Data Structrues

BIT(Fenwick Tree)

```
struct BIT
{
    ll tree[N];//不要忘记初始化N
    int n;
    BIT()
    {
        memset(tree,0,sizeof(tree));
    }
    //注意树状数组的下标一定从1开始
    int lowbit(int i)
    {
        return i & -i;
    }

    void add(int i,int v)
    {
        while(i <= n)
        {
            //将起始i二进制位不断加其本身最右边的1更新树状数组
            tree[i] += v;
            i += lowbit(i);
        }
    }
    //返回1-i范围的累加和
    ll sum(int i)
    {
        ll ans = 0;
        while(i>0)
        {
            ans += tree[i];//不断加减去二进制最右边1的数
            i -= lowbit(i);//不断减去二进制位最右边的1
        }
        return ans;
    }
    //返回l-r范围的累加和
    ll rangesum(int l,int r)
    {
        return sum(r) - sum(l-1);
    }

    // 查找满足前缀和sum(x) >= k的最小下标x
    int select(ll k)
    {
        int x = 0;
        ll cur = 0;
        int max_bit = 0;
        for (int i = N-1; i > 0; i >>= 1) max_bit++;
        for (int i = 1 << max_bit; i > 0; i >>= 1)
```

```
        {
            if (x + i <= n && cur + tree[x + i] < k)  // 修改：严格小于k
            {
                cur += tree[x + i];
                x += i;
            }
        }
        return x + 1;  // 返回x+1是因为树状数组下标从1开始
    }
};
```

Sparse Table & RMQ

```
const int N = 1e5+10;
int arr[N];
int near[N];//near[i]表示第一个小于等于i的二的次幂
void solve()
{
    int n,m;
    cin >> n >> m;
    int maxpow = log2(N)+1;
    for(int i=1;i<=n;i++)
    {
        cin >> arr[i];
    }
    int stmax[n+1][maxpow];//表示从第i个元素开始，往后2^j的范围中的最大值
    int stmin[n+1][maxpow];//表示从第i个元素开始，往后2^j的范围中的最小值
    near[0] = -1;
    for(int i=1;i<=n;i++)
    {
        near[i] = near[i >> 1] + 1;
        stmax[i][0] = arr[i];
        stmin[i][0] = arr[i];
    }
    for(int p=1;p<=near[n];p++)//按列填入数据
    {
        for(int i=1;i+(1<<p)-1<=n;i++)//不断计算第i点向后延申2^p长度所占的区间最值，因为是长度所以要-1
        {
            stmax[i][p] = max(stmax[i][p-1],stmax[i+(1<<(p-1))][p-1]);//两段区间对比出最值
            stmin[i][p] = min(stmin[i][p-1],stmin[i+(1<<(p-1))][p-1]);
        }
    }
    //查询
    while(m--)
    {
        int l,r;
        cin >> l >> r;
        int len = r - l + 1;
        int pow = near[len];
        int resmax = max(stmax[l][pow],stmax[r-(1<<pow)+1][pow]);
        int resmin = min(stmin[l][pow],stmin[r-(1<<pow)+1][pow]);
```

```
        cout << resmax - resmin * 1LL << endl;
    }
}
```

Trie Tree

```
struct trieNode
{
    int leaf[26];
    bool have;
    trieNode()
    {
        memset(leaf,0,sizeof(leaf));
        have = false;
    }
}trie[N];
ll num = 0;
void insert(char *s)
{
    //root u -> leaf v
    int v,len = strlen(s);
    int u = 0;
    for(int i=0;i<len;i++)
    {
        v = s[i] - 'a';
        if(trie[u].leaf[v]==0)
        {
            trie[u].leaf[v] = ++num;
        }
        u = trie[u].leaf[v];
    }
    trie[u].have = true;
}
int find(char *s)
{
    int u = 0,v,len = strlen(s);
    for(int i=0;i<len;i++)
    {
        v = s[i] - 'a';
        if(trie[u].leaf[v]==0) return 1;
        u = trie[u].leaf[v];
    }
    return trie[u].have;
}
```

Cmaxmin

```
// 维护最大值、次大值、最小值、次小值的数据结构
struct CM
{
    int mx1, mx2, mn1, mn2;  // 最大值、次大值、最小值、次小值

    // 确保mx1 >= mx2
    void fix_mx(){
```

```cpp
        if(mx1 < mx2){
            swap(mx1, mx2);
        }
    }

    // 确保mn1 <= mn2
    void fix_mn(){
        if(mn1 > mn2){
            swap(mn1, mn2);
        }
    }

    // 构造函数，初始化两个值
    CM(int a, int b){
        mx1 = mn1 = a;
        mx2 = mn2 = b;
        fix_mx();
        fix_mn();
    }

    // 添加新值并更新四个维护值
    void add(int x){
        mx2 = max(mx2, x);
        mn2 = min(mn2, x);
        fix_mx();
        fix_mn();
    }

    // 获取排除指定值后的区间长度
    // 如果x等于最大值/最小值，则使用次大值/次小值计算
    int get_seg(int x){
        pair<int, int> res = {mn1, mx1};
        if(x == mn1) res.first = mn2;
        if(x == mx1) res.second = mx2;
        return res.second - res.first + 1;
    }
};
void solve()
{
    int n;
    cin >> n;
    vector<pii> point(n+1);
    for(int i=1;i<=n;i++)
    {
        cin >> point[i].first >> point[i].second;
    }
    if(n<=2)
    {
        cout << n << endl;
        return;
    }
    CM xc(point[1].first,point[2].first);
    CM yc(point[1].second,point[2].second);
    for(int i=3;i<=n;i++)
    {
```

```
        xc.add(point[i].first);
        yc.add(point[i].second);
    }
    int ans = xc.get_seg(-1) * yc.get_seg(-1);
    for(int i=1;i<=n;i++)
    {
        int tempx = xc.get_seg(point[i].first);
        int tempy = yc.get_seg(point[i].second);
        if(tempx * tempy == n - 1)
        {
            ans = min(ans,min(tempx*(tempy+1),tempy*(tempx+1)));
        }
        else
        ans = min(ans,tempx*tempy);
    }
    cout << ans << endl;
}
```

