Standard Code Library

ONGLU

North Eastern University

August 2021

Contents

初始化

数据结构

ST 表

二维哈希

```
ull hs[109][109], pw1[10009], pw2[100009];
ull gethash(int lx, int ly, int rx, int ry) {
    ull hs1 = hs[lx][ly] - pw2[ry - ly + 1] * hs[lx][ry + 1];
    ull hs2 = hs[rx + 1][ly] - pw2[ry - ly + 1] * hs[rx + 1][ry + 1];
    return hs1 - pw1[rx - lx + 1] * hs2;
}
pw1[0] = pw2[0] = 1;
for(int i = 1; i <= 1000; i++) pw1[i] = pw1[i - 1] * 19260817;
for(int i = 1; i <= 1000; i++) pw2[i] = pw2[i - 1] * 135;
for(int i = n; i >= 1; i--) {
    for(int j = 1; j <= m; j++) {
        if(i == n) hs[i][j] = sum[i][j] + 2;
        else hs[i][j] = hs[i + 1][j] * 19260817 + sum[i][j] + 2;
    }
}
for(int i = 1; i <= n; i++) {
    for(int j = m - 1; j; j--) {
        hs[i][j] = hs[i][j + 1] * 135 + hs[i][j];
    }
}
```

轻重链剖分

```
void dfs1(int x, int pre) {
    siz[x] = 1; mson[x] = 0;
    dth[x] = dth[pre] + 1;
    fa[x] = pre;
    for(auto y : son[x]) if(y != pre) {
        dfs1(y, x);
        siz[x] += siz[y];
        if(!mson[x] || siz[y] > siz[mson[x]])
            mson[x] = y;
    }
}
void dfs2(int x, int pre, int ntp) {
    id[x] = ++idcnt;
    ltp[x] = ntp;
    if(mson[x]) dfs2(mson[x], x, ntp);
    for(auto y : son[x]) {
        if(y == mson[x] || y == pre) continue;
        dfs2(y, x, y);
    }
}
void link_modify(int x, int y, int z) {
    z %= mod;
    while(ltp[x] != ltp[y]) {
        dth[ltp[x]] < dth[ltp[y]] && (x ^= y ^= x ^= y);
        modify(1, n, id[ltp[x]], id[x], 1, z);
        x = fa[ltp[x]];

    }
    dth[x] < dth[y] && (x ^= y ^= x ^= y);
    modify(1, n, id[y], id[x], 1, z);
}
int link_query(int x, int y) {
    int ans = 0;
    while(ltp[x] != ltp[y]) {
        dth[ltp[x]] < dth[ltp[y]] && (x ^= y ^= x ^= y);
        ans = (1ll * ans + query(1, n, id[ltp[x]], id[x], 1)) % mod;
        x = fa[ltp[x]];
    }
```

```
    dth[x] < dth[y] && (x ^= y ^= x ^= y);
    ans = (1ll * ans + query(1, n, id[y], id[x], 1)) % mod;
    return ans;
}
```

线段树合并

搞个动态开点线段树出来

```
#define mval(x) tree[x].mval
#define mpos(x) tree[x].mpos
#define lson(x) tree[x].lson
#define rson(x) tree[x].rson
struct node {
    int mpos, mval, lson, rson;
} tree[N * 50];
void update(int rt) {
    if(mval(lson(rt)) >= mval(rson(rt))) {
        mval(rt) = mval(lson(rt));
        mpos(rt) = mpos(lson(rt));
    } else {
        mval(rt) = mval(rson(rt));
        mpos(rt) = mpos(rson(rt));

    }
}
void modify(int l, int r, int x, int v, int &rt) {
    if(!rt) rt = ++idtot;
    if(l == r) {
        mval(rt) += v;
        mpos(rt) = l;
        return ;
    }
    if(x <= Mid) modify(l, Mid, x, v, lson(rt));
    else modify(Mid + 1, r, x, v, rson(rt));
    update(rt);
}
int merge(int l, int r, int rt1, int rt2) {
    if(!rt1 || !rt2) return rt1 + rt2;
    if(l == r) {
        mval(rt1) += mval(rt2);
        mpos(rt1) = l;
        return rt1;
    }
    lson(rt1) = merge(l, Mid, lson(rt1), lson(rt2));
    rson(rt1) = merge(Mid + 1, r, rson(rt1), rson(rt2));
    update(rt1);
    return rt1;
}
```

二维树状数组

- 矩阵修改，矩阵查询

  查询前缀和公式:

  令 $d[i][j]$ 为差分数组，定义 $d[i][j] = a[i][j] - (a[i-1][j] - a[i][j-1] - a[i-1][j])$

  $\sum_{i=1}^{x}\sum_{j-1}^{y} a[i][j] = (x+1)*(y+1)*d[i][j] - (y+1)*i*d[i][j] + d[i][j]*i*j$

```
void modify(int x, int y, int v) {
    for(int rx = x; rx <= n; rx += rx & -rx) {
        for(int ry = y; ry <= m; ry += ry & -ry) {
            tree[rx][ry][0] += v;
            tree[rx][ry][1] += v * x;
            tree[rx][ry][2] += v * y;
            tree[rx][ry][3] += v * x * y;
        }
    }
}
```

```cpp
void range_modify(int x, int y, int xx, int yy, int v) {
    modify(xx + 1, yy + 1, v);
    modify(x, yy + 1, -v);
    modify(xx + 1, y, -v);
    modify(x, y, v);
}
int query(int x, int y) {
    int ans = 0;
    for(int rx = x; rx; rx -= rx & -rx) {
        for(int ry = y; ry; ry -= ry & -ry) {
            ans += (x + 1) * (y + 1) * tree[rx][ry][0]
            - tree[rx][ry][1] * (y + 1) - tree[rx][ry][2] * (x + 1)
            + tree[rx][ry][3];
        }
    }
    return ans;
}
int range_query(int x, int y, int xx, int yy) {
    return query(xx, yy) + query(x - 1, y - 1)
        - query(x - 1, yy) - query(xx, y - 1);
}
```

平衡树

- luogu P3369 【模板】普通平衡树

```cpp
#define val(x) tree[x].val
#define cnt(x) tree[x].cnt
#define siz(x) tree[x].siz
#define fa(x) tree[x].fa
#define son(x, k) tree[x].ch[k]
struct Tree {
    struct node {
        int val, cnt, siz, fa, ch[2];
    } tree[N];
    int root, tot;
    int chk(int x) {
        return son(fa(x), 1) == x;
    }
    void update(int x) {
        siz(x) = siz(son(x, 0)) + siz(son(x, 1)) + cnt(x);
    }
    void rotate(int x) {
        int y = fa(x), z = fa(y), k = chk(x), w = son(x, k ^ 1);
        son(y, k) = w; fa(w) = y;
        son(z, chk(y)) = x; fa(x) = z;
        son(x, k ^ 1) = y; fa(y) = x;
        update(y); update(x);
    }
    void splay(int x, int goal = 0) {
        while(fa(x) != goal) {
            int y = fa(x), z = fa(y);
            if(z != goal) {
                //双旋
                if(chk(y) == chk(x)) rotate(y);
                else rotate(x);
            }
            rotate(x);
        }
        if(!goal) root = x;
    }
    int New(int x, int pre) {
        tot++;
        if(pre) son(pre, x > val(pre)) = tot;
        val(tot) = x; fa(tot) = pre;
        siz(tot) = cnt(tot) = 1;
        son(tot, 0) = son(tot, 1) = 0;
        return tot;
    }
```

```cpp
    void Insert(int x) {
        int cur = root, p = 0;
        while(cur && val(cur) != x) {
            p = cur;
            cur = son(cur, x > val(cur));
        }
        if(cur) cnt(cur)++;
        else cur = New(x, p);
        splay(cur);
    }
    void Find(int x) {
        if(!root) return ;
        int cur = root;
        while(val(cur) != x && son(cur, x > val(cur)))
            cur = son(cur, x > val(cur));
        splay(cur);
    }
    int Pre(int x) {
        Find(x);
        if(val(root) < x) return root;
        int cur = son(root, 0);
        while(son(cur, 1))
            cur = son(cur, 1);
        return cur;
    }
    int Succ(int x) {
        Find(x);
        if(val(root) > x) return root;
        int cur = son(root, 1);
        while(son(cur, 0))
            cur = son(cur, 0);
        return cur;
    }
    void Del(int x) {
        int lst = Pre(x), nxt = Succ(x);
        splay(lst); splay(nxt, lst);
        int cur = son(nxt, 0);
        if(cnt(cur) > 1) cnt(cur)--, splay(cur);
        else son(nxt, 0) = 0, splay(nxt);
    }
    int Kth(int k) {
        int cur = root;
        while(1) {
            if(son(cur, 0) && siz(son(cur, 0)) >= k) cur = son(cur, 0);
            else if(siz(son(cur, 0)) + cnt(cur) >= k) return cur;
            else k -= siz(son(cur, 0)) + cnt(cur), cur = son(cur, 1);
        }
    }
} T;
```

