+n);
sort(hy+1, hy+1+n);
for(i=1; i<=n; i++){
    sumx[i] = sumx[i-1] + hx[i];
    sumy[i] = sumy[i-1] + hy[i];
}

LL calc_sum(LL i){
    LL xi = lower_bound(hx+1, hx+1+n, x[i]) - hx;
    LL yi = lower_bound(hy+1, hy+1+n, y[i]) - hy;
    return xi * x[i] - sumx[xi] + sumx[n] - sumx[xi] - (n-xi) * x[i]
    + yi * y[i] - sumy[yi] + sumy[n] - sumy[yi] - (n-yi) * y[i];
}

```

```
// 求 i 点与其他所有点曼哈顿距离之和
calc_sum(i);
// 求 i 点与其他所有点切比雪夫距离之和
calc_sum(i) / 2;
```

字符串

最小表示法

- 寻找一个字符串的循环同构串中最小的那一个，输出偏移量

```
int min_string(string s){
    int k = 0, i = 0, j = 1, n = s.length();
    while (k < n && i < n && j < n) {
        if (s[(i + k) % n] == s[(j + k) % n]) {
            k++;
        } else {
            s[(i + k) % n] > s[(j + k) % n] ? i = i + k + 1 : j = j + k + 1;
            if (i == j) i++;
            k = 0;
        }
    }
    return min(i, j);
}
```

字符串哈希

```
// 双值哈希开关
#define ENABLE_DOUBLE_HASH

typedef long long LL;
typedef unsigned long long ULL;

const int x = 135;
const int N = 4e5 + 10;
const int p1 = 1e9 + 7, p2 = 1e9 + 9;
ULL xp1[N], xp2[N], xp[N];

void init_xp() {
    xp1[0] = xp2[0] = xp[0] = 1;
    for (int i = 1; i < N; ++i) {
        xp1[i] = xp1[i - 1] * x % p1;
        xp2[i] = xp2[i - 1] * x % p2;
        xp[i] = xp[i - 1] * x;
    }
}

struct String {
    string s;
    int length, subsize;
    bool sorted;
    ULL h[N], hl[N];

    // 预处理并返回全串哈希 O(n)
    ULL hash() {
        length = s.length();
        ULL res1 = 0, res2 = 0;
        h[length] = 0;  // ATTENTION!
        for (int j = length - 1; j >= 0; --j) {
#ifdef ENABLE_DOUBLE_HASH
            res1 = (res1 * x + s[j]) % p1;
            res2 = (res2 * x + s[j]) % p2;
            h[j] = (res1 << 32) | res2;
#else
            res1 = res1 * x + s[j];
            h[j] = res1;
#endif
            // printf("%llu\n", h[j]);
        }
```

```cpp
        return h[0];
    }

    // 获取子串哈希，左闭右开区间 O(1)
    ULL get_substring_hash(int left, int right) const {
        int len = right - left;
#ifdef ENABLE_DOUBLE_HASH
        // get hash of s[left...right-1]
        unsigned int mask32 = ~(0u);
        ULL left1 = h[left] >> 32, right1 = h[right] >> 32;
        ULL left2 = h[left] & mask32, right2 = h[right] & mask32;
        return (((left1 - right1 * xp1[len] % p1 + p1) % p1) << 32) |
        (((left2 - right2 * xp2[len] % p2 + p2) % p2));
#else
        return h[left] - h[right] * xp[len];
#endif
    }

    void get_all_subs_hash(int sublen) {
        subsize = length - sublen + 1;
        for (int i = 0; i < subsize; ++i)
            hl[i] = get_substring_hash(i, i + sublen);
        sorted = 0;
    }

    void sort_substring_hash() {
        sort(hl, hl + subsize);
        sorted = 1;
    }

    bool match(ULL key) const {
//      if (!sorted) assert (0);
        if (!subsize) return false;
        return binary_search(hl, hl + subsize, key);
    }

    void init(string t) {
        length = t.length();
        s = t;
    }
};

String S, T; // 栈溢出

// 验证 S 中长度为 ans 的子串是否都存在于 T 中（是 0 否 1）
int check(String &S, String &T, int ans) {
    if (T.length < ans) return 1;
    T.get_all_subs_hash(ans); T.sort_substring_hash();
    for (int i = 0; i < S.length - ans + 1; ++i)
        if (!T.match(S.get_substring_hash(i, i + ans)))
            return 1;
    return 0;
}

// 返回是否匹配
bool match_once(String &S, String &T){
    S.get_all_subs_hash(T.length);
    S.sort_substring_hash();
    return S.match(T.get_substring_hash(0, T.length));
}

// 返回匹配下标
vector<int> match_any(const String &text, const String &pattern) {
    vector<int> positions;
    int n = text.length;
    int m = pattern.length;

    if (m == 0 || m > n) return positions;

    ULL pattern_hash = pattern.get_substring_hash(0, m);

```

```cpp
    for (int i = 0; i <= n - m; ++i) {
        ULL text_sub_hash = text.get_substring_hash(i, i + m);
        if (text_sub_hash == pattern_hash) {
            positions.push_back(i);
        }
    }
    return positions;
}