DSU

```
// 支持回滚操作的并查集
struct DSU
{
    vector<int> fa,Size;
    vector<array<int, 2>> his;  // 历史记录栈，用于撤销操作
    DSU(int n) : fa(n+1),Size(n+1,1)
    {
        iota(fa.begin(),fa.end(),0);
    }
    int find(int x)
    {
        //注意：路径压缩是在find()函数内进行，所以不可以直接用fa数组比较!
        while (x != fa[x]) x = fa[x] = fa[fa[x]];
        return x;
    }
    bool same(int x, int y)
    {
        return find(x) == find(y);
    }
    bool merge(int x, int y)
    {
        x = find(x);
        y = find(y);
        if (x == y) return false;
        if(Size[x] < Size[y]) swap(x, y);
        his.push_back({x, y});  // 记录合并操作，用于后续撤销
        Size[x] += Size[y];
        fa[y] = x;
        return true;
    }
    // 返回当前进行的合并操作次数（时间戳）
    int time() {
        return his.size();
```

```cpp
    }
    // 撤销操作，将并查集状态回滚到指定时间戳tm
    void revert(int tm)
    {
        while (his.size() > tm)  // 从栈顶依次弹出操作直到时间戳tm
        {
            auto [x, y] = his.back();  // 获取最后一次合并的两个集合
            his.pop_back();  // 弹出操作记录
            fa[y] = y;  // 恢复y的父节点为自身
            Size[x] -= Size[y];  // 恢复集合x的大小
        }
    }
    int size(int x) // 返回当前x节点所在集合的大小
    {
        return Size[find(x)];
    }
};
// 二分图可撤销并查集
// 模板实现了一个支持撤销操作的并查集，主要用于处理动态图的连通性和二分图相关问题。关键操作包含查找元素所属集合、合并集合、记录操作历史以及回滚操作等
struct DSU {
    // 用于记录历史操作，方便撤销操作。每个元素是一个 pair,
    // 第一个元素是对某个变量的引用，第二个元素是该变量原来的值
    vector<pair<int &, int>> his;
    int n;
    vector<int> f;  // 并查集的父节点数组，f[i] 表示元素 i 的父节点，若 f[i] < 0，说明 i 是根节点，且 -f[i] 表示该集合的大小
    vector<int> g;  // 用于记录与二分图相关的异或信息，辅助判断二分图性质
    vector<int> bip;  // 标记每个连通分量是否为二分图，1 表示是二分图，0 表示不是
    DSU(int n_) : n(n_), f(n, -1), g(n), bip(n, 1) {}
    // 查找元素 x 所在集合的代表元素（根节点），并返回该代表元素和从 x 到代表元素的异或值
    pair<int, int> find(int x) {
        // 如果 x 是根节点（即 f[x] 为负数）
        if (f[x] < 0) {
            return {x, 0};
        }
        auto [u, v] = find(f[x]);
        return {u, v ^ g[x]};
    }
    // 记录变量的修改历史，方便后续回滚操作
    void set(int &a, int b) {
        // 将变量 a 的引用和其原始值存入历史记录中
        his.emplace_back(a, a);
        a = b;
    }
    // 合并元素 a 和 b 所在的集合，并更新二分图的数量
    void merge(int a, int b, int &ans) {
        auto [u, xa] = find(a);
        auto [v, xb] = find(b);
        int w = xa ^ xb ^ 1;          // 计算合并时的异或值
        if (u == v)      // 如果该集合是二分图且异或值为 1，说明合并后不再是二分图
        {
            if (bip[u] && w) {
                set(bip[u], 0);                  // 标记该集合不是二分图
                ans--;
```

```
            }
            return;
        }
        if (f[u] > f[v]) {
            swap(u, v);
        }
        ans -= bip[u];          // 减去 u 集合的二分图数量
        ans -= bip[v];         // 减去 v 集合的二分图数量
        set(bip[u], bip[u] && bip[v]);        // 更新 u 集合的二分图标记
        set(f[u], f[u] + f[v]);        // 更新 u 集合的秩
        set(f[v], u);
        set(g[v], w);
        ans += bip[u];         // 加上合并后 u 集合的二分图数量
    }
    // 返回当前历史操作的数量，即时间戳
    int timeStamp() {
        return his.size();
    }
    // 回滚到指定的时间戳 t
    void rollback(int t) {
        while (his.size() > t) {
            auto [x, y] = his.back();
            x = y;
            his.pop_back();
        }
    }
};
```

Monotonic Structures

```
// Monotonic Stack
struct node {
    ll val;
    int index;
    int prev;  // 左边第一个大于该元素的下标
    int next;  // 右边第一个大于该元素的下标
    node()
    {
        prev = 0;
        next = 0;
    }
};
node arr[N];
stack<node> stk;
signed main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    cout.tie(nullptr);
    int n;
    cin >> n;
    for (int i = 1; i <= n; i++)
    {
        cin >> arr[i].val;
        arr[i].index = i;
    }
```

```
    // 从左到右遍历，找到每个元素右边第一个大于它的元素的下标
    for (int i = 1; i <= n; i++) {
        while (!stk.empty() && arr[i].val > stk.top().val)
        {
            arr[stk.top().index].next = i;
            stk.pop();
        }
        stk.push(arr[i]);
    }
    // 清空栈
    while (!stk.empty())
    {
        stk.pop();
    }
    // 从右到左遍历，找到每个元素左边第一个大于它的元素的下标
    for (int i = n; i >= 1; i--) {
        while (!stk.empty() && arr[i].val > stk.top().val)
        {
            arr[stk.top().index].prev = i;
            stk.pop();
        }
        stk.push(arr[i]);
    }
    for (int i = 1; i <= n; ++i)
    {
        cout  << arr[i].val << " " << arr[i].prev << " " << arr[i].next << endl;
    }
    return 0;
}
// Monotonic Queue
struct node
{
    int x;// 水滴落点
    int sec;// 水滴下落时间
}drip[N];

deque<int> qmax;
deque<int> qmin;

bool cmp(node a,node b) { return a.x < b.x; }
void solve()
{
    cin >> n >> D;
    for(int i=0;i<n;i++)
    {
        cin >> drip[i].x >> drip[i].sec;
    }
    sort(drip,drip+n,cmp);
    int ans = LLONG_MAX;

    int tail = drip[0].x;//尾指针指向离散化后的窗口左边界
    int index = 0;//指针指向未离散化时的窗口左边界
    int head;//头指针指向离散化后的窗口右边界
    //i为未离散化的窗口右边界
    for(int i=0;i<n;i++)
```

```
    {
        head = drip[i].x;
        //求最大值的单调递减队列
        while(!qmax.empty()&&drip[i].sec >= qmax.back()) qmax.pop_back();
        qmax.push_back(drip[i].sec);
        //求最小值的单调递增队列
        while(!qmin.empty()&&drip[i].sec <= qmin.back()) qmin.pop_back();
        qmin.push_back(drip[i].sec);
        while(qmax.front()-qmin.front() >= D)
        {
            ans = min(ans,head-tail);
            if(drip[index].sec==qmax.front()) qmax.pop_front();
            if(drip[index].sec==qmin.front()) qmin.pop_front();
            index++;
            tail = drip[index].x;
        }
    }
    if(ans==LLONG_MAX)
    {
        cout << -1;
    }
    else
    cout << ans;
}
```

LCA(Sparse Table Version)

```
// a节点到b节点的距离 = a节点到根节点的距离 + b节点到根节点的距离 - （头节点到LCA(a,b)节点的距离）* 2
const int N = 5e5 + 100;
const int M = 0;
int st[N][32]; // st[i][p]表示第i个节点往上走2^p步到达哪个节点
int depth[N];
int dis[N]; // 记录从根节点到每个节点的边权之和
int cnt = 0;
int head[N];
int maxpow;
int n,m,root;
struct node
{
    int to, next;
    int weight; // 边权
} edge[N << 1];
void add(int u, int v, int w)
{
    cnt++;
    edge[cnt].to = v;
    edge[cnt].weight = w;
    edge[cnt].next = head[u];
    head[u] = cnt;
}
// DFS 建立树上 ST 表，并记录从根节点到每个节点的边权之和
void dfs(int i, int fa)
```

```
{
    depth[i] = (fa == -1) ? 0 : (depth[fa] + 1);
    st[i][0] = fa;
    for (int p = 1; p <= maxpow; p++)
    {
        st[i][p] = (st[i][p - 1] == -1) ? -1 : st[st[i][p - 1]][p - 1];
    }
    for (int j = head[i]; j; j = edge[j].next)
    {
        int v = edge[j].to;
        if (v != fa)
        {
            dis[v] = dis[i] + edge[j].weight;
            dfs(v, i);
        }
    }
}
int LCA(int x, int y)
{
    if (depth[x] < depth[y]) swap(x, y);
    // 使得 x 和 y 来到同一层
    while (depth[x] > depth[y]) x = st[x][(int)log2(depth[x] - depth[y])];
    // 在同一层相遇说明 x, y 互为子孙和祖先，此时祖先是 x，返回 x。
    if (x == y) return x;
    // 在同一层后一起向上倍增
    for (int i = log2(depth[x]); i >= 0; i--)
    {
        // 当两个节点的落脚点不同时，持续上增
        if (st[x][i] != st[y][i])
        {
            x = st[x][i];
            y = st[y][i];
        }
    }
    // 最后得到 x, y 的相同的父节点，此时再往上走 2^0 步到达它们的最近公共祖先。
    return st[x][0];
}
// 计算两点间的边权距离
int distance(int x, int y)
{
    int lca = LCA(x, y);
    return dis[x] + dis[y] - 2 * dis[lca];
}
void solve()
{
    cin >> n >> m >> root;
    maxpow = log2(n) + 1;
    memset(st, -1, sizeof st); // 初始化 st 表元素为 -1
    for (int i = 1; i <= n - 1; i++) {
        int u, v, w;
        cin >> u >> v >> w;
        add(u, v, w);
        add(v, u, w);
    }
    dfs(root, -1);
```

```
    for (int i = 1; i <= m; i++) {
        int x, y;
        cin >> x >> y;
        cout << distance(x, y) << endl;
    }
}
```