K-D Tree

用方差最大的那一维坐标作为当前的划分点集，然后选取该维度的中位数点划分成左右两个点集。

```cpp
#include <bits/stdc++.h>
#define pt(x) cout << x << endl;
#define Mid ((l + r) / 2)
#define low(x, k) tree[x].low[k]
#define high(x, k) tree[x].high[k]
#define lson(x) tree[x].lson
#define rson(x) tree[x].rson
using namespace std;
int read() {
    char c; int num, f = 1;
    while(c = getchar(),!isdigit(c)) if(c == '-') f = -1; num = c - '0';
    while(c = getchar(), isdigit(c)) num = num * 10 + c - '0';
    return f * num;
}
const int N = 5e5 + 1009;

namespace KD_Tree{
```

```cpp

    const int dimension = 2;
    struct node {
        int lson, rson;
        int low[dimension], high[dimension];
    } tree[N];
    struct Point {
        int id;
        int v[dimension];
    } p[N];
    void update(int rt) {
        for(int i = 0; i < dimension; i++) {
            low(rt, i) = high(rt, i) = p[rt].v[i];
            if(lson(rt)) {
                low(rt, i) = min(low(rt, i), low(lson(rt), i));
                high(rt, i) = max(high(rt, i), high(lson(rt), i));
            }
            if(rson(rt)) {
                low(rt, i) = min(low(rt, i), low(rson(rt), i));
                high(rt, i) = max(high(rt, i), high(rson(rt), i));
            }

        }
    }
    int build(int l, int r) {
        if(l > r) return 0;
        double av[dimension] = {0};
        double va[dimension] = {0};
        for(int i = 0; i < dimension; i++)
            low(Mid, i) = high(Mid, i) = p[Mid].v[i];
        for(int i = l; i <= r; i++)
            for(int j = 0; j < dimension; j++)
                av[j] += p[i].v[j];
        for(int i = 0; i < dimension; i++)
            av[i] /= (double) (r - l + 1);
        for(int i = l; i <= r; i++)
            for(int j = 0; j < dimension; j++)
                va[j] += (p[i].v[j] - av[j]) * (p[i].v[j] - av[j]);
        int maxdi = 0;
        for(int i = 1; i < dimension; i++)
            if(va[i] > va[maxdi])
                maxdi = i;
        nth_element(p + l, p + Mid, p + 1 + r, [maxdi](const Point &a, const Point &b)  -> int{return a.v[maxdi] <
    ↪ b.v[maxdi];});
        lson(Mid) = build(l, Mid - 1);
        rson(Mid) = build(Mid + 1, r);
        update(Mid);
        return Mid;
    }
    int isIn(const Point &a, const Point &ld, const Point &ru) {
        for(int i = 0; i < dimension; i++)
            if(a.v[i] < ld.v[i] || a.v[i] > ru.v[i])
                return false;
        return true;
    }
    void debug(int rt, int l, int r) {
        if(l > r) return ;
        printf("%d\n", p[rt].id);
        debug(lson(rt), l, Mid - 1);
        debug(rson(rt), Mid + 1, r);

    }
    //只能处理二维
    void getNodeset(int rt, int l, int r, vector<int> &v, const Point &ld, const Point &ru) {
        if(l > r) return ;
        for(int i = 0; i < dimension; i++) {
            if(low(rt, i) > ru.v[i] || high(rt, i) < ld.v[i]) {
                return ;
            }
        }
        if(isIn(p[Mid], ld, ru))
```

```
            v.push_back(p[Mid].id);
        getNodeset(lson(rt), l, Mid - 1, v, ld, ru);
        getNodeset(rson(rt), Mid + 1, r, v, ld, ru);
    }
}
using namespace KD_Tree;
int n, q, root;
signed main()
{
    n = read();
    for(int i = 1; i <= n; i++) {
        p[i].v[0] = read();
        p[i].v[1] = read();
        p[i].id = i - 1;
    }
    root = build(1, n);
    q = read();
    for(int i = 1; i <= q; i++) {
        int x = read(), xx = read();
        int y = read(), yy = read();
        Point ld, ru;
        ld.v[0] = x; ld.v[1] = y;
        ru.v[0] = xx; ru.v[1] = yy;
        vector<int> v;
        v.clear();
        getNodeset(root, 1, n, v, ld, ru);
        sort(v.begin(), v.end());
        for(auto x : v)
            printf("%d\n", x);
        printf("\n");
    }
    return 0;
}
```

可持久化数据结构

可持久化 Trie

```
namespace Trie {
    struct node {
        int ch[2], ed, siz;
    } tree[N * 40];
    int tot = 0;
    int _new() {
        tot++;
        tree[tot].ch[0] = 0;
        tree[tot].ch[1] = 0;
        tree[tot].ed = tree[tot].siz = 0;
        return tot;
    }
    void init() {
        tot = 0;
        rt[0] = _new();
    }
    int Insert(int x, int t, int i = 15) {
        int u = _new(), f = (x >> i) & 1;
        tree[u] = tree[t];
        if(i == -1) {
            ed(u)++;
            siz(u)++;
            return u;
        }
        son(u, f) = Insert(x, son(t, f), i - 1);
        siz(u) = siz(son(u, 0)) + siz(son(u, 1));
        return u;
    }
    void print(int u, int now) {
        if(u == 0) return ;
        for(int i = 1; i <= ed(u); i++) printf("%d ", now);
        if(son(u, 0)) print(son(u, 0), now * 2);
        if(son(u, 1)) print(son(u, 1), now * 2 + 1);
```

```
    }
    int query(int u1, int u2, int x, int i = 15, int now = 0) {
        if(i == -1) return now;
        int f = (x >> i) & 1;
        if(siz(son(u1, f ^ 1)) - siz(son(u2, f ^ 1)) > 0)
            return query(son(u1, f ^ 1), son(u2, f ^ 1), x, i - 1, now * 2 + (f ^ 1));
        else return query(son(u1, f), son(u2, f), x, i - 1, now * 2 + (f));
    }
}
```

主席树（静态第 k 小）

建立权值树，那么 $[l, r]$ 的区间权值树就是第 r 个版本减去第 $l-1$ 个版本的树。

```
#include <iostream>
#include <cstdio>
#include <algorithm>
#include <cmath>
#include <assert.h>
#define Mid ((l + r) / 2)
#define lson (rt << 1)
#define rson (rt << 1 | 1)
using namespace std;
int read() {
    char c; int num, f = 1;
    while(c = getchar(),!isdigit(c)) if(c == '-') f = -1; num = c - '0';
    while(c = getchar(), isdigit(c)) num = num * 10 + c - '0';
    return f * num;
}
const int N = 1e7 + 1009;
const int M = 2e5 + 1009;
struct node {
    int ls, rs, v;
} tree[N];
int tb;
int n, m, tot, a[M], b[M], rt[M];
int _new(int ls, int rs, int v) {
    tree[++tot].ls = ls;
    tree[tot].rs = rs;
    tree[tot].v = v;
    return tot;
}
void update(int rt) {
    tree[rt].v = tree[tree[rt].ls].v + tree[tree[rt].rs].v;
}
int build(int l, int r) {
    if(l == r) return _new(0, 0, 0);
    int x = _new(build(l, Mid), build(Mid + 1, r), 0);
    update(x);
    return x;
}
int add(int l, int r, int p, int rt, int v) {
    int x = ++tot;
    tree[x] = tree[rt];
    if(l == r) {
        tree[x].v += v;
        return x;
    }
    if(p <= Mid) tree[x].ls = add(l, Mid, p, tree[x].ls, v);
    else tree[x].rs = add(Mid + 1, r, p, tree[x].rs, v);
    update(x);
    return x;
}
int query(int l, int r, int rt1, int rt2, int k) {
    if(l == r) return l;
    if(k <= tree[tree[rt1].ls].v - tree[tree[rt2].ls].v) return query(l, Mid, tree[rt1].ls, tree[rt2].ls, k);
    else return query(Mid + 1, r, tree[rt1].rs, tree[rt2].rs, k - (tree[tree[rt1].ls].v - tree[tree[rt2].ls].v));
}
void Debug(int l, int r, int rt) {
    printf("%d %d %d\n", l, r, tree[rt].v);
    if(l == r) return ;
```

```
    Debug(l, Mid, tree[rt].ls);
    Debug(Mid + 1, r, tree[rt].rs);
}
signed main()
{
    n = read(); m = read();
    for(int i = 1; i <= n; i++) a[i] = b[i] = read();
    sort(b + 1, b + 1 + n);
    tb = unique(b + 1, b + 1 + n) - b - 1;
    rt[0] = build(1, tb);
    for(int i = 1; i <= n; i++) {
        rt[i] = add(1, tb, lower_bound(b + 1, b + 1 + tb, a[i]) - b, rt[i - 1], 1);
    }
    for(int i = 1; i <= m; i++) {
        int l, r, k;
        l = read(); r = read(); k = read();
        assert(r - l + 1 >= k);
        printf("%d\n", b[query(1, tb, rt[r], rt[l - 1], k)]);
    }
    return 0;
}
```

