图论

树的重心

```
int n, siz[N], wei[N];
std::vector< int > centroid, adj[N];
void GetCentroid(int x, int p) {
    siz[x] = 1;
    wei[x] = 0;
    for (const auto &y : adj[x]) {
        if (y ^ p) {
            GetCentroid(y, x);
            siz[x] += siz[y];
            wei[x] = std::max(wei[x], siz[y]);
        }
    }
    wei[x] = std::max(wei[x], n - siz[x]);
    if (wei[x] <= n / 2) {
        centroid.push_back(x);
    }
}
```

最小树形图

有向图上的最小生成树(Directed Minimum Spanning Tree)称为最小树形图。

常用的算法是朱刘算法(也称 Edmonds 算法),可以在时间$\Theta(nm)$内解决最小树形图问题。

Kruskal 重构树

原图中两个点之间的所有简单路径上最大边权的最小值 = 最小生成树上两个点之间的简单路径上的最大值 = Kruskal 重构树上两点之间的 LCA 的权值。

```
struct Edge {
    int u, v, w;
    bool operator < (const Edge &o) const {
        return w < o.w;
    }
};
struct DSU {
    std::vector< int > f;
    DSU(int n) : f(n + 1) {
        std::iota(begin(f), end(f), 0);
    }
```

```cpp
    int find(int x) {
        return f[x] == x ? x : f[x] = find(f[x]);
    }
};
std::vector< int > adj[N];
struct Kruskal {
    std::vector< Edge > edges;
    int root;
    int val[N];
    void AddEdge(int u, int v, int w) {
        edges.push_back({u, v, w});
    }
    void solve(int n) {
        std::sort(begin(edges), end(edges));
        root = n;
        DSU dsu(n * 2 - 1);
        for (auto &g : edges) {
            int fx = dsu.find(g.u), fy = dsu.find(g.v);
            if (fx != fy) {
                dsu.f[fx] = dsu.f[fy] = ++ root;
                val[root] = g.w;
                adj[root] = {fx, fy};
            }
        }
    }
};
```

LCA

```cpp
template< typename T >
struct INFO {
    int v;
    T w;
};
// index from 1 to n
template< typename T >
struct Tree {
    int n, cur;
    std::vector< int > dfn, pos, dep, fa;
    std::vector< T > dis;
    std::vector< std::vector< INFO< T > > > adj;

    Tree(int _n) {
        n = _n;
        cur = 0;
        dfn = std::vector< int > (_n * 2);
        pos = dep = fa = std::vector< int > (_n + 1);
```

```
        dis = std::vector< T > (_n + 1);
        adj = std::vector< std::vector< INFO< T > > > (_n + 1);
    }

    void AddEdge(int u, int v, T w = 1) {
        adj[u].push_back({v, w});
    }
    void dfs(int x, int p) {
        fa[x] = p;
        dep[x] = dep[p] + 1;
        dfn[++ cur] = x;
        pos[x] = cur;
        for (const auto &[y, z] : adj[x]) {
            if (y ^ p) {
                dis[y] = dis[x] + z;
                dfs(y, x);
                dfn[++ cur] = x;
            }
        }
    }

    std::vector< int > LOG;
    std::vector< std::vector< int > > RMQ;

    void Build() {
        LOG = std::vector< int > (cur + 1);
        for (int i = 2; i <= cur; ++ i) {
            LOG[i] = LOG[i / 2] + 1;
        }
        RMQ = std::vector< std::vector< int > > (cur + 1, std::vector< int > (LOG[cur]
+ 1));
        for (int i = 1; i <= cur; ++ i) {
            RMQ[i][0] = pos[dfn[i]];
        }
        for (int j = 1; j <= LOG[cur]; ++ j) {
            for (int i = 1; i + (1 << j) - 1 <= cur; ++ i) {
                RMQ[i][j] = std::min(RMQ[i][j - 1], RMQ[i + (1 << (j - 1))][j - 1]);
            }
        }
    }
    int LCA(int x, int y) { // O(1) query lca
        x = pos[x];
        y = pos[y];
        if (x > y) {
            std::swap(x, y);
        }
        int k = LOG[y - x + 1];
        return dfn[std::min(RMQ[x][k], RMQ[y - (1 << k) + 1][k])];
    }
```

```
    int Distance(int x, int y) {
        return dis[x] + dis[y] - 2 * dis[LCA(x, y)];
    }
};
```

判断负环

Spfa

时间复杂度$\Theta(n \times m)$

```
bool vis[N];
int d[N], cnt[N];
vector< pair< int, int > > adj[N];
bool spfa(int n) {
    for (int i = 1; i <= n; ++ i) {
        adj[0].push_back({i, 0});
    }
    memset(d, -0x3f, sizeof d);
    d[0] = 0;
    vis[0] = true;
    queue< int > q;
    q.push(0);
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        vis[u] = false;
        for (const auto &[v, w] : adj[u]) {
            if (d[v] < d[u] + w) {
                d[v] = d[u] + w;
                if (!vis[v]) {
                    q.push(v);
                    if (++ cnt[v] >= n) {
                        return false;
                    }
                }
            }
        }
    }
    return true;
}
```

Bellman-Ford

时间复杂度$\Theta(n^2 \times m)$

```
int d[N];
vector< pair< int, int > > adj[N];
bool Bellman_Ford(int n) {
    for (int t = 1; t < n; ++ t) {
        bool upd = false;
        for (int i = 1; i <= n; ++ i) {
            for (const auto &[v, w] : adj[i]) {
                if (d[i] > d[v] + w) {
                    d[i] = d[v] + w;
                    upd = true;
                }
            }
        }
        if (!upd) {
            break;
        }
    }
    for (int i = 1; i <= n; ++ i) {
        for (const auto &[v, w] : adj[i]) {
            if (d[i] > d[v] + w) {
                return false;
            }
        }
    }
    return true;
}
```

最小环

给出一个图, 问其中的有`n`个节点构成的边权和最小的环$(n \geq 3)$是多大。

图的最小环也称围长。

Dijkstra

枚举所有边,每一次求删除一条边之后对这条边的起点跑一次最短路,$\mathrm{ans} = \min(\mathrm{ans}, \mathrm{dis}(\mathrm{u}, \mathrm{v}) + \mathrm{w})$

时间复杂度:$\Theta(\mathrm{m}(\mathrm{m} + \mathrm{n}) \log\ \mathrm{n})$

Floyd

$ans = \min(ans, w(i,k) + w(k,j) + dis(i,j))$

$dis(i,j) = \min(dis(i,j), dis(i,k) + dis(k,j))$

时间复杂度:$\Theta(\mathrm{n}^3)$

连通性相关

无向图

割边以及边双连通分量e-DCC的缩点

```
struct Edge {
    int from, to;

    Edge(int u, int v) : from(u), to(v) {}
};
struct e_DCC {
    int n, m, cnt, cur;
    std::vector< Edge > edges;
    std::vector< std::vector< int > > G;
    std::vector< int > dfn, low, col;
    std::vector< bool > bridge;

    e_DCC(int n) : n(n), cnt(0), cur(0), G(n), dfn(n), low(n), col(n) {}

    void AddEdge(int u, int v) {
        edges.push_back(Edge(u, v));
        G[u].push_back((int)edges.size() - 1);
        m = (int)edges.size();
    }

    void tarjan(int x, int p) {
        dfn[x] = low[x] = ++ cur;
        for (const auto &i : G[x]) {
            int y = edges[i].to;
            if (y == p) {
                continue;
            }
            if (!dfn[y]) {
                tarjan(y, x);
                low[x] = std::min(low[x], low[y]);
                if (low[y] > dfn[x]) {
                    bridge[i] = bridge[i ^ 1] = true;
                }
            } else {
```

```
                low[x] = std::min(low[x], dfn[y]);
            }
        }
    }

    void dfs(int x) {
        col[x] = cnt;
        for (const auto &i : G[x]) {
            int y = edges[i].to;
            if (col[y] || bridge[i]) {
                continue;
            }
            dfs(y);
        }
    }

    void solve() {
        bridge.assign(m, 0);
        for (int i = 0; i < n; ++ i) {
            if (!dfn[i]) {
                tarjan(i, i);
            }
        }
        for (int i = 0; i < n; ++ i) {
            if (!col[i]) {
                ++ cnt;
                dfs(i);
            }
        }
        for (int i = 1; i < edges.size(); ++ i) {
            int x = edges[i ^ 1].to, y = edges[i].to;
            if (col[x] != col[y]) {
                // Add_New_Edge
            }
        }
    }
};
```

割点和点双连通分量v-DCC的缩点

```
struct v_DCC {
    int cur, cnt, root, top;
    std::vector< int > dfn, low, col, stk;
    std::vector< bool > cut;
    std::vector< std::vector< int > > adj;

    v_DCC(int n) : cur(0), cnt(0), top(0), dfn(n), low(n), col(n), stk(n), cut(n),
adj(n) {}
```

```cpp
    void tarjan(int x) {
        dfn[x] = low[x] = ++ cur;
        stk[top ++ ] = x;
        if (x == root && adj[x].empty()) { // 孤立点
            ++ cnt;
            col[x] = cnt;
            return;
        }
        int flag = 0;
        for (const auto &y : adj[x]) {
            if (!dfn[y]) {
                tarjan(y);
                low[x] = std::min(low[x], low[y]);
                if(low[y] >= dfn[x]) {
                    ++ flag;
                    if (x != root || flag > 1) {
                        cut[x] = true;
                    }
                    ++ cnt;
                    int z;
                    do {
                        z = stk[-- top];
                        col[z] = cnt;
                    } while (z != y);
                    col[x] = cnt;
                }
            } else {
                low[x] = std::min(low[x], dfn[y]);
            }
        }
    }
};
```

有向图找强连通分量

```cpp
struct SCC {
    int cur, cnt, top;
    std::vector< int > dfn, low, col, stk;
    std::vector< std::vector< int > > adj;

    SCC(int n) : cur(0), cnt(0), top(0), dfn(n), low(n), col(n), stk(n), adj(n) {}

    void tarjan(int x) {
        dfn[x] = low[x] = ++ cur;
        stk[top ++] = x;
        for (const auto &y : adj[x]) {
            if (!dfn[y]) {
```

```
                tarjan(y);
                low[x] = std::min(low[x], low[y]);
            } else if (!col[y]) {
                low[x] = std::min(low[x], dfn[y]);
            }
        }
        if (low[x] == dfn[x]) {
            ++ cnt;
            int z;
            do {
                z = stk[-- top];
                col[z] = cnt;
            } while (z != x);
        }
    }
};
```

欧拉图

```
struct Euler {
    int n, m, top, t, tot;
    std::vector< int > head, ver, next, ans;
    std::stack< int > stk;
    std::vector< bool > vis;

    Euler(int n, int m) : n(n), m(m), head(n), ver(m), next(n), ans(n), vis(m) {}
    void add_edge(int x, int y) {
        ver[++ tot] = y, next[y] = head[x], head[x] = tot;
    }
    void euler() {
        stk.push(1);
        while (top) {
            int x = stk.top(), i = head[x];
            while (i && vis[i]) i = next[i];
            if (i) {
                stk.push(ver[i]);
                vis[i] = vis[i ^ 1] = true;
                head[x] = next[i];
            } else {
                stk.pop();
                ans[++ t] = x;
            }
        }
    }
    void show() {
        for (int i = t; i; -- i) {
            std::cout << ans[i] << '\n';
        }
```

```
    }
};
```

网络流

最大流

Dinic

[算法原理]

对于二分图$\Theta(m\sqrt{n})$

最坏情况$\Theta(n^2m)$

```
constexpr int INF = 0x3f3f3f3f;
template< typename T >
struct Edge {
    int from, to;
    T cap, flow;

    Edge(int u, int v, T c, T f) : from(u), to(v), cap(c), flow(f) {}
};
template< typename T >
struct Dinic {
    int n, m;
    int Source, Sink;
    std::vector< Edge< T > > edges;
    std::vector< std::vector< int > > G;
    std::vector< int > d, cur;
    std::vector< bool > vis;

    // number of vertices
    Dinic(int n) : n(n), d(n), cur(n), vis(n), G(n) {}

    void AddEdge(int from, int to, T cap) {
        edges.push_back(Edge< T >(from, to, cap, 0));
        edges.push_back(Edge< T >(to, from, 0, 0));
        m = edges.size();
        G[from].push_back(m - 2);
        G[to].push_back(m - 1);
    }

    bool BFS() {
        vis.assign(n, 0);
        std::queue< int > Q;
        Q.push(Source);
        d[Sink] = 0;
        vis[Source] = true;
```

```
        while (!Q.empty()) {
            int x = Q.front();
            Q.pop();
            for (int i = 0; i < G[x].size(); ++ i) {
                Edge< T > &e = edges[G[x][i]];
                if (!vis[e.to] && e.cap > e.flow) {
                    vis[e.to] = true;
                    d[e.to] = d[x] + 1;
                    Q.push(e.to);
                }
            }
        }
        return vis[Sink];
    }

    T DFS(int x, T a) {
        if (x == Sink || a == 0) {
            return a;
        }
        T flow = 0, f;
        for (int& i = cur[x]; i < G[x].size(); ++ i) {
            Edge< T > &e = edges[G[x][i]];
            if (d[x] + 1 == d[e.to] && (f = DFS(e.to, std::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;
    }

    T Maxflow(int s, int t) {
        this->Source = s;
        this->Sink = t;
        T flow = 0;
        while (BFS()) {
            cur.assign(n, 0);
            flow += DFS(Source, INF);
        }
        return flow;
    }
};
```

最小割

最大流最小割定理

$$f(s,t)_{max} = c(s,t)_{min}$$

输出方案

首先根据最小割最大流定理,我们跑一遍Dinic就可以求出最小割,这是**参量网络**中`s`和`t`已经不再联通了.我们可以从`s`开始跑一遍`dfs`,沿着所有还未满流的边搜索,所有能到达的节点就是和`s`再同一集合的节点.之后我们遍历每一条边,将边起点终点不在同一集合内的输出即可

