XCPC 模板合集

北京邮电大学　孙明志

2024 年 3 月 7 日

目录

1 图论

1.1 确定有限状态自动机最小化

```
const int maxn = 110000, sigma_size = 20;
struct Automaton {
    int n, m, cur, G[maxn][sigma_size];
    int inq[maxn], cls[maxn];
    vector<int> from[maxn][sigma_size];
    unordered_set<int> equiv[maxn]; //equiv[i] 表示编号为 i 的等价类的集合
    void init(int n, int m, vector<bool> final) { //状态数、输入字符集大小、每个结点是否为终态
        //如果要多次使用该算法，需要在此处先将状态清空。
        this->n = n;
        this->m = m;
        this->cur = 2;
        for (int i = 0; i < n; ++i) {
            int st = final[i] ^ 1;
            equiv[st].insert(i);
            cls[i] = st;
        }
    }
    void add_edge(int x, int c, int y) { //添加一条从 x 指向 y 的边，字符为 c
        G[x][c] = y;
    }
    int minimize() { //调用该方法之前应该先将不可达状态从自动机中删除
        for (int i = 0; i < n; ++i)
            for (int c = 0; c < m; ++c)
                from[G[i][c]][c].push_back(i);
        queue<int> Q;
        Q.push(0); inq[0] = true;
        while (!Q.empty()) {
            int x = Q.front(); Q.pop(); //x 是一个等价类
            inq[x] = false;
            for (int c = 0; c < m; ++c) {
                unordered_map<int, vector<int>> par; //class -> set of states
                for (auto i : equiv[x]) for (auto u : from[i][c])
                    //if (cls[u] != x)
                    par[cls[u]].push_back(u);
                for (auto &[id, member] : par) if (member.size() != equiv[id].size()) {
                    int now = cur++;
                    for (auto y : member) {
                        equiv[id].erase(y);
                        equiv[now].insert(y);
                        cls[y] = now;
                    }
                    if (inq[id] || equiv[now].size() < equiv[id].size())
                        Q.push(now), inq[now] = true;
```

```
                    else
                        Q.push(id), inq[id] = true;
                }
            }
        }
        return cur; //返回合并之后的总状态数
    }
} am;
int main() {
    int b, m;
    scanf("%d %d", &b, &m);
    vector<bool> final(m, 0); final[0] = 1;
    am.init(m, b, move(final));
    for (int i = 0; i < m; ++i)
        for (int j = 0; j < b; ++j)
            am.add_edge(i, j, (i * b + j) % m);
    int sz = am.minimize();
    printf("%d 0\n", sz);
    printf("G");
    for (int i = 1; i < sz; ++i)
        printf(" B");
    printf("\n");
    for (int i = 0; i < sz; ++i) {
        for (int j = 0; j < b; ++j) {
            int first = *am.equiv[i].begin();
            printf("%d ", am.cls[(first * b + j) % m]);
        }
        printf("\n");
    }
    return 0;
}
```

1.2 拓扑排序

```
const int maxn = 110000;
int n, m, degree[maxn];
vector<int> G[maxn];
vector<int> toposort1() {
    queue<int> Q;
    for (int i = 1; i <= n; ++i)
        degree[i] = 0;
    for (int x = 1; x <= n; ++x)
        for (auto y : G[x])
            degree[y]++;
    for (int i = 1; i <= n; ++i) if (degree[i] == 0)
        Q.push(i);
    vector<int> res;
```

```
    while (!Q.empty()) {
        int x = Q.front(); Q.pop();
        res.push_back(x);
        for (auto y : G[x]) {
            degree[y]--;
            if (degree[y] == 0)
                Q.push(y);
        }
    }
    return res;
}
vector<int> result;
int vis[maxn];
void dfs(int x) {
    vis[x] = true;
    for (auto y : G[x]) if (!vis[y])
        dfs(y);
    result.push_back(x);
}
vector<int> toposort2() { //必须保证是有向无环图
    memset(vis, 0, sizeof(vis));
    for (int i = 1; i <= n; ++i) if (!vis[i])
        dfs(i);
    reverse(result.begin(), result.end());
    return result;
}
int main() { //uva 10305
    //freopen("in.txt", "r", stdin);
    while (scanf("%d %d", &n, &m) == 2) {
        if (n == 0 && m == 0)
            break;
        for (int i = 1; i <= n; ++i)
            G[i].clear();
        for (int i = 0; i < m; ++i) {
            int x, y;
            scanf("%d %d", &x, &y);
            G[x].push_back(y);
        }
        auto ans = toposort2();
        printf("%d", ans[0]);
        for (int i = 1; i < n; ++i)
            printf(" %d", ans[i]);
        printf("\n");
    }
    return 0;
}
```

1.3 欧拉回路

```
/*
1. 无向图有欧拉路径的充要条件是：图联通且至多有两个奇点。
   若不存在奇点，则有欧拉回路。
2. 有向图有欧拉路径的充要条件是：至多有两个点的入度不等于出度，
   且必须是一个点的出度比入度大 1（起点），一个点的入度比出度大 1（终点），
   且忽略边的方向后图必须连通。
*/
const int maxn = 110000;
int t, n, m;
namespace direct_graph {
    //若有多组数据应将 G 和 result 清空，调用完 euler 之后 G 中的边会被删除
    vector<int> G[maxn];
    vector<pair<int, int>> result;
    void dfs(int x) {
        while (G[x].size()) {
            auto y = G[x].back();
            G[x].pop_back();
            dfs(y);
            result.emplace_back(x, y);
        }
    }
    bool euler(int s) {
        dfs(s);
        reverse(result.begin(), result.end());
        if (result.size() != m || (result.size() && result.back().second != s))
            return false; //若只要求有欧拉道路则: if (result.size() != m) return false;
        for (int i = 1; i < result.size(); ++i) if (result[i - 1].second != result[i].first)
            return false;
        return true;
    }
    void solve() {
        multimap<pair<int, int>, int> id;
        for (int i = 1; i <= m; ++i) {
            int x, y;
            scanf("%d %d", &x, &y);
            G[x].push_back(y);
            id.emplace(make_pair(x, y), i);
        }
        int s = 1; while (G[s].empty() && s <= n) s++;
        if (!euler(s)) {
            puts("NO");
            return;
        }
        else {
            puts("YES");
```

```
        for (auto pr : result) {
            int x = pr.first, y = pr.second;
            auto iter = id.find({ x, y });
            if (iter == id.end()) {
                iter = id.find({ y, x });
                printf("%d ", -iter->second);
            }
            else
                printf("%d ", iter->second);
            id.erase(iter);
        }
    }
}
}
namespace undirect_graph {
    //若有多组数据应将 G 和 result 清空，调用完 euler 之后 G 中的边会被删除
    multiset<int> G[maxn];
    vector<pair<int, int>> result;
    void dfs(int x) {
        while (G[x].size()) {
            auto iter = G[x].begin();
            auto y = *iter;
            G[x].erase(iter);
            G[y].erase(G[y].find(x));
            dfs(y);
            result.emplace_back(x, y);
        }
    }
    bool euler(int s) {
        dfs(s);
        reverse(result.begin(), result.end());
        if (result.size() != m || (result.size() && result.back().second != s))
            return false; //若只要求有欧拉道路则: if (result.size() != m) return false;
        for (int i = 1; i < result.size(); ++i) if (result[i - 1].second != result[i].first)
            return false;
        return true;
    }
    void solve() {
        multimap<pair<int, int>, int> id;
        for (int i = 1; i <= m; ++i) {
            int x, y;
            scanf("%d %d", &x, &y);
            G[x].insert(y);
            G[y].insert(x);
            id.emplace(make_pair(x, y), i);
        }
```

```
        int s = 1; while (G[s].empty() && s <= n) s++;
        if (!euler(s)) {
            puts("NO");
            return;
        }
        else {
            puts("YES");
            for (auto pr : result) {
                int x = pr.first, y = pr.second;
                auto iter = id.find({ x, y });
                if (iter == id.end()) {
                    iter = id.find({ y, x });
                    printf("%d ", -iter->second);
                }
                else
                    printf("%d ", iter->second);
                id.erase(iter);
            }
        }
    }
}
int main() { //uoj 117
    //freopen("in.txt", "r", stdin);
    scanf("%d %d %d", &t, &n, &m);
    if (t == 2) { //有向图
        direct_graph::solve();
    }
    else {
        undirect_graph::solve();
    }
    return 0;
}
```

1.4 最小生成树

```
const int maxn = 210000;
int n, m, from[maxn], to[maxn], weight[maxn];
int p[maxn], r[maxn];
int find(int x) {
    return p[x] == x ? x : p[x] = find(p[x]);
}
int kruskal() {
    for (int i = 1; i <= n; ++i)
        p[i] = i;
    for (int i = 1; i <= m; ++i)
        r[i] = i;
    sort(r + 1, r + m + 1, [](int a, int b) {
```

```
        return weight[a] < weight[b];
    });
    int ans = 0, edges = 0;
    for (int i = 1; i <= m; ++i) {
        int e = r[i];
        int x = find(from[e]), y = find(to[e]);
        if (x != y) {
            p[x] = y;
            ans += weight[e];
            edges += 1;
        }
    }
    if (edges != n - 1)
        return -1; //无解
    else
        return ans;
}
int main() { //洛谷 P3366
    //freopen("in.txt", "r", stdin);
    scanf("%d %d", &n, &m);
    for (int i = 1; i <= m; ++i)
        scanf("%d %d %d", &from[i], &to[i], &weight[i]);
    int ans = kruskal();
    if (ans == -1)
        puts("orz");
    else
        printf("%d\n", ans);
    return 0;
}
```

1.5 Dijkstra 算法

```
int n, m, s; //点数、边数、起点
namespace simple_dijkstra { // O(n^2)
    const int maxn = 1001;
    int vis[maxn], d[maxn], w[maxn][maxn];
    void dijkstra() {
        memset(vis, 0, sizeof(vis));
        memset(d, 0x3f, sizeof(d));
        d[s] = 0;
        for (int i = 1; i <= n; ++i) {
            int x = -1;
            for (int y = 1; y <= n; ++y) if (!vis[y])
                if (x == -1 || d[y] < d[x])
                    x = y;
            vis[x] = true;
            for (int y = 1; y <= n; ++y)
```

```
                d[y] = min(d[y], d[x] + w[x][y]);
        }
    }
    void solve() {
        scanf("%d %d %d", &n, &m, &s);
        memset(w, 0x3f, sizeof(w));
        for (int i = 0; i < m; ++i) {
            int x, y, z;
            scanf("%d %d %d", &x, &y, &z);
            w[x][y] = min(w[x][y], z);
        }
        dijkstra();
        for (int i = 1; i <= n; ++i)
            printf("%d ", d[i] == 0x3f3f3f3f ? INT_MAX : d[i]);
        printf("\n");
    }
}
namespace fast_dijkstra {
    const int maxn = 210000;
    vector<pair<int, int>> G[maxn];
    int d[maxn];
    void dijkstra() {
        using node = pair<int, int>;
        priority_queue<node, vector<node>, greater<node>> Q;
        memset(d, 0x3f, sizeof(d));
        d[s] = 0;
        Q.emplace(0, s);
        while (!Q.empty()) {
            auto [dist, x] = Q.top(); Q.pop();
            if (dist != d[x])
                continue;
            for (auto [y, w] : G[x]) {
                if (d[y] > d[x] + w) {
                    d[y] = d[x] + w;
                    Q.emplace(d[y], y);
                    //p[y] = x;
                }
            }
        }
    }
    void solve() {
        scanf("%d %d %d", &n, &m, &s);
        for (int i = 0; i < m; ++i) {
            int x, y, z;
            scanf("%d %d %d", &x, &y, &z);
            G[x].emplace_back(y, z);
```

```
        }
        dijkstra();
        for (int i = 1; i <= n; ++i)
            printf("%d ", d[i] == 0x3f3f3f3f ? INT_MAX : d[i]);
        printf("\n");
    }
}
int main() {
    //freopen("in.txt", "r", stdin);
    fast_dijkstra::solve();
    return 0;
}
```

1.6 SPFA 算法

```
int n, m, s; //点数、边数、起点
namespace BellmanFord { // O(nm)
    const int maxn = 1001;
    const int maxedges = 110000;
    int d[maxn], x[maxedges], y[maxedges], w[maxedges];
    void BellmanFord() {
        memset(d, 0x3f, sizeof(d));
        d[s] = 0;
        for (int k = 1; k < n; ++k)
            for (int i = 1; i <= m; ++i)
                d[y[i]] = min(d[y[i]], d[x[i]] + w[i]);
    }
    void solve() {
        scanf("%d %d %d", &n, &m, &s);
        for (int i = 1; i <= m; ++i)
            scanf("%d %d %d", &x[i], &y[i], &w[i]);
        BellmanFord();
        for (int i = 1; i <= n; ++i)
            printf("%d ", d[i] == 0x3f3f3f3f ? INT_MAX : d[i]);
        printf("\n");
    }
}
namespace SPFA {
    const int maxn = 210000;
    int inq[maxn], cnt[maxn], d[maxn];
    vector<pair<int, int>> G[maxn];
    bool spfa() {
        queue<int> Q;
        memset(inq, 0, sizeof(inq));
        memset(cnt, 0, sizeof(cnt));
        memset(d, 0x3f, sizeof(d));
        d[s] = 0;
```

```
        inq[s] = true;
        Q.push(s);
        while (!Q.empty()) {
            int x = Q.front(); Q.pop();
            inq[x] = false;
            for (auto [y, w] : G[x]) {
                if (d[y] > d[x] + w) {
                    d[y] = d[x] + w;
                    //p[y] = x;
                    if (!inq[y]) {
                        Q.push(y);
                        inq[y] = true;
                        if (++cnt[y] > n)
                            return false;
                    }
                }
            }
        }
        return true;
    }
    void solve() {
        scanf("%d %d %d", &n, &m, &s);
        for (int i = 1; i <= m; ++i) {
            int x, y, w;
            scanf("%d %d %d", &x, &y, &w);
            G[x].emplace_back(y, w);
        }
        spfa();
        for (int i = 1; i <= n; ++i)
            printf("%d ", d[i] == 0x3f3f3f3f ? INT_MAX : d[i]);
        printf("\n");
    }
}
int main() {
    //freopen("in.txt", "r", stdin);
    SPFA::solve();
    return 0;
}
```

1.7 一般图最大权匹配

```
#define dist(e) (lab[e.u] + lab[e.v] - g[e.u][e.v].w * 2)
const int maxn = 1023, inf = 1e9;
struct edge {
    int u, v, w;
} g[maxn][maxn];
int n, m, num, lab[maxn], match[maxn], slack[maxn], st[maxn], pa[maxn];
```

```
int from[maxn][maxn], S[maxn], vis[maxn];
vector<int> flower[maxn];
deque<int> q;
void update(int u, int x) {
    if (!slack[x] || dist(g[u][x]) < dist(g[slack[x]][x]))
        slack[x] = u;
}
void set_slack(int x) {
    slack[x] = 0;
    for (int u = 1; u <= n; ++u)
        if (g[u][x].w > 0 && st[u] != x && S[st[u]] == 0)
            update(u, x);
}
void push(int x) {
    if (x <= n)
        return q.push_back(x);
    for (int i = 0; i < flower[x].size(); i++)
        push(flower[x][i]);
}
void set_st(int x, int b) {
    st[x] = b;
    if (x <= n) return;
    for (int i = 0; i < flower[x].size(); ++i)
        set_st(flower[x][i], b);
}
int get(int b, int xr) {
    int pr = find(flower[b].begin(), flower[b].end(), xr) - flower[b].begin();
    if (pr % 2 == 1) {
        reverse(flower[b].begin() + 1, flower[b].end());
        return (int)flower[b].size() - pr;
    }
    else
        return pr;
}
void set_match(int u, int v) {
    match[u] = g[u][v].v;
    if (u <= n) return;
    edge e = g[u][v];
    int xr = from[u][e.u], pr = get(u, xr);
    for (int i = 0; i < pr; ++i)
        set_match(flower[u][i], flower[u][i ^ 1]);
    set_match(xr, v);
    rotate(flower[u].begin(), flower[u].begin() + pr, flower[u].end());
}
void augment(int u, int v) {
    int xnv = st[match[u]];
```

```
    set_match(u, v);
    if (!xnv) return;
    set_match(xnv, st[pa[xnv]]);
    augment(st[pa[xnv]], xnv);
}
int get_lca(int u, int v) {
    static int t = 0;
    for (++t; u || v; swap(u, v)) {
        if (u == 0) continue;
        if (vis[u] == t) return u;
        vis[u] = t;
        u = st[match[u]];
        if (u)u = st[pa[u]];
    }
    return 0;
}
void add(int u, int lca, int v) {
    int b = n + 1;
    while (b <= num && st[b]) ++b;
    if (b > num) ++num;
    lab[b] = 0, S[b] = 0;
    match[b] = match[lca];
    flower[b].clear();
    flower[b].push_back(lca);
    for (int x = u, y; x != lca; x = st[pa[y]])
        flower[b].push_back(x), flower[b].push_back(y = st[match[x]]), push(y);
    reverse(flower[b].begin() + 1, flower[b].end());
    for (int x = v, y; x != lca; x = st[pa[y]])
        flower[b].push_back(x), flower[b].push_back(y = st[match[x]]), push(y);
    set_st(b, b);
    for (int x = 1; x <= num; ++x)
        g[b][x].w = g[x][b].w = 0;
    for (int x = 1; x <= n; ++x)
        from[b][x] = 0;
    for (int i = 0; i < flower[b].size(); ++i) {
        int xs = flower[b][i];
        for (int x = 1; x <= num; ++x)
            if (g[b][x].w == 0 || dist(g[xs][x]) < dist(g[b][x]))
                g[b][x] = g[xs][x], g[x][b] = g[x][xs];
        for (int x = 1; x <= n; ++x)
            if (from[xs][x])
                from[b][x] = xs;
    }
    set_slack(b);
}
void expand(int b) {
```

```
    for (int i = 0; i < flower[b].size(); ++i)
        set_st(flower[b][i], flower[b][i]);
    int xr = from[b][g[b][pa[b]].u], pr = get(b, xr);
    for (int i = 0; i < pr; i += 2) {
        int xs = flower[b][i], xns = flower[b][i + 1];
        pa[xs] = g[xns][xs].u;
        S[xs] = 1, S[xns] = 0;
        slack[xs] = 0, set_slack(xns);
        push(xns);
    }
    S[xr] = 1, pa[xr] = pa[b];
    for (int i = pr + 1; i < flower[b].size(); ++i) {
        int xs = flower[b][i];
        S[xs] = -1, set_slack(xs);
    }
    st[b] = 0;
}
bool found(const edge& e) {
    int u = st[e.u], v = st[e.v];
    if (S[v] == -1) {
        pa[v] = e.u, S[v] = 1;
        int nu = st[match[v]];
        slack[v] = slack[nu] = 0;
        S[nu] = 0, push(nu);
    }
    else if (S[v] == 0) {
        int lca = get_lca(u, v);
        if (!lca) return augment(u, v), augment(v, u), true;
        else add(u, lca, v);
    }
    return false;
}
bool matching() {
    fill(S, S + num + 1, -1);
    fill(slack, slack + num + 1, 0);
    q.clear();
    for (int x = 1; x <= num; ++x)
        if (st[x] == x && !match[x])pa[x] = 0, S[x] = 0, push(x);
    if (q.empty()) return false;
    for (;;) {
        while (q.size()) {
            int u = q.front();
            q.pop_front();
            if (S[st[u]] == 1) continue;
            for (int v = 1; v <= n; ++v) {
                if (g[u][v].w > 0 && st[u] != st[v]) {
```

```
                    if (dist(g[u][v]) == 0) {
                        if (found(g[u][v]))
                            return true;
                    }
                    else update(u, st[v]);
                }
            }
        }
        int d = inf;
        for (int b = n + 1; b <= num; ++b)
            if (st[b] == b && S[b] == 1)d = min(d, lab[b] / 2);
        for (int x = 1; x <= num; ++x)
            if (st[x] == x && slack[x])
            {
                if (S[x] == -1)d = min(d, dist(g[slack[x]][x]));
                else if (S[x] == 0)d = min(d, dist(g[slack[x]][x]) / 2);
            }
        for (int u = 1; u <= n; ++u) {
            if (S[st[u]] == 0) {
                if (lab[u] <= d)
                    return false;
                lab[u] -= d;
            }
            else if (S[st[u]] == 1) lab[u] += d;
        }
        for (int b = n + 1; b <= num; ++b)
            if (st[b] == b) {
                if (S[st[b]] == 0) lab[b] += d * 2;
                else if (S[st[b]] == 1) lab[b] -= d * 2;
            }
        q.clear();
        for (int x = 1; x <= num; ++x)
            if (st[x] == x && slack[x] && st[slack[x]] != x && dist(g[slack[x]][x]) == 0)
                if (found(g[slack[x]][x]))
                    return true;
        for (int b = n + 1; b <= num; ++b)
            if (st[b] == b && S[b] == 1 && lab[b] == 0)
                expand(b);
    }
    return false;
}
pair<long long, int> weight_blossom() {
    fill(match, match + n + 1, 0);
    num = n;
    int matches = 0;
    long long tot_weight = 0;
```

```
    for (int u = 0; u <= n; ++u) st[u] = u, flower[u].clear();
    int w_max = 0;
    for (int u = 1; u <= n; ++u)
        for (int v = 1; v <= n; ++v) {
            from[u][v] = (u == v ? u : 0);
            w_max = max(w_max, g[u][v].w);
        }
    for (int u = 1; u <= n; ++u) lab[u] = w_max;
    while (matching()) ++matches;
    for (int u = 1; u <= n; ++u)
        if (match[u] && match[u] < u)
            tot_weight += g[u][match[u]].w;
    return make_pair(tot_weight, matches);
}
int main() { //边权必须是正数
    cin >> n >> m;
    for (int u = 1; u <= n; ++u)
        for (int v = 1; v <= n; ++v)
            g[u][v] = edge{ u,v,0 };
    for (int i = 0, u, v, w; i < m; ++i) {
        cin >> u >> v >> w;
        g[u][v].w = g[v][u].w = w;
    }
    cout << weight_blossom().first << '\n';
    for (int u = 1; u <= n; ++u) cout << match[u] << ' ';
}
```

1.8 有向图的传递闭包

```
namespace Closure { //求有向图的传递闭包
    const int n = 4000; //对于 n = 4000, 可以在大约 1s 的时间内求解
    bitset<n> A[n];
    void solve(int M[][n]) { //M 是一个 01 矩阵, M[i][j] 为 1 当且仅当有一条从 i 到 j 的有向边
        for (int i = 0; i < n; ++i)
            for (int j = 0; j < n; ++j)
                A[i][j] = M[i][j];
        for (int k = 0; k < n; ++k)
            for (int i = 0; i < n; ++i)
                if (A[i][k])
                    A[i] |= A[k];
        for (int i = 0; i < n; ++i)
            for (int j = 0; j < n; ++j)
                M[i][j] = A[i][j]; //将结果复制回输入矩阵中
    }
}
const int maxn = 4000;
int A[maxn][maxn], B[maxn][maxn];
```

```
int main() {
    for (int i = 0; i < maxn; ++i)
        for (int j = 0; j < maxn; ++j)
            A[i][j] = B[i][j] = rand() % 2;
    clock_t start = clock();
    for (int k = 0; k < maxn; ++k)
        for (int i = 0; i < maxn; ++i)
            for (int j = 0; j < maxn; ++j)
                B[i][j] = (B[i][j] || (B[i][k] && B[k][j]));
    clock_t end = clock();
    printf("%.5f\n", (double)(end - start) / CLOCKS_PER_SEC);
    start = clock();
    Closure::solve(A);
    end = clock();
    printf("%.5f\n", (double)(end - start) / CLOCKS_PER_SEC);
    for (int i = 0; i < maxn; ++i)
        for (int j = 0; j < maxn; ++j)
            if (A[i][j] != B[i][j])
                abort();
    return 0;
}
```

2 树算法

2.1 限定距离的子树问题

```
/*
该算法用来求解有根树中结点 v 的子树中与 v 距离不超过 d 的所有结点的权值和（也可以求最值等），
时间复杂度 O(d)，若用倍增可以优化到 O(logd)。
调用 init 函数之前应当输入 n（结点标号从 1 开始），以及用 add_edge 建图。
*/
const int maxn = 1010000;
int n, q, w[maxn], head[maxn], nxt[maxn * 2], to[maxn * 2], id[maxn], l[maxn], r[maxn];
int father[maxn], depth[maxn], seq[maxn], cur = 0;
long long sum[maxn];
void add_edge(int u, int v) {
    to[++cur] = v;
    nxt[cur] = head[u];
    head[u] = cur;
}
void init() {
    int sz = 0;
    queue<int> Q;
    father[1] = depth[1] = seq[n + 1] = 0;
    Q.push(1);
    while (!Q.empty()) {
        int x = Q.front(); Q.pop();
```

```
        seq[++sz] = x;
        id[x] = sz;
        l[x] = r[x] = 0;
        for (int i = head[x]; i; i = nxt[i]) if (to[i] != father[x]) {
            int y = to[i];
            l[x] = (l[x] ? l[x] : y);
            r[x] = y;
            father[y] = x;
            depth[y] = depth[x] + 1;
            Q.push(y);
        }
    }
    for (int j = 1; j <= n; ++j)
        sum[j] = sum[j - 1] + w[seq[j]];
    for (int j = 1; j <= n; ++j) if (!r[seq[j]])
        r[seq[j]] = r[seq[j - 1]];
    for (int j = n; j >= 1; --j) if (!l[seq[j]])
        l[seq[j]] = l[seq[j + 1]];
}
long long travel(int x, int d) { //求出以 x 为根的子树中与 x 距离不超过 d 的所有结点的权值和（可改
↪ 为求最值或种类数）
    if (d < 0) return 0;          //时间复杂度 O(d)，可以改成倍增使得时间复杂度变为 O(logd)
    long long ret = 0;
    int L = x, R = x;
    for (int i = 0; i <= d; ++i) {
        int left = id[L], right = id[R];
        if (L == 0 || R == 0 || left > right) break;
        long long s = sum[right] - sum[left - 1];
        ret += s;
        L = l[L];
        R = r[R];
    }
    return ret;
}
int main() {
    //freopen("in.txt", "r", stdin);
    scanf("%d", &n);
    for (int i = 1; i <= n; ++i)
        scanf("%d", &w[i]);
    for (int i = 0; i < n - 1; ++i) {
        int u, v;
        scanf("%d %d", &u, &v);
        add_edge(u, v);
        add_edge(v, u);
    }
    init();
```

```
    scanf("%d", &q);
    while (q--) { //每次询问求出树中与 v 距离不超过 k 的所有点的权值和，复杂度 O(k^2)
        int v, k;
        scanf("%d %d", &v, &k);
        long long ans = travel(v, k);
        int last = v;
        for (int u = father[v], t = k - 1; u != 0 && t >= 0; u = father[u], --t) {
            ans += travel(u, t) - travel(last, t - 1);
            last = u;
        }
        printf("%lld\n", ans);
    }
    return 0;
}
```

2.2 树的欧拉序

```
/*
seq 下标从 1 开始，表示欧拉序列。欧拉序列记录的是点的标号，一段欧拉序列中若一个点出现了两次则不予处理。
树上的一条路径一定对应一段连续的欧拉序列（可能还要并上一个额外的点）。
*/
const int maxn = 210000;
const int maxlog = 25;
int head[maxn], nxt[maxn], to[maxn], seq[maxn], depth[maxn], anc[maxn][maxlog], first[maxn],
↪ last[maxn], cur, sz;
struct item { //下标区间 [L, R] 的欧拉序列，若 lca != 0 则还要将 lca 考虑进来
    int L, R, lca;
    item(){}
    item(int L, int R, int lca) : L(L), R(R), lca(lca){}
};
void init() {
    cur = sz = 0;
    memset(head, 0, sizeof(head));
    memset(depth, 0, sizeof(depth));
    memset(anc, 0, sizeof(anc));
}
void add_edge(int x, int y) { //对于树中的每条边要调用两次 add_edge
    to[++cur] = y;
    nxt[cur] = head[x];
    head[x] = cur;
}
void dfs(int x, int fa) { //dfs(1, 0)
    seq[++sz] = x;
    first[x] = sz;
    for (int i = head[x]; i; i = nxt[i]) if (to[i] != fa) {
        int y = to[i];
        depth[y] = depth[x] + 1;
```

```
        anc[y][0] = x;
        for (int i = 1; (1 << i) <= depth[y]; ++i)
            anc[y][i] = anc[anc[y][i - 1]][i - 1];
        dfs(y, x);
    }
    seq[++sz] = x;
    last[x] = sz;
}
int getlca(int x, int y) {
    if (depth[x] < depth[y])
        swap(x, y);
    for (int i = maxlog - 1; i >= 0; --i)
        if (depth[x] - (1 << i) >= depth[y])
            x = anc[x][i];
    if (x == y)
        return x;
    for (int i = maxlog - 1; i >= 0; --i)
        if (anc[x][i] != anc[y][i])
            x = anc[x][i], y = anc[y][i];
    return anc[x][0];
}
item path(int x, int y) { //返回树上 x 到 y 的路径对应的欧拉序列
    if (first[x] > first[y])
        swap(x, y);
    int lca = getlca(x, y);
    if (lca == x)
        return item(first[x], first[y], 0);
    else
        return item(last[x], first[y], lca);
}
int main() {
    return 0;
}
```

2.3 倍增求 LCA

```
const int maxn = 510000;
const int maxlog = 20;
vector<int> G[maxn];
int anc[maxn][maxlog], dep[maxn];
void dfs(int x, int fa, int d) {
    anc[x][0] = fa;
    dep[x] = d;
    for (auto y : G[x]) if (y != fa)
        dfs(y, x, d + 1);
}
void preprocess(int n) { //点的编号从 1 开始
```

```
    for (int j = 1; j < maxlog; ++j)
        for (int i = 0; i <= n; ++i)
            anc[i][j] = 0;
    dfs(1, 0, 0);
    for (int j = 1; j < maxlog; ++j)
        for (int i = 1; i <= n; ++i)
            anc[i][j] = anc[anc[i][j - 1]][j - 1];
}
//返回结点 x 向上走 d 步到达的结点
int moveup(int x, int d) {
    for (int i = 0; d >> i; ++i)
        if (d >> i & 1)
            x = anc[x][i];
    return x;
}
int lca(int x, int y) {
    if (dep[x] < dep[y])
        swap(x, y);
    x = moveup(x, dep[x] - dep[y]);
    if (x == y)
        return x;
    for (int i = maxlog - 1; i >= 0; --i)
        if (anc[x][i] != anc[y][i])
            x = anc[x][i], y = anc[y][i];
    return anc[x][0];
}
int dist(int x, int y) { //返回结点 x 和 y 之间的距离
    return dep[x] + dep[y] - 2 * dep[lca(x, y)];
}
int move(int x, int y, int d) { //返回从结点 x 向结点 y 走 d 步到达的结点
    int p = lca(x, y);
    int h = dep[x] - dep[p];
    if (h >= d)
        return moveup(x, d);
    else
        return moveup(y, dep[x] + dep[y] - d - 2 * dep[p]);
}
int main() {
    //freopen("in.txt", "r", stdin);
    int n, m, s;
    scanf("%d %d %d", &n, &m, &s);
    for (int i = 1; i < n; ++i) {
        int x, y;
        scanf("%d %d", &x, &y);
        G[x].push_back(y);
        G[y].push_back(x);
```

```
    }
    dfs(s, 0, 0);
    for (int j = 1; j < maxlog; ++j)
        for (int i = 1; i <= n; ++i)
            anc[i][j] = anc[anc[i][j - 1]][j - 1];
    for (int i = 0; i < m; ++i) {
        int x, y;
        scanf("%d %d", &x, &y);
        printf("%d\n", lca(x, y));
    }
    return 0;
}
```

2.4 点分治

```
#define dfs1(x, fa, d)
#define dfs2(x, fa, d)
const int maxn = 110000;
vector<pair<int, int>> G[maxn]; //to, weight
int vis[maxn], siz[maxn], f[maxn], pos, sum;
void init(int n) {
    for (int i = 0; i <= n; ++i) {
        vis[i] = false;
        G[i].clear();
    }
}
void getroot(int x, int fa) {
    f[x] = 0; siz[x] = 1;
    for (auto[y, w] : G[x]) if (y != fa && !vis[y]) {
        getroot(y, x);
        f[x] = max(f[x], siz[y]);
        siz[x] += siz[y];
    }
    f[x] = max(f[x], sum - siz[x]);
    if (f[x] < f[pos])
        pos = x;
}
void calc(int x) { //统计经过结点 x 的所有答案
    for (auto [y, w] : G[x]) if (!vis[y]) {
        //注意在 dfs 中枚举子结点 y 的时候要判两个条件: y != fa && !vis[y]
        dfs1(y, x, w); //计算当前子树与之前子树以及 结点 x 连接构成的路径。
        dfs2(y, x, w); //维护处理过的子树的信息
    }
    //维护信息的时候有两种方案:
    //1. 在每次 calc 调用的时候新建一个数据结构维护。
    //2. 维护一个全局的数据结构，此时在 calc 末尾应当加一个循环来撤销之前的所有操作。
}
```

```
void solve(int x, int cnt) { //cnt 为子树 x 中的结点总数
    pos = maxn - 1; f[pos] = sum = cnt;
    getroot(x, -1);
    int root = pos; //因为 pos 是全局变量，递归的时候值会改变，所以此处存为局部变量。
    vis[root] = 1;
    calc(root);
    for (auto [y, w] : G[root]) if (!vis[y])
        solve(y, siz[y] < siz[root] ? siz[y] : cnt - siz[root]);
}
int main() {
    int n;
    scanf("%d", &n);
    init(n);
    for (int i = 1; i < n; ++i) {
        int x, y, w;
        scanf("%d %d %d", &x, &y, &w);
        G[x].emplace_back(y, w);
        G[y].emplace_back(x, w);
    }
    solve(1, n);
    return 0;
}
```

2.5 动态点分治

```
#define dfs(x, fa, d)
const int maxn = 110000;
vector<pair<int, int>> G[maxn]; //to, weight
int vis[maxn], siz[maxn], f[maxn], pa[maxn], pos, sum; //pa[x] 表示在点分树上结点 x 的父亲
int son[maxn], dep[maxn], father[maxn], top[maxn];
void DFS1(int x, int fa, int d) {
    dep[x] = d;
    siz[x] = 1;
    son[x] = 0;
    father[x] = fa;
    for (auto [y, w] : G[x]) if (y != fa) {
        DFS1(y, x, d + 1);
        siz[x] += siz[y];
        if (siz[son[x]] < siz[y])
            son[x] = y;
    }
}
void DFS2(int x, int tp) {
    top[x] = tp;
    if (son[x])
        DFS2(son[x], tp);
    for (auto [y, w] : G[x])
```

```
        if (y != father[x] && y != son[x])
            DFS2(y, y);
}
int lca(int x, int y) {
    while (top[x] != top[y]) {
        if (dep[top[x]] < dep[top[y]])
            swap(x, y);
        x = father[top[x]];
    }
    return dep[x] < dep[y] ? x : y;
}
int dist(int x, int y) {
    return dep[x] + dep[y] - 2 * dep[lca(x, y)];
}
void init(int n) {
    for (int i = 0; i <= n; ++i) {
        vis[i] = false;
        G[i].clear();
    }
}
void getroot(int x, int fa) {
    f[x] = 0; siz[x] = 1;
    for (auto [y, w] : G[x]) if (y != fa && !vis[y]) {
        getroot(y, x);
        f[x] = max(f[x], siz[y]);
        siz[x] += siz[y];
    }
    f[x] = max(f[x], sum - siz[x]);
    if (f[x] < f[pos])
        pos = x;
}
void prepare(int x) { //pa[x] 为结点 x 在点分树中的父结点，没有的话为-1。
    for (auto [y, w] : G[x]) if (!vis[y]) {
        //注意在 dfs 中枚举子结点 y 的时候要判两个条件：y != fa && !vis[y]
        dfs(y, x, w);
    }
}
void solve(int x, int cnt, int pre = -1) { //cnt 为子树 x 中的结点总数
    pos = maxn - 1; f[pos] = sum = cnt;
    getroot(x, -1);
    int root = pos; //因为 pos 是全局变量，递归的时候值会改变，所以此处存为局部变量。
    vis[root] = true;
    pa[root] = pre;
    for (auto [y, w] : G[root]) if (!vis[y]) {
        int total = siz[y] < siz[root] ? siz[y] : cnt - siz[root];
        solve(y, total, root);
```

```
    }
    vis[root] = false;
    prepare(root);
}
int main() {
    int n;
    scanf("%d", &n);
    init(n);
    for (int i = 1; i < n; ++i) {
        int x, y, w;
        scanf("%d %d %d", &x, &y, &w);
        G[x].emplace_back(y, w);
        G[y].emplace_back(x, w);
    }
    DFS1(1, 0, 1);
    DFS2(1, 1);
    solve(1, n);
    return 0;
}
```

2.6 快速求树中与结点 x 距离不超过 k 的点权和

```
const int maxn = 110000;
const int inf = 1 << 30;
int head[maxn * 2], nxt[maxn * 2], to[maxn * 2];
int vis[maxn], pa[maxn], pos, sz, cur; //pa[x] 表示在点分树上结点 x 的父亲
int value[maxn], root[maxn], root2[maxn];
int dep[maxn], Size[maxn], father[maxn], son[maxn], top[maxn];
void DFS1(int u, int fa, int d) {
    dep[u] = d;
    Size[u] = 1;
    son[u] = 0;
    father[u] = fa;
    for (int i = head[u]; i; i = nxt[i]) {
        int v = to[i];
        if (v != fa) {
            DFS1(v, u, d + 1);
            Size[u] += Size[v];
            if (Size[son[u]] < Size[v])
                son[u] = v;
        }
    }
}
void DFS2(int u, int tp) {
    top[u] = tp;
    if (son[u])
        DFS2(son[u], tp);
```

```
    for (int i = head[u]; i; i = nxt[i]) {
        int v = to[i];
        if (v != father[u] && v != son[u])
            DFS2(v, v);
    }
}
int lca(int x, int y) {
    while (top[x] != top[y]) {
        if (dep[top[x]] < dep[top[y]])
            swap(x, y);
        x = father[top[x]];
    }
    return dep[x] < dep[y] ? x : y;
}
int dist(int x, int y) {
    return dep[x] + dep[y] - 2 * dep[lca(x, y)];
}
struct TreeSet {
    struct node {
        int l, r, v;
    } T[maxn * 150];
    int left, right, sz; //空的线段树标号为 0
    void init(int L, int R) { //所有线段树对应区间都为 [L, R]
        left = L;
        right = R;
        sz = 1;
    }
    int newnode() {
        T[sz].l = T[sz].r = T[sz].v = 0;
        return sz++;
    }
    void insert(int x, int pos, int value) { //编号为 x 的线段树的 pos 位置上的值加上 value
        int L = left, R = right;
        while (L < R) {
            int M = (L + R) >> 1;
            T[x].v += value;
            if (pos <= M) {
                if (!T[x].l)
                    T[x].l = newnode();
                x = T[x].l;
                R = M;
            }
            else {
                if (!T[x].r)
                    T[x].r = newnode();
                x = T[x].r;
```

```
                L = M + 1;
            }
        }
        T[x].v += value;
    }
    int sum(int x, int pos) { //在编号为 x 的线段树上计算 pos 位置的前缀和
        int L = left, R = right, res = 0;
        while (x && L < R) {
            int M = (L + R) >> 1, t = T[T[x].l].v;
            if (pos <= M)
                x = T[x].l, R = M;
            else
                x = T[x].r, L = M + 1, res += t;
        }
        res += T[x].v;
        return res;
    }
}tree;
void init() {
    cur = 0;
    memset(head, 0, sizeof(head));
    memset(vis, 0, sizeof(vis));
}
void insert(int u, int v) {
    nxt[++cur] = head[u];
    head[u] = cur;
    to[cur] = v;
}
int dfs0(int u, int fa) { //求出以 u 为根的子树的大小
    int tot = 1;
    for (int i = head[u]; i; i = nxt[i]) {
        int v = to[i];
        if (v != fa && !vis[v])
            tot += dfs0(v, u);
    }
    return tot;
}
int dfs1(int u, int fa, int cnt) { //求出以 u 为根的子树的重心
    int tot = 1, num = 0;
    for (int i = head[u]; i; i = nxt[i]) {
        int v = to[i];
        if (v != fa && !vis[v]) {
            int result = dfs1(v, u, cnt);
            tot += result;
            num = max(num, result);
        }
```

```
    }
    num = max(num, cnt - tot);
    if (num < sz) {
        sz = num;
        pos = u;
    }
    return tot;
}
void dfs2(int u, int fa, int host) {
    tree.insert(root[host], dist(u, host), value[u]);
    for (int i = head[u]; i; i = nxt[i]) {
        int v = to[i];
        if (v != fa && !vis[v]) {
            dfs2(v, u, host);
        }
    }
}
void dfs3(int u, int fa, int host) {
    tree.insert(root2[host], dist(u, pa[host]), value[u]);
    for (int i = head[u]; i; i = nxt[i]) {
        int v = to[i];
        if (v != fa && !vis[v]) {
            dfs3(v, u, host);
        }
    }
}
void build(int u, int cnt, int pre = 0) { //递归处理子树 u, cnt 为子树 u 中的结点总数
    sz = inf;
    dfs1(u, -1, cnt); //求出以 u 为根的子树的重心 root, 作为新的根结点
    int root = pos;
    vis[root] = 1; //标记 root 结点不能再被访问
    pa[root] = pre;
    dfs2(root, -1, root);
    if (pre) dfs3(root, -1, root);
    for (int i = head[root]; i; i = nxt[i]) {
        int v = to[i];
        if (!vis[v]) {
            build(v, dfs0(v, -1), root);
        }
    }
}
long long query(int x, int k) {
    long long res = tree.sum(root[x], k);
    for (int i = x; pa[i]; i = pa[i]) {
        int d = dist(x, pa[i]);
        if (k - d >= 0) {
```

```
            res += tree.sum(root[pa[i]], k - d);
            res -= tree.sum(root2[i], k - d);
        }
    }
    return res;
}
void update(int x, int delta) {
    tree.insert(root[x], 0, delta);
    for (int i = x; pa[i]; i = pa[i]) {
        int d = dist(pa[i], x);
        tree.insert(root[pa[i]], d, delta);
        tree.insert(root2[i], d, delta);
    }
}
int main() {
//    freopen("in.txt", "r", stdin);
    int n, m;
    scanf("%d %d", &n, &m);
    init();
    tree.init(0, 110000);
    for (int i = 1; i <= n; ++i)
        root[i] = tree.newnode();
    for (int i = 1; i <= n; ++i)
        root2[i] = tree.newnode();
    for (int i = 1; i <= n; ++i)
        scanf("%d", &value[i]);
    for (int i = 1; i < n; ++i) {
        int u, v;
        scanf("%d %d", &u, &v);
        insert(u, v);
        insert(v, u);
    }
    DFS1(1, 0, 1);
    DFS2(1, 1);
    build(1, n);
    long long ans = 0;
    while (m--) {
        long long tp, x, y;
        scanf("%lld %lld %lld", &tp, &x, &y);
        x ^= ans; y ^= ans;
        if (tp == 0) {
            ans = query(x, y);
            printf("%lld\n", ans);
        }
        else {
            int delta = y - value[x];
```

```
            value[x] = y;
            update(x, delta);
        }
    }
    return 0;
}
```

2.7 动态维护树中白色结点的最长距离

```
/*
P2056 [ZJOI2007] 捉迷藏
动态维护树中白色结点的最长距离。
操作 C 翻转一个结点的颜色，操作 G 进行一次查询（树中白色结点的直径）。
*/
const int maxn = 101000;
template<typename T> class heap {
private:
    priority_queue<T> Q, R;
public:
    void maintain() {
        while (!R.empty() && Q.top() == R.top())
            Q.pop(), R.pop();
    }
    void push(const T& val) {
        Q.push(val);
    }
    void erase(const T& val) { //只能删除还在优先队列中的值，不能删除不存在的值。
        R.push(val);
        maintain();
    }
    void pop() {
        Q.pop();
        maintain();
    }
    T top() {
        return Q.top();
    }
    T top2() { //返回第二大的值，调用之前必须保证有 size() >= 2。
        auto val = Q.top(); pop();
        auto sec = Q.top(); push(val);
        return sec;
    }
    auto size() const {
        return Q.size() - R.size();
    }
    bool empty() const {
        return size() == 0;
    }
```

```
    }
    template<typename ...F> void emplace(F&&... args) {
        Q.emplace(std::forward<F>(args)...);
    }
};
vector<int> G[maxn]; //to, weight
int vis[maxn], siz[maxn], f[maxn], pa[maxn], pos, sum; //pa[x] 表示在点分树上结点 x 的父亲
int state[maxn], res[maxn], rt, n;
int son[maxn], dep[maxn], father[maxn], top[maxn];
heap<int> head[maxn], Q[maxn], answer;
void DFS1(int x, int fa, int d) {
    dep[x] = d;
    siz[x] = 1;
    son[x] = 0;
    father[x] = fa;
    for (auto y : G[x]) if (y != fa) {
        DFS1(y, x, d + 1);
        siz[x] += siz[y];
        if (siz[son[x]] < siz[y])
            son[x] = y;
    }
}
void DFS2(int x, int tp) {
    top[x] = tp;
    if (son[x])
        DFS2(son[x], tp);
    for (auto y : G[x])
        if (y != father[x] && y != son[x])
            DFS2(y, y);
}
int lca(int x, int y) {
    while (top[x] != top[y]) {
        if (dep[top[x]] < dep[top[y]])
            swap(x, y);
        x = father[top[x]];
    }
    return dep[x] < dep[y] ? x : y;
}
int dist(int x, int y) {
    return dep[x] + dep[y] - 2 * dep[lca(x, y)];
}
void getroot(int x, int fa) {
    f[x] = 0; siz[x] = 1;
    for (auto y : G[x]) if (y != fa && !vis[y]) {
        getroot(y, x);
        f[x] = max(f[x], siz[y]);
```

```
        siz[x] += siz[y];
    }
    f[x] = max(f[x], sum - siz[x]);
    if (f[x] < f[pos])
        pos = x;
}
void dfs(int x, int fa) {
    Q[rt].push(dist(x, pa[rt]));
    for (auto y : G[x]) if (y != fa && !vis[y]) {
        dfs(y, x);
    }
}
void prepare(int x) {
    head[x].push(0);
    res[x] = (head[x].size() == 1 ? 0 : head[x].top() + head[x].top2());
    answer.emplace(res[x]);
    if (pa[x] < 0) return;
    Q[x].push(dist(x, pa[x]));
    for (auto y : G[x]) if (!vis[y]) {
        ::rt = x;
        dfs(y, x);
    }
    head[pa[x]].push(Q[x].top());
}
void update(int x) {
    state[x] ^= 1;
    for (int y = -1, p = x; p >= 1; y = p, p = pa[y]) {
        if (y == -1) {
            if (state[x] == 0)
                head[x].push(0);
            else
                head[x].erase(0);
        }
        else {
            auto& q = Q[y];
            if (q.size())
                head[p].erase(q.top());
            if (state[x] == 0)
                q.emplace(dist(x, p));
            else
                q.erase(dist(x, p));
            if (q.size())
                head[p].push(q.top());
        }
        int last = res[p];
        if (head[p].size() >= 2)
```

```
            res[p] = head[p].top() + head[p].top2();
        else if (head[p].size() >= 1)
            res[p] = (state[p] == 0 ? head[p].top() : 0);
        else
            res[p] = -1;
        if (last != res[p]) {
            answer.erase(last);
            answer.emplace(res[p]);
        }
    }
}
void solve(int x, int cnt, int pre = -1) { //cnt 为子树 x 中的结点总数
    pos = maxn - 1; f[pos] = sum = cnt;
    getroot(x, -1);
    int root = pos; //因为 pos 是全局变量，递归的时候值会改变，所以此处存为局部变量。
    vis[root] = true;
    pa[root] = pre;
    for (auto y : G[root]) if (!vis[y]) {
        int total = siz[y] < siz[root] ? siz[y] : cnt - siz[root];
        solve(y, total, root);
    }
    vis[root] = false;
    prepare(root);
}
int main() {
    //freopen("in.txt", "r", stdin);
    scanf("%d", &n);
    for (int i = 1; i < n; ++i) {
        int x, y;
        scanf("%d %d", &x, &y);
        G[x].emplace_back(y);
        G[y].emplace_back(x);
    }
    DFS1(1, 0, 1);
    DFS2(1, 1);
    solve(1, n);
    int Q, x; scanf("%d", &Q);
    while (Q--) {
        char tp; scanf(" %c ", &tp);
        if (tp == 'G') {
            printf("%d\n", answer.top());
        }
        else {
            scanf("%d", &x);
            update(x);
        }
```

```
    }
    return 0;
}
```

2.8 虚树

```
/*
ins 函数会在每个结点 i 第一次入栈的时候，将 E[i] 清空。这样在最坏情况下空间复杂度可达 O(nlogn),
可以改成在每次询问后用一个 DFS 清空 E，这样也 E 就不会占用额外内存了。
*/
const int maxn = 1110000;
int dep[maxn], sz[maxn], pa[maxn], son[maxn], top[maxn], stk[maxn], dfn[maxn], clk, tp;
vector<int> G[maxn], E[maxn];
void dfs1(int u, int fa, int d) {
    dfn[u] = ++clk;
    dep[u] = d;
    sz[u] = 1;
    son[u] = 0;
    pa[u] = fa;
    for (auto v : G[u]) if (v != fa) {
        dfs1(v, u, d + 1);
        sz[u] += sz[v];
        if (sz[son[u]] < sz[v])
            son[u] = v;
    }
}
void dfs2(int u, int tp) {
    top[u] = tp;
    if (son[u])
        dfs2(son[u], tp);
    for (auto v : G[u]) if (v != pa[u] && v != son[u])
        dfs2(v, v);
}
int lca(int x, int y) {
    while (top[x] != top[y]) {
        if (dep[top[x]] < dep[top[y]])
            swap(x, y);
        x = pa[top[x]];
    }
    return dep[x] < dep[y] ? x : y;
}
int dist(int x, int y) {
    return dep[x] + dep[y] - 2 * dep[lca(x, y)];
}
void init(int root) {
    clk = 0; stk[0] = -1;
    dfs1(root, 0, 1);
```

```
    dfs2(root, root);
}
void ins(int x) {
    if (tp > 0) {
        int fa = lca(stk[tp], x);
        if (fa != stk[tp]) {
            while (tp > 1 && dep[fa] < dep[stk[tp - 1]]) {
                E[stk[tp - 1]].push_back(stk[tp]);
                --tp;
            }
            int last = stk[tp--];
            if (fa != stk[tp]) {
                E[fa].clear();
                stk[++tp] = fa;
            }
            E[fa].push_back(last);
        }
    }
    E[x].clear();
    stk[++tp] = x;
}
int build(vector<int> nodes) { //对集合 nodes 中的结点建立虚树，并返回根结点。
    sort(nodes.begin(), nodes.end(), [](int x, int y) {
        return dfn[x] < dfn[y];
    });
    for (auto i : nodes)
        ins(i);
    while (--tp)
        E[stk[tp]].push_back(stk[tp + 1]);
    return stk[1];
}
int main() {
    return 0;
}
```

2.9 O(1)-LCA

```
const int maxn = 1110000; //maxn 至少为最大结点数的两倍
vector<int> G[maxn], seq;
int pos[maxn], dep[maxn], lg[maxn], a[maxn][20];
void dfs(int x, int fa, int d) {
    dep[x] = d;
    pos[x] = seq.size();
    seq.push_back(x);
    for (auto y : G[x]) if (y != fa) {
        dfs(y, x, d + 1);
        seq.push_back(x);
```

```
    }
}
void init(int s) { //根结点为 s，调用之前 G 中应当已经保存了整棵树。
    seq.resize(1);
    dfs(s, -1, 1);
    const int n = seq.size() - 1;
    lg[1] = 0;
    for (int i = 2; i <= n; ++i)
        lg[i] = lg[i >> 1] + 1;
    for (int i = 1; i <= n; ++i)
        a[i][0] = seq[i];
    for (int j = 1; j <= lg[n]; ++j) {
        for (int i = 1; i + (1 << j) - 1 <= n; ++i) {
            int x = a[i][j - 1], y = a[i + (1 << (j - 1))][j - 1];
            a[i][j] = (dep[x] < dep[y] ? x : y);
        }
    }
}
inline int lca(int x, int y) {
    int L = pos[x], R = pos[y];
    if (L > R)
        swap(L, R);
    int k = lg[R - L + 1];
    x = a[L][k];
    y = a[R - (1 << k) + 1][k];
    return dep[x] < dep[y] ? x : y;
}
inline int dist(int x, int y) {
    return dep[x] + dep[y] - 2 * dep[lca(x, y)];
}
int main() {
    //freopen("in.txt", "r", stdin);
    int n, m, s;
    scanf("%d %d %d", &n, &m, &s);
    for (int i = 1; i < n; ++i) {
        int x, y;
        scanf("%d %d", &x, &y);
        G[x].push_back(y);
        G[y].push_back(x);
    }
    init(s);
    for (int i = 0; i < m; ++i) {
        int x, y;
        scanf("%d %d", &x, &y);
        printf("%d\n", lca(x, y));
    }
```

```
    return 0;
}
```

3 基础算法

3.1 Java 快速读入

```java
import java.io.*;
class InputReader {
    public BufferedReader reader;
    public StringTokenizer tokenizer;
    public InputReader(InputStream stream) {
        reader = new BufferedReader(new InputStreamReader(stream), 32768);
        tokenizer = null;
    }
    public String next() {
        while (tokenizer == null || !tokenizer.hasMoreTokens()) {
            try {
                tokenizer = new StringTokenizer(reader.readLine());
            } catch (IOException e) {
                throw new RuntimeException(e);
            }
        }
        return tokenizer.nextToken();
    }
    public int nextInt() {
        return Integer.parseInt(next());
    }
}
```

3.2 C++ 快速读入

```cpp
inline int read_positive() { //读取一个正整数
    int x = 0;
    char c = getchar();
    while (c < '0' || c > '9')
        c = getchar();
    while (c >= '0' && c <= '9') {
        x = x * 10 + c - '0';
        c = getchar();
    }
    return x;
}
inline int read_negative() { //可以读取负数
    int x = 0, y = 1;
    char c = getchar();
    while (c < '0' || c > '9') {
```

```
        if (c == '-')
            y = -1;
        c = getchar();
    }
    while (c >= '0' && c <= '9') {
        x = x * 10 + c - '0';
        c = getchar();
    }
    return x * y;
}
int main() {
}
```