cdq 分治三维偏序

先按照第一维排序, 然后对第二维归并, 归并时计算左对右的贡献, 先双指针, 满足当前统计出的第二维都有序

```
const int N = 1e6 + 1009;
struct node{
    int x, y, z, id, cnt;
}a[N], tmp[N];
bool operator ==(const node &a, const node &b) {
    return a.x == b.x && a.y == b.y && a.z == b.z;
}
int n, m, tot, ans[N], tt[N], tree[N];
int ttt[N];
bool cmp(node a, node b) {
    if(a.x == b.x && a.y == b.y) return a.z < b.z;
    if(a.x == b.x) return a.y < b.y;
    return a.x < b.x;
}
void add(int x, int y) {
    for( ; x <= m; x += x & -x)
        tree[x] += y;
}
int query(int x) {
    int ans = 0;
    for( ; x; x -= x & -x)
        ans += tree[x];
    return ans;
}
void cdq(int l, int r) {
    if(l == r) return ;
    cdq(l, Mid); cdq(Mid + 1, r);
    int i = l, j = Mid + 1, now = l - 1;
    while(i <= Mid && j <= r) {
        if(a[i].y <= a[j].y) {
            tmp[++now] = a[i];
            add(a[i].z, a[i].cnt);
            i++;
        } else {
            tmp[++now] = a[j];
            ans[a[j].id] += query(a[j].z);
            j++;
        }
    }
    while(i <= Mid) {
        tmp[++now] = a[i];
        add(a[i].z, a[i].cnt);
        i++;
    }
    while(j <= r) {
```

```
            tmp[++now] = a[j];
            ans[a[j].id] += query(a[j].z);
            j++;
        }
        for(int i = l; i <= Mid; i++) add(a[i].z, -a[i].cnt);
        for(int i = l; i <= r; i++) a[i] = tmp[i];
}
main()
{
    n = read(); m = read();
    for(int i = 1; i <= n; i++) {
        a[i].x = read();
        a[i].y = read();
        a[i].z = read();
        a[i].cnt = 1;
    }
    sort(a + 1, a + 1 + n, cmp);
    for(int i = 1; i <= n; i++) {
        if(i == 1 || !(a[i] == a[i - 1])){
            a[++tot] = a[i];
        }else a[tot].cnt += a[i].cnt;
    }
    for(int i = 1; i <= tot; i++) a[i].id = i, ttt[i] = a[i].cnt;
    cdq(1, tot);
    for(int i = 1; i <= tot; i++) tt[ans[i] + ttt[i] - 1] += ttt[i];
    for(int i = 0; i < n; i++) printf("%d\n", tt[i]);
    return 0;
}
```

数学

数论

欧拉函数

性质

1 和任何数互质。

$+\phi(1) = 1 + \phi(p) = p - 1$(p 为质数) $+ \phi(x \times p) = \phi(x) \times p (p \mid x), \phi(x \times p) = \phi(x) \times p (p \nmid x)$

线性欧拉函数筛

```
void getphi() {
    phi[1] = 1;
    for(int i = 2; i < N; i++) {
        if(!f[i]) phi[pri[++tot] = i] = i - 1;
        for(int j = 1; j <= tot && (k = i * pri[j]) < N; j++) {
            f[k] = 1;
            if(i % pri[j]) phi[k] = phi[i] * (pri[j] - 1);
            else {
                phi[k] = phi[i] * pri[j];
                break;
            }
        }
    }
}
```

$O(\sqrt{n})$ 求欧拉函数

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

排列组合

斯特林近似求组合 ($\geq$ 15 时收敛)

精度容易不够，推荐使用 python Demical 类

$$\ln n! \simeq n \ln n - n + \frac{1}{6} \ln (8n^3 + 4n^2 + n + \frac{1}{30}) + \frac{1}{2} \ln \pi$$

```
double lnfac(int n) {
    return n * log(n) - n + 1.0 / 6 * log(8 * n * n * n + 4 * n * n + n + 1.0 / 30) + 0.5 * log(acos(-1.0));
}
double C(int n, int m) {
    return exp(lnfac(n) - lnfac(n - m) - lnfac(m));
}
```

Lucas 定理

$$\binom{n}{m} = \binom{n \mod p}{m \mod p} \times \binom{n/p}{m/p}$$

```
int C(int n, int m) {
    if(m > n) return 0;
    if(n < mod) return 1ll * fac[n] * inv[n - m] % mod * inv[m] % mod;
    else return 1ll * C(n / mod, m / mod) * C(n % mod, m % mod) % mod;
}
```

Min-Max 容斥

$$max(S) = \sum_{T \subseteq S} (-1)^{|T|-1} min(T)$$

逆元

线性推

```
inv[1] = inv[0] = 1;
for(int i = 2; i < N; i++)  inv[i] = (1ll * mod - mod / i) * inv[mod % i] % mod;
```

费马小定理 (模数为质数)

```
int inv(int x) {
    return Pow(x % mod, mod - 2);
}
```

exgcd(ap 互质)

```
int inv(int x) {
    int x, y;
    exgcd(x, y, a, p);
    return (x % p + p) % p;
}
```

拓展欧几里得

求解的是类似 $ax + by = gcd(a, b)$ 的一组解。

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

拓展中国剩余定理

拓展中国剩余定理用于解决同余方程组。

$$x \equiv a_i (mod\ b_i)$$

构造 $M_k = lcm_{i=1}^{k-1} b_i$
假设前面的解为 p 显然新解 $p + M_k \times y$ 仍然是前面方程的解。
exgcd 求出 $M_k \times x + b_i \times y = gcd(M_k, b_i)$ 的解。
于是 $p' = p + x \times M_k \times (a_i - p)/gcd(M_k, b_i)$。
实际处理的时候可以直接让 $b_i = b_i/gcd(b_i, M_k)$ 防止溢出。

```
#define long long ll
ll gcd(ll a, ll b) {
    return b == 0 ? a : gcd(b, a % b);
}
ll lcm(ll a, ll b) {
    return a / gcd(a, b) * b;
}
ll exgcd(ll &x, ll &y, ll a, ll b) {
    if(b == 0) return x = 1, y = 0, a;
    ll t = exgcd(y, x, b, a % b);
    y -= a / b * x;
    return t;
}
inline ll mul(ll x, ll y, ll mod){
    return (x * y - (ll)((long double)x / mod * y) * mod + mod) % mod;
}

ll excrt(ll n, ll *a, ll *b) {
    ll ans = a[1], M = b[1];
    for(ll i = 2; i <= n; i++) {
        ll c = ((a[i] - ans) % b[i] + b[i]) % b[i], x, y;
        ll t = exgcd(x, y, M, b[i]), pb = b[i] / t;
        if(c % t != 0) return -1;
        x = mul(x, c / t, pb);
        ans = ans + x * M;
        M = M *pb;
        ans = (ans % M + M) % M;
    }
    return ans;
}
```

Miller_rabbin 素数测试

```
namespace Isprime{
    ll mul(ll x, ll y, ll mod){
        return (x * y - (ll)((long double)x / mod * y) * mod + mod) % mod;
    }
    ll Pow(ll a, ll p, ll mod) {
        ll ans = 1;
        for( ; p; p >>= 1, a = mul(a, a, mod))
            if(p & 1)
                ans = mul(ans, a, mod);
        return ans % mod;
    }
    int check(ll P){
        const ll test[11] = {0, 2, 3, 5, 7, 11, 13, 17, 19, 23, 29};
        if(P == 1) return false;
        if(P > 6 && P % 6 != 1 && P % 6 != 5) return false;
        ll k = 0, t = P - 1;
        while(!(t & 1)) k++, t >>= 1;
        for(int i = 1; i <= 10 && test[i] <= P; i++) {
            if(P == test[i]) return true;
            ll nxt, a = Pow(test[i], t, P);
            for(int j = 1; j <= k; j++) {
                nxt = mul(a, a, P);
                if(nxt == 1 && a != 1 && a != P - 1) return false;
                a = nxt;
```

```
            }
            if(a != 1)return false;
        }
        return true;
    }
}
```

多项式

结论

1. 自然数幂之和 $s(n)=\sum_{i=0}^{n} i^k$ 是关于 n 的 $k+1$ 次多项式

拉格朗日插值法

令拉格朗日函数

$$l_i(x)=\prod_{j\neq i}\frac{x-x_j}{x_i-x_j}$$

注意到这个函数有一些性质：
1. 次数为 n
2. 在 $x=x_i$ 位置值为 1,$x=x_j(j\neq i)$ 位置值为 0
于是可以凑出唯一的多项式表达式为：

$$f(x)=\sum_{i=0}^{n} y_i\prod_{j\neq i}\frac{x-x_j}{x_i-x_j}$$

如果要取模的话得求逆元，逆元先求好分母再一起求即可。

```
int interpolation(int *x, int *y, int n) {
    int f = 0;
    for(int i = 1; i <= n; i++) {
        int s1 = 1, s2 = 1;
        for(int j = 1; j <= n; j++) {
            if(i != j) {
                s1 = 1ll * s1 * (k - x[j] + mod) % mod;
                s2 = 1ll * s2 * (x[i] - x[j] + mod) % mod;
            }
        }
        f = (f + 1ll * y[i] * s1 % mod * inv(s2) % mod) % mod;
    }
    return f;
}
```

FFT 快速傅里叶变换

FFT 的想法是把第 k 号位置变成 $f(\omega_n^k)$，注意到 $\omega_n^k=-\omega_n^{k+n/2}$，于是可以进行变换。
几条公式：

$$\omega_n^n=1$$

$$\omega_n^k=\omega_{2n}^{2k}$$

$$\omega_{2n}^{k+n}=-\omega_{2n}^k$$

蝴蝶变换：相邻的位置为二进制的 reverse
DFT 变换公式 ($DFT(f)$ 为矩阵):
令

$$G(x)=a_0+a_2x+a_4x^2+...$$

$$H(x)=a_1+a_3x+a_5x^3+...$$

则有

$$f(x) = G(x^2) + x \times H(x^2)$$
$$DFT(f(\omega_n^k)) = DFT(G(\omega_{n/2}^k) + \omega_n^k \times DFT(H(\omega_{n/2}^k)))$$
$$DFT(f(\omega_n^{k+n/2})) = DFT(G(\omega_{n/2}^k) - \omega_n^k \times DFT(H(\omega_{n/2}^k)))$$