输出方案即输出除源点外的S集合，这里有一个~~待研究的~~性质：**dinic最后一次bfs后，vis为1的节点集合即为最小割的S集合.**

最小费用最大流

Dinic

```
constexpr int INF = 0x3f3f3f3f;
template< typename T >
struct Edge {
    int from, to;
    T flow, cost;

    Edge(int from, int to, T flow, T cost) : from(from), to(to), flow(flow), cost(cost)
{}
};
template< typename T >
struct Dinic {
    int n, m;
    int Source, Sink;
    std::vector< Edge< T > > edges;
    std::vector< std::vector< int > > G;
    std::vector< T > incf, d;
    std::vector< int > pre;
    std::vector< bool > vis;

    // number of vertices
    Dinic(int n) : n(n), G(n), incf(n), d(n), pre(n), vis(n) {}

    void AddEdge(int from, int to, T flow, T cost) {
        edges.push_back(Edge< T >(from, to, flow, cost));
        edges.push_back(Edge< T >(to, from, 0, -cost));
        m = edges.size();
        G[from].push_back(m - 2);
        G[to].push_back(m - 1);
    }

    bool SPFA() {
```

```
        d.assign(n, INF);
        incf.assign(n, 0);
        vis.assign(n, false);
        std::queue< int > Q;
        Q.push(Source);
        vis[Source] = true;
        incf[Source] = INF;
        d[Source] = 0;
        while (!Q.empty()) {
            int x = Q.front();
            Q.pop();
            vis[x] = false;
            for (int i = 0; i < G[x].size(); ++ i) {
                Edge< T > &e = edges[G[x][i]];
                if (d[x] + e.cost < d[e.to] && e.flow) {
                    d[e.to] = d[x] + e.cost;
                    pre[e.to] = G[x][i];
                    incf[e.to] = std::min(incf[x], e.flow);
                    if (!vis[e.to]) {
                        vis[e.to] = true;
                        Q.push(e.to);
                     }
                }
            }
        }
        return d[Sink] != INF;
    }

    std::pair< T, T > MCMF(int s, int t) {
        this->Source = s;
        this->Sink = t;
        T maxflow = 0, mincost = 0;
        while (SPFA()) {
            maxflow += incf[Sink];
            mincost += incf[Sink] * d[Sink];
            for (int i = Sink; i != Source; i = edges[pre[i] ^ 1].to) {
                edges[pre[i]].flow -= incf[Sink];
                edges[pre[i] ^ 1].flow += incf[Sink];
            }
        }
        return {maxflow, mincost};
    }
};
```

重链剖分

```
struct Tree {
    int dfn[N], siz[N], dep[N], son[N], f[N], top[N];
```

```
    vector< int > adj[N];
    void dfs1(int x, int p) {
        dep[x] = dep[p] + 1;
        f[x] = p;
        siz[x] = 1;
        for (auto &y : adj[x]) {
            if (y ^ p) {
                dfs1(y, x);
                siz[x] += siz[y];
                if (siz[y] > siz[son[x]]) {
                    son[x] = y;
                }
            }
        }
    }
    void dfs2(int x, int tp) {
        top[x] = tp;
        dfn[x] = ++ dfn[0];
        if (!son[x]) {
            return;
        }
        dfs2(son[x], tp);
        for (auto &y : adj[x]) {
            if (y != son[x] && y != f[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 = f[top[x]];
        }
        return dfn[x] < dfn[y] ? x : y;
    }
};
```

长链剖分

还没学,鸽...

虚树

```
template< typename T >
struct rmq {
    vector< T > v;
```

```
    int n;
    static const int b = 30;
    vector<int> mask, t;

    int op(int x, int y) {
        return v[x] < v[y] ? x : y;
    }

    int msb(int x) { return __builtin_clz(1) - __builtin_clz(x); }

    rmq() {}

    rmq(const vector<T> &v_) : v(v_), n(v.size()), mask(n), t(n) {
        for (int i = 0, at = 0; i < n; mask[i++] = at |= 1) {
            at = (at << 1) & ((1 << b) - 1);
            while (at and op(i, i - msb(at & -at)) == i) at ^= at & -at;
        }
        for (int i = 0; i < n / b; i++) t[i] = b * i + b - 1 - msb(mask[b * i + b -
1]);
        for (int j = 1; (1 << j) <= n / b; j++)
            for (int i = 0; i + (1 << j) <= n / b; i++)
                t[n / b * j + i] = op(t[n / b * (j - 1) + i], t[n / b * (j - 1) + i +
(1 << (j - 1))]);
    }

    int small(int r, int sz = b) { return r - msb(mask[r] & ((1 << sz) - 1)); }

    T query(int l, int r) {
        if (r - l + 1 <= b) return small(r, r - l + 1);
        int ans = op(small(l + b - 1), small(r));
        int x = l / b + 1, y = r / b - 1;
        if (x <= y) {
            int j = msb(y - x + 1);
            ans = op(ans, op(t[n / b * j + x], t[n / b * j + y - (1 << j) + 1]));
        }
        return ans;
    }
};

namespace lca {
    vector<int> g[N];
    int v[2 * N], pos[N], dep[2 * N];
    int t;
    rmq<int> RMQ;

    void dfs(int i, int d = 0, int p = -1) {
        v[t] = i, pos[i] = t, dep[t++] = d;
        for (int j : g[i])
            if (j != p) {
```

```
                dfs(j, d + 1, i);
                v[t] = i, dep[t++] = d;
            }
    }

    void build(int n, int root) {
        t = 0;
        dfs(root);
        RMQ = rmq<int>(vector<int>(dep, dep + 2 * n - 1));
    }

    int lca(int a, int b) {
        a = pos[a], b = pos[b];
        return v[RMQ.query(min(a, b), max(a, b))];
    }

    int dist(int a, int b) {
        return dep[pos[a]] + dep[pos[b]] - 2 * dep[pos[lca(a, b)]];
    }
}

vector<int> virt[N];

int build_virt(vector<int> v) {
    auto cmp = [&](int i, int j) { return lca::pos[i] < lca::pos[j]; };
    sort(v.begin(), v.end(), cmp);
    for (int i = v.size() - 1; i; i--) v.push_back(lca::lca(v[i], v[i - 1]));
    sort(v.begin(), v.end(), cmp);
    v.erase(unique(v.begin(), v.end()), v.end());
    for (int i : v) virt[i].clear();
    for (int i = 1; i < v.size(); i++) {
        virt[lca::lca(v[i - 1], v[i])].push_back(v[i]);
    }
    return v[0];
}
```

2-SAT

```
struct TWO_SAT {
    int head[N], ver[N], Next[N];
    int dfn[N], low[N], col[N];
    int n, m, tot, num, cnt;
    stack< int > st;
    bool vis[N];
    void tarjan(int x) {
        dfn[x] = low[x] = ++ num;
        st.push(x);
```

```
        for (int i = head[x]; i; i = Next[i]) {
            int y = ver[i];
            if (!dfn[y]) {
                tarjan(y);
                low[x] = min(low[x], low[y]);
            } else if (!col[y]) {
                low[x] = min(low[x], dfn[y]);
            }
        }
        if (low[x] == dfn[x]) {
            col[x] = ++ cnt;
            while (st.top() != x) {
                col[st.top()] = cnt;
                st.pop();
            }
            st.pop();
        }
    }
    bool check() {
        for (int i = 1; i <= n; ++ i) {
            if (col[i] == col[i + n]) return false;
        }
        return true;
    }
    void find() {
        vector< int > ret;
        for (int i = 1; i <= n; ++ i) {
            int x = col[i], y = col[i + n];
            if (vis[x]) {
                ret.push_back(1); continue;
            }
            if (vis[y]) {
                ret.push_back(0); continue;
            }
            if (x < y) {
                ret.push_back(1); vis[x] = true;
            } else {
                ret.push_back(0); vis[y] = true;
            }
        }
    }
};
```

图的匹配

二分图最大匹配

找增广路$\Theta(nm)$

```
struct augment_path {
    std::vector< std::vector< int > > g;
    std::vector< int > pa;  // 匹配
    std::vector< int > pb;
    std::vector< int > vis;  // 访问
    int n, m;          // 两个点集中的顶点数量
    int dfn;           // 时间戳记
    int res;           // 匹配数

    augment_path(int _n, int _m) : n(_n), m(_m) {
        assert(0 <= n && 0 <= m);
        pa = std::vector< int >(n, -1);
        pb = std::vector< int >(m, -1);
        vis = std::vector< int >(n);
        g.resize(n);
        res = 0;
        dfn = 0;
    }

    void AddEdge(int from, int to) {
        assert(0 <= from && from < n && 0 <= to && to < m);
        g[from].push_back(to);
    }

    bool dfs(int v) {
        vis[v] = dfn;
        for (int u : g[v]) {
            if (pb[u] == -1) {
                pb[u] = v;
                pa[v] = u;
                return true;
            }
        }
        for (int u : g[v]) {
            if (vis[pb[u]] != dfn && dfs(pb[u])) {
                pa[v] = u;
                pb[u] = v;
                return true;
            }
        }
        return false;
    }

    std::pair< int, std::vector< int > > find_max_unweighted_matching() {
        while (true) {
```

```
            ++ dfn;
            int cnt = 0;
            for (int i = 0; i < n; ++ i) {
                if (pa[i] == -1 && dfs(i)) {
                    ++ cnt;
                }
            }
            if (cnt == 0) {
                break;
            }
            res += cnt;
        }
        return {res, pa};
    }
};
```

二分图最大权匹配

Hungarian Algorithm（Kuhn-Munkres Algorithm）

$\Theta(n^3)$

```
template< typename T >
struct hungarian {  // km
    int n;
    std::vector< int > matchx;  // 左集合对应的匹配点
    std::vector< int > matchy;  // 右集合对应的匹配点
    std::vector< int > pre;     // 连接右集合的左点
    std::vector< bool > visx;   // 拜访数组 左
    std::vector< bool > visy;   // 拜访数组 右
    std::vector< T > lx;
    std::vector< T > ly;
    std::vector< std::vector< T > > g;
    std::vector< T > slack;
    T inf;
    T res;
    std::queue< int > q;
    int org_n;
    int org_m;

    hungarian(int _n, int _m) {
        org_n = _n;
        org_m = _m;
        n = std::max(_n, _m);
        inf = std::numeric_limits< T >::max();
        res = 0;
        g = std::vector< std::vector< T > >(n, std::vector< T >(n));
        matchx = std::vector< int >(n, -1);
        matchy = std::vector< int >(n, -1);
```

```
        pre = std::vector< int >(n);
        visx = std::vector< bool >(n);
        visy = std::vector< bool >(n);
        lx = std::vector< T >(n, -inf);
        ly = std::vector< T >(n);
        slack = std::vector< T >(n);
    }

    void AddEdge(int u, int v, T w) {
        g[u][v] = std::max(w, 0);  // 负值还不如不匹配 因此设为0不影响
    }

    bool check(int v) {
        visy[v] = true;
        if (matchy[v] != -1) {
            q.push(matchy[v]);
            visx[matchy[v]] = true;  // in S
            return false;
        }
        // 找到新的未匹配点 更新匹配点 pre 数组记录着"非匹配边"上与之相连的点
        while (v != -1) {
            matchy[v] = pre[v];
            std::swap(v, matchx[pre[v]]);
        }
        return true;
    }

    void bfs(int i) {
        while (!q.empty()) {
            q.pop();
        }
        q.push(i);
        visx[i] = true;
        while (true) {
            while (!q.empty()) {
                int u = q.front();
                q.pop();
                for (int v = 0; v < n; ++ v) {
                    if (!visy[v]) {
                        T delta = lx[u] + ly[v] - g[u][v];
                        if (slack[v] >= delta) {
                            pre[v] = u;
                            if (delta) {
                                slack[v] = delta;
                            } else if (check(v)) {  // delta=0 代表有机会加入相等子图 找增广
路

                                // 找到就return 重建交错树
                                return;
                            }
```

```
                    }
                }
            }
        }
        // 没有增广路 修改顶标
        T a = inf;
        for (int j = 0; j < n; ++ j) {
            if (!visy[j]) {
                a = std::min(a, slack[j]);
            }
        }
        for (int j = 0; j < n; ++ j) {
            if (visx[j]) {  // S
                lx[j] -= a;
            }
            if (visy[j]) {  // T
                ly[j] += a;
            } else {  // T'
                slack[j] -= a;
            }
        }
        for (int j = 0; j < n; ++ j) {
            if (!visy[j] && slack[j] == 0 && check(j)) {
                return;
            }
        }
    }
}

std::pair< T, std::vector< int > > find_max_weighted_matching() {
    // 初始顶标
    for (int i = 0; i < n; ++ i) {
        for (int j = 0; j < n; ++ j) {
            lx[i] = std::max(lx[i], g[i][j]);
        }
    }

    for (int i = 0; i < n; ++ i) {
        fill(slack.begin(), slack.end(), inf);
        fill(visx.begin(), visx.end(), false);
        fill(visy.begin(), visy.end(), false);
        bfs(i);
    }

    // custom
    for (int i = 0; i < n; ++ i) {
        if (g[i][matchx[i]] > 0) {
            res += g[i][matchx[i]];
        } else {
```

```
                matchx[i] = -1;
            }
        }
        /*
        std::cout << res << "\n";
        for (int i = 0; i < org_n; i++) {
            std::cout << matchx[i] + 1 << " \n"[i + 1 == org_n];
        }
         */
        return {res, matchx};
    }
};
```