3.3 RMQ 算法

```
const int maxn = 110000, maxlog = 20;
int d[maxn][maxlog], lg[maxn], A[maxn];
void preprocess(int n) { //下标范围 [1, n], 调用之前应当完成数组 A 的赋值
    lg[1] = 0;
    for (int i = 2; i <= n; ++i)
        lg[i] = lg[i >> 1] + 1;
    for (int i = 1; i <= n; ++i)
        d[i][0] = A[i];
    for (int j = 1; j <= lg[n]; ++j)
        for (int i = 1; i + (1 << j) - 1 <= n; ++i)
            d[i][j] = max(d[i][j - 1], d[i + (1 << (j - 1))][j - 1]);
}
inline int rmq(int L, int R) {
    int k = lg[R - L + 1];
    return max(d[L][k], d[R - (1 << k) + 1][k]);
}
int main() {
    return 0;
}
```

3.4 哈希表

```
//key_t 应当为整数类型，且实际值必须非负
template<typename key_t, typename type> struct hash_table {
    static const int maxn = 1000010;
    static const int table_size = 11110007;
    int first[table_size], nxt[maxn], sz; //init: memset(first, 0, sizeof(first)), sz = 0
    key_t id[maxn];
    type data[maxn];
    type& operator[] (key_t key) {
        const int h = key % table_size;
        for (int i = first[h]; i; i = nxt[i])
```

```
            if (id[i] == key)
                return data[i];
        int pos = ++sz;
        nxt[pos] = first[h];
        first[h] = pos;
        id[pos] = key;
        return data[pos] = type();
    }
    bool count(key_t key) {
        for (int i = first[key % table_size]; i; i = nxt[i])
            if (id[i] == key)
                return true;
        return false;
    }
    type get(key_t key) { //如果 key 对应的值不存在，则返回 type()。
        for (int i = first[key % table_size]; i; i = nxt[i])
            if (id[i] == key)
                return data[i];
        return type();
    }
};
unordered_map<long long, long long> A;
hash_table<long long, long long> B;
const int maxn = 1000000;
int main() {
    default_random_engine e;
    uniform_int_distribution<long long> d(0, LLONG_MAX);
    for (int i = 0; i < maxn; ++i) {
        long long key = d(e);
        long long value = d(e);
        A[key] = value;
        B[key] = value;
    }
    for (int i = 0; i < maxn * 10; ++i) {
        long long key = d(e);
        if (A.count(key) != B.count(key))
            abort();
        if (A.count(key)) {
            if (A[key] != B.get(key))
                abort();
        }
    }
    return 0;
}
```

3.5 基数排序

```
const int maxn = 1100000; //maxn 应当大于 data 数组的大小
namespace RadixSort_Int {
    const int base = (1 << 17) - 1;
    int c[base + 10], tmp[maxn];
    void sort(int* data, int n) {            //只能用于对非负数离散化，若要处理负数，
        for (int i = 0; i < 32; i += 16) {    //可以将所有的值都加上一个特别大的数使其变为非负数然
↪   后再离散化
            memset(c, 0, sizeof(c));
            for (int j = 0; j < n; ++j)
                c[(data[j] >> i) & base]++;
            for (int j = 1; j <= base; ++j)
                c[j] += c[j - 1];
            for (int j = n - 1; j >= 0; --j)
                tmp[--c[(data[j] >> i) & base]] = data[j];
            for (int j = 0; j < n; ++j)
                data[j] = tmp[j];
        }
    }
}
namespace RadixSort_LLong {
    const int base = (1 << 17) - 1;
    int c[base + 10];
    long long tmp[maxn];
    void sort(long long* data, int n) {       //只能用于对非负数离散化，若要处理负数，
        for (int i = 0; i < 64; i += 16) {    //可以将所有的值都加上一个特别大的数使其变为非负数然
↪   后再离散化
            memset(c, 0, sizeof(c));
            for (int j = 0; j < n; ++j)
                c[(data[j] >> i) & base]++;
            for (int j = 1; j <= base; ++j)
                c[j] += c[j - 1];
            for (int j = n - 1; j >= 0; --j)
                tmp[--c[(data[j] >> i) & base]] = data[j];
            for (int j = 0; j < n; ++j)
                data[j] = tmp[j];
        }
    }
}
int A[maxn], B[maxn];
int main() {
    int n = 1000000;
    default_random_engine e;
    uniform_int_distribution<int> d(0, INT_MAX);
    for (int i = 0; i < n; ++i)
        A[i] = B[i] = d(e);
```

```
    printf("start....\n");
    auto start = clock();
    sort(A, A + n);
    printf("std::sort: %f\n", static_cast<double>(clock() - start) / CLOCKS_PER_SEC);
    start = clock();
    RadixSort_Int::sort(B, n);
    printf("std::sort: %f\n", static_cast<double>(clock() - start) / CLOCKS_PER_SEC);
    for (int i = 0; i < n; ++i)
        if (A[i] != B[i])
            abort();
    return 0;
}
```

3.6 快速离散化

```
const int maxn = 11000000; //maxn 应当大于 data 数组的大小
namespace Discretization_Int {
    const int base = (1 << 17) - 1;
    int c[base + 10];
    pair<int, int> data[maxn], tmp[maxn];
    void discretization(int* input, int n) {    //只能用于对非负数离散化，若要处理负数，
        for (int i = 0; i < n; ++i)              //可以将所有的值都加上一个特别大的数使其变为非负
↪   数然后再离散化
            data[i] = make_pair(input[i], i);
        for (int i = 0; i < 32; i += 16) {
            memset(c, 0, sizeof(c));
            for (int j = 0; j < n; ++j)
                c[(data[j].first >> i) & base]++;
            for (int j = 1; j <= base; ++j)
                c[j] += c[j - 1];
            for (int j = n - 1; j >= 0; --j)
                tmp[--c[(data[j].first >> i) & base]] = data[j];
            for (int j = 0; j < n; ++j)
                data[j] = tmp[j];
        }
        for (int i = 0, j = -1; i < n; ++i) {
            if (i == 0 || data[i].first != data[i - 1].first)
                ++j;
            input[data[i].second] = j;
        }
    }
}
namespace Discretization_LLong {
    const int base = (1 << 17) - 1;
    int c[base + 10];
    pair<long long, int> data[maxn], tmp[maxn];
    void discretization(long long* input, int n) {    //只能用于对非负数离散化，若要处理负数，
```

```
        for (int i = 0; i < n; ++i)                    //可以将所有的值都加上一个特别大的数使其变为
↪   非负数然后再离散化
            data[i] = make_pair(input[i], i);
        for (int i = 0; i < 64; i += 16) {
            memset(c, 0, sizeof(c));
            for (int j = 0; j < n; ++j)
                c[(data[j].first >> i) & base]++;
            for (int j = 1; j <= base; ++j)
                c[j] += c[j - 1];
            for (int j = n - 1; j >= 0; --j)
                tmp[--c[(data[j].first >> i) & base]] = data[j];
            for (int j = 0; j < n; ++j)
                data[j] = tmp[j];
        }
        for (int i = 0, j = -1; i < n; ++i) {
            if (i == 0 || data[i].first != data[i - 1].first)
                ++j;
            input[data[i].second] = j;
        }
    }
}
long long A[maxn], B[maxn], tmp[maxn];
int main() {
    int n = 10000000;
    default_random_engine e;
    uniform_int_distribution<long long> d(0, LLONG_MAX);
    e.seed(time(0));
    for (int i = 0; i < n; ++i)
        A[i] = B[i] = tmp[i] = d(e);
    printf("start....\n");
    auto start = clock();
    sort(tmp, tmp + n);
    int sz = unique(tmp, tmp + n) - tmp;
    for (int i = 0; i < n; ++i)
        A[i] = lower_bound(tmp, tmp + sz, A[i]) - tmp;
    printf("std::sort: %f\n", static_cast<double>(clock() - start) / CLOCKS_PER_SEC);
    start = clock();
    Discretization_LLong::discretization(B, n);
    printf("std::sort: %f\n", static_cast<double>(clock() - start) / CLOCKS_PER_SEC);
    for (int i = 0; i < n; ++i)
        if (A[i] != B[i])
            abort();
    return 0;
}
```

3.7 带删除的优先队列

```
template<typename T> class heap {
private:
    priority_queue<T> Q, R;
public:
    void maintain() {
        while (!R.empty() && Q.top() == R.top())
            Q.pop(), R.pop();
    }
    void push(const T& val) {
        Q.push(val);
    }
    void erase(const T& val) { //只能删除还在优先队列中的值，不能删除不存在的值。
        R.push(val);
        maintain();
    }
    void pop() {
        Q.pop();
        maintain();
    }
    T top() {
        return Q.top();
    }
    T top2() { //返回第二大的值，调用之前必须保证有 size() >= 2。
        auto val = Q.top(); pop();
        auto sec = Q.top(); push(val);
        return sec;
    }
    auto size() const {
        return Q.size() - R.size();
    }
    bool empty() const {
        return size() == 0;
    }
    template<typename ...F> void emplace(F&&... args) {
        Q.emplace(std::forward<F>(args)...);
    }
};
int main() {
    return 0;
}
```

3.8 排列组合枚举

```
void enumerate(int n, int k) {
    //枚举 [0, n) 的所有的子集的子集
```

```
    for (int s = 0; s < (1 << n); ++s) {
        for (int s0 = s; s0; s0 = (s0 - 1) & s) { //枚举集合 s 的所有子集
        }
    }
    //枚举 [0, n) 中所有大小为 k 的子集
    for (int x, y, s = (1 << k) - 1; s < (1 << n); x = s & -s, y = s + x, s = (((s & ~y) / x) >>
↪   1) | y) {
        //cout << bitset<5>(s) << endl;
    }
    //枚举 [0, n) 中所有大小为 k 的排列
    for (int x, y, s = (1 << k) - 1; s < (1 << n); x = s & -s, y = s + x, s = (((s & ~y) / x) >>
↪   1) | y) {
        vector<int> P;
        for (int i = 0; i < n; ++i) if (s & (1 << i))
            P.push_back(i);
        do {
            //for (int i = 0; i < k; ++i) printf("%d ", P[i]); printf("\n");
        } while (next_permutation(P.begin(), P.end()));
    }
}
int main() {
    enumerate(4, 3);
    return 0;
}
```

3.9 动态维护连续相同数字区间

```
/*
动态维护由相同数字构成的区间，A 是原始数组。将 A 中相同数字只保留最左侧，其余设为-1 得到 val 数组。
该算法动态维护 val 数组，时间复杂度 O(1)。
*/
const int maxn = 210000;
int A[maxn], val[maxn], n;
void build() {
}
inline void add(int index) { //将 A[index] 添加到 val[index] 中构成一个新的区间
    val[index] = A[index];
    //此处添加代码维护数据结构
}
inline void del(int index) { //删除 val[index]，也就是删除下标为 index 的区间
    val[index] = -1;
    //此处添加代码维护数据结构
}
void init() { //调用 init 函数初始化之前需要读入 n 和 A
    A[0] = A[n + 1] = -1;
    val[1] = A[1];
    for (int i = 2; i <= n; ++i)
```

```
        val[i] = (A[i] == A[i - 1] ? -1 : A[i]);
    build(); //用 val 数组构建数据结构
}
void change(int index, int v) { //将 A 中下标为 index 的位置的元素修改为 v，函数会维护 A 和 val 数
↪ 组
    if (A[index] == v)
        return;
    if (A[index] == A[index + 1])
        add(index + 1);
    if (val[index] != -1)
        del(index);
    if (val[index + 1] == v)
        del(index + 1);
    A[index] = v;
    if (A[index - 1] != v)
        add(index);
}
int main() {
    freopen("in.txt", "r", stdin);
    n = 5;
    for (int i = 1; i <= n; ++i)
        A[i] = 1;
    init();
    for (int i = 1; i <= n; ++i) printf("%d ", val[i]); printf("\n");
    change(3, 2); for (int i = 1; i <= n; ++i) printf("%d ", val[i]); printf("\n");
    change(4, 2); for (int i = 1; i <= n; ++i) printf("%d ", val[i]); printf("\n");
    change(2, 2); for (int i = 1; i <= n; ++i) printf("%d ", val[i]); printf("\n");
    change(2, 1); for (int i = 1; i <= n; ++i) printf("%d ", val[i]); printf("\n");
    change(1, 2); for (int i = 1; i <= n; ++i) printf("%d ", val[i]); printf("\n");
    change(2, 2); for (int i = 1; i <= n; ++i) printf("%d ", val[i]); printf("\n");
    change(3, 1); for (int i = 1; i <= n; ++i) printf("%d ", val[i]); printf("\n");
    change(3, 2); for (int i = 1; i <= n; ++i) printf("%d ", val[i]); printf("\n");
    return 0;
}
```

4 匹配算法

4.1 匈牙利算法

```
// UVa11419 SAM I AM
// Rujia Liu
const int maxn = 1000 + 5; // 单侧顶点的最大数目
// 二分图最大基数匹配
struct BPM {
    int n, m; // 左右顶点个数
    vector<int> G[maxn]; // 邻接表
    int left[maxn]; // left[i] 为右边第 i 个点的匹配点编号，-1 表示不存在
```

```
bool T[maxn]; // T[i] 为右边第 i 个点是否已标记
int right[maxn]; // 求最小覆盖用
bool S[maxn]; // 求最小覆盖用
void init(int n, int m) {
    this->n = n;
    this->m = m;
    for(int i = 0; i < n; i++) G[i].clear();
}
void AddEdge(int u, int v) {
    G[u].push_back(v);
}
bool match(int u){
    S[u] = true;
    for(int i = 0; i < G[u].size(); i++) {
        int v = G[u][i];
        if (!T[v]){
            T[v] = true;
            if (left[v] == -1 || match(left[v])){
                left[v] = u;
                right[u] = v;
                return true;
            }
        }
    }
    return false;
}
// 求最大匹配
int solve() {
    memset(left, -1, sizeof(left));
    memset(right, -1, sizeof(right));
    int ans = 0;
    for(int u = 0; u < n; u++) { // 从左边结点 u 开始增广
        memset(S, 0, sizeof(S));
        memset(T, 0, sizeof(T));
        if(match(u)) ans++;
    }
    return ans;
}
// 求最小覆盖。X 和 Y 为最小覆盖中的点集（最大独立集与最小覆盖集互补）
int mincover(vector<int>& X, vector<int>& Y) {
    int ans = solve();
    memset(S, 0, sizeof(S));
    memset(T, 0, sizeof(T));
    for(int u = 0; u < n; u++)
        if(right[u] == -1) match(u); // 从所有 X 未盖点出发增广
    for(int u = 0; u < n; u++)
```

```
            if(!S[u]) X.push_back(u); // X 中的未标记点
        for(int v = 0; v < m; v++)
            if(T[v]) Y.push_back(v); // Y 中的已标记点
     return ans;
    }
};
BPM solver;
int R, C, N;
int main(){
    int kase = 0;
    while(scanf("%d%d%d", &R, &C, &N) == 3 && R && C && N) {
        solver.init(R, C);
        for(int i = 0; i < N; i++) {
            int r, c;
            scanf("%d%d", &r, &c); r--; c--;
            solver.AddEdge(r, c);
        }
        vector<int> X, Y;
        int ans = solver.mincover(X, Y);
        printf("%d", ans);
        for(int i = 0; i < X.size(); i++) printf(" r%d", X[i]+1);
        for(int i = 0; i < Y.size(); i++) printf(" c%d", Y[i]+1);
        printf("\n");
    }
    return 0;
}
```

4.2 KM 算法

```
const long long inf = 1LL << 60;
const int maxn = 505;
//若要求不完美匹配，需要把所有的-inf 替换成 0。
struct KM {
    int n, py[maxn], vy[maxn], pre[maxn];
    long long G[maxn][maxn], slk[maxn], kx[maxn], ky[maxn];
    void init(int n) { //左右两侧各有 n 个结点，结点编号从 1 开始
        this->n = n;
        for (int i = 1; i <= n; ++i)
            for (int j = 1; j <= n; ++j)
                G[i][j] = -inf;
        for (int i = 1; i <= n; ++i)
            ky[i] = py[i] = 0;
    }
    void add_edge(int x, int y, long long w) {
        G[y][x] = max(G[y][x], w);
    }
    long long solve() {
```

```
        int k, p;
        for (int i = 1; i <= n; ++i)
            kx[i] = *max_element(G[i] + 1, G[i] + n + 1);
        for (int i = 1; i <= n; i++) {
            for (int j = 0; j <= n; ++j)
                vy[j] = pre[j] = 0, slk[j] = inf;
            for (py[k = 0] = i; py[k]; k = p) {
                long long d = inf;
                int x = py[k];
                vy[k] = 1;
                for (int j = 1; j <= n; j++) if (!vy[j]) {
                    long long t = kx[x] + ky[j] - G[x][j];
                    if (t < slk[j])
                        slk[j] = t, pre[j] = k;
                    if (slk[j] < d)
                        d = slk[j], p = j;
                }
                for (int j = 0; j <= n; j++) {
                    if (vy[j])
                        kx[py[j]] -= d, ky[j] += d;
                    else
                        slk[j] -= d;
                }
            }
            for (; k; k = pre[k])
                py[k] = py[pre[k]];
        }
        long long ans = 0;
        for (int i = 1; i <= n; i++) if (G[py[i]][i] > -inf)
            ans += kx[i] + ky[i];
        return ans;
    }
    vector<int> result() { //返回左侧每个结点对应的右侧匹配点，没有则为 0
        vector<int> res(1);//调用该函数之前，应当先调用 solve 函数
        for (int i = 1; i <= n; ++i)
            res.push_back(G[py[i]][i] > -inf ? py[i] : 0);
        return res;
    }
}km;
struct KM_old {
    static const int inf = 1 << 30;
    int n;
    vector<int> G[maxn];
    int W[maxn][maxn];
    int lx[maxn], ly[maxn];
    int left[maxn];
```

```
bool S[maxn], T[maxn];
void init(int n) {
    this->n = n;
    for (int i = 0; i < n; ++i)
        G[i].clear();
    memset(W, 0, sizeof(W));
}
void add_edge(int u, int v, int w) {
    G[u].push_back(v);
    W[u][v] = w;
}
bool match(int u) {
    S[u] = true;
    for (unsigned i = 0; i < G[u].size(); ++i) {
        int v = G[u][i];
        if (lx[u] + ly[v] == W[u][v] && !T[v]) {
            T[v] = true;
            if (left[v] == -1 || match(left[v])) {
                left[v] = u;
                return true;
            }
        }
    }
    return false;
}
void update() {
    int a = inf;
    for (int u = 0; u < n; ++u) if (S[u]) {
        for (unsigned i = 0; i < G[u].size(); ++i) {
            int v = G[u][i];
            if (!T[v]) a = min(a, lx[u] + ly[v] - W[u][v]);
        }
    }
    for (int i = 0; i < n; ++i) {
        if (S[i]) lx[i] -= a;
        if (T[i]) ly[i] += a;
    }
}
int solve() {
    for (int i = 0; i < n; ++i) {
        lx[i] = *max_element(W[i], W[i] + n);
        left[i] = -1;
        ly[i] = 0;
    }
    for (int u = 0; u < n; ++u) {
        for (;;) {
```

```
                for (int i = 0; i < n; ++i) S[i] = T[i] = false;
                if (match(u)) break; else update();
            }
        }
        int ans = 0;
        for (int i = 0; i < n; ++i) if (left[i] != -1)
            ans += W[left[i]][i];
        return ans;
    }
};
int main() {
    int n, m;
    scanf("%d %d", &n, &m);
    km.init(n);
    for (int i = 1; i <= m; ++i) {
        int x, y, w;
        scanf("%d %d %d", &x, &y, &w);
        km.add_edge(y, x, w);
    }
    printf("%lld\n", km.solve());
    auto res = km.result();
    for (int i = 1; i <= n; ++i)
        printf("%d ", res[i]);
    printf("\n");
    return 0;
}
```

4.3 带花树算法

```
const int maxn = 1005;
struct Blossom {
    int mate[maxn], nxt[maxn], pa[maxn], st[maxn], vis[maxn], t, n;
    int ban[maxn];
    vector<int> G[maxn];
    queue<int> Q;
    void init(int n) { //编号从 0 开始
        this->n = n;
        this->t = 0;
        memset(mate, -1, sizeof(mate));
        memset(ban, 0, sizeof(ban));
        for (int i = 0; i < n; ++i)
            G[i].clear();
    }
    inline void add_edge(int x, int y) { //添加双向边 (x, y)
        G[x].push_back(y);
        G[y].push_back(x);
    }
```

```cpp
inline int find(int x) {
    return pa[x] == x ? x : pa[x] = find(pa[x]);
}
inline void merge(int a, int b) {
    pa[find(a)] = find(b);
}
int lca(int x, int y) {
    for (t++;;swap(x, y)) if (~x) {
        if (vis[x = find(x)] == t)
            return x;
        vis[x] = t;
        x = ~mate[x] ? nxt[mate[x]] : -1;
    }
}
void group(int a, int p) {
    for (int b, c; a != p; merge(a, b), merge(b, c), a = c) {
        b = mate[a], c = nxt[b];
        if (find(c) != p)
            nxt[c] = b;
        if (st[b] == 2)
            st[b] = 1, Q.push(b);
        if (st[c] == 2)
            st[c] = 1, Q.push(c);
    }
}
void augment(int s) {
    for (int i = 0; i < n; ++i)
        nxt[i] = vis[i] = -1, pa[i] = i, st[i] = 0;
    Q = queue<int>();
    Q.push(s);
    st[s] = 1;
    while (mate[s] == -1 && !Q.empty()) {
        int x = Q.front();Q.pop();
        for (auto y : G[x]) {
            if (!ban[y] && y != mate[x] && find(x) != find(y) && st[y] != 2) {
                if (st[y] == 1) {
                    int p = lca(x, y);
                    if (find(x) != p)
                        nxt[x] = y;
                    if (find(y) != p)
                        nxt[y] = x;
                    group(x, p);
                    group(y, p);
                }
                else if (mate[y] == -1) {
                    nxt[y] = x;
```

```
                    for (int j = y, k, i; ~j; j = i)
                        k = nxt[j], i = mate[k], mate[j] = k, mate[k] = j;
                    break;
                }
                else
                    nxt[y] = x, Q.push(mate[y]), st[mate[y]] = 1, st[y] = 2;
            }
        }
    }
}
void solve() { //求最大匹配
    for (int i = 0; i < n; ++i) if (mate[i] == -1) //从所有的点开始增广
        augment(i);
    int ans = 0;
    for (int i = 0; i < n; ++i) if (mate[i] != -1)
        ans++;
    printf("%d\n", ans / 2); //匹配数
    for (int i = 0; i < n; ++i) //打印每个点的匹配点，-1 表示未匹配
        printf("%d ", mate[i]);
    printf("\n");
}
vector<int> unnecessary() { //求出所有的非必要匹配点（存在一个最大匹配不包含这个点）
    for (int i = 0; i < n; ++i) if (mate[i] == -1)//先求出最大匹配
        augment(i);
    vector<int> ret;
    for (int x = 0; x < n; ++x) {
        if (mate[x] == -1)
            ret.push_back(x);
        else {
            int y = mate[x];
            mate[y] = -1;
            mate[x] = -1;
            ban[x] = 1;
            augment(y);
            ban[x] = 0;
            if (mate[y] != -1)
                ret.push_back(x);
            else
                augment(y);
        }
    }
    return ret;
}
}mp;
int main() {
    int n, m;
```

```
    scanf("%d %d", &n, &m);
    mp.init(n);
    for (int i = 0; i < m; ++i) {
        int x, y;
        scanf("%d %d", &x, &y); --x; --y;
        mp.add_edge(x, y);
    }
    mp.solve();
    return 0;
}
```

5 高精度

5.1 大整数类

```
typedef long long ll;
const int base = 100000000;
const int num_digit = 8;
const int maxn = 1000;
ll mul_mod (ll x, ll y, ll n){
    ll T = floor(sqrt(n) + 0.5);
    ll t = T * T - n;
    ll a = x / T; ll b = x % T;
    ll c = y / T; ll d = y % T;
    ll e = a * c / T; ll f = a * c % T;
    ll v = ((a * d + b * c) % n + e * t) % n;
    ll g = v / T; ll h = v % T;
    ll ans = (((f + g) * t % n + b * d) % n + h * T) % n;
    while (ans < 0) ans += n;
    return ans;
}
struct bign {
    int len;
    int s[maxn];
    bign(const char *str = "0"){ (*this) = str; }
    bign operator= (const char *str){
        int i;
        int j = strlen(str) - 1;
        len = j / num_digit + 1;
        for(i = 0; i <= len ; i++) s[i] = 0;
        for(i = 0; i <= j; i++){
            int k = (j - i) / num_digit + 1;
            s[k] = s[k] * 10 + str[i] - '0';
        }
        return *this ;
    }
};
```

```
void print(const bign &a){
    printf("%d", a.s[a.len]);
    for(int i = a.len - 1; i >= 1; i--)
        printf("%0*d", num_digit, a.s[i]);
}
//比较的前提是整数没有前导 0
int compare (const bign &a, const bign &b){
    if(a.len > b.len) return 1;
    if(a.len < b.len) return -1;
    int i = a.len;
    while ((i > 1) && (a.s[i] == b.s[i])) i--;
    return a.s[i] - b.s[i];
}
inline bool operator< (const bign &a, const bign &b)
{
    return compare(a, b) < 0;
}
inline bool operator<= (const bign &a, const bign &b)
{
    return compare(a, b) <= 0;
}
inline bool operator== (const bign &a, const bign &b)
{
    return compare(a, b) == 0;
}
//加法和减法很容易写出，只需注意不要忽略前导 0
bign operator+ (const bign &a, const bign &b){
    bign c;
    int i;
    for(i = 1; i <= a.len || i <= b.len || c.s[i]; i++){
        if(i <= a.len) c.s[i] += a.s[i];
        if(i <= b.len) c.s[i] += b.s[i];
        c.s[i+1] = c.s[i] / base;
        c.s[i] %= base;
    }
    c.len = i-1;
    if(c.len == 0) c.len = 1;
    return c;
}
//减法的前提是 a > b
bign operator- (const bign &a, const bign &b){
    bign c;
    int i, j;
    for(i = 1, j = 0; i <= a.len; i++){
        c.s[i] = a.s[i] - j;
        if(i <= b.len ) c.s[i] -= b.s[i];
```

```
        if(c.s[i] < 0){ j = 1; c.s[i] += base; }
        else j = 0;
    }
    c.len = a.len ;
    while (c.len > 1 && ! c.s[c.len]) c.len--;
    return c;
}
bign operator* (const bign &a, const bign &b){
    bign c;
    ll g = 0;
    int i, k;
    c.len = a.len + b.len;
    c.s[0] = 0;
    for(i = 1; i <= c.len; i++) c.s[i] = 0;
    for(k = 1; k <= c.len; k++){
        ll tmp = g;
        i = k + 1 - b.len;
        if(i < 1) i = 1;
        for (; i <= k && i <= a.len; i++)
            tmp += (ll)a.s[i] * (ll)b.s[k+1-i];
        g = tmp / base;
        c.s[k] = tmp % base;
    }
    while (c.len > 1 && !c.s[c.len]) c.len--;
    return c;
}
bign operator/ (const bign &a, int n)
{
    ll g = 0;
    bign c;
    c.len = a.len;
    for (int i = a.len; i > 0; --i)
    {
        ll tmp = g * base + a.s[i];
        c.s[i] = tmp / n;
        g = tmp % n;
    }
    while (c.len > 1 && !c.s[c.len]) c.len--;
    return c;
}
bign operator/ (const bign &a, const bign &b)
{
    bign L = "0", R = a;
    while (L < R)
    {
        bign M = L + (R - L + "1") / 2;
```

```
        if (M * b <= a) L = M;
        else R = M - "1";
    }
    return L;
}
ll bigmod(const bign &a, ll m){
    ll d = 0;
    for(int i = a.len; i > 0; --i){
        d = mul_mod(d, base, m);
        d = (d + a.s[i]) % m;
    }
    return d;
}
bign sqrt(const bign &n){
    bign c, d, x, y = n;
    do
    {
        x = y;
        y = (x + n / x) / 2;
    }
    while (y < x);
    return x;
}
bign gcd(bign a, bign b)
{
    bign c = "1";
    for (;;)
    {
        if (a == b)
            return a * c;
        else if (a.s[1] % 2 == 0 && b.s[1] % 2 == 0)
        {
            a = a / 2;
            b = b / 2;
            c = c * "2";
        }
        else if (a.s[1] % 2 == 0)
        {
            a = a / 2;
        }
        else if (b.s[1] % 2 == 0)
        {
            b = b / 2;
        }
        else if (b < a)
        {
```

```
            a = a - b;
        }
        else
        {
            b = b - a;
        }
    }
}
int main()
{
    bign a("345345345436546"), b("26768"), c, d;
    //divide(a, b, c, d);
    c = gcd(a, b);
    print(c);
    //print(a*b);
    //cout << '\n' << 3445453953435LL * 897676LL;
    return 0;
}
```

5.2 分数类

```
using integer = long long;
struct frac {
    integer num, den;
    frac() : num(0), den(1){}
    frac(integer val) {
        num = val;
        den = 1;
    }
    frac(integer a, integer b) {
        integer g = gcd(a, b);
        num = a / g;
        den = b / g;
        if (den < 0) {
            num = -num;
            den = -den;
        }
    }
};
frac operator+ (frac x, frac y) {
    return frac(x.num * y.den + y.num * x.den, x.den * y.den);
}
frac operator- (frac x, frac y) {
    return frac(x.num * y.den - y.num * x.den, x.den * y.den);
}
frac operator* (frac x, frac y) {
    return frac(x.num * y.num, x.den * y.den);
```

```
}
frac operator/ (frac x, frac y) {
    return frac(x.num * y.den, x.den * y.num);
}
frac operator+ (frac x, integer val) {
    return frac(x.num + x.den * val, x.den);
}
frac operator* (frac x, integer val) {
    return frac(x.num * val, x.den);
}
frac operator/ (frac x, integer val) {
    return frac(x.num, x.den * val);
}
bool operator== (frac x, frac y) {
    return x.num == y.num && x.den == y.den;
}
bool operator!= (frac x, frac y) {
    return x.num != y.num || x.den != y.den;
}
bool operator< (frac x, frac y) {
    return x.num * y.den < y.num * x.den;
}
frac abs(frac x) {
    if (x.num < 0)
        return frac(-x.num, x.den);
    return x;
}
int main() {
    printf("%d\n", gcd(0, 6));
    return 0;
}
```

6 动态规划

6.1 斯坦纳树（点权）

```
/*
BZOJ2595
输入一张带有点权的图，求将所有权值为 0 的结点连接起来所需要的最小费用（最小权值和）。
*/
const int inf = 0x3f3f3f3f; //inf + inf 必须小于 INT_MAX
                            //如果此处修改 inf，那么也应当在 init 函数中修改 dp 数组的初始值。
const int maxn = 130;
const int maxedges = 1100;
const int maxstate = 11000;
struct SteinerTree {
    int n, k; //n 为图中点的个数，k 为斯坦纳树的结点数
```

```
int dp[maxn][maxstate], st[maxn], val[maxn]; //若点 i 为点集中的点则 st[i] 为该点对应的状态，否
↪ 则为 0
int first[maxn], nxt[maxedges * 2], to[maxedges * 2], cur; //图的邻接表
pair<int, int> pre[maxn][maxstate];
bool inq[maxn][maxstate], vis[maxn];
queue<int> Q;
void init(int n, int* w, vector<int> c) { //n 为图中的所有结点个数，c 为斯坦纳树的结点集合
    this->n = n; this->k = c.size();       //w 为点权
    memset(dp, 0x3f, sizeof(dp));
    memset(pre, 0x3f, sizeof(pre));
    memset(st, 0, sizeof(st));
    for (int i = 0; i < k; ++i)
        st[c[i]] = (1 << i);
    for (int i = 1; i <= n; ++i)
        dp[i][st[i]] = 0;
    for (int i = 1; i <= n; ++i)
        val[i] = w[i];
    memset(inq, 0, sizeof(inq));
    memset(vis, 0, sizeof(vis));
    while (!Q.empty()) Q.pop();
    memset(first, 0, sizeof(first));
    cur = 0;
}
void add_edge(int u, int v) {
    nxt[++cur] = first[u];
    first[u] = cur;
    to[cur] = v;
}
void spfa(int s) { //对当前点集状态为 s 的 dp 值进行松弛
    while (!Q.empty()) {
        int u = Q.front(); Q.pop();
        for (int i = first[u]; i; i = nxt[i]) {
            int v = to[i];
            if (dp[v][s] > dp[u][s] + val[v]) {
                dp[v][s] = dp[u][s] + val[v];
                pre[v][s] = make_pair(u, s);
                if (!inq[v][s]) {
                    Q.push(v);
                    inq[v][s] = true;
                }
            }
        }
        inq[u][s] = false;
    }
}
void solve() {    //斯坦纳树的权值和为 min{dp[i][(1 << k) - 1]}, 1 <= i <= n
```

```
        for (int j = 1; j < (1 << k); ++j) {
            for (int i = 1; i <= n; ++i) {
                for (int sub = (j - 1) & j; sub; sub = (sub - 1) & j) {
                    int x = sub, y = j - sub;
                    int t = dp[i][x] + dp[i][y] - val[i];
                    if (dp[i][j] > t) {
                        dp[i][j] = t;
                        pre[i][j] = make_pair(i, sub);
                    }
                }
                if (dp[i][j] < inf) {
                    Q.push(i);
                    inq[i][j] = true;
                }
            }
            spfa(j);
        }
    }
    void dfs(int i, int state) {
        if (i == inf || pre[i][state].second == 0)
            return;
        vis[i] = 1; //vis[i] 表示结点 i 存在于斯坦纳树中
        pair<int, int> pr = pre[i][state];
        dfs(pr.first, pr.second);
        if (pr.first == i)
            dfs(i, state - pr.second);
    }
}tree;
int A[maxn][maxn], weight[maxn], dr[] = { -1, 1, 0, 0 }, dc[] = { 0, 0, -1, 1 };
int main() {
#define node(i, j) ((i) * m + (j) + 1)
    //freopen("in.txt", "r", stdin);
    int n, m, nd = -1;
    scanf("%d %d", &n, &m);
    vector<int> c;
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < m; ++j) {
            scanf("%d", &A[i][j]);
            weight[node(i, j)] = A[i][j];
            if (A[i][j] == 0) {
                c.push_back(node(i, j));
                if (nd == -1) nd = node(i, j);
            }
        }
    }
    tree.init(n * m, weight, c);
```

```
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < m; ++j) {
            for (int k = 0; k < 4; ++k) {
                int x = i + dr[k], y = j + dc[k];
                if (x < 0 || y < 0 || x >= n || y >= m) continue;
                tree.add_edge(node(i, j), node(x, y));
            }
        }
    }
    tree.solve();
    printf("%d\n", tree.dp[nd][(1 << c.size()) - 1]);
    tree.dfs(nd, (1 << c.size()) - 1);
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            if (!A[i][j]) printf("x");
            else if (tree.vis[node(i, j)]) printf("o");
            else printf("_");
            if (j == m - 1) printf("\n");
        }
    }
    return 0;
}
```

6.2 斯坦纳树（边权）

```
/*
HDU4085
给定 n 个点，前 k 个点属于集合 A，后 k 个点属于集合 B。点之间通过 m 条带权边相连。
要求选出权值和最小的边集使得 A 集合中的每个点都可以与 B 中的某个点进行配对（若 a 能走到 b 则可以考虑
↪ 是使其配对）。
解决方案：将集合 A 与集合 B 中的点作为关键点求出斯坦纳树，因为最后的结果可以是一个森林，所以我们在进行
↪ 一次 DP。
*/
const int inf = 0x3f3f3f3f; //inf + inf 必须小于 INT_MAX
                            //如果此处修改 inf，那么也应当在 init 函数中修改 dp 数组的初始值。
const int maxn = 110;
const int maxedges = 1100;
const int maxstate = 11000;
struct SteinerTree{
    int n, k; //n 为图中点的个数，k 为斯坦纳树的结点数
    int dp[maxn][maxstate], st[maxn]; //若点 i 为点集中的点则 st[i] 为该点对应的状态，否则为 0
    int first[maxn], nxt[maxedges * 2], to[maxedges * 2], weight[maxedges * 2], cur; //图的邻接表
    bool inq[maxn][maxstate];
    queue<int> Q;
    void init(int n, vector<int> c) { //n 为图中的所有结点个数，c 为斯坦纳树的结点集合
        this->n = n; this->k = c.size();
        memset(dp, 0x3f, sizeof(dp));
```

```
    memset(st, 0, sizeof(st));
    for(int i = 0; i < k; ++i)
        st[c[i]] = (1 << i);
    for(int i = 1; i <= n; ++i)
        dp[i][st[i]] = 0;
    memset(inq, 0, sizeof(inq));
    while(!Q.empty()) Q.pop();
    memset(first, 0, sizeof(first));
    cur = 0;
}
void add_edge(int u, int v, int w) {
    nxt[++cur] = first[u];
    first[u] = cur;
    to[cur] = v;
    weight[cur] = w;
}
void spfa(int s) { //对当前点集状态为 s 的 dp 值进行松弛
    while (!Q.empty()) {
        int u = Q.front(); Q.pop();
        inq[u][s] = false;
        for (int i = first[u]; i; i = nxt[i]) {
            int v = to[i], w = weight[i];
            int state = st[v] | s;
            if (dp[v][state] > dp[u][s] + w) {
                dp[v][state] = dp[u][s] + w;
                if (state != s || inq[v][s])
                    continue;
                Q.push(v);
                inq[v][s] = true;
            }
        }
    }
}
void solve() {    //斯坦纳树的权值和为 min{dp[i][(1 << k) - 1]}, 1 <= i <= n
    for (int j = 1; j < (1 << k); ++j) {
        for (int i = 1; i <= n; ++i) {
            if (st[i] && (st[i] & j) == 0)
                continue;
            for (int sub = (j - 1) & j; sub; sub = (sub - 1) & j) {
                int x = st[i] | sub, y = st[i] | (j - sub);
                dp[i][j] = min(dp[i][j], dp[i][x] + dp[i][y]);
            }
            if (dp[i][j] != inf) {
                Q.push(i);
                inq[i][j] = true;
            }
```

```
            }
            spfa(j);
        }
        /*
        int ans = inf;
        for (int i = 1; i <= n; ++i)
            ans = min(ans, tree.dp[i][(1 << k) - 1]);
        */
    }
}tree;
int n, m, k, d[maxstate];
bool check(int s) { //当且仅当状态 s 中 A 集合与 B 集合结点个数相等时，才是合法状态
    int res = 0;
    for (int i = 0; s; i++, s >>= 1)
        res += (s & 1) * (i < k ? 1 : -1);
    return (res == 0 ? true : false);
}
int main() {
    //freopen("in.txt", "r", stdin);
    int T;
    scanf("%d", &T);
    while (T--) {
        scanf("%d %d %d", &n, &m, &k);
        vector<int> c;
        for (int i = 1; i <= k; ++i)
            c.push_back(i);
        for (int i = 1; i <= k; ++i)
            c.push_back(n - k + i);
        tree.init(n, c);
        for (int i = 0; i < m; ++i) {
            int u, v, w;
            scanf("%d %d %d", &u, &v, &w);
            tree.add_edge(u, v, w);
            tree.add_edge(v, u, w);
        }
        tree.solve();
        const int mask = (1 << (2 * k)) - 1;
        for (int s = 0; s <= mask; s++) {
            d[s] = inf;
            for (int i = 1; i <= n; i++)
                d[s] = min(d[s], tree.dp[i][s]); //初始值为状态 s 中的点连成一棵树的情况
        }
        for (int s = 1; s <= mask; s++) if (check(s))
            for (int p = (s - 1) & s; p; p = (p - 1) & s) if (check(p))
                d[s] = min(d[s], d[p] + d[s - p]); //考虑将树分解为森林
        if (d[mask] >= inf)
```

```
            puts("No solution");
        else
            printf("%d\n", d[mask]);
    }
    return 0;
}
```

6.3 插头 DP

```
const int INF = 100000000;
int nrows, ncols;
int G[10][10];
// 插头编号：0 表示无插头，1 表示和数字 2 连通，2 表示和数字 3 连通
struct State {
    int up[9]; // up[i](0<=i<m) 表示第 i 列处轮廓线上方的插头编号
    int left; // 当前格（即下一个要放置的方格）左侧的插头
    // 三进制编码
    int encode() const {
        int key = left;
        for (int i = 0; i < ncols; i++)
            key = key * 3 + up[i];
        return key;
    }
    // 在 (row,col) 处放一个新方格。UDLR 分别为该方格上下左右四个边界上的插头编号
    // 产生的新状态存放在 T 里，成功返回 true，失败返回 false
    bool next(int row, int col, int U, int D, int L, int R, State& T) const {
        if (row == nrows - 1 && D != 0) return false; // 最下行下方不能有插头
        if (col == ncols - 1 && R != 0) return false; // 最右列右边不能有插头
        int must_left = (col > 0 && left != 0); // 是否必须要有左插头
        int must_up = (row > 0 && up[col] != 0); // 是否必须要有上插头
        if ((must_left && L != left) || (!must_left && L != 0)) return false; // 左插头不匹配
        if ((must_up && U != up[col]) || (!must_up && U != 0)) return false; // 上插头不匹配
        if (must_left && must_up && left != up[col]) return false; // 若左插头和上插头都存在，二者
↪   必须匹配
        // 产生新状态。实际上只有当前列的下插头和 left 插头有变化
        for (int i = 0; i < ncols; i++) T.up[i] = up[i];
        T.up[col] = D;
        T.left = R;
        return true;
    }
};
int memo[9][9][59049]; // 3^10
// 当前要放置格子 (row, col)，状态为 S。返回最小总长度
int rec(int row, int col, const State& S) {
    if (col == ncols) { col = 0; row++; }
    if (row == nrows) return 0;
    int key = S.encode();
```

```
    int& res = memo[row][col][key];
    if (res >= 0) return res;
    res = INF;
    State T;
    if (G[row][col] <= 1) { // 空格（0）或者障碍格（1）
        if (S.next(row, col, 0, 0, 0, 0, T)) res = min(res, rec(row, col + 1, T)); // 整个格子里都不连线
        if (G[row][col] == 0) // 如果是空格，可以连线。由于线不能分叉，所以这条线一定连接格子的某两个边界（6 种情况）
            for (int t = 1; t <= 2; t++) { // 枚举线的种类。t=1 表示 2 线，t=2 表示 3 线
                if (S.next(row, col, t, t, 0, 0, T)) res = min(res, rec(row, col + 1, T) + 2); // 上 <-> 下
                if (S.next(row, col, t, 0, t, 0, T)) res = min(res, rec(row, col + 1, T) + 2); // 上 <-> 左
                if (S.next(row, col, t, 0, 0, t, T)) res = min(res, rec(row, col + 1, T) + 2); // 上 <-> 右
                if (S.next(row, col, 0, t, t, 0, T)) res = min(res, rec(row, col + 1, T) + 2); // 下 <-> 左
                if (S.next(row, col, 0, t, 0, t, T)) res = min(res, rec(row, col + 1, T) + 2); // 下 <-> 右
                if (S.next(row, col, 0, 0, t, t, T)) res = min(res, rec(row, col + 1, T) + 2); // 左 <-> 右
            }
    }
    else {
        int t = G[row][col] - 1; // 数字为 2 和 3，但插头类型是 1 和 2，所以要减 1
        // 由于线不能分叉，所以这条线一定连接格子中间的数字和某一个边界（4 种情况）
        if (S.next(row, col, t, 0, 0, 0, T)) res = min(res, rec(row, col + 1, T) + 1); // 从上边界出来
        if (S.next(row, col, 0, t, 0, 0, T)) res = min(res, rec(row, col + 1, T) + 1); // 从下边界出来
        if (S.next(row, col, 0, 0, t, 0, T)) res = min(res, rec(row, col + 1, T) + 1); // 从左边界出来
        if (S.next(row, col, 0, 0, 0, t, T)) res = min(res, rec(row, col + 1, T) + 1); // 从右边界出来
    }
    return res;
}
int main() {
    while (scanf("%d%d", &nrows, &ncols) == 2 && nrows && ncols) {
        for (int i = 0; i < nrows; i++)
            for (int j = 0; j < ncols; j++)
                scanf("%d", &G[i][j]);
        State S;
        memset(&S, 0, sizeof(S));
        memset(memo, -1, sizeof(memo));
```

```
        int ans = rec(0, 0, S);
        if (ans == INF) ans = 0;
        printf("%d\n", ans / 2);
    }
    return 0;
}
/*
    void normalize() {
        int rep[maxn] = {}, num = 0;
        for (int i = 0; i < m; ++i) if (cp[i] > 0) {
            if (rep[cp[i]] <= 0)
                rep[cp[i]] = ++num;
            cp[i] = rep[cp[i]];
        }
        if (left > 0) {
            if (rep[left] <= 0)
                rep[left] = ++num;
            left = rep[left];
        }
    }
    // 把所有编号为 b 的连通分量改成 a
    void merge(int a, int b) {
        if (a == b) return;
        for (int i = 0; i < ncols; i++)
            if (comp[i] == b) comp[i] = a;
    }
*/
```

6.4 最长上升子序列

```
const int maxn = 110000;
const int inf = 1 << 30;
//d[i] 表示以下标 i 结尾的最长上升子序列长度
int a[maxn], g[maxn], d[maxn], n;
int main() {
    scanf("%d", &n);
    for (int i = 1; i <= n; ++i)
        scanf("%d", &a[i]);
    for (int i = 1; i <= n; ++i)
        g[i] = inf;
    for (int i = 1; i <= n; ++i) {
        int k = lower_bound(g + 1, g + n + 1, a[i]) - g;
        g[k] = a[i];
        d[i] = k;
    }
    int ans = *max_element(d + 1, d + n + 1);
    printf("%d\n", ans);
```

```
    return 0;
}
```

7 二分搜索

7.1 二分查找

```
#define MAXN 1000000
int BSearch_Upper(function<bool(int)> ok) //求满足 ok 函数的值的上界（满足 ok 函数的最大值）
{
    int L = 0, R = MAXN;
    while (L < R)
    {
        int M = L + (R - L + 1) / 2;
        if (ok(M)) L = M;
        else R = M - 1;
    }
    return L;
}
int BSearch_Lower(function<bool(int)> ok) //求满足 ok 函数的值的下界（满足 ok 函数的最小值）
{
    int L = 0, R = MAXN;
    while (L < R)
    {
        int M = L + (R - L) / 2;
        if (ok(M)) R = M;
        else L = M + 1;
    }
    return L;
}
double BSearch_Upper_Double(function<bool(double)> ok) //精度型二分法（满足 ok 函数的最大值）
{
    double L = 0, R = MAXN;
    while (R - L > 1e-5)
    {
        double M = (L + R) / 2;
        if (ok(M)) L = M;
        else R = M;
    }
    return L;
}
double BSearch_Lower_Double(function<bool(double)> ok) //精度型二分法（满足 ok 函数的最小值）
{
    double L = 0, R = MAXN;
    while (R - L > 1e-5)
    {
        double M = (L + R) / 2;
```

```
        if (ok(M)) R = M;
        else L = M;
    }
    return L;
}
int main()
{
    std::cout << BSearch_Upper([](int v) {return v <= 5; }) << std::endl;
    std::cout << BSearch_Lower([](int v) {return v >= 5; }) << std::endl;
    std::cout << BSearch_Upper_Double([](double v) {return v <= 5; }) << std::endl;
    std::cout << BSearch_Lower_Double([](double v) {return v >= 5; }) << std::endl;
    return 0;
}
```

8 莫队算法

8.1 莫队算法

```
const int maxn = 110000; // maxn > n + sqrt(n)
inline void del(int index) { //删除下标为 index 的元素
}
inline void add(int index) { //添加下标为 index 的元素
}
inline int get_ans() { //用小于 O(sqrt(n)) 的时间复杂度求出当前区间的解
    return 0;
}
namespace MoDui {
    struct query { //查询区间为 [l, r], id 表示是第几次查询
        int l, r, id;
    };
    int belong[maxn], ans[maxn];
    bool operator<(query a, query b) {
        if (belong[a.l] != belong[b.l])
            return belong[a.l] < belong[b.l];
        else if (belong[a.l] & 1)
            return a.r < b.r;
        else
            return a.r > b.r;
    }
    void work(int n, vector<query> q) { //n 指定数组大小（下标从 1 开始）
        int size = sqrt(n);
        int number = ceil((double)n / size);
        for (int i = 1; i <= number; ++i)
            for (int j = (i - 1) * size + 1; j <= i * size; ++j)
                belong[j] = i;
        sort(q.begin(), q.end());
        int l = 1, r = 0;
```

```
        for (auto item : q) {
            while (l < item.l) del(l++);
            while (l > item.l) add(--l);
            while (r < item.r) add(++r);
            while (r > item.r) del(r--);
            ans[item.id] = get_ans();
        }
    }
}
int main() {
    return 0;
}
```

8.2 回滚莫队

```
/*
left[i] 表示第 i 块的左端点编号（块编号从 1 开始）
right[i] 表示第 i 块的右端点编号
belong[i] 表示下标 i 所在的块编号（下标从 1 开始）
*/
const int maxn = 110000;
void init() { //用 O(1) 的时间复杂度创建一个空的区间
}
void add_right(int index) { //将输入数组中下标 index 的元素添加到当前区间的右边
}
void add_left(int index) {    //将输入数组中下标 index 的元素添加到当前区间的左边
                              //该函数可能需要保存原状态，以便于之后状态的恢复
}
void breakpoint() { //记录下当前的状态以便于之后恢复
                    //调用该函数之后会调用 add_left 修改状态，最后用 resume 恢复到调用 breakpoint
↪   之前的状态
}
void resume() { //撤销所有的 add_left 操作，恢复到调用 breakpoint 之前的状态（一次性撤销所有的
↪   add_left）
}
int get_ans() { //用小于 O(sqrt(n)) 的时间复杂度求出当前区间的解
    return 0;
}
namespace MoDui {
    struct query { //查询区间为 [l, r]，id 表示是第几次查询
        int l, r, id;
    };
    int left[maxn], right[maxn], belong[maxn], ans[maxn]; // ans 可能要用 long long 存
    bool operator< (query a, query b) {
        return belong[a.l] != belong[b.l] ? belong[a.l] < belong[b.l] : a.r < b.r;
    }
    void work(int n, vector<query> q) { //n 指定数组大小（下标从 1 开始）
```

```
        int size = sqrt(n);
        int number = ceil((double)n / size);
        for (int i = 1; i <= number; ++i) {
            left[i] = size * (i - 1) + 1;
            right[i] = size * i;
            for (int j = left[i]; j <= right[i]; ++j)
                belong[j] = i;
        }
        right[number] = n;
        sort(q.begin(), q.end());
        for (auto item : q) if (belong[item.l] == belong[item.r]) {
            init();
            for (int i = item.l; i <= item.r; ++i)
                add_right(i);
            ans[item.id] = get_ans();
        }
        for (int i = 0, k = 1; k <= number; ++k) {
            init();
            for (int r = right[k]; i < q.size() && belong[q[i].l] == k; ++i) {
                int ql = q[i].l, qr = q[i].r;
                int l = right[k] + 1;
                if (belong[ql] != belong[qr]) {
                    while (r < qr) {
                        ++r;
                        add_right(r);
                    }
                    breakpoint();
                    while (l > ql) {
                        --l;
                        add_left(l);
                    }
                    ans[q[i].id] = get_ans();
                    resume();
                }
            }
        }
    }
}
int main() {
    return 0;
}
```

8.3 带修改莫队

```
const int maxn = 210000; // maxn > n + pow(n, 2.0 / 3.0)
struct query {             //tp == 0 对应查询操作：查询区间为 [l, r]
    int tp, l, r, id, t;//tp == 1 对应修改操作：将下标为 l 的位置修改为 r
```

```
};
inline void del(int index) { //删除下标为 index 的元素
}
inline void add(int index) { //添加下标为 index 的元素
}
inline void change(query& q, bool flag) { //将输入数组下标为 q.l 的值修改为 q.r, flag 表示这次修改
↪  是否位于当前区间内
                                           //flag 为 true 则考虑这次修改对答案的影响，否则不考虑
}
inline void undo(query& q, bool flag) { //撤销修改操作，flag 表示这次撤销是否位于当前处理的区间内
}
inline int get_ans() { //用小于 O(sqrt(n)) 的时间复杂度求出当前区间的解
    return 0;
}
int belong[maxn], ans[maxn];
bool operator< (query a, query b) {
    if (belong[a.l] != belong[b.l])
        return belong[a.l] < belong[b.l];
    else if (belong[a.r] != belong[b.r])
        return belong[a.r] < belong[b.r];
    else
        return a.t < b.t;
}
void work(int n, vector<query> q) { //n 指定数组大小（下标从 1 开始）
    int size = pow(n, 2.0 / 3.0);
    int number = ceil((double)n / size);
    for (int i = 1; i <= number; ++i)
        for (int j = (i - 1) * size + 1; j <= i * size; ++j)
            belong[j] = i;
    vector<query> a, b; //a 记录查询，b 记录修改
    for (auto item : q) {
        if (item.tp == 0) {
            item.id = a.size();
            item.t = b.size();
            a.push_back(item);
        }
        else {
            b.push_back(item);
        }
    }
    sort(a.begin(), a.end());
    int l = 1, r = 0, t = 0;
    for (auto item : a) {
        while (l < item.l) del(l++);
        while (l > item.l) add(--l);
        while (r < item.r) add(++r);
```

```
        while (r > item.r) del(r--);
        while (t < item.t) {
            change(b[t], item.l <= b[t].l && item.r >= b[t].l);
            ++t;
        }
        while (t > item.t) {
            --t;
            undo(b[t], item.l <= b[t].l && item.r >= b[t].l);
        }
        ans[item.id] = get_ans();
    }
    /*
    for (int i = 0; i < a.size(); ++i)
        printf("%d\n", ans[i]);
    */
}
int main() {
    return 0;
}
```