Get Kth Ancestor On Tree Node

```
const int N = 5e4 + 100;
const int M = 0;
int st[N][17]; // st[i][p]表示第i个节点往上走2^p步到达哪个节点
int depth[N];
int cnt = 0;
int head[N];
int maxpow;
struct node
{
    int from, to, next;
} edge[N];

void add(int u, int v)
{
    cnt++;
    edge[cnt].from = u;
    edge[cnt].to = v;
    edge[cnt].next = head[u];
    head[u] = cnt;
}
int n;

// 预处理所有可能幂次
void dfs(int i, int fa)
{
    depth[i] = (fa == -1) ? 0 : (depth[fa] + 1);
    st[i][0] = fa;

    // 处理所有预定义maxpow层
    for (int p = 1; p <= maxpow; p++)
    {
        st[i][p] = (st[i][p-1] == -1) ? -1 : st[st[i][p-1]][p-1];
    }

    for (int j = head[i]; j; j = edge[j].next)
    {
        int v = edge[j].to;
        dfs(v, i);
    }
}

int getAncestor(int i, int k)
{
```

```
    if (k == 0) return i;  // 直接处理k=0
    if (depth[i] < k) return -1;

    int s = depth[i] - k;
    for (int p = maxpow; p >= 0; p--)
    {
        if (depth[i] - (1 << p) >= s)
        {
            i = st[i][p];
            if (i == -1) break;
        }
    }
    return i;
}

void solve() {
    cin >> n;
    maxpow = log2(n) + 1;
    memset(st,-1,sizeof st);
    vector<bool> hasParent(n + 1,false);
    for (int i = 1; i <= n - 1; i++)
    {
        int fa, to;
        cin >> fa >> to;
        add(fa,to);
        hasParent[to] = true;
    }

    int root = find(hasParent.begin()+1, hasParent.end(), false) -
hasParent.begin();
    dfs(root, -1);

    int q;
    cin >> q;
    while (q--)
    {
        int node, k;
        cin >> node >> k;
        cout << getAncestor(node, k) << endl;
    }
}
```

