一、数据结构

1.1 并查集

1.1.1 路径压缩并查集。

```
const int N = 1e5 + 10;
int dad[N];
void init() {
    std::fill(std::begin(dad), std::end(dad), -1);
}
int find(int x) {
    if (dad[x] == -1)
        return x;
    return dad[x] = find(dad[x]);
}
bool merge(int a, int b) {
    a = find(a); b = find(b);
    if (a == b)
        return false;
    dad[b] = a;
    return true;
}
bool is_same(int a, int b) {
    return find(a) == find(b);
}
```

1.2 线段树

1.2.1 单点修改线段树

```
const int N = 1e5 + 10;
struct Info {
    int val;
    Info(int v = {}) : val(v) {}
    friend Info operator + (const Info &a, const Info &b) {
        return a.val + b.val;
    }
};
struct Node {
    Info val;
} seg[N * 4];
int a[N];
void rise(int id) {
    seg[id].val = seg[id * 2].val + seg[id * 2 + 1].val;
}
void build(int l, int r, int id) {
    if (l == r) {
```

```
        seg[id].val = a[l];
    } else {
        int mid = l + ((r - l) >> 1);
        build(l, mid, id * 2);
        build(mid + 1, r, id * 2 + 1);
        rise(id);
    }
}
void change(int l, int r, int id, int tag, int d) {
    if (l == r) {
        seg[id].val = d;
    } else {
        int mid = l + ((r - l) >> 1);
        if (tag <= mid)
            change(l, mid, id * 2, tag, d);
        else
            change(mid + 1, r, id * 2 + 1, tag, d);
        rise(id);
    }
}
Info query(int l, int r, int id, int ql, int qr) {
    if (l == ql && r == qr) {
        return seg[id].val;
    } else {
        int mid = l + ((r - l) >> 1);
        if (qr <= mid)
            return query(l, mid, id * 2, ql, qr);
        else if (ql > mid)
            return query(mid + 1, r, id * 2 + 1, ql, qr);
        else
            return query(l, mid, id * 2, ql, mid) + query(mid + 1, r, id * 2 + 1);
    }
}
```

1.2.2 势能线段树

原题意：对区间不断执行 $round(10\sqrt{x})$ 。

```
using ll = int64_t;
const int N = 1e5 + 10;
struct Info {
    ll val;
    bool ok;  // 判断是否改动
    Info(ll v = {}, bool ok = false) : val(v), ok(ok) {}
    void calc() {
        //...
    }
    friend Info operator + (const Info &a, const Info &b) {
        return {
            a.val + b.val,
            a.ok && b.ok
```

```
        };
    }
};
struct Node {
    Info val;
} seg[N * 4];
int a[N];
void rise(int id) {
    seg[id].val = seg[id * 2].val + seg[id * 2 + 1].val;
}
bool have = false;
void modify(int l, int r, int id, int ml, int mr) {
    if (seg[id].ok)
        return;
    if (l == ml && r == mr) {
        if (l == r) {
            auto bef = seg[id].val;
            seg[id].val.calc();
            if (bef != seg[id].val.val)
                have = true;
        } else {
            int mid = l + ((r - l) >> 1);
            if (!seg[id * 2].val.ok)
                modify(l, mid, id * 2, ml, mid);
            if (!seg[id * 2 + 1].val.ok)
                modify(mid + 1, r, id * 2 + 1, mid + 1, mr);
            rise(id);
        }
        return;
    }
    int mid = l + ((r - l) >> 1);
    if (mr <= mid) {
        modify(l, mid, id * 2, ml, mr);
    } else if (ml > mid) {
        modify(mid + 1, r, id * 2 + 1, ml, mr);
    } else {
        modify(l, mid, id * 2, ml, mid);
        modify(mid + 1, r, id * 2 + 1, mid + 1, mr);
    }
    rise(id);
}
```