9 数据结构

9.1 可修改优先队列

```
template<typename T, int maxsize> struct ModifiablePriorityQueue { // 大根堆
    int data[maxsize * 2];
    int pos[maxsize * 2];
    T value[maxsize * 2];
    int sz;
    ModifiablePriorityQueue() : pos(), sz(0) {}
    void up(int i) { // 值变大
        auto index = data[i];
        auto val = value[index];
        while (i > 1) {
            int fa = i / 2;
            if (val > value[data[fa]]) {
                data[i] = data[fa];
                pos[data[i]] = i;
                i = fa;
            }
            else {
                break;
            }
        }
        data[i] = index;
        pos[index] = i;
    }
```

```
void down(int i) { // 值变小
    auto index = data[i];
    auto val = value[index];
    while (i * 2 <= sz) {
        int child = i * 2;
        if (i * 2 + 1 <= sz && value[data[i * 2 + 1]] > value[data[child]]) {
            child = i * 2 + 1;
        }
        if (val < value[data[child]]) {
            data[i] = data[child];
            pos[data[i]] = i;
            i = child;
        }
        else {
            break;
        }
    }
    data[i] = index;
    pos[index] = i;
}
void push(int index, const T& val) {
    sz += 1;
    data[sz] = index;
    value[index] = val;
    up(sz);
}
void pop() {
    pos[data[1]] = 0;
    data[1] = data[sz];
    sz -= 1;
    if (sz > 0) {
        down(1);
    }
}
void erase(int index) {
    index = pos[index];
    pos[data[index]] = 0;
    data[index] = data[sz];
    sz -= 1;
    if (index <= sz) {
        down(index);
    }
}
void modify(int index, const T& val) {
    if (pos[index] == 0) {
        push(index, val);
```

```
        }
        else if (val > value[index]) {
            value[index] = val;
            up(pos[index]);
        }
        else {
            value[index] = val;
            down(pos[index]);
        }
    }
    int top() const {
        return data[1];
    }
    T get(int index) const {
        return value[index];
    }
    T maximum() const {
        return value[data[1]];
    }
    bool contains(int index) const {
        return pos[index] != 0;
    }
    int size() const {
        return sz;
    }
    bool empty() const {
        return sz == 0;
    }
};
int main() { // 洛谷 P4779
    static int d[210000];
    static vector<pair<int, int>> G[210000];
    int n, m, s;
    scanf("%d %d %d", &n, &m, &s);
    for (int i = 0; i < m; ++i) {
        int x, y, z;
        scanf("%d %d %d", &x, &y, &z);
        G[x].emplace_back(y, z);
    }
    static ModifiablePriorityQueue<int, 210000> Q;
    memset(d, 0x3f, sizeof(d));
    d[s] = 0;
    Q.push(s, 0);
    while (!Q.empty()) {
        auto x = Q.top(); Q.pop();
        for (auto [y, w] : G[x]) {
```

```
            if (d[y] > d[x] + w) {
                d[y] = d[x] + w;
                Q.modify(y, -d[y]);
            }
        }
    }
    for (int i = 1; i <= n; ++i) {
        printf("%d ", d[i] == 0x3f3f3f3f ? INT_MAX : d[i]);
    }
    printf("\n");
    return 0;
}
```

9.2 AVL Tree

```
struct node {
    node *ch[2]; // 左右子树
    int height, value;
    node() : height(0), value(0), ch() {}
    void maintain() {
        height = max(ch[0]->height, ch[1]->height) + 1;
    }
}*nil = new node;
void rotate(node* &o, int d) {
    node* k = o->ch[d ^ 1];
    o->ch[d ^ 1] = k->ch[d];
    k->ch[d] = o;
    o->maintain(); k->maintain();
    o = k;
}
node *Insert(int X, node *T)
{
    if (T == nil) {
        T = new node;
        T->value = X;
        T->height = 0;
        T->ch[0] = T->ch[1] = nil;
    }
    else if (X < T->value) {
        T->ch[0] = Insert(X, T->ch[0]);
        if (T->ch[0]->height - T->ch[1]->height == 2) {
            if (X >= T->ch[0]->value)
                rotate(T->ch[0], 0);
            rotate(T, 1);
        }
    }
    else if (X > T->value) {
```

```
        T->ch[1] = Insert(X, T->ch[1]);
        if (T->ch[1]->height - T->ch[0]->height == 2){
            if (X <= T->ch[1]->value)
                rotate(T->ch[1], 1);
            rotate(T, 0);
        }
    }
    T->height = max(T->ch[0]->height, T->ch[1]->height) + 1;
    return T;
}
int main() {
    //    freopen("in.txt", "r", stdin);
    int n;
    scanf("%d", &n);
    node *root = nil;
    for (int i = 0; i < n; ++i) {
        int val;
        scanf("%d", &val);
        root = Insert(val, root);
    }
    printf("%d\n", root->value);
    return 0;
}
```

9.3 TopTree

```
/*
K=0 表示子树修改，后面 x,y，表示以 x 为根的子树的点权值改成 y
K=1 表示换根，后面 x，表示把这棵树的根变成 x
K=2 表示链修改，后面 x,y,z，表示把这棵树中 x-y 的路径上点权值改成 z
K=3 表示子树询问 min，后面 x，表示以 x 为根的子树中点的权值 min
K=4 表示子树询问 max，后面 x，表示以 x 为根的子树中点的权值 max
K=5 表示子树加，后面 x,y，表示 x 为根的子树中点的权值 +y
K=6 表示链加，后面 x,y,z，表示把这棵树中 x-y 的路径上点权值改成 +z
K=7 表示链询问 min，后面 x,y，表示把这棵树中 x-y 的路径上点的 min
K=8 表示链询问 max，后面 x,y，表示把这棵树中 x-y 的路径上点的 max
K=9 表示换父亲，后面 x,y，表示把 x 的父亲换成 y，如果 y 在 x 子树里不操作。
K=10 表示链询问 sum，后面 x,y,z，表示表示把这棵树中 x-y 的路径上点的 sum
K=11 表示子树询问 sum，后面 x，表示以 x 为根的子树的点权 sum
*/
//const int E_ALL = 0, E_CH = 1, E_RE = 2;
//const int S_LCT = 0, S_AAA = 2;
const int maxn = 110000 << 2, inf = INT_MAX;
struct mark { //splay 树上的标记类
    int k, b;
    mark(int k = 1, int b = 0) :k(k), b(b) {} //k == 0 表示设置值，k == 1 表示增加值，b 表示修改或
↪   增加的值
```

```
    int operator ()(int x, int sz = 1) { //将当前标记类作用于值 x（sz 指定值的个数）
        return k * x + sz * b;
    }
    mark& operator +=(mark a) { //将当前标记与 a 结合
        b = b * a.k + a.b; //计算新的标记值
        k *= a.k; //若两个标记中有一个为 set 则结果为 set
        return *this;
    }
    bool operator !=(mark a) {
        return k != a.k || b != a.b;
    }
}none;
struct info { //splay 树上的信息类
    int minv, maxv, sumv, sz;
    info(int minv = inf, int maxv = -inf, int sumv = 0, int sz = 0) :minv(minv), maxv(maxv),
↪   sumv(sumv), sz(sz) {}
    info operator +(info rhs) { //将信息进行合并
        return info(min(minv, rhs.minv), max(maxv, rhs.maxv), sumv + rhs.sumv, sz + rhs.sz);
    }
    info& operator +=(mark b) { //将标记作用于信息
        if (sz) {
            minv = b(minv);
            maxv = b(maxv);
            sumv = b(sumv, sz);
        }
        return *this;
    }
};
struct node {
    node* son[4], * fa;
    info se, re, ch, all;
    mark mch, mre;
    bool rev, imag;
    node() { //construction for null node
        fill(son, son + 4, fa = 0);
        se = re = ch = all = info();
        mch = mre = none;
        rev = imag = 0;
    }
    int kind() {
        for (int i = 0; i < 4; ++i)
            if (fa->son[i] == this)
                return i;
    }
    bool isroot(int type) {
        return type ? (!fa->imag || !imag) : (fa->son[0] != this && fa->son[1] != this);
```

```
    }
    void link(int d, node* s) {
        son[d] = s;
        s->fa = this;
    }
    void set(node* s, int d) {
        down();
        son[d] = s;
        s->fa = this;
        up();
    }
    void reverse() {
        swap(son[0], son[1]);
        rev ^= 1;
    }
    void edit(mark f, int type) {
        //if (this == null) return;
        if (type == 0 || type == 1) {
            se += f;
            mch += f;
            ch += f;
        }
        if (type == 0 || type == 2) {
            mre += f;
            re += f;
        }
        all = ch + re;
    }
    void up() {
        ch = son[0]->ch + son[1]->ch + se;
        re = son[0]->re + son[1]->re + son[2]->all + son[3]->all;
        all = ch + re;
    }
    void down() {
        if (rev) {
            son[0]->reverse();
            son[1]->reverse();
            rev = 0;
        }
        if (mre != none) {
            son[0]->edit(mre, 2);
            son[1]->edit(mre, 2);
            son[2]->edit(mre, 0);
            son[3]->edit(mre, 0);
            mre = none;
        }
```

```
        if (mch != none) {
            son[0]->edit(mch, 1);
            son[1]->edit(mch, 1);
            mch = none;
        }
    }
}pool[maxn], *me = pool, *bin[maxn], **ptr = bin, *null = new(me++) node, *nd[maxn], *root;
node* newnode(int val = inf) { //val 表示结点的初始权值
    node* cur = *ptr ? *ptr-- : me++;
    fill(cur->son, cur->son + 4, cur->fa = null);
    cur->re = info();
    cur->mch = cur->mre = none;
    cur->rev = 0;
    if (val == inf) {
        cur->se = cur->all = cur->ch = info();
        cur->imag = 1;
    }
    else {
        cur->se = cur->all = cur->ch = info(val, val, val, 1);
        cur->imag = 0;
    }
    return cur;
}
void delnode(node*& pos) {
    *++ptr = pos;
    pos = 0;
}
void trans(node* pos) {
    node* fa = pos->fa, * grf = fa->fa;
    fa->down(), pos->down();
    int d = pos->kind();
    if (grf != null)
        grf->son[fa->kind()] = pos;
    pos->fa = grf;
    fa->link(d, pos->son[d ^ 1]);
    pos->link(d ^ 1, fa);
    fa->up();
}
void splay(node* pos, int type) {
    while (!pos->isroot(type)) {
        node* fa = pos->fa;
        if (!fa->isroot(type))
            trans(fa->kind() == pos->kind() ? fa : pos);
        trans(pos);
    }
    pos->up();
```

```
}
void addvirt(node* pos, node* fa) {
    if (pos == null) return;
    fa->down();
    for (int i = 2; i < 4; ++i) {
        if (fa->son[i] == null) {
            fa->set(pos, i);
            return;
        }
    }
    while (fa->son[2]->imag)
        fa = fa->son[2];
    splay(fa, 2);
    node* vi = newnode();
    vi->link(2, fa->son[2]);
    vi->link(3, pos);
    fa->set(vi, 2);
    splay(vi, 2);
}
void delvirt(node* pos) {
    if (pos == null) return;
    static node** top = (node * *)malloc(maxn * sizeof(node*));
    for (node* p = pos->fa; p->imag; p = p->fa)* ++top = p;
    if (*top) (*top)->fa->down();
    for (; *top; top--) (*top)->down();
    node* fa = pos->fa;
    int d = pos->kind();
    if (fa->imag) {
        fa->fa->set(fa->son[d ^ 1], fa->kind());
        splay(fa->fa, 2);
        delnode(fa);
    }
    else {
        fa->set(null, d);
        splay(fa, 2);
    }
    pos->fa = null;
}
void access(node* pos) {
    node* pred = null;
    while (pos != null) {
        splay(pos, 0);
        addvirt(pos->son[1], pos);
        delvirt(pred);
        pos->set(pred, 1);
        pred = pos;
```

```
        for (pos = pos->fa; pos->imag; pos = pos->fa);
    }
}
void touch(node* pos) {
    access(pos);
    splay(pos, 0);
}
void beroot(node* pos) { //将结点 pos 设置为根结点
    touch(pos);
    pos->reverse();
}
void link(node* x, node* y) {
    beroot(x);
    touch(y);
    addvirt(x, y);
}
void cut(node* x, node* y) {
    beroot(x);
    touch(y);
    y->set(null, 0);
    x->fa = null;
}
node* findroot(node* pos) {
    touch(pos);
    while (pos->son[0] != null) {
        pos->down();
        pos = pos->son[0];
    }
    splay(pos, 0);
    return pos;
}
node* findfa(node* pos) {
    touch(pos);
    pos = pos->son[0];
    if (pos == null) return 0;
    while (pos->son[1] != null) {
        pos->down();
        pos = pos->son[1];
    }
    splay(pos, 0);
    return pos;
}
void changefather(node *pos, node *father) {
    if (pos == root) return;
    node* fa = findfa(pos);
    cut(fa, pos);
```

```
    if (findroot(pos) != findroot(father))
        link(pos, father);
    else
        link(pos, fa);
}
namespace chain {
    info query(node *x, node *y) { //对 x 到 y 的链进行查询
        beroot(x);
        touch(y);
        return y->ch;
    }
    void edit(node *x, node *y, mark f) //对 x 到 y 的链进行修改
    {
        beroot(x);
        touch(y);
        y->edit(f, 1);
    }
}
namespace sub {
    info query(node *rt) { //在以 root 为根的情况下，对 rt 子树进行查询
        touch(rt);
        return rt->se + rt->son[2]->all + rt->son[3]->all;
    }
    void edit(node *rt, mark f) { //在以 root 为根的情况下，对 rt 子树进行修改
        touch(rt);
        rt->se += f;
        rt->son[2]->edit(f, 0);
        rt->son[3]->edit(f, 0);
    }
}
int main() {
#define get(x) scanf("%d",&x)
#define put(x) printf("%d",x)
    static int w[maxn], edge[maxn][2];
    int n, m, tmp; get(n), get(m);
    for (int i = 1; i <= n - 1; ++i)
        get(edge[i][0]), get(edge[i][1]);
    for (int i = 1; i <= n; ++i) {
        get(w[i]);
        nd[i] = newnode(w[i]);
    }
    get(tmp); root = nd[tmp];
    for (int i = 1; i <= n - 1; ++i)  link(nd[edge[i][0]], nd[edge[i][1]]);
    for (int i = 1; i <= m; ++i) {
        int opt, x, y, z, Ans; get(opt);
        info res;
```

```
beroot(root);
switch (opt)
{
case 0:
case 5: //子树修改
    get(x), get(y);
    sub::edit(nd[x], mark(opt == 0 ? 0 : 1, y)); //0 set 5 add
    break;
case 1:
    get(tmp); root = nd[tmp]; //换根
    break;
case 2:
case 6: //链修改
    get(x), get(y), get(z);
    chain::edit(nd[x], nd[y], mark(opt == 2 ? 0 : 1, z));
    break;
case 3://min
case 4://max
case 11://sum
    get(x);
    res = sub::query(nd[x]);
    switch (opt)
    {
    case 3:Ans = res.minv; break;
    case 4:Ans = res.maxv; break;
    case 11:Ans = res.sumv; break;
    }
    put(Ans), putchar('\n');
    break;
case 7://min
case 8://max
case 10://sum
    get(x), get(y);
    res = chain::query(nd[x], nd[y]);
    switch (opt)
    {
    case 7:Ans = res.minv; break;
    case 8:Ans = res.maxv; break;
    case 10:Ans = res.sumv; break;
    }
    put(Ans), putchar('\n');
    break;
case 9: //换父亲
    get(x), get(y);
    changefather(nd[x], nd[y]);
    break;
```

```
        }
    }
    return 0;
}
```

9.4 可持久化数组

```
#define lc t[p].lchild
#define rc t[p].rchild
template<typename T, int begin, int end, int factor = 30>
struct Array {
    static const int size = end - begin + 1;
    struct segment {
        int lchild, rchild;
        T val;
    }t[size * factor];
    int sz;
    void ins(int& p, int x, int y, T A[]) {
        p = ++sz;
        if (x == y)
            t[p].val = A[x];
        else {
            int mid = (x + y) >> 1;
            ins(lc, x, mid, A);
            ins(rc, mid + 1, y, A);
        }
    }
    //将数组 A 中的值作为初始值，并返回初始数组的版本号
    int init(T A[]) {
        int root = sz = 0;
        ins(root, begin, end, A);
        return root;
    }
    //将 val 作为可持久化数组的初始值，并返回初始数组的版本号（0）
    //采用此初始化方案时，数组的区间范围可以很大，
    //且可以在 get 函数中加入 p==0 就直接返回 t[0].val 的优化
    int init(T val) {
        t[0].val = val;
        return 0;
    }
    //获取版本号 p 的下标为 index 的值
    T get(int p, int index, int x = begin, int y = end) {
        if (x == y)
            return t[p].val;
        int mid = (x + y) >> 1;
        if (index <= mid)
            return get(lc, index, x, mid);
```

```
        else
            return get(rc, index, mid + 1, y);
    }
    //修改版本号为 p 的数组的下标 index 位置的值为 v
    //使用方法为: root[i] = root[i-1]; set(root[i], idx, v);
    void set(int &p, int index, const T& v, int x = begin, int y = end) {
        t[++sz] = t[p]; p = sz; //如果类型 T 的复制开销较大，此处可以考虑只复制左右子树的编号
        if (x == y)
            t[p].val = v;
        else {
            int mid = (x + y) >> 1;
            if (index <= mid)
                set(lc, index, v, x, mid);
            else
                set(rc, index, v, mid + 1, y);
        }
    }
};
const int maxn = 1000;
Array<int, 0, maxn, 100> arr;
int A[maxn];
int main() {
    for (int i = 0; i < maxn; ++i)
        A[i] = -1;
    int p = arr.init(-1);
    for (int i = 0; i < maxn; ++i)
        if (arr.get(p, i) != A[i])
            abort();
    vector<int> total[maxn];
    vector<int> root; root.push_back(p);
    total[0] = vector<int>(A, A + maxn);
    int T = 10000;
    while (T--) {
        int tp = rand() % 10;
        int pos = rand() % maxn, nd = rand() % root.size();
        int value = rand();
        if (tp != 0) {
            if (arr.get(root[nd], pos) != total[nd][pos]) {
                abort();
            }
        }
        else {
            int sz = root.size();
            total[sz] = total[nd];
            total[sz][pos] = value;
            root.push_back(root[nd]);
```

```
            arr.set(root.back(), pos, value);
        }
    }
    return 0;
}
```

9.5 四分树

```
const int maxn = 1100;
const int inf = 1 << 30;
int A[2000][2000];
int cur = 0, tot, mn, mx;
struct area {
    int x1, x2, y1, y2;
    int a, b, c, d;
    int sum, min, max, set, add; //set 必须是非负数
}t[maxn * maxn * 4];
void init() {
    cur = 0;
    t[0].min = inf;
    t[0].max = -inf;
    t[0].sum = 0;
}
void maintain(int p) { //maintain 函数用来维护结点信息，应当保证多次对一个结点调用 maintain 结果不变
    if (p == 0) return;
    int a = t[p].a, b = t[p].b, c = t[p].c, d = t[p].d;
    t[p].sum = t[p].min = t[p].max = 0;
    if (t[p].x1 != t[p].x2 || t[p].y1 != t[p].y2) {
        t[p].sum = t[a].sum + t[b].sum + t[c].sum + t[d].sum;
        t[p].min = min({ t[a].min, t[b].min, t[c].min, t[d].min });
        t[p].max = max({ t[a].max, t[b].max, t[c].max, t[d].max });
    }
    if (t[p].set >= 0) {
        t[p].min = t[p].max = t[p].set;
        t[p].sum = t[p].set * (t[p].x2 - t[p].x1 + 1) * (t[p].y2 - t[p].y1 + 1);
    }
    if (t[p].add) {
        t[p].min += t[p].add;
        t[p].max += t[p].add;
        t[p].sum += t[p].add * (t[p].x2 - t[p].x1 + 1) * (t[p].y2 - t[p].y1 + 1);
    }
}
void pushdown(int p) { //pushdown 将标记传递给子结点，不影响当前结点的信息。
    int a = t[p].a, b = t[p].b, c = t[p].c, d = t[p].d;
    if (t[p].set >= 0) {
        t[a].set = t[b].set = t[c].set = t[d].set = t[p].set;
        t[a].add = t[b].add = t[c].add = t[d].add = 0;
```

```
        t[p].set = -1;
    }
    if (t[p].add) {
        t[a].add += t[p].add;
        t[b].add += t[p].add;
        t[c].add += t[p].add;
        t[d].add += t[p].add;
        t[p].add = 0;
    }
}
int build(int x1, int x2, int y1, int y2) {
    int p = ++cur;
    t[p].x1 = x1;
    t[p].x2 = x2;
    t[p].y1 = y1;
    t[p].y2 = y2;
    t[p].a = t[p].b = t[p].c = t[p].d = 0;
    t[p].add = 0; t[p].set = -1; //清空结点标记
    if (x1 == x2 && y1 == y2) {
        t[p].add = A[x1][y1];
    }
    else {
        int xm = (x1 + x2) >> 1;
        int ym = (y1 + y2) >> 1;
        t[p].a = build(x1, xm, y1, ym);
        if (ym < y2)
            t[p].b = build(x1, xm, ym + 1, y2);
        if (xm < x2)
            t[p].c = build(xm + 1, x2, y1, ym);
        if (xm < x2 && ym < y2)
            t[p].d = build(xm + 1, x2, ym + 1, y2);
    }
    maintain(p);
    return p;
}
void update(int p, int U, int D, int L, int R, int op, int v) { //[U, D] 对应行号区间, [L, R] 对应
↪   列号区间。
    if (U <= t[p].x1 && D >= t[p].x2 && L <= t[p].y1 && R >= t[p].y2) {
        if (op == 0) t[p].add += v;
        else {
            t[p].set = v;
            t[p].add = 0;
        }
    }
    else {
        pushdown(p);
```

```
        int xm = (t[p].x1 + t[p].x2) >> 1;
        int ym = (t[p].y1 + t[p].y2) >> 1;
        if (U <= xm && L <= ym)
            update(t[p].a, U, D, L, R, op, v);
        else
            maintain(t[p].a);
        if (U <= xm && ym < R && ym < t[p].y2)
            update(t[p].b, U, D, L, R, op, v);
        else
            maintain(t[p].b);
        if (xm < D && L <= ym && xm < t[p].x2)
            update(t[p].c, U, D, L, R, op, v);
        else
            maintain(t[p].c);
        if (xm < D && ym < R && xm < t[p].x2 && ym < t[p].y2)
            update(t[p].d, U, D, L, R, op, v);
        else
            maintain(t[p].d);
    }
    maintain(p);
}
void query(int p, int U, int D, int L, int R) { //调用之前要设置: mn = inf; mx = -inf; tot = 0;
    if (U <= t[p].x1 && D >= t[p].x2 && L <= t[p].y1 && R >= t[p].y2) {
        tot += t[p].sum;
        mn = min(mn, t[p].min);
        mx = max(mx, t[p].max);
    }
    else {
        pushdown(p);
        maintain(t[p].a);
        maintain(t[p].b);
        maintain(t[p].c);
        maintain(t[p].d);
        int xm = (t[p].x1 + t[p].x2) >> 1;
        int ym = (t[p].y1 + t[p].y2) >> 1;
        if (U <= xm && L <= ym)
            query(t[p].a, U, D, L, R);
        if (U <= xm && ym < R && ym < t[p].y2)
            query(t[p].b, U, D, L, R);
        if (xm < D && L <= ym && xm < t[p].x2)
            query(t[p].c, U, D, L, R);
        if (xm < D && ym < R && xm < t[p].x2 && ym < t[p].y2)
            query(t[p].d, U, D, L, R);
    }
}
int main() {
```

```
#define ran engine(e)
    int n = 500, m = 500, q = 10000;
    default_random_engine e;
    e.seed(123);
    uniform_int_distribution<unsigned> engine(1, n);
    for (int i = 1; i <= n; ++i)
        for (int j = 1; j <= m; ++j)
            A[i][j] = ran;
    init();
    int root = build(1, n, 1, m);
    while (q--) {
        int a = ran, b = ran, c = ran, d = ran;
        int x1 = min(a, b), x2 = max(a, b), y1 = min(c, d), y2 = max(c, d);
        int tp = rand() % 3, v = ran;
        if (tp == 0) {
            for (int i = x1; i <= x2; ++i)
                for (int j = y1; j <= y2; ++j)
                    A[i][j] += v;
            update(root, x1, x2, y1, y2, 0, v);
        }
        else if (tp == 1) {
            for (int i = x1; i <= x2; ++i)
                for (int j = y1; j <= y2; ++j)
                    A[i][j] = v;
            update(root, x1, x2, y1, y2, 1, v);
        }
        else {
            mn = inf; mx = -inf; tot = 0;
            int MN = inf, MX = -inf, TOT = 0;
            for (int i = x1; i <= x2; ++i)
                for (int j = y1; j <= y2; ++j)
                    MN = min(MN, A[i][j]), MX = max(MX, A[i][j]), TOT += A[i][j];
            query(root, x1, x2, y1, y2);
            if (MN != mn)
                printf("mn: %d %d\n", MN, mn);
            if (mx != MX)
                printf("mx: %d %d\n", MX, mx);
            if (tot != TOT)
                printf("sum: %d %d\n", TOT, tot);
        }
    }
    return 0;
}
```

9.6 Treap

```
const int maxn = 5000000;
struct node {
    node *ch[2]; // 左右子树
    int r; // 随机优先级
    int v; // 值
    int s; // 结点总数
    int cmp(int x) const {
        if (x == v) return -1;
        return x < v ? 0 : 1;
    }
    void maintain() {
        s = ch[0]->s + ch[1]->s + 1;
    }
}nodes[maxn], *cur, *nil;
node *newnode(int x)
{
    node *o = cur++;
    o->ch[0] = o->ch[1] = nil;
    o->r = rand();
    o->v = x;
    o->s = 1;
    return o;
}
void init()
{
    cur = nodes;
    nil = newnode(0);
    nil->s = 0;
}
void rotate(node* &o, int d) {
    node* k = o->ch[d^1]; o->ch[d^1] = k->ch[d]; k->ch[d] = o;
    o->maintain(); k->maintain(); o = k;
}
void insert(node* &o, int x) {
    if(o == nil) o = newnode(x);
    else {
        int d = (x < o->v ? 0 : 1); // 不要用 cmp 函数，因为可能会有相同结点
        insert(o->ch[d], x);
        if(o->ch[d]->r > o->r) rotate(o, d^1);
    }
    o->maintain();
}
node *find(node* o, int x) {
    while (o != nil)
    {
```

```
        int d = o->cmp(x);
        if (d == -1) return o;
        else o = o->ch[d];
    }
    return nil;
}
// 要确保结点存在
void remove(node* &o, int x) {
    int d = o->cmp(x);
    if (d == -1)
    {
        if (o->ch[0] == nil)
            o = o->ch[1];
        else if (o->ch[1] == nil)
            o = o->ch[0];
        else
        {
            int d2 = (o->ch[0]->r > o->ch[1]->r ? 1 : 0);
            rotate(o, d2); remove(o->ch[d2], x);
        }
    }
    else
        remove(o->ch[d], x);
    if(o != nil) o->maintain();
}
int kth(node* o, int k) {
    if(o == nil || k <= 0 || k > o->s) return 0;
    if(k == o->ch[0]->s+1) return o->v;
    else if(k <= o->ch[0]->s) return kth(o->ch[0], k);
    else return kth(o->ch[1], k - o->ch[0]->s - 1);
}
// 返回在以 o 为根的子树中 x 的排名，没有 x 则返回 x 的后继的排名
int rank(node* o, int x) {
    if(o == nil) return 1;
    if(x <= o->v) return rank(o->ch[0], x);
    else return rank(o->ch[1], x) + o->ch[0]->s + 1;
}
// 返回在以 o 为根的子树中 x 的前驱的排名
int rank1(node *o, int x)
{
    if (o == nil)
        return 0;
    if (x <= o->v)
        return rank1(o->ch[0], x);
    else return rank1(o->ch[1], x) + o->ch[0]->s + 1;
}
```

```
// 返回在以 o 为根的子树中 x 的后继的排名
int rank2(node *o, int x)
{
    if (o == nil)
        return 1;
    if (x >= o->v)
        return rank2(o->ch[1], x) + o->ch[0]->s + 1;
    else return rank2(o->ch[0], x);
}
/*
对于 rank、rank1、rank2 的理解:
给定序列 A = {1, 2, 3, 3, 4, 4, 5}
下标 (排名): 1 2 3 4 5 6 7
将序列 A 插入到平衡树 root 中。
rank(root, 3): 因为只要 x<=o->v 就一直往左走, 所以会走到排名 2~3 的位置, 到达 nil 结点返回 1, 所以
↪ 最外层 rank 返回值为 3
rank1(root, 3): 因为只要 x<=o->v 就一直往左走, 所以会走到排名 2~3 的位置, 到达 nil 结点返回 0, 所以
↪ 最外层 rank 返回值为 2
rank2(root, 3): 因为只要 x>=o->v 就一直往右走, 所以会走到排名 4~5 的位置, 到达 nil 结点返回 1, 所以
↪ 最外层 rank 返回值为 5
*/
int main() {
    init();
    node *root = nil;
    int A[] = {1, 2, 3, 3, 4, 4, 5};
    for (unsigned i = 0; i < sizeof(A) / sizeof(*A); ++i)
        insert(root, A[i]);
    printf("%d\n", rank(root, 3));
    printf("%d\n", rank1(root, 3));
    printf("%d\n", rank2(root, 3));
    return 0;
}
```

9.7 link-cut-tree

```
const int maxn = 110000;
//若要修改一个点的点权, 应当先将其 splay 到根, 然后修改, 最后还要调用 pushup 维护。
namespace lct {
    int ch[maxn][2], fa[maxn], stk[maxn], rev[maxn];
    void init() { //初始化 link-cut-tree
        memset(ch, 0, sizeof(ch));
        memset(fa, 0, sizeof(fa));
        memset(rev, 0, sizeof(rev));
    }
    inline bool son(int x) {
        return ch[fa[x]][1] == x;
    }
```

```
inline bool isroot(int x) {
    return ch[fa[x]][1] != x && ch[fa[x]][0] != x;
}
inline void reverse(int x) { //给结点 x 打上反转标记
    swap(ch[x][1], ch[x][0]);
    rev[x] ^= 1;
}
inline void pushup(int x) { }
inline void pushdown(int x) {
    if (rev[x]) {
        reverse(ch[x][0]);
        reverse(ch[x][1]);
        rev[x] = 0;
    }
}
void rotate(int x) {
    int y = fa[x], z = fa[y], c = son(x);
    if (!isroot(y))
        ch[z][son(y)] = x;
    fa[x] = z;
    ch[y][c] = ch[x][!c];
    fa[ch[y][c]] = y;
    ch[x][!c] = y;
    fa[y] = x;
    pushup(y);
}
void splay(int x) {
    int top = 0;
    stk[++top] = x;
    for (int i = x; !isroot(i); i = fa[i])
        stk[++top] = fa[i];
    while (top)
        pushdown(stk[top--]);
    for (int y = fa[x]; !isroot(x); rotate(x), y = fa[x]) if (!isroot(y))
        son(x) ^ son(y) ? rotate(x) : rotate(y);
    pushup(x);
}
void access(int x) {
    for (int y = 0; x; y = x, x = fa[x]) {
        splay(x);
        ch[x][1] = y;
        pushup(x);
    }
}
void makeroot(int x) { //将 x 变为树的新的根结点
    access(x);
```

```
        splay(x);
        reverse(x);
    }
    int findroot(int x) { //返回 x 所在树的根结点
        access(x);
        splay(x);
        while (ch[x][0])
            pushdown(x), x = ch[x][0];
        return x;
    }
    void split(int x, int y) { //提取出来 y 到 x 之间的路径，并将 y 作为根结点
        makeroot(x);
        access(y);
        splay(y);
    }
    void cut(int x) { //断开结点 x 与它的父结点之间的边
        access(x);
        splay(x);
        ch[x][0] = fa[ch[x][0]] = 0;
        pushup(x);
    }
    void cut(int x, int y) { //切断 x 与 y 相连的边（必须保证 x 与 y 在一棵树中）
        makeroot(x); //将 x 置为整棵树的根
        cut(y); //删除 y 与其父结点之间的边
    }
    void link(int x, int y) { //连接 x 与 y（必须保证 x 和 y 属于不同的树）
        makeroot(x);
        fa[x] = y;
    }
    bool sametree(int x, int y) { //判断结点 x 与 y 是否属于同一棵树
        makeroot(x);
        return findroot(y) == x;
    }
}
int main() {
    return 0;
}
```

9.8 link-cut-tree（指针）

```
const int maxn = 400000;
const int inf = 1 << 30;
struct node {
    node* p, * ch[2];
    int mx, rev, val, add;
}nodes[maxn], * cur, * nil;
node* newnode(int key) {
```

```
    cur->p = cur->ch[0] = cur->ch[1] = nil;
    cur->mx = cur->val = key;
    cur->add = cur->rev = 0;
    return cur++;
}
void init() {
    cur = nodes;
    nil = newnode(-inf);
}
bool isroot(node* x) {
    return x->p == nil || x->p->ch[0] != x && x->p->ch[1] != x;
}
void increase(node* x, int v) {
    x->val += v;
    x->add += v;
    x->mx += v;
}
void pushup(node* x) {
    x->mx = x->val;
    if (x->ch[0] != nil)
        x->mx = max(x->mx, x->ch[0]->mx);
    if (x->ch[1] != nil)
        x->mx = max(x->mx, x->ch[1]->mx);
}
void pushdown(node* x) {
    if (x->rev) {
        x->rev = 0;
        if (x->ch[0] != nil) x->ch[0]->rev ^= 1;
        if (x->ch[1] != nil) x->ch[1]->rev ^= 1;
        swap(x->ch[0], x->ch[1]);
    }
    if (x->add) {
        if (x->ch[0] != nil)
            increase(x->ch[0], x->add);
        if (x->ch[1] != nil)
            increase(x->ch[1], x->add);
        x->add = 0;
    }
}
void rotate(node* x, int d) {
    if (isroot(x)) return;
    node* y = x->p;
    y->ch[!d] = x->ch[d];
    x->ch[d]->p = y;
    x->p = y->p;
    if (!isroot(y))
```

```
        y->p->ch[y == y->p->ch[1]] = x;
    x->ch[d] = y;
    y->p = x;
    pushup(y);
}
void splay(node* x) { //若要修改一个点的权值则要先将其伸展到根
    static node* sta[maxn];
    int top = 1;
    sta[0] = x;
    for (node* y = x; !isroot(y); y = y->p)
        sta[top++] = y->p;
    while (top) pushdown(sta[--top]);
    while (!isroot(x)) {
        node* y = x->p;
        if (isroot(y))
            rotate(x, x == y->ch[0]);
        else {
            int d = y == y->p->ch[0];
            if (x == y->ch[d])
                rotate(x, !d);
            else
                rotate(y, d);
            rotate(x, d);
        }
    }
    pushup(x);
}
node* access(node* x) { //打通结点 x 到根结点的路径
    node* y = nil;
    while (x != nil) {
        splay(x);
        y->p = x;
        x->ch[1] = y;
        pushup(x);
        y = x;
        x = x->p;
    }
    return y;
}
void changeroot(node* x) { //将 x 置为整棵树的根结点
    access(x)->rev ^= 1;
}
void link(node* x, node* y) { //将结点 x 与 y 连接起来（必须保证 x 与 y 属于不同的树）
    access(x);
    splay(x);
    x->rev ^= 1;
```

```
    x->p = y;
}
void cut(node* x) { //断开结点 x 与它的父结点之间的边
    access(x);
    splay(x);
    x->ch[0] = x->ch[0]->p = nil;
    pushup(x);
}
void cut(node* x, node* y) { //断开结点 x 与 y 之间的边
    changeroot(x); //将 x 置为整棵树的根
    cut(y); //删除 y 与其父结点之间的边
}
node* getroot(node* x) { //返回结点 x 所在的树当前的根结点
    access(x);
    splay(x);
    while (pushdown(x), x->ch[0] != nil)
        x = x->ch[0];
    splay(x);
    return x;
}
bool sametree(node* x, node* y) { //判断结点 x 与 y 是否属于同一棵树
    changeroot(x);
    return getroot(y) == x;
}
int n, m, x, y, w;
int eu[maxn], ev[maxn];
int main()
{
    //freopen("in.txt", "r", stdin);
    while (scanf("%d", &n) != -1)
    {
        init();
        for (int i = 1; i < n; i++)
            scanf("%d%d", &eu[i], &ev[i]);
        for (int i = 1; i <= n; i++)
        {
            int a;
            scanf("%d", &a);
            newnode(a);
        }
        for (int i = 1; i < n; i++)
            link(nodes + eu[i], nodes + ev[i]);
        scanf("%d", &m);
        for (int i = 1; i <= m; i++)
        {
            scanf("%d", &x);
```

```
if (x == 1)
{
    scanf("%d%d", &x, &y);
    if (sametree(nodes + x, nodes + y))
    {
        printf("-1\n");
        continue;
    }
    link(nodes + x, nodes + y);
}
else if (x == 2)
{
    scanf("%d%d", &x, &y);
    if (x == y || !sametree(nodes + x, nodes + y))
    {
        printf("-1\n");
        continue;
    }
    changeroot(nodes + x);
    cut(nodes + y);
}
else if (x == 3)
{
    scanf("%d%d%d", &w, &x, &y);
    if (!sametree(nodes + x, nodes + y))
    {
        printf("-1\n");
        continue;
    }
    changeroot(nodes + x);
    access(nodes + y);
    splay(nodes + y);
    node* q = nodes + y;
    increase(q, w);
}
else
{
    scanf("%d%d", &x, &y);
    if (!sametree(nodes + x, nodes + y))
    {
        printf("-1\n");
        continue;
    }
    changeroot(nodes + x);
    access(nodes + y);
    splay(nodes + y);
```

```
                printf("%d\n", (nodes + y)->mx);
            }
        }
        printf("\n");
    }
    return 0;
}
```

9.9 link-cut-tree（边权）

```
const int maxn = 400000;
const int inf = 1 << 30;
struct node { //mn 记录最小值, pos 记录最小值所在的结点
    node* p, *ch[2], *pos;
    int mn, rev, val;
}nodes[maxn], * cur, * nil;
pair<node*, node*> edge[maxn];
node* newnode(int key) {
    cur->p = cur->ch[0] = cur->ch[1] = nil;
    cur->mn = cur->val = key;
    cur->rev = 0;
    return cur++;
}
void init() {
    cur = nodes;
    nil = newnode(inf);
}
bool isroot(node* x) {
    return x->p == nil || x->p->ch[0] != x && x->p->ch[1] != x;
}
void pushup(node* x) {
    x->mn = x->val;
    x->pos = x;
    if (x->ch[0] != nil && x->ch[0]->mn < x->mn) {
        x->mn = x->ch[0]->mn;
        x->pos = x->ch[0]->pos;
    }
    if (x->ch[1] != nil && x->ch[1]->mn < x->mn) {
        x->mn = x->ch[1]->mn;
        x->pos = x->ch[1]->pos;
    }
}
void pushdown(node* x) {
    if (x->rev) {
        x->rev = 0;
        if (x->ch[0] != nil) x->ch[0]->rev ^= 1;
        if (x->ch[1] != nil) x->ch[1]->rev ^= 1;
```

```
        swap(x->ch[0], x->ch[1]);
    }
}
void rotate(node* x, int d) {
    if (isroot(x)) return;
    node* y = x->p;
    y->ch[!d] = x->ch[d];
    x->ch[d]->p = y;
    x->p = y->p;
    if (!isroot(y))
        y->p->ch[y == y->p->ch[1]] = x;
    x->ch[d] = y;
    y->p = x;
    pushup(y);
}
void splay(node* x) {
    static node* sta[maxn];
    int top = 1;
    sta[0] = x;
    for (node* y = x; !isroot(y); y = y->p)
        sta[top++] = y->p;
    while (top) pushdown(sta[--top]);
    while (!isroot(x)) {
        node* y = x->p;
        if (isroot(y))
            rotate(x, x == y->ch[0]);
        else {
            int d = y == y->p->ch[0];
            if (x == y->ch[d])
                rotate(x, !d);
            else
                rotate(y, d);
            rotate(x, d);
        }
    }
    pushup(x);
}
node* access(node* x) { //打通结点 x 到根结点的路径
    node* y = nil;
    while (x != nil) {
        splay(x);
        y->p = x;
        x->ch[1] = y;
        pushup(x);
        y = x;
        x = x->p;
```

```
    }
    return y;
}
void changeroot(node* x) { //将 x 置为整棵树的根结点
    access(x)->rev ^= 1;
}
void link(node* x, node* y) {
    access(x);
    splay(x);
    x->rev ^= 1;
    x->p = y;
}
node *link(node* x, node* y, int value) { //在结点 x 与结点 y 之间连接一条权值为 v 的边（必须保证 x
↪   与 y 属于不同的树）
    node* e = newnode(value);
    link(x, e);
    link(y, e);
    edge[e - nodes] = make_pair(x, y);
    return e; //返回新建的边对应的结点
}
void cut(node* x, node* y) { //断开结点 x 与 y 之间的边
    changeroot(x); //将 x 置为整棵树的根
    access(y);
    splay(y);
    y->ch[0] = y->ch[0]->p = nil;
    pushup(y);
}
void cut(node* x) { //删除结点 x 所代表的边
    pair<node*, node*> pr = edge[x - nodes];
    cut(pr.first, x);
    cut(pr.second, x);
    edge[x - nodes] = pair<node *, node*>(0, 0);
}
bool exist(node* x) { //检查点 x 所对应的边是否还存在于树中（没有被删除）
    return edge[x - nodes].first != 0;
}
node* getroot(node* x) { //返回结点 x 所在的树当前的根结点
    access(x);
    splay(x);
    while (pushdown(x), x->ch[0] != nil)
        x = x->ch[0];
    splay(x);
    return x;
}
bool sametree(node* x, node* y) { //判断结点 x 与 y 是否属于同一棵树
    changeroot(x);
```

```
    return getroot(y) == x;
}
int main() {
}
```

9.10 link-cut-tree（维护子树）

```
/*
用 link-cut-tree 维护子树信息：对于维护的信息，每个点要开两个属性，其中一个是在原树中的子树信息，
另一个是在 LCT 中虚子树的信息。除了要在 pushup 中维护这些信息外，还要在 access 和 link 函数中维护。
若要访问一个结点 x 的子树信息，应当先将其 access 到根，此时 x->size 表示 x 的子树中除去 x 外所有结点
↪  的信息（大小），
与 x 自身的信息结合即可得到 x 的子树信息（x->size + 1 即为子树大小）。
若要访问整棵树的信息，可以考虑结合 changeroot 函数，并使用结点的 sum 信息。
*/
const int maxn = 400000;
struct node {
    node* p, * ch[2];
    int rev, sum, size; //sum 维护当前点的虚子树与实子树大小之和，size 是在 LCT 中虚子树的大小
}nodes[maxn], * cur, * nil;
node* newnode() {
    cur->p = cur->ch[0] = cur->ch[1] = nil;
    cur->sum = 1;
    cur->size = 0;
    cur->rev = 0;
    return cur++;
}
void init() {
    cur = nodes;
    nil = newnode();
    nil->sum = 0;
}
bool isroot(node* x) {
    return x->p == nil || x->p->ch[0] != x && x->p->ch[1] != x;
}
void pushup(node* x) {
    x->sum = x->ch[0]->sum + x->ch[1]->sum + x->size + 1;
}
void pushdown(node* x) {
    if (x->rev) {
        x->rev = 0;
        if (x->ch[0] != nil) x->ch[0]->rev ^= 1;
        if (x->ch[1] != nil) x->ch[1]->rev ^= 1;
        swap(x->ch[0], x->ch[1]);
    }
}
void rotate(node* x, int d) {
```

```
    if (isroot(x)) return;
    node* y = x->p;
    y->ch[!d] = x->ch[d];
    x->ch[d]->p = y;
    x->p = y->p;
    if (!isroot(y))
        y->p->ch[y == y->p->ch[1]] = x;
    x->ch[d] = y;
    y->p = x;
    pushup(y);
}
void splay(node* x) { //若要修改一个点的权值则要先将其伸展到根
    static node* sta[maxn];
    int top = 1;
    sta[0] = x;
    for (node* y = x; !isroot(y); y = y->p)
        sta[top++] = y->p;
    while (top) pushdown(sta[--top]);
    while (!isroot(x)) {
        node* y = x->p;
        if (isroot(y))
            rotate(x, x == y->ch[0]);
        else {
            int d = y == y->p->ch[0];
            if (x == y->ch[d])
                rotate(x, !d);
            else
                rotate(y, d);
            rotate(x, d);
        }
    }
    pushup(x);
}
node* access(node* x) { //打通结点 x 到根结点的路径
    node* y = nil;
    while (x != nil) {
        splay(x);
        x->size += x->ch[1]->sum - y->sum; //动态维护虚子树信息
        y->p = x;
        x->ch[1] = y;
        pushup(x);
        y = x;
        x = x->p;
    }
    return y;
}
```

```
void changeroot(node* x) { //将 x 置为整棵树的根结点
    access(x)->rev ^= 1;
    splay(x); //将 x 伸展到 splay 的根结点，这样才能保证 link 的正确性
}
void link(node* x, node* y) { //将结点 x 与 y 连接起来（必须保证 x 与 y 属于不同的树且不为 nil 结点）
↪
    changeroot(x);
    changeroot(y);
    x->p = y;
    y->size += x->sum; //动态维护虚子树信息
    pushup(y);
}
void cut(node* x) { //断开结点 x 与它的父结点之间的边
    access(x);
    splay(x);
    x->ch[0] = x->ch[0]->p = nil;
    pushup(x);
}
void cut(node* x, node* y) { //断开结点 x 与 y 之间的边
    changeroot(x); //将 x 置为整棵树的根
    cut(y); //删除 y 与其父结点之间的边
}
node* getroot(node* x) { //返回结点 x 所在的树当前的根结点
    access(x);
    splay(x);
    while (pushdown(x), x->ch[0] != nil)
        x = x->ch[0];
    splay(x);
    return x;
}
bool sametree(node* x, node* y) { //判断结点 x 与 y 是否属于同一棵树
    changeroot(x);
    return getroot(y) == x;
}
node* nd[maxn];
int main() {
    //freopen("in.txt", "r", stdin);
    int n, q;
    scanf("%d %d", &n, &q);
    init();
    for (int i = 1; i <= n; ++i)
        nd[i] = newnode();
    while (q--) {
        char tp;
        int x, y;
        scanf(" %c %d %d", &tp, &x, &y);
```

```
        if (tp == 'A') {
            link(nd[x], nd[y]);
        }
        else {
            changeroot(nd[x]);
            access(nd[y]);
            long long L = nd[y]->size + 1; //将虚子树的大小加 1, 即为子树大小。
            changeroot(nd[y]);
            access(nd[x]);
            long long R = nd[x]->size + 1;
            printf("%lld\n", L * R);
        }
    }
    return 0;
}
```

9.11 可持久化 Treap

```
#define rank Rank
const int maxn = 510000;
namespace treap_old {
    //在每次更新标记的时候对被更新节点的属性进行维护。
    //在 pushdown 的时候不修改根节点的值。
    //在 pushup 的时候直接访问子树的属性, 而无需考虑 flip 等特殊标记。
    struct node {
        node* lch, * rch;
        int r;
        int v;
        int s;
        void up() {
            s = lch->s + rch->s + 1;
        }
        void down() {
            /*
            if (rev)
            {
                rev = 0;
                lch->rev ^= 1;
                rch->rev ^= 1;
                swap(lch, rch);
            }
            */
        }
    }nodes[maxn], * cur, * nil;
    node* newnode(int x) {
        node* o = cur++;
        o->lch = o->rch = nil;
```

```
        o->r = rand();
        o->v = x;
        o->s = 1;
        return o;
    }
    void init() {
        cur = nodes;
        nil = newnode(0);
        nil->s = 0;
    }
    node* copy(node* o) {
        *cur = *o;
        return cur++;
    }
    //如果有 down 操作则可持久化操作放在 down 中, down 中复制左右子树后进行标记下传
    void merge(node*& o, node* a, node* b) {
        if (a == nil) o = b;
        else if (b == nil) o = a;
        else if (a->r > b->r) {
            o = a; //o = copy(a);
            o->down();
            merge(o->rch, a->rch, b);
            //o->rch->fa = o;
            o->up();
        }
        else {
            o = b; //o = copy(b);
            o->down();
            merge(o->lch, a, b->lch);
            //o->lch->fa = o;
            o->up();
        }
    }
    void split(node* o, node*& a, node*& b, int k) {
        if (k == 0)
            a = nil, b = o;
        else if (o->s <= k)
            a = o, b = nil;
        else if (o->lch->s >= k) { //如果有序列的翻转操作则应当将 o->down() 放在该 if 语句之前, 以
↪   确保正确访问左子树的大小
            o->down();
            b = o; //b = copy(o);
            split(o->lch, a, b->lch, k);
            //a->fa = nil;
            //b->lch->fa = b;
            b->up();
```

```
        }
        else {
            o->down();
            a = o; //a = copy(o);
            split(o->rch, a->rch, b, k - o->lch->s - 1);
            //a->rch->fa = a;
            //b->fa = nil;
            a->up();
        }
    }
    bool find(node* o, int x) {
        while (o != nil) {
            if (x == o->v)
                return true;
            else if (x < o->v)
                o = o->lch;
            else
                o = o->rch;
        }
        return false;
    }
    int rank(node* o, int x) {
        if (o == nil)
            return 1;
        if (x <= o->v)
            return rank(o->lch, x);
        else return rank(o->rch, x) + o->lch->s + 1;
    }
    int kth(node* o, int k) {
        if (o == nil || k <= 0 || k > o->s) return 0;
        if (k == o->lch->s + 1) return o->v;
        else if (k <= o->lch->s) return kth(o->lch, k);
        else return kth(o->rch, k - o->lch->s - 1);
    }
    //insert(root, rank(root, x), x);
    void insert(node*& o, int k, int x) {
        node* a, * b;
        split(o, a, b, k - 1);
        merge(a, a, newnode(x));
        merge(o, a, b);
    }
    //del(root, rank(root, x));
    void del(node*& o, int k) {
        node* a, * b, * c;
        split(o, a, b, k - 1);
        split(b, b, c, 1);
```

```
        merge(o, a, c);
        //freenode(b);
    }
    //node *root = nil; //创建一棵空树
}
namespace treap {
    struct node {
        node* lch, * rch;
        int v;
        int s;
        void up() {
            s = lch->s + rch->s + 1;
        }
    }nodes[maxn], * cur, * nil;
    node* newnode(int v) {
        node* x = cur++;
        x->lch = x->rch = nil;
        x->v = v;
        x->s = 1;
        return x;
    }
    void init() {
        cur = nodes;
        nil = newnode(0);
        nil->s = 0;
    }
    node* copy(node* x) {
        *cur = *x;
        return cur++;
    }
    node* merge(node* a, node* b) {
        if (a == nil) return b;
        if (b == nil) return a;
        if (rand() % (a->s + b->s) < a->s) { // 必须保证 rand 产生的随机数足够大
            node* x = copy(a);
            x->rch = merge(a->rch, b);
            x->up();
            return x;
        }
        else {
            node* x = copy(b);
            x->lch = merge(a, b->lch);
            x->up();
            return x;
        }
    }
```

```
node* merge(node* a, node* b, node* c) {
    return merge(a, merge(b, c));
}
pair<node*, node*> split(node* x, int k) {
    if (k == 0)
        return { nil, x };
    if (x->s <= k)
        return { x, nil };
    if (x->lch->s >= k) {
        node* b = copy(x);
        auto [a, m] = split(x->lch, k);
        b->lch = m;
        b->up();
        return { a, b };
    }
    else {
        node* a = copy(x);
        auto [m, b] = split(x->rch, k - x->lch->s - 1);
        a->rch = m;
        a->up();
        return { a, b };
    }
}
tuple<node*, node*, node*> split(node* x, int L, int R) {
    auto [a, m] = split(x, L - 1);
    auto [b, c] = split(m, R - L + 1);
    return { a, b, c };
}
bool find(node* x, int v) {
    while (x != nil) {
        if (v == x->v)
            return true;
        else if (v < x->v)
            x = x->lch;
        else
            x = x->rch;
    }
    return false;
}
int rank(node* x, int v) { //v 在 x 中是第几大的值
    if (x == nil)
        return 1;
    if (v <= x->v)
        return rank(x->lch, v);
    else
        return rank(x->rch, v) + x->lch->s + 1;
```

```
    }
    int kth(node* x, int k) {
        if (x == nil || k <= 0 || k > x->s) return 0;
        if (k == x->lch->s + 1) return x->v;
        else if (k <= x->lch->s) return kth(x->lch, k);
        else return kth(x->rch, k - x->lch->s - 1);
    }
    //insert(root, rank(root, x), x);
    [[nodiscard]] node* insert(node* x, int k, int v) {
        auto [a, b] = split(x, k - 1);
        return merge(a, newnode(v), b);
    }
    //del(root, rank(root, x));
    [[nodiscard]] node* del(node* x, int k) {
        auto [a, m] = split(x, k - 1);
        auto [v, b] = split(m, 1);
        return merge(a, b);
    }
    //node *root = nil; //创建一棵空树
}
namespace shared_treap { //用智能指针优化可持久化 treap 的内存
    struct node;
    struct pointer {
        node* ptr;
        pointer() : ptr(nullptr) {}
        pointer(node* ptr);
        pointer(const pointer& ptr);
        pointer& operator= (const pointer& rhs);
        operator node* () const;
        node* operator-> ();
        node& operator* ();
        ~pointer();
        void del();
    };
    struct node {
        pointer lch, rch;
        int v, s, ref;
        void up();
    }nodes[maxn];
    node* const nil = nodes;
    vector<node*> pool;
    inline pointer::pointer(const pointer& ptr) : pointer(ptr.ptr) {}
    inline pointer::pointer(node* ptr) : ptr(ptr) {
        if (ptr)
            ptr->ref += 1;
    }
```

```
inline pointer& pointer::operator= (const pointer& rhs) {
    if (ptr != rhs.ptr) {
        del();
        ptr = rhs.ptr;
        if (ptr)
            ptr->ref += 1;
    }
    return *this;
}
inline node* pointer::operator-> () {
    return ptr;
}
inline pointer::operator node* () const {
    return ptr;
}
inline void pointer::del() {
    if (ptr) {
        ptr->ref -= 1;
        if (ptr->ref == 0) {
            ptr->lch = ptr->rch = nullptr;
            pool.push_back(ptr);
        }
    }
}
inline pointer::~pointer() {
    del();
}
inline void node::up() {
    s = lch->s + rch->s + 1;
}
void init() {
    pool.clear();
    for (int i = 1; i < maxn; ++i)
        pool.push_back(nodes + i);
}
pointer newnode(int v) {
    node* x = pool.back(); //类型是 node* !!!!!
    pool.pop_back();
    x->lch = x->rch = nil;
    x->v = v;
    x->s = 1;
    x->ref = 0;
    return x;
}
inline node* copy(node* x) {
    if (x == nil) return x;
```

```
        node* y = pool.back(); pool.pop_back();
        *y = *x;
        y->ref = 0;
        return y;
    }
    pointer merge(node* a, node* b) {
        if (a == nil) return b;
        if (b == nil) return a;
        if (rand() % (a->s + b->s) < a->s) { // 必须保证 rand 产生的随机数足够大
            auto x = copy(a);
            x->rch = merge(a->rch, b);
            x->up();
            return x;
        }
        else {
            auto x = copy(b);
            x->lch = merge(a, b->lch);
            x->up();
            return x;
        }
    }
    pointer merge(node* a, node* b, node* c) {
        return merge(a, merge(b, c));
    }
    pair<pointer, pointer> split(node* x, int k) {
        if (k == 0)
            return { nil, x };
        if (x->s <= k)
            return { x, nil };
        if (x->lch->s >= k) {
            pointer b = copy(x);
            auto [a, m] = split(x->lch, k);
            b->lch = m;
            b->up();
            return { a, b };
        }
        else {
            pointer a = copy(x);
            auto [m, b] = split(x->rch, k - x->lch->s - 1);
            a->rch = m;
            a->up();
            return { a, b };
        }
    }
    tuple<pointer, pointer, pointer> split(node* x, int L, int R) {
        auto [a, m] = split(x, L - 1);
```

```
        auto [b, c] = split(m, R - L + 1);
        return { a, b, c };
    }
    bool find(node* x, int v) {
        while (x) {
            if (v == x->v)
                return true;
            else if (v < x->v)
                x = x->lch;
            else
                x = x->rch;
        }
        return false;
    }
    int rank(node* x, int v) { //v 在 x 中是第几大的值
        if (x == nil)
            return 1;
        if (v <= x->v)
            return rank(x->lch, v);
        else
            return rank(x->rch, v) + x->lch->s + 1;
    }
    int kth(node* x, int k) {
        if (x == nil || k <= 0 || k > x->s) return 0;
        if (k == x->lch->s + 1) return x->v;
        else if (k <= x->lch->s) return kth(x->lch, k);
        else return kth(x->rch, k - x->lch->s - 1);
    }
    //insert(root, rank(root, x), x);
    [[nodiscard]] pointer insert(node* x, int k, int v) {
        auto [a, b] = split(x, k - 1);
        return merge(a, newnode(v), b);
    }
    //del(root, rank(root, x));
    [[nodiscard]] pointer del(node* x, int k) {
        auto [a, m] = split(x, k - 1);
        auto [v, b] = split(m, 1);
        return merge(a, b);
    }
    //pointer root = nil; //创建一棵空树
}
namespace shared_treap_pushdown {
    struct node;
    struct pointer {
        node* ptr;
        pointer() : ptr(nullptr) {}
```

```
    pointer(node* ptr);
    pointer(const pointer& ptr);
    pointer& operator= (const pointer& rhs);
    operator node* () const;
    node* operator-> ();
    node& operator* ();
    ~pointer();
    void del();
};
struct node {
    pointer lch, rch;
    int s, ref;
    int val, add;
    long long sum;
    void up();
    void down();
    void mark(int tag) { //在 mark 之前要先 copy 当前结点!!!
        add += tag;
        val += tag;
        sum += 1LL * tag * s;
    }
}nodes[maxn];
node* const nil = nodes;
vector<node*> pool;
inline pointer::pointer(const pointer& ptr) : pointer(ptr.ptr) {}
inline pointer::pointer(node* ptr) : ptr(ptr) {
    if (ptr)
        ptr->ref += 1;
}
inline pointer& pointer::operator= (const pointer& rhs) {
    if (ptr != rhs.ptr) {
        del();
        ptr = rhs.ptr;
        if (ptr)
            ptr->ref += 1;
    }
    return *this;
}
inline node* pointer::operator-> () {
    return ptr;
}
inline pointer::operator node* () const {
    return ptr;
}
inline void pointer::del() {
    if (ptr) {
```

```
            ptr->ref -= 1;
            if (ptr->ref == 0) {
                ptr->lch = ptr->rch = nullptr;
                pool.push_back(ptr);
            }
        }
    }
    inline pointer::~pointer() {
        del();
    }
    void init() {
        pool.clear();
        for (int i = 1; i < maxn; ++i)
            pool.push_back(nodes + i);
    }
    pointer newnode(int v) {
        node* x = pool.back(); //类型是 node* !!!!!
        pool.pop_back();
        x->lch = x->rch = nil;
        x->val = x->sum = v;
        x->add = 0;
        x->s = 1;
        x->ref = 0;
        return x;
    }
    inline node* copy(node* x) {
        if (x == nil) return x;
        node* y = pool.back(); pool.pop_back();
        *y = *x;
        y->ref = 0;
        return y;
    }
    pointer merge(node* a, node* b) {
        if (a == nil) return b;
        if (b == nil) return a;
        if (rand() % (a->s + b->s) < a->s) { // 必须保证 rand 产生的随机数足够大
            auto x = copy(a);
            x->down();
            x->rch = merge(x->rch, b);
            x->up();
            return x;
        }
        else {
            auto x = copy(b);
            x->down();
            x->lch = merge(a, x->lch);
```

```
        x->up();
        return x;
    }
}
inline pointer merge(node* a, node* b, node* c) {
    return merge(a, merge(b, c));
}
pair<pointer, pointer> split(node* x, int k) {
    if (k == 0)
        return { nil, x };
    if (x->s <= k)
        return { x, nil };
    x = copy(x);
    x->down();
    if (x->lch->s >= k) {
        auto [a, m] = split(x->lch, k);
        x->lch = m;
        x->up();
        return { a, x };
    }
    else {
        auto [m, b] = split(x->rch, k - x->lch->s - 1);
        x->rch = m;
        x->up();
        return { x, b };
    }
}
inline tuple<pointer, pointer, pointer> split(node* x, int L, int R) {
    auto [a, m] = split(x, L - 1);
    auto [b, c] = split(m, R - L + 1);
    return { a, b, c };
}
inline void node::up() {
    s = lch->s + rch->s + 1;
    sum = lch->sum + rch->sum + val;
}
inline void node::down() {
    if (add) {
        if (lch.ptr != nil) {
            lch = copy(lch.ptr);
            lch->mark(add);
        }
        if (rch.ptr != nil) {
            rch = copy(rch.ptr);
            rch->mark(add);
        }
```

```
            add = 0;
        }
    }
}
int main() {
    using namespace shared_treap_pushdown;
    init();
    pointer root = nil;
    return 0;
}
```