SegmentTree

```
#include <bits/stdc++.h>
#define int long long
#define endl '\n'
#define Single
#define ls(p) (p << 1)       // 左子节点索引
#define rs(p) (p << 1 | 1)   // 右子节点索引
using namespace std;
const int N = 1e6+10;      // 最大数据规模
int n,m;
/*
    线段树常见方法一览
```

```
    void push_up(..){} : 根据子范围的查询信息，把父范围的查询信息更新正确
    void push_down(..){} : 父范围的懒信息，往下下发一层，并给左范围，右范围，然后父范围的懒信息清空
    void apply(..){} : 一段范围的整体被任务整体全覆盖或是父范围下发的懒信息，该如何处理，可以理解为一旦apply了此操作，该线段的值该如何去处理
    void build(..){} : 建树
    void update(..){} : 范围上的更新任务
    void query(..){} : 范围上的查询任务
*/
struct SegTree
{
    private:
        struct Node
        {
            int l, r;
            int sum, max, min, gcd, xor_sum, lcm;
            int add, mul;
            int set;
            bool change;
            // 仅需初始化初始状态所有线段信息都相同的信息(懒更新信息)，其余信息会在build过程中初始化，防止卡常
            Node() : change(false), add(0), mul(1), set(0) {}
            inline int len()
            {
                return r - l + 1;
            }
        } tree[N << 2];
        vector<int> arr;

        void push_up(int p)
        {
            tree[p].sum = tree[ls(p)].sum + tree[rs(p)].sum;
            tree[p].max = max(tree[ls(p)].max, tree[rs(p)].max);
            tree[p].min = min(tree[ls(p)].min, tree[rs(p)].min);
            tree[p].gcd = __gcd(tree[ls(p)].gcd, tree[rs(p)].gcd);
            tree[p].xor_sum = tree[ls(p)].xor_sum ^ tree[rs(p)].xor_sum;
            tree[p].lcm = (tree[ls(p)].lcm * tree[rs(p)].lcm) /
__gcd(tree[ls(p)].lcm, tree[rs(p)].lcm);
        }

        void apply_set(int p, int val)
        {
            tree[p].sum = val * (tree[p].r - tree[p].l + 1);
            tree[p].max = tree[p].min = val;
            tree[p].gcd = val;
            tree[p].xor_sum = 0;
            tree[p].lcm = val;
            tree[p].set = val;
            tree[p].change = true;
            tree[p].add = 0;
            tree[p].mul = 1;
        }

        void apply_add(int p, int val)
        {
```

```
        tree[p].sum += val * (tree[p].r - tree[p].l + 1);
        tree[p].max += val;
        tree[p].min += val;
        tree[p].gcd += val;
        tree[p].xor_sum ^= (val * (tree[p].r - tree[p].l + 1));
        tree[p].lcm += val;
        tree[p].add += val;
    }

    void apply_mul(int p, int val)
    {
        tree[p].sum *= val;
        tree[p].max *= val;
        tree[p].min *= val;
        tree[p].gcd *= val;
        tree[p].xor_sum *= val;
        tree[p].lcm *= val;  // Multiply lcm
        tree[p].mul *= val;
        tree[p].add *= val;
    }

    void push_down(int p)
    {
        if (tree[p].change)
        {
            apply_set(ls(p), tree[p].set);
            apply_set(rs(p), tree[p].set);
            tree[p].change = false;
        }
        if (tree[p].mul != 1)
        {
            apply_mul(ls(p), tree[p].mul);
            apply_mul(rs(p), tree[p].mul);
            tree[p].mul = 1;
        }
        if (tree[p].add != 0)
        {
            apply_add(ls(p), tree[p].add);
            apply_add(rs(p), tree[p].add);
            tree[p].add = 0;
        }
    }

public:
    SegTree(const vector<int>& _arr) : arr(_arr)
    {
        build(1, 1, (int)arr.size()-1);
      //build(1, 0, (int)arr.size()-1);
    }
    void build(int p, int l, int r)
    {
        tree[p].l = l;
        tree[p].r = r;
        if (l == r)
        {
```

```cpp
            tree[p].sum = tree[p].max = tree[p].min = arr[l];
            tree[p].gcd = arr[l];
            tree[p].xor_sum = arr[l];
            tree[p].lcm = arr[l];
            return;
        }
        int mid = (l + r) >> 1;
        build(ls(p), l, mid);
        build(rs(p), mid+1, r);
        push_up(p);
    }

    void updateSet(int p, int ul, int ur, int val)
    {
        if (ul <= tree[p].l && tree[p].r <= ur)
        {
            apply_set(p, val);
            return;
        }
        push_down(p);
        int mid = (tree[p].l + tree[p].r) >> 1;
        if (ul <= mid) updateSet(ls(p), ul, ur, val);
        if (ur > mid) updateSet(rs(p), ul, ur, val);
        push_up(p);
    }

    void updateAdd(int p, int ul, int ur, int val)
    {
        if (ul <= tree[p].l && tree[p].r <= ur)
        {
            apply_add(p, val);
            return;
        }
        push_down(p);
        int mid = (tree[p].l + tree[p].r) >> 1;
        if (ul <= mid) updateAdd(ls(p), ul, ur, val);
        if (ur > mid) updateAdd(rs(p), ul, ur, val);
        push_up(p);
    }

    void updateMul(int p, int ul, int ur, int val)
    {
        if (ul <= tree[p].l && tree[p].r <= ur)
        {
            apply_mul(p, val);
            return;
        }
        push_down(p);
        int mid = (tree[p].l + tree[p].r) >> 1;
        if (ul <= mid) updateMul(ls(p), ul, ur, val);
        if (ur > mid) updateMul(rs(p), ul, ur, val);
        push_up(p);
    }
    // 对区间开平方的操作无法根据懒更新去维护区间和与区间最大值，所以只能进行暴力下传到叶节点进行修改，但下传过程可以剪枝
```

```
        // 时间复杂度O(6 * N * log(N)) 6为数据最大值1e13所能开平方的最大次数，因为是暴力下传到叶节点，所以需乘上树的高度logN
        void updateSqrt(int p, int ul, int ur)
        {
            if (tree[p].l == tree[p].r)
            {
                arr[tree[p].l] = sqrt(arr[tree[p].l]);
                tree[p].sum = arr[tree[p].l];
                tree[p].max = arr[tree[p].l];
                return;
            }
            int mid = (tree[p].l + tree[p].r) >> 1;
            //剪枝：如果区间最大值已经为1，则此区间无需再进行开平方操作.
            if (ul <= mid && tree[ls(p)].max > 1)
            {
                updateSqrt(ls(p), ul, ur);
            }
            if (ur > mid && tree[rs(p)].max > 1)
            {
                updateSqrt(rs(p), ul, ur);
            }
            push_up(p);
        }
        //对区间取模数的操作无法根据懒更新去维护区间和与区间最大值，所以只能进行暴力下传到叶节点进行修改，但下传过程可以剪枝
        void updateMod(int p, int ul, int ur,int mod)
        {
            //剪枝：如果区间最大值小于mod，则此区间无需再进行取模操作.
            if(mod > tree[p].max) return;
            if (tree[p].l == tree[p].r)
            {
                arr[tree[p].l] %= mod;
                tree[p].sum = arr[tree[p].l];
                tree[p].max = arr[tree[p].l];
                return;
            }
            int mid = (tree[p].l + tree[p].r) >> 1;
            if (ul <= mid)
            {
                updateMod(ls(p), ul, ur, mod);
            }
            if (ur > mid)
            {
                updateMod(rs(p), ul, ur, mod);
            }
            push_up(p);
        }
        //修改单个元素的值O(logn)
        void updateval(int p, int ul, int ur, int val)
        {
            if(tree[p].l == tree[p].r)
            {
                arr[tree[p].l] = val;
                tree[p].sum = arr[tree[p].l];
                tree[p].max = arr[tree[p].l];
```

```
            return;
        }
        int mid = (tree[p].l + tree[p].r) >> 1;
        if(ul <= mid) updateval(ls(p), ul, ur, val);
        if(ur > mid) updateval(rs(p), ul, ur, val);
        push_up(p);
    }
    int querySum(int p, int ql, int qr)
    {
        if (ql <= tree[p].l && tree[p].r <= qr) return tree[p].sum;
        push_down(p);
        int mid = (tree[p].l + tree[p].r) >> 1, res = 0;
        if (ql <= mid) res += querySum(ls(p), ql, qr);
        if (qr > mid) res += querySum(rs(p), ql, qr);
        return res;
    }

    int queryMax(int p, int ql, int qr)
    {
        if (ql <= tree[p].l && tree[p].r <= qr) return tree[p].max;
        push_down(p);
        int mid = (tree[p].l + tree[p].r) >> 1, res = LLONG_MIN;
        if (ql <= mid) res = max(res, queryMax(ls(p), ql, qr));
        if (qr > mid) res = max(res, queryMax(rs(p), ql, qr));
        return res;
    }

    int queryMin(int p, int ql, int qr)
    {
        if (ql <= tree[p].l && tree[p].r <= qr) return tree[p].min;
        push_down(p);
        int mid = (tree[p].l + tree[p].r) >> 1, res = LLONG_MAX;
        if (ql <= mid) res = min(res, queryMin(ls(p), ql, qr));
        if (qr > mid) res = min(res, queryMin(rs(p), ql, qr));
        return res;
    }

    int queryGCD(int p, int ql, int qr)
    {
        if (ql <= tree[p].l && tree[p].r <= qr) return tree[p].gcd;
        push_down(p);
        int mid = (tree[p].l + tree[p].r) >> 1, res = 0;
        if (ql <= mid) res = __gcd(res, queryGCD(ls(p), ql, qr));
        if (qr > mid) res = __gcd(res, queryGCD(rs(p), ql, qr));
        return res;
    }

    int queryXOR(int p, int ql, int qr)
    {
        if (ql <= tree[p].l && tree[p].r <= qr) return tree[p].xor_sum;
        push_down(p);
        int mid = (tree[p].l + tree[p].r) >> 1, res = 0;
        if (ql <= mid) res ^= queryXOR(ls(p), ql, qr);
        if (qr > mid) res ^= queryXOR(rs(p), ql, qr);
        return res;
```

```
        }

        int queryLCM(int p, int ql, int qr)
        {
            if (ql <= tree[p].l && tree[p].r <= qr) return tree[p].lcm;
            push_down(p);
            int mid = (tree[p].l + tree[p].r) >> 1, res = 1;
            if (ql <= mid) res = (res * queryLCM(ls(p), ql, qr)) / __gcd(res,
queryLCM(ls(p), ql, qr));
            if (qr > mid) res = (res * queryLCM(rs(p), ql, qr)) / __gcd(res,
queryLCM(rs(p), ql, qr));
            return res;
        }
};
void solve()
{
    int n,m;
    cin >> n >> m;
    vector<int> arr(n+1);
    for(int i=1;i<=n;i++) cin >> arr[i];
    SegTree st(arr);
    while(m--)
    {
        int op;
        cin >> op;
        if(op==1)
        {
            int l,r,x;
            cin >> l >> r >> x;
            st.updateSet(1,l,r,x);
        }
        if(op==2)
        {
            int l,r,x;
            cin >> l >> r >> x;
            st.updateAdd(1,l,r,x);
        }
        if(op==3)
        {
            int l,r;
            cin >> l >> r;
            cout << st.queryMax(1,l,r) << endl;
        }
    }
}
```

Extra Segment Tree Info Collection

$二进制反转数字(Reverse)$

```
        void apply_reverse(int p)
        {
            tree[p].sum = tree[p].len() - tree[p].sum; // 例: 求1的和
            ... ...
            tree[p].reverse = !tree[p].reverse; // 取反, 并非false
```

```
        }
        void push_down(int p)
        {
            if(tree[p].reverse)
            {
                apply_reverse(ls(p));
                apply_reverse(rs(p));
                tree[p].reverse = false;
            }
        }
```