1.2.3 区间修改线段树

```
using ll = int64_t;
const int N = 1e5 + 10;
struct Info {
    ll val;
    Info(ll v = {}) : val(v) {}
    friend Info operator + (const Info &a, const Info &b) {
        return Info {
```

```
            a.val + b.val
        };
    }
};
struct Lazy {
    ll add;
    Lazy(ll v = {}) : add(v) {}
};
struct Node {
    Info val;
    Lazy lz;
} seg[N * 4];
int a[N];

void rise(int id) {
    seg[id].val = seg[id * 2].val + seg[id * 2 + 1].val;
}
void update(int l, int r, int id, ll d) {
    seg[id].val.val += (r - l + 1) * d;
    seg[id].lz.add += d;
}
void down(int l, int r, int id) {
    if (seg[id].lz.add == 0)
        return;
    auto &d = seg[id].lz.add;
    int mid = l + ((r - l) >> 1);
    update(l, mid, id * 2, d);
    update(mid + 1, r, id * 2 + 1, d);
    d = 0;
}

void build(int l, int r, int id) {
    if (l == r) {
        seg[id].val = a[l];
    } else {
        int mid = l + ((r - l) >> 1);
        build(l, mid, id * 2);
        build(mid + 1, r, id * 2 + 1);
        rise(id);
    }
}
void add(int l, int r, int id, int ml, int mr, ll d) {
    if (l == ml && r == mr) {
        update(l, r, id, d);
    } else {
        int mid = l + ((r - l) >> 1);
        down(l, r, id);
        if (mr <= mid) {
            add(l, mid, id * 2, ml, mr, d);
        } else if (ml > mid) {
            add(mid + 1, r, id * 2 + 1, ml, mr, d);
        } else {
            add(l, mid, id * 2, ml, mid, d);
            add(mid + 1, r, id * 2 + 1, mid + 1, mr, d);
```

```
        }
        rise(id);
    }
}
Info query(int l, int r, int id, int ql, int qr) {
    if (l == ql && r == qr) {
        return seg[id].val;
    } else {
        int mid = l + ((r - l) >> 1);
        down(l, r, id);
        if (qr <= mid)
            return query(l, mid, id * 2, ql, qr);
        else if (ql > mid)
            return query(mid + 1, r, id * 2 + 1, ql, qr);
        else
            return query(l, mid, id * 2, ql, mid) + query(mid + 1, r, id * 2 + 1,
mid + 1, qr);
    }
}
```

1.2.4 可持久化线段树

```
using ll = int64_t;
const int N = 1e5 + 10;
struct Node {
    ll val;
    int lc, rc;
    Node() : val{}, lc{}, rc{} {}
} seg[N * 30];
int ver[N], tt = 0;
int a[N];
int getNode() {
    return ++ tt;
}
void extend(int oid, int nid) {
    seg[nid] = seg[oid];
}
void rise(int id) {
    seg[id].val = seg[id * 2].val + seg[id * 2 + 1].val;
}
void build(int l, int r, int &id) {
    id = getNode();
    if (l != r) {
        int mid = l + ((r - l) >> 1);
        build(l, mid, id * 2, seg[id].lc);
        build(mid + 1, r, seg[id].rc);
    }
}
void insert(int l, int r, int oid, int &nid, int tag, int d) {
    nid = getNode();
    if (l == r) {
```

```
            seg[nid].val = seg[oid].val + d;
        } else {
            int mid = l + ((r - l) >> 1);
            extend(oid, nid);
            if (tag <= mid) insert(l, mid, seg[oid].lc, seg[nid].lc, tag, d);
            else insert(mid + 1, r, seg[oid].rc, seg[nid].rc, tag, d);
            rise(nid);
        }
    }
    ll query(int l, int r, int lv, int rv, int ql, int qr) {
        if (l == ql && r == qr) {
            return seg[rv].val - seg[lv].val;
        } else {
            int mid = l + ((r - l) >> 1);
            if (qr <= mid)
                return query(l, mid, seg[lv].lc, seg[rv].lc, ql, qr);
            else if (ql > mid)
                return query(mid + 1, r, seg[lv].rc, seg[rv].rc, ql, qr);
            else
                return query(l, mid, seg[lv].lc, seg[rv].lc, ql, mid) + query(mid + 1,
    r, seg[lv].rc, seg[rv].rc, mid + 1, qr);
        }
    }
```