9.12 树链剖分

```
/*
1.    如果权值在边上的话，则先将权值压到深度大的点上，再建立线段树。
2.    结点 x 的子树对应的 DFS 序区间为 [id[x], id[x] + siz[x] - 1]。
*/
#define lc t[p].lchild
#define rc t[p].rchild
const int maxn = 110000;
int cur, root, dfn, dep[maxn], siz[maxn], pa[maxn], id[maxn], son[maxn], top[maxn];
int val[maxn], A[maxn], ans;
vector<int> G[maxn];
struct segment {
    int l, r, lchild, rchild;
    int sum, add;
}t[maxn * 4];
inline void maintain(int p) {
    t[p].sum = t[lc].sum + t[rc].sum;
}
inline void mark(int p, int addv) { //给结点打标记
    if (addv) {
        t[p].add += addv;
        t[p].sum += addv * (t[p].r - t[p].l + 1);
    }
}
inline void pushdown(int p) { //pushdown 将标记传递给子结点，不影响当前结点的信息。
    mark(lc, t[p].add);
    mark(rc, t[p].add);
    t[p].add = 0;
}
int build(int L, int R) {
    int p = ++cur;
    t[p].l = L;
    t[p].r = R;
    t[p].add = 0;
    if (L == R) {
```

```
        mark(p, val[L]);
    }
    else {
        int mid = (L + R) >> 1;
        lc = build(L, mid);
        rc = build(mid + 1, R);
        maintain(p);
    }
    return p;
}
void update(int p, int L, int R, int v) {
    if (L <= t[p].l && R >= t[p].r) {
        mark(p, v);
    }
    else {
        pushdown(p); //如果没有 pushdown 只需要在最后调用一次 maintain 即可。
        int mid = (t[p].l + t[p].r) >> 1;
        if (L <= mid)
            update(lc, L, R, v);
        if (R > mid)
            update(rc, L, R, v);
        maintain(p);
    }
}
void query(int p, int L, int R) {
    if (L <= t[p].l && R >= t[p].r) {
        ans += t[p].sum;
    }
    else {
        pushdown(p);
        int mid = (t[p].l + t[p].r) >> 1;
        if (L <= mid)
            query(lc, L, R);
        if (R > mid)
            query(rc, L, R);
    }
}
void dfs1(int x, int fa, int d) {
    dep[x] = d;
    siz[x] = 1;
    son[x] = 0;
    pa[x] = fa;
    for (auto y : G[x]) if (y != fa) {
        dfs1(y, x, d + 1);
        siz[x] += siz[y];
        if (siz[son[x]] < siz[y])
```

```
            son[x] = y;
    }
}
void dfs2(int x, int tp) {
    top[x] = tp;
    id[x] = ++dfn; //id[x] 表示结点 x 的 DFS 序编号
    if (son[x])
        dfs2(son[x], tp);
    for (auto y : G[x])
        if (y != pa[x] && y != son[x])
            dfs2(y, y);
}
int ask(int x, int y) {
    ::ans = 0;
    while (top[x] != top[y]) {
        if (dep[top[x]] < dep[top[y]])
            swap(x, y);
        query(root, id[top[x]], id[x]);
        x = pa[top[x]];
    }
    //如果权值在边上则加上: if (x == y) return ans;
    if (dep[x] > dep[y]) swap(x, y);
    query(root, id[x], id[y]); //如果权值在边上则查询 id[son[x]]
    return ans;
}
void add(int x, int y, int v) {
    while (top[x] != top[y]) {
        if (dep[top[x]] < dep[top[y]])
            swap(x, y);
        update(root, id[top[x]], id[x], v);
        x = pa[top[x]];
    }
    //如果权值在边上则加上: if (x == y) return ans;
    if (dep[x] > dep[y]) swap(x, y);
    update(root, id[x], id[y], v); //如果权值在边上则改为 id[son[x]]
}
void init(int n) { //调用该函数之前应当先完成建图
    cur = dfn = siz[0] = 0;
    dfs1(1, -1, 1); //根结点为 s 的话, 改为: dfs1(s, -1, 1), dfs2(s, s);
    dfs2(1, 1);
    for (int i = 1; i <= n; ++i)
        val[id[i]] = A[i]; //A[i] 表示结点 i 的权值, 将其复制到 DFS 序上。
    ::root = build(1, n);
}
int main() {
    return 0;
```

```
}
```

9.13 树链剖分求 LCA 和距离

```
const int maxn = 510000;
const int inf = 1 << 30;
int dep[maxn], sz[maxn], pa[maxn], son[maxn], top[maxn];
vector<int> G[maxn];
void dfs1(int u, int fa, int d) {
    dep[u] = d;
    sz[u] = 1;
    son[u] = 0;
    pa[u] = fa;
    for (auto v : G[u]) if (v != fa) {
        dfs1(v, u, d + 1);
        sz[u] += sz[v];
        if (sz[son[u]] < sz[v])
            son[u] = v;
    }
}
void dfs2(int u, int tp) {
    top[u] = tp;
    if (son[u])
        dfs2(son[u], tp);
    for (auto v : G[u]) if (v != pa[u] && v != son[u])
        dfs2(v, v);
}
int lca(int x, int y) {
    while (top[x] != top[y]) {
        if (dep[top[x]] < dep[top[y]])
            swap(x, y);
        x = pa[top[x]];
    }
    return dep[x] < dep[y] ? x : y;
}
int dist(int x, int y) {
    return dep[x] + dep[y] - 2 * dep[lca(x, y)];
}
void init(int root) {
    dfs1(root, 0, 1);
    dfs2(root, root);
}
int main() {
    int n, m, s;
    scanf("%d %d %d", &n, &m, &s);
    for (int i = 0; i < n - 1; ++i) {
        int x, y;
```

```
        scanf("%d %d", &x, &y);
        G[x].push_back(y);
        G[y].push_back(x);
    }
    init(s);
    while (m--) {
        int x, y;
        scanf("%d %d", &x, &y);
        printf("%d\n", dist(x, y));
    }
    return 0;
}
```

9.14 树状数组

```
#define lowbit(x) ((x)&(-x))
namespace Fenwick_Tree
{
    const int maxn = 110000;
    int C[maxn], n;
    void add(int x, int d)
    {
        for (int i = x; i <= n; i += lowbit(i))
            C[i] += d;
    }
    int sum(int x)
    {
        int ret = 0;
        for (int i = x; i > 0; i -= lowbit(i))
            ret += C[i];
        return ret;
    }
}
namespace Fenwick_Tree_2D
{
    const int maxn = 1100;
    int C[maxn][maxn], n, m;
    void add(int x, int y, int d)
    {
        for (int i = x; i <= n; i += lowbit(i))
            for (int j = y; j <= m; j += lowbit(j))
                C[i][j] += d;
    }
    int sum(int x, int y)
    {
        int ret = 0;
        for (int i = x; i > 0; i -= lowbit(i))
```

```
            for (int j = y; j > 0; j -= lowbit(j))
                ret += C[i][j];
        return ret;
    }
}
int main()
{
    return 0;
}
```

9.15 李超线段树

```
const int maxn = 200005;
const int inf = 1 << 30;
int cur = 0;
struct segment {
    int l, r, lc, rc;
    double k, b;
}t[maxn * 4];
void init() {
    cur = 0;
}
int build(int L, int R) { //新建一棵线段树，并返回根结点的编号
    int p = ++cur;
    t[p].l = L;
    t[p].r = R;
    if (L < R) {
        int mid = (L + R) >> 1;
        t[p].lc = build(L, mid);
        t[p].rc = build(mid + 1, R);
    }
    t[p].k = 0; t[p].b = -inf;
    return p;
}
void pushdown(int p, double k, double b) {
    double l1 = k * t[p].l + b, r1 = k * t[p].r + b;
    double l2 = t[p].k * t[p].l + t[p].b, r2 = t[p].k * t[p].r + t[p].b;
    if (l1 >= l2 && r1 >= r2)
        t[p].k = k, t[p].b = b;
    else if (l2 < l1 || r2 < r1) {
        double pos = (b - t[p].b) / (t[p].k - k);
        int mid = (t[p].l + t[p].r) >> 1;
        if (pos <= mid) {
            if (r1 > r2)
                swap(t[p].k, k), swap(t[p].b, b);
            pushdown(t[p].lc, k, b);
        }
```

```
        else {
            if (l1 > l2)
                swap(t[p].k, k), swap(t[p].b, b);
            pushdown(t[p].rc, k, b);
        }
    }
}
void insert(int p, int L, int R, double k, double b) { //在区间 [L, R] 中插入直线 y = kx + b
    if (L <= t[p].l && R >= t[p].r)
        pushdown(p, k, b);
    else {
        int mid = (t[p].l + t[p].r) >> 1;
        if (L <= mid)
            insert(t[p].lc, L, R, k, b);
        if (R > mid)
            insert(t[p].rc, L, R, k, b);
    }
}
double query(int p, int x) { //p 是线段树的根结点，x 是查询的横坐标，返回所有直线在 x 处的最大值
    double ans = t[p].k * x + t[p].b;
    if (t[p].l < t[p].r) {
        int mid = (t[p].l + t[p].r) >> 1;
        if (x <= mid)
            ans = max(ans, query(t[p].lc, x));
        else
            ans = max(ans, query(t[p].rc, x));
    }
    return ans;
}
const double eps = 1e-9;
struct Seg {
    Seg() : k(), b(), id(1) {}
    Seg(double k, double b) : k(k), b(b) {}
    double k, b;
    int id;
}A[maxn];
int main() {
    srand(time(0));
    int n = 200000, m = 300;
    for (int i = 1; i <= m; ++i)
        A[i].k = rand() % 10 + 1, A[i].b = rand() % 1000 - 500, A[i].id = i;
    init();
    int root = build(1, n);
    for (int i = 1; i <= m; ++i)
        insert(root, 1, n, A[i].k, A[i].b);
    for (int i = 1; i <= n; ++i) {
```

```
        long long ans = query(root, i) + eps;
        long long res = -inf;
        for (int j = 1; j <= m; ++j)
            res = max(res, (long long)(A[j].k * i + A[j].b + eps));
        if (res != ans)
            printf("%lld %lld\n", ans, res);
    }
    return 0;
}
```

9.16 整数集合

```
const int inf = 1 << 30;
static const int maxn = 210000; //maxn 应当大于所有集合大小的总和
struct node {        //该算法实现的整数集合为可重集合（multiset）
    int l, r, v;    //左节点编号 右节点编号 值
} T[maxn * 25];
int len, sz, info[maxn]; //len 是值域线段树的长度
int ranking(int v) {
    //如果不需要数据离散化直接 return v 即可，同时还要将所有的 info[v] 改为 v，此时线段树的值域为
    ↪ [1, len]。
    return lower_bound(info + 1, info + len + 1, v) - info;
}
void ins(int& i, int l, int r, int p) { //在区间 [l, r] 中插入 p
    if (!i) {
        i = ++sz;
        T[i].v = T[i].l = T[i].r = 0;
    }
    int m = (l + r) / 2;
    T[i].v++; //递增当前区间内值的个数
    if (l == r) return;
    if (p <= m) ins(T[i].l, l, m, p);
    else ins(T[i].r, m + 1, r, p);
}
void insert(int& x, int v) { //向集合 x 中添加数值 v
    ins(x, 1, len, ranking(v));
}
void init(int* A, int length) { //输入数组 A 的下标从 1 开始，length 表示 A 的长度
    sz = 0;                          //A 记录集合中所有可能出现的整数
    copy(A + 1, A + length + 1, info + 1);
    sort(info + 1, info + length + 1); //如果没有离散化需要对 len 进行赋值来指明线段树的值域。
    len = unique(info + 1, info + length + 1) - info - 1; //对 info 数组进行排序去重
}
int kth(int x, int k) { //查找集合 x 中的第 k 大元素
    int l = 1, r = len;
    while (l < r) {
        int m = (l + r) / 2, t = T[T[x].l].v;
```

```
        if (k <= t)
            x = T[x].l, r = m;
        else
            x = T[x].r, l = m + 1, k -= t;
    }
    if (k > T[x].v) //如果 k 大于集合 x 的大小，则返回 inf
        return inf;
    return info[r];
}
int ask(int x, int v) { //返回集合 x 中比 v 小的数的个数
    int l = 1, r = len, k = 0;
    int p = ranking(v) - 1;
    if (p <= 0) return 0;
    while (l < r) {
        int m = (l + r) / 2, t = T[T[x].l].v;
        if (p <= m)
            x = T[x].l, r = m;
        else
            x = T[x].r, l = m + 1, k += t;
    }
    k += T[x].v;
    return k;
}
int pre(int x, int v) { //在集合 x 中查询 v 的前驱，找不到返回-inf
    int k = ask(x, v);
    if (k == 0) return -inf;
    return kth(x, k);
}
int next(int x, int v) { //在集合 x 中查询 v 的后继，找不到返回 inf
    int k = ask(x, v + 1) + 1;
    return kth(x, k);
}
int count(int x, int v) { //返回集合 x 中数值 v 的个数
    int l = 1, r = len;
    int p = ranking(v);
    if (p > len || info[p] != v)
        return 0;
    while (l < r && T[x].v> 0) {
        int m = (l + r) / 2;
        if (p <= m)
            x = T[x].l, r = m;
        else
            x = T[x].r, l = m + 1;
    }
    return T[x].v;
}
```

```
bool find(int x, int v) { //返回集合 x 中是否存在 v
    return count(x, v) >= 1;
}
int maximum(int x) { //返回集合 x 中的最大值
    int l = 1, r = len;
    if (T[x].v == 0) //如果集合为空则返回-inf
        return -inf;
    while (l < r) {
        int m = (l + r) / 2, t = T[T[x].r].v;
        if (t == 0)
            x = T[x].l, r = m;
        else
            x = T[x].r, l = m + 1;
    }
    return info[r];
}
int minimum(int x) { //返回集合 x 中的最小值
    int l = 1, r = len;
    if (T[x].v == 0) //如果集合为空则返回 inf
        return inf;
    while (l < r) {
        int m = (l + r) / 2, t = T[T[x].l].v;
        if (t > 0)
            x = T[x].l, r = m;
        else
            x = T[x].r, l = m + 1;
    }
    return info[r];
}
int merge(int x, int y, int l = 1, int r = len) {    //将集合 x 与集合 y 合并成一个新的集合后返回
    if (!x) return y;                                //合并之后 x、y 均会失效
    if (!y) return x;
    if (l == r) {
        T[x].v += T[y].v;
    }
    else {
        int m = (l + r) / 2;
        T[x].l = merge(T[x].l, T[y].l, l, m);
        T[x].r = merge(T[x].r, T[y].r, m + 1, r);
        T[x].v = T[T[x].l].v + T[T[x].r].v;
    }
    return x;
}
int diff(int x, int y, int l = 1, int r = len) {    //返回集合 x 与集合 y 的差集（x - y），执行之
↪   后 x、y 均会失效
    if (l == r) {                                    //时间复杂度 size(y)log(len)
```

```
        T[x].v = max(0, T[x].v - T[y].v);
    }
    else {
        int m = (l + r) / 2;
        if (T[y].l)
            T[x].l = diff(T[x].l, T[y].l, l, m);
        if (T[y].r)
            T[x].r = diff(T[x].r, T[y].r, m + 1, r);
        T[x].v = T[T[x].l].v + T[T[x].r].v;
    }
    return x;
}
int intersect(int x, int y, int l = 1, int r = len) {    //返回集合 x 与集合 y 的交集，执行之后 x、
↪ y 均会失效
    if (l == r) {                                      //时间复杂度 min{size(x),size(y)} * log(len)
        T[x].v = min(T[x].v, T[y].v);
    }
    else {
        int m = (l + r) / 2;
        T[x].l = (!T[x].l || !T[y].l ? 0 : intersect(T[x].l, T[y].l, l, m));
        T[x].r = (!T[x].r || !T[y].r ? 0 : intersect(T[x].r, T[y].r, m + 1, r));
        T[x].v = T[T[x].l].v + T[T[x].r].v;
    }
    return T[x].v == 0 ? 0 : x;
}
int symmetric(int x, int y, int l = 1, int r = len) {    //返回集合 x 与集合 y 的对称差分，执行之
↪ 后 x、y 均会失效
    if (!x) return y;                                   //时间复杂度 min{size(x),size(y)} *
↪ log(len)
    if (!y) return x;
    if (l == r) {
        T[x].v = abs(T[x].v - T[y].v);
    }
    else {
        int m = (l + r) / 2;
        T[x].l = symmetric(T[x].l, T[y].l, l, m);
        T[x].r = symmetric(T[x].r, T[y].r, m + 1, r);
        T[x].v = T[T[x].l].v + T[T[x].r].v;
    }
    return x;
}
void left(vector<int>& X, vector<int>& Y) {
    for (auto& i : X)
        i = T[i].l;
    for (auto& i : Y)
        i = T[i].l;
```

```
}
void right(vector<int>& X, vector<int>& Y) {
    for (auto& i : X)
        i = T[i].r;
    for (auto& i : Y)
        i = T[i].r;
}
int left_value(vector<int>& X, vector<int>& Y) {
    int tot = 0;
    for (auto i : X)
        tot += T[T[i].l].v;
    for (auto i : Y)
        tot -= T[T[i].l].v;
    return tot;
}
int right_value(vector<int>& X, vector<int>& Y) {
    int tot = 0;
    for (auto i : X)
        tot += T[T[i].r].v;
    for (auto i : Y)
        tot -= T[T[i].r].v;
    return tot;
}
int value(vector<int>& X, vector<int>& Y) {
    int tot = 0;
    for (auto i : X)
        tot += T[i].v;
    for (auto i : Y)
        tot -= T[i].v;
    return tot;
}
int kth(vector<int> X, vector<int> Y, int k) {    //计算 X 中集合的并集 减去 Y 中集合的并集 所得集
↪   合的第 k 大
    int l = 1, r = len;                            //必须保证 Y 的并集 是 X 的并集 的子集
    while (l < r) {
        int m = (l + r) / 2, t = left_value(X, Y);
        if (k <= t)
            left(X, Y), r = m;
        else
            right(X, Y), l = m + 1, k -= t;
    }
    if (k > value(X, Y))
        return inf;
    return info[r];
}
```

```cpp
int ask(vector<int> X, vector<int> Y, int v) {    //计算 X 中集合的并集 减去 Y 中集合的并集 所得集合中比 v 小的数的个数
    int l = 1, r = len, k = 0;                    //必须保证 Y 的并集 是 X 的并集 的子集
    int p = ranking(v) - 1;
    if (p <= 0) return 0;
    while (l < r) {
        int m = (l + r) / 2;
        if (p <= m)
            left(X, Y), r = m;
        else {
            k += left_value(X, Y);
            right(X, Y);
            l = m + 1;
        }
    }
    k += value(X, Y);
    return k;
}
int pre(vector<int> X, vector<int> Y, int v) {    //计算 X 中集合的并集 减去 Y 中集合的并集 所得集合中 v 的前驱
    int k = ask(X, Y, v);                        //必须保证 Y 的并集 是 X 的并集 的子集
    if (k == 0) return -inf; //找不到返回-inf
    return kth(X, Y, k);
}
int next(vector<int> X, vector<int> Y, int v) {    //计算 X 中集合的并集 减去 Y 中集合的并集 所得集合中 v 的后继
    int k = ask(X, Y, v + 1) + 1;                //必须保证 Y 的并集 是 X 的并集 的子集
    return kth(X, Y, k); //找不到返回 inf
}
int count(vector<int> X, vector<int> Y, int v) {    //计算 X 中集合的并集 减去 Y 中集合的并集 所得集合中 v 的个数
    int l = 1, r = len;                                //必须保证 Y 的并集 是 X 的并集 的子集
    int p = ranking(v);
    if (p > len || info[p] != v)
        return 0;
    while (l < r) {
        int m = (l + r) / 2;
        if (p <= m)
            left(X, Y), r = m;
        else
            right(X, Y), l = m + 1;
    }
    return value(X, Y);
}
int maximum(vector<int> X, vector<int> Y) {    //计算 X 中集合的并集 减去 Y 中集合的并集 所得集合中的最大值
```

```
    int l = 1, r = len;                      //必须保证 Y 的并集 是 X 的并集 的子集
    if (value(X, Y) == 0) //如果集合为空则返回-inf
        return -inf;
    while (l < r) {
        int m = (l + r) / 2, t = right_value(X, Y);
        if (t == 0)
            left(X, Y), r = m;
        else
            right(X, Y), l = m + 1;
    }
    return info[r];
}
int minimum(vector<int> X, vector<int> Y) {    //计算 X 中集合的并集 减去 Y 中集合的并集 所得集合中
↪   的最小值
    int l = 1, r = len;                      //必须保证 Y 的并集 是 X 的并集 的子集
    if (value(X, Y) == 0) //如果集合为空则返回 inf
        return inf;
    while (l < r) {
        int m = (l + r) / 2, t = left_value(X, Y);
        if (t > 0)
            left(X, Y), r = m;
        else
            right(X, Y), l = m + 1;
    }
    return info[r];
}
#define VALUE_TYPE 100
int tmp[maxn], root[maxn];
vector<int> A[maxn], B[maxn], C[maxn];
int main() {
    int n = 10, q = 400000;
    default_random_engine e;
    uniform_int_distribution<int> d(1, VALUE_TYPE);
    uniform_int_distribution<int> big(-100000000, 100000000);
    e.seed(time(0));
    for (int i = 1; i <= VALUE_TYPE; ++i)
        tmp[i] = big(e);
    init(tmp, VALUE_TYPE);
    int rt = 0;
    vector<int> vec;
    for (int i = 1; i <= 100000; ++i) {
        int val = d(e); val = tmp[val];
        if (rand() % 2)
            insert(root[0], val), A[0].push_back(val);
        else
            insert(root[1], val), A[1].push_back(val);
```

```cpp
    }
    rt = merge(root[0], root[1]);
    sort(A[0].begin(), A[0].end());
    sort(A[1].begin(), A[1].end());
    //std::set_union(A[0].begin(), A[0].end(), A[1].begin(), A[1].end(), back_inserter(vec));
    vec = A[0]; for (auto v : A[1]) vec.push_back(v);
    vec.push_back(-inf); vec.push_back(inf);
    sort(vec.begin(), vec.end());
    printf("%d %d\n", vec.size(), T[rt].v);
    for (int q = 0; q < 100000; ++q) {
        int tp = rand() % 8, val = d(e);
        val = tmp[val];
        if (tp == 0) {
            insert(rt, val);
            vec.push_back(val);
            sort(vec.begin(), vec.end());
        }
        else if (tp == 1) {
            int k = rand() % T[rt].v + 1;
            if (vec[k] != kth(rt, k))
                abort();
        }
        else if (tp == 2) {
            if (lower_bound(vec.begin(), vec.end(), val) - vec.begin() - 1 != ask(rt, val))
                abort();
        }
        else if (tp == 3) {
            int p = lower_bound(vec.begin(), vec.end(), val) - vec.begin() - 1;
            if (vec[p] != pre(rt, val))
                abort();
        }
        else if (tp == 4) {
            int p = upper_bound(vec.begin(), vec.end(), val) - vec.begin();
            if (vec[p] != next(rt, val))
                abort();
        }
        else if (tp == 5) {
            if (count(rt, val) != count(vec.begin(), vec.end(), val))
                abort();
        }
        else if (tp == 6) {
            if (*-- --vec.end() != maximum(rt))
                abort();
        }
        else if (tp == 7) {
            if (*++vec.begin() != minimum(rt))
```

```
            abort();
        }
    }
    return 0;
}
```

9.17 可持久化整数集合

```
const int inf = 1 << 30;
static const int maxn = 210000; //maxn 应当大于所有集合大小的总和
struct node {        //该算法实现的整数集合为可重集合（multiset）
    int l, r, v;    //左节点编号 右节点编号 值
} T[maxn * 25];
int len, sz, info[maxn]; //len 是值域线段树的长度
int ranking(int v) {
    //如果不需要数据离散化直接 return v 即可，同时还要将所有的 info[v] 改为 v，此时线段树的值域为
↪   [1, len]。
    return lower_bound(info + 1, info + len + 1, v) - info;
}
void ins(int& i, int l, int r, int p) { //在区间 [l, r] 中插入 p
    T[++sz] = T[i]; i = sz;
    T[i].v++; //递增当前区间内值的个数
    if (l == r) return;
    int m = (l + r) / 2;
    if (p <= m) ins(T[i].l, l, m, p);
    else ins(T[i].r, m + 1, r, p);
}
int insert(int x, int v) { //向集合 x 中添加数值 v，返回新的集合的编号，原来的集合不会改变
    ins(x, 1, len, ranking(v));
    return x;
}
void init(int* A, int length) { //输入数组 A 的下标从 1 开始，length 表示 A 的长度
    sz = 0;                         //A 记录集合中所有可能出现的整数
    copy(A + 1, A + length + 1, info + 1);
    sort(info + 1, info + length + 1); //如果没有离散化需要对 len 进行赋值来指明线段树的值域。
    len = unique(info + 1, info + length + 1) - info - 1; //对 info 数组进行排序去重
}
int kth(int x, int k) { //查找集合 x 中的第 k 大元素
    int l = 1, r = len;
    while (l < r) {
        int m = (l + r) / 2, t = T[T[x].l].v;
        if (k <= t)
            x = T[x].l, r = m;
        else
            x = T[x].r, l = m + 1, k -= t;
    }
    if (k > T[x].v) //如果 k 大于集合 x 的大小，则返回 inf
```

```
        return inf;
    return info[r];
}
int ask(int x, int v) { //返回集合 x 中比 v 小的数的个数
    int l = 1, r = len, k = 0;
    int p = ranking(v) - 1;
    if (p <= 0) return 0;
    while (l < r) {
        int m = (l + r) / 2, t = T[T[x].l].v;
        if (p <= m)
            x = T[x].l, r = m;
        else
            x = T[x].r, l = m + 1, k += t;
    }
    k += T[x].v;
    return k;
}
int pre(int x, int v) { //在集合 x 中查询 v 的前驱，找不到返回-inf
    int k = ask(x, v);
    if (k == 0) return -inf;
    return kth(x, k);
}
int next(int x, int v) { //在集合 x 中查询 v 的后继，找不到返回 inf
    int k = ask(x, v + 1) + 1;
    return kth(x, k);
}
int count(int x, int v) { //返回集合 x 中数值 v 的个数
    int l = 1, r = len;
    int p = ranking(v);
    if (p > len || info[p] != v)
        return 0;
    while (l < r && T[x].v> 0) {
        int m = (l + r) / 2;
        if (p <= m)
            x = T[x].l, r = m;
        else
            x = T[x].r, l = m + 1;
    }
    return T[x].v;
}
bool find(int x, int v) { //返回集合 x 中是否存在 v
    return count(x, v) >= 1;
}
int maximum(int x) { //返回集合 x 中的最大值
    int l = 1, r = len;
    if (T[x].v == 0) //如果集合为空则返回-inf
```

```
        return -inf;
    while (l < r) {
        int m = (l + r) / 2, t = T[T[x].r].v;
        if (t == 0)
            x = T[x].l, r = m;
        else
            x = T[x].r, l = m + 1;
    }
    return info[r];
}
int minimum(int x) { //返回集合 x 中的最小值
    int l = 1, r = len;
    if (T[x].v == 0) //如果集合为空则返回 inf
        return inf;
    while (l < r) {
        int m = (l + r) / 2, t = T[T[x].l].v;
        if (t > 0)
            x = T[x].l, r = m;
        else
            x = T[x].r, l = m + 1;
    }
    return info[r];
}
int merge(int x, int y, int l = 1, int r = len) {     //将集合 x 与集合 y 合并成一个新的集合后返回
    if (!x) return y;                                  //不会改变 x 和 y
    if (!y) return x;
    int i = ++sz;
    if (l == r) {
        T[i].v = T[x].v + T[y].v;
        T[i].l = T[i].r = 0;
    }
    else {
        int m = (l + r) / 2;
        T[i].l = merge(T[x].l, T[y].l, l, m);
        T[i].r = merge(T[x].r, T[y].r, m + 1, r);
        T[i].v = T[T[i].l].v + T[T[i].r].v;
    }
    return i;
}
int diff(int x, int y, int l = 1, int r = len) {    //返回集合 x 与集合 y 的差集 (x - y), 不会改变
↪   x 和 y
    int i = ++sz; T[i] = T[x];                         //时间复杂度 size(y)log(len)
    if (l == r) {
        T[i].v = max(0, T[x].v - T[y].v);
    }
    else {
```

```
        int m = (l + r) / 2;
        if (T[y].l)
            T[i].l = diff(T[x].l, T[y].l, l, m);
        if (T[y].r)
            T[i].r = diff(T[x].r, T[y].r, m + 1, r);
        T[i].v = T[T[i].l].v + T[T[i].r].v;
    }
    return i;
}
int intersect(int x, int y, int l = 1, int r = len) {    //返回集合 x 与集合 y 的交集（不会改变 x
↪   和 y）
    int i = ++sz;                                        //时间复杂度 min{size(x),size(y)} *
↪   log(len)
    if (l == r) {
        T[i].v = min(T[x].v, T[y].v);
        T[i].l = T[i].r = 0;
    }
    else {
        int m = (l + r) / 2;
        T[i].l = (!T[x].l || !T[y].l ? 0 : intersect(T[x].l, T[y].l, l, m));
        T[i].r = (!T[x].r || !T[y].r ? 0 : intersect(T[x].r, T[y].r, m + 1, r));
        T[i].v = T[T[i].l].v + T[T[i].r].v;
    }
    return T[i].v == 0 ? 0 : i;
}
int symmetric(int x, int y, int l = 1, int r = len) {    //返回集合 x 与集合 y 的对称差分（不会改
↪   变 x 和 y）
    if (!x) return y;                                    //时间复杂度 min{size(x),size(y)} *
↪   log(len)
    if (!y) return x;
    int i = ++sz;
    if (l == r) {
        T[i].v = abs(T[x].v - T[y].v);
        T[i].l = T[i].r = 0;
    }
    else {
        int m = (l + r) / 2;
        T[i].l = symmetric(T[x].l, T[y].l, l, m);
        T[i].r = symmetric(T[x].r, T[y].r, m + 1, r);
        T[i].v = T[T[i].l].v + T[T[i].r].v;
    }
    return i;
}
void left(vector<int>& X, vector<int>& Y) {
    for (auto& i : X)
        i = T[i].l;
```

```
    for (auto& i : Y)
        i = T[i].l;
}
void right(vector<int>& X, vector<int>& Y) {
    for (auto& i : X)
        i = T[i].r;
    for (auto& i : Y)
        i = T[i].r;
}
int left_value(vector<int>& X, vector<int>& Y) {
    int tot = 0;
    for (auto i : X)
        tot += T[T[i].l].v;
    for (auto i : Y)
        tot -= T[T[i].l].v;
    return tot;
}
int right_value(vector<int>& X, vector<int>& Y) {
    int tot = 0;
    for (auto i : X)
        tot += T[T[i].r].v;
    for (auto i : Y)
        tot -= T[T[i].r].v;
    return tot;
}
int value(vector<int>& X, vector<int>& Y) {
    int tot = 0;
    for (auto i : X)
        tot += T[i].v;
    for (auto i : Y)
        tot -= T[i].v;
    return tot;
}
int kth(vector<int> X, vector<int> Y, int k) {    //计算 X 中集合的并集 减去 Y 中集合的并集 所得集
↪   合的第 k 大
    int l = 1, r = len;                          //必须保证 Y 的并集 是 X 的并集 的子集
    while (l < r) {
        int m = (l + r) / 2, t = left_value(X, Y);
        if (k <= t)
            left(X, Y), r = m;
        else
            right(X, Y), l = m + 1, k -= t;
    }
    if (k > value(X, Y))
        return inf;
    return info[r];
```

```
}
int ask(vector<int> X, vector<int> Y, int v) {    //计算 X 中集合的并集 减去 Y 中集合的并集 所得集
↪  合中比 v 小的数的个数
    int l = 1, r = len, k = 0;                    //必须保证 Y 的并集 是 X 的并集 的子集
    int p = ranking(v) - 1;
    if (p <= 0) return 0;
    while (l < r) {
        int m = (l + r) / 2;
        if (p <= m)
            left(X, Y), r = m;
        else {
            k += left_value(X, Y);
            right(X, Y);
            l = m + 1;
        }
    }
    k += value(X, Y);
    return k;
}
int pre(vector<int> X, vector<int> Y, int v) {    //计算 X 中集合的并集 减去 Y 中集合的并集 所得集
↪  合中 v 的前驱
    int k = ask(X, Y, v);                         //必须保证 Y 的并集 是 X 的并集 的子集
    if (k == 0) return -inf; //找不到返回-inf
    return kth(X, Y, k);
}
int next(vector<int> X, vector<int> Y, int v) {    //计算 X 中集合的并集 减去 Y 中集合的并集 所得
↪  集合中 v 的后继
    int k = ask(X, Y, v + 1) + 1;                 //必须保证 Y 的并集 是 X 的并集 的子集
    return kth(X, Y, k); //找不到返回 inf
}
int count(vector<int> X, vector<int> Y, int v) {    //计算 X 中集合的并集 减去 Y 中集合的并集 所得
↪  集合中 v 的个数
    int l = 1, r = len;                              //必须保证 Y 的并集 是 X 的并集 的子集
    int p = ranking(v);
    if (p > len || info[p] != v)
        return 0;
    while (l < r) {
        int m = (l + r) / 2;
        if (p <= m)
            left(X, Y), r = m;
        else
            right(X, Y), l = m + 1;
    }
    return value(X, Y);
}
```

```
int maximum(vector<int> X, vector<int> Y) {    //计算 X 中集合的并集 减去 Y 中集合的并集 所得集合中
↪  的最大值
    int l = 1, r = len;                        //必须保证 Y 的并集 是 X 的并集 的子集
    if (value(X, Y) == 0) //如果集合为空则返回-inf
        return -inf;
    while (l < r) {
        int m = (l + r) / 2, t = right_value(X, Y);
        if (t == 0)
            left(X, Y), r = m;
        else
            right(X, Y), l = m + 1;
    }
    return info[r];
}
int minimum(vector<int> X, vector<int> Y) {    //计算 X 中集合的并集 减去 Y 中集合的并集 所得集合中
↪  的最小值
    int l = 1, r = len;                        //必须保证 Y 的并集 是 X 的并集 的子集
    if (value(X, Y) == 0) //如果集合为空则返回 inf
        return inf;
    while (l < r) {
        int m = (l + r) / 2, t = left_value(X, Y);
        if (t > 0)
            left(X, Y), r = m;
        else
            right(X, Y), l = m + 1;
    }
    return info[r];
}
#define VALUE_TYPE 5000
int tmp[maxn], root[maxn];
vector<int> A[maxn];
int main() {
    int n = 10, q = 400000;
    default_random_engine e;
    uniform_int_distribution<int> d(1, VALUE_TYPE);
    uniform_int_distribution<int> big(-100000000, 100000000);
    //e.seed(time(0));
    for (int i = 1; i <= VALUE_TYPE; ++i)
        tmp[i] = big(e);
    init(tmp, VALUE_TYPE);
    int rt = 0;
    vector<int> vec;
    for (int i = 1; i <= 100000; ++i) {
        int val = d(e); val = tmp[val];
        if (rand() % 2)
            root[0] = insert(root[0], val), A[0].push_back(val);
```

```
        else
            root[1] = insert(root[1], val), A[1].push_back(val);
    }
    rt = symmetric(root[0], root[1]);
    sort(A[0].begin(), A[0].end());
    sort(A[1].begin(), A[1].end());
    std::set_symmetric_difference(A[0].begin(), A[0].end(), A[1].begin(), A[1].end(),
↪   back_inserter(vec));
    //rt = root[1]; vec = A[1];
    vec.push_back(-inf); vec.push_back(inf);
    sort(vec.begin(), vec.end());
    printf("%d %d\n", vec.size(), T[rt].v);
    for (int q = 0; q < 100000; ++q) {
        if (q % 1000 == 0) printf("%d\n", q);
        int tp = rand() % 8, val = d(e);
        val = tmp[val];
        if (tp == 0) {
            rt = insert(rt, val);
            vec.push_back(val);
            sort(vec.begin(), vec.end());
        }
        else if (tp == 1) {
            int k = rand() % T[rt].v + 1;
            if (vec[k] != kth(rt, k))
                abort();
        }
        else if (tp == 2) {
            if (lower_bound(vec.begin(), vec.end(), val) - vec.begin() - 1 != ask(rt, val))
                abort();
        }
        else if (tp == 3) {
            int p = lower_bound(vec.begin(), vec.end(), val) - vec.begin() - 1;
            if (vec[p] != pre(rt, val))
                abort();
        }
        else if (tp == 4) {
            int p = upper_bound(vec.begin(), vec.end(), val) - vec.begin();
            if (vec[p] != next(rt, val))
                abort();
        }
        else if (tp == 5) {
            if (count(rt, val) != count(vec.begin(), vec.end(), val))
                abort();
        }
        else if (tp == 6) {
            if (*-- --vec.end() != maximum(rt))
```

```
                abort();
        }
        else if (tp == 7) {
            if (*++vec.begin() != minimum(rt))
                abort();
        }
    }
    return 0;
}
```

9.18 线段树

```
const int maxn = 210000, offset = 210000;
const int inf = 1 << 30;
struct tmp {
    int data[maxn * 5];
    int& operator[] (int idx) {
        return data[idx + offset];
    }
}A;
#define lc t[p].lchild
#define rc t[p].rchild
int cur = 0, tot, mn, mx;
struct segment {
    int l, r, lchild, rchild;
    int sum, min, max, set, add;
}t[maxn * 4];
void init() {
    cur = 0;
}
inline void maintain(int p) {
    t[p].sum = t[lc].sum + t[rc].sum;
    t[p].min = min(t[lc].min, t[rc].min);
    t[p].max = max(t[lc].max, t[rc].max);
}
inline void mark(int p, int setv, int addv) { //给结点打标记
    if (setv >= 0) {
        t[p].set = setv; t[p].add = 0;
        t[p].min = t[p].max = setv;
        t[p].sum = setv * (t[p].r - t[p].l + 1);
    }
    if (addv) {
        t[p].add += addv;
        t[p].min += addv;
        t[p].max += addv;
        t[p].sum += addv * (t[p].r - t[p].l + 1);
    }
```

```
}
inline void pushdown(int p) { //pushdown 将标记传递给子结点，不影响当前结点的信息。
    mark(lc, t[p].set, t[p].add);
    mark(rc, t[p].set, t[p].add);
    t[p].set = -1;
    t[p].add = 0;
}
int build(int L, int R) { //只要计算 mid 的方式是 (L + R) >> 1 而不是 (L + R) / 2，就可以建立负坐标
↪   线段树。
    int p = ++cur;
    t[p].l = L;
    t[p].r = R;
    t[p].add = 0; t[p].set = -1; //清空结点标记
    if (t[p].l == t[p].r) {
        mark(p, 0, A[L]);
    }
    else {
        int mid = (t[p].l + t[p].r) >> 1;
        lc = build(L, mid);
        rc = build(mid + 1, R);
        maintain(p);
    }
    return p;
}
void update(int p, int L, int R, int op, int v) {
    if (L <= t[p].l && R >= t[p].r) {
        if (op == 0)
            mark(p, -1, v);
        else
            mark(p, v, 0);
    }
    else {
        pushdown(p); //如果没有 pushdown 只需要在最后调用一次 maintain 即可。
        int mid = (t[p].l + t[p].r) >> 1;
        if (L <= mid)
            update(lc, L, R, op, v);
        if (R > mid)
            update(rc, L, R, op, v);
        maintain(p);
    }
}
void update(int p, int pos, int v) { //单点修改
    if (t[p].l == t[p].r) {
        mark(p, -1, v);
    }
    else {
```

```
        pushdown(p);
        int mid = (t[p].l + t[p].r) >> 1;
        if (pos <= mid)
            update(lc, pos, v);
        else
            update(rc, pos, v);
        maintain(p);
    }
}
void query(int p, int L, int R) { //调用之前要设置: mn = inf; mx = -inf; tot = 0;
    if (L <= t[p].l && R >= t[p].r) {
        tot += t[p].sum;
        mn = min(mn, t[p].min);
        mx = max(mx, t[p].max);
    }
    else {
        pushdown(p);
        int mid = (t[p].l + t[p].r) >> 1;
        if (L <= mid)
            query(lc, L, R);
        if (R > mid)
            query(rc, L, R);
    }
}
int main() {
    default_random_engine e;
    int n = 100000, m = 100000;
    uniform_int_distribution<int> d(-n, n);
    for (int i = -n; i <= n; ++i)
        A[i] = d(e);
    init();
    int root = build(-n, n);
    for (int i = 1; i <= m; ++i) {
        int op = rand() % 4, a = d(e), b = d(e), v = rand();
        int L = min(a, b), R = max(a, b);
        if (op == 0) {
            for (int i = L; i <= R; ++i)
                A[i] += v;
            update(root, L, R, op, v);
        }
        else if (op == 1) {
            for (int i = L; i <= R; ++i)
                A[i] = v;
            update(root, L, R, op, v);
        }
        else if (op == 2) {
```

```
            mn = inf; mx = -inf; tot = 0;
            query(root, L, R);
            if (mn != *min_element(A.data + offset + L, A.data + offset + R + 1))
                abort();
            if (mx != *max_element(A.data + offset + L, A.data + offset + R + 1))
                abort();
            if (tot != accumulate(A.data + offset + L, A.data + offset + R + 1, 0))
                abort();
        }
        else {
            A[L] += v;
            update(root, L, v);
        }
    }
    return 0;
}
```

9.19 动态开点线段树

```
const int maxn = 210000, offset = 210000;
const int inf = 1 << 30;
struct tmp {
    int data[maxn * 5];
    int& operator[] (int idx) {
        return data[idx + offset];
    }
}A;
#define lc t[p].lchild
#define rc t[p].rchild
int cur = 0, tot, mn, mx;
struct segment {
    int lchild, rchild;
    int sum, min, max, set, add;
}t[maxn * 4];
void init() {
    cur = 0;
}
inline void newnode(int& p) {
    if (p == 0) {
        p = ++cur;
        memset(&t[p], 0, sizeof(t[p]));
        t[p].set = -1;
    }
}
inline void maintain(int p) {
    t[p].sum = t[lc].sum + t[rc].sum;
    t[p].min = min(t[lc].min, t[rc].min);
```

```
    t[p].max = max(t[lc].max, t[rc].max);
}
inline void mark(int &p, int setv, int addv, int x, int y) { //给结点打标记
    newnode(p);
    if (setv >= 0) {
        t[p].set = setv; t[p].add = 0;
        t[p].min = t[p].max = setv;
        t[p].sum = setv * (y - x + 1);
    }
    if (addv) {
        t[p].add += addv;
        t[p].min += addv;
        t[p].max += addv;
        t[p].sum += addv * (y - x + 1);
    }
}
inline void pushdown(int p, int x, int y) { //pushdown 将标记传递给子结点，不影响当前结点的信息。
    int mid = (x + y) >> 1;
    mark(lc, t[p].set, t[p].add, x, mid);
    mark(rc, t[p].set, t[p].add, mid+1, y);
    t[p].set = -1;
    t[p].add = 0;
}
void update(int &p, int L, int R, int op, int v, int x, int y) {
    newnode(p);
    if (L <= x && R >= y) {
        if (op == 0)
            mark(p, -1, v, x, y);
        else
            mark(p, v, 0, x, y);
    }
    else {
        pushdown(p, x, y); //如果没有 pushdown 只需要在最后调用一次 maintain 即可。
        int mid = (x + y) >> 1;
        if (L <= mid)
            update(lc, L, R, op, v, x, mid);
        if (R > mid)
            update(rc, L, R, op, v, mid+1, y);
        maintain(p);
    }
}
void query(int &p, int L, int R, int x, int y) { //调用之前要设置: mn = inf; mx = -inf; tot = 0;
    newnode(p);
    if (L <= x && R >= y) {
        tot += t[p].sum;
        mn = min(mn, t[p].min);
```

```
        mx = max(mx, t[p].max);
    }
    else {
        pushdown(p, x, y);
        int mid = (x + y) >> 1;
        if (L <= mid)
            query(lc, L, R, x, mid);
        if (R > mid)
            query(rc, L, R, mid+1, y);
    }
}
int main() {
    default_random_engine e;
    int n = 100000, m = 100000;
    uniform_int_distribution<int> d(-n, n);
    for (int i = -n; i <= n; ++i)
        A[i] = 0;
    init();
    int root = 0;
    for (int i = 1; i <= m; ++i) {
        int op = rand() % 3, a = d(e), b = d(e), v = rand();
        int L = min(a, b), R = max(a, b);
        if (op == 0) {
            for (int i = L; i <= R; ++i)
                A[i] += v;
            update(root, L, R, op, v, -n, n);
        }
        else if (op == 1) {
            for (int i = L; i <= R; ++i)
                A[i] = v;
            update(root, L, R, op, v, -n, n);
        }
        else {
            mn = inf; mx = -inf; tot = 0;
            query(root, L, R, -n, n);
            if (mn != *min_element(A.data + offset + L, A.data + offset + R + 1))
                abort();
            if (mx != *max_element(A.data + offset + L, A.data + offset + R + 1))
                abort();
            if (tot != accumulate(A.data + offset + L, A.data + offset + R + 1, 0))
                abort();
        }
    }
    return 0;
}
```

9.20 主席树

```
/*
动态区间颜色数:
    把主席树第 i 个位置的权值设为这个数上一次出现的位置，然后查询区间 [L, R] 中的颜色数转化为
    查询区间 [L, R] 中比 L 小的值有多少个。(可以直接用 ask 函数查询)
    更快的方法：直接在主席树的第 R 个版本中查询比 L 小的数有多少个，然后减去 (L - 1) 即可。
*/
const int inf = 1 << 30;
struct ZXTree
{
    static const int maxn = 100010;
    struct node
    {
        int l, r, v; //左节点编号 右节点编号 值
    } T[maxn * 25];
    int n, sz, root[maxn], data[maxn]; //n 是值域线段树的长度
    void ins(int &i, int l, int r, int p) //在区间 [l, r] 中插入 p
    {
        int m = (l + r) / 2;
        T[++sz] = T[i]; i = sz;
        T[i].v++; //递增当前区间内值的个数
        if (l == r) return;
        if (p <= m) ins(T[i].l, l, m, p);
        else ins(T[i].r, m + 1, r, p);
    }
    int rank(int v)
    {
        //如果不需要数据离散化直接 return v 即可，同时还要将所有的 data[v] 改为 v，此时主席树的值域为 [1, n]。
        return lower_bound(data + 1, data + n + 1, v) - data;
    }
    void init(int *A, int length) //输入数组 A 的下标从 1 开始，length 表示 A 的长度
    {
        root[0] = sz = 0;
        copy(A + 1, A + length + 1, data + 1);
        sort(data + 1, data + length + 1); //如果没有离散化需要对 n 进行赋值来指明主席树的值域。
        this->n = unique(data + 1, data + length + 1) - data - 1; //对 data 数组进行排序去重
        for (int i = 1; i <= length; ++i) //依次把原数组中每个位置的排名插入到可持久化线段树中
            ins(root[i] = root[i - 1], 1, n, rank(A[i])); //插入 rank[i]，如果没有对 A 数组进行离散化，此处直接插入 A[i] 即可。
                                                          //root[i] 表示用 A[1] 到 A[i] 的所有值构建的权值线段树。
    }
    int kth(int x, int y, int k)  //查询原数组中区间 [x, y] 中的第 k 小的值，如果 k 大于区间长度会返回无效值!!!
    {
```

```
    int l = 1, r = n;
    x = root[x - 1], y = root[y];
    while (l < r)
    {
        int m = (l + r) / 2, t = T[T[y].l].v - T[T[x].l].v;
        if (k <= t)
            x = T[x].l, y = T[y].l, r = m;
        else
            x = T[x].r, y = T[y].r, l = m + 1, k -= t;
    }
    return data[r];
}
int ask(int x, int y, int v) //查询原数组中区间 [x, y] 中比 v 小的值的个数
{
    int l = 1, r = n, k = 0;
    x = root[x - 1], y = root[y];
    int p = rank(v) - 1;
    if (p <= 0) return 0;
    while (l < r)
    {
        int m = (l + r) / 2, t = T[T[y].l].v - T[T[x].l].v;
        if (p <= m)
            x = T[x].l, y = T[y].l, r = m;
        else
            x = T[x].r, y = T[y].r, l = m + 1, k += t;
    }
    k += T[y].v - T[x].v;
    return k;
}
int pre(int x, int y, int l, int r, int p)
{
    int m = (l + r) / 2, v = T[y].v - T[x].v;
    if (l == r) return v > 0 ? data[r] : -inf;
    int t = T[T[y].r].v - T[T[x].r].v;
    if (p <= m || t == 0) return pre(T[x].l, T[y].l, l, m, p); //如果 p 在中点的左侧或者右子树为空，则直接在左子树查找
    int k = pre(T[x].r, T[y].r, m + 1, r, p);
    if (k != -inf) return k;
    return pre(T[x].l, T[y].l, l, m, p);
}
int pre(int x, int y, int v) //在区间 [x, y] 中查询 v 的前驱，找不到返回-inf
{
    int p = rank(v) - 1;
    if (p <= 0) return -inf;
    return pre(root[x - 1], root[y], 1, n, p);
}
```

```
int pre2(int x, int y, int v) //在区间 [x, y] 中查询 v 的前驱，找不到返回-inf
{
    int k = ask(x, y, v);
    if (k == 0) return -inf;
    return kth(x, y, k);
}
int next(int x, int y, int l, int r, int p)
{
    int m = (l + r) / 2, v = T[y].v - T[x].v;
    if (l == r) return v > 0 ? data[r] : inf;
    int t = T[T[y].l].v - T[T[x].l].v;
    if (p > m || t == 0) return next(T[x].r, T[y].r, m + 1, r, p);
    int k = next(T[x].l, T[y].l, l, m, p);
    if (k != inf) return k;
    return next(T[x].r, T[y].r, m + 1, r, p);
}
int next(int x, int y, int v) //在区间 [x, y] 中查询 v 的后继，找不到返回 inf
{
    int p = rank(v + 1);
    if (p > n) return inf;
    return next(root[x - 1], root[y], 1, n, p);
}
int next2(int x, int y, int v) //在区间 [x, y] 中查询 v 的后继，找不到返回 inf
{
    int k = ask(x, y, v + 1) + 1;
    if (k > y - x + 1) return inf;
    return kth(x, y, k);
}
int count(int x, int y, int v)    //返回区间 [x, y] 中 v 的个数
{
    int l = 1, r = n;
    x = root[x - 1], y = root[y];
    int p = rank(v);
    if (p > n || data[p] != v)
        return 0;
    while (l < r && T[y].v - T[x].v > 0)
    {
        int m = (l + r) / 2;
        if (p <= m)
            x = T[x].l, y = T[y].l, r = m;
        else
            x = T[x].r, y = T[y].r, l = m + 1;
    }
    return T[y].v - T[x].v;
}
bool find(int x, int y, int v)    //查询原数组的区间 [x, y] 中是否有元素 v
```

```
    {
        return count(x, y, v) >= 1;
    }
}zxtree;
const int maxn = 100000;
int A[maxn], n, q;
int main()
{
    freopen("in.txt", "r", stdin);
    freopen("out.txt", "w", stdout);
    scanf("%d %d", &n, &q);
    for (int i = 1; i <= n; ++i)
        scanf("%d", &A[i]);
    zxtree.init(A, n);
    for (int i = 0; i < q; ++i)
    {
        int tp, L, R, v;
        scanf("%d %d %d %d", &tp, &L, &R, &v);
        if (tp == 0)
        {
            printf("%d\n", zxtree.pre(L, R, v));
        }
        else if (tp == 1)
        {
            printf("%d\n", zxtree.next(L, R, v));
        }
        else if (tp == 2)
        {
            printf("%d\n", zxtree.kth(L, R, v));
        }
        else if (tp == 3)
        {
            printf("%d\n", zxtree.ask(L, R, v));
        }
        else if (tp == 4)
        {
            printf("%d\n", zxtree.count(L, R, v));
        }
        else
        {
            bool result = zxtree.find(L, R, v);
            printf("%d\n", (int)result);
        }
    }
    return 0;
}
```