$线段树的信息合并维护$

```
        struct Node
        {
            //例：求解区间[l,r]中的最长1字串长度
            int l,r;
            int sum; // 有多少1
            int set; // 重置为哪个数字
            int pre0,suf0; // 前缀0和后缀0的长度
            int pre1,suf1; // 前缀1和后缀1的长度
            int len1; // 1的最长连续字串长度
            int len0; // 0的最长连续字串长度
            bool change;
            bool reverse;
            Node() : l(0), r(0), sum(0), set(0), change(false), reverse(false),
pre0(0), suf0(0), pre1(0), suf1(0), len1(0), len0(0) {}
            inline int len()
            {
                return r - l + 1;
            }
        } tree[N << 2];
        void push_up(int p)
        {
            tree[p].sum = tree[ls(p)].sum + tree[rs(p)].sum;
            tree[p].len1 =
max({tree[ls(p)].len1,tree[rs(p)].len1,tree[ls(p)].suf1 + tree[rs(p)].pre1});
            tree[p].len0 =
max({tree[ls(p)].len0,tree[rs(p)].len0,tree[ls(p)].suf0 + tree[rs(p)].pre0});
            tree[p].pre1 = tree[ls(p)].pre1 + (tree[ls(p)].pre1 ==
tree[ls(p)].len()) * tree[rs(p)].pre1;
            tree[p].pre0 = tree[ls(p)].pre0 + (tree[ls(p)].pre0 ==
tree[ls(p)].len()) * tree[rs(p)].pre0;
            tree[p].suf1 = tree[rs(p)].suf1 + (tree[rs(p)].suf1 ==
tree[rs(p)].len()) * tree[ls(p)].suf1;
            tree[p].suf0 = tree[rs(p)].suf0 + (tree[rs(p)].suf0 ==
tree[rs(p)].len()) * tree[ls(p)].suf0;
        }
        int queryLen1(int p,int ql,int qr)
        {
            if(ql <= tree[p].l and tree[p].r <= qr) return tree[p].len1;

            push_down(p);
            int mid = (tree[p].l + tree[p].r) >> 1;
           // 注意此时为qr <= mid , ql > mid 与正常线段树相反
```

```
        // 如果mid不把当前线段分割
        if(qr <= mid) return queryLen1(ls(p),ql,qr);
        if(ql > mid) return queryLen1(rs(p),ql,qr);
        // 如果mid会分割当前线段，则需要结合前后线段信息更新
        // 获得左右线段各自的最长字串长度
        int left = queryLen1(ls(p),ql,qr);
        int right = queryLen1(rs(p),ql,qr);
       // 维护合并两段中间的信息
        int leftsuf = min(tree[ls(p)].suf1,mid - ql + 1);
        int rightpre = min(tree[rs(p)].pre1,qr - (mid + 1) + 1);
       // 三者之间的最大值即为答案
        return max({left,right,leftsuf+rightpre});
    }
```

$开点线段树的应用$

适用于需要维护范围很大（空间约为 操作数量 * 树高 * 2），但是操作数不大的情况，

可以支持很大的范围，但是一开始不为每个范围分配空间，当真的需要开辟左侧，右侧的空间时，再临时申请，父亲范围的空间编号i，利用cnt自增给左右两侧申请的空间，记录在left[i] 和 right[i] 里，

除此之外与普通线段树再无区别。

```
// 开点线段树 实现区间求和区间加和
// 范围1~1e9，树高约等于30，查询次数1000
// 故其空间占用大约为1000 * 30 * 2 = 60000
// 适度开大即可
const int N = 80001;
int cnt; // 下标记录已开辟节点
struct DynamicSegTree
{
    private:
        struct Node
        {
            int l,r; // 左右孩子索引（0表示不存在）
            int tl,tr; // 当前节点覆盖的区间[tl,tr]
            int sum; // 区间和
            int add; // 懒更新加法
            Node() : l(0),r(0),tl(0),tr(0),sum(0),add(0) {}
            inline int len()
            {
                return tr - tl + 1;
            }
        }tree[N]; // 无需*4

        void push_up(int p)
        {
            tree[p].sum = 0;
            if (tree[p].l) tree[p].sum += tree[tree[p].l].sum;
            if (tree[p].r) tree[p].sum += tree[tree[p].r].sum;
        }

        void apply_add(int p,int val)
        {
            tree[p].sum += val * tree[p].len();
            tree[p].add += val;
```

```
    }

    void push_down(int p)
    {
        if(tree[p].add != 0)
        {
            int tl = tree[p].tl;
            int tr = tree[p].tr;
            int mid = (tl + tr) >> 1;
            // 懒更新任务下发，则左右两侧空间需要准备好
            // 动态创建左孩子
            if(tree[p].l == 0)
            {
                tree[p].l = ++cnt;
                tree[tree[p].l].tl = tl;
                tree[tree[p].l].tr = mid;
            }
            // 动态创建右孩子
            if(tree[p].r == 0)
            {
                tree[p].r = ++cnt;
                tree[tree[p].r].tl = mid + 1;
                tree[tree[p].r].tr = tr;
            }
            // 下发懒标记
            apply_add(tree[p].l,tree[p].add);
            apply_add(tree[p].r,tree[p].add);
            // 清除当前节点懒标记
            tree[p].add = 0;
        }
    }

public:
    DynamicSegTree(int l,int r)
    {
        cnt = 1;
        tree[1].tl = l;
        tree[1].tr = r;
    }

    void updateadd(int p,int ul,int ur,int val)
    {
        int tl = tree[p].tl;
        int tr = tree[p].tr;
        // 无交集
        if(tr < ul or tl > ur) return;
        // 完全包含
        if(ul <= tl and tr <= ur)
        {
            apply_add(p,val);
            return;
        }
        // 部分交集则需分裂
        push_down(p);
        int mid = (tl + tr) >> 1;
```

```
            if (ul <= mid && tree[p].l == 0)
            {
                tree[p].l = ++cnt;
                tree[tree[p].l].tl = tl;
                tree[tree[p].l].tr = mid;
            }
            if (ur > mid && tree[p].r == 0)
            {
                tree[p].r = ++cnt;
                tree[tree[p].r].tl = mid+1;
                tree[tree[p].r].tr = tr;
            }
            // 递归更新左右孩子
            if(tree[p].l != 0) updateadd(tree[p].l,ul,ur,val);
            if(tree[p].r != 0) updateadd(tree[p].r,ul,ur,val);
            push_up(p);
        }

        int querysum(int p,int ql,int qr)
        {
            int tl = tree[p].tl;
            int tr = tree[p].tr;
            // 无交集
            if (tr < ql or tl > qr) return 0;
            // 完全包含
            if (ql <= tl and tr <= qr) return tree[p].sum;
            // 部分交集，需分裂
            push_down(p);
            int res = 0;
            // 查询左右孩子
            if (tree[p].l != 0) res += querysum(tree[p].l,ql,qr);
            if (tree[p].r != 0) res += querysum(tree[p].r,ql,qr);
            return res;
        }
};

void solve()
{
    int n,m;
    cin >> n >> m;
    DynamicSegTree dst(1,n);
    for(int i=1;i<=m;i++)
    {
        int op;
        cin >> op;
        if(op == 1)
        {
            int l,r,k;
            cin >> l >> r >> k;
            dst.updateadd(1,l,r,k);
        }
        else
        {
            int l,r;
            cin >> l >> r;
```

```
            cout << dst.querysum(1,l,r) << endl;
        }
    }
}
```