一般图最大匹配

开花算法（Blossom Algorithm，也被称做带花树）

```
// graph
template< typename T >
class graph {
public:
    struct edge {
        int from;
        int to;
        T cost;
    };

    std::vector< edge > edges;
    std::vector< std::vector< int > > g;
    int n;

    graph(int _n) : n(_n) {
        g.resize(n);
    }

    virtual int add(int from, int to, T cost) = 0;
};

// undirectedgraph
template< typename T >
class undirectedgraph : public graph< T > {
public:
    using graph< T >::edges;
    using graph< T >::g;
    using graph< T >::n;

    undirectedgraph(int _n) : graph< T >(_n) {}
```

```
    int AddEdge(int from, int to, T cost = 1) {
        assert(0 <= from && from < n && 0 <= to && to < n);
        int id = (int) edges.size();
        g[from].push_back(id);
        g[to].push_back(id);
        edges.push_back({from, to, cost});
        return id;
    }
};

// blossom / find_max_unweighted_matching
template< typename T >
std::vector< int > find_max_unweighted_matching(const undirectedgraph< T > &g) {
    std::mt19937 rng(std::chrono::steady_clock::now().time_since_epoch().count());
    std::vector< int > match(g.n, -1);   // 匹配
    std::vector< int > aux(g.n, -1);     // 时间戳记
    std::vector< int > label(g.n);       // "o" or "i"
    std::vector< int > orig(g.n);        // 花根
    std::vector< int > parent(g.n, -1);  // 父节点
    std::queue< int > q;
    int aux_time = -1;

    auto lca = [&](int v, int u) {
        aux_time++;
        while (true) {
            if (v != -1) {
                if (aux[v] == aux_time) {  // 找到拜访过的点 也就是LCA
                    return v;
                }
                aux[v] = aux_time;
                if (match[v] == -1) {
                    v = -1;
                } else {
                    v = orig[parent[match[v]]];  // 以匹配点的父节点继续寻找
                }
            }
            std::swap(v, u);
        }
    };  // lca

    auto blossom = [&](int v, int u, int a) {
        while (orig[v] != a) {
            parent[v] = u;
            u = match[v];
            if (label[u] == 1) {  // 初始点设为"o" 找增广路
                label[u] = 0;
                q.push(u);
            }
```

```
            orig[v] = orig[u] = a;  // 缩花
            v = parent[u];
        }
    };  // blossom

    auto augment = [&](int v) {
        while (v != -1) {
            int pv = parent[v];
            int next_v = match[pv];
            match[v] = pv;
            match[pv] = v;
            v = next_v;
        }
    };  // augment

    auto bfs = [&](int root) {
        fill(label.begin(), label.end(), -1);
        iota(orig.begin(), orig.end(), 0);
        while (!q.empty()) {
            q.pop();
        }
        q.push(root);
        // 初始点设为 "o", 这里以"0"代替"o", "1"代替"i"
        label[root] = 0;
        while (!q.empty()) {
            int v = q.front();
            q.pop();
            for (int id : g.g[v]) {
                auto &e = g.edges[id];
                int u = e.from ^ e.to ^ v;
                if (label[u] == -1) {  // 找到未拜访点
                    label[u] = 1;         // 标记 "i"
                    parent[u] = v;
                    if (match[u] == -1) {  // 找到未匹配点
                        augment(u);           // 寻找增广路径
                        return true;
                    }
                    // 找到已匹配点 将与她匹配的点丢入queue 延伸交错树
                    label[match[u]] = 0;
                    q.push(match[u]);
                    continue;
                } else if (label[u] == 0 && orig[v] != orig[u]) {
                    // 找到已拜访点 且标记同为"o" 代表找到"花"
                    int a = lca(orig[v], orig[u]);
                    // 找LCA 然后缩花
                    blossom(u, v, a);
                    blossom(v, u, a);
                }
            }
```

```
        }
        return false;
    };  // bfs

    auto greedy = [&]() {
        std::vector< int > order(g.n);
        // 随机打乱 order
        iota(order.begin(), order.end(), 0);
        shuffle(order.begin(), order.end(), rng);

        // 将可以匹配的点匹配
        for (int i : order) {
            if (match[i] == -1) {
                for (auto id : g.g[i]) {
                    auto &e = g.edges[id];
                    int to = e.from ^ e.to ^ i;
                    if (match[to] == -1) {
                        match[i] = to;
                        match[to] = i;
                        break;
                    }
                }
            }
        }
    };  // greedy

    // 一开始先随机匹配
    greedy();
    // 对未匹配点找增广路
    for (int i = 0; i < g.n; i++) {
        if (match[i] == -1) {
            bfs(i);
        }
    }
    return match;
}
```

数据结构

并查集

维护联通性

```
struct UnionFind {
    std::vector< int > f, siz;
    UnionFind(int n) : f(n), siz(n, 1) {
        iota(begin(f), end(f), 0);
    }
    int find(int x) {
        return f[x] == x ? x : f[x] = find(f[x]);
    }
    bool Merge(int x, int y) { // 按秩合并
        x = find(x), y = find(y);
        if (x != y) {
            if (siz[x] < siz[y]) {
                std::swap(x, y);
            }
            siz[x] += siz[y];
            f[y] = x;
        }
        return false;
    }
    int operator [] (int x) {
        return find(x);
    }
};
```

带权并查集

```
int find(int x) {
    if (f[x] == x) {
        return x;
    }
    int rt = find(f[x]);
    val[x] += val[f[x]];
    return f[x] = rt;
}
```

可持久化并查集

给定 n 个集合，第 i 个集合内初始状态下只有一个数，为 i。

有 m 次操作。操作分为 3 种：

- `1 a b` 合并 a, b 所在集合；
- `2 k` 回到第 k 次操作（执行三种操作中的任意一种都记为一次操作）之后的状态；

- `3 a b` 询问 a, b 是否属于同一集合，如果是则输出 1，否则输出 0。

```
#include<bits/stdc++.h>
using namespace std;
const int N = 2e5 + 5;

int n, m;
struct node {
    int val, rnk, ls, rs; // 父节点、树的深度、左右儿子编号
} pt[N << 5];
int a[N], rt[N], tot;
int build(int l, int r) {
    int now = ++ tot;
    if (l == r) {
        pt[now].val = a[l];
        pt[now].rnk = 1;
        return now;
    }
    pt[now].ls = build(l, mid);
    pt[now].rs = build(mid + 1, r);
    return now;
}
int update(int x, int l, int r, int tarjet, int newroot) {
    int now = ++ tot;
    pt[now] = pt[x];
    if (l == r) {
        pt[now].val = newroot;
        pt[newroot].rnk = max(pt[newroot].rnk, pt[now].rnk + 1);
        return now;
    }
    if (tarjet <= mid) pt[now].ls = update(pt[x].ls, l, mid, tarjet, newroot);
    else pt[now].rs = update(pt[x].rs, mid + 1, r, tarjet, newroot);
    return now;
}
int query(int x, int l, int r, int tarjet) {
    if (l == r) return pt[x].val;
    if (tarjet <= mid) return query(pt[x].ls, l, mid, tarjet);
    else return query(pt[x].rs, mid + 1, r, tarjet);
}
int getroot(int x, int l, int r, int tarjet) {
    int ans = query(x, l, r, tarjet);
    if (ans == tarjet) return ans;
    else return getroot(x, l, r, ans);
}
signed main() {
    cin.tie(nullptr)->sync_with_stdio(false);

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

```
    rt[0] = build(1, n);
    int op, a, b, k;
    for (int i = 1; i <= m; ++ i) {
        cin >> op;
        if (op == 1) {
            cin >> a >> b;
            int root_a = getroot(rt[i - 1], 1, n, a);
            int root_b = getroot(rt[i - 1], 1, n, b);
            if (pt[root_a].rnk < pt[root_b].rnk) {
                rt[i] = update(rt[i - 1], 1, n, root_a, root_b);
            } else {
                rt[i] = update(rt[i - 1], 1, n, root_b, root_a);
            }
        } else if (op == 2) {
            cin >> k;
            rt[i] = rt[k];
        } else {
            cin >> a >> b;
            rt[i] = rt[i - 1];
            cout << (getroot(rt[i], 1, n, a) == getroot(rt[i], 1, n, b)) << '\n';
        }
    }
    return 0;
}
```

单调栈

第 i 个元素之后第一个大于a_i的元素的**下标**

```
#include<bits/stdc++.h>
using LL = long long;

signed main() {
    std::cin.tie(nullptr)->sync_with_stdio(false);

    int n;
    std::cin >> n;
    std::vector< int > a(n), ans(n);
    for (auto &x : a) {
        std::cin >> x;
    }
    std::stack< int > st;
    for (int i = n - 1; i >= 0; -- i) {
        while (!st.empty() && a[st.top()] <= a[i]) {
            st.pop();
        }
```

```
        ans[i] = st.empty() ? -1 : st.top();
        st.push(i);
    }
    for (int i = 0; i < n; ++ i) {
        std::cout << ans[i] + 1 << " \n"[i + 1 == n];
    }
    return 0;
}
```

单调队列

区间长度为k的最大最小值

```
#include<bits/stdc++.h>
using LL = long long;

signed main() {
    std::cin.tie(nullptr)->sync_with_stdio(false);

    int n, k;
    std::cin >> n >> k;
    std::vector< int > a(n), Max(n), Min(n);
    for (auto &x : a) {
        std::cin >> x;
    }
    std::deque< int > p, q;
    for (int i = 0; i < n; ++ i) {
        while (!q.empty() && a[q.back()] >= a[i]) {
            q.pop_back();
        }
        q.push_back(i);
        if (q.back() - q.front() >= k) {
            q.pop_front();
        }
        if (i >= k - 1) {
            Min[i] = a[q.front()];
        }

        while (!p.empty() && a[p.back()] < a[i]) {
            p.pop_back();
        }
        p.push_back(i);
        if (p.back() - p.front() >= k) {
            p.pop_front();
        }
        if (i >= k - 1) {
            Max[i] = a[p.front()];
```

```
        }
    }
    for (int i = k - 1; i < n; ++ i) {
        std::cout << Min[i] << " \n"[i + 1 == n];
    }
    for (int i = k - 1; i < n; ++ i) {
        std::cout << Max[i] << " \n"[i + 1 == n];
    }
    return 0;
}
```

ST表

```
int lg[N];
struct Sparse_Table {
    vector< vector< int > > f;
    Sparse_Table(int n) : f(n + 1, vector< int > (lg[n] + 1, 0)) {}
    void Build(int n, int *a) {
        for (int i = 2; i <= n; ++ i) {
            lg[i] = lg[i >> 1] + 1;
        }
        for (int i = 1; i <= n; ++ i) {
            f[i][0] = a[i];
        }
        for (int j = 1; j <= lg[n]; ++ j) {
            for (int i = 1; i + (1 << j) - 1 <= n; ++ i) {
                f[i][j] = max(f[i][j - 1], f[i + (1 << (j - 1))][j - 1]);
            }
        }
    }
    int Query(int l, int r) {
        int k = lg[r - l + 1];
        return max(f[l][k], f[r - (1 << k) + 1][k]);
    }
};
```

树状数组

一维树状数组

单点修改,区间查询 OR 区间修改,单点查询

```
template< typename T >
struct FenWick {
```

```
    int n;
    std::vector< T > c;
    FenWick(int n) : n(n), c(n + 1) {}
    void add(int i, T d) {
        for (; i <= n; i += i & -i) {
            c[i] += d;
        }
    }
    void add(int l, int r, T d) {
        add(l, d);
        add(r + 1, -d);
    }
    T get(int i) {
        T sum = 0;
        for (; i; i -= i & -i) {
            sum += c[i];
        }
        return sum;
    }
    T get(int l, int r) {
        return get(r) - get(l - 1);
    }
};
```

区间修改,区间查询

```
template< typename T >
struct Fenwick {
    int n;
    std::vector< T > c1, c2;
    Fenwick(int n) : n(n), c1(n + 1), c2(c1) {}
    void add(int i, T d) {
        for (int j = i; i <= n; i += i & -i) {
            c1[i] += d;
            c2[i] += j * d;
        }
    }
    void add(int l, int r, T d) {
        add(l, d);
        add(r + 1, -d);
    }
    T get(int i) {
        T sum = 0;
        for (int j = i; i; i -= i & -i) {
            sum += (j + 1) * c1[i] - c2[i];
        }
        return sum;
    }
    T get(int l, int r) {
```

```
        return get(r) - get(l - 1);
    }
};
```

二维树状数组

单点修改,区间查询OR区间修改,单点查询

```
template< typename T >
struct Fenwick {
    int n, m;
    std::vector< vector< T > > c;
    Fenwick(int n, int m) : n(n), m(m), c(n + 1, vector< T > (m + 1)) {}
    void add(int x, int y, T d) {
        for (; x <= n; x += x & -x) {
            for (; y <= m; y += y & -y) {
                c[x][y] += d;
            }
        }
    }
    void add(int x1, int y1, int x2, int y2, T d) {
        add(x1, y1, d);
        add(x1, y2 + 1, -d);
        add(x2 + 1, y1, -d);
        add(x2 + 1, y2 + 1, d);
    }
    T get(int x, int y, T sum = 0) {
        for (; x; x -= x & -x) {
            for (; y; y -= y & -y) {
                sum += c[x][y];
            }
        }
        return sum;
    }
    T get(int x1, int y1, int x2, int y2) {
        return get(x2, y2) - get(x1 - 1, y2) - get(x2, y1 - 1) + get(x1 - 1, y1 - 1);
    }
};
```

区间修改,区间查询(左上角为$(x1,y1)$,右下角为$(x2,y2)$)

```
template< typename T >
struct Fenwick {
    int n, m;
    std::vector< vector< T > > c1, c2, c3, c4;
    Fenwick(int n, int m) : n(n), m(m), c1(n + 1, vector< T > (m + 1)), c2(c1), c3(c1),
c4(c1) {}
    void add(int x, int y, T d) {
```

```
        for (int i = x; i <= n; i += i & -i) {
            for (int j = y; j <= m; j += j & -j) {
                c1[i][j] += d;
                c2[i][j] += x * d;
                c3[i][j] += y * d;
                c4[i][j] += x * y * d;
            }
        }
    }
    void add(int x1, int y1, int x2, int y2, T d) {
        add(x1, y1, d);
        add(x1, y2 + 1, -d);
        add(x2 + 1, y1, -d);
        add(x2 + 1, y2 + 1, d);
    }
    T get(int x, int y, T sum = 0) {
        for (int i = x; i; i -= i & -i) {
            for (int j = y; j; j -= j & -j) {
                sum += (x + 1) * (y + 1) * c1[i][j];
                sum -= (y + 1) * c2[i][j];
                sum -= (x + 1) * c3[i][j];
                sum += c4[i][j];
            }
        }
        return sum;
    }
    T get(int x1, int y1, int x2, int y2) {
        return get(x2, y2) - get(x1 - 1, y2) - get(x2, y1 - 1) + get(x1 - 1, y1 - 1);
    }
};
```

二叉搜索树&平衡树

旋转Treap

```
#include <bits/stdc++.h>
using namespace std;
struct Node {
    Node *ch[2];
    int val, rank;
    int rep_cnt;
    int siz;
    Node(int val) : val(val), rep_cnt(1), siz(1) {
        ch[0] = ch[1] = nullptr;
        rank = rand();
    }
    void upd_siz() {
        siz = rep_cnt;
```