9.21 动态主席树

```
#define lowbit(x) ((x) & (-x))
const int inf = 1 << 30;
struct Dynamic_ZXTree {
    static const int maxn = 51000;
    struct node {
        int l, r, v; //左节点编号 右节点编号 值
    } T[maxn * 100]; //size(T) = maxn * logmaxn * logmaxn, size(data) = maxn + maxq
    int n, sz, length, nx, ny, data[maxn * 3], seq[maxn], X[maxn], Y[maxn], C[maxn]; //n 是值域线
↪   段树的长度
    void sum(int x, int y) {
        nx = ny = 0;
        for (int i = x; i > 0; i -= lowbit(i))
            X[nx++] = C[i];
        for (int i = y; i > 0; i -= lowbit(i))
            Y[ny++] = C[i];
    }
    void add(int x, int value, int v) { //内部函数，不要在类外访问！
        int p = rank(value);
        for (int i = x; i <= length; i += lowbit(i))
            ins(C[i], 1, n, p, v);
    }
    void set(int x, int value) { //将序列中下标为 x 的位置的数修改为 value
        add(x, seq[x], -1);
        add(x, value, 1);
        seq[x] = value;
    }
    void ins(int& i, int l, int r, int p, int v = 1) {
        int m = (l + r) / 2;
        if (i == 0) {
            T[++sz] = T[i];
            i = sz;
        }
        T[i].v += v;
        if (l == r) return;
        if (p <= m) ins(T[i].l, l, m, p, v);
        else ins(T[i].r, m + 1, r, p, v);
    }
    int rank(int v) {
        //如果不需要数据离散化直接 return v 即可，同时还要将所有的 data[v] 改为 v，此时线段树的值域
↪   为 [1, n]。
        return lower_bound(data + 1, data + n + 1, v) - data;
    }
    void init(int* A, int length, int* all, int size)    //输入数组 A 的下标从 1 开始，length 表示
↪   A 的长度
```

```
    {                                              //数组 all 记录所有可能出现的值（包括可能
↪   修改为的值），下标从 1 开始，size 为数组 all 的长度
        this->length = length;
        copy(A + 1, A + length + 1, seq + 1);
        copy(all + 1, all + size + 1, data + 1);    //data[i] 储存原序列中所有可能出现的值（包括可
↪   能修改成的值）离散化之后第 i 小的值
        sort(data + 1, data + size + 1);            //如果不用离散化，赋值 data[i] = i
        n = unique(data + 1, data + size + 1) - data - 1;
        for (int i = 1; i <= length; ++i)
            add(i, rank(seq[i]), 1);
    }
    void left() {
        for (int i = 0; i < ny; ++i)
            Y[i] = T[Y[i]].l;
        for (int i = 0; i < nx; ++i)
            X[i] = T[X[i]].l;
    }
    void right() {
        for (int i = 0; i < ny; ++i)
            Y[i] = T[Y[i]].r;
        for (int i = 0; i < nx; ++i)
            X[i] = T[X[i]].r;
    }
    int left_value() {
        int tot = 0;
        for (int i = 0; i < ny; ++i)
            tot += T[T[Y[i]].l].v;
        for (int i = 0; i < nx; ++i)
            tot -= T[T[X[i]].l].v;
        return tot;
    }
    int value() {
        int tot = 0;
        for (int i = 0; i < ny; ++i)
            tot += T[Y[i]].v;
        for (int i = 0; i < nx; ++i)
            tot -= T[X[i]].v;
        return tot;
    }
    int kth(int x, int y, int k) { //查询原数组中区间 [x, y] 中的第 k 小的值，如果 k 大于区间长度会
↪   返回无效值!!!
        int l = 1, r = n;
        sum(x - 1, y);
        while (l < r) {
            int m = (l + r) / 2, t = left_value();
            if (k <= t)
```

```
                left(), r = m;
            else
                right(), l = m + 1, k -= t;
        }
        return data[r];
    }
    int ask(int x, int y, int v) { //查询原数组中区间 [x, y] 中比 v 小的值的个数
        int l = 1, r = n, k = 0;
        sum(x - 1, y);
        int p = rank(v) - 1;
        if (p <= 0) return 0;
        while (l < r) {
            int m = (l + r) / 2;
            if (p <= m)
                left(), r = m;
            else {
                k += left_value();
                right();
                l = m + 1;
            }
        }
        k += value();
        return k;
    }
    int pre(int x, int y, int v) { //在区间 [x, y] 中查询 v 的前驱，找不到返回-inf
        int k = ask(x, y, v);
        if (k == 0) return -inf;
        return kth(x, y, k);
    }
    int next(int x, int y, int v) { //在区间 [x, y] 中查询 v 的后继，找不到返回 inf
        int k = ask(x, y, v + 1) + 1;
        if (k > y - x + 1) return inf;
        return kth(x, y, k);
    }
    int count(int x, int y, int v) { //返回区间 [x, y] 中 v 的个数
        int l = 1, r = n;
        sum(x - 1, y);
        int p = rank(v);
        if (p > n || data[p] != v)
            return 0;
        while (l < r) {
            int m = (l + r) / 2;
            if (p <= m)
                left(), r = m;
            else
                right(), l = m + 1;
```

```
        }
        return value();
    }
    bool find(int x, int y, int v) {    //查询原数组的区间 [x, y] 中是否有元素 v
        return count(x, y, v) >= 1;
    }
}tree;
const int maxn = 100000;
int A[maxn], B[maxn], n, q;
int main() {
    freopen("in.txt", "r", stdin);
    freopen("out.txt", "w", stdout);
    scanf("%d %d", &n, &q);
    for (int i = 1; i <= n; ++i)
        scanf("%d", &A[i]);
    for (int i = 1; i <= n; i++)
        B[i] = i;
    tree.init(A, n, B, n);
    for (int i = 0; i < q; ++i) {
        int tp, L, R, v;
        scanf("%d %d %d %d", &tp, &L, &R, &v);
        if (tp == 0) {
            printf("%d\n", tree.pre(L, R, v));
        }
        else if (tp == 1) {
            printf("%d\n", tree.next(L, R, v));
        }
        else if (tp == 2) {
            printf("%d\n", tree.kth(L, R, v));
        }
        else if (tp == 3) {
            printf("%d\n", tree.ask(L, R, v));
        }
        else if (tp == 4) {
            printf("%d\n", tree.count(L, R, v));
        }
        else if (tp == 5) {
            bool result = tree.find(L, R, v);
            printf("%d\n", (int)result);
        }
        else {
            tree.set(L, v);
        }
    }
    return 0;
}
```

9.22 动态开点线段树（单点加-区间求和-合并）

```
const int inf = 1 << 30;
const int maxn = 110000;
struct TreeSet {
    struct node {
        int l, r, v;
    } T[maxn * 25];
    int left, right, sz; //空的线段树标号为 0
    void init(int L, int R) { //所有线段树对应区间都为 [L, R]
        left = L;
        right = R;
        sz = 1;
    }
    int newnode() {
        T[sz].l = T[sz].r = T[sz].v = 0;
        return sz++;
    }
    void insert(int x, int pos, int value) { //编号为 x 的线段树的 pos 位置上的值加上 value
        int L = left, R = right;
        while (L < R) {
            int M = (L + R) >> 1;
            T[x].v += value;
            if (pos <= M) {
                if (!T[x].l)
                    T[x].l = newnode();
                x = T[x].l;
                R = M;
            }
            else {
                if (!T[x].r)
                    T[x].r = newnode();
                x = T[x].r;
                L = M + 1;
            }
        }
        T[x].v += value;
    }
    int sum(int x, int pos) { //在编号为 x 的线段树上计算 pos 位置的前缀和
        int L = left, R = right, res = 0;
        while (x && L < R) {
            int M = (L + R) >> 1, t = T[T[x].l].v;
            if (pos <= M)
                x = T[x].l, R = M;
            else
                x = T[x].r, L = M + 1, res += t;
        }
```

```
        res += T[x].v;
        return res;
    }
    int merge(int x, int y, int l, int r) {    //将集合 x 与集合 y 合并成一个新的集合后返回
        if (!x) return y;                      //合并之后 x、y 均会失效
        if (!y) return x;
        if (l == r)
            T[x].v += T[y].v;
        else {
            int m = (l + r) >> 1;
            T[x].l = merge(T[x].l, T[y].l, l, m);
            T[x].r = merge(T[x].r, T[y].r, m + 1, r);
            T[x].v = T[T[x].l].v + T[T[x].r].v;
        }
        return x;
    }
    int merge(int x, int y) {
        return merge(x, y, left, right);
    }
}tree;
int A[210000];
int main() {
    default_random_engine e;
    uniform_int_distribution<int> d(0, 100000), v(1, 10);
    e.seed(334);
    int n = 100000;
    tree.init(0, n);
    int root = tree.newnode();
    for (int i = 0; i < 100000; ++i) {
        int tp = rand() % 2;
        int pos = d(e), value = v(e);
        if (tp == 0) {
            int tmp = tree.newnode();
            tree.insert(tmp, pos, value);
            root = tree.merge(root, tmp);
            A[pos] += value;
        }
        else {
            int ans = tree.sum(root, pos), res = accumulate(A, A + pos + 1, 0);
            if (ans != res) {
                printf("%d %d\n", ans, res);
                abort();
            }
        }
    }
    return 0;
```

```
}
```

9.23 可持久化线段树

```
/*
可持久化线段树：区间取 min
每次 update 之后返回新版本的线段树的根结点。
查询操作传入相应版本的线段树的根结点即可。
因为要可持久化，所以必须标记永久化，不能 pushdown。
*/
const int maxn = 110000;
const int inf = 1 << 30;
struct node {
    int l, r, lc, rc;
    int minv, tag;
} t[maxn * 20];
int cur = 0, mn;
void init() {
    cur = 0;
}
void maintain(int p) {
    int lc = t[p].lc, rc = t[p].rc;
    t[p].minv = t[p].tag;
    if (t[p].r > t[p].l) {
        t[p].minv = min({ t[lc].minv, t[rc].minv, t[p].tag });
    }
}
int build(int L, int R) { //只要计算 mid 的方式是 (L + R) >> 1 而不是 (L + R) / 2，就可以建立负坐标
↪   线段树。
    int p = ++cur;
    t[p].l = L;
    t[p].r = R;
    t[p].tag = inf; //重置结点标记
    if (L < R) {
        int mid = (L + R) >> 1;
        t[p].lc = build(L, mid);
        t[p].rc = build(mid + 1, R);
    }
    maintain(p);
    return p;
}
int update(int i, int L, int R, int v) { //区间取 min
    int p = ++cur; t[p] = t[i];
    if (L <= t[p].l && R >= t[p].r) {
        t[p].tag = min(t[p].tag, v);
    }
    else {
```

```
        int mid = (t[p].l + t[p].r) >> 1;
        if (L <= mid)
            t[p].lc = update(t[p].lc, L, R, v);
        if (R > mid)
            t[p].rc = update(t[p].rc, L, R, v);
    }
    maintain(p);
    return p;
}
void query(int p, int L, int R) { //调用之前设置 mn = inf, 查询区间最小值
    mn = min(mn, t[p].tag);
    if (L <= t[p].l && R >= t[p].r) {
        mn = min(mn, t[p].minv);
    }
    else {
        int mid = (t[p].l + t[p].r) >> 1;
        if (L <= mid)
            query(t[p].lc, L, R);
        if (R > mid)
            query(t[p].rc, L, R);
    }
}
int main() {
    init();
    int root[10], n = 10;
    root[0] = build(1, n);
    root[1] = update(root[0], 1, 7, 10);
    root[2] = update(root[1], 5, 8, 0);
    mn = inf; query(root[2], 7, 7);
    printf("%d\n", mn);
    return 0;
}
```

9.24 线段树（历史最大值）

```
/*
1. 标记 (a, b) 表示对区间中的所有值 v 执行: v = max(v + a, b)
2. 将标记 (c, d) 合并到 (a, b) 上得到: (a + c, max(b + c, d))
3. 标记 (a, b) 与 (c, d) 取最大得到: (max(a, c), max(b, d))
4. 必须满足 inf > 操作次数 * 单次修改的最大增加值
*/
#define lc t[p].lchild
#define rc t[p].rchild
const int maxn = 510000;
const long long inf = 1LL << 50;
long long A[maxn], mx, hmax;
int cur = 0;
```

```
struct segment {
    int l, r, lchild, rchild;
    long long max, hmax; //最大值、历史最大值
    long long a, b, x, y; //(a, b) 为当前标记，(x, y) 为历史最大标记
}t[maxn * 4];
inline void maintain(int p) {
    t[p].max = max(t[lc].max, t[rc].max);
    t[p].hmax = max(t[p].hmax, t[p].max);
}
inline void mark(int p, long long c, long long d) {
    t[p].a += c;
    t[p].b = max(t[p].b + c, d);
    t[p].max = max(t[p].max + c, d);
    t[p].x = max(t[p].x, t[p].a);
    t[p].y = max(t[p].y, t[p].b);
    t[p].hmax = max(t[p].hmax, t[p].max);
    t[p].a = max(t[p].a, -inf); //防止出现：(操作次数 * inf) > LLONG_MAX
}
inline void change(int p, long long c, long long d) {
    long long i = t[p].a + c, j = max(t[p].b + c, d);
    t[p].x = max(t[p].x, i);
    t[p].y = max(t[p].y, j);
    t[p].hmax = max(t[p].hmax, max(t[p].max + c, d));
}
inline void pushdown(int p) {
    change(lc, t[p].x, t[p].y);
    change(rc, t[p].x, t[p].y);
    mark(lc, t[p].a, t[p].b);
    mark(rc, t[p].a, t[p].b);
    t[p].x = t[p].a = 0;
    t[p].y = t[p].b = -inf;
}
int build(int L, int R) {
    int p = ++cur;
    t[p].l = L;
    t[p].r = R;
    t[p].x = t[p].a = 0;
    t[p].y = t[p].b = -inf;
    t[p].hmax = -inf;
    if (t[p].l == t[p].r) {
        t[p].hmax = t[p].max = A[L];
    }
    else {
        int mid = (t[p].l + t[p].r) >> 1;
        lc = build(L, mid);
        rc = build(mid + 1, R);
```

```
        maintain(p);
    }
    return p;
}
void query(int p, int L, int R) {
    if (L <= t[p].l && R >= t[p].r) {
        mx = max(mx, t[p].max);
        hmax = max(hmax, t[p].hmax);
    }
    else {
        pushdown(p);
        int mid = (t[p].l + t[p].r) >> 1;
        if (L <= mid)
            query(lc, L, R);
        if (R > mid)
            query(rc, L, R);
    }
}
void update(int p, int L, int R, long long a, long long b) {
    if (L <= t[p].l && R >= t[p].r) {
        mark(p, a, b);
    }
    else {
        pushdown(p);
        int mid = (t[p].l + t[p].r) >> 1;
        if (L <= mid)
            update(lc, L, R, a, b);
        if (R > mid)
            update(rc, L, R, a, b);
        maintain(p);
    }
}
int main() {
    //freopen("in.txt", "r", stdin);
    int n, m;
    scanf("%d", &n);
    for (int i = 1; i <= n; ++i)
        scanf("%lld", &A[i]);
    int root = build(1, n);
    scanf("%d", &m);
    while (m--) {
        char tp;
        int L, R;
        long long v;
        scanf(" %c %d %d\n", &tp, &L, &R);
        if (tp == 'Q') { //查询区间最大值
```

```
            mx = -inf;
            query(root, L, R);
            printf("%lld\n", mx);
        }
        else if (tp == 'A') { //查询区间历史最大值
            hmax = -inf;
            query(root, L, R);
            printf("%lld\n", hmax);
        }
        else if (tp == 'P') { //区间 add
            scanf("%lld", &v);
            update(root, L, R, v, -inf);
        }
        else { //区间 set
            scanf("%lld", &v);
            update(root, L, R, -inf, v);
        }
    }
    return 0;
}
```

9.25 zkw 线段树（单点加-区间加-单点查询-区间求和）

```
const int maxn = 110000;
long long A[maxn];
struct {
    int n, m;
    long long sum[maxn * 4], add[maxn * 4];
    void build(int size) {
        this->n = size;
        for (m = 1; m <= n; m <<= 1);
        memset(add, 0, sizeof(long long) * (n + m + 1));
        for (int i = 1; i <= n; ++i)
            sum[m + i] = A[i];
        for (int i = m - 1; i; --i)
            sum[i] = sum[i << 1] + sum[i << 1 | 1];
    }
    void update(int pos, int v) { //单点加 v
        for (int x = pos + m; x > 1; x >>= 1)
            sum[x] += v;
    }
    void update(int s, int t, int v) { //区间加 v
        long long lc = 0, rc = 0, len = 1;
        for (s += m - 1, t += m + 1; s ^ t ^ 1; s >>= 1, t >>= 1, len <<= 1) {
            if (~s & 1)
                add[s ^ 1] += v, lc += len;
            if (t & 1)
```

```
                add[t ^ 1] += v, rc += len;
            sum[s >> 1] += v * lc, sum[t >> 1] += v * rc;
        }
        for (lc += rc; s; s >>= 1)
            sum[s >> 1] += v * lc;
    }
    long long query(int pos) { //单点查询
        long long ans = sum[pos + m];
        for (int x = pos + m; x; x >>= 1)
            ans += add[x];
        return ans;
    }
    long long query(int s, int t) { //计算区间和
        long long ans = 0, lc = 0, rc = 0, len = 1;
        for (s += m - 1, t += m + 1; s ^ t ^ 1; s >>= 1, t >>= 1, len <<= 1) {
            if (s & 1 ^ 1)
                ans += sum[s ^ 1] + len * add[s ^ 1], lc += len;
            if (t & 1)
                ans += sum[t ^ 1] + len * add[t ^ 1], rc += len;
            ans += add[s >> 1] * lc;
            ans += add[t >> 1] * rc;
        }
        for (lc += rc, s >>= 1; s; s >>= 1)
            ans += add[s] * lc;
        return ans;
    }
}tree;
int main() {
    default_random_engine e;
    uniform_int_distribution<int> d(1, 100000);
    e.seed(35345);
    int n = 100000;
    for (int i = 1; i <= n; ++i)
        A[i] = d(e);
    auto start = clock();
    tree.build(n);
    for (int i = 0; i < 1000000; ++i) {
        int tp = d(e) % 4, a = d(e) % n + 1, b = d(e) % n + 1;
        int L = min(a, b), R = max(a, b);
        if (tp == 0) {
            tree.update(L, R, 1);
            for (int i = L; i <= R; ++i)
                A[i] += 1;
        }
        else if (tp == 1) {
            long long ans = tree.query(L, R), res = accumulate(A + L, A + R + 1, 0LL);
```

```
            if (ans != res)
                abort();
        }
        else if (tp == 2) {
            tree.update(L, 7);
            A[L] += 7;
        }
        else {
            long long ans = tree.query(L);
            if (ans != A[L])
                abort();
        }
    }
    auto end = clock();
    printf("time: %.2f\n", static_cast<double>(end - start) / CLOCKS_PER_SEC);
    for (int i = 1; i <= 100000; ++i)
        if (A[i] != tree.query(i))
            abort();
    return 0;
}
```

9.26 SplayTree

```
const int maxn = 110000;
//若要修改一个点的点权，应当先将其 splay 到根，然后修改，最后还要调用 pushup 维护。
//调用完 splay 之后根结点会改变，应该用 splay 的返回值更新根结点。
namespace splay_tree {
    int ch[maxn][2], fa[maxn], stk[maxn], rev[maxn], sz[maxn], key[maxn], cur;
    void init() {
        cur = 0;
    }
    int newnode(int val) {
        int x = ++cur;
        ch[x][0] = ch[x][1] = fa[x] = rev[x] = 0;
        sz[x] = 1;
        key[x] = val;
        return x;
    }
    inline bool son(int x) {
        return ch[fa[x]][1] == x;
    }
    inline void pushup(int x) {
        sz[x] = sz[ch[x][0]] + sz[ch[x][1]] + 1;
    }
    inline void pushdown(int x) {
        if (rev[x]) {
            rev[x] = 0;
```

```
        swap(ch[x][0], ch[x][1]);
        rev[ch[x][0]] ^= 1;
        rev[ch[x][1]] ^= 1;
    }
}
void rotate(int x) {
    int y = fa[x], z = fa[y], c = son(x);
    if (fa[y])
        ch[z][son(y)] = x;
    fa[x] = z;
    ch[y][c] = ch[x][!c];
    fa[ch[y][c]] = y;
    ch[x][!c] = y;
    fa[y] = x;
    pushup(y);
}
void ascend(int x) {
    for (int y = fa[x]; y; rotate(x), y = fa[x]) if (fa[y])
        son(x) ^ son(y) ? rotate(x) : rotate(y);
    pushup(x);
}
int splay(int x) { //没有 pushdown 操作时，可以直接用 ascend 替换 splay
    int top = 0;
    for (int i = x; i; i = fa[i])
        stk[++top] = i;
    while (top)
        pushdown(stk[top--]);
    ascend(x);
    return x;
}
int splay(int x, int k) { //将以 x 为根的子树中的第 k 个结点旋转到根结点
    while (pushdown(x), k != sz[ch[x][0]] + 1) {
        if (k <= sz[ch[x][0]])
            x = ch[x][0];
        else
            k -= sz[ch[x][0]] + 1, x = ch[x][1];
    }
    if (x) ascend(x);
    return x;
}
template<typename ...T> int merge(int x, int y, T... args) {
    if constexpr (sizeof...(args) == 0) {
        if (x == 0) return y; //swap(x, y);
        x = splay(x, sz[x]);
        ch[x][1] = y; fa[y] = x;
        pushup(x);
```

```
            return x;
        }
        else {
            return merge(merge(x, y), args...);
        }
    }
    auto split(int x, int pos) { //分成两个区间 [1, pos - 1] 和 [pos, n]
        if (pos == sz[x] + 1)
            return make_pair(x, 0);
        x = splay(x, pos);
        int y = ch[x][0];
        fa[y] = ch[x][0] = 0;
        pushup(x);
        return make_pair(y, x);
    }
    auto extract(int x, int L, int R) {
        auto [left, y] = split(x, L);
        auto [mid, right] = split(y, R - L + 2);
        return make_tuple(left, mid, right);
    }
    void traverse(int x) {
        if (x != 0) {
            traverse(ch[x][0]);
            printf("%d ", key[x]);
            //printf("%d (left: %d, right: %d) sz(%d) key(%d)\n", x, ch[x][0], ch[x][1], sz[x],
    ↪   key[x]);
            traverse(ch[x][1]);
        }
    }
}
using namespace splay_tree;
int main() {
    init();
    int nd[50], root = 0;
    for (int i = 1; i <= 10; ++i) {
        nd[i] = newnode(i);
        root = merge(root, nd[i]);
    }
    traverse(get<1>(extract(root, 3, 10))); printf("\n");
    return 0;
}
```

9.27 可持久化并查集

```
const int maxn = 210000;
struct node {
    int lc, rc, fa, size;
```

```
} t[maxn * 40]; //若内存不够，可以考虑改成按深度合并
int cur, length;
void init(int n) {
    cur = 0;
    length = n;
}
int build(int L = 1, int R = length) {
    int p = ++cur;
    t[p].fa = L;
    t[p].size = 1;
    if (L < R) {
        int mid = (L + R) >> 1;
        t[p].lc = build(L, mid);
        t[p].rc = build(mid + 1, R);
    }
    return p;
}
int link(int i, int pos, int fa, int L = 1, int R = length) {
    int p = ++cur; t[p] = t[i];
    t[p].fa = fa;
    if (L < R) {
        int mid = (L + R) >> 1;
        if (pos <= mid)
            t[p].lc = link(t[p].lc, pos, fa, L, mid);
        else
            t[p].rc = link(t[p].rc, pos, fa, mid + 1, R);
    }
    return p;
}
int update(int i, int pos, int size, int L = 1, int R = length) {
    int p = ++cur; t[p] = t[i];
    if (L == R)
        t[p].size += size;
    else {
        int mid = (L + R) >> 1;
        if (pos <= mid)
            t[p].lc = update(t[p].lc, pos, size, L, mid);
        else
            t[p].rc = update(t[p].rc, pos, size, mid + 1, R);
    }
    return p;
}
int query(int p, int pos, int L = 1, int R = length) {
    if (L == R)
        return p;
    else {
```

```
        int mid = (L + R) >> 1;
        if (pos <= mid)
            return query(t[p].lc, pos, L, mid);
        else
            return query(t[p].rc, pos, mid + 1, R);
    }
}
int find(int i, int pos) {
    int p = query(i, pos);
    return t[p].fa == pos ? p : find(i, t[p].fa);
}
int join(int i, int x, int y) { //在版本为 i 的并查集中连接点 x 和点 y，返回新的并查集
    int a = find(i, x);
    int b = find(i, y);
    if (t[a].fa != t[b].fa) { //如果 x 和 y 已经在一个集合中了，则不修改直接返回
        if (t[a].size > t[b].size)
            swap(a, b);
        int p = link(i, t[a].fa, t[b].fa);
        p = update(p, t[b].fa, t[a].size);
        return p;
    }
    return i;
}
bool same(int i, int x, int y) { //判断在版本为 i 的并查集中，点 x 和点 y 是否属于相同的集合
    int a = find(i, x);
    int b = find(i, y);
    return t[a].fa == t[b].fa;
}
int main() {
    //freopen("in.txt", "r", stdin);
    static int root[maxn];
    int n, m;
    scanf("%d %d", &n, &m);
    init(n);
    int r = root[0] = build();
    for (int i = 1; i <= m; ++i) {
        int tp, x, y, k;
        scanf("%d", &tp);
        if (tp == 1) {
            scanf("%d %d", &x, &y);
            r = join(r, x, y);
        }
        else if (tp == 2) {
            scanf("%d", &k);
            r = root[k];
        }
```

```
        else {
            scanf("%d %d", &x, &y);
            int ans = same(r, x, y);
            printf("%d\n", ans);
        }
        root[i] = r;
    }
    return 0;
}
```

9.28 KD-Tree

```
#define sqr(x) ((long long)(x)*(x))
const int maxn = 210000;
const int inf = 1 << 30;
const int k = 2;
using point = array<int, k>;
namespace kdt {
    int cur, sz[maxn], ch[maxn][2]; //如果没有插入操作，只需要询问最近点，则可以不维护 sz 属性
    int mark[maxn], real[maxn]; //没有删除操作时可以不记录这两个属性
    point mn[maxn], mx[maxn], val[maxn];
    int ans;
    void init() {
        cur = 0;
        for (int i = 0; i < k; ++i)
            mn[0][i] = inf, mx[0][i] = -inf;
    }
    inline int newnode(point pt) {
        int x = ++cur;
        sz[x] = 1;
        ch[x][0] = ch[x][1] = 0;
        mn[x] = mx[x] = val[x] = pt;
        mark[x] = real[x] = 1; //如果是静态 KD-Tree 则不需要记录 mark 标记和 real 标记
        return x;
    }
    inline void pushup(int x) {
        sz[x] = sz[ch[x][0]] + sz[ch[x][1]] + 1;
        real[x] = real[ch[x][0]] + real[ch[x][1]] + mark[x];
        for (int i = 0; i < k; ++i) {
            mn[x][i] = min(val[x][i], min(mn[ch[x][0]][i], mn[ch[x][1]][i]));
            mx[x][i] = max(val[x][i], max(mx[ch[x][0]][i], mx[ch[x][1]][i]));
        }
        //要么特判不合并 0 号结点的值，要么将 0 号结点的值设为极限值
    }
    inline void pushdown(int x) {
        //如果 pushdown 会将标记传给 0 号结点的话，则要在 pushup 中要判断不能合并 0 号结点的值
    }
```

```
//A 序列是已经用 newnode 分配好的结点序列，参数区间 [L, R] 是闭区间
template<int d = 0> int build(int L, int R, int* A) {
    if (L > R) return 0;
    int mid = (L + R) >> 1;
    nth_element(A + L, A + mid, A + R + 1, [](int x, int y) {
        return val[x][d] < val[y][d];
        });
    int x = A[mid];
    ch[x][0] = build<(d + 1) % k>(L, mid - 1, A);
    ch[x][1] = build<(d + 1) % k>(mid + 1, R, A);
    pushup(x);
    return x;
}
inline bool in(int x, const point& lower, const point& upper) { //x in [lower, upper]
    for (int i = 0; i < k; ++i)
        if (mn[x][i] < lower[i] || mx[x][i] > upper[i])
            return false;
    return true;
}
inline bool out(int x, const point& lower, const point& upper) { //x out of [lower, upper]
    for (int i = 0; i < k; ++i)
        if (mn[x][i] > upper[i] || mx[x][i] < lower[i])
            return true;
    return false;
}
inline bool contain(int x, const point& lower, const point& upper) { //val[x] in [lower,
↪ upper]
    for (int i = 0; i < k; ++i)
        if (val[x][i] < lower[i] || val[x][i] > upper[i])
            return false;
    return true;
}
//-----------------区间查询和修改-------------------------------
int query(int x, const point& lower, const point& upper) {
    if (x == 0 || out(x, lower, upper))
        return 0;
    if (in(x, lower, upper))
        return sz[x];
    pushdown(x);
    return contain(x, lower, upper) + query(ch[x][0], lower, upper) + query(ch[x][1], lower,
↪ upper);
}
void update(int x, const point& lower, const point& upper) {
    if (x == 0 || out(x, lower, upper))
        return;
    if (in(x, lower, upper)) {
```

```
        ; //对区间打上标记
        return;
    }
    if (contain(x, lower, upper))
        ; //直接修改当前结点的值
    pushdown(x);
    update(ch[x][0], lower, upper);
    update(ch[x][1], lower, upper);
    pushup(x);
}
//----------------------圆（球）形查询--------------------------
inline bool contain(int x, const point& center, long long r) { //val[x] 是否在以 center 为圆
↪ 心半径为 r 的圆里
    long long dist = 0;
    for (int i = 0; i < k; ++i)
        dist += sqr(val[x][i] - center[i]);
    return dist <= r * r;
}
int query(int x, const point& center, long long r) {
    if (x == 0 || gmin(x, center) > r * r) //当前区域与查询圆没有交
        return 0;
    if (gmax(x, center) <= r * r) //当前区域在圆内
        return sz[x];
    pushdown(x);
    return contain(x, center, r) + query(ch[x][0], center, r) + query(ch[x][1], center, r);
}
//-----------------查询与 a 曼哈顿距离最近和最远的点--------------
inline int manhattan(const point& a, const point& b) {
    int dist = 0;
    for (int i = 0; i < k; ++i)
        dist += abs(a[i] - b[i]);
    return dist;
}
inline int fmin(int x, const point& a) {
    int ret = 0;
    for (int i = 0; i < k; ++i) //如果坐标范围可以到 2e9，下面的加法会溢出
        ret += max(mn[x][i] - a[i], 0) + max(a[i] - mx[x][i], 0);
    return ret;
}
inline int fmax(int x, const point& a) {
    int ret = 0;
    for (int i = 0; i < k; ++i) //如果坐标范围可以到 2e9，下面的加法会溢出
        ret += max(abs(mn[x][i] - a[i]), abs(mx[x][i] - a[i]));
    return ret;
}
void querymin(int x, const point& a) { //查询之前应先将 ans 设为无穷大
```

```
    ans = min(ans, manhattan(val[x], a));
    int f[2];
    f[0] = ch[x][0] ? fmin(ch[x][0], a) : inf;
    f[1] = ch[x][1] ? fmin(ch[x][1], a) : inf;
    int d = f[0] >= f[1];
    if (f[d] < ans)
        querymin(ch[x][d], a);
    if (f[!d] < ans)
        querymin(ch[x][!d], a);
}
void querymax(int x, const point& a) { //查询之前应先将 ans 设为无穷小
    ans = max(ans, manhattan(val[x], a));
    int f[2];
    f[0] = ch[x][0] ? fmax(ch[x][0], a) : -inf;
    f[1] = ch[x][1] ? fmax(ch[x][1], a) : -inf;
    int d = f[0] <= f[1];
    if (f[d] > ans)
        querymax(ch[x][d], a);
    if (f[!d] > ans)
        querymax(ch[x][!d], a);
}
//-----------------查询与 a 欧几里得距离最近和最远的点------------
//如果要查询欧几里得距离，应当将 point 的类型改为 array<long long, k>，同时调大 inf 的值
inline long long euclid(const point& a, const point& b) {
    long long dist = 0;
    //如果坐标范围可以到 2e9 且 k>=3，则 dist 就会超出 long long 的表示范围
    //可以考虑用 double 或 long double 存储
    for (int i = 0; i < k; ++i)
        dist += sqr((long long)a[i] - b[i]);
    return dist;
}
inline long long gmin(int x, const point& a) { //点 a 到 x 表示区域的最近欧氏距离
    long long ret = 0;
    for (int i = 0; i < k; ++i) //可能超出 long long 的表示范围
        ret += sqr(max(mn[x][i] - a[i], 0)) + sqr(max(a[i] - mx[x][i], 0));
    return ret;
}
inline long long gmax(int x, const point& a) { //点 a 到 x 表示区域的最远欧氏距离
    long long ret = 0;
    for (int i = 0; i < k; ++i) //可能超出 long long 的表示范围
        ret += max(sqr(mn[x][i] - a[i]), sqr(mx[x][i] - a[i]));
    return ret;
}
//注意 ans 保存的是距离的平方
void findmin(int x, const point& a) { //查询之前应先将 ans 设为无穷大
    //此处乘 1LL，只用于与前面的模板同步，没有实际意义
```

```
        ans = min(1LL * ans, euclid(val[x], a)); //ans 应当用 long long 或 double 存储
        long long f[2];
        f[0] = ch[x][0] ? gmin(ch[x][0], a) : inf;
        f[1] = ch[x][1] ? gmin(ch[x][1], a) : inf;
        int d = f[0] >= f[1];
        if (f[d] < ans)
            findmin(ch[x][d], a);
        if (f[!d] < ans)
            findmin(ch[x][!d], a);
    }
    void findmax(int x, const point& a) { //查询之前应先将 ans 设为无穷小
        ans = max(1LL * ans, euclid(val[x], a)); //ans 应当用 long long 或 double 存储
        long long f[2];
        f[0] = ch[x][0] ? gmax(ch[x][0], a) : -inf;
        f[1] = ch[x][1] ? gmax(ch[x][1], a) : -inf;
        int d = f[0] <= f[1];
        if (f[d] > ans)
            findmax(ch[x][d], a);
        if (f[!d] > ans)
            findmax(ch[x][!d], a);
    }
    //-----------------动态 KD-Tree: 插入删除操作---------------------
    //考虑到算法整体的时间复杂度瓶颈不在插入删除上，此处直接重用了前面的 build 函数来重构，
    //单次时间复杂度 O(nlogn)，可以通过直接取中间的位置优化为单次 O(n)
    const double alpha = 0.6;
    int* pos, length, arr[maxn];
    decltype(build<0>)* func;
    void dfs(int x) {
        if (!x) return;
        pushdown(x);
        dfs(ch[x][0]);
        if (mark[x]) //被标记为删除的点不参与重构，若没有删除则直接执行下面的操作，不需要 if 判断
            arr[++length] = x; //如果结点还有权值等属性的话也要一起参与重构
        dfs(ch[x][1]);
    }
    template<int d = 0> void add(int& x, point a) {
        if (!x) {
            x = newnode(a);
            return;
        }
        pushdown(x);
        if (a[d] < val[x][d])
            add<(d + 1) % k>(ch[x][0], a);
        else
            add<(d + 1) % k>(ch[x][1], a);
        pushup(x);
```

```
        if (sz[ch[x][0]] > sz[x] * alpha || sz[ch[x][1]] > sz[x] * alpha)
            pos = &x, func = build<d>;
    }
    void insert(int& x, point a) { //应当保证没有重复的点
        pos = nullptr;
        add(x, a);
        if (pos) {
            length = 0;
            dfs(*pos);
            *pos = func(1, length, arr);
        }
    }
    template<int d = 0> void del(int& x, point a) {
        if (!x)
            return;
        pushdown(x);
        if (a == val[x])
            mark[x] = 0; //如果还有其他标记的话也要一起清空
        else if (a[d] < val[x][d])
            del<(d + 1) % k>(ch[x][0], a);
        else
            del<(d + 1) % k>(ch[x][1], a);
        pushup(x);
    }
    void remove(int& x, point a) { //注意在查询操作的时候不要考虑已经被删掉的点
        del(x, a);
        if (real[x] >= sz[x] / 2) { //超过一半的结点被删除则重建树
            length = 0;
            dfs(x);
            x = func(1, length, arr);
        }
    }
};
point a[maxn];
int node[maxn];
int main() {
    freopen("in.txt", "r", stdin);
    kdt::init();
    int n;
    scanf("%d", &n);
    for (int i = 0; i < n; ++i)
        scanf("%d %d", &a[i][0], &a[i][1]);
    for (int i = 0; i < n; ++i)
        node[i] = kdt::newnode(a[i]);
    int root = kdt::build(0, n, node);
    return 0;
```

```
}
```

9.29 Euler-Tour-Tree

```
const int maxn = 310000; //maxn 至少为结点数 + 操作数的两倍
namespace ett {
    //--------------------------Splay Tree------------------------------
    //若要修改一个点的点权，应当先将其 splay 到根，然后修改，最后还要调用 pushup 维护。
    //调用完 splay 之后根结点会改变，应该用 splay 的返回值更新根结点。
    int ch[maxn][2], fa[maxn], stk[maxn], sum[maxn], key[maxn], cur;
    int alloc(int val) {
        int x = ++cur;
        ch[x][0] = ch[x][1] = fa[x] = 0;
        sum[x] = key[x] = val;
        return x;
    }
    inline bool son(int x) {
        return ch[fa[x]][1] == x;
    }
    inline void pushup(int x) {
        sum[x] = sum[ch[x][0]] + sum[ch[x][1]] + key[x];
    }
    inline void pushdown(int x) {}
    void rotate(int x) {
        int y = fa[x], z = fa[y], c = son(x);
        if (fa[y])
            ch[z][son(y)] = x;
        fa[x] = z;
        ch[y][c] = ch[x][!c];
        fa[ch[y][c]] = y;
        ch[x][!c] = y;
        fa[y] = x;
        pushup(y);
    }
    void ascend(int x) {
        for (int y = fa[x]; y; rotate(x), y = fa[x]) if (fa[y])
            son(x) ^ son(y) ? rotate(x) : rotate(y);
        pushup(x);
    }
    int splay(int x) { //没有 pushdown 操作时，可以直接用 ascend 替换 splay
        int top = 0;
        for (int i = x; i; i = fa[i])
            stk[++top] = i;
        while (top)
            pushdown(stk[top--]);
        ascend(x);
        return x;
```

```
}
template<typename ...T> int merge(int x, int y, T... args) {
    if constexpr (sizeof...(args) == 0) {
        if (x == 0) return y;
        while (pushdown(x), ch[x][1])
            x = ch[x][1];
        ascend(x);
        ch[x][1] = y; fa[y] = x;
        pushup(x);
        return x;
    }
    else {
        return merge(merge(x, y), args...);
    }
}
inline int split(int x, int d) {
    int y = ch[x][d];
    fa[y] = ch[x][d] = 0;
    pushup(x);
    return y;
}
//--------------------------Euler Tour Tree----------------------------
int idx;
unordered_map<long long, int> locate;
#define edge(x, y) (((1LL * (x)) << 30) + (y))
void init() {
    cur = idx = 0;
    locate.clear();
}
inline int newnode(int val) {
    int x = ++idx;
    locate[edge(x, x)] = alloc(val);
    return x;
}
inline int newedge(int x, int y) {
    return locate[edge(x, y)] = alloc(0);
}
inline int changeroot(int x) {
    int a = locate[edge(x, x)];
    splay(a);
    int b = split(a, 0);
    return merge(a, b);
}
inline int gettree(int x) {
    return splay(locate[edge(x, x)]);
}
```

```
    inline bool sametree(int x, int y) {
        if (x == y) return true;
        auto a = locate[edge(x, x)];
        auto b = locate[edge(y, y)];
        splay(a);
        splay(b);
        return fa[a] != 0;
    }
    void link(int x, int y) {
        auto a = newedge(x, y);
        auto b = newedge(y, x);
        merge(changeroot(x), a, changeroot(y), b);
    }
    void cut(int x, int y) {
        auto a = locate[edge(x, y)];
        auto b = locate[edge(y, x)];
        splay(a);
        auto i = split(a, 0), j = split(a, 1);
        splay(b);
        bool flag = (i != 0 && fa[i] != 0) || b == i;
        auto L = split(b, 0), R = split(b, 1);
        if (flag)
            merge(L, j);
        else
            merge(i, R);
    }
}
using namespace ett;
int node[maxn];
//访问结点 node[x] 所在树的权值和: sum[gettree(node[x])]
//修改结点 node[x] 的权值为 v: int a = gettree(node[x]); key[a] = v; pushup(a);
int main() { //洛谷 P4219
    //freopen("in.txt", "r", stdin);
    int n, q;
    scanf("%d %d", &n, &q);
    init();
    for (int i = 1; i <= n; ++i)
        node[i] = newnode(1);
    for (int i = 0; i < q; ++i) {
        char tp;
        int x, y;
        scanf(" %c %d %d", &tp, &x, &y);
        if (tp == 'A') {
            link(node[x], node[y]);
        }
        else {
```

```
            cut(node[x], node[y]);
            long long a = sum[gettree(node[x])];
            long long b = sum[gettree(node[y])];
            printf("%lld\n", a * b);
            link(node[x], node[y]);
        }
    }
    return 0;
}
```

9.30 rope

```
/*
// #include <ext/rope>
// using namespace __gnu_cxx;
rope<int> r;
1. r.push_back(a) 往后插入 a
   r.pop_back() 删除最后一个元素
2. r.insert(idx, a) 在下标 idx 处插入 a
   r.insert(idx, rope) 在下标 idx 处插入 rope 对象，时间复杂度 O(log(n))
3. r.erase(idx, length) 删除从 idx 开始的 length 个字符
4. r.replace(idx, a) 将下标 idx 的位置设为 a
5. r.substr(pos, length) 返回从 pos 开始长度为 length 的子串，时间复杂度 O(log(n)),
   修改 substr 返回的 rope 不会改变原 rope。
6. 支持基本的字符串运算比如 ==、+=、<，拼接的时间复杂度为 O(log(n))
7. rope 的拷贝是 O(1) 的，修改是 O(log(n)) 的。
8. 注意 r[i] 返回值是左值，不能修改。修改需调用 replace 函数。
*/
using namespace __gnu_cxx;
const int maxn = 110000;
rope<int> version[maxn];
int main() {
    const int n = 100000;
    for (int i = 0; i < n; ++i)
        version[0].push_back(i);
    for (int i = 1; i < n; ++i) {
        version[i] = version[i - 1];
        version[i].replace(i, i * 10);
    }
    const int pos = 88888;
    printf("%d %d %d\n", version[pos][pos], version[pos][pos + 1], version[pos + 1][pos + 1]);
    rope<int> r;
    r.push_back(0); r.push_back(1);
    for (int i = 0; i < 32; ++i)
        r += r;
    int total = 0;
    for (int i = 0; i < n; ++i)
```

```
        total += r[998244353 + i];
    printf("%d\n", total);
    return 0;
}
```

9.31 pb-ds 平衡树

```
/*
// #include <ext/pb_ds/assoc_container.hpp>
// #include <ext/pb_ds/tree_policy.hpp>
// using namespace __gnu_pbds;
tree<int, null_type, less<int>, rb_tree_tag, tree_order_statistics_node_update> t;
1. 第二个模板参数为值域的类型，如果只要实现 set 的功能，则用 null_type 即可
2. 基本操作与 std::map/std::set 兼容
3. t.erase(val) 从平衡树中删除值为 val 的结点
4. t.order_of_key(val) 返回平衡树中比 val 小的数有多少个
5. t.find_by_order(k) 返回排名为 k 的位置的迭代器（排名从 0 开始）
6. 不支持多重值，如果需要多重值，可以再开一个 unordered_map 来记录值出现的次数。
   将 val<<32 后加上出现的次数后插入 tree，注意此时应该为 long long 类型。
*/
using namespace __gnu_pbds;
const int maxn = 110000;
tree<int, null_type, less<int>, rb_tree_tag, tree_order_statistics_node_update> t;
int main() {
    for (int i = 1; i <= 5; ++i)
        t.insert(i);
    printf("=>%d\n", *t.find_by_order(2));
    return 0;
}
```

10 数学算法

10.1 SG 函数

```
//注意：异或运算符的优先级小于比较运算符!!!
const int maxn = 1000;
int sg[maxn], vis[maxn];
void getSG()
{
    memset(sg, 0, sizeof(sg));
    for (int i = 1; i < maxn; ++i)
    {
        memset(vis, 0, sizeof(vis));
        for (int j = 1; j <= i; ++j)
            vis[sg[i - j]] = true;
        for (int j = 0; j < maxn; ++j)
        {
```

```
            if (!vis[j])
            {
                sg[i] = j;
                break;
            }
        }
    }
}
int main()
{
    getSG();
    printf("%d\n", sg[5]);
    return 0;
}
```

10.2 自适应辛普森积分

```
double f(double x) { //任意一个自定义的函数
    return x * x;
}
double simpson(double l, double r) { //计算函数 f 在区间 [l, r] 上的辛普森积分
    return (f(l) + f(r) + 4 * f((l + r) / 2)) * (r - l) / 6;
}
double solve(double l, double r, double eps, double val) { //递归求解积分
    double mid = (l + r) / 2;
    double L = simpson(l, mid), R = simpson(mid, r);
    if (fabs(L + R - val) <= 15 * eps)
        return L + R + (L + R - val) / 15;
    return solve(l, mid, eps / 2, L) + solve(mid, r, eps / 2, R);
}
double asme(double l, double r, double eps = 1e-7) { //自适应辛普森积分
    return solve(l, r, eps, simpson(l, r));            //求函数 f 在区间 [l, r] 上的自适应辛普森
↪   积分，eps 指定精度
}
int main() {
    printf("%.15f\n", asme(0, 1)); //ans = 0.333333333333333
    return 0;
}
```

10.3 高斯消元

```
const int maxn = 100;
using matrix = double[maxn][maxn];
using vect = array<double, maxn>;
//当方程有唯一解的时候，算出唯一解
//矩阵 A 的大小为 n * (n + 1)
void gauss_elimination(matrix A, int n) {
```

```
    for (int i = 0; i < n; ++i) {
        int r = i;
        for (int j = i + 1; j < n; ++j)
            if (fabs(A[j][i]) > fabs(A[r][i]))
                r = j;
        if (r != i) for (int j = 0; j <= n; ++j)
            swap(A[r][j], A[i][j]);
        for (int k = i + 1; k < n; ++k)
            for (int j = n; j >= i; --j)
                A[k][j] -= A[k][i] / A[i][i] * A[i][j];
    }
    for (int i = n - 1; i >= 0; --i) {
        for (int j = i + 1; j < n; ++j)
            A[i][n] -= A[j][n] * A[i][j];
        A[i][n] /= A[i][i];
    }
}
//无解返回-1，有唯一解返回 0，有无穷多解返回 1。
//在有解的情况下通过 ans 返回任意一个解。
//矩阵 A 的大小为 n * (m + 1)，表示有 n 个方程，m 个变量。
const double eps = 1e-8;
int row[maxn], var[maxn];
int one_possible(matrix A, int n, int m, vect& ans) {
    memset(row, -1, sizeof(row));
    int r = 0;
    for (int c = 0; c < m && r < n; ++c) {
        int x = r;
        for (int i = x + 1; i < n; ++i)
            if (fabs(A[i][c]) > fabs(A[x][c]))
                x = i;
        if (x != r) for (int j = 0; j <= m; ++j)
            swap(A[x][j], A[r][j]);
        if (fabs(A[r][c]) < eps)
            continue;
        for (int k = r + 1; k < n; ++k)
            for (int j = m; j >= c; --j)
                A[k][j] -= A[k][c] / A[r][c] * A[r][j];
        row[c] = r++;
    }
    for (int i = r; i < n; ++i) if (fabs(A[i][m]) > eps)
        return -1;
    for (int c = m - 1; c >= 0; --c) {
        int x = row[c];
        if (x < 0)
            ans[c] = 0;
        else {
```

```
            for (int i = x - 1; i >= 0; --i)
                A[i][m] -= A[i][c] / A[x][c] * A[x][m];
            ans[c] = A[x][m] / A[x][c];
        }
    }
    return r < m;
}
//计算方程的最小二乘解
void least_square(matrix A, int n, int m, vect& ans) {
    static matrix T;
    for (int i = 0; i < n; ++i)
        for (int j = 0; j <= m; ++j)
            T[i][j] = A[i][j];
    for (int i = 0; i < m; ++i) {
        for (int j = 0; j <= m; ++j) {
            A[i][j] = 0;
            for (int k = 0; k < n; ++k)
                A[i][j] += T[k][i] * T[k][j];
        }
    }
    one_possible(A, m, m, ans);
}
//将矩阵 A 化为简化阶梯形
//ans 为方程组的一个特解, basis 为齐次方程组解空间的一组基。
//该算法针对稀疏矩阵进行了优化，在矩阵中有大量 0 元素时，时间复杂度会小于 O(n^3)。
int row_simplify(matrix A, int n, int m, vect& ans, vector<vect>& basis) {
    memset(row, -1, sizeof(row));
    int r = 0;
    for (int c = 0; c < m && r < n; ++c) {
        int x = r;
        for (int i = x + 1; i < n; ++i)
            if (fabs(A[i][c]) > fabs(A[x][c]))
                x = i;
        if (x != r) for (int j = 0; j <= m; ++j)
            swap(A[x][j], A[r][j]);
        if (fabs(A[r][c]) < eps)
            continue;
        for (int j = m; j >= c; --j)
            A[r][j] /= A[r][c];
        for (int k = r + 1; k < n; ++k) if (fabs(A[k][c]) > eps)
            for (int j = m; j >= c; --j)
                A[k][j] -= A[k][c] * A[r][j];
        var[r] = c;
        row[c] = r++;
    }
    for (int i = r; i < n; ++i) if (fabs(A[i][m]) > eps)
```

```
        return -1;
    for (int c = m - 1; c >= 0; --c) {
        int x = row[c];
        if (x < 0)
            ans[c] = 0;
        else {
            for (int i = x - 1; i >= 0; --i) if (fabs(A[i][c]) > eps)
                for (int j = m; j >= c; --j)
                    A[i][j] -= A[i][c] * A[x][j];
            ans[c] = A[x][m];
        }
    }
    //求出基础解系
    for (int c = m - 1; c >= 0; --c) if (row[c] < 0) {
        vect now = {};
        for (int i = 0; i < r; ++i)
            now[var[i]] = -A[i][c];
        now[c] = 1;
        basis.push_back(now);
    }
    return r < m;
}
namespace rectangle_mod {
    const int mod = 998244353;
    using matrix = int[maxn][maxn];
    inline int pow_mod(int a, int n) {
        int ans = 1;
        while (n) {
            if (n & 1)
                ans = 1LL * ans * a % mod;
            n >>= 1;
            a = 1LL * a * a % mod;
        }
        return ans;
    }
    inline int inv(int n) {
        return pow_mod(n, mod - 2);
    }
    //对 n 行 m 列的矩阵进行取模意义下的高斯消元，必须保证矩阵行满秩
    inline void gauss(matrix A, int n, int m) {
        for (int i = 0; i < n; ++i) {
            int r = i;
            for (int j = i; j < n; ++j) {
                if (A[j][i] != 0) {
                    r = j;
                    break;
```

```
            }
        }
        if (r != i) for (int j = 0; j < m; ++j)
            swap(A[r][j], A[i][j]);
        int pivot = inv(A[i][i]);
        for (int k = i + 1; k < n; ++k) {
            int val = 1LL * A[k][i] * pivot % mod;
            for (int j = m - 1; j >= i; --j)
                A[k][j] = (A[k][j] - 1LL * A[i][j] * val % mod + mod) % mod;
        }
    }
    for (int i = n - 1; i >= 0; --i) {
        for (int j = i + 1; j < n; ++j) {
            for (int k = n; k < m; ++k)
                A[i][k] = (A[i][k] - 1LL * A[j][k] * A[i][j] % mod + mod) % mod;
        }
        int pivot = inv(A[i][i]);
        for (int k = n; k < m; ++k)
            A[i][k] = 1LL * A[i][k] * pivot % mod;
    }
  }
}
int main() { //3 -5 -2
    matrix A = {
        {1, 1, 0, 0, -3},
        {1, 1, 0, 0, -1},
        {1, 0, 1, 0, 0},
        {1, 0, 1, 0, 2},
        {1, 0, 0, 1, 5},
        {1, 0, 0, 1, 1}
    };
    vect ans;
    vector<vect> basis;
    least_square(A, 6, 4, ans);
    for (int i = 0; i < 4; ++i)
        printf("%f\n", ans[i]);
    printf("\n");
    return 0;
}
```