$DFT(G(\omega_{n/2}^k), DFT(H(\omega_{n/2}^k)))$ 可递归计算

NTT 快速数论变换

NTT 使用原根代替复数进行运算。
原根 g 的重要性质：$g^t \equiv k \mod n, t \in [0, n-2]$，$k$ 遍取 $1 \sim n-1$
原根存在的充要条件是：模数 $n = 2, 4, p^\alpha, 2p^\alpha (p$ 为奇质数)。
对于一个质数 $p = qn + 1 (n = 2^m)$, 原根满足性质 $g^{qn} \equiv 1 \mod p$。
它满足和复数近似的性质，我们把 q 看成复数中的 2π，就可以套用 FFT 实现 NTT 了。
$g_n^n \equiv 1, g_n^n \equiv -1$

通常取

$$p = 1004535809 = 7 \times 479 \times 2^{21} + 1, g = 3$$

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

```cpp
const int P = 998244353, G = 3, Gi = 332748118;
struct Complex {double x, y;};
Complex operator+(const Complex &a, const Complex &b) {return (Complex) {a.x + b.x, a.y + b.y};}
Complex operator-(const Complex &a, const Complex &b) {return (Complex) {a.x - b.x, a.y - b.y};}
Complex operator*(const Complex &a, const Complex &b) {return (Complex) {a.x * b.x - a.y * b.y, a.x * b.y + a.y * b.x};}
namespace Polynomial {
    const double Pi = acos(-1.0);
    int rev[N];
    template <typename T>
    void change(T *y, int n) {
        for(int i = 0; i < n; i++)
            rev[i] = (rev[i >> 1] >> 1) | ((i & 1) ? (n >> 1) : 0);
        for(int i = 0; i < n; i++)
            if(i < rev[i])
                swap(y[i], y[rev[i]]);
    }
    void FFT(Complex *A, int n, int type) {
        //type = 1 DFT
        //type = -1 IDFT
        //确保 n 是 2 的幂次
        change(A, n);
        for(int m = 1; m < n; m <<= 1) {
            Complex Wn = (Complex) {cos(Pi / m), type * sin(Pi / m)};
            for(int i = 0; i < n; i += 2 * m) {
                Complex w = (Complex) {1.0, 0};
                for(int j = 0; j < m; j++, w = w * Wn) {
                    Complex x = A[i + j], y = w * A[i + j + m];
                    A[i + j] = x + y;
                    A[i + j + m] = x - y;
                }
            }
        }
        if(type == -1) {
            for(int i = 0; i < n; i++)
                A[i].x = A[i].x / n;
        }
    }
    void NTT(int *A, int n, int type) {
        //type = 1 DFT
        //type = -1 IDFT
        change(A, n);
```

```
        for(int m = 1; m < n; m <<= 1) {
            int Wn = Pow(type == 1 ? G : Gi, (P - 1) / (m << 1));
            for(int i = 0; i < n; i += 2 * m) {
                int w = 1;
                for(int j = 0; j < m; j++, w = 1ll * w * Wn % P) {
                    int x = A[i + j], y = 1ll * w * A[i + j + m] % P;
                    A[i + j] = (x + y) % P;
                    A[i + j + m] = (x - y + P) % P;
                }
            }
        }
        if(type == -1) {
            int inv = Pow(n, P - 2);
            for(int i = 0; i < n; i++)
                A[i] = 1ll * A[i] * inv % P;
        }
    }

}
//以下代码加在主函数内
limit = 1;
while(limit <= n + m) limit <<= 1;
Polynomial :: FFT(A, limit, 1);
Polynomial :: FFT(B, limit, 1);
for(int i = 0; i < limit; i++) A[i] = A[i] * B[i];
Polynomial :: FFT(A, limit, -1);
```

FWT 快速沃尔什变换

FWT 用于计算下列多项式

$$C[k] = \sum_{i \oplus j = k} A[i] \times B[j]$$

先通过 FWT 将 A, B 变为 $FWT(A), FWT(B)$，这样有 $FWT(C) = FWT(A) \times FWT(B)$。
当然位运算符不同的时候对应的变换形式也需要改变。

$a \in S, b \in S$ 可以表示为 $a|b \in S$
FWT 为线性变换 $\sum FWT(F) = FWT(\sum F)$

与卷积
当 $\oplus = and$ 的时候
$FWT(A) = (FWT(A_0) + FWT(A_1), FWT(A_1))$
$FWT(A) = A$(长度为 1)
$IFWT(A) = (IFWT(A_0) - IFWT(A_1), IFWT(A_1))$

或卷积
当 $\oplus = or$ 的时候
$FWT(A) = (FWT(A_0), FWT(A_0) + FWT(A_1))$
$FWT(A) = A$(长度为 1)
$IFWT(A) = (IFWT(A_0), IFWT(A_1) - IFWT(A_0))$

异或卷积
当 $\oplus = xor$ 的时候
$FWT(A) = (FWT(A_0) + FWT(A_1), FWT(A_0) - FWT(A_1))$
$FWT(A) = A$(长度为 1)
$IFWT(A) = (\frac{IFWT(A_0)+IFWT(A_1)}{2}, \frac{IFWT(A_0)-IFWT(A_1)}{2})$

```
namespace Polynomial {
    void FWT_or(int *A, int n, int type) {
        for(int m = 1; m < n; m <<= 1) {
            for(int i = 0; i < n; i += 2 * m) {
                for(int j = 0; j < m; j++) {
                    A[i + j + m] = (1ll * A[i + j + m] + A[i + j] * type + mod) % mod;
                }
            }
```

```
        }
    }
    void FWT_and(int *A, int n, int type) {
        for(int m = 1; m < n; m <<= 1) {
            for(int i = 0; i < n; i += 2 * m) {
                for(int j = 0; j < m; j++) {
                    A[i + j] = (1ll * A[i + j + m] * type + A[i + j] + mod) % mod;
                }
            }
        }
    }
    void FWT_xor(int *A, int n, int type) {
        int inv_2 = Pow(2, mod - 2);
        for(int m = 1; m < n; m <<= 1) {
            for(int i = 0; i < n; i += 2 * m) {
                for(int j = 0; j < m; j++) {
                    int x = A[i + j], y = A[i + j + m];
                    A[i + j] = (1ll * x + y) * (type == 1 ? 1 : inv_2) % mod;
                    A[i + j + m] = (1ll * x - y + mod) * (type == 1 ? 1 : inv_2) % mod;
                }
            }
        }
    }
}
```

子集卷积

子集卷积求的是下面一个式子:

$$c_k = \sum_{i|j=k, i\&j=0} a_i \times b_j$$

就是把集合 k 划分成两个集合。
后面那个与的条件通过 $|k| = |i| + |j|$ 干掉，加一维集合元素个数，就变成了

$$c[i+j][mask_k] = \sum_{i|j=k} a[i][mask_i] \times b[j][mask_j]$$

这个可以用 FWT 算。

```
namespace ssc{
    int f[21][1 << 21], g[21][1 << 21], ans[21][1 << 21];
    void subset_convolution(int *A, int *B, int *C, int n, int lim) {
    //  memset(f, 0, sizeof(f));
    //  memset(g, 0, sizeof(g));
        for(int i = 0; i < lim; i++) f[__builtin_popcount(i)][i] = A[i];
        for(int i = 0; i < lim; i++) g[__builtin_popcount(i)][i] = B[i];
        for(int i = 0; i <= n; i++) FWT_or(f[i], lim, 1), FWT_or(g[i], lim, 1);
        for(int i = 0; i <= n; i++)
            for(int j = 0; j <= i; j++)
                for(int k = 0; k < lim; k++)
                    ans[i][k] = (ans[i][k] + 1ll * f[j][k] * g[i - j][k] % mod) % mod;
        for(int i = 0; i <= n; i++) FWT_or(ans[i], lim, -1);
        for(int i = 0; i < lim; i++) C[i] = ans[__builtin_popcount(i)][i];
    }
}
```