1.3 树状数组

1.3.1 一维树状数组

```
using ll = int64_t;
const int N = 1e6 + 10;
ll bit[N];
void update(int x, int n, ll d) {
    for (int i = x; i <= n; i += i & -i) bit[i] += d;
}
ll aks(int x) {
    ll ret = 0;
    for (int i = x; i > 0; i -= i & -i)
        ret += bit[i];
    return ret;
}
ll aks(int l, int r) {
    return ask(r) - ask(l - 1);
}
```

1.3.2 二维树状数组

```
using ll = int64_t;
const int N = 1e3 + 10;
ll bit[N][N];
```

```
void update(int x, int y, int n, int m, ll d) {
    for (int i = x; i <= n; i += i & -i)
        for (int j = y; j <= m; j += j & -j)
            bit[i][j] += d;
}
ll ask(int x, int y) {
    ll ret = 0;
    for (int i = x; i > 0; i -= i & -i)
        for (int j = y; j > 0; j -= j & -j)
            ret += bit[i][j];
    return ret;
}
ll ask(ll x1, ll y1, ll x2, ll y2) {
    return ask(x2, y2) - ask(x1 - 1, y2) - ask(x2, y2 - 1) + ask(x2 - 1, y2 - 1);
}
```

1.4 Spare Table

当区间满足以下几点时可以使用ST表。

1. 问题属于可重复贡献问题，即问题重复计算时结果不变；
2. 满足结合律。

诸如区间最大、最小、gcd等问题，符合上述要求。

```
template <typename T>
constexpr T inf = std::numeric_limits<T>::max() / 2;
void Main() {
    int n;
    cin >> n;
    vector<int> a(n + 1);
    for (int i = 1; i <= n; i ++)
        cin >> a[i];

    vector st(__lg(n) + 1, vector<int>(n + 1, -inf<int>));
    for (int i = 1; i <= n; i ++)
        st[0][i] = a[i];
    for (int i = 1; i <= __lg(n); i ++)
        for (int j = 1; j + (1 << i) - 1 <= n; j ++)
            f[i][j] = max(f[i - 1][j], f[i - 1][j + (1 << (i - 1))]);

    auto query = [&](int l, int r) {
        int s = lg(r - l + 1);
        return max(f[s][l], f[s][r - (1 << s) + 1]);
    };
}
```

1.4 树链剖分

1.4.1 重链剖分

1.4.1.1 点权线段树维护

```
const int N = 1e5 + 10;
vevctor<int> edge[N];
void add_edge(int from, int to) {
    edge[from].emplace_back(to);
    edge[to].emplace_back(from);
}
int vw[N], n;
int fa[N], siz[N], top[N], son[N], dep[N], dfn[N], rnk[N], stamp = 0;
void build(int root) {
    auto dfs1 = [&](auto &&dfs, int from) -> void {
        son[from] = -1;
        siz[from] = 1;
        for (auto to : edge[from])
            if (dep[to] == -1) {
                dep[to] = dep[from] + 1;
                fa[to] = from;
                dfs(dfs,to);
                siz[from] += siz[to];
                if (son[from] == -1 || siz[to] > siz[son[from]])
                    son[from] = to;
            }
    };
    auto dfs2 = [&](auto &&dfs, int from, int root) -> void {
        top[from] = root;
        dfn[from] = ++ stamp;
        rnk[stamp] = from;
        if (son[from] == -1)
            return;
        dfs(dfs, son[from], to);
        for (auto to : edge[from])
            if (to != fa[to] && to != son[from])
                dfs(dfs, to, to);
    };
    memset(dep, -1, sizeof dep);
    dep[from] = stamp = 0;
    dfs1(dfs1,root);
    dfs2(dfs2,root,root);
}

struct Info {
    int val;
    Info(int v = {}) : val(v) {}
    friend Info operator + (const Info &a, const Info &b) {
        return Info{
            a.val + b.val
        };
    }
};
struct Node {
    Info val;
```

```
} seg[N * 4];

void rise(int id) {
    seg[id].val = seg[id * 2].val + seg[id * 2 + 1].val;
}

// l, r 是dfn序
void build(int l, int r, int id) {
    if (l == r) {
        seg[id].val = a[rnk[l]];
    } else {
        int mid = l + ((r - l) >> 1);
        build(l, mid, id * 2);
        build(mid + 1, r, id * 2 + 1);
        rise(id);
    }
}

Info query(int l, int r, int id, int ql, int qr) {
    if (l == ql && r == qr) {
        return seg[id].val;
    } else {
        int mid = l + ((r - l) >> 1);
        if (qr <= mid)
            return query(l, mid, id * 2, ql, qr);
        else if (ql > mid)
            return query(mid + 1, r, id * 2 + 1, ql, qr);
        else
            return query(l, mid, id * 2, ql, mid) + query(mid + 1, r, id * 2 + 1,
mid + 1, qr);
    }
}
```