10.4 稀疏矩阵的高斯消元

```
const int maxn = 110000;
const long long mod = 1e9 + 7;
const double eps = 1e-9;
long long pow_mod(long long a, long long n, long long p) {
    long long ans = 1;
```

```
    while (n) {
        if (n & 1)
            ans = ans * a % p;
        a = a * a % p;
        n >>= 1;
    }
    return ans;
}
long long inv(long long a) {
    return pow_mod(a, mod - 2, mod);
}
struct fast_gauss_mod { //基于十字链表的高斯消元
    static const int maxstate = 3000000;
    static const int table_size = 5110007;
    struct {
        long long val;
        int r, c, next, right, down;
    } node[maxstate];
    int first[table_size], row[maxn], column[maxn], vis[maxn], length[maxn], sz, n;
    void init(int n) { //待消元的矩阵大小是 n * (n + 1) 的
        this->n = n;
        sz = 0;
        memset(first, 0, sizeof(first));
        memset(row, 0, sizeof(row));
        memset(column, 0, sizeof(column));
        memset(vis, 0, sizeof(vis));
        memset(length, 0, sizeof(length));
    }
    long long& A(int r, int c) {
        const int h = ((long long)r << 20 | c) % table_size;
        for (int i = first[h]; i; i = node[i].next)
            if (node[i].r == r && node[i].c == c)
                return node[i].val;
        int i = ++sz;
        node[i].next = first[h];
        first[h] = i;
        node[i].right = row[r];
        row[r] = i;
        node[i].down = column[c];
        column[c] = i;
        node[i].r = r;
        node[i].c = c;
        node[i].val = 0;
        length[r] += 1;
        return node[i].val;
    }
```

```
void insert(int r, int c, long long v) { //在矩阵的第 r 行第 c 列填上 v
    A(r, c) = v;
}
vector<long long> solve() { //无解或用无穷解时返回空 vector
    vector<long long> ans(n);
    for (int i = 0; i < n; ++i) {
        int r = -1;
        for (int x = column[i]; x; x = node[x].down) {
            int j = node[x].r;
            if (!vis[j] && node[x].val != 0)
                if (r < 0 || length[j] < length[r])
                    r = j;
        }
        if (r == -1)
            return {};
        int* last = &row[r];
        for (int y = row[r]; y; y = node[y].right) {
            if (node[y].val == 0)
                *last = node[y].right;
            else
                last = &node[y].right;
        }
        auto pivot = mod - inv(A(r, i));
        for (int x = column[i]; x; x = node[x].down) {
            int j = node[x].r;
            if (!vis[j] && j != r) {
                auto ratio = node[x].val * pivot % mod;
                for (int y = row[r]; y; y = node[y].right) {
                    int k = node[y].c;
                    A(j, k) = (A(j, k) + ratio * node[y].val) % mod;
                }
                length[j] -= 1;
            }
        }
        ans[i] = r;
        vis[r] = true;
    }
    for (int i = n - 1; i >= 0; --i) {
        int r = ans[i];
        auto pivot = (mod - A(r, n)) * inv(A(r, i)) % mod;
        for (int x = column[i]; x; x = node[x].down) {
            int j = node[x].r;
            if (j != r)
                A(j, n) = (A(j, n) + node[x].val * pivot) % mod;
        }
    }
```

```
        for (int i = 0; i < n; ++i)
            ans[i] = A(ans[i], n) * inv(A(ans[i], i)) % mod;
        return ans;
    }
}gauss;
struct fast_gauss_double {
    static const int maxstate = 2000000;
    static const int table_size = 5110007;
    struct {
        double val;
        int r, c, next, right, down;
    } node[maxstate];
    int first[table_size], row[maxn], column[maxn], vis[maxn], length[maxn], sz, n;
    void init(int n) { //待消元的矩阵大小是 n * (n + 1) 的
        this->n = n;
        sz = 0;
        memset(first, 0, sizeof(first));
        memset(row, 0, sizeof(row));
        memset(column, 0, sizeof(column));
        memset(vis, 0, sizeof(vis));
        memset(length, 0, sizeof(length));
    }
    double& A(int r, int c) {
        const int h = ((long long)r << 20 | c) % table_size;
        for (int i = first[h]; i; i = node[i].next)
            if (node[i].r == r && node[i].c == c)
                return node[i].val;
        int i = ++sz;
        node[i].next = first[h];
        first[h] = i;
        node[i].right = row[r];
        row[r] = i;
        node[i].down = column[c];
        column[c] = i;
        node[i].r = r;
        node[i].c = c;
        node[i].val = 0;
        length[r] += 1;
        return node[i].val;
    }
    void insert(int r, int c, double v) { //在矩阵的第 r 行第 c 列填上 v
        A(r, c) = v;
    }
    vector<double> solve() { //无解或用无穷解时返回空 vector
        vector<double> ans(n);
        vector<int> res(n);
```

```
        for (int i = 0; i < n; ++i) {
            int r = -1;
            for (int x = column[i]; x; x = node[x].down) {
                int j = node[x].r;
                if (!vis[j] && fabs(node[x].val) > eps)
                    if (r < 0 || length[j] < length[r])
                        r = j;
            }
            if (r == -1)
                return {};
            int* last = &row[r];
            for (int y = row[r]; y; y = node[y].right) {
                if (fabs(node[y].val) < eps)
                    *last = node[y].right;
                else
                    last = &node[y].right;
            }
            auto pivot = -A(r, i);
            for (int x = column[i]; x; x = node[x].down) {
                int j = node[x].r;
                if (!vis[j] && j != r) {
                    auto ratio = node[x].val / pivot;
                    for (int y = row[r]; y; y = node[y].right) {
                        int k = node[y].c;
                        A(j, k) += ratio * node[y].val;
                    }
                    length[j] -= 1;
                }
            }
            res[i] = r;
            vis[r] = true;
        }
        for (int i = n - 1; i >= 0; --i) {
            int r = res[i];
            auto pivot = -A(r, n) / A(r, i);
            for (int x = column[i]; x; x = node[x].down) {
                int j = node[x].r;
                if (j != r)
                    A(j, n) += node[x].val * pivot;
            }
        }
        for (int i = 0; i < n; ++i)
            ans[i] = A(res[i], n) / A(res[i], i);
        return ans;
    }
}gas;
```

```
namespace gauss_mod { //稀疏矩阵时，复杂度接近 O(n^2)
    const int maxn = 1000;
    using matrix = int[maxn][maxn];
    inline int pow_mod(int a, int n) {
        int ans = 1;
        while (n) {
            if (n & 1)
                ans = 1LL * ans * a % mod;
            n >>= 1;
            a = 1LL * a * a % mod;
        }
        return ans;
    }
    inline int inv(int n) {
        return pow_mod(n, mod - 2);
    }
    inline void gauss(matrix A, int n) {
        int column[maxn];
        for (int i = 0; i < n; ++i) {
            int r = i;
            for (int j = i; j < n; ++j) {
                if (A[j][i] != 0) {
                    r = j;
                    break;
                }
            }
            if (r != i) for (int j = 0; j <= n; ++j)
                swap(A[r][j], A[i][j]);
            int pivot = inv(A[i][i]), sz = 0;
            for (int j = n; j >= i; --j) if (A[i][j])
                column[sz++] = j;
            for (int k = i + 1; k < n; ++k) if (A[k][i]) {
                int val = 1LL * A[k][i] * pivot % mod;
                for (int t = 0; t < sz; ++t) {
                    int j = column[t];
                    A[k][j] = (A[k][j] - 1LL * A[i][j] * val % mod + mod) % mod;
                }
            }
        }
        for (int i = n - 1; i >= 0; --i) {
            for (int j = i + 1; j < n; ++j) if (A[i][j]) {
                A[i][n] = (A[i][n] - 1LL * A[j][n] * A[i][j] % mod + mod) % mod;
            }
            A[i][n] = 1LL * A[i][n] * inv(A[i][i]) % mod;
        }
    }
```

```
}
int main() {
    int n;
    scanf("%d", &n);
    gas.init(n);
    for (int i = 0; i < n; ++i) {
        for (int v, j = 0; j <= n; ++j) {
            scanf("%d", &v);
            if (v) gas.insert(i, j, v);
        }
    }
    auto ans = gas.solve();
    if (ans.empty()) {
        puts("No Solution");
        return 0;
    }
    for (int i = 0; i < n; ++i)
        printf("%.2f\n", ans[i]);
    return 0;
}
```

10.5 求解异或方程组

```
const int maxn = 110;
bitset<maxn> a[maxn];
void Gauss(int n) {
    int now = 0; //now 记录当前正在处理的行号
    for (int i = 0; i < n; ++i) { //i 记录当前正在处理的列号
        int j = now;
        while (j < n && !a[j][i]) ++j;
        if (j == n) continue;
        if (j != now) swap(a[now], a[j]);
        for (int k = 0; k < n; ++k)
            if (k != now && a[k][i])
                a[k] ^= a[now];
        ++now;
    }
}
int main() {
    return 0;
}
```

10.6 矩阵与状态转移

```
/*
对于状态图中的每一条边 (i, j)，设置矩阵 A[i][j] = 1。
令 B = pow(A, n)，
```

```
则从 i 开始走 n 步到达 j 的方案数为 B[i][j]。
*/
const int maxn = 100;
int n;
struct matrix
{
    int data[maxn][maxn];
    matrix() : data(){}
    int *operator[] (int idx)
    {
        return data[idx];
    }
};
matrix operator*(matrix &A, matrix &B)
{
    matrix C;
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
            for (int k = 0; k < n; ++k)
                C[i][j] += A[i][k] * B[k][j];
    return C;
}
matrix pow(matrix A, int t)
{
    matrix ans;
    for (int i = 0; i < n; ++i)
        ans[i][i] = 1;
    while (t)
    {
        if (t & 1)
            ans = ans * A;
        t >>= 1;
        A = A * A;
    }
    return ans;
}
int main()
{
    return 0;
}
```

10.7 递推式求解

```
const int mod = 1e9 + 7;
int a[2100], c[2100], tmp[2100], tmpa[2100], res[2100], k;
long long n;
void mul(int* p, int* q, int* c)
```

```
{
    memset(tmp, 0, sizeof(tmp));
    for (int i = 0; i < k; i++)
        for (int j = 0; j < k; j++)
            tmp[i + j] = (1ll * p[i] * q[j] + tmp[i + j]) % mod;
    for (int i = 2 * k - 2; i >= k; i--)
        for (int j = 0; j < k; j++)
            tmp[i - j - 1] = (1ll * tmp[i] * c[j] + tmp[i - j - 1]) % mod;
    for (int i = 0; i < k; i++)
        p[i] = tmp[i];
}
int solve(int* a, int* c, long long n) //a 为初始值数组，c 为系数矩阵
{
    if (n < k) return a[n];
    memset(tmpa, 0, sizeof(tmpa));
    memset(res, 0, sizeof(res));
    tmpa[1] = res[0] = 1;
    while (n)
    {
        if (n & 1)
            mul(res, tmpa, c);
        mul(tmpa, tmpa, c);
        n >>= 1;
    }
    int ans = 0;
    for (int i = 0; i < k; i++)
        ans = (ans + 1ll * a[i] * res[i]) % mod;
    return ans;
}
int pow_mod(int a, long long N) {
    int ans = 1;
    a %= mod;
    while (N) {
        if (N & 1)
            ans = 1ll * ans * a % mod;
        a = 1ll * a * a % mod;
        N >>= 1;
    }
    return ans;
}
int main() {
    int T;
    scanf("%d", &T);
    while (T--) {
        scanf("%d %lld", &k, &n);
        memset(a, 0, sizeof(a));
```

```
        if (k == 1) {
            printf("1\n");
            continue;
        }
        if (n == -1) {
            printf("%d\n", 2 * pow_mod(k + 1, mod - 2) % mod);
            continue;
        }
        int rev = pow_mod(k, mod - 2);
        for (int i = 0; i < k; ++i)
            c[i] = rev;
        a[0] = 1;
        for (int i = 1; i < k; ++i) {
            for (int j = i - k; j < i; ++j) if (j >= 0) {
                a[i] += 1ll * rev * a[j] % mod;
                if (a[i] >= mod)a[i] -= mod;
            }
        }
        int ans = solve(a, c, n);
        printf("%d\n", ans);
    }
    return 0;
}
```

10.8 快速傅里叶变换

```
const int maxn = 1 << 22; //必须是 2 的幂
const double pi = acos(-1);
complex<double> w[maxn];
void init() {
    for (int i = 0;i < maxn;++i)
        w[i] = { cos(pi * 2 * i / maxn), sin(pi * 2 * i / maxn) };
}
void fft(vector<complex<double>> &a, bool inverse) {
    for (int i = 0, j = 0; i < a.size(); ++i) {
        if (i < j)
            std::swap(a[i], a[j]);
        for (int k = a.size() >> 1; (j ^= k) < k; k >>= 1);
    }
    for (int step = 2; step <= a.size(); step *= 2) {
        int h = step / 2, d = maxn / step;
        for (int i = 0; i < a.size(); i += step)
            for (int j = 0; j < h; ++j) {
                auto &x = a[i + j];
                auto &y = a[i + j + h];
                auto t = w[d * j] * y;
                y = x - t;
```

```
                x = x + t;
            }
    }
    if (inverse) {
        std::reverse(a.begin() + 1, a.end());
        for (auto& x : a)
            x /= a.size();
    }
}
vector<double> operator* (const vector<double> &a, const vector<double> &b) {
    int size = 2;
    while (size < a.size() + b.size()) size <<= 1;
    vector<complex<double>> x(size), y(size);
    copy(a.begin(), a.end(), x.begin());
    copy(b.begin(), b.end(), y.begin());
    fft(x, false);
    fft(y, false);
    for (int i = 0; i < size; i++)
        x[i] *= y[i];
    fft(x, true);
    vector<double> res(a.size() + b.size() - 1);
    for (int i = 0; i < res.size(); ++i)
        res[i] = x[i].real();
    return res;
}
int main() {
    int n, m;
    init();
    scanf("%d %d", &n, &m);
    vector<double> a(n + 1), b(m + 1);
    for (int i = 0; i <= n; ++i)
        scanf("%lf", &a[i]);
    for (int i = 0; i <= m; ++i)
        scanf("%lf", &b[i]);
    auto ans = a * b;
    for (int i = 0; i < ans.size(); ++i)
        printf("%.0f ", ans[i] + 1e-8);
    return 0;
}
```

10.9 快速幂运算

```
const int maxn = 100;
const int mod = 1e9 + 7;
int n;
struct matrix
{
```

```
    int data[maxn][maxn];
    matrix() : data() {}
    int *operator[] (int idx)
    {
        return data[idx];
    }
    const int *operator[] (int idx) const
    {
        return data[idx];
    }
};
matrix operator*(const matrix &A, const matrix &B)
{
    matrix C;
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
            for (int k = 0; k < n; ++k)
                C[i][j] += A[i][k] * B[k][j];
    return C;
}
matrix operator+ (const matrix &A, const matrix &B)
{
    matrix C;
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
            C[i][j] = A[i][j] + B[i][j];
    return C;
}
matrix pow(matrix A, int m)
{
    matrix ans;
    for (int i = 0; i < n; ++i)
        ans[i][i] = 1;
    while (m)
    {
        if (m & 1)
            ans = ans * A;
        A = A * A;
        m >>= 1;
    }
    return ans;
}
long long pow_mod(long long a, int m)
{
    long long ans = 1;
    while (m)
```

```
    {
        if (m & 1)
            ans = ans * a % mod;
        a = a * a % mod;
        m >>= 1;
    }
    return ans;
}
matrix pow_sum(matrix A, int m) //pow_sum(A, m) = sigma pow(A, i)
{                               //               1<=i<=m
    matrix ans, B = A;
    while (m)
    {
        if (m & 1)
            ans = ans * A + B;
        B = B * A + B;
        A = A * A;
        m >>= 1;
    }
    return ans;
}
long long pow_sum(long long a, int m) //pow_sum(A, m) = sigma pow(A, i)
{                               //                     1<=i<=m
    long long ans = 0, b = a;
    while (m)
    {
        if (m & 1)
            ans = (ans * a + b) % mod;
        b = (b * a + b) % mod;
        a = (a * a) % mod;
        m >>= 1;
    }
    return ans;
}
//通过预处理，可以在 O(1) 的时间内求出 x 的 n 次幂
struct Power_Int { //指数在 int 范围内
    static const int base = (1 << 16) - 1;
    long long f[2][base + 1];
    void init(int x) {
        f[0][0] = f[1][0] = 1;
        for (int i = 1; i <= base; ++i)
            f[0][i] = f[0][i - 1] * x % mod;
        f[1][1] = f[0][base] * x % mod;
        for (int i = 2; i <= base; ++i)
            f[1][i] = f[1][i - 1] * f[1][1] % mod;
    }
```

```
    long long pow(int n) {
        return f[0][n & base] * f[1][(n >> 16) & base] % mod;
    }
};
struct Power_LLong { //指数在 long long 范围内
    static const int base = (1 << 16) - 1;
    long long f[4][base + 1];
    void init(int x) {
        for (int i = 0; i < 4; ++i)
            f[i][0] = 1;
        for (int i = 1; i <= base; ++i)
            f[0][i] = f[0][i - 1] * x % mod;
        for (int j = 1; j <= 3; ++j) {
            f[j][1] = f[j - 1][base] * f[j - 1][1] % mod;
            for (int i = 2; i <= base; ++i)
                f[j][i] = f[j][i - 1] * f[j][1] % mod;
        }
    }
    long long pow(long long n) {
        return f[0][n & base] * f[1][(n >> 16) & base] % mod
            * f[2][(n >> 32) & base] % mod * f[3][(n >> 48) & base] % mod;
    }
}power;
int main()
{
    long long ans = pow_sum(2, 10);
    printf("%lld\n", ans);
    n = 2;
    matrix A; A[0][0] = 4; A[0][1] = 2; A[1][1] = 7; A[1][0] = 1;
    matrix B = pow_sum(A, 3);
    A = A * A * A + A * A + A;
    for (int i = 0; i < n; ++i)
    {
        for (int j = 0; j < n; ++j)
        {
            printf("%d ", B[i][j]);
        }
        printf("\n");
    }
    for (int i = 0; i < n; ++i)
    {
        for (int j = 0; j < n; ++j)
        {
            printf("%d ", A[i][j]);
        }
        printf("\n");
```

```
    }
    system("pause");
    return 0;
}
```

10.10 莫比乌斯反演

```
/*
莫比乌斯函数性质:
sigma mu(d) = [n == 1]
 d|n
*/
const int maxn = 1000000;
int vis[maxn], prime[maxn], mu[maxn], cnt;
void init()
{
    memset(vis, 0, sizeof(vis));
    mu[1] = 1;
    cnt = 0;
    for (int i = 2; i < maxn; ++i)
    {
        if (!vis[i])
        {
            prime[cnt++] = i;
            mu[i] = -1;
        }
        for (int j = 0; j < cnt && i * prime[j] < maxn; ++j)
        {
            int t = i * prime[j];
            vis[t] = 1;
            if (i % prime[j] == 0)
            {
                mu[t] = 0;
                break;
            }
            else
            {
                mu[t] = -mu[i];
            }
        }
    }
}
int main()
{
    init();
    printf("%d\n", mu[3*7*11*13]);
    return 0;
```

```
}
```

10.11 逆元

```
const long long maxn = 1000005, mod = 1000000007;
long long pow(long long a, long long n, long long p)
{
    long long ans = 1;
    while (n)
    {
        if (n & 1)
            ans = ans * a % p;
        a = a * a % p;
        n >>= 1;
    }
    return ans;
}
long long inverse1(long long a, long long n)   //费马小定理求逆元
{
    return pow(a, n - 2, n);
}
void extgcd(long long a, long long b, long long& d, long long& x, long long& y)
{
    if (!b) { d = a; x = 1; y = 0; }
    else { extgcd(b, a % b, d, y, x); y -= x * (a / b); }
}
long long inverse2(long long a, long long n)
{
    long long d, x, y;
    extgcd(a, n, d, x, y);
    return d == 1 ? (x + n) % n : -1;
}
long long inv[maxn];
void inverse3(long long n, long long p)
{
    inv[1] = 1;
    for (long long i = 2; i <= n; ++i)
        inv[i] = (p - p / i) * inv[p % i] % p;
}
int main()
{
    int number = 888;
    inverse3(100005, mod);
    printf("%lld %lld %lld\n", inverse1(number, mod), inverse2(number, mod), inv[number]);
    return 0;
}
```

10.12 欧拉函数

```
//phi(n) 表示小于 n 且与 n 互素的整数个数
const int maxn = 1000000;
int vis[maxn], prime[maxn], phi[maxn], cnt;
void init() {
    memset(vis, 0, sizeof(vis));
    phi[1] = 1;
    cnt = 0;
    for (int i = 2; i < maxn; i++) {
        if (!vis[i]) {
            prime[cnt++] = i;
            phi[i] = i - 1;
        }
        for (int j = 0; j < cnt && i * prime[j] < maxn; j++) {
            int t = i * prime[j];
            vis[t] = 1;
            if (i % prime[j] == 0) {
                phi[t] = phi[i] * prime[j];
                break;
            }
            else {
                phi[t] = phi[i] * phi[prime[j]];
            }
        }
    }
}
int euler(int n) { //时间复杂度 O(sqrt(n))
    int ans = n;
    for (int i = 2; i * i <= n; ++i) if (n % i == 0) {
        ans = ans / i * (i - 1);
        while (n % i == 0)
            n /= i;
    }
    if (n > 1)
        ans = ans / n * (n - 1);
    return ans;
}
int main() {
    init();
    for (int i = 0; i <= 10000; ++i) if (phi[i] != euler(i))
        printf("%d\n", i);
    return 0;
}
```

10.13 线性筛素数

```
const int maxn = 1000000;
int vis[maxn], prime[maxn], cnt;
void init()
{
    memset(vis, 0, sizeof(vis));
    cnt = 0;
    for (int i = 2; i < maxn; i++)
    {
        if (!vis[i])
            prime[cnt++] = i;
        for (int j = 0; j < cnt && i * prime[j] < maxn; j++)
        {
            int t = i * prime[j];
            vis[t] = 1;
            if (i % prime[j] == 0)
                break;
        }
    }
}
int main()
{
    init();
    for (int i = 0; i < 10; ++i)
        printf("%d\n", prime[i]);
    return 0;
}
```

10.14 三分求极值

```
#define f(x) fabs(x - 8)
const double eps = 1e-6;
int maximum_int(int L, int R) { //三分求 f 函数的最大值（定义域为整数）
    while (R > L) {
        int m1 = (2 * L + R) / 3;
        int m2 = (2 * R + L + 2) / 3;
        if (f(m1) > f(m2))
            R = m2 - 1;
        else
            L = m1 + 1;
    }
    return L; //f(L) 为最大值
}
int minimun_int(int L, int R) { //三分求 f 函数的最小值（定义域为整数）
    while (R > L) {
        int m1 = (2 * L + R) / 3;
```

```
        int m2 = (2 * R + L + 2) / 3;
        if (f(m1) < f(m2))
            R = m2 - 1;
        else
            L = m1 + 1;
    }
    return L; //f(L) 为最小值
}
double maximum_double(double L, double R) { //三分求 f 函数的最大值（定义域为实数）
    while (R - L > eps) { // for i in range(100):
        double m1 = (2 * L + R) / 3;
        double m2 = (2 * R + L) / 3;
        if (f(m1) > f(m2))
            R = m2;
        else
            L = m1;
    }
    return L; //f(L) 为最大值
}
double minimun_double(double L, double R) { //三分求 f 函数的最小值（定义域为实数）
    while (R - L > eps) { // for i in range(100):
        double m1 = (2 * L + R) / 3;
        double m2 = (2 * R + L) / 3;
        if (f(m1) < f(m2))
            R = m2;
        else
            L = m1;
    }
    return L; //f(L) 为最小值
}
int main() {
    double pos = minimun_double(-1, 100);
    printf("%f\n", pos);
    return 0;
}
```

10.15 多项式拟合（辛普森积分）

```
const int maxn = 110;
const long double pi = acos(-1.0L);
const long double L = -pi, R = pi; //函数 f(x) 的定义域为 [L, R]
const int T = 15; //用不超过 T 次的多项式对函数 f(x) 进行拟合
long double a[maxn][maxn]; //a[i] 表示第 i 个规范正交基，a[i][j] 表示第 i 个基的 x^j 的系数
long double pl[maxn], pr[maxn]; //pl[i] = pow(L, i), pr[i] = pow(R, i)
long double c[maxn]; //拟合出来的多项式系数
int pos = 0;
long double f(long double x) { //待拟合函数
```

```
    return sin(x);
}
long double calc(long double x) {
    long double result = 0, y = 1;
    for (int i = 0; i <= pos; ++i) {
        result += a[pos][i] * y;
        y *= x;
    }
    return result * f(x);
}
long double simpson(long double l, long double r) { //计算函数 calc 在区间 [l, r] 上的辛普森积分
    return (calc(l) + calc(r) + 4 * calc((l + r) / 2)) * (r - l) / 6;
}
long double solve(long double l, long double r, long double eps, long double val) { //递归求解积
↪   分
    long double mid = (l + r) / 2;
    long double L = simpson(l, mid), R = simpson(mid, r);
    if (fabs(L + R - val) <= 15 * eps)
        return L + R + (L + R - val) / 15;
    return solve(l, mid, eps / 2, L) + solve(mid, r, eps / 2, R);
}
//eps 设置太小可能会导致死循环
long double asme(long double l, long double r, long double eps = 1e-16l) { //自适应辛普森积分
    return solve(l, r, eps, simpson(l, r));    //求函数 calc 在区间 [l, r] 上的自适应辛普森积分,
↪   eps 指定精度
}
int main() {
    pl[0] = pr[0] = 1;
    for (int i = 1; i < maxn; ++i) {
        pl[i] = pl[i - 1] * L;
        pr[i] = pr[i - 1] * R;
    }
    a[0][0] = 1 / sqrt(R - L);
    for (int i = 1; i <= T; ++i) { //计算第 i 个规范正交基
        a[i][i] = 1;
        for (int j = 0; j < i; ++j) {
            long double sum = 0; //sum = <x^i, a[j]>
            for (int k = 0; k <= j; ++k)
                sum += a[j][k] * (pr[i + k + 1] - pl[i + k + 1]) / (i + k + 1);
            for (int k = 0; k <= j; ++k)
                a[i][k] -= sum * a[j][k];
        }
        long double total = 0;
        for (int j = 0; j <= i; ++j)
            for (int k = 0; k <= i; ++k)
                total += a[i][j] * a[i][k] * (pr[j + k + 1] - pl[j + k + 1]) / (j + k + 1);
```

```
        long double length = sqrt(total);
        for (int j = 0; j <= i; ++j)
            a[i][j] /= length;
    }
    for (::pos = 0; pos <= T; ++pos) {
        long double res = asme(L, R);
        for (int i = 0; i <= pos; ++i)
            c[i] += a[pos][i] * res;
    }
    for (int i = 0; i <= T; ++i)
        printf("%.17le,", c[i]);
    return 0;
}
```

10.16 多项式拟合（泰勒展开）

```
# 该程序展示了用 15 次多项式去拟合函数 f(x) = ∫_0^x e^{-x^2} dx
# 这个积分是求不出来原函数的，但是我们可以将 e^{-x^2} 泰勒展开，然后再对得到的每个多项式积分就得到了一个和
↪ 式：
#∑_{n=0}^∞ (-1)^n x^{2n+1} / (n!(2n+1))
# 这个和式就是 f(x) 的泰勒展开式，我们用其前 50 项作为对 f(x) 的近似，并求出拟合函数。
from decimal import *
getcontext().prec = 100
maxn = 210
b = Decimal(1)
F = [0, 1]
for i in range(1, 50): #F 为待求函数的泰勒展开式的系数，F[i] 为 x^i 的系数，F 中的项越多近似越好
    b *= -i
    F.append(0)
    F.append(1 / b / Decimal(2 * i + 1))
def asme(pos):
    length = pos + len(F) + 2
    t = [Decimal(0) for i in range(length)]
    for i in range(pos + 1):
        for j in range(len(F)):
            t[i + j] += a[pos][i] * F[j]
    res = Decimal(0)
    for i in range(length):
        res += t[i] * (pr[i + 1] - pl[i + 1]) / (i + 1)
    return res
L, R, T = Decimal(0), Decimal(2), 15 # 待拟合区间为 [L, R]，所用多项式的最高次数为 T
a = [[Decimal(0) for j in range(maxn)] for i in range(maxn)]
c = [Decimal(0) for i in range(maxn)]
pl = [Decimal(1)]
pr = [Decimal(1)]
for i in range(maxn):
    pl.append(pl[-1] * L)
```

```
    pr.append(pr[-1] * R)
a[0][0] = 1 / (R - L).sqrt()
for i in range(1, T + 1):
    a[i][i] = Decimal(1)
    for j in range(i):
        sum = Decimal(0)
        for k in range(j + 1):
            sum += a[j][k] * (pr[i + k + 1] - pl[i + k + 1]) / (i + k + 1)
        for k in range(j + 1):
            a[i][k] -= sum * a[j][k]
    total = Decimal(0)
    for j in range(i + 1):
        for k in range(i + 1):
            total += a[i][j] * a[i][k] * (pr[j + k + 1] - pl[j + k + 1]) / (j + k + 1)
    length = total.sqrt()
    for j in range(i + 1):
        a[i][j] /= length
for pos in range(T + 1):
    res = asme(pos);
    for i in range(pos + 1):
        c[i] += a[pos][i] * res;
for i in range(T + 1):
    print("%.20e" % c[i], end=", ")
```

10.17 雅可比方法

```
/*
用雅可比方法求出实对称矩阵的特征值和特征向量，注意必须是实对称矩阵。
时间复杂度 O(n^3)，大常数。
*/
const int maxn = 210;
const double eps = 1e-8;
using matrix = double[maxn][maxn];
using vec = array<double, maxn>;
using pair_t = pair<double, vec>;
struct {
    matrix A, V;
    int column[maxn], n;
    void update(int r, int c, double v) {
        A[r][c] = v;
        if (column[r] == c || fabs(A[r][c]) > fabs(A[r][column[r]])) {
            for (int i = 0; i < n; ++i) if (i != r)
                if (fabs(A[r][i]) > fabs(A[r][column[r]]))
                    column[r] = i;
        }
    }
    void Jacobi() {
```

```
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j)
            V[i][j] = 0;
        V[i][i] = 1;
        column[i] = (i == 0 ? 1 : 0);
    }
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
            if (j != i && fabs(A[i][j]) > fabs(A[i][column[i]]))
                column[i] = j;
    for (int T = 0; ; ++T) { //迭代次数限制
        int x, y;
        double val = 0;
        for (int i = 0; i < n; ++i)
            if (fabs(A[i][column[i]]) > val)
                val = fabs(A[i][column[i]]), x = i, y = column[i];
        if (val < eps) //精度限制
            break;
        double phi = atan2(-2 * A[x][y], A[y][y] - A[x][x]) / 2;
        double sinp = sin(phi), cosp = cos(phi);
        for (int i = 0; i < n; ++i) if (i != x && i != y) {
            double a = A[x][i] * cosp + A[y][i] * sinp;
            double b = A[x][i] * -sinp + A[y][i] * cosp;
            update(x, i, a);
            update(y, i, b);
        }
        for (int i = 0; i < n; ++i) if (i != x && i != y) {
            double a = A[i][x] * cosp + A[i][y] * sinp;
            double b = A[i][x] * -sinp + A[i][y] * cosp;
            update(i, x, a);
            update(i, y, b);
        }
        for (int i = 0; i < n; ++i) {
            double a = V[i][x] * cosp + V[i][y] * sinp;
            double b = V[i][x] * -sinp + V[i][y] * cosp;
            V[i][x] = a, V[i][y] = b;
        }
        double a = A[x][x] * cosp * cosp + A[y][y] * sinp * sinp + 2 * A[x][y] * cosp * sinp;
        double b = A[x][x] * sinp * sinp + A[y][y] * cosp * cosp - 2 * A[x][y] * cosp * sinp;
        double tmp = (A[y][y] - A[x][x]) * sin(2 * phi) / 2 + A[x][y] * cos(2 * phi);
        update(x, y, tmp);
        update(y, x, tmp);
        A[x][x] = a, A[y][y] = b;
    }
}
//返回特征值和特征向量组成的 pair，按照特征值从大到小排序
```

```
    auto solve(const matrix& input, int n) {
        this->n = n;
        for (int i = 0; i < n; ++i)
            for (int j = 0; j < n; ++j)
                A[i][j] = input[i][j];
        Jacobi();
        vector<pair_t> result;
        for (int i = 0; i < n; ++i)
            result.emplace_back(A[i][i], vec());
        for (int i = 0; i < n; ++i)
            for (int j = 0; j < n; ++j)
                result[i].second[j] = V[j][i];
        sort(result.begin(), result.end(), greater<pair_t>());
        return result;
    }
}jacobi;
matrix tmp = { {15.980000, 3.400000, -10.370000, },
    {3.400000, 8.492000, -8.062000, },
    {-10.370000, -8.062000, 11.572000, }, };
int main() {
    auto result = jacobi.solve(tmp, 3);
    for (int i = 0; i < 3; ++i) {
        auto eigenvalue = result[i].first;
        auto eigenvector = result[i].second;
        printf("(%f)", eigenvalue);
        for (int i = 0; i < 3; ++i)
            printf(" %f", eigenvector[i]);
        printf("\n");
    }
    return 0;
}
```

10.18 矩阵求逆

```
const int maxn = 555;
namespace inverse_double {
    using matrix = double[maxn][maxn];
    double temp[maxn][maxn * 2];
    void inverse(const matrix& A, matrix& res, int n) {
        for (int i = 0; i < n; ++i) {
            for (int j = 0; j < n; ++j)
                temp[i][j] = A[i][j], temp[i][n + j] = 0;
            temp[i][n + i] = 1;
        }
        for (int i = 0; i < n; ++i) {
            int r = i;
            for (int j = i + 1; j < n; ++j)
```

```
                if (fabs(temp[j][i]) > fabs(temp[r][i]))
                    r = j;
            if (r != i)
                for (int j = 0; j < 2 * n; ++j)
                    swap(temp[i][j], temp[r][j]);
            assert(fabs(temp[i][i]) > 1e-10);
            for (int k = i + 1; k < n; ++k)
                for (int j = 2 * n - 1; j >= i; --j)
                    temp[k][j] -= temp[k][i] / temp[i][i] * temp[i][j];
        }
        for (int i = n - 1; i >= 0; --i) {
            for (int j = i + 1; j < n; ++j)
                for (int k = n; k < 2 * n; ++k)
                    temp[i][k] -= temp[j][k] * temp[i][j];
            for (int k = n; k < 2 * n; ++k)
                temp[i][k] /= temp[i][i];
        }
        for (int i = 0; i < n; ++i)
            for (int j = 0; j < n; ++j)
                res[i][j] = temp[i][n + j];
    }
}
namespace inverse_mod {
    const long long mod = 998244353;
    using matrix = long long[maxn][maxn];
    matrix temp;
    long long pow_mod(long long a, long long n) {
        long long ans = 1;
        while (n) {
            if (n & 1)
                ans = ans * a % mod;
            n >>= 1;
            a = a * a % mod;
        }
        return ans;
    }
    long long inv(long long x) {
        return pow_mod(x, mod - 2);
    }
    void inverse(const matrix& A, matrix& res, int n) {
        for (int i = 0; i < n; ++i) {
            for (int j = 0; j < n; ++j)
                temp[i][j] = A[i][j], temp[i][n + j] = 0;
            temp[i][n + i] = 1;
        }
        for (int i = 0; i < n; ++i) {
```

```
            int r = i;
            for (int j = i; j < n; ++j) {
                if (temp[j][i] != 0) {
                    r = j;
                    break;
                }
            }
            if (r != i) for (int j = 0; j < n * 2; ++j)
                swap(temp[i][j], temp[r][j]);
            auto pivot = inv(temp[i][i]);
            for (int k = i + 1; k < n; ++k)
                for (int j = 2 * n - 1; j >= i; --j)
                    temp[k][j] = (temp[k][j] - temp[k][i] * pivot % mod * temp[i][j] % mod + mod)
↪   % mod;
        }
        for (int i = n - 1; i >= 0; --i) {
            for (int j = i + 1; j < n; ++j) {
                for (int k = n; k < n * 2; ++k) {
                    temp[i][k] = (temp[i][k] - temp[j][k] * temp[i][j] % mod + mod) % mod;
                }
            }
            auto pivot = inv(temp[i][i]);
            for (int k = n; k < n * 2; ++k)
                temp[i][k] = temp[i][k] * pivot % mod;
        }
        for (int i = 0; i < n; ++i)
            for (int j = 0; j < n; ++j)
                res[i][j] = temp[i][n + j];
    }
}
using namespace inverse_double;
int main() {
    matrix res, A = { {0, 1, 2}, {1, 0, 3}, {4, -3, 8} };
    inverse(A, res, 3);
    for (int i = 0; i < 3; ++i) {
        for (int j = 0; j < 3; ++j)
            printf("%.4f ", res[i][j]);
        printf("\n");
    }
    return 0;
}
```

10.19 牛顿迭代求非线性方程组

```
//const int maxn = 111;
//using matrix = double[maxn][maxn];
using vec = array<double, 3>; //用 vector 的话会很慢
```

```
//x 为初始解向量
//calc 函数返回一个 n * (n + 1) 的矩阵，左边是雅可比矩阵，右边是函数值向量
vec solve(vec x, function<void(vec, matrix&)> calc) {
    static matrix A;
    static vect res;
    int n = x.size();
    for (int T = 0; T < 50; ++T) { //不同的问题应当设置不同的迭代次数
        calc(x, A);
        least_square(A, n, n, res);
        for (int i = 0; i < n; ++i)
            x[i] -= res[i];
    }
    return x;
}
//求向量值函数 f 的零点
const double h = 1e-5; //h 指定差分的幅度
vec solve(vec x, function<vec(vec)> f) {
    static matrix A;
    static vect res;
    int n = x.size();
    for (int T = 0; T < 50; ++T) { //不同的问题应当设置不同的迭代次数
        vec y = f(x);
        for (int i = 0; i < n; ++i) { //差分法近似求雅可比矩阵
            vec u = x, v = x;
            u[i] += h / 2; v[i] -= h / 2;
            vec a = f(u), b = f(v);
            for (int j = 0; j < n; ++j)
                A[j][i] = (a[j] - b[j]) / h;
            A[i][n] = y[i];
        }
        least_square(A, n, n, res);
        for (int i = 0; i < n; ++i)
            x[i] -= res[i];
    }
    return x;
}
int main() {
    // (x - 1)^2 + y^2 + z^2 - 1 = 0
    // x^2 + (y - 1)^2 + z^2 - 1 = 0
    // x^2 + y^2 + (z - 1)^2 - 1 = 0
    auto f = [](vec arg) ->vec {
        auto x = arg[0], y = arg[1], z = arg[2];
        return vec{
            (x - 1) * (x - 1) + y * y + z * z - 1,
            x * x + (y - 1) * (y - 1) + z * z - 1,
            x * x + y * y + (z - 1) * (z - 1) - 1,
```

```
        };
    };
    auto calc = [&](vec x, matrix& A) {
        //Jacobi
        for (int i = 0; i < x.size(); ++i) {
            for (int j = 0; j < x.size(); ++j) {
                if (i == j)
                    A[i][j] = 2 * (x[i] - 1);
                else
                    A[i][j] = 2 * x[j];
            }
        }
        //f(x)
        auto res = f(x);
        for (int i = 0; i < x.size(); ++i)
            A[i][x.size()] = res[i];
    };
    vec x = { -122, 100, 30 };
    auto res = solve(x, f);
    for (auto i : res)
        printf("%.12f ", i);
    printf("\n");
    return 0;
}
```

10.20 QR 迭代

```
const int maxn = 111;
const double eps = 1e-10;
using matrix = double[maxn][maxn];
matrix Q, R;
double u[maxn], w[maxn];
vector<double> eigenvalues(matrix& A, int n) {
    for (int T = 0; T < 1000000; ++T) {
        for (int i = 0; i < n; ++i) {
            for (int j = 0; j < n; ++j) {
                Q[i][j] = 0;
                R[i][j] = A[i][j];
            }
            Q[i][i] = 1;
        }
        for (int i = 0; i < n; ++i) {
            double sum = 0;
            for (int j = i; j < n; ++j)
                sum += fabs(R[i][j]);
            if (sum < eps)
                continue;
```

```
            double sigma = R[i][i] <= 0 ? 1 : -1;
            double tot = 0;
            for (int j = i; j < n; ++j) {
                tot += R[j][i] * R[j][i];
                w[j] = -sigma * R[j][i];
            }
            w[i] += sqrt(tot);
            tot = sqrt(tot - R[i][i] * R[i][i] + w[i] * w[i]);
            for (int j = i; j < n; ++j)
                u[j] = w[j] / tot;
            for (int j = 0; j < n; ++j) {
                double product = 0;
                for (int k = i; k < n; ++k)
                    product += u[k] * Q[k][j];
                for (int k = i; k < n; ++k)
                    Q[k][j] = sigma * (Q[k][j] - 2 * product * u[k]);
            }
            for (int j = i; j < n; ++j) {
                double product = 0;
                for (int k = i; k < n; ++k)
                    product += u[k] * R[k][j];
                for (int k = i; k < n; ++k)
                    R[k][j] = sigma * (R[k][j] - 2 * product * u[k]);
            }
        }
        for (int i = 0; i < n; ++i) {
            for (int j = 0; j < n; ++j) {
                A[i][j] = 0;
                for (int k = 0; k < n; ++k)
                    A[i][j] += R[i][k] * Q[j][k];
            }
        }
    }
    vector<double> ret;
    for (int i = 0; i < n; ++i)
        ret.push_back(R[i][i]);
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j)
            printf("%.10f ", R[i][j]);
        printf("\n");
    }
    return ret;
}
matrix A = { {-1, 1, 0}, {-4, 3, 0 }, {1, 0, 2} };
int main() {
    eigenvalues(A, 3);
```

```
    return 0;
}
```

10.21 行列式计算

```
/*
矩阵树定理与部分扩展:
1.  给出一个无向无权图，设 A 为邻接矩阵，D 为度数矩阵 (D[i][i] = 结点 i 的度数，其他的无值)。
    则基尔霍夫 (Kirchhoff) 矩阵即为: K = D - A,
    K 的任意一个 n - 1 阶子式即为该图的生成树个数。
2.  把度数矩阵重新定义为 D[i][i] = 与结点 i 相连的所有边的权值和,
    把邻接矩阵重新定义为 A[i][j] = 结点 i 与结点 j 之间所有边的权值和,
    那么矩阵树定理求的就是: 所有生成树边权乘积的总和。
3.  有向图的情况也是可以做的，若 D[i][i] = 结点 i 的入边的权值和，此时求的就是外向树 (从根向外),
    若 D[i][i] = 结点 i 的出边的权值和，此时求的就是内向树 (从外向根),
    既然是有向的，那么就需要指定根，求行列式的时候去掉哪一行就是那一个元素为根。
*/
const long long mod = 1e9 + 7;
const int maxn = 500;
using matrix = long long[maxn][maxn];
//时间复杂度 n^3logm，如果超时可换成一般的高斯消元
//注意矩阵 A 的下标从 1 开始
long long det(matrix A, int n) {
    long long res = 1;
    for (int i = 0; i < n; ++i) {
        for (int j = i + 1; j < n; ++j) {
            while (A[j][i]) { //辗转相除法
                long long t = A[i][i] / A[j][i];
                for (int k = i; k < n; ++k) {
                    A[i][k] = (A[i][k] - t * A[j][k] % mod + mod) % mod;
                    swap(A[i][k], A[j][k]);
                }
                res = -res;
            }
        }
        if (!A[i][i]) return 0;
        res = (res * A[i][i] % mod + mod) % mod;
    }
    return res;
}
int main() { //洛谷 P6178
    static matrix A;
    int n, m, t;
    scanf("%d %d %d", &n, &m, &t);
    for (int i = 0; i < m; ++i) {
        int x, y, w;
        scanf("%d %d %d", &x, &y, &w);
```

```
        --x; --y;
        if (t == 0) {
            A[x][y] = (A[x][y] - w + mod) % mod;
            A[y][x] = (A[y][x] - w + mod) % mod;
            A[x][x] = (A[x][x] + w) % mod;
            A[y][y] = (A[y][y] + w) % mod;
        }
        else {
            if (x == 0) x = n - 1; else if (x == n - 1) x = 0;
            if (y == 0) y = n - 1; else if (y == n - 1) y = 0;
            A[x][y] = (A[x][y] - w + mod) % mod;
            A[y][y] = (A[y][y] + w) % mod;
        }
    }
    printf("%lld\n", det(A, n - 1));
    return 0;
}
```

10.22 二元一次不定方程

```
void gcd(long long a, long long b, long long& d, long long& x, long long& y) {
    if (!b) {
        d = a;
        x = 1;
        y = 0;
    }
    else {
        gcd(b, a % b, d, y, x);
        y -= x * (a / b);
    }
}
int main() {
    //freopen("in.txt", "r", stdin);
    int T;
    scanf("%d", &T);
    while (T--) {
        long long a, b, c, g, x, y;
        scanf("%lld %lld %lld", &a, &b, &c);
        gcd(a, b, g, x, y);
        if (c % g != 0) { //ax + by = c 无解
            puts("-1");
            continue;
        }
        x = x * c / g, y = y * c / g; //算出一组特解
        long long dx = b / g, dy = a / g;
        //通解的形式为 (x + s * dx, y - s * dy)
        //若要求 x 和 y 都大于 0, 则 s 的取值范围为 [L, R]
```

```
        long long L = x > 0 ? (1 - x) / dx : -x / dx + 1;
        long long R = y > 0 ? (y - 1) / dy : y / dy - 1;
        if (L <= R)
            printf("%lld %lld %lld %lld %lld\n", R - L + 1, x + L * dx, y - R * dy,
                x + R * dx, y - L * dy);
        else //无正整数数解时，分别输出 x 和 y 的最小正整数值
            printf("%lld %lld\n", x + L * dx, y - R * dy);
    }
    return 0;
}
```

11 网络流

11.1 Dinic

```
const int maxn = 100000 + 10;
const int inf = 1 << 30;
struct edge{
    int from, to, cap, flow;
    edge(int u, int v, int c, int f) : from(u), to(v), cap(c), flow(f) {}
};
struct Dinic {
    int n, m, s, t;
    vector<edge> edges;     // 边数的两倍
    vector<int> G[maxn];    // 邻接表，G[i][j] 表示结点 i 的第 j 条边在 e 数组中的序号
    bool vis[maxn];         // BFS 使用
    int d[maxn];            // 从起点到 i 的距离
    int cur[maxn];          // 当前弧指针
    void init(int n) {
        this->n = n;
        for (int i = 0; i < n; i++)
            G[i].clear();
        edges.clear();
    }
    void clear() {
        for (int i = 0; i < edges.size(); i++)
            edges[i].flow = 0;
    }
    void reduce() {
        for (int i = 0; i < edges.size(); i++)
            edges[i].cap -= edges[i].flow;
    }
    void addedge(int from, int to, int cap) {
        edges.push_back(edge(from, to, cap, 0));
        edges.push_back(edge(to, from, 0, 0));
        m = edges.size();
        G[from].push_back(m - 2);
```

```
        G[to].push_back(m - 1);
    }
    bool BFS() {
        memset(vis, 0, sizeof(vis));
        queue<int> Q;
        Q.push(s);
        vis[s] = 1;
        d[s] = 0;
        while (!Q.empty()) {
            int x = Q.front(); Q.pop();
            for (int i = 0; i < G[x].size(); i++) {
                edge& e = edges[G[x][i]];
                if (!vis[e.to] && e.cap > e.flow) {
                    vis[e.to] = 1;
                    d[e.to] = d[x] + 1;
                    Q.push(e.to);
                }
            }
        }
        return vis[t];
    }
    int DFS(int x, int a) {
        if (x == t || a == 0) return a;
        int flow = 0, f;
        for (int& i = cur[x]; i < G[x].size(); i++) {
            edge& e = edges[G[x][i]];
            if (d[x] + 1 == d[e.to] && (f = DFS(e.to, min(a, e.cap - e.flow))) > 0) {
                e.flow += f;
                edges[G[x][i] ^ 1].flow -= f;
                flow += f;
                a -= f;
                if (a == 0) break;
            }
        }
        return flow;
    }
    int Maxflow(int s, int t) {
        this->s = s; this->t = t;
        int flow = 0;
        while (BFS()) {
            memset(cur, 0, sizeof(cur));
            flow += DFS(s, inf);
        }
        return flow;
    }
    vector<int> Mincut() { // call this after maxflow
```

```
        vector<int> ans;
        for (int i = 0; i < edges.size(); i++) {
            edge& e = edges[i];
            if (vis[e.from] && !vis[e.to] && e.cap > 0)
                ans.push_back(i);
        }
        return ans;
    }
}dinic;
int main()
{
    freopen("D:\\in.txt", "r", stdin);
    int n, m;
    scanf("%d %d", &n, &m);
    dinic.init(n + 5);
    while (m--) {
        int s, t, u;
        scanf("%d %d %d", &s, &t, &u);
        dinic.addedge(s, t, u);
    }
    auto start = clock();
    printf("%d\n", dinic.Maxflow(1, n));
    double tot = static_cast<double>(clock() - start) / CLOCKS_PER_SEC;
    printf("Dinic: %f\n", tot);
    return 0;
}
```

11.2 ISAP

```
const int maxn = 1100;
const int maxedges = 51000;
const int inf = 1 << 30;
struct edge {
    int to, flow;
    edge *next, *pair;
    edge() {}
    edge(int to, int flow, edge *next) : to(to), flow(flow), next(next) {}
    void *operator new(unsigned, void *p) { return p; }
};
struct ISAP {
    int gap[maxn], h[maxn], n, s, t;
    edge *cur[maxn], *first[maxn], edges[maxedges * 2], *ptr;
    void init(int n) {
        this->n = n;
        ptr = edges;
        memset(first, 0, sizeof(first));
        memset(gap, 0, sizeof(gap));
```

```
        memset(h, 0, sizeof(h));
        gap[0] = n;
    }
    void add_edge(int from, int to, int cap) {
        first[from] = new(ptr++)edge(to, cap, first[from]);
        first[to] = new(ptr++)edge(from, 0, first[to]);
        first[from]->pair = first[to];
        first[to]->pair = first[from];
    }
    int augment(int x, int limit) {
        if (x == t)
            return limit;
        int rest = limit;
        for (edge*& e = cur[x]; e; e = e->next) if (e->flow && h[e->to] + 1 == h[x]) {
            int d = augment(e->to, min(rest, e->flow));
            e->flow -= d, e->pair->flow += d, rest -= d;
            if (h[s] == n || !rest)
                return limit - rest;
        }
        int minh = n;
        for (edge *e = cur[x] = first[x]; e; e = e->next) if (e->flow)
            minh = min(minh, h[e->to] + 1);
        if (--gap[h[x]] == 0)
            h[s] = n;
        else
            ++gap[h[x] = minh];
        return limit - rest;
    }
    int solve(int s, int t, int limit = inf) {
        this->s = s; this->t = t;
        memcpy(cur, first, sizeof(first)); // memcpy!
        int flow = 0;
        while (h[s] < n && flow < limit)
            flow += augment(s, limit - flow);
        return flow;
    }
}isap;
int main()
{
    freopen("D:\\in.txt", "r", stdin);
    int n, m;
    scanf("%d %d", &n, &m);
    isap.init(n + 5);
    while (m--)
    {
        int s, t, u;
```

```
        scanf("%d %d %d", &s, &t, &u);
        isap.add_edge(s, t, u);
    }
    auto start = clock();
    printf("%d\n", isap.solve(1, n));
    double tot = static_cast<double>(clock() - start) / CLOCKS_PER_SEC;
    printf("Isap: %f\n", tot);
    return 0;
}
```