```
        if (ch[0] != nullptr) siz += ch[0]->siz;
        if (ch[1] != nullptr) siz += ch[1]->siz;
    }
};
class Treap {
private:
    Node *root;
    enum rot_type {
        LF = 1, RT = 0
    };
    int q_prev_tmp = 0, q_nex_tmp = 0;
    void _rotate(Node *&cur, rot_type dir) {  // 0为右旋，1为左旋
        Node *tmp = cur->ch[dir];
        cur->ch[dir] = tmp->ch[!dir];
        tmp->ch[!dir] = cur;
        tmp->upd_siz(), cur->upd_siz();
        cur = tmp;
    }
    void _insert(Node *&cur, int val) {
        if (cur == nullptr) {
            cur = new Node(val);
            return;
        } else if (val == cur->val) {
            cur->rep_cnt++;
            cur->siz++;
        } else if (val < cur->val) {
            _insert(cur->ch[0], val);
            if (cur->ch[0]->rank < cur->rank) {
                _rotate(cur, RT);
            }
            cur->upd_siz();
        } else {
            _insert(cur->ch[1], val);
            if (cur->ch[1]->rank < cur->rank) {
                _rotate(cur, LF);
            }
            cur->upd_siz();
        }
    }
    void _del(Node *&cur, int val) {
        if (val > cur->val) {
            _del(cur->ch[1], val);
            cur->upd_siz();
        } else if (val < cur->val) {
            _del(cur->ch[0], val);
            cur->upd_siz();
        } else {
            if (cur->rep_cnt > 1) {
                cur->rep_cnt--, cur->siz--;
```

```
                return;
            }
            uint8_t state = 0;
            state |= (cur->ch[0] != nullptr);
            state |= ((cur->ch[1] != nullptr) << 1);
            // 00都无，01有左无右，10，无左有右，11都有
            Node *tmp = cur;
            switch (state) {
                case 0:
                    delete cur;
                    cur = nullptr;
                    break;
                case 1:  // 有左无右
                    cur = tmp->ch[0];
                    delete tmp;
                    break;
                case 2:  // 有右无左
                    cur = tmp->ch[1];
                    delete tmp;
                    break;
                case 3:
                    rot_type dir = cur->ch[0]->rank < cur->ch[1]->rank ? RT : LF;
                    _rotate(cur, dir);
                    _del(cur->ch[!dir], val);
                    cur->upd_siz();
                    break;
            }
        }
    }
    int _query_rank(Node *cur, int val) {
        int less_siz = cur->ch[0] == nullptr ? 0 : cur->ch[0]->siz;
        if (val == cur->val)
            return less_siz + 1;
        else if (val < cur->val) {
            if (cur->ch[0] != nullptr)
                return _query_rank(cur->ch[0], val);
            else
                return 1;
        } else {
            if (cur->ch[1] != nullptr)
                return less_siz + cur->rep_cnt + _query_rank(cur->ch[1], val);
            else
                return cur->siz + 1;
        }
    }
    int _query_val(Node *cur, int rank) {
        int less_siz = cur->ch[0] == nullptr ? 0 : cur->ch[0]->siz;
        if (rank <= less_siz)
            return _query_val(cur->ch[0], rank);
```

```
        else if (rank <= less_siz + cur->rep_cnt)
            return cur->val;
        else
            return _query_val(cur->ch[1], rank - less_siz - cur->rep_cnt);
    }
    int _query_prev(Node *cur, int val) {
        if (val <= cur->val) {
            if (cur->ch[0] != nullptr) return _query_prev(cur->ch[0], val);
        } else {
            q_prev_tmp = cur->val;
            if (cur->ch[1] != nullptr) _query_prev(cur->ch[1], val);
            return q_prev_tmp;
        }
        return -1145;
    }
    int _query_nex(Node *cur, int val) {
        if (val >= cur->val) {
            if (cur->ch[1] != nullptr) return _query_nex(cur->ch[1], val);
        } else {
            q_nex_tmp = cur->val;
            if (cur->ch[0] != nullptr) _query_nex(cur->ch[0], val);
            return q_nex_tmp;
        }
        return -1145;
    }
public:
    void insert(int val) { _insert(root, val); }
    void del(int val) { _del(root, val); }
    int query_rank(int val) { return _query_rank(root, val); }
    int query_val(int rank) { return _query_val(root, rank); }
    int query_prev(int val) { return _query_prev(root, val); }
    int query_nex(int val) { return _query_nex(root, val); }
};

Treap tr;
int main() {
    srand(0);
    int t;
    scanf("%d", &t);
    while (t--) {
        int mode;
        int num;
        scanf("%d%d", &mode, &num);
        switch (mode) {
            case 1: tr.insert(num); break;
            case 2: tr.del(num); break;
            case 3: printf("%d\n", tr.query_rank(num)); break;
            case 4: printf("%d\n", tr.query_val(num)); break;
            case 5: printf("%d\n", tr.query_prev(num)); break;
```

```
            case 6: printf("%d\n", tr.query_nex(num)); break;
        }
    }
}
```

无旋Treap

区间翻转

```
#include <bits/stdc++.h>
using namespace std;
struct Node {
    Node *ch[2];
    int val, prio;
    int cnt;
    int siz;
    bool to_rev = false;
    Node(int _val) : val(_val), cnt(1), siz(1) {
        ch[0] = ch[1] = nullptr;
        prio = rand();
    }
    inline int upd_siz() {
        siz = cnt;
        if (ch[0] != nullptr) {
            siz += ch[0]->siz;
        }
        if (ch[1] != nullptr) {
            siz += ch[1]->siz;
        }
        return siz;
    }
    inline void pushdown() {
        swap(ch[0], ch[1]);
        if (ch[0] != nullptr) {
            ch[0]->to_rev ^= 1;
        }
        if (ch[1] != nullptr) {
            ch[1]->to_rev ^= 1;
        }
        to_rev = false;
    }
    inline void check_tag() {
        if (to_rev) {
            pushdown();
        }
    }
};
struct Seg_treap {
```

```cpp
Node *root;
#define siz(_) (_ == nullptr ? 0 : _->siz)
array< Node*, 2 > split(Node *cur, int sz) {
    if (cur == nullptr) return {nullptr, nullptr};
    cur->check_tag();
    if (sz <= siz(cur->ch[0])) {
        auto temp = split(cur->ch[0], sz);
        cur->ch[0] = temp[1];
        cur->upd_siz();
        return {temp[0], cur};
    } else {
        auto temp = split(cur->ch[1], sz - siz(cur->ch[0]) - 1);
        cur->ch[1] = temp[0];
        cur->upd_siz();
        return {cur, temp[1]};
    }
}
Node *merge(Node *sm, Node *bg) {
    if (sm == nullptr && bg == nullptr) {
        return nullptr;
    }
    if (sm != nullptr && bg == nullptr) {
        return sm;
    }
    if (sm == nullptr && bg != nullptr) {
        return bg;
    }
    sm->check_tag(), bg->check_tag();
    if (sm->prio < bg->prio) {
        sm->ch[1] = merge(sm->ch[1], bg);
        sm->upd_siz();
        return sm;
    } else {
        bg->ch[0] = merge(sm, bg->ch[0]);
        bg->upd_siz();
        return bg;
    }
}
void insert(int val) {
    auto temp = split(root, val);
    auto l_tr = split(temp[0], val - 1);
    Node *new_node;
    if (l_tr[1] == nullptr) new_node = new Node(val);
    Node *l_tr_combined = merge(l_tr[0], l_tr[1] == nullptr ? new_node : l_tr[1]);
    root = merge(l_tr_combined, temp[1]);
}
void seg_rev(int l, int r) {
    auto less = split(root, l - 1);
    auto more = split(less[1], r - l + 1);
```

```
        more[0]->to_rev = true;
        root = merge(less[0], merge(more[0], more[1]));
    }
    void print(Node *cur) {
        if (cur == nullptr) return;
        cur->check_tag();
        print(cur->ch[0]);
        cout << cur->val << " ";
        print(cur->ch[1]);
    }
};
Seg_treap tr;
int main() {
    srand(time(NULL));
    int n, m;
    cin >> n >> m;
    for (int i = 1; i <= n; i++) {
        tr.insert(i);
    }
    while (m--) {
        int l, r;
        cin >> l >> r;
        tr.seg_rev(l, r);
    }
    tr.print(tr.root);
}
```

线段树

拓展性极强

好像也没什么板子, 纯手敲...

关于区间gcd,可以维护差分数组d,然后查询$\gcd\left(a[L], \gcd\left(d[L+1], \cdots, d[R]\right)\right)$

若有区间修改,则$\mathrm{d[L]+x, d[R+1]-x}$, 再维护一个区间加和单点查询

```
struct Info {
    int l, r;
    LL sum, lz;
    friend Info operator + (const Info &l, const Info &r) {
        Info rt;
        rt.l = l.l;
        rt.r = r.r;
        rt.sum = l.sum + r.sum;
        rt.lz = 0;
```

```
        return rt;
    }
};
namespace SegmentTree {
    Info inf[N << 2];
    void Push_up(int x) {
        inf[x] = inf[x << 1] + inf[x << 1 | 1];
    }
    void Build(int l, int r, int x = 1) {
        if (l == r) {
            inf[x] = {l, l, 0, 0};
            return;
        }
        int mid = (l + r) >> 1;
        Build(l, mid, x << 1);
        Build(mid + 1, r, x << 1 | 1);
        Push_up(x);
    }
    void Set_lz(int x, LL v) {
        inf[x].lz += inf[x].lz;
        inf[x].sum += (inf[x].r - inf[x].l + 1) * inf[x].lz;
    }
    void Push_down(int x) {
        if (!inf[x].lz) {
            return;
        }
        Set_lz(x << 1, inf[x].lz);
        Set_lz(x << 1 | 1, inf[x].lz);
        inf[x].lz = 0;
    }
    void Update(int ql, int qr, LL val, int x = 1) {
        if (ql > qr || ql > inf[x].r || qr < inf[x].l) {
            return;
        }
        if (ql <= inf[x].l && inf[x].r <= qr) {
            Set_lz(x, val);
            return;
        }
        Push_down(x);
        Update(ql, qr, val, x << 1);
        Update(ql, qr, val, x << 1 | 1);
        Push_up(x);
    }
    Info Query(int ql, int qr, int x = 1) {
        if (ql <= inf[x].l && inf[x].r <= qr) {
            return inf[x];
        }
        Push_down(x);
        int mid = (inf[x].l + inf[x].r) >> 1;
```

```
        if (qr <= mid) {
            return Query(ql, qr, x << 1);
        } else if (ql > mid) {
            return Query(ql, qr, x << 1 | 1);
        } else {
            return Query(ql, qr, x << 1) + Query(ql, qr, x << 1 | 1);
        }
    }
};
```

线段树合并&分裂

[P5494 【模板】线段树分裂]

```
struct Info {
    int cnt;
    Info() : cnt(0) {}
    friend Info operator + (const Info &l, const Info &r) {
        Info rt;
        rt.cnt = l.cnt + r.cnt;
        return rt;
    }
};
int n, q, a[N], col[N], rt[N];
set< int > s;
set< int >::iterator it;
struct SegmentTree {
    int tot;
    int col[N];
    int ls[N << 5], rs[N << 5], rub[N << 5], Bottom;
    Info inf[N << 5];

    SegmentTree() : tot(0), Bottom(0) {}
    // 新开节点
    int New() {
        return Bottom ? rub[Bottom --] : ++ tot;
    }
    // 删除节点
    void Delete(int &x) {
        ls[x] = rs[x] = 0;
        inf[x] = Info();
        rub[++ Bottom] = x;
        x = 0;
    }
    void Push_Up(int x) {
        inf[x] = inf[ls[x]] + inf[rs[x]];
    }
    // 使树x里权值为pos的个数增加k
    void Modify(int &x, int l, int r, int pos, LL k) {
```

```
    if (!x) {
        x = New();
    }
    if (l == r) {
        inf[x].cnt += k;
        return;
    }
    int mid = (l + r) >> 1;
    if (pos <= mid) {
        Modify(ls[x], l, mid, pos, k);
    } else {
        Modify(rs[x], mid + 1, r, pos, k);
    }
    Push_Up(x);
}
// 合并xy
int Merge(int x, int y, int l, int r) {
    if (!x || !y) {
        return x | y;
    }
    int z = New();
    if (l == r) {
        inf[z] = inf[x] + inf[y];
    } else {
        int mid = (l + r) >> 1;
        ls[z] = Merge(ls[x], ls[y], l, mid);
        rs[z] = Merge(rs[x], rs[y], mid + 1, r);
        Push_Up(z);
        Delete(x); // 删除,看情况
        Delete(y);
    }
    return z;
}
// 将树x的[ql,qr]里的信息分裂到树y里
void Split(int &x, int &y, int l, int r, int ql, int qr) {
    if (ql > qr || ql > r || qr < l) {
        return;
    }
    if (ql <= l && r <= qr) {
        y = x;
        x = 0;
        return;
    }
    y = ++ tot;
    int mid = (l + r) >> 1;
    Split(ls[x], ls[y], l, mid, ql, qr);
    Split(rs[x], rs[y], mid + 1, r, ql, qr);
    Push_Up(x);
    Push_Up(y);
```

```
}
// 前k个给x,其余给y
void split(int p, int &x, int &y, int l, int r, int k) {
    if (!p) {
        x = y = 0;
        return;
    }
    if (l == r) { // 断边
        if (k) {
            x = p;
            y = 0;
        } else {
            y = p;
            x = 0;
        }
        return;
    }
    int mid = (l + r) >> 1;
    if (inf[ls[p]].cnt >= k) { // 右儿子给y, 递归左儿子
        y = p;
        x = New();
        split(ls[p], ls[x], ls[y], l, mid, k);
    } else { // 左儿子给x, 递归右儿子
        x = p;
        y = New();
        split(rs[p], rs[x], rs[y], mid + 1, r, k - inf[ls[p]].cnt);
    }
    Push_Up(x);
    Push_Up(y);
}
// 在树x里查询第k大
int Query_Kth(int x, int l, int r, int k) {
    if (l == r) {
        return l;
    }
    int mid = (l + r) >> 1;
    if (inf[ls[x]].cnt >= k) {
        return Query_Kth(ls[x], l, mid, k);
    }
    return Query_Kth(rs[x], mid + 1, r, k - inf[ls[x]].cnt);
}
// 在树x里查询[ql,qr]的个数和
LL Query_Cnt(int x, int l, int r, int ql, int qr) {
    if (ql > qr || ql > r || qr < l) {
        return 0;
    }
    if (ql <= l && r <= qr) {
        return inf[x].cnt;
    }
```