二、数论

三、图论

3.1 拓扑排序

```
const int N = 1e5 + 10;
vector<int> edge[N];
int ind[N];  // 入度
int n;

bool toposort() {
    vector<int> q;  // 拓扑序列
    q.reverse(n);
    for (int i = 1; i <= n; i ++)
        if (!ind[i])
```

```
            q.emplace_back(i);
    for (int i = 0; i < q.size(); i ++) {
        int from = q[i];
        for (auto to : edge[from])
            if (-- ind[to] == 0)
                q.emplace_back(to);
    }
    if (q.size() != n)
        return false;
    return true;
}
```

3.2 最短路

3.2.1 Floyd

```
const int N = 1e5 + 10;
int f[N][N], n;
void solve() {
    for (int k = 1; k <= n; k ++)
        for (int i = 1; i <= n; i ++)
            for (int j = 1; i <= n; j ++)
                f[i][j] = min(f[i][j], f[i][k] + f[k][j]);
}
```

3.2.2 Dijkstra

```
template <typename T>
constexpr T inf = std::numeric_limits<T>::max() / 2;
const int N = 1e5 + 10;
vector<pair<int,int>> edge[N];
int dis[N];
bool vis[N];

void dijk(int sta) {
    fill(begin(dis), end(dis), inf<int>);
    memset(vis, 0, sizeof vis);
    priority<pair<int,int>,vector<pair<int,int>>,greater<pair<int,int>>> q;
    dis[sta] = 0;
    q.emplace(dis[sta], sta);
    while (!q.empty()) {
        int from = q.top().second;
        q.pop();
        if (vis[from]) continue;
        vis[from] = true;
        for (auto [to, w] : edge[from])
            if (!vis[to] && dis[to] > dis[from] + w) {
                dis[to] = dis[from] + w;
                q.emplace(dis[to], to);
```

```
            }
        }
    }
```

3.2.3 SPFA已经死了

3.2.3.1 SPFA 最短路

请在带有负权边的图中跑SPFA。

```
template <typename T>
constexpr T inf = std::numeric_limits<T>::max() / 2;
const int N = 1e5 + 10;
vector<pair<int,int>> edge[N];
int dis[N], cnt[N];  // 距离，入队次数
bool vis[N];  // 在队列中
int n;
bool spfa(int sta) {
    fill(begin(dis), end(dis), inf<int>);
    memset(cnt, 0, sizeof cnt);
    memset(vis, 0, sizeof vis);
    queue<int> q;
    dis[sta] = 0;
    q.emplace(sta);
    while (!q.empty()) {
        int from = q.front();
        q.pop();
        vis[from] = false;
        for (auto [to, w] : edge[from])
            if (dis[to] > dis[from] + w) {
                dis[to] = dis[from] + w;
                cnt[to] = cnt[from] + 1;
                if (cnt[to] >= n) // 最短路边数
                    return false;  // 单源负环
                if (!vis[to]) {
                    q.emplace(to);
                    vis[to] = true;
                }
            }
    }
    return true;
}
```