群论

结论

1. **子群检验法**: 群 G 是群 H 的子群的充分必要条件：对于所有元素 h,g，只需检查 $g^{-1} \cdot h \in H$。

线性代数

矩阵运算全家桶

```
struct mat {
    int g[5][5], n, m;
};
void operator+=(mat &a, const mat &b) {
    if(a.n != b.n || a.m != b.m) cerr << "+= size error" << endl, exit(0);
    for(int i = 1; i <= a.n; i++)
        for(int j = 1; j <= a.m; j++) {
            a.g[i][j] = (a.g[i][j] + b.g[i][j]);
            if(a.g[i][j] >= mod) a.g[i][j] -= mod;
        }
}
void operator-=(mat &a, const mat &b) {
    if(a.n != b.n || a.m != b.m) cerr << "-= size error" << endl, exit(0);
    for(int i = 1; i <= a.n; i++)
        for(int j = 1; j <= a.m; j++) {
            a.g[i][j] -= b.g[i][j];
            if(a.g[i][j] < 0) a.g[i][j] += mod;
        }
}
mat operator+(const mat &a, const mat &b) {
    if(a.n != b.n || a.m != b.m) cerr << "+ size error" << endl, exit(0);
    mat c;
    c.n = a.n; c.m = a.m;
    for(int i = 1; i <= a.n; i++)
        for(int j = 1; j <= a.m; j++) {
            c.g[i][j] = (a.g[i][j] + b.g[i][j]);
            if(c.g[i][j] >= mod) c.g[i][j] -= mod;
        }
    return c;
}
mat operator-(const mat &a, const mat &b) {
    if(a.n != b.n || a.m != b.m) cerr << "- size error" << endl, exit(0);
    mat c;
    c.n = a.n; c.m = a.m;
    for(int i = 1; i <= a.n; i++)
        for(int j = 1; j <= a.m; j++) {
            c.g[i][j] = (a.g[i][j] - b.g[i][j]);
            if(c.g[i][j] < 0) c.g[i][j] += mod;
        }
    return c;
}
mat operator*(const mat &a, const mat &b) {
    if(a.m != b.n) cerr << "* size error" << endl, exit(0);
    mat c;
    c.n = a.n; c.m = b.m;
    for(int i = 1; i <= a.n; i++) {
        for(int j = 1; j <= b.m; j++) {
            c.g[i][j] = 0;
            for(int k = 1; k <= a.m; k++) {
                c.g[i][j] = c.g[i][j] + 1ll * a.g[i][k] * b.g[k][j] % mod;
                if(c.g[i][j] >= mod) c.g[i][j] -= mod;
            }
        }
    }
    return c;
}
mat Pow(mat a, int p) {
    if(a.n != a.m) cerr << "* size error" << endl, exit(0);
    mat ans;
    ans.n = ans.m = a.n;
    memset(ans.g, 0, sizeof(ans.g));
    for(int i = 1; i <= ans.n; i++) ans.g[i][i] = 1;
    for(; p; p >>= 1, a = a * a)
        if(p & 1)
            ans = ans * a;
    return ans;
}
```

图论

树论

树的直径

模板：POJ - 1985

- 两遍 DFS

```
void dfs(int x, int fa) {
    for(int i = 0; i < E[x].size(); i++) {
        int y = E[x][i].ver;
        int w = E[x][i].val;
        if(y == fa) continue;
        d[y] = d[x] + w;
        if(d[y] > d[c]) c = y;
        dfs(y, x);
    }
}
signed main()
{
    n = read();
    for(int i = 1; i < n; i++) {
        int x = read(), y = read(), w = read();
        E[x].push_back((Edge) {y, w});
        E[y].push_back((Edge) {x, w});
    }
    dfs(1, 0);
    d[c] = 0;
    dfs(c, 0);
    printf("%d\n", d[c]);
    return 0;
}
```

- 树形 DP

```
void dfs(int x, int fa) {
    d1[x] = d2[x] = 0;
    for(int i = 0; i < E[x].size(); i++) {
        int y = E[x][i].ver;
        int w = E[x][i].val;
        if(y == fa) continue;
        dfs(y, x);
        int t = d1[y] + w;
        if(t > d1[x]) {
            d2[x] = d1[x];
            d1[x] = t;
        } else if(t > d2[x]) {
            d2[x] = t;
        }
    }
    d = max(d, d1[x] + d2[x]);
}
signed main()
{
    n = read();
    for(int i = 1; i < n; i++) {
        int x = read(), y = read(), w = read();
        E[x].push_back((Edge) {y, w});
        E[y].push_back((Edge) {x, w});
    }
    dfs(1, 0);
    printf("%d\n", d);
    return 0;
}
```

求 LCA

- 树链剖分

```cpp
namespace Tree {
    int siz[N], mson[N], ltp[N], fa[N], dth[N];
    vector<int> son[N];
    void dfs1(int x, int pre) {
        siz[x] = 1;
        mson[x] = 0;
        fa[x] = pre;
        dth[x] = dth[pre] + 1;
        for(auto y : son[x]) if(y != pre) {
            dfs1(y, x);
            if(mson[x] == 0 || siz[y] > siz[mson[x]]) mson[x] = y;
        }
    }
    void dfs2(int x, int pre, int tp) {
        ltp[x] = tp;
        if(mson[x]) dfs2(mson[x], x, tp);
        for(auto y : son[x]) if(y != pre && y != mson[x]) {
            dfs2(y, x, y);
        }
    }
    void init() {
        dfs1(1, 0);
        dfs2(1, 0, 1);
    }
    int LCA(int x, int y) {
        while(ltp[x] != ltp[y]) {
            if(dth[ltp[x]] > dth[ltp[y]]) x = fa[ltp[x]];
            else y = fa[ltp[y]];
        }
        return dth[y] > dth[x] ? x : y;
    }
}
```

- 倍增

```cpp
namespace Tree {
    vector<int> son[N];
    int root, fa[N][31], dth[N];
    void dfs(int x, int pre) {
        fa[x][0] = pre;
        dth[x] = dth[pre] + 1;
        for(int i = 1; i <= 30; i++)
            fa[x][i] = fa[fa[x][i - 1]][i - 1];
        for(auto y : son[x]) if(y != pre)
            dfs(y, x);
    }
    void init() {
        dfs(root, 0);
    }
    int LCA(int x, int y) {
        if(dth[x] > dth[y]) swap(x, y);
        for(int i = 30; ~i; i--)
            if(dth[fa[y][i]] >= dth[x])
                y = fa[y][i];
        if(x == y) return x;
        for(int i = 30; ~i; i--)
            if(fa[y][i] != fa[x][i]) {
                x = fa[x][i];
                y = fa[y][i];
            }
        return fa[x][0];
    }
}
```

树上启发式合并

长春站的痛.jpg

- 先递归计算轻儿子的答案
- 计算重儿子的答案，并且保留重儿子的状态数组
- 把其他所有轻儿子的答案加到状态数组中，更新当前点的答案

```
void dfs1(int x, int pre) {
    siz[x] = 1;
    mson[x] = 0;
    for(auto y : son[x]) if(y != pre) {
        dfs1(y, x);
        siz[x] += siz[y];
        if(!mson[x] || siz[y] > siz[mson[x]]) mson[x] = y;
    }
}
void add(int x, int pre, int v) {
    cnt[col[x]] += v;
    if(cnt[col[x]] > Mx) Mx = cnt[col[x]], sum = col[x];
    else if(cnt[col[x]] == Mx) sum += col[x];
    for(auto y : son[x]) {
        if(y == pre || y == Son) continue;
        add(y, x, v);
    }
}
void dfs2(int x, int pre, int keep) {
    for(auto y : son[x]) {
        if(y == pre || y == mson[x]) continue;
        dfs2(y, x, 0);
    }
    if(mson[x]) dfs2(mson[x], x, 1), Son = mson[x];
    add(x, pre, 1); Son = 0;
    ans[x] = sum;
    if(!keep) add(x, pre, -1), sum = 0, Mx = 0;

}
```

图论

第 k 短路

模板：HDU-6351

估值函数：$h(x) = f(x) + g(x)$，其中 $f(x)$ 为从起点到现在的距离，$g(x)$ 为起点到当前点的最短路。

```
bool operator<(const node &a, const node &b) {
    return a.f + a.g > b.f + b.g;
}
priority_queue<node> q;
signed main()
{
    n = read(); m = read();
    for(int i = 1; i <= m; i++) {
        int x, y, w;
        x = read(); y = read(); w = read();
        E[x].push_back((Edge) {y, w});
        re[y].push_back((Edge) {x, w});
    }
    s = read(); t = read(); k = read();
    memset(dis, 0x3f, sizeof(dis)); dis[t] = 0;
    q.push((node) {t, 0, 0});
    while(q.size()) {
        int x = q.top().x, d = q.top().f;
        q.pop();
        if(dis[x] < d) continue;
        for(int i = 0; i < re[x].size(); i++) {
            int y = re[x][i].y, w = re[x][i].w;
            if(dis[y] > dis[x] + w) {
                dis[y] = dis[x] + w;
                q.push((node) {y, dis[y], 0});
```

```
            }
        }
    }
    for(int i = 1; i <= n; i++) cnt[i] = k;
    cnt[s]++;
    q.push((node) {s, 0, dis[s]});
    while(q.size()) {
        int x = q.top().x, f = q.top().f, g = q.top().g;
        q.pop();
        if(cnt[x] == 0) continue;
        cnt[x]--;
        if(x == t && cnt[x] == 0) {
            printf("%lld\n", f);
            return 0;
        }
        for(int i = 0; i < E[x].size(); i++) {
            int y = E[x][i].y, w = E[x][i].w;
            q.push((node) {y, f + w, dis[y]});
        }
    }
    printf("-1\n");
    return 0;
}
```

二分图匹配

结论

最大匹配数：最大匹配的匹配边的数目

最小点/边覆盖数：选取最少的点/边，使任意一条边至少有一个点被选择 / 点至少连有一条边。

最大独立数：选取最多的点，使任意所选两点均不相连

最小路径覆盖数：对于一个 DAG（有向无环图），选取最少条路径，使得每个顶点属于且仅属于一条路径。路径长可以为 0（即单个点）。

1. 最大匹配数 = 最小点覆盖数（这是 Konig 定理）
2. 最大匹配数 = 最大独立数
3. 最小路径覆盖数 = 顶点数 - 最大匹配数
4. 原图的最大团 = 补图的最大独立集原图的最大独立集 = 补图的最大团
5. 最小边覆盖 = 顶点数 - 最大匹配数