```cpp
        int mid = (l + r) >> 1;
        return Query_Cnt(ls[x], l, mid, ql, qr) + Query_Cnt(rs[x], mid + 1, r, ql, qr);
    }
    int query(int x, int l, int r) {
        if (l == r) {
            return l;
        }
        int mid = (l + r) >> 1;
        if (inf[ls[x]].cnt) {
            return query(ls[x], l, mid);
        }
        return query(rs[x], mid + 1, r);
    }
    // 将区间分裂成[l,x]和(x,r]
    void split(int p) {
        it = s.lower_bound(p);
        if (*it == p) {
            return;
        }
        int r = *it, l = *prev(it) + 1;
        if (col[r]) {
            split(rt[r], rt[r], rt[p], 1, n, r - p);
        } else {
            split(rt[r], rt[p], rt[r], 1, n, p - l + 1);
        }
        col[p] = col[r];
        s.insert(p);
    }
    // 区间排序,0升序,1降序
    void Sort(int l, int r, int op) {
        split(l - 1);
        split(r);
        int x = 0;
        for (it = s.lower_bound(l); *it <= r;) {
            x = Merge(x, rt[*it], 1, n);
            rt[*it] = 0;
            set< int >::iterator IT = it;
            ++ it;
            s.erase(IT);
        }
        rt[r] = x;
        col[r] = op;
        s.insert(r);
    }
} st;
```

可持久化权值线段树

```
struct President_Tree {
    int tot;
    int rt[N], ls[N << 5], rs[N << 5], sum[N << 5], cnt[N << 5];

    int insert(int x, int l, int r, int pos, int k) {
        int now = ++ tot;
        cnt[now] = cnt[x] + k;
        sum[now] = sum[x] + pos * k; // 若需离散化,则为初始值
        ls[now] = ls[x], rs[now] = rs[x];
        if (l == r) {
            return now;
        }
        int mid = (l + r) >> 1;
        if (pos <= mid) {
            ls[now] = insert(ls[x], l, mid, pos, k);
        } else {
            rs[now] = insert(rs[x], mid + 1, r, pos, k);
        }
        return now;
    }
    // 第k小
    int query_kth(int x, int y, int l, int r, int kth) {
        if (l == r) {
            return l;
        }
        int lcnt = cnt[ls[x]] - cnt[ls[y]];
        int mid = (l + r) >> 1;
        if (lcnt >= kth) {
            return query_kth(ls[x], ls[y], l, mid, kth);
        }
        return query_kth(rs[x], rs[y], mid + 1, r, kth - lcnt);
    }
    int query_cnt(int x, int y, int l, int r, int ql, int qr) {
        if (ql > qr || ql > r || qr < l) {
            return 0;
        }
        if (ql <= l && r <= qr) {
            return cnt[x] - cnt[y];
        }
        int mid = (l + r) >> 1;
        return query_cnt(ls[x], ls[y], l, mid, ql, qr) + query_cnt(rs[x], rs[y], mid +
1, r, ql, qr);
    }
    int query_sum(int x, int y, int l, int r, int ql, int qr) {
        if (ql > qr || ql > r || qr < l) {
            return 0;
        }
```

```
        if (ql <= l && r <= qr) {
            return sum[x] - sum[y];
        }
        int mid = (l + r) >> 1;
        return query_sum(ls[x], ls[y], l, mid, ql, qr) + query_sum(rs[x], rs[y], mid +
1, r, ql, qr);
    }
};
```

树上主席树

$\mathrm{rt}[\mathrm{x}] = \mathrm{insert}(\mathrm{rt}[\mathrm{y}], 1, \mathrm{m}, \mathrm{val}[\mathrm{y}]), \mathrm{y} \in \mathrm{son}[\mathrm{x}]$

查询第k小时需要对$x, y, \mathrm{lca}(\mathrm{x}, \mathrm{y}), \mathrm{fa}[\mathrm{lca}(\mathrm{x}, \mathrm{y})]$一并查询

$\mathrm{lcnt} = \mathrm{ls}[\mathrm{x}] + \mathrm{ls}[\mathrm{y}] - \mathrm{ls}[\mathrm{lca}(\mathrm{x}, \mathrm{y})] - \mathrm{ls}[\mathrm{fa}[\mathrm{lca}(\mathrm{x}, \mathrm{y})]]$

带修主席树

```
#include<bits/stdc++.h>
using namespace std;
typedef long long ll;
const int N = 2e5 + 5, int INF = 0x3f3f3f3f, int mod = 1e9 + 7;

int n, m;
int a[N];
int b[N], num;
int lnum, rnum;
int root[N], L[N], R[N], tot;
struct Seg { int l, r, cnt;} tree[N * 200];
#define l(x) tree[x].l
#define r(x) tree[x].r
#define cnt(x) tree[x].cnt
int x[N], y[N], z[N];
bool vis[N];
inline int lowbit(int x) { return x & -x;}
int Get_Pos(int x) {
    return lower_bound(b + 1, b + num + 1, x) - b;
}
int update(int now, int l, int r, int pos, int val) {
    int x = ++ tot;
    tree[x] = Seg{l(now), r(now), cnt(now) + val};
    if(l == r) return x;
    int mid = (l + r) / 2;
    if(pos <= mid) l(x) = update(l(now), l, mid, pos, val);
    else r(x) = update(r(now), mid + 1, r, pos, val);
    return x;
}
```

```cpp
void add(int x, int pos, int val) {
    for(int i = x; i <= n; i += lowbit(i)) {
        root[i] = update(root[i], 1, num, pos, val);
    }
}
int query(int l, int r, int k) {
    if(l == r) return l;
    int lcnt = 0;
    for(int i = 1; i <= lnum; ++ i) lcnt -= cnt(l(L[i]));
    for(int i = 1; i <= rnum; ++ i) lcnt += cnt(l(R[i]));
    int mid = (l + r) / 2;
    if(lcnt >= k) {
        for(int i = 1; i <= lnum; ++ i) L[i] = l(L[i]);
        for(int i = 1; i <= rnum; ++ i) R[i] = l(R[i]);
        return query(l, mid, k);
    } else {
        for(int i = 1; i <= lnum; ++ i) L[i] = r(L[i]);
        for(int i = 1; i <= rnum; ++ i) R[i] = r(R[i]);
        return query(mid + 1, r, k - lcnt);
    }
}
signed main() {
    cin.tie(nullptr)->sync_with_stdio(false);

    cin >> n >> m;
    for(int i = 1; i <= n; ++ i) {
        cin >> a[i];
        b[++ num] = a[i];
    }
    for(int i = 1; i <= m; ++ i) {
        string s;
        cin >> s;
        if(s == "C") {
            cin >> x[i] >> y[i];
            vis[i] = true;
            b[++ num] = y[i];
        } else {
            cin >> x[i] >> y[i] >> z[i];
        }
    }
    sort(b + 1, b + num + 1);
    num = unique(b + 1, b + num + 1) - (b + 1);
    for(int i = 1; i <= n; ++ i) add(i, Get_Pos(a[i]), 1);
    for(int i = 1; i <= m; ++ i) {
        if(vis[i]) {
            add(x[i], Get_Pos(a[x[i]]), -1);
            a[x[i]] = y[i];
            add(x[i], Get_Pos(a[x[i]]), 1);
        } else {
```

```
            lnum = rnum = 0;
            for(int j = x[i] - 1; j; j -= lowbit(j)) L[++ lnum] = root[j];
            for(int j = y[i]; j; j -= lowbit(j)) R[++ rnum] = root[j];
            int pos = query(1, num, z[i]);
            cout << b[pos] << '\n';
        }
    }
    return 0;
}
```

珂朵莉树

[珂朵莉树 - OI Wiki]

核心思想：把值相同的区间合并成几个节点保存在set里。

时间复杂度：$\Theta(q \times \log n \times \log n)$

```
struct Node {
    int l, r;
    mutable int v; // 永远都可以被修改
    bool operator < (const Node &o) const {
        return l < o.l;
    }
};
set< Node > odt;
set< Node >::iterator Split(int x) {
    auto it = prev(odt.upper_bound({x, 0, 0}));
    if (it->l == x) {
        return it;
    }
    auto [l, r, v] = *it;
    odt.erase(it);
    odt.insert({l, x - 1, v});
    return odt.insert({x, r, v}).first;
}
void Assign(int l, int r, int v) {
    auto R = Split(r + 1), L = Split(l);
    // TODO
    odt.erase(L, R);
    odt.insert({l, r, v});
}
void Maintain(int l, int r) {
    auto R = Split(r + 1), L = Split(l);
    for (auto x = L; x != R; ++ x) {
        // TODO
    }
}
```

可持久化01Trie

```
constexpr int ALPHA_SIZE = 30, N = 2e5 + 5;
struct Trie {
    int cur, rt[N], ch[N * ALPHA_SIZE][2], val[N * ALPHA_SIZE];

    void insert(int o, int lst, int v) {
        for (int i = ALPHA_SIZE; i >= 0; -- i) {
            val[o] = val[lst] + 1;  // 在原版本的基础上更新
            if ((v & (1 << i)) == 0) {
                if (!ch[o][0]) {
                    ch[o][0] = ++ cur;
                }
                ch[o][1] = ch[lst][1];
                o = ch[o][0];
                lst = ch[lst][0];
            } else {
                if (!ch[o][1]) {
                    ch[o][1] = ++ cur;
                }
                ch[o][0] = ch[lst][0];
                o = ch[o][1];
                lst = ch[lst][1];
            }
        }
        val[o] = val[lst] + 1;
        // printf("%d\n",o);
    }
    // 查询[l, r]内max{a[i]^v}
    // cout << query(st.rt[r], st.rt[l - 1], v) << '\n';
    int query(int o1, int o2, int v) {
        int ret = 0;
        for (int i = ALPHA_SIZE; i >= 0; -- i) {
            // printf("%d %d %d\n",o1,o2,val[o1]-val[o2]);
            int t = ((v & (1 << i)) ? 1 : 0);
            if (val[ch[o1][!t]] - val[ch[o2][!t]]) {
                ret += (1 << i);
                o1 = ch[o1][!t];
                o2 = ch[o2][!t];  // 尽量向不同的地方跳
            } else {
                o1 = ch[o1][t];
                o2 = ch[o2][t];
            }
        }
```

```
        return ret;
    }
};
```

例题:Five Day Couple

```
#include<bits/stdc++.h>
using namespace std;
using LL = long long;
const int N = 5e6 + 5, MOD = 998244353, INF = 0x3f3f3f3f;

struct Trie {
    int cnt, rt[N], ch[N * 33][2], val[N * 33];

    void insert(int o, int lst, int v) {
        for (int i = 30; i >= 0; i--) {
            val[o] = val[lst] + 1;  // 在原版本的基础上更新
            if ((v & (1 << i)) == 0) {
                if (!ch[o][0]) ch[o][0] = ++cnt;
                ch[o][1] = ch[lst][1];
                o = ch[o][0];
                lst = ch[lst][0];
            } else {
                if (!ch[o][1]) ch[o][1] = ++cnt;
                ch[o][0] = ch[lst][0];
                o = ch[o][1];
                lst = ch[lst][1];
            }
        }
        val[o] = val[lst] + 1;
    }
    int query(int o1, int o2, int v) {
        int ret = 0;
        for (int i = 30; i >= 0; i--) {
            int t = ((v & (1 << i)) ? 1 : 0);
            if (val[ch[o1][!t]] - val[ch[o2][!t]])
                ret += (1 << i), o1 = ch[o1][!t],
                o2 = ch[o2][!t];  // 尽量向不同的地方跳
            else
                o1 = ch[o1][t], o2 = ch[o2][t];
        }
        return ret;
    }
} st;
int n, m, a[N];
signed main() {
    cin.tie(nullptr)->sync_with_stdio(false);

    cin >> n;
```

```
    for (int i = 1; i <= n; ++ i) {
        cin >> a[i];
        st.rt[i] = ++ st.cnt;
        st.insert(st.rt[i], st.rt[i - 1], a[i]);
    }
    cin >> m;
    while (m -- ) {
        int c, l, r;
        cin >> c >> l >> r;
        cout << st.query(st.rt[r], st.rt[max(l - 1, 0)], c) << '\n';
    }
    return 0;
}
```

根号数据结构

分块

思想: 将长度为n的序列均分为$\sqrt{n}$个长度为$\sqrt{n}$的连续子序列,并记录每个'块'的编号、始末下标,然后进行操作.

每次操作时间复杂度为$\theta\left(\sqrt{\mathrm{n}}\right)$,总的时间复杂度为$\Theta\left(\mathrm{n}\times\sqrt{\mathrm{n}}\right)$

初始化

$\Theta\left(n\right)$

```
int num, st[N], ed[N], belong[N], siz[N];
void Pre_work(int n) {
    num = sqrt(n);
    for (int i = 1; i <= num; ++ i) {
        st[i] = n / num * (i - 1) + 1;
        ed[i] = n / num * i;
    }
    ed[num] = n;
    for (int i = 1; i <= num; ++ i) {
        for (int j = st[i]; j <= ed[i]; ++ j) {
            belong[j] = i;
        }
        siz[i] = ed[i] - st[i] + 1;
    }
}
```

例题:[Yuno loves sqrt technology III]

询问区间众数出现次数

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

```cpp
using namespace std;
using LL = long long;
const int N = 5e5 + 5, MOD = 998244353, INF = 0x3f3f3f3f;

int n, m, a[N], b[N], tot;
vector< int > pos[N];
int st[N], ed[N], belong[N], siz[N], num;
int f[720][720], cnt[N], idx[N];
int Query(int l, int r) {
    int x = belong[l], y = belong[r], ans = 0;
    if (x == y) {
        for (int i = l; i <= r; ++ i)
            cnt[a[i]] = 0;
        for (int i = l; i <= r; ++ i)
            ans = max(ans, ++ cnt[a[i]]);
        return ans;
    }
    ans = f[x + 1][y - 1];
    for (int i = l; i <= ed[x]; ++ i) {
        while (idx[i] + ans < pos[a[i]].size() && pos[a[i]][idx[i] + ans] <= r)
            ++ ans;
    }
    for (int i = st[y]; i <= r; ++ i) {
        while (idx[i] - ans >= 0 && pos[a[i]][idx[i] - ans] >= l)
            ++ ans;
    }
    return ans;
}
signed main() {
    cin.tie(nullptr)->sync_with_stdio(false);

    cin >> n >> m;
    for (int i = 1; i <= n; ++ i) {
        cin >> a[i];
        b[i] = a[i];
    }
    sort(b + 1, b + n + 1);
    tot = unique(b + 1, b + n + 1) - (b + 1);
    for (int i = 1; i <= n; ++ i) {
        a[i] = lower_bound(b + 1, b + tot + 1, a[i]) - b;
        pos[a[i]].push_back(i);
        idx[i] = (int)pos[a[i]].size() - 1;
    }
    num = sqrt(n);
    for (int i = 1; i <= num; ++ i) {
        st[i] = n / num * (i - 1) + 1;
        ed[i] = n / num * i;
    }
    ed[num] = n;
```