// 最长公共前缀 a[ai...] == b[bi...]
int LCP(const String &a, const String &b, int ai, int bi) {
    int l = 0, r = min(a.length - ai, b.length - bi);
    while (l < r) {
        int mid = (l + r + 1) / 2;
        if (a.get_substring_hash(ai, ai + mid) == b.get_substring_hash(bi, bi + mid))
            l = mid;
        else r = mid - 1;
    }
    return l;
}

// --------------------------- Template End ----------------------------
// !!!!!!!!!!!!!!!!!!!!!!!!!!! Tester Start !!!!!!!!!!!!!!!!!!!!!!!!!!!!

void solve(){
//  cout << "AA\n";
    init_xp();  // DON'T FORGET TO DO THIS!
//  cout << "BB\n";
    string s, t;
    cin >> s >> t;
    S.init(s), T.init(t);
    S.hash(), T.hash();
    cout << match_once(S, T) << '\n';

    vector<int> v = match_any(S, T);
    for(int ii: v) cout << ii << ' ';
    cout << '\n';

    cout << "LCP:" << LCP(S, T, 0, 0) << '\n';

    // S 中所有长度为 l 的子串均在 T 中出现, 且 l 最大
    LL l=0, r=S.length;
    while (l < r){
        int mid = l + r + 1 >> 1;
        if (!check(S, T, mid)) l = mid;
        else r = mid - 1;
    }
    cout << "check: " << l << '\n';
}
```

杂项

日期

```cpp
string day_of_week[] = {"Mo", "Tu", "We", "Th", "Fr", "Sa", "Su"};

// 格里高利历 (yyyy-mm-dd) 转儒略历 (整型/天)
int date_to_int(int y, int m, int d){
    return
    1461 * (y + 4800 + (m - 14) / 12) / 4 +
    367 * (m - 2 - (m - 14) / 12 * 12) / 12 -
    3 * ((y + 4900 + (m - 14) / 12) / 100) / 4 +
    d - 32075;
}

// 儒略历转格里高利历
void int_to_date(int jd, int &y, int &m, int &d){
    int x, n, i, j;
    x = jd + 68569;
```

```
    n = 4 * x / 146097;
    x -= (146097 * n + 3) / 4;
    i = (4000 * (x + 1)) / 1461001;
    x -= 1461 * i / 4 - 31;
    j = 80 * x / 2447;
    d = x - 2447 * j / 80;
    x = j / 11;
    m = j + 2 - 12 * x;
    y = 100 * (n - 49) + i + x;
}
```

随机

随机素数表

NTT 素数表

$p = r2^k + 1$，原根是 g。

$ (MOD, G, K, C) $ 满足：MOD 是质数，G 是 MOD 的原根，$MOD - 1 = C \times 2^K$

挑选方法:

- MOD 大于系数最大值的平方乘以多项式长度
- $2^m \leq 2^K$，其中 2^m 为多项式长度

3, 1, 1, 2; 5, 1, 2, 2; 17, 1, 4, 3; 97, 3, 5, 5; 193, 3, 6, 5; 257, 1, 8, 3; 7681, 15, 9, 17; 12289, 3, 12, 11; 40961, 5, 13, 3; 65537, 1, 16, 3; 786433, 3, 18, 10; 5767169, 11, 19, 3; 7340033, 7, 20, 3; 23068673, 11, 21, 3; 104857601, 25, 22, 3; 167772161, 5, 25, 3; 469762049, 7, 26, 3; 1004535809, 479, 21, 3; 2013265921, 15, 27, 31; 2281701377, 17, 27, 3; 3221225473, 3, 30, 5; 75161927681, 35, 31, 3; 77309411329, 9, 33, 7; 206158430209, 3, 36, 22; 2061584302081, 15, 37, 7; 2748779069441, 5, 39, 3; 6597069766657, 3, 41, 5; 39582418599937, 9, 42, 5; 79164837199873, 9, 43, 5; 263882790666241, 15, 44, 7; 1231453023109121, 35, 45, 3; 1337006139375617, 19, 46, 3; 3799912185593857, 27, 47, 5; 4222124650659841, 15, 48, 19; 7881299347898369, 7, 50, 6; 31525197391593473, 7, 52, 3; 180143985094819841, 5, 55, 6; 1945555039024054273, 27, 56, 5; 4179340454199820289, 29, 57, 3.

根号分治

二维可交换操作，将时间复杂度较低的操作分配给规模较大的维度。

注意事项

- `1LL << k`
- `(LL)v.size()`
- 输入要读完
- 不要把 `while` 写成 `if`
- 树链剖分/dfs 序，初始化或者询问不要忘记 idx, ridx
- 想清楚到底是要 `multiset` 还是 `set`
- 数据结构注意数组大小（2 倍，4 倍）
- 模意义下不要用除法
- 998244353
- 取模取全