11.3 HLPP

```
const int maxn = 2e5 + 5, maxedges = 4e6 + 5, inf = 0x3f3f3f3f;
int n, m, s, t, tot;
int v[maxedges * 2], w[maxedges * 2], first[maxn], nxt[maxedges * 2];
int h[maxn], e[maxn], gap[maxn * 2], inq[maxn];//节点高度是可以到达 2n-1 的
struct cmp{
    inline bool operator()(int a, int b) const {
        return h[a] < h[b];//因为在优先队列中的节点高度不会改变，所以可以直接比较
    }
};
queue<int> Q;
priority_queue<int, vector<int>, cmp> heap;
inline void add_edge(int from, int to, int flow) {
    tot += 2;
    v[tot + 1] = from; v[tot] = to; w[tot] = flow; w[tot + 1] = 0;
    nxt[tot] = first[from]; first[from] = tot;
    nxt[tot + 1] = first[to]; first[to] = tot + 1;
    return;
}
inline bool bfs() {
    memset(h + 1, 0x3f, sizeof(int) * n);
    h[t] = 0;
    Q.push(t);
    while (!Q.empty())
    {
        int now = Q.front(); Q.pop();
        for (int go = first[now]; go; go = nxt[go])
            if (w[go ^ 1] && h[v[go]] > h[now] + 1)
                h[v[go]] = h[now] + 1, Q.push(v[go]);
    }
    return h[s] != inf;
}
inline void push(int now) {
    for (int go = first[now]; go; go = nxt[go]) {
        if (w[go] && h[v[go]] + 1 == h[now]) {
            int d = min(e[now], w[go]);
```

```
            w[go] -= d; w[go ^ 1] += d; e[now] -= d; e[v[go]] += d;
            if (v[go] != s && v[go] != t && !inq[v[go]])
                heap.push(v[go]), inq[v[go]] = 1;
            if (!e[now])//已经推送完毕可以直接退出
                break;
        }
    }
}
inline void relabel(int now) {
    h[now] = inf;
    for (int go = first[now]; go; go = nxt[go])
        if (w[go] && h[v[go]] + 1 < h[now])
            h[now] = h[v[go]] + 1;
    return;
}
inline int hlpp() {
    int now, d;
    if (!bfs()) //s 和 t 不连通
        return 0;
    h[s] = n;
    memset(gap, 0, sizeof(int) * (n * 2));
    for (int i = 1; i <= n; i++)
        if (h[i] < inf)
            ++gap[h[i]];
    for (int go = first[s]; go; go = nxt[go]) {
        if (d = w[go]) {
            w[go] -= d; w[go ^ 1] += d; e[s] -= d; e[v[go]] += d;
            if (v[go] != s && v[go] != t && !inq[v[go]])
                heap.push(v[go]), inq[v[go]] = 1;
        }
    }
    while (!heap.empty()) {
        inq[now = heap.top()] = 0; heap.pop(); push(now);
        if (e[now]) {
            if (!--gap[h[now]]) //gap 优化，因为当前节点是最高的所以修改的节点一定不在优先队列中，不
↪  必担心修改对优先队列会造成影响
                for (int i = 1; i <= n; i++)
                    if (i != s && i != t && h[i] > h[now] && h[i] < n + 1)
                        h[i] = n + 1;
            relabel(now); ++gap[h[now]];
            heap.push(now); inq[now] = 1;
        }
    }
    return e[t];
}
int main()
```

```
{
    freopen("D:\\in.txt", "r", stdin);
    scanf("%d %d", &n, &m); s = 1; t = n;
    while (m--)
    {
        int s, t, u;
        scanf("%d %d %d", &s, &t, &u);
        add_edge(s, t, u);
    }
    auto start = clock();
    printf("%d\n", hlpp());
    double tot = static_cast<double>(clock() - start) / CLOCKS_PER_SEC;
    printf("HLPP: %f\n", tot);
    return 0;
}
```

11.4 MCMF-spfa

```
const int maxn = 20100;
const int inf = 1 << 30;
struct edge {
    int from, to, cap, flow, cost;
    edge(int u, int v, int c, int f, int w) : from(u), to(v), cap(c), flow(f), cost(w) {}
};
struct MCMF {
    int n, m;
    vector<edge> edges;
    vector<int> G[maxn];
    int inq[maxn];
    int d[maxn];
    int p[maxn];
    int a[maxn];
    void init(int n) {
        this->n = n;
        for (int i = 0; i < n; ++i)
            G[i].clear();
        edges.clear();
    }
    void add_edge(int from, int to, int cap, int cost) {
        edges.push_back(edge(from, to, cap, 0, cost));
        edges.push_back(edge(to, from, 0, 0, -cost));
        m = edges.size();
        G[from].push_back(m - 2);
        G[to].push_back(m - 1);
    }
    bool BellmanFord(int s, int t, int &flow, int &cost, int limit) {
        for (int i = 0; i < n; ++i)
```

```
            d[i] = inf;
        memset(inq, 0, sizeof(inq));
        d[s] = 0; inq[s] = 1; p[s] = 0; a[s] = inf;
        queue<int> Q;
        Q.push(s);
        while (!Q.empty()) {
            int u = Q.front(); Q.pop();
            inq[u] = false;
            for (unsigned i = 0; i < G[u].size(); ++i) {
                edge &e = edges[G[u][i]];
                if (e.cap > e.flow && d[e.to] > d[u] + e.cost) {
                    d[e.to] = d[u] + e.cost;
                    p[e.to] = G[u][i];
                    a[e.to] = min(a[u], e.cap - e.flow);
                    if (!inq[e.to]) {
                        Q.push(e.to);
                        inq[e.to] = true;
                    }
                }
            }
        }
        if (d[t] == inf)
            return false;
        a[t] = min(a[t], limit - flow);
        flow += a[t];
        cost += d[t] * a[t];
        for (int u = t; u != s; u = edges[p[u]].from) {
            edges[p[u]].flow += a[t];
            edges[p[u] ^ 1].flow -= a[t];
        }
        return true;
    }
    int solve(int s, int t, int limit = inf) {
        int flow = 0, cost = 0;
        while (flow < limit && BellmanFord(s, t, flow, cost, limit));
        return cost;
    }
}mcmf;
int main()
{
    freopen("D:\\in.txt", "r", stdin);
    int n, m, k;
    scanf("%d %d %d", &n, &m, &k);
    mcmf.init(n + 10);
    for (int i = 1; i <= m; ++i) {
        int x, y, c, w;
```

```
        scanf("%d %d %d %d", &x, &y, &c, &w);
        mcmf.add_edge(x, y, c, w);
    }
    auto start = clock();
    printf("%d\n", mcmf.solve(1, n, k));
    double tot = static_cast<double>(clock() - start) / CLOCKS_PER_SEC;
    printf("MCMF-spfa: %f\n", tot);
    return 0;
}
```

11.5 MCMF-dijkstra

```
const int maxn = 21000;
const int inf = 1 << 30;
struct edge {
    int to, cap, cost, rev;
    edge() {}
    edge(int to, int cap, int cost, int rev) : to(to), cap(cap), cost(cost), rev(rev) {}
};
struct MCMF {
    int n, h[maxn], d[maxn], pre[maxn], num[maxn];
    vector<edge> G[maxn];
    void init(int n) {
        this->n = n;
        for (int i = 0; i <= n; ++i)
            G[i].clear();
    }
    void add_edge(int from, int to, int cap, int cost) {
        G[from].push_back(edge(to, cap, cost, G[to].size()));
        G[to].push_back(edge(from, 0, -cost, G[from].size() - 1));
    }
    //flow 是自己传进去的变量，就是最后的最大流，返回的是最小费用
    int solve(int s, int t, int &flow, int limit = inf) {
        int cost = 0;
        memset(h, 0, sizeof(h));
        while (limit) {
            priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>> > Q;
            for (int i = 0; i <= n; ++i)
                d[i] = inf;
            d[s] = 0;
            Q.emplace(0, s);
            while (!Q.empty()) {
                auto now = Q.top(); Q.pop();
                int u = now.second;
                if (d[u] < now.first) continue;
                for (int i = 0; i < G[u].size(); ++i) {
                    edge &e = G[u][i];
```

```
                    if (e.cap > 0 && d[e.to] > d[u] + e.cost + h[u] - h[e.to]) {
                        d[e.to] = d[u] + e.cost + h[u] - h[e.to];
                        pre[e.to] = u;
                        num[e.to] = i;
                        Q.emplace(d[e.to], e.to);
                    }
                }
            }
            if (d[t] == inf) break;
            for (int i = 0; i <= n; ++i)
                h[i] += d[i];
            int a = limit;
            for (int u = t; u != s; u = pre[u])
                a = min(a, G[pre[u]][num[u]].cap);
            limit -= a; flow += a; cost += a * h[t];
            for (int u = t; u != s; u = pre[u]) {
                edge &e = G[pre[u]][num[u]];
                e.cap -= a;
                G[u][e.rev].cap += a;
            }
        }
        return cost;
    }
}mcmf;
int main()
{
    freopen("D:\\in.txt", "r", stdin);
    int n, m, k, flow = 0;
    scanf("%d %d %d", &n, &m, &k);
    mcmf.init(n + 10);
    for (int i = 1; i <= m; ++i) {
        int x, y, c, w;
        scanf("%d %d %d %d", &x, &y, &c, &w);
        mcmf.add_edge(x, y, c, w);
    }
    auto start = clock();
    printf("%d\n", mcmf.solve(1, n, flow, k));
    printf("flow: %d\n", flow);
    double tot = static_cast<double>(clock() - start) / CLOCKS_PER_SEC;
    printf("MCMF-dijkstra: %f\n", tot);
    return 0;
}
```

12 线性规划

12.1 线性规划

```
// 改进单纯型法的实现
// 参考：http://en.wikipedia.org/wiki/Simplex_algorithm
// 输入矩阵 a 描述线性规划的标准形式。a 为 m+1 行 n+1 列，其中行 0~m-1 为不等式，行 m 为目标函数（最
↪  大化）。列 0~n-1 为变量 0~n-1 的系数，列 n 为常数项
// 第 i 个约束为 a[i][0]*x[0] + a[i][1]*x[1] + ... <= a[i][n]
// 目标为 max(a[m][0]*x[0] + a[m][1]*x[1] + ... + a[m][n-1]*x[n-1] - a[m][n])
// 注意：变量均有非负约束 x[i] >= 0
const int maxm = 500; // 约束数目上限
const int maxn = 500; // 变量数目上限
const double INF = 1e100;
const double eps = 1e-10;
struct Simplex {
    int n; // 变量个数
    int m; // 约束个数
    double a[maxm][maxn]; // 输入矩阵
    int f[maxm], d[maxn]; // 算法辅助变量
    void pivot(int r, int c)
    {
        swap(d[c], f[r]);
        a[r][c] = 1 / a[r][c];
        for (int j = 0; j <= n; j++) if (j != c) a[r][j] *= a[r][c];
        for (int i = 0; i <= m; i++) if (i != r)
        {
            for (int j = 0; j <= n; j++) if (j != c) a[i][j] -= a[i][c] * a[r][j];
            a[i][c] = -a[i][c] * a[r][c];
        }
    }
    bool feasible()
    {
        for (;;)
        {
            int r, c;
            double p = INF;
            for (int i = 0; i < m; i++) if (a[i][n] < p) p = a[r = i][n];
            if (p > -eps) return true;
            p = 0;
            for (int i = 0; i < n; i++) if (a[r][i] < p) p = a[r][c = i];
            if (p > -eps) return false;
            p = a[r][n] / a[r][c];
            for (int i = r + 1; i < m; i++) if (a[i][c] > eps)
            {
                double v = a[i][n] / a[i][c];
                if (v < p)  r = i, p = v;
```

```
            }
            pivot(r, c);
        }
    }
    // 解有界返回 1，无解返回 0，无界返回-1。f[i] 为 x[i] 的值，ret 为目标函数的值
    int simplex(int n, int m, double x[maxn], double& ret)
    {
        this->n = n;
        this->m = m;
        for (int i = 0; i < n; i++) d[i] = i;
        for (int i = 0; i < m; i++) f[i] = n + i;
        if (!feasible()) return 0;
        for (;;)
        {
            int r, c;
            double p = 0;
            for (int i = 0; i < n; i++) if (a[m][i] > p) p = a[m][c = i];
            if (p < eps)
            {
                for (int i = 0; i < n; i++) if (d[i] < n) x[d[i]] = 0;
                for (int i = 0; i < m; i++) if (f[i] < n) x[f[i]] = a[i][n];
                ret = -a[m][n];
                return 1;
            }
            p = INF;
            for (int i = 0; i < m; i++) if (a[i][c] > eps)
            {
                double v = a[i][n] / a[i][c];
                if (v < p)  r = i, p = v;
            }
            if (p == INF) return -1;
            pivot(r, c);
        }
    }
};
int main()
{
    return 0;
}
```

13 字符串算法

13.1 后缀树

```
/*
1.    ch[u][c] 表示从结点 u 出发走字符 c 后到达的结点（0 为空结点）。
2.    len[u] 表示从结点 u 的父结点走到 u 的这条边的长度（可能为 0）。
```

```
3.	从结点 u 的父结点走到 u 的字符串的下标区间为 [start[u], start[u] + len[u] - 1]。
4.	link[u] 记录结点 u 的后缀链接，若内部结点 u 表示的字符串为 [s, t]，则其后缀链接的结点对应字符串
↪ 为 [s+1, t]。
5.	depth[u] 表示结点 u 的加权深度，dep[u] 表示结点的不加权深度（根结点到 u 经过的边数）。
6.	s 数组记录输入的字符串。
7.	对于叶结点 pos[i] 表示结点 i 对应的后缀的下标，非叶结点 pos[i] 为-1。
8.	node 是 pos 的反函数，node[i] 表示后缀 s[i, m] 对应的结点（下标从 1 开始）。
*/
const int sigma_size = 26;
const int maxlog = 20;
const int maxn = 4e5 + 10;
const int inf = 1e9;
struct Suffix_Tree {
public: //后缀树基本结构
    int ch[maxn][sigma_size + 1], len[maxn], start[maxn], link[maxn], depth[maxn], s[maxn], cur,
↪   n, rem, now;
    int pos[maxn], node[maxn];
    void init() {
        memset(ch, 0, sizeof(ch));
        cur = now = 1;
        n = rem = 0;
        len[0] = inf;
        depth[0] = 0;
    }
    int newnode(int p, int w, int d) {
        start[++cur] = p;
        len[cur] = w;
        depth[cur] = d;
        link[cur] = 1;
        return cur;
    }
    void extend(int x) { //0 <= x <= sigma_size
        s[++n] = x, ++rem;
        for (int last = 1; rem; now == 1 ? --rem : now = link[now]) {
            while (rem > len[ch[now][s[n - rem + 1]]])
                rem -= len[now = ch[now][s[n - rem + 1]]];
            int ed = s[n - rem + 1];
            int& v = ch[now][ed];
            int c = s[start[v] + rem - 1];
            link[last] = now;
            if (!v) {
                v = newnode(n - rem + 1, inf, n - rem + 1 - depth[now]);
                last = now;
            }
            else if (x == c) {
                last = now;
```

```
            break;
        }
        else {
            int u = newnode(start[v], rem - 1, depth[now] + rem - 1);
            ch[u][x] = newnode(n, inf, n - depth[u]);
            ch[u][c] = v;
            start[v] += rem - 1;
            if (len[v] != inf)
                len[v] -= rem - 1;
            link[last] = v = u;
            last = v;
        }
    }
}
void maintain(int m) { //m 为输入字符串的长度
//对后缀树进行调整，并消除最后 extend(sigma_size) 对后缀树结构的影响。
//调整后的后缀树的最后没有字符 sigma_size，但是树边的长度可能为 0，
//并且遍历的时候要枚举所有的字符：0 <= c <= sigma_size
    ch[1][sigma_size] = 0;
    for (int i = 1; i <= cur; ++i) {
        if (len[i] == inf) {
            len[i] = m - start[i] + 1;
            pos[i] = depth[i];
            node[pos[i]] = i;
            depth[i] = m - depth[i] + 1;
        }
        else
            pos[i] = -1;
    }
}
void insert(const char* str) { //插入的字符串的下标从 1 开始
    assert(str[0] == 0);
    int m = strlen(str + 1);
    for (int i = 1; i <= m; ++i)
        extend(str[i] - 'a');
    extend(sigma_size);
    maintain(m);
}
int go(const char* str, int nd = 1) { //如果终止在边内则会到子结点
    int m = strlen(str);
    for (int i = 0; i < m; i += len[nd]) {
        int c = str[i] - 'a';
        if (!ch[nd][c])
            return 0;
        nd = ch[nd][c];
        for (int j = 0; j < len[nd] && i + j < m; ++j)
```

```
                if (str[i + j] - 'a' != s[start[nd] + j])
                    return 0;
        }
        //如果有长度为 0 的边则直接走过去。在处理子树问题时应当直接返回 nd。
        return ch[nd][sigma_size] != 0 ? ch[nd][sigma_size] : nd;
    }
public: //后缀树上倍增
    int L[maxn], R[maxn], dep[maxn], anc[maxn][maxlog]; //anc[i][j] 表示结点 i 往上走 2^j 个点到达
↪   的点
    vector<int> seq;
    void dfs(int x, int fa) {
        dep[x] = dep[fa] + 1;
        anc[x][0] = fa;
        L[x] = seq.size();
        if (pos[x] > 0)
            seq.push_back(pos[x]);
        for (int i = 0; i <= sigma_size; ++i) if (ch[x][i])
            dfs(ch[x][i], x);
        R[x] = seq.size() - 1;
    }
    void preprocess() { //后缀树上倍增初始化
        for (int i = 1; i <= cur; ++i) {
            for (int j = 1; (1 << (j - 1)) <= cur; ++j)
                anc[i][j] = 0;
        }
        for (int j = 1; (1 << j) <= cur; ++j) {
            for (int i = 1; i <= cur; ++i) {
                if (anc[i][j - 1] != 0) {
                    int a = anc[i][j - 1];
                    anc[i][j] = anc[a][j - 1];
                }
            }
        }
    }
    void build() {          //求出后缀树的 DFS 序
        seq.clear();      //可以在此处向 seq 中加入一个元素 0，从而使得 DFS 序区间从 1 开始
        dep[0] = -1;
        dfs(1, 0);          //求后缀树的 DFS 序
        preprocess();     //后缀树上倍增初始化
    }
    int ascend(int x, int length) { //将结点上升到加权深度为 length 的位置。
        for (int i = maxlog - 1; i >= 0; --i)
            if (depth[anc[x][i]] >= length) //调用该函数之前需要先调用 preprocess 初始化
                x = anc[x][i];
        return x;
    }
```

```
    int query(int L, int R) { //返回从根结点开始沿着字符串 s[L, R] 走到的结点（若最后结束在边上，则
↪   走到子结点）
        int x = node[L], length = R - L + 1;
        return ascend(x, length);
    }
    int lca(int x, int y) { //返回后缀树中结点 x 与 y 的最近公共祖先。
        if (dep[x] < dep[y])
            swap(x, y);
        for (int i = maxlog - 1; i >= 0; --i)
            if (dep[x] - (1 << i) >= dep[y])
                x = anc[x][i];
        if (x != y) {
            for (int i = maxlog - 1; i >= 0; --i)
                if (anc[x][i] && anc[x][i] != anc[y][i])
                    x = anc[x][i], y = anc[y][i];
            x = anc[x][0];
        }
        return x;
    }
    int lcp(int i, int j) { //返回后缀 s[i, m] 与后缀 s[j, m] 的最长公共前缀。
        return depth[lca(node[i], node[j])];
    }
public: //后缀树上快速下降
    int sz[maxn], leaf[maxn];
    void travel(int x) { //调用 travel(1) 对后缀树进行树剖，之后才能调用 go(x, L, R)。
        sz[x] = 1;
        leaf[x] = pos[x];
        for (int z = 0, i = 0; i <= sigma_size; ++i) {
            int y = ch[x][i];
            if (y) travel(y);
            sz[x] += sz[y];
            if (sz[y] > sz[z]) {
                z = y;
                leaf[x] = leaf[y];
            }
        }
    }
    int go(int x, int L, int R) { //从结点 x 开始往下走字符串 s[L, R]，时间复杂度 O(log^2n)
        while (x && L <= R) {
            if (s[L] == s[leaf[x] + depth[x]]) {
                int length = min(R - L + 1, lcp(leaf[x] + depth[x], L));
                L += length;
                int y = ascend(node[leaf[x]], depth[x] + length);
                if (L <= R && depth[y] != depth[x] + length)
                    return 0;
                x = y;
```

```
        }
        else {
            int y = ch[x][s[L]];
            if (lcp(start[y], L) < min(len[y], R - L + 1))
                return 0;
            L += len[y];
            x = y;
        }
    }
    return x;
    //return ch[x][sigma_size] != 0 ? ch[x][sigma_size] : x;
}
int recognise(int l, int r, int L, int R) { //返回从根结点开始先走 s[l, r] 再走 s[L, R] 所到的
↪ 结点。
    int x = query(l, r);
    int diff = depth[x] - (r - l + 1);
    if (diff > 0 && lcp(start[x] + len[x] - diff, L) < min(diff, R - L + 1))
        return 0;
    else
        return go(x, L + diff, R);
    //因为树中有长度为 0 的边，所以即使要识别的字符串为原串的后缀也不一定走到叶结点，
    //若要保证在这种情况下走到叶结点则应有：x = (ch[x][sigma_size] != 0 ? ch[x][sigma_size] :
↪ x);
}
} st;
int main() {
    static char s[maxn];
    srand(time(0));
    int n = 200000;
    for (int i = 1; i <= n; ++i)
        s[i] = 'a' + (rand() % 2 == 0);
    auto start = clock();
    st.init();
    st.insert(s);
    st.build();
    st.travel(1);
    for (int i = 1; i <= n - 10; ++i) {
        int mid = (i + n) / 2;
        int x = st.recognise(i, mid, mid + 1, n);
        x = (st.ch[x][sigma_size] != 0 ? st.ch[x][sigma_size] : x);
        auto j = st.pos[x];
        if (j != i || st.depth[x] != n - i + 1) {
            printf("%d %d (%d)\n", j, i, x);
            abort();
        }
    }
```

```
    auto end = clock();
    printf("time: %f\n", double(end - start) / CLOCKS_PER_SEC);
    return 0;
}
```

13.2 扩展 KMP

```
char T[] = "aaababa", P[] = "aa";
const int maxn = 10000;
int nxt[maxn], extend[maxn];
//注意：字符串 T 与字符串 P 必须以不同的特殊字符结尾（该特殊字符不算在字符串内，故不能算入字符串的长度
↪ 里）
void getnext(char P[])
{
    int po = 1, m = strlen(P);
    nxt[0] = m;
    nxt[1] = mismatch(P + 1, P + m, P).second - P;
    for (int i = 2; i < m; ++i)
    {
        if (nxt[i - po] + i < nxt[po] + po)
            nxt[i] = nxt[i - po];
        else
        {
            int j = max(nxt[po] + po - i, 0);
            nxt[i] = mismatch(P + j + i, P + m, P + j).second - P;
            po = i;
        }
    }
}
void exkmp(char T[], char P[])
{
    int po = 0, n = strlen(T), m = strlen(P);
    P[m] = 0; T[n] = 1;
    extend[0] = mismatch(T, T + n, P).second - P;
    getnext(P); //注意：若字符串 P 最后的特殊字符不是 0，则字符串的长度会加 1，next[0] 会与实际不符。
    for (int i = 1; i < n; i++)
    {
        if (nxt[i - po] + i < extend[po] + po)
            extend[i] = nxt[i - po];
        else
        {
            int j = max(extend[po] + po - i, 0);
            extend[i] = mismatch(T + i + j, T + n, P + j).second - P;
            po = i;
        }
    }
}
```

```
int main()
{
    exkmp(T, P);
    for (int i = 0; i < 15; ++i)
        printf("%d\n", extend[i]);
    return 0;
}
```

13.3 AC 自动机

```
/*
通过 insert 将字符串插入 Trie 之后不要忘记调用 getfail 建立 AC 自动机！
对于 AC 自动机中的状态 u，不能仅仅通过 val[u] != 0 来判断 u 是否能匹配输入字符串，还要结合 last。
对于 AC 自动机中的状态 u，其不能匹配任何一个字符串的条件为 val[u] == 0 && last[u] == 0。
*/
const int maxnode = 100000;
const int sigma_size = 26;
struct trie
{
    int ch[maxnode][sigma_size];
    int f[maxnode];        // fail 函数
    int val[maxnode];     // 每个字符串的结尾结点都有一个非 0 的 val
    int last[maxnode]; // 输出链表的下一个结点
    int sz;
    void init()
    {
        sz = 1;
        memset(ch[0], 0, sizeof(ch[0]));
    }
    // 字符 c 的编号
    int idx(char c)
    {
        return c - 'a';
    }
    // 插入字符串。v 必须非 0
    void insert(const char *s, int v)
    {
        int u = 0, n = strlen(s);
        for (int i = 0; i < n; i++)
        {
            int c = idx(s[i]);
            if (!ch[u][c])
            {
                memset(ch[sz], 0, sizeof(ch[sz]));
                val[sz] = 0;
                ch[u][c] = sz++;
            }
```

```
        u = ch[u][c];
    }
    val[u] = v;
}
// 计算 fail 函数
void getfail()
{
    queue<int> Q;
    f[0] = 0;
    // 初始化队列
    for (int c = 0; c < sigma_size; c++)
    {
        int u = ch[0][c];
        if (u) { f[u] = 0; Q.push(u); last[u] = 0; }
    }
    // 按 BFS 顺序计算 fail
    while (!Q.empty())
    {
        int r = Q.front(); Q.pop();
        for (int c = 0; c < sigma_size; c++)
        {
            int u = ch[r][c];
            if (!u)
            {
                ch[r][c] = ch[f[r]][c];
                continue;
            }
            Q.push(u);
            int v = f[r];
            while (v && !ch[v][c]) v = f[v];
            f[u] = ch[v][c];
            last[u] = val[f[u]] ? f[u] : last[f[u]];
        }
    }
}
// 在 T 中找模板
void find(const char *T)
{
    int n = strlen(T);
    int j = 0; // 当前结点编号，初始为根结点
    for (int i = 0; i < n; i++)
    {
        // 文本串当前指针
        int c = idx(T[i]);
        j = ch[j][c];
        if (val[j]) print(j);
```

```
            else if (last[j]) print(last[j]); // 找到了！
        }
    }
    // 递归打印以结点 j 结尾的所有字符串
    void print(int j)
    {
        if (j)
        {
            //对查找到的编号为 val[j] 的字符串进行操作
            printf("%d\n", val[j]);
            print(last[j]);
        }
    }
}ac;
int main()
{
    ac.init();
    ac.insert("aabbc", 1);
    ac.insert("bbcff", 2);
    ac.insert("bb", 3);
    ac.getfail();
    ac.find("aabbcffbbbop");
    return 0;
}
```

13.4 KMP 算法

```
char T[] = "abcdefabc", P[] = "abc";
const int maxn = 10000;
int f[maxn];    //f[i] 表示字符串 s[0, i-1] 的后缀与前缀的最长公共部分（后缀与前缀均不包含字符串本
↪   身）
                //若 f[i] = k 则，字符串 s[0, k-1] 与字符串 s[i-k, i-1] 相同
void getfail(char *P, int *f)
{
    int m = strlen(P);
    f[0] = 0; f[1] = 0;
    for (int i = 1; i < m; ++i)
    {
        int j = f[i];
        while (j && P[j] != P[i])
            j = f[j];
        f[i + 1] = P[j] == P[i] ? j + 1 : 0;
    }
}
void find(char *T, char *P, int *f)
{
    int n = strlen(T), m = strlen(P);
```

```
    getfail(P, f);
    int j = 0;
    for (int i = 0; i < n; ++i)
    {
        while (j && P[j] != T[i])
            j = f[j];
        if (P[j] == T[i])
            ++j;
        if (j == m)
            printf("%d\n", i - m + 1); //在串 T 中找到了 P, 下标为 i - m + 1
    }
}
int main()
{
    getfail(P, f);
    find(T, P, f);
    return 0;
}
```

13.5 Manacher 算法

```
const int maxn = 1.1e7 * 2 + 100; //maxn 应当大于 原字符串长度的两倍加二
char s[maxn];
int len[maxn];
//回文串在原字符串中的长度为 len[i] - 1
void manacher(char* str) {
    int n = strlen(str), m = 0;
    for (int i = 0; i < n; ++i) {
        s[++m] = '#';
        s[++m] = str[i];
    }
    s[++m] = '#'; s[0] = '$'; //s 是加入新字符后的字符串
    len[1] = 1;
    int r = 1, po = 1; //r 是当前极长回文子串的最右的端点 po 为 r 对应的回文子串的中心
    for (int i = 1; i <= m; ++i) {
        if (i < r)
            len[i] = min(len[2 * po - i], r - i + 1); //2*po-i 为 i 在当前这个极长回文子串中在左边
↪   相对应的位置
        else
            len[i] = 1;
        while (s[i + len[i]] == s[i - len[i]])
            ++len[i];
        if (i + len[i] > r) {
            r = i + len[i] - 1;
            po = i;
        }
    }
```

```
    //原字符串的最长回文子串为 max{ len[i] - 1 }
    //以 i 为中心的奇回文串的长度为 len[2 * i + 2] - 1
    //以 i 为左中心的偶回文串的长度为 len[2 * i + 3] - 1
}
char a[maxn / 2];
int main() {
    scanf("%s", a);
    const int n = strlen(a);
    manacher(a);
    int ans = *max_element(len, len + n * 2 + 4) - 1;
    printf("%d\n", ans);
    return 0;
}
```

13.6 后缀数组

```
/*
1.    rank[i] 表示下标 i 的排名（排名从 0 开始）。
2.    sa[i] 表示第 i 小的后缀的下标（i 从 0 开始）。
3.    height[i] 表示 sa[i - 1] 与 sa[i] 的最长公共前缀。
*/
const int maxn = 210000; //maxn 应当开到最大字符串长度的两倍，否则 (1) 处下标访问可能越界。
const int maxlog = 20;
struct Suffix_Array {
    char s[maxn];
    int sa[maxn], rank[maxn], height[maxn];
    int t[maxn], t2[maxn], c[maxn], n;
    void init(const char* str) {
        strcpy(s, str);
        n = strlen(s);
        memset(t, 0, sizeof(int) * (2 * n + 10));  //为了保证 (1) 处访问越界时得到的数组值恒为 0,
↪   应当将 t 和 t2 数组清空
        memset(t2, 0, sizeof(int) * (2 * n + 10));
    }
    void build_sa(int m = 256) {
        int* x = t, * y = t2;
        for (int i = 0; i < m; ++i) c[i] = 0;
        for (int i = 0; i < n; ++i) c[x[i] = s[i]]++;
        for (int i = 1; i < m; ++i) c[i] += c[i - 1];
        for (int i = n - 1; i >= 0; --i) sa[--c[x[i]]] = i;
        for (int k = 1; k <= n; k <<= 1) {
            int p = 0;
            for (int i = n - 1; i >= n - k; --i) y[p++] = i;
            for (int i = 0; i < n; ++i) if (sa[i] >= k) y[p++] = sa[i] - k;
            for (int i = 0; i < m; ++i) c[i] = 0;
            for (int i = 0; i < n; ++i) c[x[y[i]]]++;
            for (int i = 1; i < m; ++i) c[i] += c[i - 1];
```

```
        for (int i = n - 1; i >= 0; --i) sa[--c[x[y[i]]]] = y[i];
        swap(x, y);
        p = 1; x[sa[0]] = 0;
        for (int i = 1; i < n; ++i)
            x[sa[i]] = y[sa[i - 1]] == y[sa[i]] && y[sa[i - 1] + k] == y[sa[i] + k] ? p - 1 :
↪ p++; //(1)
        if (p >= n) break;
        m = p;
    }
}
void getheight() {
    int k = 0;
    for (int i = 0; i < n; ++i) rank[sa[i]] = i;
    for (int i = 0; i < n; ++i) if (rank[i] > 0) {
        if (k) k--;
        int j = sa[rank[i] - 1];
        while (i + k < n && j + k < n && s[i + k] == s[j + k]) k++;
        height[rank[i]] = k;
    }
}
int d[maxn][maxlog], log[maxn];
void RMQ_init() {
    log[0] = -1;
    for (int i = 1; i <= n; ++i)
        log[i] = log[i / 2] + 1;
    for (int i = 0; i < n; ++i)
        d[i][0] = height[i];
    for (int j = 1; j <= log[n]; ++j)
        for (int i = 0; i + (1 << j) - 1 < n; ++i)
            d[i][j] = min(d[i][j - 1], d[i + (1 << (j - 1))][j - 1]);
}
int lcp(int i, int j) { //返回下标 i 开始的后缀与下标 j 开始的后缀的最长公共前缀。
    if (i == j)
        return n - i;
    if (rank[i] > rank[j])
        swap(i, j);
    int x = rank[i] + 1, y = rank[j];
    int k = log[y - x + 1];
    return min(d[x][k], d[y - (1 << k) + 1][k]);
}
pair<int, int> locate(int l, int r) {
    //返回一个最长的区间 [L, R] 使得 sa 中下标从 L 到 R 的所有后缀都以 s[l, r] 为前缀。
    int pos = rank[l], length = r - l + 1;
    int L = 0, R = pos;
    while (L < R) {
        int M = L + (R - L) / 2;
```

```
        if (lcp(l, sa[M]) >= length) R = M;
        else L = M + 1;
    }
    int tmp = L;
    L = pos, R = n - 1;
    while (L < R) {
        int M = L + (R - L + 1) / 2;
        if (lcp(l, sa[M]) >= length) L = M;
        else R = M - 1;
    }
    return make_pair(tmp, L);
}
pair<int, int> locate(vector<pair<int, int>> ranges) {
    //将 ranges 中的所有下标区间对应的子串拼接到一起，得到字符串 T,
    //返回一个最长的区间 [L, R] 使得 sa 中下标从 L 到 R 的所有后缀都以 T 为前缀，
    //无解返回 { 0, -1 }。
    int l = 0, r = n - 1, pos = 0;
    for (auto [x, y] : ranges) {
        int L = l, R = r, len = y - x + 1;
        while (L < R) {
            int M = L + (R - L) / 2;
            int pre = sa[M] + pos < n ? lcp(sa[M] + pos, x) : 0;
            int ch1 = s[sa[M] + pos + pre], ch2 = s[x + pre];
            if (pre >= len || ch1 > ch2) R = M;
            else L = M + 1;
        }
        int left = L;
        L = l, R = r;
        while (L < R) {
            int M = L + (R - L + 1) / 2;
            int pre = sa[M] + pos < n ? lcp(sa[M] + pos, x) : 0;
            int ch1 = s[sa[M] + pos + pre], ch2 = s[x + pre];
            if (pre >= len || ch1 < ch2) L = M;
            else R = M - 1;
        }
        int right = L;
        if (sa[left] + pos > n || lcp(sa[left] + pos, x) < len)
            return make_pair(0, -1);
        else
            l = left, r = right;
        pos += len;
    }
    return make_pair(l, r);
}
pair<int, int> go(int x, int y, int h, char c) {
    //设 sa 数组中区间 [x, y] 的最长公共前缀至少为 h，则返回区间中的后缀 i 满足 s[i + h] == c
```

```
    //若要识别 s 中所有以 c 开头的后缀，则调用 go(0, n - 1, 0, c)
    int L = x, R = y;
    while (L < R) {
        int M = L + (R - L) / 2;
        if (sa[M] + h < n && s[sa[M] + h] >= c) R = M;
        else L = M + 1;
    }
    int tmp = L;
    R = y;
    while (L < R) {
        int M = L + (R - L + 1) / 2;
        if (s[sa[M] + h] == c) L = M;
        else R = M - 1;
    }
    if (sa[L] + h >= n || s[sa[L] + h] != c)
        return { 0, -1 }; //空区间
    return { tmp, L };
}
pair<int, int> go(int x, int y, int h, const string &str) {
    //设 sa 数组中区间 [x, y] 的最长公共前缀至少为 h，则返回区间中的后缀 i 满足 s[i + h + j] ==
↪   str[j]
    //若要识别 s 中所有以 str 开头的后缀，则调用 go(0, n - 1, 0, str)
    for (auto c : str) {
        int L = x, R = y;
        while (L < R) {
            int M = L + (R - L) / 2;
            if (sa[M] + h < n && s[sa[M] + h] >= c) R = M;
            else L = M + 1;
        }
        x = L, R = y;
        while (L < R) {
            int M = L + (R - L + 1) / 2;
            if (s[sa[M] + h] == c) L = M;
            else R = M - 1;
        }
        if (sa[L] + h >= n || s[sa[L] + h] != c)
            return { 0, -1 }; //空区间
        y = L;
        h += 1;
    }
    return { x, y };
}
pair<int, int> go_rev(int x, int y, char c) {
    //设区间 [x, y] 表示 sa 中的一个区间，将 [x, y] 中的每个后缀都往前走一个字符 c,
    //得到一个新的连续区间并返回，注意这个新区间不一定是 [x, y] 的子区间
    //注意：因为 sa[0, n-1] 不包含空串，所以 go(0, n - 1, 0, c) != go_rev(0, n - 1, c)
```

```
        int L = 0, R = n - 1;
        while (L < R) {
            int M = L + (R - L) / 2;
            if (s[sa[M]] < c || s[sa[M]] == c && (sa[M] == n - 1 || rank[sa[M] + 1] < x))
                L = M + 1;
            else
                R = M;
        }
        int tmp = L;
        R = n - 1;
        while (L < R) {
            int M = L + (R - L + 1) / 2;
            if (s[sa[M]] > c || s[sa[M]] == c && rank[sa[M] + 1] > y)
                R = M - 1;
            else
                L = M;
        }
        if (s[sa[L]] != c || rank[sa[L] + 1] < x || rank[sa[L] + 1] > y)
            return { 0, -1 }; //空区间
        return { tmp, L };
    }
}arr;
int main() {
    static char s[maxn];
    scanf("%s", s);
    int n = strlen(s), Q;
    scanf("%d", &Q);
    arr.init(s);
    arr.build_sa();
    arr.getheight();
    arr.RMQ_init();
    while (Q--) {
        int x, y, a, b;
        scanf("%d %d %d %d", &x, &y, &a, &b); --x; --y; --a; --b;
        auto [L, R] = arr.locate(vector<pair<int, int>>{make_pair(x, y), make_pair(a, b)});
        printf("%d\n", R - L + 1);
    }
}
```

13.7 后缀自动机

```
/*
在 Trie 树中建立后缀自动机：
只需 DFS 一遍 Trie 树，对于 Trie 中的当前结点 u，它的父结点为 p，p 在后缀自动机中对应的结点为 v。
则将 v 作为 last，将 u 对应的字符插入后缀自动机中即可。（一定要先插入结点 u，再插入结点 u 在 Trie 树中
↪ 的子结点）
在建立广义后缀自动机时：
```

```
对于当前插入的字符串 s，若 s 的前缀 s[0, i] 也是之前某个字符串的前缀，则在插入 s[0, i] 时，字符 s[0]
↪  对应的结点出度为 0，
而对于 s[1, i] 中的每个字符 s[t]，它对应的结点在后缀自动机中入度为 1，来源于 s[t-1]。所以若从根结点开
↪  始走，不会走到这些结点，
故在后缀自动机中插入多个字符串不会破坏自动机的结构。同时对于 s[0, i] 中的每一个字符 s[t]，它对应的结
↪  点虽然不可达，
但是会在 parent 树中出现，所以广义后缀自动机中记录的信息是完全的。
设在自动机中输入字符串 str 后到达结点 u，并设 u 的 right 集合为 R（一个结点 right 集合为其子结点
↪  right 集合的并集），
则 R 记录了 str 在每个字符串中的所有的结束位置。
*/
//结论：从空串开始不断随机字符放到结尾直到 s 作为子串出现的期望次数 = 对 s 的所有 border i 计算 f(i)
↪  的和，f(i) 表示从头开始直接随机出 i 的概率的倒数
const int maxn = 210000; //maxn 应当大于最大字符串长度的 2 倍
//后缀自动机的基本数据
int par[maxn], len[maxn], go[maxn][26]; //min[s] = len[par[s]] + 1，当前结点表示的最小长度等于父结
↪  点的最大长度加一
int cur, root, last;
//用于求 parent 树的 DFS 序
int first[maxn], nxt[maxn], info[maxn], id[maxn], L[maxn], R[maxn]; //right 集合中的下标是子串 **
↪  右 ** 端点的位置
vector<int> seq;
//用于对后缀自动机进行拓扑排序
int cnt[maxn], arr[maxn];
int newstate(int length, int index = -1) {
    len[cur] = length;
    id[cur] = index;
    return cur++;
}
void init() {
    memset(go, 0, sizeof(go)); //如果数据组数很多，可能超时，应当改为 memset(go, 0, sizeof(go[0]) *
↪  (2 * n + 10));
    cur = 1;
    root = last = newstate(0);
}
void extend(int w, int index) {
    int x = last;
    int nx = newstate(len[x] + 1, index);
    while (x && !go[x][w])
        go[x][w] = nx, x = par[x];
    if (!x)
        par[nx] = root;
    else {
        int y = go[x][w];
        if (len[x] + 1 == len[y])
            par[nx] = y;
```

```
        else {
            int ny = newstate(len[x] + 1);
            memcpy(go[ny], go[y], sizeof(go[y]));
            par[ny] = par[y];
            par[y] = ny;
            par[nx] = ny;
            while (x && go[x][w] == y)
                go[x][w] = ny, x = par[x];
        }
    }
    last = nx;
    //每加入一个新的字符，新产生的本质不同的子串个数为 len[last] - len[par[last]]
}
set<int> travel(int x) { //采用按秩合并的方式处理得到结点 x 的 right 集合，最坏时间复杂度 nlog^2n
    set<int> right;
    vector<set<int> > child;
    for (int i = first[x]; i != -1; i = nxt[i]) {
        child.push_back(travel(info[i]));
    }
    sort(child.begin(), child.end(), [](const auto& A, const auto& B) {
        return A.size() > B.size();
        });
    if (!child.empty())
        right = std::move(child[0]);
    for (int i = 1; i < child.size(); ++i)
        for (auto item : child[i])
            right.insert(item);
    if (id[x] != -1)
        right.insert(id[x]);
    //此处通过前面得到的 right 集合来计算答案，注意此处代码的时间复杂度不能达到 O(n)，
    //否则总的时间复杂度可能会到 O(n^2) 甚至更高。
    return right;
}
void dfs(int x) {
    L[x] = seq.size();
    if (id[x] != -1)
        seq.push_back(id[x]);
    for (int i = first[x]; i != -1; i = nxt[i])
        dfs(info[i]);
    R[x] = seq.size() - 1;
}
void build() { //建树
    int sz = 0;
    seq.clear(); //可以在此处向 seq 中加入一个元素 0，从而使得 DFS 序区间从 1 开始
    memset(first, -1, sizeof(first));
    for (int i = 1; i < cur; ++i) {
```

```
        int p = par[i];
        info[sz] = i;
        nxt[sz] = first[p];
        first[p] = sz++;
    }
    //travel(root); //遍历子树求解问题
    dfs(root); //求 parent 树的 DFS 序
}
void topsort() { //对后缀自动机进行拓扑排序
    for (int i = 1; i < cur; ++i)
        cnt[i] = 0;
    for (int i = 1; i < cur; ++i)
        cnt[len[i]]++;
    for (int i = 1; i < cur; ++i)
        cnt[i] += cnt[i - 1];
    for (int i = cur - 1; i >= 1; --i) //arr: [1, cur)
        arr[cnt[len[i]]--] = i;
    for (int i = 2; i < cur; ++i) //只有建立广义后缀自动机的时候才需要这个循环
        if (par[arr[i - 1]] == arr[i])
            swap(arr[i - 1], arr[i]);
    /*
        拓扑排序后从叶结点开始递推更新 id 数组，注意循环完成后 id[0] 为所有结点 id 的或。
        for (int i = cur - 1; i >= 1; --i) {
            int j = arr[i];
            if (id[par[j]] == -1)
                id[par[j]] = id[j];
            else
                id[par[j]] |= id[j];
        }
    */
}
void insert(char* str, int id) { //通过多次调用 insert 函数向自动机中插入多个字符串，来建立广义后缀
↪   自动机。
    int n = strlen(str);
    last = root; //!!!
    for (int i = 0; i < n; ++i)
        extend(str[i] - 'a', id * 100 + i); //可以将第二个参数改为 pair，来记录插入字符来源于的字符
↪   串编号和下标。
}
const int maxlog = 25;
int anc[maxn][maxlog]; //anc[i][j] 表示结点 i 往上走 2^j 个点到达的点
map<int, int> trans; //id 函数的反函数
void preprocess() { //parent 树上倍增初始化
    trans.clear();
    for (int i = 1; i < cur; ++i)
        if (id[i] != -1)
```

```
            trans[id[i]] = i;
    for (int i = 1; i < cur; ++i) {
        anc[i][0] = par[i];
        for (int j = 1; (1 << (j - 1)) < cur; ++j)
            anc[i][j] = 0;
    }
    for (int j = 1; (1 << j) < cur; ++j) {
        for (int i = 1; i < cur; ++i) {
            if (anc[i][j - 1] != 0) {
                int a = anc[i][j - 1];
                anc[i][j] = anc[a][j - 1];
            }
        }
    }
}
int query(int id, int length) { //从标号 id 的结点往上走到 len[p] >= length 的深度最小的结点
    int p = trans[id];            //若要识别字符串的子串 [L, R]，则调用 query(id(R), R-L+1)，其中
↪   id(R) 表示下标 R 对应的 id
    for (int i = maxlog - 1; i >= 0; --i)
        if (len[anc[p][i]] >= length)
            p = anc[p][i];
    return p;
}
namespace Suffix_Tree { //建立反串的后缀树（后缀自动机的 parent 树是反串的后缀树）
    int pos[maxn]; //pos[x] 表示结点 x 对应的字符串在原串中的结束下标。
    int ch[maxn][26]; //ch[x][w] 表示结点 x 在后缀树中沿着字符 w 走到达的结点。
    char s[maxn];
    void extend(int w, int index) { //比之前的 extend 多了维护 pos 数组的代码。
        int x = last;
        int nx = newstate(len[x] + 1, index);
        pos[nx] = index;
        while (x && !go[x][w])
            go[x][w] = nx, x = par[x];
        if (!x)
            par[nx] = root;
        else {
            int y = go[x][w];
            if (len[x] + 1 == len[y])
                par[nx] = y;
            else {
                int ny = newstate(len[x] + 1);
                memcpy(go[ny], go[y], sizeof(go[y]));
                pos[ny] = pos[y];
                par[ny] = par[y];
                par[nx] = par[y] = ny;
                while (x && go[x][w] == y)
```

```
                go[x][w] = ny, x = par[x];
            }
        }
        last = nx;
    }
    void build(const char* str) { //对字符串 str 建立后缀自动机和反串的后缀树。
        strcpy(s, str);
        int n = strlen(s);
        last = root;
        for (int i = 0; i < n; ++i)
            extend(s[i] - 'a', i);
        for (int i = 2; i < cur; ++i) {
            int w = s[pos[i] - len[par[i]]] - 'a';
            ch[par[i]][w] = i;
        }
    }
    int walk(int x, int length, const char* str) { //从结点 x 开始沿着后缀树走字符串 str，如果终止
↪   在边内则会到子结点。
        int m = strlen(str);
        for (int i = 0; i < m && i < len[x] - length; ++i)
            if (str[i] != s[pos[x] - length - i])
                return 0;
        for (int i = len[x] - length; i < m; i += len[x] - len[par[x]]) {
            int c = str[i] - 'a';
            if (!ch[x][c])
                return 0;
            x = ch[x][c];
            for (int j = 0; j < len[x] - len[par[x]] && i + j < m; ++j)
                if (str[i + j] != s[pos[x] - len[par[x]] - j])
                    return 0;
        }
        return x;
    }
    int expand(int L, int R, const char* str) {
        int length = R - L + 1;
        int x = query(R, length);
        return walk(x, length, str);
    }
    /*
    建立的后缀树是后缀自动机的辅助数据结构，假设对字符串 s 建立后缀自动机，令结点 x 表示字符串 s[L,
↪   R]，
    则 go[x][c] 表示字符串 s[L, R] + c 对应的结点，ch[x][c] 表示字符串 c + s[L, R] 对应的结点。
    注意因为 ch 对应的边是后缀树中的边，边上是压缩的字符串而不是一个字符，所以连续向左扩展的时候应该
    调用 walk 函数或 expand 函数，而不是直接走 ch。
    walk 函数用字符串 str 向左扩展状态 x，因为一个状态可能对应多个字符串，所以用 length 来明确指明字
↪   符串的长度，
```

```
    length 应满足条件: len[par[x]] < length <= len[x]
    注意该函数会先扩展 str[0] 然后 str[1]、str[2]...，最终结果是将 str 的反串拼接在 x 所表示的字符串
↪   的左边得到的结果。
    expand 函数返回将 str 反向拼接在 s[L, R] 的左边所得到的状态。
    */
}
namespace MultiString {
    //label[i] 表示结点 i 的标记，times[i] 记录结点 i 在多少个字符串中出现
    //real[i] 表示结点 i 所能表示的最大长度（所有字符串在结点 i 所能表示的最大长度的最小值）
    int label[maxn], mx[maxn], times[maxn], real[maxn];
    void preprocess() { //初始化数据结构，要在 match 之前 extend 之后调用（必须在后缀自动机建立完成
↪   后调用）
        memset(label, -1, sizeof(label));
        memset(times, 0, sizeof(times));
        for (int i = 0; i <= cur; ++i)
            real[i] = len[i];
    }
    void match(const char* str, int index) {
        vector<pair<int, int>> nodes;
        int x = root, length = 0;
        for (const char* s = str; *s; ++s) {
            int c = *s - 'a';
            while (x != root && !go[x][c]) {
                x = par[x];
                length = len[x];
            }
            if (go[x][c]) {
                x = go[x][c];
                length += 1;
            }
            nodes.emplace_back(x, length);
        }
        sort(nodes.begin(), nodes.end(), [](auto a, auto b) {
            return len[a.first] > len[b.first];
        });
        vector<int> vec;
        for (auto pr : nodes) {
            int x = pr.first;
            while (x && label[x] != index) {
                vec.push_back(x);
                label[x] = index;
                mx[x] = 0;
                x = par[x];
            }
            mx[pr.first] = max(mx[pr.first], pr.second);
        }
```

```
        sort(vec.begin(), vec.end(), [](int a, int b) {
            return len[a] > len[b];
        });
        for (auto x : vec) {
            mx[par[x]] = max(mx[par[x]], len[par[x]]);
            real[x] = min(real[x], mx[x]);
            times[x] += 1;
        }
    }
    int longest_common_string(int n) {
        //设有 n 个字符串，任选一个构建后缀自动机，对其余的串调用 match 函数，
        //然后调用该函数，返回这些字符串的最长公共子串长度。
        int ans = 0;
        for (int i = 1; i < cur; ++i) if (times[i] == n - 1)
            ans = max(ans, real[i]);
        return ans;
    }
    long long number_of_common_string(int n) {
        //设有 n 个字符串，任选一个构建后缀自动机，对其余的串调用 match 函数，
        //然后调用该函数，返回这些字符串的本质不同的公共子串个数。
        long long ans = 0;
        for (int i = 1; i < cur; ++i) if (times[i] == n - 1)
            ans += max(0, real[i] - len[par[i]]);
        return ans;
    }
}
namespace Generalized_Suffix_Automaton { //广义后缀自动机
    //pos[i] 表示结点 i 对应的子串在 index==0 的字符串中最左侧出现的下标
    //times[i] 表示在 parent 树中结点 i 的子树中包含的类型数
    int par[maxn], len[maxn], go[maxn][26];
    int label[maxn], times[maxn];
    int n, cur, root, last, pos[maxn];
    int newstate(int length, int index = maxn) {
        len[cur] = length;
        pos[cur] = index;
        return cur++;
    }
    void init() { //n 表示字符串的个数，需要在外部初始化
        memset(label, -1, sizeof(label));
        cur = 1;
        root = last = newstate(0);
    }
    void extend(int w, int index) {
        int x = last;
        int nx = newstate(len[x] + 1, index);
        while (x && !go[x][w])
```

```
        go[x][w] = nx, x = par[x];
    if (!x)
        par[nx] = root;
    else {
        int y = go[x][w];
        if (len[x] + 1 == len[y])
            par[nx] = y;
        else {
            int ny = newstate(len[x] + 1, pos[y]);
            memcpy(go[ny], go[y], sizeof(go[y]));
            times[ny] = times[y];
            par[ny] = par[y];
            par[y] = ny;
            par[nx] = ny;
            while (x && go[x][w] == y)
                go[x][w] = ny, x = par[x];
        }
    }
    last = nx;
}
//多次调用 insert 建立广义自动机，index 从 0 开始
void insert(const string& s, int index) {
    last = root;
    int pre = cur;
    for (int i = 0; i < s.size(); ++i)
        extend(s[i] - 'a', index == 0 ? i : maxn);
    for (int i = pre; i < cur; ++i) {
        for (int x = i; x && label[x] != index; x = par[x]) {
            label[x] = index;
            times[x] += 1;
            pos[par[x]] = min(pos[par[x]], pos[x]);
        }
    }
}
//dp[i] 表示从结点 i 对应的任意字符串开始，能走到的本质不同的子串个数
long long dp[maxn];
long long dfs(int x) {
    if (dp[x] > 0)
        return dp[x];
    dp[x] = 0;
    for (int i = 0; i < 26; ++i) if (go[x][i] && times[go[x][i]] == n)
        dp[x] += dfs(go[x][i]) + 1;
    return dp[x];
}
void output(int x, long long k) { //输出第 k 小的本质不同公共子串
    if (k == 0) return;
```

```
        for (int i = 0; i < 26; ++i) if (go[x][i] && times[go[x][i]] == n) {
            if (k - dp[go[x][i]] - 1 > 0)
                k -= dp[go[x][i]] + 1;
            else {
                putchar('a' + i);
                output(go[x][i], k - 1);
                return;
            }
        }
    }
}
int main() {
    //freopen("in.txt", "r", stdin);
    static char s[maxn];
    init();
    scanf("%s", s);
    insert(s, 0);
    int n = 1;
    MultiString::preprocess();
    while (scanf("%s", s) == 1)
        MultiString::match(s, ++n);
    printf("%d\n", MultiString::longest_common_string(n));
    return 0;
}
```

13.8 回文自动机

```
/*
1.   len[i] 表示编号为 i 的节点对应的回文串的长度（一个节点表示一个回文串）。
2.   fail[i] 表示结点 i 失配以后跳转到的最长后缀回文串对应的结点。
3.   par[i] 表示结点 i 删除掉最外层的一个字符后得到的回文串对应的结点。
    例如结点 i 表示的回文串为 cabbac，则结点 par[i] 对应的回文串为 abba。
4.   node[i] 表示输入字符串中以下标 i 结尾的最长回文串对应的回文树结点。
5.   cnt[i] 表示结点 i 对应的字符串出现的次数（建树时求出的不是完全的，最后 calc() 函数跑一遍以后才
↪   是正确的）。
6.   num[i] 表示 以结点 i 表示的最长回文串的最右端点 为回文串结尾的回文串个数。
7.   若要判断回文自动机中是否包含回文串 P[0, m - 1]，则根据 P 的长度是奇数还是偶数来选择根结点（长度
↪   为偶数的回文串以 0 为根结点，长度为奇数的回文串以 1 为根结点），
    然后将字符串 P[m / 2, m - 1] 输入自动机，到达了结点 node，自动机包含回文串 P 当且仅当 len[node]
↪   == m。
8.   一般来说不需要对回文自动机进行 dfs，因为循环 for (int i = 2; i <= cur; ++i) 就可以按照 fail
↪   树的拓扑序
    （也是 par 树的拓扑序）访问结点。
*/
const int maxn = 110000;
const int sigma_size = 26;
int info[maxn], ch[maxn][sigma_size], fail[maxn], len[maxn];
```

```
int cnt[maxn], num[maxn], par[maxn], node[maxn], last, cur, sz;
void init() {
    memset(ch, 0, sizeof(ch));
    memset(cnt, 0, sizeof(cnt));
    fail[0] = 1;
    info[0] = -1;
    len[1] = -1;
    cur = last = 1;
    sz = 0;
}
void extend(int w) {
    int p = last;
    info[++sz] = w;
    while (info[sz - len[p] - 1] != w)
        p = fail[p];
    if (!ch[p][w]) {
        int u = ++cur, x = fail[p];
        while (info[sz - len[x] - 1] != w)
            x = fail[x];
        par[u] = p;
        len[u] = len[p] + 2;
        fail[u] = ch[x][w]; //(*)
        ch[p][w] = u;
        num[u] = num[fail[u]] + 1;
    }
    last = ch[p][w];
    cnt[last]++;
}
void insert(char* str) { //输入字符串下标从 0 开始
    int n = strlen(str);
    for (int i = 0; i < n; ++i) {
        extend(str[i] - 'a');
        node[i] = last;
    }
}
void calc() {
    //更新 fail 的地方只有 (*) 处，此时将新建的结点 u 连接到之前的结点上而不改变之前的结点的连接状态，
↪
    //这样，对于从 cur 到 2 的循环，就是对 fail 树按拓扑序逆序循环（也是对 par 树按拓扑序逆序循环）。
    for (int i = cur; i >= 2; --i)
        cnt[fail[i]] += cnt[i];
}
int main() {
    int A[] = { 1, 2, 3, 2, 1, 3, 3 }, B[] = { 3, 2, 1 };
    init();
    for (int i = 0; i < sizeof(A) / sizeof(*A); ++i)
```

```
        extend(A[i]);
    int st = 0; //0 偶数, 1 奇数
    for (int i = 0; i < sizeof(B) / sizeof(*B); ++i)
        st = ch[st][B[i]];
    printf("%d\n", len[st]);
    return 0;
}
```