```
    for (int i = 1; i <= num; ++ i) {
        for (int j = st[i]; j <= ed[i]; ++ j) {
            belong[j] = i;
        }
        siz[i] = ed[i] - st[i] + 1;
        memset(cnt, 0, sizeof cnt);
        for (int j = i; j <= num; ++ j) {
            f[i][j] = f[i][j - 1];
            for (int k = st[j]; k <= ed[j]; ++ k) {
                f[i][j] = max(f[i][j], ++ cnt[a[k]]);
            }
        }
    }
    int lastans = 0;
    while (m -- ) {
        int l, r;
        cin >> l >> r;
        l ^= lastans;
        r ^= lastans;
        cout << (lastans = Query(l, r)) << '\n';

    }
    return 0;
}
```

莫队

奇偶化排序,减少左右指针移动次数

```
bool operator < (const Query &o) const {
        return l / block == o.l / block ? r != o.r && ((l / block) & 1) ^ (r < o.r) : l
< o.l;
    }
```

普通莫队套路都一样

```
int L = 1, R = 0;
    for (int i = 1; i <= q; ++ i) {
        auto &[l, r, id] = Q[i];
        while (L > l) {
            add(-- L);
        }
        while (R < r) {
            add(++ R);
        }
        while (L < l) {
            del(L ++ );
        }
```

```
        while (R > r) {
            del(R -- );
        }
        ans[id] = res;
    }
```

树上莫队通过欧拉序将查询路径线段化,特判lca

例题:COT2 - Count on a tree II

求$u \Rightarrow v$路径上不同颜色个数

```
#include<bits/stdc++.h>
using namespace std;
using LL = long long;
const int N = 5e5 + 5;

int n, q, a[N];
int b[N], m;
vector< int > adj[N];
int seg[N], fir[N], las[N], num;
int dep[N], f[N][20], t;
void dfs(int x, int p) {
    seg[++ num] = x; fir[x] = num;
    for (int i = 1; i <= t; ++ i) {
        f[x][i] = f[f[x][i - 1]][i - 1];
    }
    for (auto &y : adj[x]) {
        if (y != p) {
            dep[y] = dep[x] + 1;
            f[y][0] = x;
            dfs(y, x);
        }
    }
    seg[++ num] = x; las[x] = num;
}
int lca(int x, int y) {
    if (dep[x] > dep[y]) swap(x, y);
    for (int i = t; ~i; -- i) {
        if (dep[f[y][i]] >= dep[x]) y = f[y][i];
    }
    if (x == y) return x;
    for (int i = t; ~i; -- i) {
        if (f[x][i] != f[y][i]) {
            x = f[x][i];
            y = f[y][i];
        }
    }
    return f[x][0];
}
```

```cpp
int block;
struct Query {
    int l, r, id, p;
    bool operator < (const Query &x) const {
        return l / block != x.l / block ? l / block < x.l / block : r < x.r;
    }
} Q[N];
bool vis[N];
int ans[N], cnt[N], res;
void add(int x) {
    if (++ cnt[a[x]] == 1) ++ res;
}
void del(int x) {
    if (-- cnt[a[x]] == 0) -- res;
}
void option(int x) {
    vis[x] ^= 1;
    if (!vis[x]) del(x);
    else add(x);
}
signed main() {
    cin.tie(nullptr)->sync_with_stdio(false);

    cin >> n >> q;
    for (int i = 1; i <= n; ++ i) {
        cin >> a[i];
        b[i] = a[i];
    }
    sort(b + 1, b + n + 1);
    m = unique(b + 1, b + n + 1) - (b + 1);
    for (int i = 1; i <= n; ++ i) {
        a[i] = lower_bound(b + 1, b + m + 1, a[i]) - b;
    }
    for (int i = 1; i < n; ++ i) {
        int u, v;
        cin >> u >> v;
        adj[u].push_back(v);
        adj[v].push_back(u);
    }
    t = ceil(log2(n));
    dep[1] = 1;
    dfs(1, 0);
    block = sqrt(n * 2);
    for (int i = 1; i <= q; ++ i) {
        int x, y;
        cin >> x >> y;
        if (fir[x] > fir[y]) swap(x, y);
        int p = lca(x, y);
        if (x == p) Q[i] = {fir[x], fir[y], i, 0};
```

```
        else Q[i] = {las[x], fir[y], i, p};
    }
    sort(Q + 1, Q + q + 1);
    int L = 1, R = 0;
    for (int i = 1; i <= q; ++ i) {
        auto &[l, r, id, p] = Q[i];
        while (L > l) option(seg[-- L]);
        while (R < r) option(seg[++ R]);
        while (L < l) option(seg[L ++ ]);
        while (R > r) option(seg[R -- ]);
        if (p) option(p);
        ans[id] = res;
        if (p) option(p);
    }
    for (int i = 1; i <= q; ++ i) {
        cout << ans[i] << '\n';
    }
    return 0;
}
```

树上启发式合并

子树信息的查询

- 遍历所有轻儿子，递归结束时消除它们的贡献
- 遍历所有重儿子，保留它的贡献
- 再计算当前子树中所有轻子树的贡献
- 更新答案
- 如果当前点是轻儿子，消除当前子树的贡献

```
/* 第一个dfs计算每个节点的重儿子son */
void calc(int x, int p, bool flag) {
    if (flag) {
        /* 相当于add操作 */
    } else {
        /* 相当于delete操作 */
    }
    for (int i = head[x]; i; i = Next[i]) {
        int y = ver[i];
        if (!vis[y] && y != p) {
            calc(y, x, flag);
        }
    }
}
void dfs2(int x, int p, bool keep) {
    for (int i = head[x]; i; i = Next[i]) {
        int y = ver[i];
        if (y != son[x] && y != p) {
```

```
            dfs2(y, x, false);
        }
    }
    if (son[x]) {
        dfs2(son[x], x, true);
        vis[son[x]] = true;
    }
    calc(x, p, true), vis[son[x]] = false;
    /* 统计答案 */
    if (!keep) {
        calc(x, p, false); // 消除轻子树的贡献
    }
}
```

由于每个节点到跟节点最多经过$\log n$条轻边和$\log n$条重边,所以总的时间复杂度是优秀的$O(n \log n)$！

路径信息的查询

计算几何

还没学.先放个板子

```
const double eps = 1e-8;
int sign(double x) {
    if (fabs(x) < eps) {
        return 0;
    }
    if (x < 0) {
        return -1;
    }
    return 1;
}
int cmp(double x, double y) {
    if (fabs(x - y) < eps) {
        return 0;
    }
    if (x < y) {
        return -1;
    }
    return 1;
}
struct Point {
    double x, y;
    Point(double x = 0, double y = 0) : x(x), y(y) {}
    bool operator<(const Point& B) const {
        return x == B.x ? y < B.y : x < B.x;
    }
    bool operator==(const Point& B) const {
```

```
        return !sign(x - B.x) && !sign(y - B.y);
    }
    Point operator+(const Point& B) const {
        return Point(x + B.x, y + B.y);
    }
    Point operator-(const Point& B) const {
        return Point(x - B.x, y - B.y);
    }
    Point operator*(const double a) const {
        return Point(x * a, y * a);
    }
    Point operator/(const double a) const {
        return Point(x / a, y / a);
    }
    double operator*(const Point& B) const {
        return x * B.x + y * B.y;
    }
    double operator^(const Point& B) const {
        return x * B.y - y * B.x;
    }
    double length() {
        return sqrt(x * x + y * y);
    }
    Point trunc(double r) {  // 化为长度为r的向量
        double l = length();
        if (!sign(l)) {
            return *this;
        }
        r /= l;
        return Point(x * r, y * r);
    }
    Point toleft() {
        return Point(-y, x);
    }
    friend int relation(Point a, Point b, Point c) {
        int flag = sign((b - a) ^ (c - a));
        if (flag > 0) {
            return 1;
        } else if (flag < 0) {
            return -1;
        }
        return 0;
    }
};
using Vector = Point;
struct Line {
    Point p;
    Vector v;
    double rad;
```

```
    Line() {}
    Line(Point P, Vector V) : p(P), v(V) {
        rad = atan2(v.y, v.x);
    }
    Point get_point_in_line(double t) {
        return p + v * t;
    }
    bool operator<(const Line& L) const {
        if (!cmp(rad, L.rad)) {
            return L.onLeft(p) > 0;
        }
        return rad < L.rad;
    }
    int onLeft(const Point& a) const {  // 点a是否在直线的左边(>0:左  <0:右)
        return relation(p, p + v, a);
    }
    friend Point getIntersection(Line a, Line b) {
        Vector u = a.p - b.p;
        double t = (b.v ^ u) / (a.v ^ b.v);
        return a.get_point_in_line(t);
    }
    friend bool on_right(const Line& a, const Line& b, const Line& c) {  // b,c的交点是否在直线a的右边
        Point o = getIntersection(b, c);
        return a.onLeft(o) <= 0;
    }
    friend Line getTranslation(Line a, double dist) {  // 将直线a向左(逆时针)平移dist
        Vector u = a.v.toleft().trunc(dist);
        return Line(a.p + u, a.v);
    };
};
vector<Point> half_plane(vector<Line> line) {
    sort(line.begin(), line.end());
    int hh = 0, tt = -1, num = line.size();
    vector<int> q(num + 10);
    for (int i = 0; i < num; ++ i) {
        if (i && !cmp(line[i].rad, line[i - 1].rad)) {
            continue;
        }
        while (hh + 1 <= tt && on_right(line[i], line[q[tt - 1]], line[q[tt]])) {
            -- tt;
        }
        while (hh + 1 <= tt && on_right(line[i], line[q[hh]], line[q[hh + 1]])) {
            ++ hh;
        }
        q[++ tt] = i;
    }
    while (hh + 1 <= tt && on_right(line[q[hh]], line[q[tt - 1]], line[q[tt]])) {
        -- tt;
```

```
    }
    while (hh + 1 <= tt && on_right(line[q[tt]], line[q[hh]], line[q[hh + 1]])) {
        ++ hh;
    }
    if (tt - hh + 1 <= 2) {
        return {};
    }
    q[++ tt] = q[hh];
    vector<Point> res;
    for (int i = hh; i < tt; ++ i) {
        res.push_back(getIntersection(line[q[i]], line[q[i + 1]]));
    }
    return res;
}
double getArea(vector<Point>& p) {
    int n = p.size();
    double ans = 0;
    for (int i = 0; i < n; ++ i) {
        ans += (p[i] ^ p[(i + 1) % n]);
    }
    return ans / 2;
}
```

字符串

字符串Hash

$Hash[S+T] = Hash[S] \times P^{length[T]} + Hash[T]$其中,P代表将字符串以P进制存储

```
using ULL = unsigned long long;
struct Hash {
    ULL pw[N];
    const ULL base = 131ULL;
    void Calc_Power() {
        pw[0] = 1;
        for (int i = 1; i < N; ++ i) {
            pw[i] = pw[i - 1] * base;
        }
    }
    vector< ULL > Get_hash(const string &str) {
        vector< ULL > hs(str.size() + 1);
        for (int i = 0; i < str.size(); ++ i) {
            hs[i + 1] = hs[i] * base + str[i];
        }
        return hs;
```

```
    }
    ULL Get_val(const vector< ULL > &hs, int l, int r) {
        return hs[r] - hs[l - 1] * pw[r - l + 1];
    }
};
```

KMP

$next[i]$代表S中以i结尾的非前缀子串与S的前缀能够匹配的最长长度

```
// index from 1 to n
vector< int > calc_next(char *s, int n) {
    vector< int > next(n + 1);
    for (int i = 2, j = 0; i <= n; ++ i) {
        while (j > 0 && s[i] != s[j + 1]) {
            j = next[j];
        }
        if (s[i] == s[j + 1]) {
            ++ j;
        }
        next[i] = j;
    }
    return next;
}
// if i % (i - next[i]) == 0
// s[1 ~ i]的最小循环节长度为i - next[i], 循环次数为i / (i - next[i])
vector< int > match(char *s, int n, char *t, int m) {
    vector< int > f(m + 1);
    vector< int > next = calc_next(s, n);
    for (int i = 1, j = 0; i <= m; ++ i) {
        while (j > 0 && (j == n || t[i] != s[j + 1])) {
            j = next[j];
        }
        if (t[i] == s[j + 1]) {
            ++ j;
        }
        f[i] = j; // if (f[i] == n) 此时就是A在B中的某一次出现
    }
    return f;
}
```

exKMP

定义$z[i]$代表S的前缀与S中以i开头的后缀能够匹配的最长长度

```
vector<int> z_function(string s) {
```

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

Manacher

```
char s[N], str[N * 2];
int Len[N * 2];
int len;

void init() {
    int k = 0;
    str[k++] = '@';
    for (int i = 0; i < len; ++i) {
        str[k++] = '#';
        str[k++] = s[i];
    }
    str[k++] = '#';
    len = k;
    str[k] = '?';
}

int manacher() {
    int mx = 0, id = 0;
    int ans = 0;
    for (int i = 1; i <= len - 1; ++ i) {
        if (mx > i) {
            Len[i] = min(Len[id * 2 - i], mx - i);
        }  else {
            Len[i] = 1;
```

```
        }
        while (str[i - Len[i]] == str[i + Len[i]]) {
            ++ Len[i];
        }
        if (Len[i] + i > mx) {
            mx = Len[i] + i;
            id = i;
        }
        ans = max(ans, Len[i]);
    }
    return ans - 1;
}
```

AC自动机

fail 指针指向所有模式串的前缀中匹配当前状态的最长后缀

fail指针相当于S的后缀集合

[P5357 【模板】AC 自动机（二次加强版）](P5357)

求出每个模式串T_i在文本串S中出现次数