在一般图中：

最小不相交路径覆盖：每个点拆点为 $2x-1, 2x$，那么一条边 (x, y)，则连边 $(2x-1, 2y)$，答案是 $n - maxmatch$

最小可相交路径覆盖：跑一遍传递闭包，按传递闭包上的边建边之后转化为最小不相交路径覆盖。

二分图最大匹配的必须边：

在完备匹配中：

匹配边从左到右方向，非匹配边从右到左方向，则一条边为必须边当且仅当边在最大匹配中，并且边所连的两个点**不在**同一个强连通分量中。

在非完备匹配中：

匈牙利算法

```
int dfs(int x) {
    for(int i = head[x]; i; i = nxt[i]) {
        int y = ver[i];
        if(vis[y]) continue;
        vis[y] = 1;
        if(!match[y] || dfs(match[y])) {
            match[y] = x;
            return true;
        }
    }
    return false;
```

```
}
for(int i = 1; i <= n; i++) {
    memset(vis, 0, sizeof(vis));
    if(dfs(i)) ans++;
}
```

KM 算法二分图最大权匹配

KM 算法只支持二分图最大权完美匹配，若图不一定存在完美匹配，注意补 0 边和补点。

KM 算法引入了顶标的概念，用 $la[x]$ 和 $lb[x]$ 分别保存两侧点的顶标，顶标必须满足大于所有边。
每次对每个点进行循环匹配，匹配中统计一个 $delta$ 表示最小的权值使得一条边可以加入。
然后修改顶标再继续匹配。

```
int la[N], lb[N], va[N], vb[N], delta, match[N], g[N][N], n;
int dfs(int x) {
    va[x] = 1;
    for(int y = 1; y <= n; y++) {
        if(!vb[y]) {
            if(la[x] + lb[y] - g[x][y] == 0) {
                vb[y] = 1;
                if(!match[y] || dfs(match[y])) {
                    match[y] = x;
                    return true;
                }
            } else delta = min(delta, la[x] + lb[y] - g[x][y]);
        }
    }
    return false;
}
void work() {
    for(int i = 1; i <= n; i++)
        for(int j = 1; j <= n; j++)
            g[i][j] = read();
    memset(match, 0, sizeof(match));
    for(int i = 1; i <= n; i++) {
        la[i] = g[i][1];
        lb[i] = 0;
        for(int j = 2; j <= n; j++)
            la[i] = max(la[i], g[i][j]);
    }
    for(int i = 1; i <= n; i++) {
        while(true) {
            memset(va, 0, sizeof(va));
            memset(vb, 0, sizeof(vb));
            delta = 0x3f3f3f3f;
            if(dfs(i)) break;
            for(int j = 1; j <= n; j++) {
                if(va[j]) la[j] -= delta;
                if(vb[j]) lb[j] += delta;
            }
        }
    }
    long long ans = 0;
    for(int i = 1; i <= n; i++)
        ans += g[match[i]][i];
    printf("%lld\n", ans);
}
```

网络流

Dinic 算法

```
const int inf = 0x3f3f3f3f;
queue<int> q;
int d[N];
int bfs() {
    memset(d, 0, sizeof(int) * (t + 10)); d[s] = 1;
    while(q.size()) q.pop(); q.push(s);
    while(q.size()) {
```

```cpp
        int x = q.front(); q.pop();
        for(int i = head[x]; i; i = nxt[i]) {
            if(d[ver[i]]) continue;
            if(edge[i] <= 0) continue;
            d[ver[i]] = d[x] + 1;
            q.push(ver[i]);
        }
    }
    return d[t];
}
int dinic(int x, int flow) {
    if(x == t) return flow;
    int k, res = flow;
    for(int i = head[x]; i && res; i = nxt[i]) {
        if(d[ver[i]] != d[x] + 1 || edge[i] <= 0) continue;
        k = dinic(ver[i], min(res, edge[i]));
        if(k == 0) d[ver[i]] = 0;
        edge[i] -= k;
        edge[i ^ 1] += k;
        res -= k;
    }
    return flow - res;
}
```

EK 算法费用流

```cpp
//反向边 cost 为负数，容量为 0
int SPFA() {
    queue<int> q; q.push(s);
    memset(dis, 0x3f, sizeof(dis)); dis[s] = 0;
    memset(vis, 0, sizeof(vis)); vis[s] = 1;
    q.push(s); flow[s] = 0x3f3f3f3f;
    while(q.size()) {
        int x = q.front();
        vis[x] = 0; q.pop();
        for(int i = head[x]; i; i = nxt[i]) {
            if(edge[i] <= 0) continue;
            if(dis[ver[i]] > dis[x] + cost[i]) {
                dis[ver[i]] = dis[x] + cost[i];
                pre[ver[i]] = i;
                flow[ver[i]] = min(flow[x], edge[i]);
                if(!vis[ver[i]]) {
                    q.push(ver[i]);
                    vis[ver[i]] = 1;
                }
            }
        }
    }
    return dis[t] != 0x3f3f3f3f;
}
void update() {
    int x = t;
    while(x != s) {
        int i = pre[x];
        edge[i] -= flow[t];
        edge[i ^ 1] += flow[t];
        x = ver[i ^ 1];
    }
    maxflow += flow[t];
    minncost += dis[t] * flow[t];
}
```

Dinic 算法费用流

```cpp
int SPFA() {
    while(q.size()) q.pop(); q.push(s);
    memset(d, 0x3f, sizeof(int) * (n + 10)); d[s] = 0;
    memset(vis, 0, sizeof(int) * (n + 10)); vis[s] = 1;
    while(q.size()) {
        int x = q.front(); q.pop();
```

```
        vis[x] = 0;
        for(int i = head[x]; i; i = nxt[i]) {
            if(edge[i] <= 0) continue;
            if(d[ver[i]] > d[x] + cost[i]) {
                d[ver[i]] = d[x] + cost[i];
                if(!vis[ver[i]]) {
                    vis[ver[i]] = 1;
                    q.push(ver[i]);
                }
            }
        }
    }
    return d[t] != 0x3f3f3f3f;
}
int dinic(int x, int flow) {
    if(x == t) return flow;
    vis[x] = 1;
    int k, res = flow;
    for(int i = head[x]; i && res; i = nxt[i]) {
        if(vis[ver[i]]) continue;
        if(d[ver[i]] != d[x] + cost[i] || edge[i] <= 0) continue;
        k = dinic(ver[i], min(edge[i], res));
        if(!k) d[ver[i]] = -1;
        edge[i] -= k;
        edge[i ^ 1] += k;
        res -= k;
        mincost += cost[i] * k;
    }
    vis[x] = 0;
    return flow - res;
}
```

无源汇上下界可行流

$x->y$，则 s 向 y，s 向 x 连 l,x 向 y 连 $r-l$，有可行流的条件是 s 出边全满流，解通过残量网络构造出。

```
for(int i = 1; i <= m; i++) {
    int x = read(), y = read();
    int l = read(), r = read();
    low[i] = l;
    add(x, y, r - l); add(y, x, 0);
    id[i] = tot;
    add(s, y, l); add(y, s, 0);
    add(x, t, l); add(t, x, 0);
}
while(bfs())
    dinic(s, inf);
int f = 1;
for(int i = head[s]; i; i = nxt[i]) {
    f &= (edge[i] == 0);
}
printf("%s\n", f ? "YES" : "NO");
if(!f) return 0;
for(int i = 1; i <= m; i++) {
    printf("%d\n", edge[id[i]] + low[i]);
}
```

连通性算法

Tarjan 强连通分量

$dfn[x]$：dfs 序。

$low[x]$：追溯值，指 x 的子树内部，通过一条非树边能到达的最小的 dfn 值。

如果 $dfn[x] == low[x]$，当前栈中，x 以后的元素为一个强连通。

```
void tarjan(int x) {
    low[x] = dfn[x] = ++dfncnt;
    s[++t] = x; vis[x] = 1;
    for(int i = head[x]; i; i = nxt[i]) {
```

```cpp
        if(!dfn[ver[i]]) {
            tarjan(ver[i]);
            low[x] = min(low[x], low[ver[i]]);
        } else if(vis[ver[i]]) {
            low[x] = min(low[x], dfn[ver[i]]);
        }
    }
    if(dfn[x] == low[x]) {
        int z = -1;
        ++sc;
        while(z != x) {
            scc[s[t]] = sc;
            siz[sc]++;
            vis[s[t]] = 0;
            z = s[t];
            t--;
        }
    }
}
//从任意点开始跑，但是注意如果图不连通，需要每个点跑一次
for(int i = 1; i <= n; i++)
    if(!dfn[i])
        tarjan(i);
```

点双连通

Tarjan 割点判定

```cpp
int cut[N];
namespace v_dcc {
    int root, low[N], dfn[N], dfntot;
    void tarjan(int x) {
        low[x] = dfn[x] = ++dfntot;
        int flag = 0;
        for(int i = head[x]; i; i = nxt[i]) {
            int y = ver[i];
            if(!dfn[y]) {
                tarjan(y);
                low[x] = min(low[x], low[y]);
                if(low[y] >= dfn[x]) {
                    flag++;
                    if(x != root || flag > 1) cut[x] = 1;

                }
            } else low[x] = min(low[x], dfn[y]);
        }
    }
    void getcut() {
        for(int i = 1; i <= n; i++)
            if(!dfn[i])
                tarjan(root = i);
    }
}
```