Treap Tree

```
// 实现一种结构，支持如下操作，要求单次调用的时间复杂度O(log n)
// 1，增加x，重复加入算多个词频
// 2，删除x，如果有多个，只删掉一个
// 3，查询x的排名，x的排名为，比x小的数的个数+1
// 4，查询数据中排名为x的数
// 5，查询x的前驱，x的前驱为，小于x的数中最大的数，不存在返回整数最小值
// 6，查询x的后继，x的后继为，大于x的数中最小的数，不存在返回整数最大值
// 所有操作的次数 <= 10^5
// -10^7 <= x <= +10^7
#include <bits/stdc++.h>
#define Single
using namespace std;
const int MAXN = 100001;
struct Treap
{
    int cnt = 0;  // 节点计数器，记录当前使用的节点数量
    int head = 0; // 根节点编号
    int key[MAXN]; // 存储节点的键值
    int key_count[MAXN]; // 存储每个键值的出现次数
    int ls[MAXN]; // 存储左子节点编号
    int rs[MAXN]; // 存储右子节点编号
    int SIZE[MAXN]; // 存储以该节点为根的子树的节点总数
    double priority[MAXN]; // 存储每个节点的优先级
    // 更新以节点 i 为根的子树的节点总数
    void up(int i) {
        SIZE[i] = SIZE[ls[i]] + SIZE[rs[i]] + key_count[i];
    }
    // 左旋操作，用于维护 Treap 的堆性质
    int leftRotate(int i) {
        int r = rs[i]; // 取出右子节点
        rs[i] = ls[r]; // 将右子节点的左子节点作为当前节点的右子节点
        ls[r] = i; // 当前节点成为右子节点的左子节点
        up(i); // 更新当前节点的子树节点总数
        up(r); // 更新右子节点的子树节点总数
        return r; // 返回新的根节点
    }
    // 右旋操作，用于维护 Treap 的堆性质
    int rightRotate(int i) {
        int l = ls[i]; // 取出左子节点
        ls[i] = rs[l]; // 将左子节点的右子节点作为当前节点的左子节点
        rs[l] = i; // 当前节点成为左子节点的右子节点
        up(i); // 更新当前节点的子树节点总数
        up(l); // 更新左子节点的子树节点总数
        return l; // 返回新的根节点
    }
    // 递归添加键值 num 到以节点 i 为根的子树中
```

```
    int add(int i, int num) {
        if (i == 0) { // 如果当前节点为空
            key[++cnt] = num; // 创建新节点并存储键值
            key_count[cnt] = SIZE[cnt] = 1; // 初始化键值出现次数和子树节点总数
            priority[cnt] = static_cast<double>(rand()) / RAND_MAX; // 随机生成优先
级
            return cnt; // 返回新节点编号
        }
        if (key[i] == num) { // 如果键值已存在
            key_count[i]++; // 增加键值出现次数
        } else if (key[i] > num) { // 如果键值小于当前节点的键值
            ls[i] = add(ls[i], num); // 递归添加到左子树
        } else { // 如果键值大于当前节点的键值
            rs[i] = add(rs[i], num); // 递归添加到右子树
        }
        up(i); // 更新当前节点的子树节点总数
        if (ls[i] != 0 && priority[ls[i]] > priority[i]) { // 如果左子节点优先级更高
            return rightRotate(i); // 进行右旋操作
        }
        if (rs[i] != 0 && priority[rs[i]] > priority[i]) { // 如果右子节点优先级更高
            return leftRotate(i); // 进行左旋操作
        }
        return i; // 返回当前节点编号
    }
    // 对外提供的添加键值的接口
    void add(int num) {
        head = add(head, num); // 从根节点开始添加
    }
    // 递归计算以节点 i 为根的子树中小于 num 的节点数量
    int small(int i, int num) {
        if (i == 0) { // 如果当前节点为空
            return 0; // 返回 0
        }
        if (key[i] >= num) { // 如果当前节点的键值大于等于 num
            return small(ls[i], num); // 递归计算左子树中小于 num 的节点数量
        } else { // 如果当前节点的键值小于 num
            return SIZE[ls[i]] + key_count[i] + small(rs[i], num); // 加上左子树节
点总数、当前节点键值出现次数和右子树中小于 num 的节点数量
        }
    }
    // 计算键值 num 的排名
    int getRank(int num) {
        return small(head, num) + 1; // 小于 num 的节点数量加 1
    }
    // 递归查找以节点 i 为根的子树中排名为 x 的节点的键值
    int index(int i, int x) {
        if (SIZE[ls[i]] >= x) { // 如果左子树节点总数大于等于 x
            return index(ls[i], x); // 递归在左子树中查找
        } else if (SIZE[ls[i]] + key_count[i] < x) { // 如果左子树节点总数加上当前节点
键值出现次数小于 x
            return index(rs[i], x - SIZE[ls[i]] - key_count[i]); // 递归在右子树中
查找，排名减去左子树节点总数和当前节点键值出现次数
        }
        return key[i]; // 返回当前节点的键值
    }
```

```
    // 对外提供的查找排名为 x 的节点的键值的接口
    int index(int x) {
        return index(head, x); // 从根节点开始查找
    }
    // 递归查找以节点 i 为根的子树中小于 num 的最大键值
    int pre(int i, int num) {
        if (i == 0) { // 如果当前节点为空
            return INT_MIN; // 返回最小整数
        }
        if (key[i] >= num) { // 如果当前节点的键值大于等于 num
            return pre(ls[i], num); // 递归在左子树中查找
        } else { // 如果当前节点的键值小于 num
            return max(key[i], pre(rs[i], num)); // 返回当前节点键值和右子树中小于 num
的最大键值的较大值
        }
    }
    // 对外提供的查找小于 num 的最大键值的接口
    int pre(int num) {
        return pre(head, num); // 从根节点开始查找
    }
    // 递归查找以节点 i 为根的子树中大于 num 的最小键值
    int post(int i, int num) {
        if (i == 0) { // 如果当前节点为空
            return INT_MAX; // 返回最大整数
        }
        if (key[i] <= num) { // 如果当前节点的键值小于等于 num
            return post(rs[i], num); // 递归在右子树中查找
        } else { // 如果当前节点的键值大于 num
            return min(key[i], post(ls[i], num)); // 返回当前节点键值和左子树中大于
num 的最小键值的较小值
        }
    }
    // 对外提供的查找大于 num 的最小键值的接口
    int post(int num) {
        return post(head, num); // 从根节点开始查找
    }
    // 递归从以节点 i 为根的子树中删除键值 num
    int remove(int i, int num) {
        if (key[i] < num) { // 如果当前节点的键值小于 num
            rs[i] = remove(rs[i], num); // 递归在右子树中删除
        } else if (key[i] > num) { // 如果当前节点的键值大于 num
            ls[i] = remove(ls[i], num); // 递归在左子树中删除
        } else { // 如果当前节点的键值等于 num
            if (key_count[i] > 1) { // 如果键值出现次数大于 1
                key_count[i]--; // 减少键值出现次数
            } else { // 如果键值出现次数为 1
                if (ls[i] == 0 && rs[i] == 0) { // 如果当前节点是叶子节点
                    return 0; // 返回 0 表示删除该节点
                } else if (ls[i] != 0 && rs[i] == 0) { // 如果只有左子节点
                    i = ls[i]; // 用左子节点替换当前节点
                } else if (ls[i] == 0 && rs[i] != 0) { // 如果只有右子节点
                    i = rs[i]; // 用右子节点替换当前节点
                } else { // 如果有左右子节点
                    if (priority[ls[i]] >= priority[rs[i]]) { // 如果左子节点优先级
更高
```

```
                    i = rightRotate(i); // 进行右旋操作
                    rs[i] = remove(rs[i], num); // 递归在右子树中删除
                } else { // 如果右子节点优先级更高
                    i = leftRotate(i); // 进行左旋操作
                    ls[i] = remove(ls[i], num); // 递归在左子树中删除
                }
            }
        }
    }
    up(i); // 更新当前节点的子树节点总数
    return i; // 返回当前节点编号
}
// 对外提供的删除键值的接口
void remove(int num) {
    if (getRank(num) != getRank(num + 1)) { // 如果键值存在
        head = remove(head, num); // 从根节点开始删除
    }
}
// 清空 Treap 树 时间复杂度并非O(T*N),而是树中的节点个数，多组样例用此清空
void clear() {
    fill(key + 1, key + cnt + 1, 0); // 清空键值数组
    fill(key_count + 1, key_count + cnt + 1, 0); // 清空键值出现次数数组
    fill(ls + 1, ls + cnt + 1, 0); // 清空左子节点编号数组
    fill(rs + 1, rs + cnt + 1, 0); // 清空右子节点编号数组
    fill(SIZE + 1, SIZE + cnt + 1, 0); // 清空子树节点总数数组
    fill(priority + 1, priority + cnt + 1, 0); // 清空优先级数组
    cnt = 0; // 重置节点计数器
    head = 0; // 重置根节点编号
}
};
void solve()
{
    int n;
    cin >> n;
    Treap treap; // 创建 Treap 对象
    for (int i = 1, op, x; i <= n; i++)
    {
        cin >> op >> x; // 读取操作类型和操作数
        if (op == 1)
        {   // 插入操作
            treap.add(x);
        }
        if (op == 2)
        {   // 删除操作
            treap.remove(x);
        }
        if (op == 3)
        {   // 查询排名操作
            cout << treap.getRank(x) << endl;
        }
        if (op == 4)
        {   // 查询排名对应的键值操作
            cout << treap.index(x) << endl;
        }
        if (op == 5)
```

```
        {   // 查询前驱操作
            cout << treap.pre(x) << endl;
        }
        if(op == 6)
        {   // 查询后继操作
            cout << treap.post(x) << endl;
        }
    }
    treap.clear();
}

signed main()
{
    ios::sync_with_stdio(false);cin.tie(nullptr);cout.tie(nullptr);
    srand(time(0)); // 初始化随机数种子
    ... ...
}
```