```
struct Aho_Corasick_Automaton {
    static constexpr int ALPFABET_SIZE = 26, N = 1e5;
    int trie[N][ALPFABET_SIZE], fail[N], cnt[N], tot;

    void Insert(const std::string &str) {
        int rt = 0;
        for (const auto &x : str) {
            if (!trie[rt][x - 'a']) {
                trie[rt][x - 'a'] = ++ tot;
            }
            rt = trie[rt][x - 'a'];
        }
        ++ cnt[rt];
    }

    void Build() {
        std::queue< int > q;
        for (int i = 0; i < ALPFABET_SIZE; ++ i) {
            if (trie[0][i]) {
                q.push(trie[0][i]);
            }
        }
        while (!q.empty()) {
            int x = q.front();
            q.pop();
            for (int i = 0; i < ALPFABET_SIZE; ++ i) {
                if (trie[x][i]) {
```

```
                    fail[trie[x][i]] = trie[fail[x]][i];
                    q.push(trie[x][i]);
                } else {
                    trie[x][i] = trie[fail[x]][i]; // 路径压缩
                }
            }
        }
    }
};
```

后缀自动机

len代表每个endpos等价类串的最长长度,$\mathrm{len}[\mathrm{fa}[\mathrm{a}]]+1=\mathrm{minlen}(\mathrm{a})$

[【模板】后缀自动机 (SAM)]

请你求出s的所有出现次数不为1的子串的出现次数乘上该子串长度的最大值

```
struct Suffix_Automaton {
    static constexpr int ALPHBET_SIZE = 26, N = 1e6;
    int last = 1, cntNodes = 1;
    int ch[N * 2][ALPHBET_SIZE], fa[N * 2], len[N * 2];


    int Extend(int c) {
        int p = last, np = last = ++ cntNodes;
        len[np] = len[p] + 1;
        while (p && !ch[p][c]) {
            ch[p][c] = np;
            p = fa[p];
        }
        if (!p) {
            fa[np] = 1;
        } else {
            int q = ch[p][c];
            if (len[p] + 1 == len[q]) {
                fa[np] = q;
            } else {
                int nq = ++ cntNodes;
                len[nq] = len[p] + 1;
                std::copy(ch[q], ch[q] + ALPHBET_SIZE, ch[nq]);
                fa[nq] = fa[q];
                fa[q] = fa[np] = nq;
                while (p && ch[p][c] == q) {
                    ch[p][c] = nq;
                    p = fa[p];
                }
            }
```

```
        }
        return np;
    }
};
```

回文自动机

每个节点代表了一个回文串,$\mathrm{ch}[\mathrm{S}][\mathrm{c}]$代表在状态$S$的左右各添加一个字符$c$

那么从根节点出发沿着$\mathrm{ch}[\mathrm{S}][\mathrm{c}]$即可组成$\mathrm{cSc}$

$\mathrm{fail}[\mathrm{S}]$代表回文串$S$的最长回文后缀,$\mathrm{last}$代表上一个字符结尾的最长回文子串

[P5496 【模板】回文自动机（PAM）](#)

求以$s[i]$结尾的回文串个数

```
struct Palindromic_Automaton {
    int s[N], top; // 原串
    int ch[N][26], fail[N], len[N], tot, last;

    Palindromic_Automaton() {
        s[0] = -114514;
        tot = -1;
        New(0); // 偶根
        New(-1); // 奇根
        fail[0] = 1;
        last = 0;
    }

    int New(int length) {
        len[++ tot] = length;
        return tot;
    }

    int Get_Fail(int x) { // 找最长的回文后缀
        while (s[top - len[x] - 1] != s[top]) {
            x = fail[x];
        }
        return x;
    }
    void Extend(int c) {
        s[++ top] = c;
        int now = Get_Fail(last);
        if (!ch[now][c]) {
            int x = New(len[now] + 2);
            fail[x] = ch[Get_Fail(fail[now])][c];
            ch[now][c] = x;
```

```
        }
        last = ch[now][c];
    }
};
```

动态规划

数位dp

例题 windy数

不含前导零且相邻两个数字之差至少为 2 的正整数被称为 windy 数。windy 想知道，在 a 和 b 之间，包括 a 和 b ，总共有多少个 windy 数?

对于全部的测试点，保证 $1 \leq a \leq b \leq 2 \times 10^9$。

```
#include<bits/stdc++.h>
using namespace std;
using LL = long long;
const int N = 205;

int a[N], cnt;
int dp[N][N][2][2];
int dfs(int pos, int pre, bool lead, bool limit) {
    if (pos > cnt) return 1;
    if (dp[pos][pre][lead][limit]) return dp[pos][pre][lead][limit];
    int res = limit ? a[pos] : 9;
    int ans = 0;
    for (int i = 0; i <= res; ++ i) {
        if (abs(i - pre) < 2) continue;
        if (lead && i == 0) ans += dfs(pos + 1, pre, lead, limit && i == a[pos]);
        else ans += dfs(pos + 1, i, 0, limit && i == a[pos]);
    }
    return dp[pos][pre][lead][limit] = ans;
}
int calc(int x) {
    cnt = 0;
    memset(dp, 0, sizeof dp);
    for (cnt = 0; x ; x /= 10) a[++ cnt] = x % 10;
    reverse(a + 1, a + cnt + 1);
    return dfs(1, -2, 1, 1);
}
signed main() {
    cin.tie(nullptr)->sync_with_stdio(false);
```

```
    int l, r;
    cin >> l >> r;
    cout << calc(r) - calc(l - 1) << '\n';
    return 0;
}
```

最长上升子序列计数

$\Theta(n^2)$

```
pair< int, int > dp[N];
cin >> n;
for (int i = 1; i <= n; ++ i) {
    cin >> a[i];
}
int max_len = 1;
for (int i = 1; i <= n; ++ i) {
    dp[i] = {1, 1};
    for (int j = 1; j < i; ++ j) {
        if (a[j] <= a[i]) {
            if (dp[j].first + 1 > dp[i].first) {
                dp[i] = {dp[j].first + 1, dp[j].second};
            } else if (dp[j].first + 1 == dp[i].first) {
                dp[i].second += dp[j].second;
            }
        }
    }
    max_len = max(max_len, dp[i].first);
}
int ans = 0;
for (int i = 1; i <= n; ++ i) {
    if (dp[i].first == max_len)
        ans += dp[i].second;
}
cout << ans << '\n';
```

$\Theta(n \log n)$

```
#include<bits/stdc++.h>
using namespace std;
using LL = long long;
const int N = 4e5 + 5, MOD = 1e9 + 7;

int n, a[N];
struct SegTree {
    int len, cnt;
```

```cpp
} st[N << 2];
SegTree operator + (const SegTree &l, const SegTree &r) {
    if (!l.len || r.len > l.len)
        return r;
    if (!r.len || l.len > r.len)
        return l;
    return {l.len, (l.cnt + r.cnt) % MOD};
}
void build(int x, int l, int r) {
    if (l == r) {
        st[x] = {0, 1};
        return;
    }
    int mid = (l + r) >> 1;
    build(x << 1, l, mid);
    build(x << 1 | 1, mid + 1, r);
    st[x] = st[x << 1] + st[x << 1 | 1];
}
void update(int x, int l, int r, int pos, const SegTree &val) {
    if (l == r) {
        st[x] = val;
        return;
    }
    int mid = (l + r) >> 1;
    if (pos <= mid)
        update(x << 1, l, mid, pos, val);
    else
        update(x << 1 | 1, mid + 1, r, pos, val);
    st[x] = st[x << 1] + st[x << 1 | 1];
}
SegTree query(int x, int l, int r, int ql, int qr) {
    if (ql > qr || ql > r || qr < l)
        return {0, 1};
    if (ql <= l && r <= qr)
        return st[x];
    int mid = (l + r) >> 1;
    return query(x << 1, l, mid, ql, qr) + query(x << 1 | 1, mid + 1, r, ql, qr);
}
int b[N], m;
signed main() {
    cin.tie(nullptr)->sync_with_stdio(false);

    cin >> n;
    for (int i = 1; i <= n; ++ i) {
        cin >> a[i];
        b[i] = a[i];
    }
    sort(b + 1, b + n + 1);
    m = unique(b + 1, b + n + 1) - (b + 1);
```

```
    build(1, 1, m);
    int max_len = 0;
    for (int i = 1; i <= n; ++ i) {
        a[i] = lower_bound(b + 1, b + m + 1, a[i]) - b;
        auto x = query(1, 1, m, 1, a[i] - 1);
        ++ x.len;
        max_len = max(max_len, x.len);
        auto y = query(1, 1, m, a[i], a[i]);
        // 当且仅当f[j]+1=f[i]才能将其合并
        update(1, 1, m, a[i], x + y);
        // auto z = query(1, 1, m, a[i], a[i]);
        // cout << z.len << ' ' << z.cnt << '\n';
    }
    int ans = 0;
    for (int i = 1; i <= m; ++ i) {
        auto x = query(1, 1, m, i, i);
        if (x.len == max_len)
            ans = (ans + x.cnt) % MOD;
    }
    cout << ans << '\n';
    return 0;
}
```

数学

数论分块

给定n和k, 求$\sum_{i=1}^{n} k\%i, 1 \leq n, k \leq 10^9$

```
LL ans = 0;
for (int x = 1, gx; x <= n; x = gx + 1) {
    gx = k / x ? min(k / (k / x), n) : n;
    ans -= 1LL * (k / x) * (x + gx) * (gx - x + 1) / 2;
}
```

扩展欧几里得算法

对于任意整数a, b,存在一对整数x, y,满足$a \cdot x + \mathrm{b} \cdot \mathrm{y} = \mathrm{gcd}(\mathrm{a}, \mathrm{b})$

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

矩阵快速幂

求Fibonacci第n项

$(0 \leq n \leq 2 \times 10^9)$

```
void mul(int f[2], int a[2][2]) {
    int c[2];
    memset(c, 0, sizeof c);
    for (int j = 0; j < 2; ++ j) {
        for (int k = 0; k < 2; ++ k) {
            c[j] = (c[j] + (LL)f[k] * a[k][j]) % MOD;
        }
    }
    memcpy(f, c, sizeof c);
}
void mulself(int a[2][2]) {
    int c[2][2];
    memset(c, 0, sizeof c);
    for (int i = 0; i < 2; ++ i) {
        for (int j = 0; j < 2; ++ j) {
            for (int k = 0; k < 2; ++ k) {
                c[i][j] = (c[i][j] + (LL)a[i][k] * a[k][j]) % MOD;
            }
        }
    }
    memcpy(a, c, sizeof c);
}
void solve() {
    cin >> n;
    int f[2] = {0, 1};
    int a[2][2] = {{0, 1}, {1, 1}};
    for (; n; n >>= 1) {
        if (n & 1) mul(f, a);
        mulself(a);
    }
    cout << f[0] << '\n';
}
```

高斯消元

```
double a[20][20], b[20], c[20][20];
int n;
void solve() {
    cin >> n;
    for (int i = 1; i <= n + 1; ++ i) {
        for (int j = 1; j <= n; ++ j) {
            cin >> a[i][j];
        }
    }
    // c:系数矩阵,b:常数,二者一起构成增广矩阵
    for (int i = 1; i <= n; ++ i) {
        for (int j = 1; j <= n; ++ j) {
            c[i][j] = 2 * (a[i][j] - a[i + 1][j]);
            b[i] += a[i][j] * a[i][j] - a[i + 1][j] * a[i + 1][j];
        }
    }
    // 高斯消元(数据保证一定有解)
    for (int i = 1; i <= n; ++ i) {
        // 找到x[i]的系数不为0的一个方程
        for (int j = i; j <= n; ++ j) {
            if (fabs(c[j][i]) > 1e-8) {
                for (int k = 1; k <= n; ++ k) {
                    swap(c[i][k], c[j][k]);
                }
                swap(b[i], b[j]);
            }
        }
        // 消去其他方程的x[i]的系数
        for (int j = 1; j <= n; ++ j) {
            if (i == j) {
                continue;
            }
            double rate = c[j][i] / c[i][i];
            for (int k = i; k <= n; ++ k) {
                c[j][k] -= c[i][k] * rate;
            }
            b[j] -= b[i] * rate;
        }
    }
    for (int i = 1; i <= n; ++ i) {
        printf("%.3f%c", b[i] / c[i][i], i == n ? '\n' : ' ');
    }
}
```

组合数取模相关

```cpp
Mint inv[N], fac[N], finv[N];
void Pre_Work() {
    fac[0] = fac[1] = 1;
    inv[1] = 1;
    finv[0] = finv[1] = 1;
    for(int i = 2; i < N; ++ i) {
        fac[i] = fac[i-1] * i;
        inv[i] = P - P / i * inv[P % i];
        finv[i] = finv[i - 1] * inv[i];
    }
}
Mint C(int x, int y) {
    return fac[x] * finv[x - y] * finv[y];
}
```

线性筛

最小质因子

```cpp
constexpr int NUMBER_SIZE = 2e5 + 5;
int v[NUMBER_SIZE], pri[NUMBER_SIZE], Pcnt;
void Pre_Work() {
    Pcnt = 0;
    for (int i = 2; i < NUMBER_SIZE; ++ i) {
        if (!v[i]) {
            v[i] = i;
            pri[tot ++ ] = i;
        }
        for (int j = 0; j < Pcnt && 1LL * pri[j] * i < NUMBER_SIZE; ++ j) {
            if (pri[j] > v[i]) {
                break;
            }
            v[pri[j] * i] = pri[j];
        }
    }
}
```

欧拉函数

```cpp
constexpr int NUMBER_SIZE = 2e5 + 5;
int phi[NUMBER_SIZE], pri[NUMBER_SIZE], Pcnt;
bool is_prime[NUMBER_SIZE];
void Pre_Work() {
    std::fill(is_prime + 1, is_prime + NUMBER_SIZE, 1);
    Pcnt = 0;
```

```
    is_prime[1] = 0;
    phi[1] = 1;
    for (int i = 2; i < NUMBER_SIZE; ++ i) {
        if (is_prime[i]) {
            pri[++ Pcnt] = i;
            phi[i] = i - 1;
        }
        for (int j = 1; j <= Pcnt && 1LL * i * pri[j] < NUMBER_SIZE; ++ j) {
            is_prime[i * pri[j]] = 0;
            if (i % pri[j]) {
                phi[i * pri[j]] = phi[i] * phi[pri[j]];
            } else {
                phi[i * pri[j]] = phi[i] * pri[j];
                break;
            }
        }
    }
}
```

莫比乌斯函数