3.2.3.2 SPFA 全源判负环

判断图中是否存在负环。

```
template <typename T>
constexpr T inf = std::numeric_limits<T>::max() / 2;
```

```
const int N = 1e5 + 10;
vector<pair<int,int>> edge[N];
int dis[N], cnt[N];  // 距离，入队次数
bool vis[N];  // 在队列中
int n;
bool spfa(int sta) {
    queue<int> q;
    for (int i = 1; i <= n; i ++) {
        dis[i] = cnt[i] = 0;
        vis[i] = true;
        q.emplace(sta);
    }
    while (!q.empty()) {
        int from = q.front();
        q.pop();
        vis[from] = false;
        for (auto [to, w] : edge[from])
            if (dis[to] > dis[from] + w) {
                dis[to] = dis[from] + w;
                cnt[to] = cnt[from] + 1;
                if (cnt[to] >= n)
                    return false;
                if (!vis[to]) {
                    q.emplace(to);
                    vis[to] = true;
                }
            }
    }
    return true;
}
```

3.3 网络流

3.3.1 最大流

3.3.1.1 Dinic

```
template <typename Capacity>
struct MaxFlow_Dinic {
    using Capacity_Type = Capacity;
    static constexpr Capacity __INF = std::numeric_limits<Capacity>::max() / 2;
    struct Edge {
        int to, next;
        Capacity cap;
        Edge(const int &_to = {}, const Capacity &_cap = {}, const int &_next =
{-1}) : to(_to), next(_next), cap(_cap) {}
    };
    std::vector<Edge> edge;
    std::vector<int> head, cur, lv;
    int last_edge_pos;
    int S, T;  // Source and Sink
```

```
    MaxFlow_Dinic(int _n = {}, int _m = {}, int _S = {}, int _T = {}) : S(_S),
T(_T) {
        assign(_n, _m);
    }
    void assign(int _n, int _m) {
        last_edge_pos = -1;
        head.assign(_n, -1);
        cur.resize(_n);
        lv.resize(_n);
        edge.clear();
        edge.reserve(_m);
    }
    void add_edge(int from, int to, const Capacity &cap) {
        edge.emplace_back(to, cap, head[from]); head[from] = ++ last_edge_pos;
    }
    void add_edges(int from, int to, const Capacity &cap) {
        add_edge(from, to, cap);
        add_edge(to, from, Capacity{0});
    }
    auto maxflow(int s, int t) {
        auto bfs = [&] {
            std::copy(std::begin(head), std::end(head), std::begin(cur));
            std::fill(std::begin(lv), std::end(lv), -1);
            std::queue<int> q;
            q.emplace(s);
            lv[s] = 0;
            while (!q.empty()) {
                int from = q.front();
                q.pop();
                for (int ed = head[from]; ed != -1; ed = edge[ed].next) {
                    if (edge[ed].cap > 0 && lv[edge[ed].to] == -1) {
                        lv[edge[ed].to] = lv[from] + 1;
                        q.emplace(edge[ed].to);
                    }
                }
            }
            return lv[t] != -1;
        };
        auto dfs = [&](auto &&dfs, int from, Capacity flow) -> Capacity {
            if (from == t) return flow;
            auto lesf = flow;
            for (int ed = cur[from]; ed != -1 && lesf; ed = edge[ed].next) {
                cur[from] = ed;
                if (edge[ed].cap > 0 && lv[edge[ed].to] == lv[from] + 1) {
                    auto ret = dfs(dfs, edge[ed].to, std::min<Capacity>(lesf,
edge[ed].cap));
                    lesf -= ret;
                    edge[ed].cap -= ret;
                    edge[ed ^ 1].cap += ret;
                }
            }
            return flow - lesf;
        };
        Capacity flow = 0;
```

```
        while (bfs()) flow += dfs(dfs, s, __INF);
        return flow;
    }
    auto maxflow() { return maxflow(S, T); }
    auto& operator [] (int idx) { return edge[idx]; }
    const auto& operator [] (int idx) const { return edge[idx]; }
    auto& begin() { return edge.begin(); }
    const auto& cbegin() const { return edge.cbegin(); }
    auto& end() { return edge.end(); }
    const auto& cend() const { return edge.cend(); }
    auto& rbegin() { return edge.rbegin(); }
    const auto& crbegin() const { return edge.crbegin(); }
    auto& rend() { return edge.rend(); }
    const auto& crend() const { return edge.crend(); }
};
```

四、杂项