Old Driver Tree

```
const int N = 1e5+10;
const int M = 0;
struct ODT
{
    struct Node
    {
        int l, r;
        mutable int val;
        Node(int l, int r = -1, int val = 0) : l(l), r(r), val(val) {}
        bool operator<(const Node& o) const
        {
            return l < o.l;
        }
    };
    set<Node> tree;
    // 分裂操作
    auto split(int pos)
    {
        auto it = tree.lower_bound(Node(pos));
        if (it != tree.end() && it->l == pos) return it;
        --it;
        int l = it->l, r = it->r, val = it->val;
        tree.erase(it);
        tree.insert(Node(l, pos - 1, val));
        return tree.insert(Node(pos, r, val)).first;
    }
    // 初始化珂朵莉树
    void init(int n, int val)
    {
        tree.insert(Node(1, n, val));
    }
    // 区间赋值操作
    void assign(int l, int r, int val)
    {
        auto itr = split(r + 1), itl = split(l);
```

```cpp
        tree.erase(itl, itr);
        tree.insert(Node(l, r, val));
    }
    // 区间上的数统一加上x
    void add(int l, int r, int x)
    {
        set<Node>::iterator itr = split(r + 1), itl = split(l);
        for (set<Node>::iterator it = itl; it != itr; ++it)
        {
            it->val += x;
        }
    }
    //返回区间内颜色种类
    int output(int l, int r)
    {
        auto itr = split(r + 1), itl = split(l);
        int colors = 0;
        for (auto it = itl; it != itr; ++it) {
            colors |= (1 << (it->val - 1));
        }
        return __builtin_popcount(colors);
    }
    //返回整个区间的可视颜色（底层颜色会被表层颜色覆盖）
    int countDistinctValues()
    {
        set<int> values;
        for (const auto& node : tree)
        {
            if (node.val != 0) {
                values.insert(node.val);
            }
        }
        return values.size();
    }
    //输出将[l,r] 区间从小到大排序后的第x 个数
    struct Rank
    {
        int num, cnt;

        bool operator<(const Rank &a) const {
            return num < a.num;
        }

        Rank(int num, int cnt) : num(num), cnt(cnt) {}
    };
    int findrank(int l, int r, int x)
    {
        set<Node>::iterator itr = split(r + 1), itl = split(l);
        vector<Rank> v;
        for (set<Node>::iterator i = itl; i != itr; ++i)
        {
            v.push_back(Rank(i->val, i->r - i->l + 1));
        }
        sort(v.begin(), v.end());
        int i;
```

```
        for (i = 0; i < v.size(); ++i)
        {
            if (v[i].cnt < x) {
                x -= v[i].cnt;
            } else {
                break;
            }
        }
        return v[i].num;
    }
    void print()
    {
        for(const auto &node : tree)
        {
            cout << "[" << node.l << ", " << node.r << "] = " << node.val <<
endl;
        }
    }
};
```

Int128

```
namespace my128
{
    using int128 = __int128_t;
    int128 abs(const int128 &x)
    {
        return x > 0 ? x : -x;
    }
    auto &operator>>(istream &it,int128 &j)
    {
        string val;
        it >> val;
        reverse(val.begin(),val.end());
        int128 ans = 0;
        bool f = false;
        char c = val.back();
        val.pop_back();
        for(;c<'0'||c>'9';c=val.back(),val.pop_back())
        {
            if(c=='-') f = 1;
        }
        for(;c>='0'&&c<='9';c=val.back(),val.pop_back())
        {
            ans = ans * 10 + c - '0';
        }
        j = f ? -ans : ans;
        return it;
    }
    auto &operator<<(ostream &os,const int128 &j)
    {
        string ans;
        function<void(int128)> write = [&](int128 x)
        {
            if(x<0) ans += '-',x = -x;
```

```
            if(x>9) write(x/10);
            ans += x % 10 + '0';
        };
        write(j);
        return os << ans;
    }
}
using namespace my128;
void solve()
{
    int128 a,b;
    cin >> a >> b;
    cout << a + b << endl;
}
```

Discretization

```
struct Trans
{
    vector<int> F;
    // 从数组初始化离散化数据
    void init(int A[],int n)
    {
        // 清空 F 向量
        vector<int>().swap(F);
        for (int i = 1;i<=n;i++)F.push_back(A[i]);
        sort(F.begin(), F.end());
        F.erase(unique(F.begin(), F.end()), F.end());
    }
    // 从vector初始化离散化数据
    void init(const vector<int>& A)
    {
        // 将向量 A 中的元素添加到 F 向量中
        for (auto x : A) F.push_back(x);
        sort(F.begin(),F.end());
        F.erase(unique(F.begin(),F.end()),F.end());
    }
    // 找到第一个大于等于 val 的离散化后的值
    int findhigh(int val)
    {
        int x = lower_bound(F.begin(), F.end(), val) - F.begin() + 1;
        return x;
    }
    // 找到最后一个小于等于 val 的离散化后的值
    int findlow(int val)
    {
        int x = upper_bound(F.begin(), F.end(), val) - F.begin();
        return x;
    }
    // 将数组 A 中的元素转换为离散化后的值
    void change(int A[], int n)
    {
        for (int i = 1; i <= n; ++i)
        {
            A[i] = findhigh(A[i]);
```

```
        }
    }
};
int main() {
    int A[] = {0,100,22,33,1};
    int n = 4;
    Trans lis;
    // 使用数组初始化离散化数据
    lis.init(A,n);
    // 输出离散化后存储在 F 中的数据
    cout << "离散化后存储的数据: ";
    for (int num : lis.F)
    {
        cout << num << " ";
    }
    cout << endl;
    int val = 2;
    int high = lis.findhigh(val);
    int low = lis.findlow(val);
    cout << "第一个大于等于 " << val << " 的离散化后的值是: " << high << endl;
    cout << "最后一个小于等于 " << val << " 的离散化后的值是: " << low << endl;
    // 将数组 A 中的元素转换为离散化后的值
    lis.change(A,n);
    cout << "离散化后的数组: ";
    for (int i = 1; i <= n; ++i) {
        cout << A[i] << " ";
    }
    cout << endl;
    return 0;
}
```

Heavy Light Decomposition

树链剖分中的重链剖分：

> 树上每个节点都有如下信息：
>
> $父节点-fa$
>
> $深度-dep$
>
> $子树大小-size$
>
> $重儿子(子树规模最大的儿子，如果多个子树都是最大规模，任选一个孩子成为重孩子)-son$
>
> $所在重链的头节点-top$
>
> $递归序的编号-dfn$
>
> $seg[i] = j，代表dfn序号为i的节点，原始节点编号为j-seg$

dfs1过程：设置fa,dep,size,son,这个过程不分配dfn序号

dfs2过程：设置top,dfn,seg **先遍历重儿子，再遍历剩下的轻儿子，依次分配dfn序号，父节点和重儿子属于同一条重链，拥有同样的top；轻儿子新开一条重链，轻儿子作为新重链的头**