求点双连通分量

点双连通分量比较复杂，一个点可能存在于多个点双连通分量当中，一个点删除与搜索树中的儿子节点断开时，不能在栈中弹掉父亲点，但是父亲点属于儿子的 v-dcc。

```cpp
int cut[N];
vector<int> dcc[N];
namespace v_dcc {
    int s[N], t, root;
    int es[N], et;
    void tarjan(int x) {
        dfn[x] = low[x] = ++dfntot;
        s[++t] = x;
        if(x == root && head[x] == 0) {
            dcc[++dc].clear();
            dcc[dc].push_back(x);
```

```
            return ;
        }
        int flag = 0;
        for(int i = head[x]; i; i = nxt[i]) {
            int y = ver[i];
            if(!dfn[y]) {
                tarjan(y);
                low[x] = min(low[x], low[y]);
                if(low[y] >= dfn[x]) {
                    flag++;
                    if(x != root || flag > 1) cut[x] = true;
                    dcc[++dc].clear();
                    int z = -1;
                    while(z != y) {
                        z = s[t--];
                        dcc[dc].push_back(z);
                    }
                    dcc[dc].push_back(x);
                }
            } else low[x] = min(low[x], dfn[y]);
        }
    }
    void get_cut() {
        for(int i = 1; i <= n; i++)
            if(!dfn[i])
                tarjan(root = i);
    }
}
```

边双连通

搜索树上的点 x，若它的一个儿子 y，满足严格大于号 $low[y] > dfn[x]$，那么这条边就是桥。

注意由于会有重边，不能仅仅考虑他的父亲编号，而应该记录入边编号。

```
namespace e_dcc {
    int low[N], dfn[N], dfntot;
    vector<int> E[N];
    void tarjan(int x, int in_edge) {
        low[x] = dfn[x] = ++dfntot;
        for(int i = head[x]; i; i = nxt[i]) {
            int y = ver[i];
            if(!dfn[y]) {
                tarjan(y, i);
                low[x] = min(low[x], low[y]);
                if(low[y] > dfn[x])
                    bridge[i] = bridge[i ^ 1] = true;
            } else if(i != (in_edge ^ 1))
            //注意运算优先级
                low[x] = min(low[x], dfn[y]);
        }
    }
    void getbridge() {
        for(int i = 1; i <= n; i++)
            if(!dfn[i])
                tarjan(i, 0);
    }
    void dfs(int x) {
        dcc[x] = dc;
        for(int i = head[x]; i; i = nxt[i]) {
            if(!dcc[ver[i]] && !bridge[i]) {
                dfs(ver[i]);
            }
        }
    }
    void getdcc() {
        for(int i = 1; i <= n; i++) {
            if(!dcc[i]) {
                ++dc;
                dfs(i);
            }
```

```
        }
    }
    void getgraphic() {
        for(int x = 1; x <= n; x++) {
            for(int i = head[x]; i; i = nxt[i]) {
                if(dcc[ver[i]] != dcc[x]) {
                    E[dcc[x]].push_back(dcc[ver[i]]);
                    E[dcc[ver[i]]].push_back(dcc[x]);
                }
            }
        }
    }
}
```

2-SAT

2-SAT 用于解决每个变量的 01 取值问题，用于判断是否存在一种不冲突取值方法。

建边方法：假如选了 A 之后，B 的取值**确定**，那么就 A 的这个取值向 B 的这个取值建边，否则不要建边。

判定方法：如果，$\exists A$, 使得 A 和 $\neg A$ 在同一个强连通分量里面，说明不存在一种合法取值，否则存在。

输出方案：自底向上确定每个变量的取值, 由于 tarjan 求解强连通分量是自底向上，所以编号比较小的强连通是位于 DAG 底部的。

基于 tarjan 的方案输出就变得十分简单了，只要判断一个点和对立节点哪个 scc 的编号小就行了。

例如：$A->B->C$，那么 C 的编号最小。

```
for(int i = 1; i <= m; i++) {
    int x = read() + 1, y = read() + 1;
    int w = read();
    char c[10];
    scanf("%s", c + 1);
    if(c[1] == 'A') {
        if(w) {
            add(2 * x - 0, 2 * x - 1);
            add(2 * y - 0, 2 * y - 1);
        } else {
            add(2 * x - 1, 2 * y - 0);
            add(2 * y - 1, 2 * x - 0);
        }
    }
    if(c[1] == 'O') {
        if(w) {
            add(2 * x - 0, 2 * y - 1);
            add(2 * y - 0, 2 * x - 1);
        } else {
            add(2 * x - 1, 2 * x - 0);
            add(2 * y - 1, 2 * y - 0);
        }
    }
    if(c[1] == 'X') {
        if(w) {
            add(2 * x - 0, 2 * y - 1);
            add(2 * x - 1, 2 * y - 0);
            add(2 * y - 0, 2 * x - 1);
            add(2 * y - 1, 2 * x - 0);
        } else {
            add(2 * x - 0, 2 * y - 0);
            add(2 * x - 1, 2 * y - 1);
            add(2 * y - 0, 2 * x - 0);
            add(2 * y - 1, 2 * x - 1);
        }
    }
}
for(int i = 1; i <= 2 * n; i++)
    if(!dfn[i])
        tarjan(i);
for(int i = 1; i <= n; i++) {
    if(scc[2 * i - 0] == scc[2 * i - 1]) {
        printf("NO\n");
```

```
            return 0;
        }
    }
    printf("YES\n");
    //2 * x - a -> 2 * y - b 的边表示，假如 x 取值为 a，那么 y 的取值必须为 b

    //输出方案
    for(int i = 2; i <= 2 * n; i += 2) {
        if(scc[i - 0] == scc[i - 1]) {
            printf("NO\n");
            return 0;
        } else ans[(i + 1) / 2] = scc[i - 1] < scc[i - 0];
    }
```

计算几何

字符串

字串哈希

```
namespace String {
    const int x = 135;
    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 HashString {
        char s[N];
        int length, subsize;
        bool sorted;
        ull h[N], hl[N];
        ull init(const char *t) {
            if(xp[0] != 1) init_xp();
            length = strlen(t);
            strcpy(s, t);
            ull res1 = 0, res2 = 0;
            h[length] = 0;
            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
            }
            return h[0];
        }
        //获取子串哈希，左闭右开
        ull get_substring_hash(int left, int right)  {
            int len = right - left;
        #ifdef ENABLE_DOUBLE_HASH
            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);
        }
    };
}
```

Trie

```
namespace trie {
    int t[N][26], sz, ed[N];
    int _new() {
        sz++;
        memset(t[sz], 0, sizeof(t[sz]));
        return sz;
    }
    void init() {
        sz = 0;
        _new();
        memset(ed, 0, sizeof(ed));
    }
    void Insert(char *s, int n) {
        int u = 1;
        for(int i = 0; i < n; i++) {
            int c = s[i] - 'a';
            if(!t[u][c]) t[u][c] = _new();
            u = t[u][c];
        }
        ed[u]++;
    }
    int find(char *s, int n) {
        int u = 1;
        for(int i = 0; i < n; i++) {
            int c = s[i] - 'a';
            if(!t[u][c]) return -1;
            u = t[u][c];
        }
        return u;
    }
}
```

KMP 算法

```
namespace KMP {
    void get_next(char *t, int m, int *nxt) {
        int j = nxt[0] = 0;
        for(int i = 1; i < m; i++) {
            while(j && t[i] != t[j]) j = nxt[j - 1];
            nxt[i] = j += (t[i] == t[j]);
        }
    }
    vector<int> find(char *t, int m, int *nxt, char *s, int n) {
        vector<int> ans;
        int j = 0;
        for(int i = 0; i < n; i++) {
            while(j && s[i] != t[j]) j = nxt[j - 1];
            j += s[i] == t[j];
            if(j == m) {
```

```cpp
                    ans.push_back(i - m + 1);
                    j = nxt[j - 1];
                }
            }
            return ans;
        }
}
```

manacher 算法

```cpp
namespace manacher {
    char s[N];
    int p[N], len;
    void getp(string tmp) {
        len = 0;
        for(auto x : tmp) {
            s[len++] = '#';
            s[len++] = x;
        }
        s[len++] = '#';
        memset(p, 0, sizeof(int) * (len + 10));
        int c = 0, r = 0;
        for(int i = 0; i < len; i++) {
            if(i <= r) p[i] = min(p[2 * c - i], r - i);
            else p[i] = 1;
            while(i - p[i] >= 0 && i + p[i] < len && s[i - p[i]] == s[i + p[i]])
                p[i]++;
            if(i + p[i] - 1 > r) {
                r = i + p[i] - 1;
                c = i;
            }
        }
        for(int i = 0; i < len; i++) p[i]--;
    }
    void getp(char *tmp, int n) {
        len = 0;
        for(int i = 0; i < n; i++) {
            s[len++] = '#';
            s[len++] = tmp[i];
        }
        s[len++] = '#';
        memset(p, 0, sizeof(int) * (len + 10));
        int c = 0, r = 0;
        for(int i = 0; i < len; i++) {
            if(i <= r) p[i] = min(p[2 * c - i], r - i);
            else p[i] = 1;
            while(i - p[i] >= 0 && i + p[i] < len && s[i - p[i]] == s[i + p[i]])
                p[i]++;
            if(i + p[i] - 1 > r) {
                r = i + p[i] - 1;
                c = i;
            }
        }
        for(int i = 0; i < len; i++) p[i]--;
    }
    int getlen() {
        return *max_element(p, p + len);
    }
    int getlen(string s) {
        getp(s);
        return getlen();
    }
}
```