13.9 回文串 Border

```
/*
定义 dif[x] = len[x] - len[fail[x]],
slink[x] 为 x 后缀链接路径上第一个 dif[x] dif[fail[x]] 的祖先。
*/
const int maxn = 110000;
const int sigma_size = 26;
int info[maxn], ch[maxn][sigma_size], fail[maxn], len[maxn], dif[maxn], slink[maxn];
int node[maxn], last, cur, sz;
void init() {
    memset(ch, 0, sizeof(ch));
    fail[0] = 1;
    info[0] = -1;
    len[1] = -1;
    cur = last = 1;
    sz = 0;
}
void extend(int w) {
    int p = last;
    info[++sz] = w;
    while (info[sz - len[p] - 1] != w)
        p = fail[p];
    if (!ch[p][w]) {
        int u = ++cur, x = fail[p];
        while (info[sz - len[x] - 1] != w)
            x = fail[x];
        len[u] = len[p] + 2;
        fail[u] = ch[x][w]; //(*)
        ch[p][w] = u;
        dif[u] = len[u] - len[fail[u]];
        if (dif[u] != dif[fail[u]])
            slink[u] = fail[u];
        else
            slink[u] = slink[fail[u]];
    }
    last = ch[p][w];
}
void insert(char* str) { //输入字符串下标从 0 开始
```

```
    int n = strlen(str);
    for (int i = 0; i < n; ++i) {
        extend(str[i] - 'a');
        node[i] = last;
    }
}
int main() {
    return 0;
}
```

13.10 双端回文自动机

```
/*
len[i] 表示编号为 i 的节点表示的回文串的长度（一个节点表示一个回文串）
fail[i] 表示结点 i 失配以后跳转到的最长后缀回文串对应的结点
cnt[i] 表示结点 i 对应的字符串出现的次数（建树时求出的不是完全的，最后 calc() 函数跑一遍以后才是正确
↪ 的）
num[i] 表示以结点 i 表示的最长回文串的最右端点为回文串结尾的回文串个数
num[i] 也表示结点 i 在 fail 树中的深度
若要判断回文自动机中是否包含回文串 P[0, m - 1]，则根据 P 的长度是奇数还是偶数来选择根结点（长度为偶数
↪ 的回文串以 0 为根结点，长度为奇数的回文串以 1 为根结点），
然后将字符串 P[m / 2, m - 1] 输入自动机，到达了结点 node，自动机包含回文串 P 当且仅当 len[node] ==
↪ m。
*/
const int maxn = 300010; //maxn 至少为字符串最大可能长度的两倍
long long ans = 0; //统计当前的字符串中有多少个回文串（位置不同即不同）
struct Palindromic_Tree {
    int ch[maxn][26], fail[maxn], info[maxn];
    int cnt[maxn], num[maxn], len[maxn];
    int last[2], cur, L, R;
    void init() {
        memset(ch, 0, sizeof(ch));
        memset(cnt, 0, sizeof(cnt));
        memset(info, -1, sizeof(info)); //info 的初始值为不会出现在输入字符串中的值即可
        len[1] = -1;
        cur = fail[0] = last[1] = 1;
        last[0] = 0;
        L = maxn / 2;
        R = L - 1;
    }
    int get_fail(int x, int back) {
        int k = back ? R : L;
        int tp = back ? 1 : -1;
        while (info[k - (len[x] + 1) * tp] != info[k])
            x = fail[x];
        return x;
    }
```

```
    void insert(int w, int back) { //back 为 1 则插入 w 到字符串的结尾，为 0 则插入到开头。
        if (back) info[++R] = w;
        else info[--L] = w;
        int p = get_fail(last[back], back);
        if (!ch[p][w]) {
            int u = ++cur;
            fail[u] = ch[get_fail(fail[p], back)][w];
            len[u] = len[p] + 2;
            ch[p][w] = u;
            num[u] = num[fail[u]] + 1;
        }
        last[back] = ch[p][w];
        if (len[last[back]] == R - L + 1)
            last[back ^ 1] = last[back];
        ans += num[last[back]]; //当前结点的插入会使得字符串中新增 num[last[back]] 个新的回文串
        cnt[last[back]]++;
    }
    void calc() {
        for (int i = cur; i >= 2; --i)
            cnt[fail[i]] += cnt[i];
    }
}tree;
int main(){
    //freopen("in.txt", "r", stdin);
    int n;
    while (scanf("%d", &n) == 1) {
        tree.init(); ans = 0;
        while (n--) {
            int tp;
            scanf("%d", &tp);
            if (tp == 1) {
                char c;
                scanf(" %c", &c);
                tree.insert(c - 'a', false);
            }
            else if (tp == 2) {
                char c;
                scanf(" %c", &c);
                tree.insert(c - 'a', true);
            }
            else if (tp == 3) {
                printf("%d\n", tree.cur - 1);
            }
            else {
                printf("%lld\n", ans);
            }
```

```
        }
    }
}
```

13.11 非势能分析回文自动机

```
/*
每次插入时间复杂度都为 O(1) 的回文自动机（不基于势能分析），可用于回滚莫队。
*/
const int maxn = 300007;
struct Palindromic_Tree {
    int cur, cnt[maxn], fail[maxn], ch[maxn][26], len[maxn], quick[maxn][26];
    int last[2], info[maxn * 2], L, R;
    void init() {
        fail[0] = cur = 1;
        len[1] = -1;
        last[0] = last[1] = 0;
        L = maxn;
        R = L - 1;
        memset(ch, 0, sizeof(ch)); //多组数据时应当在新建结点的时候清空
        memset(cnt, 0, sizeof(cnt));
        memset(info, -1, sizeof(info));
        for (int i = 0; i < 26; i++)
            quick[1][i] = quick[0][i] = 1;
    }
    void extend(int w, int back = 1) { //back 为 1 则插入 w 到字符串的结尾，为 0 则插入到开头。
        int tp = back ? -1 : 1;
        int x = back ? ++R : --L;
        int p = last[back];
        info[x] = w;
        if (info[x + tp * (len[p] + 1)] != info[x])
            p = quick[p][w];
        if (!ch[p][w]) {
            int u = ++cur;
            len[u] = len[p] + 2;
            int now = fail[p];
            if (info[x + tp * (len[now] + 1)] != info[x])
                now = quick[now][w];
            fail[u] = ch[now][w];
            memcpy(quick[u], quick[fail[u]], sizeof(quick[u]));
            quick[u][info[x + tp * len[fail[u]]]] = fail[u];
            ch[p][w] = u;
        }
        last[back] = ch[p][w];
        if (len[last[back]] == R - L + 1)
            last[back ^ 1] = last[back];//注意这里
        cnt[last[back]]++;
```

```
    }
} tree;
namespace Palindromic_Tree_Back {
    int info[maxn], ch[maxn][26], quick[maxn][26], fail[maxn], len[maxn];
    int last, cur, sz, n, m;
    vector<int> node;
    void init() {
        node.clear();
        memset(ch, 0, sizeof(ch[0]) * 5);
        fail[0] = 1;
        info[0] = len[1] = -1;
        cur = last = 1;
        sz = 0;
        for (int i = 0; i < 26; ++i)
            quick[1][i] = quick[0][i] = 1;
    }
    void extend(int w) { // 0 <= w <= sigma_size
        int p = last;
        info[++sz] = w;
        if (info[sz - len[p] - 1] != w)
            p = quick[p][w];
        if (!ch[p][w]) {
            int u = ++cur, x = fail[p];
            memset(ch[u], 0, sizeof(ch[u]));
            if (info[sz - len[x] - 1] != w)
                x = quick[x][w];
            len[u] = len[p] + 2;
            fail[u] = ch[x][w];
            ch[p][w] = u;
            memcpy(quick[u], quick[fail[u]], sizeof(quick[u]));
            quick[u][info[sz - len[fail[u]]]] = fail[u];
            //value[u] = (value[fail[u]] + (n - len[u] + 1) * pow_mod(26, n - len[u])) % mod;
        }
        last = ch[p][w];
        //answer = (answer + value[last]) % mod;
    }
    void rollback() { //回滚
        int x = node.back();
        node.pop_back();
        last = node.back();
        sz -= 1;
        //answer = (answer - value[x] + mod) % mod;
    }
}
char A[maxn];
int c[maxn], id[maxn];
```

```
int main() {
    scanf("%s", A + 1);
    int L = strlen(A + 1);
    tree.init();
    long long ans = 0;
    int T = L >> 1;
    for (int i = T + 1; i <= L; i++) tree.extend(A[i] - 'a', 1);
    for (int i = T; i; i--) tree.extend(A[i] - 'a', 0);
    for (int i = 2; i <= tree.cur; i++) c[tree.len[i]]++;
    for (int i = 1; i <= L; i++) c[i] += c[i - 1];
    for (int i = 2; i <= tree.cur; i++) id[c[tree.len[i]]--] = i;
    for (int i = tree.cur - 1; i; i--)  tree.cnt[tree.fail[id[i]]] += tree.cnt[id[i]];
    for (int i = 2; i <= tree.cur; i++) ans = max(ans, 1ll * tree.len[i] * tree.cnt[i]);
    printf("%lld\n", ans);
}
```

13.12 序列自动机

```
/*
序列自动机：能够识别给定序列的所有子序列（非连续子序列）
last[c] 表示字符 c 在序列中最后一次出现的位置对应的自动机结点
par[i] 表示下标 i 处的字符的上一次的出现位置对应的自动机结点
ch[i][x] 表示以下标 i 为起点，字符 x 的第一次出现位置对应的自动机结点
对于用 build 建立的自动机，自动机结点与原数组中的位置一一对应（自动机中下标 i 的位置对应原数组中下标
↪ i 的位置）。
对于用 extend 函数建立的自动机，自动机结点与原数组中的位置有常数值的偏移。
基本原理：
对于给定序列 A、B 判断 B 是否为 A 的子序列，只要从左往右找到字符 B[0] 在 A 中的第一次出现位置 A[i]，
↪ 然后再从 i + 1 开始找到 B[1] 的第一次出现位置，
以此类推。若按照如上算法在找到 B 之前已经到达了 A 的结尾，则 B 不是 A 的子序列，否则 B 是 A 的子序列。
*/
const int maxn = 110000;
const int sigma_size = 26;
int cur, root, par[maxn], last[sigma_size], ch[maxn][sigma_size];
void init()
{
    cur = root = 1;
    memset(ch, 0, sizeof(ch));
    for (int i = 0; i < sigma_size; ++i)
        last[i] = root;
}
void extend(int x) {
    par[++cur] = last[x];
    for (int c = 0; c < sigma_size; ++c)
        for (int i = last[c]; i && !ch[i][x]; i = par[i])
            ch[i][x] = cur;
    last[x] = cur;
```

```
}
void build(int A[], int n)    //若数组已知，则可以直接构造序列自动机，此时 root = 0.
{                             //注意：数组 A 的下标从 1 开始
    for (int i = n; i > 0; --i)
    {
        memcpy(ch[i - 1], ch[i], sizeof(ch[i]));
        ch[i - 1][A[i]] = i;
    }
}
int main()
{
    int A[] = { 0, 1, 1, 2, 3, 5, 1, 2 }, B[] = { 1, 1, 2, 5, 1 };
    //init();
    //for (int i = 0; i < sizeof(A) / sizeof(*A); ++i)
    //    extend(A[i]);
    //int st = root;
    build(A, sizeof(A) / sizeof(*A) - 1);
    int st = 0;
    for (int i = 0; i < sizeof(B) / sizeof(*B); ++i)
        st = ch[st][B[i]];
    printf("%d\n", st);
    return 0;
}
```

13.13 hash

```
const int maxn = 210000;
const unsigned long long x = 123;
unsigned long long H[maxn], xp[maxn];
char s[maxn];
int n;
void init()
{
    n = strlen(s);
    H[n] = 0;
    for (int i = n - 1; i >= 0; --i)
        H[i] = H[i + 1] * x + s[i] - 'a' + 1;
    xp[0] = 1;
    for (int i = 1; i <= n; ++i)
        xp[i] = xp[i - 1] * x;
}
unsigned long long Hash(int i, int L)
{
    return H[i] - H[i + L] * xp[L];
}
int main()
{
```

```
    strcpy(s, "abcdabcfg");
    init();
    cout << Hash(0, 3) << endl;
    cout << Hash(4, 3) << endl;
    cout << Hash(1, 3) << endl;
    return 0;
}
```

13.14 LCT 维护隐式后缀树

```
/*
在每次调用 extend 函数构建后缀自动机的时候，extend 函数会 logn 次调用 solve 函数，
这些次调用的 [L, R] 构成了连续区间 [1, index]。solve 函数的参数意思是对于以下标 index 结尾的字符串，
长度区间从 [L, R] 的这些子串上一次出现的位置都是 x（x == -1 表示之前没有出现过）。
可以通过在 LCT 上多维护一个标记，来维护上上次出现的位置。
对于询问母串中一个区间子串的问题，可以考虑用这个模型来解决。每次 extend(w, index) 的时候，
处理所有下标以 index 结尾的询问，并且动态维护左端点的询问值。
当前以 index 结尾的询问应该以 index - 1 的询问为基础，通过 solve 函数更新产生。
solve 函数每次会对一个左端点区间的询问值进行更新，可以考虑用线段树维护。
*/
const int maxn = 210000; //maxn 应当大于最大字符串长度的 2 倍
int par[maxn], len[maxn], go[maxn][26];
int ch[maxn][2], fa[maxn], stk[maxn], pos[maxn];
int cur, root, last;
long long ans = 0, base = 0;
void solve(int L, int R, int x, int index) {
    //当前算法对于一个字符串中的所有子串 求出了其本质不同的子串个数 的和。
    if (x == -1) {
        int t = R - L + 1;
        base += t * (t + 1) / 2;
    }
    else {
        base += (index - x) * (R - L + 1);
    }
}
inline bool son(int x) {
    return ch[fa[x]][1] == x;
}
inline bool isroot(int x) {
    return ch[fa[x]][1] != x && ch[fa[x]][0] != x;
}
inline void pushdown(int x) {
    pos[ch[x][0]] = pos[ch[x][1]] = pos[x];
}
void rotate(int x) {
    int y = fa[x], z = fa[y], c = son(x);
    if (!isroot(y))
```

```
        ch[z][son(y)] = x;
    fa[x] = z;
    ch[y][c] = ch[x][!c];
    fa[ch[y][c]] = y;
    ch[x][!c] = y;
    fa[y] = x;
}
void splay(int x) {
    int top = 0;
    stk[++top] = x;
    for (int i = x; !isroot(i); i = fa[i])
        stk[++top] = fa[i];
    while (top)
        pushdown(stk[top--]);
    for (int y = fa[x]; !isroot(x); rotate(x), y = fa[x]) if (!isroot(y))
        son(x) ^ son(y) ? rotate(x) : rotate(y);
}
void access(int nd, int index) {
    for (int y = nd, x = fa[nd]; y > 1; y = x, x = fa[x]) {
        if (x) {
            splay(x);
            ch[x][1] = y;
        }
        solve(len[x] + 1, len[y], pos[y], index);
    }
}
void cut(int x) { //断开结点 x 与它的父结点之间的边
    splay(x);
    fa[ch[x][0]] = fa[x];
    fa[x] = ch[x][0] = 0;
}
int newstate(int length) {
    len[cur] = length;
    return cur++;
}
void init() {
    memset(ch, 0, sizeof(ch));
    memset(fa, 0, sizeof(fa));
    memset(go, 0, sizeof(go));
    memset(pos, -1, sizeof(pos));
    cur = 1;
    root = last = newstate(0);
}
void extend(int w, int index) {
    int x = last;
    int nx = newstate(len[x] + 1);
```

```
    while (x && !go[x][w])
        go[x][w] = nx, x = par[x];
    if (!x)
        par[nx] = root, fa[nx] = root;
    else {
        int y = go[x][w];
        if (len[x] + 1 == len[y])
            par[nx] = y, fa[nx] = y;
        else {
            int ny = newstate(len[x] + 1);
            memcpy(go[ny], go[y], sizeof(go[y]));
            fa[ny] = par[y], par[ny] = par[y];
            cut(y);
            pos[ny] = pos[y];
            fa[y] = ny, par[y] = ny;
            fa[nx] = ny, par[nx] = ny;
            while (x && go[x][w] == y)
                go[x][w] = ny, x = par[x];
        }
    }
    last = nx;
    access(last, index);
    splay(last);
    pos[last] = index;
}
char A[maxn];
int main() {
    freopen("in.txt", "r", stdin);
    init();
    scanf("%s", A);
    int n = strlen(A);
    for (int i = 0; i < n; ++i) {
        extend(A[i] - 'A', i);
        ans += base;
    }
    printf("ans: %lld\n", ans);
    return 0;
}
```

13.15 区间本质不同子串个数

```
const int maxn = 210000; //maxn 应当大于最大字符串长度的 2 倍
int par[maxn], len[maxn], go[maxn][26];
int ch[maxn][2], fa[maxn], stk[maxn], pos[maxn];
int cur, root, last;
struct stnode {
    int l, r;
```

```
    int lc, rc;
    long long a, d;
};
struct segment {
    stnode tree[maxn * 4];
    int tc;
    int root;
    int init_tree(int l, int r) {
        int mid = (l + r) / 2;
        int pos = ++tc;
        tree[pos].l = l;
        tree[pos].r = r;
        tree[pos].a = tree[pos].d = 0;
        if (l != r) {
            tree[pos].lc = init_tree(l, mid);
            tree[pos].rc = init_tree(mid + 1, r);
        }
        return pos;
    }
    long long query_tree(int pos, int p) {
        int mid = (tree[pos].l + tree[pos].r) / 2;
        if (tree[pos].l == tree[pos].r) {
            return tree[pos].a;
        }
        else {
            if (p <= mid) {
                return query_tree(tree[pos].lc, p) + tree[pos].a + (p - tree[pos].l) *
↪   tree[pos].d;
            }
            else {
                return query_tree(tree[pos].rc, p) + tree[pos].a + (p - tree[pos].l) *
↪   tree[pos].d;
            }
        }
    }
    void update_tree(int pos, int l, int r, long long a, long long d) {
        int mid = (tree[pos].l + tree[pos].r) / 2;
        if (tree[pos].l == l and r == tree[pos].r) {
            tree[pos].a += a;
            tree[pos].d += d;
        }
        else {
            if (r <= mid) {
                update_tree(tree[pos].lc, l, r, a, d);
            }
            else if (l <= mid) {
```

```
                update_tree(tree[pos].lc, l, mid, a, d);
                update_tree(tree[pos].rc, mid + 1, r, a + (mid + 1 - l) * d, d);
            }
            else {
                update_tree(tree[pos].rc, l, r, a, d);
            }
        }
    }
    long long query(int p) {
        return query_tree(root, p + 1);
    }
    void update(int l, int r, long long a, long long d) {
        if (l <= r)
            update_tree(root, l + 1, r + 1, a, d); //下标从 1 开始
    }
    void init(int x) {
        tc = 0;
        root = init_tree(1, x);
    }
} tree;
void solve(int L, int R, int x, int index) {
    if (x == -1) {
        tree.update(index - R + 1, index - L + 1, R - L + 1, -1);
    }
    else {
        int length = min(R - L + 1, index - x);
        tree.update(x - R + 2, x - R + length, 1, 1);
        tree.update(index - L - length + 3, index - L + 1, length - 1, -1);
        tree.update(x - R + length + 1, index - L - length + 2, length, 0);
    }
}
inline bool son(int x) {
    return ch[fa[x]][1] == x;
}
inline bool isroot(int x) {
    return ch[fa[x]][1] != x && ch[fa[x]][0] != x;
}
inline void pushdown(int x) {
    pos[ch[x][0]] = pos[ch[x][1]] = pos[x];
}
void rotate(int x) {
    int y = fa[x], z = fa[y], c = son(x);
    if (!isroot(y))
        ch[z][son(y)] = x;
    fa[x] = z;
    ch[y][c] = ch[x][!c];
```

```
    fa[ch[y][c]] = y;
    ch[x][!c] = y;
    fa[y] = x;
}
void splay(int x) {
    int top = 0;
    stk[++top] = x;
    for (int i = x; !isroot(i); i = fa[i])
        stk[++top] = fa[i];
    while (top)
        pushdown(stk[top--]);
    for (int y = fa[x]; !isroot(x); rotate(x), y = fa[x]) if (!isroot(y))
        son(x) ^ son(y) ? rotate(x) : rotate(y);
}
void access(int nd, int index) {
    for (int y = nd, x = fa[nd]; y > 1; y = x, x = fa[x]) {
        if (x) {
            splay(x);
            ch[x][1] = y;
        }
        solve(len[x] + 1, len[y], pos[y], index);
    }
}
void cut(int x) { //断开结点 x 与它的父结点之间的边
    splay(x);
    fa[ch[x][0]] = fa[x];
    fa[x] = ch[x][0] = 0;
}
int newstate(int length) {
    len[cur] = length;
    return cur++;
}
void init() {
    memset(par, 0, sizeof(par));
    memset(ch, 0, sizeof(ch));
    memset(fa, 0, sizeof(fa));
    memset(go, 0, sizeof(go));
    memset(pos, -1, sizeof(pos));
    cur = 1;
    root = last = newstate(0);
}
void extend(int w, int index) {
    int x = last;
    int nx = newstate(len[x] + 1);
    while (x && !go[x][w])
        go[x][w] = nx, x = par[x];
```

```
    if (!x)
        par[nx] = root, fa[nx] = root;
    else {
        int y = go[x][w];
        if (len[x] + 1 == len[y])
            par[nx] = y, fa[nx] = y;
        else {
            int ny = newstate(len[x] + 1);
            memcpy(go[ny], go[y], sizeof(go[y]));
            fa[ny] = par[y], par[ny] = par[y];
            cut(y);
            pos[ny] = pos[y];
            fa[y] = ny, par[y] = ny;
            fa[nx] = ny, par[nx] = ny;
            while (x && go[x][w] == y)
                go[x][w] = ny, x = par[x];
        }
    }
    last = nx;
    access(last, index);
    splay(last);
    pos[last] = index;
}
char s[maxn];
vector<pair<int, int>> Q[maxn]; //多组数据时要清空
long long answer[maxn];
int main() {
    //freopen("in.txt", "r", stdin);
    scanf("%s", s);
    int n = strlen(s);
    init();
    tree.init(n + 10);
    int m;
    scanf("%d", &m);
    for (int i = 0; i < m; ++i) {
        int L, R;
        scanf("%d %d", &L, &R);
        --L; --R;
        Q[R].emplace_back(L, i);
    }
    for (int i = 0; i < n; ++i) {
        extend(s[i] - 'a', i);
        for (auto [L, id] : Q[i])
            answer[id] = tree.query(L);
    }
    for (int i = 0; i < m; ++i)
```

```
        printf("%lld\n", answer[i]);
    return 0;
}
```

13.16 Lyndon 分解

```
//Lyndon 分解将一个字符串分解为若干个连续的子串 S1, S2, ..., Sn, 并且满足
//S1 >= S2 >= ... >= Sn, 并且所有这些子串都满足最小后缀是其本身。
//该函数返回 Lyndon 分解成的所有子串的起始下标。
vector<int> Lyndon(const char* s) {
    int n = strlen(s);
    vector<int> res;
    for (int i = 0; i < n; ) {
        int j = i, k = i + 1;
        while (k < n && s[j] <= s[k])
            j = (s[j] == s[k++] ? j + 1 : i);
        while (i <= j) {
            res.push_back(i);
            i += k - j;
        }
    }
    return res;
}
const int maxn = 1110000;
int pos[maxn]; //pos[i] 表示前缀 s[0, i] 的最小后缀的下标
void preprocess(const char* s) {
    int n = strlen(s);
    for (int i = 0; i < n; ) {
        int j = i, k = i + 1;
        pos[i] = i;
        while (k < n && s[j] <= s[k]) {
            if (s[j] < s[k]) {
                pos[k] = j = i;
            }
            else {
                pos[k] = pos[j] + k - j;
                j += 1;
            }
            k += 1;
        }
        while (i <= j)
            i += k - j;
    }
}
char s[maxn];
int main() { //hdu6761
    const int mod = 1e9 + 7;
```

```
    int T;
    scanf("%d", &T);
    while (T--) {
        scanf("%s", s);
        preprocess(s);
        int n = strlen(s);
        long long ans = 0, val = 1;
        for (int i = 0; i < n; ++i) {
            ans = (ans + (pos[i] + 1) * val) % mod;
            val = val * 1112 % mod;
        }
        printf("%lld\n", ans);
    }
    return 0;
}
```

13.17 后缀平衡树

```
const int maxn = 1110000;
const double alpha = 0.6;
struct SuffixBalancedTree {
    string data;
    double val[maxn], left, right;
    int sz[maxn], ch[maxn][2], root;
    int* pos, length, A[maxn];
    void init() {
        data.resize(1);
        root = 0;
    }
    void pushup(int x) {
        sz[x] = sz[ch[x][0]] + sz[ch[x][1]] + 1;
    }
    void dfs(int x) {
        if (!x) return;
        dfs(ch[x][0]);
        A[++length] = x;
        dfs(ch[x][1]);
    }
    int build(int a, int b, double L, double R) {
        if (a > b) return 0;
        int mid = (a + b) >> 1;
        int x = A[mid];
        sz[x] = b - a + 1;
        val[x] = (L + R) / 2;
        ch[x][0] = build(a, mid - 1, L, val[x]);
        ch[x][1] = build(mid + 1, b, val[x], R);
        return x;
```

```
}
void add(int& x, double L, double R) {
    if (!x) {
        x = data.size() - 1;
        ch[x][0] = ch[x][1] = 0;
        val[x] = (L + R) / 2;
        sz[x] = 1;
        return;
    }
    int y = data.size() - 1;
    if (data[y] < data[x] || data[y] == data[x] && val[y - 1] < val[x - 1])
        add(ch[x][0], L, val[x]);
    else
        add(ch[x][1], val[x], R);
    pushup(x);
    if (sz[ch[x][0]] > sz[x] * alpha || sz[ch[x][1]] > sz[x] * alpha)
        pos = &x, left = L, right = R;
}
void push_front(char c) { //向开头添加一个新的字符之后会改变其他位置的下标
    data.push_back(c);
    pos = nullptr;
    add(root, 0, 1);
    if (pos) {
        length = 0;
        dfs(*pos);
        *pos = build(1, length, left, right);
    }
}
int merge(int x, int y) {
    if (!x || !y)
        return x | y;
    if (sz[x] > sz[y]) {
        ch[x][1] = merge(ch[x][1], y);
        pushup(x);
        return x;
    }
    else {
        ch[y][0] = merge(x, ch[y][0]);
        pushup(y);
        return y;
    }
}
void del(int& x) {
    const int y = data.size() - 1;
    sz[x] -= 1;
    if (x == y)
```

```
            x = merge(ch[x][0], ch[x][1]);
        else if (val[y] < val[x])
            del(ch[x][0]);
        else
            del(ch[x][1]);
    }
    void pop_front() {
        del(root);
        data.pop_back();
    }
    double weight(int index) { //返回下标 index 的权值，权值越小，字典序越小（下标从 1 开始）
        return val[data.size() - index];
    }
    int rank(int index) { //返回下标 index 的排名（下标从 1 开始）
        double key = weight(index);
        int ret = 0, x = root;
        while (x) {
            if (key < val[x])
                x = ch[x][0];
            else
                ret += sz[ch[x][0]] + 1, x = ch[x][1];
        }
        return ret;
    }
    int rank(const char *s) { //返回字典序小于 s 的后缀的个数
        int ret = 0, x = root, n = strlen(s);
        while (x) {
            int L = min(x, n) + 1;
            int flag = 0;
            for (int i = 0; i < L; ++i) {
                if (s[i] != data[x - i]) {
                    flag = s[i] - data[x - i];
                    break;
                }
            }
            if (flag <= 0)
                x = ch[x][0];
            else
                ret += sz[ch[x][0]] + 1, x = ch[x][1];
        }
        return ret;
    }
}tree;
int main() {
    return 0;
}
```

13.18 模糊匹配

```cpp
/*
init(s) 用 s 作为主串来初始化
query(t) 查询 t 在 s 中出现的次数（只要不同字符的个数不超过一个就算匹配）
*/
const int maxn = 410000;
int n, sz, root[maxn];
pair<int, int> range[maxn];
struct SuffixArray {
    int sa[maxn], rank[maxn];
    int t[maxn], t2[maxn], c[maxn];
    char s[maxn];
    void build_sa(const char *str, int m = 256) {
        strcpy(s, str);
        memset(t, 0, sizeof(int) * (2 * n + 10));
        memset(t2, 0, sizeof(int) * (2 * n + 10));
        int* x = t, * y = t2;
        for (int i = 0; i < m; ++i) c[i] = 0;
        for (int i = 0; i < n; ++i) c[x[i] = s[i]]++;
        for (int i = 1; i < m; ++i) c[i] += c[i - 1];
        for (int i = n - 1; i >= 0; --i) sa[--c[x[i]]] = i;
        for (int k = 1; k <= n; k <<= 1) {
            int p = 0;
            for (int i = n - 1; i >= n - k; --i) y[p++] = i;
            for (int i = 0; i < n; ++i) if (sa[i] >= k) y[p++] = sa[i] - k;
            for (int i = 0; i < m; ++i) c[i] = 0;
            for (int i = 0; i < n; ++i) c[x[y[i]]]++;
            for (int i = 1; i < m; ++i) c[i] += c[i - 1];
            for (int i = n - 1; i >= 0; --i) sa[--c[x[y[i]]]] = y[i];
            swap(x, y);
            p = 1; x[sa[0]] = 0;
            for (int i = 1; i < n; ++i)
                x[sa[i]] = y[sa[i - 1]] == y[sa[i]] && y[sa[i - 1] + k] == y[sa[i] + k] ? p - 1 :
↪   p++;
            if (p >= n) break;
            m = p;
        }
    }
    void calc() { //将空串放在 sa 数组的最前面，并计算 rank 数组
        for (int i = n; i >= 1; --i)
            sa[i] = sa[i - 1];
        sa[0] = n;
        for (int i = 0; i <= n; ++i)
            rank[sa[i]] = i;
    }
    pair<int, int> advance(pair<int, int> range, char c) {
```

```
        //设区间 [x, y] 表示 sa 中的一个区间，返回将 [x, y] 中的每个后缀都往前走一个字符 c 得到的区
↪   间
        auto [x, y] = range;
        int L = 1, R = n;
        while (L < R) {
            int M = L + (R - L) / 2;
            if (s[sa[M]] < c || s[sa[M]] == c && rank[sa[M] + 1] < x)
                L = M + 1;
            else
                R = M;
        }
        int tmp = L;
        R = n;
        while (L < R) {
            int M = L + (R - L + 1) / 2;
            if (s[sa[M]] > c || s[sa[M]] == c && rank[sa[M] + 1] > y)
                R = M - 1;
            else
                L = M;
        }
        if (s[sa[L]] != c || rank[sa[L] + 1] < x || rank[sa[L] + 1] > y)
            return { 0, -1 }; //空区间
        return { tmp, L };
    }
}pre, suf;
struct node {
    int l, r, v;
} T[maxn * 25];
void ins(int& i, int l, int r, int p) {
    int m = (l + r) >> 1;
    T[++sz] = T[i]; i = sz;
    T[i].v++;
    if (l == r) return;
    if (p <= m) ins(T[i].l, l, m, p);
    else ins(T[i].r, m + 1, r, p);
}
int ask(int x, int y, int v) {
    int l = 0, r = n, k = 0;
    x = root[x - 1], y = root[y];
    int p = v - 1;
    if (p < 0) return 0;
    while (l < r) {
        int m = (l + r) >> 1, t = T[T[y].l].v - T[T[x].l].v;
        if (p <= m)
            x = T[x].l, y = T[y].l, r = m;
        else
```

```
            x = T[x].r, y = T[y].r, l = m + 1, k += t;
    }
    k += T[y].v - T[x].v;
    return k;
}
void init(string s) {
    ::sz = 0; ::n = s.size();
    pre.build_sa(s.c_str());
    pre.calc();
    reverse(s.begin(), s.end());
    suf.build_sa(s.c_str());
    suf.calc();
    int tree = 0;
    for (int i = 0; i <= n; ++i) {
        int j = n - suf.sa[i] - 1;
        if (j + 2 <= n)
            ins(tree, 0, n, pre.rank[j + 2]);
        root[i + 1] = tree;
    }
}
long long query(string t) {
    long long ans = 0;
    range[t.size()] = { 0, n };
    for (int i = t.size() - 1; i >= 0; --i) {
        if (range[i + 1].first > range[i + 1].second)
            range[i] = range[i + 1];
        else
            range[i] = pre.advance(range[i + 1], t[i]);
    }
    pair<int, int> now(0, n);
    for (int i = 0; i < t.size() && now.first <= now.second; ++i) {
        if (range[i + 1].first <= range[i + 1].second) {
            ans += ask(now.first + 1, now.second + 1, range[i + 1].second + 1);
            ans -= ask(now.first + 1, now.second + 1, range[i + 1].first);
        }
        now = suf.advance(now, t[i]);
    }
    if (now.first <= now.second)
        ans -= (now.second - now.first + 1) * (t.size() - 1);
    return ans;
}
int main() {
    //freopen("in.txt", "r", stdin);
    int T, Q;
    cin >> T;
    while (T--) {
```

```
        string s, t;
        cin >> s;
        init(s);
        cin >> Q;
        while (Q--) {
            cin >> t;
            printf("%lld\n", query(t));
        }
    }
    return 0;
}
```

13.19 基于后缀自动机构建后缀树

```
const int maxn = 2010000;
int par[maxn], len[maxn], cur, root, last; //par[x] 是后缀树上 x 的父结点
int ch[maxn][26]; //ch[x][w] 表示结点 x 在后缀树中沿着字符 w 走到达的结点。
bool issuf[maxn]; //issuf[x] 记录结点 x 是否表示一个后缀，若是则 pos[x] 就是后缀的下标。
int pos[maxn]; //pos[x] 表示结点 x 对应的字符串在原串中的起始下标。
int node[maxn]; //node[i] 表示下标 i 开始的后缀在树中的结点编号。
int newstate(int length, int index) {
    len[cur] = length;
    pos[cur] = index;
    return cur++;
}
void init() {
    memset(ch, 0, sizeof(ch));
    cur = 1;
    root = last = newstate(0, -1);
}
void extend(int w, int index) {
    int x = last;
    int nx = newstate(len[x] + 1, index);
    node[index] = nx;
    issuf[nx] = true;
    while (x && !ch[x][w])
        ch[x][w] = nx, x = par[x];
    if (!x)
        par[nx] = root;
    else {
        int y = ch[x][w];
        if (len[x] + 1 == len[y])
            par[nx] = y;
        else {
            int ny = newstate(len[x] + 1, pos[y]);
            memcpy(ch[ny], ch[y], sizeof(ch[y]));
            par[ny] = par[y];
```

```
            par[nx] = par[y] = ny;
            while (x && ch[x][w] == y)
                ch[x][w] = ny, x = par[x];
        }
    }
    last = nx;
}
void build(const char* s) {
    int n = strlen(s);
    for (int i = n - 1; i >= 0; --i)
        extend(s[i] - 'a', i);
    memset(ch, 0, sizeof(ch[0]) * (cur + 10));
    for (int i = 2; i < cur; ++i) {
        int w = s[pos[i] + len[par[i]]] - 'a';
        ch[par[i]][w] = i;
    }
}
const int maxlog = 23;
int anc[maxn][maxlog]; //anc[i][j] 表示结点 i 往上走 2^j 个点到达的点
void preprocess() { //后缀树上倍增初始化
    for (int i = 1; i < cur; ++i) {
        anc[i][0] = par[i];
        for (int j = 1; (1 << (j - 1)) < cur; ++j)
            anc[i][j] = 0;
    }
    for (int j = 1; (1 << j) < cur; ++j) {
        for (int i = 1; i < cur; ++i) {
            if (anc[i][j - 1] != 0) {
                int a = anc[i][j - 1];
                anc[i][j] = anc[a][j - 1];
            }
        }
    }
}
int query(int index, int length) { //从下标 index 的后缀往上走到 len[p] >= length 的深度最小的结点
    int p = node[index];           //若要识别字符串的子串 [L, R]，则调用 query(L, R-L+1)
    for (int i = maxlog - 1; i >= 0; --i)
        if (len[anc[p][i]] >= length)
            p = anc[p][i];
    return p;
}
int main() {
    return 0;
}
```

14 最小树形图

14.1 最小树形图

```
// 固定根的最小树型图，邻接矩阵写法
struct MDST {
  int n;
  int w[maxn][maxn]; // 边权
  int vis[maxn];     // 访问标记，仅用来判断无解
  int ans;           // 计算答案
  int removed[maxn]; // 每个点是否被删除
  int cid[maxn];     // 所在圈编号
  int pre[maxn];     // 最小入边的起点
  int iw[maxn];      // 最小入边的权值
  int max_cid;       // 最大圈编号
  void init(int n) {
    this->n = n;
    for(int i = 0; i < n; i++)
      for(int j = 0; j < n; j++) w[i][j] = INF;
  }
  void AddEdge(int u, int v, int cost) {
    w[u][v] = min(w[u][v], cost); // 重边取权最小的
  }
  // 从 s 出发能到达多少个结点
  int dfs(int s) {
    vis[s] = 1;
    int ans = 1;
    for(int i = 0; i < n; i++)
      if(!vis[i] && w[s][i] < INF) ans += dfs(i);
    return ans;
  }
  // 从 u 出发沿着 pre 指针找圈
  bool cycle(int u) {
    max_cid++;
    int v = u;
    while(cid[v] != max_cid) { cid[v] = max_cid; v = pre[v]; }
    return v == u;
  }
  // 计算 u 的最小入弧，入弧起点不得在圈 c 中
  void update(int u) {
    iw[u] = INF;
    for(int i = 0; i < n; i++)
      if(!removed[i] && w[i][u] < iw[u]) {
        iw[u] = w[i][u];
        pre[u] = i;
      }
  }
```

```
  // 根结点为 s，如果失败则返回 false
  bool solve(int s) {
    memset(vis, 0, sizeof(vis));
    if(dfs(s) != n) return false;
    memset(removed, 0, sizeof(removed));
    memset(cid, 0, sizeof(cid)); //注意：是将 cid 清空，而不是 pre
    for(int u = 0; u < n; u++) update(u);
    pre[s] = s; iw[s] = 0; // 根结点特殊处理
    ans = max_cid = 0;
    for(;;) {
      bool have_cycle = false;
      for(int u = 0; u < n; u++) if(u != s && !removed[u] && cycle(u)){
        have_cycle = true;
        // 以下代码缩圈，圈上除了 u 之外的结点均删除
        int v = u;
        do {
          if(v != u) removed[v] = 1;
          ans += iw[v];
          // 对于圈外点 i，把边 i->v 改成 i->u（并调整权值）；v->i 改为 u->i
          // 注意圈上可能还有一个 v'使得 i->v'或者 v'->i 存在，因此只保留权值最小的 i->u 和 u->i
          for(int i = 0; i < n; i++) if(cid[i] != cid[u] && !removed[i]) {
            if(w[i][v] < INF) w[i][u] = min(w[i][u], w[i][v]-iw[v]);
            w[u][i] = min(w[u][i], w[v][i]);
            if(pre[i] == v) pre[i] = u;
          }
          v = pre[v];
        } while(v != u);
        update(u);
        break;
      }
      if(!have_cycle) break;
    }
    for(int i = 0; i < n; i++)
      if(!removed[i]) ans += iw[i];
    return true;
  }
};
```

15 支配树

15.1 支配树

```
/*
在调用 dominator_tree::add_edge 之前要先调用 init 函数进行初始化，init 的参数 n 表示支配树中有 n 的
↪  点，编号为 1-n。
idom[x] 表示点 x 的最近支配点，idom[root] == 0。
head 保存为原图的链表。
```

```
back 保存为反图的链表。
dfn[x] 表示结点 x 的 DFS 序。
id[i] 表示 DFS 序为 i 的结点编号。
tarjan(s) 可以求出以 s 为起点的支配树。
性质：dfn[x] > dfn[idom[x]]，即结点 x 的 DFS 序一定大于它的支配点的 DFS 序
    因此可以按照 DFS 序从小到大枚举点 x，此时它在支配树上的父结点已经计算完毕。
    for (int i = 1; i <= n; ++i) {
        int x = id[i];
        ans[x] = F(ans[idom[x]], x);
    }
*/
const int maxn = 210000;
const int maxedges = 310000;
struct dominator_tree {
    int n, cnt, tot, head[maxn], fa[maxn], p[maxn], top[maxn], back[maxn], val[maxn], dfn[maxn],
↪   id[maxn], semi[maxn], idom[maxn];
    struct edge {
        int to, next;
    }e[maxedges * 3]; //min: maxedges * 2 + maxn
    inline bool cmp(int x, int y) {
        return dfn[semi[x]] > dfn[semi[y]];
    }
    inline void ins(int* first, int from, int to) {
        e[++cnt].to = to;
        e[cnt].next = first[from];
        first[from] = cnt;
    }
    void add_edge(int x, int y) {
        ins(head, x, y);
        ins(back, y, x);
    }
    void dfs(int x) {
        id[dfn[x] = ++tot] = x;
        for (int i = head[x]; i; i = e[i].next)
            if (!dfn[e[i].to])
                fa[e[i].to] = x, dfs(e[i].to);
    }
    int find(int x) {
        if (p[x] == x) return x;
        int y = find(p[x]);
        if (cmp(val[x], val[p[x]]))
            val[x] = val[p[x]];
        return p[x] = y;
    }
    void tarjan(int s) {
        dfs(s);
```

```
        for (int i = tot; i > 1; --i) {
            int u = id[i];
            for (int j = back[u]; j; j = e[j].next) {
                if (dfn[e[j].to]) {
                    find(e[j].to);
                    if (cmp(u, val[e[j].to]))
                        semi[u] = semi[val[e[j].to]];
                }
            }
            ins(top, semi[u], u);
            p[u] = fa[u];
            u = fa[u];
            for (int j = top[u]; j; j = e[j].next) {
                find(e[j].to);
                if (semi[val[e[j].to]] == semi[u])
                    idom[e[j].to] = u;
                else
                    idom[e[j].to] = val[e[j].to];
            }
            top[u] = 0;
        }
        for (int i = 2; i <= tot; ++i) {
            int x = id[i];
            if (idom[x] != semi[x])
                idom[x] = idom[idom[x]];
        }
    }
    void init(int n) {
        this->n = n;
        cnt = tot = 0;
        memset(head, 0, sizeof(int) * (n + 2));
        memset(back, 0, sizeof(int) * (n + 2));
        memset(top, 0, sizeof(int) * (n + 2));
        memset(dfn, 0, sizeof(int) * (n + 2));
        memset(idom, 0, sizeof(int) * (n + 2));
        for (int i = 1; i <= n; ++i)
            val[i] = semi[i] = p[i] = i;
    }
}tree;
long long ans[maxn];
int main() {
    //freopen("in.txt", "r", stdin);
    int n, m;
    scanf("%d %d", &n, &m);
    tree.init(n);
    for (int i = 1; i <= m; ++i) {
```

```
        int u, v;
        scanf("%d %d", &u, &v);
        tree.add_edge(u, v);
    }
    tree.tarjan(1);
    for (int i = 1; i <= n; ++i)
        ans[i] = 1;
    for (int i = n; i >= 1; --i) {
        int x = tree.id[i];
        ans[tree.idom[x]] += ans[x];
    }
    for (int i = 1; i <= n; ++i)
        printf("%d ", ans[i]);
    printf("\n");
    return 0;
}
```

16 人工智能

16.1 主成分分析

```
const int maxn = 210000;
const int maxdim = 1001;
const double eps = 1e-8;
using matrix = double[maxdim][maxdim];
using vec = array<double, maxdim>;
using pair_t = pair<double, vec>;
struct PCA {
    matrix A, V;
    int column[maxdim], n;
    void update(int r, int c, double v) {
        A[r][c] = v;
        if (column[r] == c || fabs(A[r][c]) > fabs(A[r][column[r]])) {
            for (int i = 0; i < n; ++i) if (i != r)
                if (fabs(A[r][i]) > fabs(A[r][column[r]]))
                    column[r] = i;
        }
    }
    void Jacobi() {
        for (int i = 0; i < n; ++i) {
            for (int j = 0; j < n; ++j)
                V[i][j] = 0;
            V[i][i] = 1;
            column[i] = (i == 0 ? 1 : 0);
        }
        for (int i = 0; i < n; ++i)
            for (int j = 0; j < n; ++j)
```

```
            if (j != i && fabs(A[i][j]) > fabs(A[i][column[i]]))
                column[i] = j;
    for (int T = 0; ; ++T) { //迭代次数限制
        int x, y;
        double val = 0;
        for (int i = 0; i < n; ++i)
            if (fabs(A[i][column[i]]) > val)
                val = fabs(A[i][column[i]]), x = i, y = column[i];
        if (val < eps) //精度限制
            break;
        double phi = atan2(-2 * A[x][y], A[y][y] - A[x][x]) / 2;
        double sinp = sin(phi), cosp = cos(phi);
        for (int i = 0; i < n; ++i) if (i != x && i != y) {
            double a = A[x][i] * cosp + A[y][i] * sinp;
            double b = A[x][i] * -sinp + A[y][i] * cosp;
            update(x, i, a);
            update(y, i, b);
        }
        for (int i = 0; i < n; ++i) if (i != x && i != y) {
            double a = A[i][x] * cosp + A[i][y] * sinp;
            double b = A[i][x] * -sinp + A[i][y] * cosp;
            update(i, x, a);
            update(i, y, b);
        }
        for (int i = 0; i < n; ++i) {
            double a = V[i][x] * cosp + V[i][y] * sinp;
            double b = V[i][x] * -sinp + V[i][y] * cosp;
            V[i][x] = a, V[i][y] = b;
        }
        double a = A[x][x] * cosp * cosp + A[y][y] * sinp * sinp + 2 * A[x][y] * cosp * sinp;
        double b = A[x][x] * sinp * sinp + A[y][y] * cosp * cosp - 2 * A[x][y] * cosp * sinp;
        double tmp = (A[y][y] - A[x][x]) * sin(2 * phi) / 2 + A[x][y] * cos(2 * phi);
        update(x, y, tmp);
        update(y, x, tmp);
        A[x][x] = a, A[y][y] = b;
    }
}
//a 为输入向量组
//n 为向量的维数
//center 指针用来保存输入向量组的中心点
//返回特征值和特征向量的 pair，按照特征值从大到小排序
//特征值是各个点在对应特征向量方向的坐标平方和，除以 (a.size() - 1) 为方差。
auto solve(vector<vec> a, int n, vec* center = nullptr) {
    this->n = n;
    vec s = {};
    for (int i = 0; i < a.size(); ++i)
```

```
            for (int j = 0; j < n; ++j)
                s[j] += a[i][j];
        for (int j = 0; j < n; ++j)
            s[j] /= a.size();
        for (int i = 0; i < a.size(); ++i)
            for (int j = 0; j < n; ++j)
                a[i][j] -= s[j];
        if (center) *center = s;
        for (int i = 0; i < n; ++i) {
            for (int j = 0; j < n; ++j) {
                A[i][j] = 0;
                for (int k = 0; k < a.size(); ++k)
                    A[i][j] += a[k][i] * a[k][j];
            }
        }
        Jacobi();
        vector<pair_t> result;
        for (int i = 0; i < n; ++i)
            result.emplace_back(A[i][i], vec());
        for (int i = 0; i < n; ++i)
            for (int j = 0; j < n; ++j)
                result[i].second[j] = V[j][i];
        sort(result.begin(), result.end(), greater<pair_t>());
        return result;
    }
}pca;
int main() { //1329070654.526
    freopen("in.txt", "r", stdin);
    vector<vec> a;
    int n, m;
    scanf("%d %d", &n, &m);
    for (int i = 0; i < n; ++i) {
        vec v = {};
        for (int j = 0; j < m; ++j)
            scanf("%lf", &v[j]);
        a.push_back(v);
    }
    auto now = clock();
    auto result = pca.solve(a, m);
    for (int i = 0; i < m; ++i)
        printf("%.3f\n", result[i].first);
    printf("time: %f\n", double(clock() - now) / CLOCKS_PER_SEC);
    return 0;
}
```

16.2 Adam 算法

```
const int dim = 2;
using vec = array<double, dim>;
/*
* grad: 计算梯度的函数
* pos: 搜索的起始点
* lr: 学习率（移动的步长）
* decay: 学习率的衰减指数
* exit: 当学习率小于 exit 时算法结束
* 注意该算法求出的是 函数的极 ** 小 ** 值点
*/
vec Adam(function<vec(vec)> grad, vec pos, double lr, double decay, double exit,
    const double beta1 = 0.9, const double beta2 = 0.9) {
    double pw1 = 1, pw2 = 1;
    vec m = {}, v = {};
    for (int t = 1; lr >= exit; ++t) {
        vec g = grad(pos);
        pw1 *= beta1;
        pw2 *= beta2;
        for (int i = 0; i < dim; ++i) {
            m[i] = m[i] * beta1 + g[i] * (1 - beta1);
            v[i] = v[i] * beta2 + g[i] * g[i] * (1 - beta2);
            double update = m[i] / (1 - pw1);
            double factor = lr / (sqrt(v[i] / (1 - pw2)) + 1e-8);
            pos[i] -= update * factor;
        }
        lr *= decay;
    }
    return pos;
}
int main() {
    auto grad = [](vec A) {
        double x = A[0], y = A[1];
        return vec{ 2 * y + -4 / (x * x), 2 * x + -4 / (y * y) };
    };
    auto ans = Adam(grad, vec{ 1, 1 }, 0.01, 0.999, 1e-8);
    printf("%.10f %.10f\n", ans[0], ans[1]); //1.2599210498948732
    return 0;
}
```

16.3 Dyna-Q

```
const int nrow = 10, ncol = 10, max_state = nrow * ncol, max_action = 4;
const int dr[4] = { -1, 1, 0, 0 };
const int dc[4] = { 0, 0, -1, 1 };
struct CliffWalkingEnv {
```

```
char s[11][11];
int x, y;
vector<pair<int, int>> path;
CliffWalkingEnv() {
    reset();
}
int reset() {
    x = 0, y = 0;
    path.clear();
    path.emplace_back(x, y);
    memset(s, 0, sizeof(s));
    for (int i = 0; i < nrow; ++i) {
        for (int j = 0; j < ncol; ++j) {
            s[i][j] = '.';
        }
    }
    for (int i = 0; i < 5; ++i) {
        for (int j = 1; j < ncol - 1; ++j) {
            s[i][j] = '#';
        }
    }
    return x * ncol + y;
}
void print() {
    char t[11][11];
    memcpy(t, s, sizeof(s));
    for (auto [x, y] : path) {
        if (t[x][y] >= '1' && t[x][y] < '9')
            t[x][y] += 1;
        else
            t[x][y] = '1';
    }
    for (int i = 0; i < nrow; ++i)
        printf("%s\n", t[i]);
    printf("\n");
}
tuple<int, int, int> step(int number) {
    int nx = x + dr[number], ny = y + dc[number];
    if (nx >= 0 && nx < nrow && ny >= 0 && ny < ncol)
        x = nx, y = ny;
    int reward = -1, done = false;
    if (s[x][y] == '#') {
        reward = -100;
        done = true;
    }
    if (x == 0 && y == ncol - 1)
```

```
            done = true;
        path.emplace_back(x, y);
        return make_tuple(x * ncol + y, reward, done);
    }
} env;
const double alpha = 1e-1; /* 学习率 */
const double gamma = 0.9; /* 折扣因子 */
const double epsilon = 1e-2; /*epsilon-greedy*/
const int N = 10; //Q-Planning 的次数
const int max_episode = 200; //训练多少轮
uniform_real_distribution<double> p;
uniform_int_distribution<int> d;
default_random_engine e;
double Q[max_state][max_action];
map<pair<int, int>, int> id;
vector<tuple<int, int, int, int>> model;
int epsilon_greedy(int state) { /* 基于 epsilon-greedy 选择动作 */
    if (p(e) < epsilon)
        return d(e) % max_action;
    return max_element(Q[state], Q[state] + max_action) - Q[state];
}
void learn(int s0, int a0, int r, int s1) {
    auto td_error = r + gamma * *max_element(Q[s1], Q[s1] + max_action) - Q[s0][a0];
    Q[s0][a0] += alpha * td_error;
}
void update(int s0, int a0, int r, int s1) {
    learn(s0, a0, r, s1);
    pair<int, int> pr(s0, a0);
    if (!id.count(pr)) {
        id[pr] = model.size();
        model.emplace_back(s0, a0, r, s1);
    }
    else {
        model[id[pr]] = make_tuple(s0, a0, r, s1);
    }
    for (int i = 0; i < N; ++i) {
        int idx = d(e) % model.size();
        auto [s0, a0, r, s1] = model[idx];
        learn(s0, a0, r, s1);
    }
}
void DynaQ() {
    for (int i = 0; i < max_episode; ++i) {
        int state = env.reset();
        for (;;) {
            int action = epsilon_greedy(state);
```

```
            auto [next_state, reward, done] = env.step(action);
            update(state, action, reward, next_state);
            state = next_state;
            if (done)
                break;
        }
        env.print();
    }
}
int main() {
    DynaQ();
    return 0;
}
```