```
constexpr int NUMBER_SIZE = 2e5 + 5;
int mu[NUMBER_SIZE], pri[NUMBER_SIZE], v[NUMBER_SIZE], Pcnt;
void Pre_Work() {
    Pcnt = 0;
    mu[1] = 1;
    for (int i = 2; i < NUMBER_SIZE; ++ i) {
        if (!v[i]) {
            mu[i] = -1;
            pri[Pcnt ++ ] = i;
        }
        for (int j = 0; j < Pcnt && 1LL * i * pri[j] < NUMBER_SIZE; ++ j) {
            v[i * pri[j]] = 1;
            if (i % pri[j] == 0) {
                mu[i * pri[j]] = 0;
                break;
            }
            mu[i * pri[j]] = -mu[i];
        }
    }
}
```

约数个数

```
constexpr int NUMBER_SIZE = 2e5 + 5;
int d[NUMBER_SIZE], pri[NUMBER_SIZE], num[NUMBER_SIZE], Pcnt;
bool v[NUMBER_SIZE];
void Pre_Work() {
    Pcnt = 0;
    d[1] = 1;
    for (int i = 2; i < NUMBER_SIZE; ++ i) {
        if (!v[i]) {
            v[i] = 1;
            pri[Pcnt ++ ] = i;
            d[i] = 2;
            num[i] = 1;
        }
        for (int j = 0; j < Pcnt && 1LL * i * pri[j] < NUMBER_SIZE; ++ j) {
            v[pri[j] * i] = 1;
            if (i % pri[j] == 0) {
                num[i * pri[j]] = num[i] + 1;
                d[i * pri[j]] = d[i] / num[i * pri[j]] * (num[i * pri[j]] + 1);
                break;
            } else {
                num[i * pri[j]] = 1;
                d[i * pri[j]] = d[i] * 2;
            }
        }
    }
}
```

约数和

```
constexpr int NUMBER_SIZE = 2e5 + 5;
int g[NUMBER_SIZE], f[NUMBER_SIZE], pri[NUMBER_SIZE], Pcnt;
bool v[NUMBER_SIZE];
void Pre_Work() {
    Pcnt = 0;
    g[1] = f[1] = 1;
    for (int i = 2; i < NUMBER_SIZE; ++ i) {
        if (!v[i]) {
            v[i] = 1;
            pri[Pcnt ++ ] = i;
            g[i] = i + 1;
            f[i] = i + 1;
        }
        for (int j = 0; j < Pcnt && 1LL * i * pri[j] < NUMBER_SIZE; ++ j) {
            v[pri[j] * i] = 1;
            if (i % pri[j] == 0) {
                g[i * pri[j]] = g[i] * pri[j] + 1;
```

```
                f[i * pri[j]] = f[i] / g[i] * g[i * pri[j]];
                break;
            } else {
                f[i * pri[j]] = f[i] * f[pri[j]];
                g[i * pri[j]] = 1 + pri[j];
            }
        }
    }
}
```

杂项

快读快写

```
template < typename T > void read(T &x) {
    x = 0;
    T f = 1;
    char ch = getchar();
    while (ch < '0' || ch > '9')  {
        if (ch == '-') {
            f = -1;
        }
        ch = getchar();
    }
    while (ch >= '0' && ch <= '9') {
        x = (x << 1) + (x << 3) + (ch ^ 48);
        ch = getchar();
    }
    x = x * f;
}
template < typename T > void write(T x) {
    if (x < 0) {
        putchar('-');
        x = -x;
    }
    if (x < 10) {
        putchar(x + '0');
    } else {
        write(x / 10);
        putchar(x % 10 + '0');
    }
}
template < typename T, typename ...Arg > void read(T &x, Arg &...arg) {
    read(x);
```

```
    read(arg...);
}
template < typename T, typename ...Arg > void write(T &x,Arg &...arg) {
    write(x);
    putchar(' ');
    write(arg...);
}
```

吸氧优化

```
#pragma GCC optimize(3, "Ofast")
#pragma GCC optimize(2, "Ofast")
```

取模运算

```
constexpr int P = 998244353;
// assume -P <= x < 2P
int norm(int x) {
    if (x < 0) {
        x += P;
    }
    if (x >= P) {
        x -= P;
    }
    return x;
}
template<class T>
T power(T a, LL b) {
    T res = 1;
    for (; b; b /= 2, a *= a) {
        if (b % 2) {
            res *= a;
        }
    }
    return res;
}
struct Mint {
    int x;
    Mint(int x = 0) : x(norm(x)) {}
    Mint(LL x) : x(norm(x % P)) {}
    int val() const {
        return x;
    }
    Mint operator-() const {
        return Mint(norm(P - x));
    }
    Mint inv() const {
```

```
        assert(x != 0);
        return power(*this, P - 2);
    }
    Mint &operator*=(const Mint &rhs) {
        x = LL(x) * rhs.x % P;
        return *this;
    }
    Mint &operator+=(const Mint &rhs) {
        x = norm(x + rhs.x);
        return *this;
    }
    Mint &operator-=(const Mint &rhs) {
        x = norm(x - rhs.x);
        return *this;
    }
    Mint &operator/=(const Mint &rhs) {
        return *this *= rhs.inv();
    }
    friend Mint operator*(const Mint &lhs, const Mint &rhs) {
        Mint res = lhs;
        res *= rhs;
        return res;
    }
    friend Mint operator+(const Mint &lhs, const Mint &rhs) {
        Mint res = lhs;
        res += rhs;
        return res;
    }
    friend Mint operator-(const Mint &lhs, const Mint &rhs) {
        Mint res = lhs;
        res -= rhs;
        return res;
    }
    friend Mint operator/(const Mint &lhs, const Mint &rhs) {
        Mint res = lhs;
        res /= rhs;
        return res;
    }
    friend std::istream &operator>>(std::istream &is, Mint &a) {
        LL v;
        is >> v;
        a = Mint(v);
        return is;
    }
    friend std::ostream &operator<<(std::ostream &os, const Mint &a) {
        return os << a.val();
    }
};
```

随机化

随机排序

```
std::mt19937 rng(std::chrono::system_clock::now().time_since_epoch().count());
std::shuffle(begin(a), end(a), rng);
```

程序运行时间

```
std::cout << (double)clock() / CLOCKS_PER_SEC * 1000 << " ms\n";
```

高精度运算

```
class bign {
public:
    int len, s[N];//数的长度，记录数组
//构造函数
    bign();

    bign(const char *);

    bign(int);

    bool sign;//符号 1正数 0负数
    string toStr() const;//转化为字符串，主要是便于输出
    friend istream &operator>>(istream &, bign &);//重载输入流
    friend ostream &operator<<(ostream &, bign &);//重载输出流
//重载复制
    bign operator=(const char *);

    bign operator=(int);

    bign operator=(const string);

//重载各种比较
    bool operator>(const bign &) const;

    bool operator>=(const bign &) const;

    bool operator<(const bign &) const;
```

```
    bool operator<=(const bign &) const;

    bool operator==(const bign &) const;

    bool operator!=(const bign &) const;

//重载四则运算
    bign operator+(const bign &) const;

    bign operator++();

    bign operator++(int);

    bign operator+=(const bign &);

    bign operator-(const bign &) const;

    bign operator--();

    bign operator--(int);

    bign operator-=(const bign &);

    bign operator*(const bign &) const;

    bign operator*(const int num) const;

    bign operator*=(const bign &);

    bign operator/(const bign &) const;

    bign operator/=(const bign &);

//四则运算的衍生运算
    bign operator%(const bign &) const;//取模（余数）
    bign factorial() const;//阶乘
    bign Sqrt() const;//整数开根（向下取整）
    bign pow(const bign &) const;//次方
//一些乱乱的函数
    void clean();

    ~bign();
};

bign::bign() {
    memset(s, 0, sizeof(s));
    len = 1;
    sign = 1;
```

```
}

bign::bign(const char *num) {
    *this = num;
}

bign::bign(int num) {
    *this = num;
}

string bign::toStr() const {
    string res;
    res = "";
    for (int i = 0; i < len; i++) {
        res = (char) (s[i] + '0') + res;
    }
    if (res == "") {
        res = "0";
    }
    if (!sign && res != "0") {
        res = "-" + res;
    }
    return res;
}

istream &operator>>(istream &in, bign &num) {
    string str;
    in >> str;
    num = str;
    return in;
}

ostream &operator<<(ostream &out, bign &num) {
    out << num.toStr();
    return out;
}

bign bign::operator=(const char *num) {
    memset(s, 0, sizeof(s));
    char a[N] = "";
    if (num[0] != '-') {
        strcpy(a, num);
    } else {
        for (int i = 1; i < strlen(num); ++ i) {
            a[i - 1] = num[i];
        }
    }
    sign = !(num[0] == '-');
    len = strlen(a);
```

```
    for (int i = 0; i < strlen(a); ++ i) {
        s[i] = a[len - i - 1] - 48;
    }
    return *this;
}

bign bign::operator=(int num) {
    char temp[N];
    sprintf(temp, "%d", num);
    *this = temp;
    return *this;
}

bign bign::operator=(const string num) {
    const char *tmp;
    tmp = num.c_str();
    *this = tmp;
    return *this;
}

bool bign::operator<(const bign &num) const {
    if (sign ^ num.sign) {
        return num.sign;
    }
    if (len != num.len) {
        return len < num.len;
    }
    for (int i = len - 1; i >= 0; -- i) {
        if (s[i] != num.s[i]) {
            return sign ? (s[i] < num.s[i]) : (!(s[i] < num.s[i]));
        }
    }
    return !sign;
}

bool bign::operator>(const bign &num) const {
    return num < *this;
}

bool bign::operator<=(const bign &num) const {
    return !(*this > num);
}

bool bign::operator>=(const bign &num) const {
    return !(*this < num);
}

bool bign::operator!=(const bign &num) const {
    return *this > num || *this < num;
```

```cpp
}

bool bign::operator==(const bign &num) const {
    return !(num != *this);
}

bign bign::operator+(const bign &num) const {
    if (sign ^ num.sign) {
        bign tmp = sign ? num : *this;
        tmp.sign = 1;
        return sign ? *this - tmp : num - tmp;
    }
    bign result;
    result.len = 0;
    int temp = 0;
    for (int i = 0; temp || i < (max(len, num.len)); ++ i) {
        int t = s[i] + num.s[i] + temp;
        result.s[result.len ++ ] = t % 10;
        temp = t / 10;
    }
    result.sign = sign;
    return result;
}

bign bign::operator++() {
    *this = *this + 1;
    return *this;
}

bign bign::operator++(int) {
    bign old = *this;
    ++(*this);
    return old;
}

bign bign::operator+=(const bign &num) {
    *this = *this + num;
    return *this;
}

bign bign::operator-(const bign &num) const {
    bign b = num, a = *this;
    if (!num.sign && !sign) {
        b.sign = 1;
        a.sign = 1;
        return b - a;
    }
    if (!b.sign) {
        b.sign = 1;
```

```
        return a + b;
    }
    if (!a.sign) {
        a.sign = 1;
        b = bign(0) - (a + b);
        return b;
    }
    if (a < b) {
        bign c = (b - a);
        c.sign = false;
        return c;
    }
    bign result;
    result.len = 0;
    for (int i = 0, g = 0; i < a.len; ++ i) {
        int x = a.s[i] - g;
        if (i < b.len) {
            x -= b.s[i];
        }
        if (x >= 0) {
            g = 0;
        }
        else {
            g = 1;
            x += 10;
        }
        result.s[result.len ++ ] = x;
    }
    result.clean();
    return result;
}

bign bign::operator*(const bign &num) const {
    bign result;
    result.len = len + num.len;

    for (int i = 0; i < len; ++ i) {
        for (int j = 0; j < num.len; ++ j) {
            result.s[i + j] += s[i] * num.s[j];
        }
    }

    for (int i = 0; i < result.len; ++ i) {
        result.s[i + 1] += result.s[i] / 10;
        result.s[i] %= 10;
    }
    result.clean();
    result.sign = !(sign ^ num.sign);
    return result;
```

```
}

bign bign::operator*(const int num) const {
    bign x = num;
    bign z = *this;
    return x * z;
}

bign bign::operator*=(const bign &num) {
    *this = *this * num;
    return *this;
}

bign bign::operator/(const bign &num) const {
    bign ans;
    ans.len = len - num.len + 1;
    if (ans.len < 0) {
        ans.len = 1;
        return ans;
    }

    bign divisor = *this, divid = num;
    divisor.sign = divid.sign = 1;
    int k = ans.len - 1;
    int j = len - 1;
    while (k >= 0) {
        while (divisor.s[j] == 0) {
            -- j;
        }
        if (k > j) k = j;
        char z[N];
        memset(z, 0, sizeof(z));
        for (int i = j; i >= k; -- i) {
            z[j - i] = divisor.s[i] + '0';
        }
        bign dividend = z;
        if (dividend < divid) {
            -- k;
            continue;
        }
        int key = 0;
        while (divid * key <= dividend) {
            ++ key;
        }
        -- key;
        ans.s[k] = key;
        bign temp = divid * key;
        for (int i = 0; i < k; ++ i) {
            temp = temp * 10;
```

```
        }
        divisor = divisor - temp;
        -- k;
    }
    ans.clean();
    ans.sign = !(sign ^ num.sign);
    return ans;
}

bign bign::operator/=(const bign &num) {
    *this = *this / num;
    return *this;
}

bign bign::operator%(const bign &num) const {
    bign a = *this, b = num;
    a.sign = b.sign = 1;
    bign result, temp = a / b * b;
    result = a - temp;
    result.sign = sign;
    return result;
}

bign bign::pow(const bign &num) const {
    bign result = 1;
    for (bign i = 0; i < num; ++ i) {
        result = result * (*this);
    }
    return result;
}

bign bign::factorial() const {
    bign result = 1;
    for (bign i = 1; i <= *this; ++ i) {
        result *= i;
    }
    return result;
}

void bign::clean() {
    if (len == 0) {
        ++ len;
    }
    while (len > 1 && s[len - 1] == '\0') {
        -- len;
    }
}

bign bign::Sqrt() const {
```

```
    if (*this < 0)return -1;
    if (*this <= 1)return *this;
    bign l = 0, r = *this, mid;
    while (r - l > 1) {
        mid = (l + r) / 2;
        if (mid * mid > *this) {
            r = mid;
        } else {
            l = mid;
        }
    }
    return l;
}

bign::~bign() {}
```