AC 自动机

```cpp
struct ac_automaton {
    int t[N][26], danger[N], tot, fail[N];
    int dp[N][N];
    void init() {
```

```
        tot = -1;
        _new();
    }
    int _new() {
        tot++;
        memset(t[tot], 0, sizeof(t[tot]));
        danger[tot] = 0;
        fail[tot] = 0;
        return tot;
    }
    void Insert(const char *s) {
        int u = 0;
        for(int i = 0; s[i]; i++) {
            if(!t[u][mp[s[i]]]) t[u][s[i] - 'a'] = _new();
            u = t[u][mp[s[i]]];
        }
        danger[u] = 1;
    }
    void build() {
        queue<int> q;
        for(int i = 0; i < 26; i++) {
            if(t[0][i]) {
                fail[i] = 0;
                q.push(t[0][i]);
            }
        }
        while(q.size()) {
            int u = q.front(); q.pop();
            danger[u] |= danger[fail[u]];
            for(int i = 0; i < 26; i++) {
                if(t[u][i]) {
                    fail[t[u][i]] = t[fail[u]][i];
                    q.push(t[u][i]);
                } else t[u][i] = t[fail[u]][i];
            }
        }
    }
    int query(const char *s) {
        memset(dp, 0x3f, sizeof(dp));
        int n = strlen(s);
        dp[0][0] = 0;
        for(int i = 0; i < n; i++) {
            for(int j = 0; j <= tot; j++) if(!danger[j]) {
                for(int k = 0; k < 26; k++) if(!danger[t[j][k]]) {
                    dp[i + 1][t[j][k]] = min(dp[i + 1][t[j][k]], dp[i][j] + (s[i] - 'a' != k));
                }
            }
        }
        int ans = 0x3f3f3f3f;
        for(int i = 0; i <= tot; i++) if(!danger[i]) {
            ans = min(ans, dp[n][i]);
        }
        return ans == 0x3f3f3f3f ? -1 : ans;
    }
};
```

杂项

int128

```
typedef __uint128_t u128;
inline u128 read() {
    static char buf[100];
    scanf("%s", buf);
    // std::cin >> buf;
    u128 res = 0;
    for(int i = 0;buf[i];++i) {
        res = res << 4 | (buf[i] <= '9' ? buf[i] - '0' : buf[i] - 'a' + 10);
    }
    return res;
```

```
}
inline void output(u128 res) {
    if(res >= 16)
        output(res / 16);
    putchar(res % 16 >= 10 ? 'a' + res % 16 - 10 : '0' + res % 16);
    //std::cout.put(res % 16 >= 10 ? 'a' + res % 16 - 10 : '0' + res % 16);
}
```

Java,BigInteger

```
public BigInteger add(BigInteger val)   返回当前大整数对象与参数指定的大整数对象的和
public BigInteger subtract(BigInteger val)  返回当前大整数对象与参数指定的大整数对象的差
public BigInteger multiply(BigInteger val)   返回当前大整数对象与参数指定的大整数对象的积
public BigInteger devide(BigInteger val)   返回当前大整数对象与参数指定的大整数对象的商
public BigInteger remainder(BigInteger val)   返回当前大整数对象与参数指定的大整数对象的余
public int compareTo(BigInteger val)   返回当前大整数对象与参数指定的大整数对象的比较结果，返回值是 1、-1、0，分别表示当前大整数对象大于、小于或等于参数指定的大整数。
public BigInteger abs()   返回当前大整数对象的绝对值
public BigInteger pow(int exponent)   返回当前大整数对象的 exponent 次幂。
public String toString()   返回当前当前大整数对象十进制的字符串表示。
public String toString(int p)   返回当前大整数对象 p 进制的字符串表示。
public BigInteger negate()   返回当前大整数的相反数。
```

奇技淫巧

**_builtin_ 内建函数 **

- ~~__builtin_popcount(unsigned int n) 该函数是判断 n 的二进制中有多少个 1~~
- __builtin_parity(unsigned int n) 该函数是判断 n 的二进制中 1 的个数的奇偶性
- __builtin_ffs(unsigned int n) 该函数判断 n 的二进制末尾最后一个 1 的位置，从一开始
- __builtin_ctz(unsigned int n) 该函数判断 n 的二进制末尾后面 0 的个数，当 n 为 0 时，和 n 的类型有关
- __builtin_clz (unsigned int x) 返回前导的 0 的个数

真·popcount

```
int _popcount(int x) {
    return __builtin_popcount(x & (0ull - 1)) + __builtin_popcount(x >> 32);
}
```

随机数种子

```
srand(std :: chrono :: system_clock :: now().time_since_epoch().count());
```

T(5) 求任意 int log2

```
inline int LOG2_1(unsigned x){
    static const int tb[32]={0,9,1,10,13,21,2,29,11,14,16,18,22,25,3,30,8,12,20,28,15,17,24,7,19,27,23,6,26,5,4,31};
    x|=x>>1; x|=x>>2; x|=x>>4; x|=x>>8; x|=x>>16;
    return tb[x*0x07C4ACDDu>>27];
}
```

O(1) 求 2 的整幂次 log2

```
inline int LOG2(unsigned x){  //x=2^k
    static const int tb[32]={31,0,27,1,28,18,23,2,29,21,19,12,24,9,14,3,30,26,17,22,20,11,8,13,25,16,10,7,15,6,5,4};
    return tb[x*263572066>>27];
}
```

开启编译优化

```
作者：qwqwqer
链接：https://www.zhihu.com/question/264251178/answer/2155420801
来源：知乎
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

#pragma GCC optimize(2)
```

```
#pragma GCC optimize(3)
#pragma GCC optimize("Ofast")
#pragma GCC optimize("inline")
#pragma GCC optimize("-fgcse")
#pragma GCC optimize("-fgcse-lm")
#pragma GCC optimize("-fipa-sra")
#pragma GCC optimize("-ftree-pre")
#pragma GCC optimize("-ftree-vrp")
#pragma GCC optimize("-fpeephole2")
#pragma GCC optimize("-ffast-math")
#pragma GCC optimize("-fsched-spec")
#pragma GCC optimize("unroll-loops")
#pragma GCC optimize("-falign-jumps")
#pragma GCC optimize("-falign-loops")
#pragma GCC optimize("-falign-labels")
#pragma GCC optimize("-fdevirtualize")
#pragma GCC optimize("-fcaller-saves")
#pragma GCC optimize("-fcrossjumping")
#pragma GCC optimize("-fthread-jumps")
#pragma GCC optimize("-funroll-loops")
#pragma GCC optimize("-fwhole-program")
#pragma GCC optimize("-freorder-blocks")
#pragma GCC optimize("-fschedule-insns")
#pragma GCC optimize("inline-functions")
#pragma GCC optimize("-ftree-tail-merge")
#pragma GCC optimize("-fschedule-insns2")
#pragma GCC optimize("-fstrict-aliasing")
#pragma GCC optimize("-fstrict-overflow")
#pragma GCC optimize("-falign-functions")
#pragma GCC optimize("-fcse-skip-blocks")
#pragma GCC optimize("-fcse-follow-jumps")
#pragma GCC optimize("-fsched-interblock")
#pragma GCC optimize("-fpartial-inlining")
#pragma GCC optimize("no-stack-protector")
#pragma GCC optimize("-freorder-functions")
#pragma GCC optimize("-findirect-inlining")
#pragma GCC optimize("-fhoist-adjacent-loads")
#pragma GCC optimize("-frerun-cse-after-loop")
#pragma GCC optimize("inline-small-functions")
#pragma GCC optimize("-finline-small-functions")
#pragma GCC optimize("-ftree-switch-conversion")
#pragma GCC optimize("-foptimize-sibling-calls")
#pragma GCC optimize("-fexpensive-optimizations")
#pragma GCC optimize("-funsafe-loop-optimizations")
#pragma GCC optimize("inline-functions-called-once")
#pragma GCC optimize("-fdelete-null-pointer-checks")
```

快速乘

```
ll mul(ll x, ll y, ll mod){
    return (x * y - (ll)((long double)x / mod * y) * mod + mod) % mod;
}
ll mul(ll a, ll b, ll MOD) {
    __int128 x = a, y = b, m = MOD;
    return (ll)(x * y % m);
}
```

子集枚举

枚举 s 的子集

```
for(int i = s; i; i = (i - 1) & s))
```

枚举所有大小为 r 的集合

```
for(int s = (1 << r) - 1; s < (1 << n); ) {
    int x = s & -s;
    int y = s + x;
    s = ((y ^ s) >> __builtin_ctz(x) + 2) | y;
}
```

mt19937_64 随机数生成器

```cpp
std::mt19937_64 rng(std::chrono::steady_clock::now().time_since_epoch().count());
template <typename T>
T rd(T l, T r) {
    std::uniform_int_distribution<T> u(l, r);
    return u(rng);
}
template <>
double rd<double>(double l, double r) {
    std::uniform_real_distribution<double> u(l, r);
    return u(rng);
}
```

tips:

- 如果使用 sort 比较两个函数，不能出现 $a < b$ 和 $a > b$ 同时为真的情况，否则会运行错误。
- 多组数据清空线段树的时候，不要忘记清空全部数组（比如说 lazytag 数组）。
- 注意树的深度和节点到根的距离是两个不同的东西，深度是点数，距离是边长，如果求 LCA 时用距离算会出错。
- 连通性专题：注意判断 $dfn[x]$ 和 $low[y]$ 的关系时是否不小心两个都达成 low 了
- 推不等式确定范围的时候，仅需要考虑所有不等式限定的范围，然后判断左端点是否大于右端点，不要加额外的臆想条件。
- 矩阵快速幂如果常数十分大的时候，可以考虑 unordered_map 保存结果，可以明显加速。
- **__builtin_popcount** 只支持 unsigned int 型，不支持 long long!!!!!!!!