完成dfs1，dfs2后，整棵树会被若干条重链分解掉：

同一条重链的节点，dfn序号是连续的

同一个子树的节点，dfn序号是连续的

利用这两个性质，树链剖分能够与其他结构结合，最常见的情况是与线段树结合

修改/查询 任意子树上的信息：

就是线段树的单次操作

由于dfn序号都是连续的，我们可以对dfn进行建线段树，对一颗以u为根的子树上信息的修改对应的连续范围是：

$$[dfn[u], dfn[u] + size[u] - 1]$$

修改/查询 两点间的路径信息：

与子树操作类似，不断向上找到多条重链，每条重链的信息利用线段树查询，所有重链信息进行拼接整合

路径上单次查询时间复杂度为O(logN的平方)，字树上单次查询复杂度O(logN)，证明类似启发式合并

```
struct SegTree
{
    ... ...
};
struct HLD
{
    struct Edge
    {
        int to,next;
    };
    vector<int> size,top,dep,son,dfn,seg,fa,out; // out为离开时间戳
    int cnt;    // 建边计数器
    int cntd; // dfn 计数器
    int n; // 节点数量
    vector<int> head;
    vector<Edge> edge;
    HLD() {}
    HLD(int n)
    {
        init(n);
    }
    void init(int n)
    {
        this -> n = n; // 记录节点数量
        size.resize(n+1);
        top.resize(n+1);
        head.resize(n+1);
        dep.resize(n+1);
        son.resize(n+1);
        dfn.resize(n+1);
        out.resize(n+1);
        seg.resize(n+1);
        fa.resize(n+1);
        edge.resize((n+1)<<1);
        cnt = 0;
        cntd = 0;
    }
```

```
void addedge(int u,int v)
{
    edge[++cnt] = {v,head[u]};
    head[u] = cnt;
}
void dfs(int u,int f)
{
    fa[u] = f;
    dep[u] = (dep[f] == -1) ? 1 : dep[f] + 1;
    size[u] = 1;
    for(int i=head[u];i;i=edge[i].next)
    {
        int v = edge[i].to;
        if(v != f)
        {
            dfs(v,u);
        }
    }
    for(int i=head[u];i;i=edge[i].next)
    {
        int v = edge[i].to;
        if(v != f)
        {
            size[u] += size[v];
            // 重孩子没被更新过或者已找到更重的孩子
            if(son[u] == 0 or size[son[u]] < size[v])
            {
                son[u] = v;
            }
        }
    }
}
// 来到节点u，节点u所在重链的头节点是t
void dfs2(int u,int t)
{
    top[u] = t;
    dfn[u] = ++cntd;
    seg[cntd] = u;
    if(!son[u]) return; // 如果已经到叶子节点则终止
    dfs2(son[u],t);// 有的话以他的重儿子为起始继续递归
    for(int i=head[u];i;i=edge[i].next)
    {
        int v = edge[i].to;
        if(v != fa[u] and v != son[u])
        {
            dfs2(v,v);// 遍历剩余轻儿子，轻儿子以自己编号
        }
    }
    out[u] = cntd;// 记录离开时间，子树区间为[dfn[u],out[u]]
}
int lca(int u,int v)
{
    while(top[u] != top[v])
    {
        if(dep[top[u]] > dep[top[v]]) u = fa[top[u]];
```

```
            else v = fa[top[v]];
        }
        return dep[u] < dep[v] ? u : v;
    }
    int dist(int u,int v)
    {
        return dep[u] + dep[v] - 2 * dep[lca(u, v)];
    }
    // 从u向上跳k步
    int jump(int u,int k)
    {
        // 跳的步数超过深度
        if(dep[u] - k < 1) return -1;
        int tar = dep[u] - k;
        while(dep[top[u]] > tar)
        {
            u = fa[top[u]];
        }
        // 计算在当前链上需要向上跳的步数
        return seg[dfn[u]-(dep[u]-tar)];
    }
    // 判断u是否是v的祖先
    bool isAncestor(int u,int v)
    {
        return dfn[u] <= dfn[v] and out[u] >= dfn[v];
    }
    // 在以v为根的树中，u的父节点
    int rootedParent(int u,int v)
    {
        if(u == v) return u;
        if(!isAncestor(u, v)) return fa[u];
        // 找到u的子树中包含v的那个孩子
        for(int i = head[u];i;i = edge[i].next)
        {
            int w = edge[i].to;
            if(w != fa[u] && isAncestor(w, v)) return w;
        }
        return -1;  // 理论上不会走到这里
    }
    // 在以v为根的树中，u的子树大小
    int rootedSize(int u,int v)
    {
        if(u == v) return n;
        if(!isAncestor(v, u)) return size[u];
        // 子树大小 = 总节点数 - 原树中u的父节点方向的子树大小
        return n - size[rootedParent(u,v)];
    }
    // 在以c为根的树中，a和b的LCA
    int rootedLca(int a,int b,int c)
    {
        return lca(a, b) ^ lca(b, c) ^ lca(c, a);
    }
};
void solve()
{
```

```
    int n,m,r;
    cin >> n >> m >> r; // r 为树的指定根
    vector<int> arr(n+1);
    for(int i=1;i<=n;i++) cin >> arr[i];
    HLD hld(n);
    for(int i=1;i<n;i++)
    {
        int u,v;
        cin >> u >> v;
        hld.addedge(u,v);
        hld.addedge(v,u);
    }
    hld.dfs(r,-1);
    hld.dfs2(r,r);
    vector<int> w(n+1); // 对dfn序号建树
    for(int i=1;i<=n;i++)
    {
        w[i] = arr[hld.seg[i]]; // 为dfn序号为i的原始节点赋点权
    }
    SegTree st(w);
    // 询问x -> y树上简单路径上的和
    auto querypathsum = [&](int x,int y) // O(logn * logn)
    {
        int res = 0;
        while(hld.top[x] != hld.top[y])
        {
            if(hld.dep[hld.top[x]] < hld.dep[hld.top[y]]) swap(x,y);
            res += st.querySum(1,hld.dfn[hld.top[x]],hld.dfn[x]);
            x = hld.fa[hld.top[x]];
        }
        if(hld.dfn[x] > hld.dfn[y]) swap(x,y);
        res += st.querySum(1,hld.dfn[x],hld.dfn[y]);
        return res;
    };
    // 将 x -> y 简单路径上的点权加 val
    auto updatepathsum = [&](int x,int y,int val) // O(logn * logn)
    {
        while(hld.top[x] != hld.top[y]) //
        {
            if(hld.dep[hld.top[x]] < hld.dep[hld.top[y]]) swap(x,y);
            st.updateAdd(1,hld.dfn[hld.top[x]],hld.dfn[x],val);
            x = hld.fa[hld.top[x]];
        }
        if(hld.dfn[x] > hld.dfn[y]) swap(x,y);
        st.updateAdd(1,hld.dfn[x],hld.dfn[y],val);
    };
    // 将根为u的子树上的所有节点点权加val
    auto updatenodesum = [&](int u,int val) // O(logn)
    {
        st.updateAdd(1,hld.dfn[u],hld.dfn[u]+hld.size[u]-1,val);
    };
    // 询问以u为根子树上的点的权值和
    auto querynodesum = [&](int u)// O(logn)
    {
        return st.querySum(1,hld.dfn[u],hld.dfn[u]+hld.size[u]-1);
```

```
    };
}
```

$线段树合并结合树链剖分$

此时不仅要在线段树的线段合并处讨论，也需要在重链连接处的信息合并处讨论：

例题：实现区间颜色覆盖，询问区间中颜色段数目，如11222111，中有3个颜色段分别为11，222，111

```
struct SegTree
{
        struct Node
        {
            int l,r;
            int lc,rc; // 左右边界的颜色
            int sum; // 线段中的颜色段
            int set;
            bool change;
            Node() : set(0), change(false) {}
            inline int len()
            {
                return r - l + 1;
            }
        } tree[N << 2];
        vector<int> arr;

        void push_up(int p)
        {
            tree[p].sum = tree[ls(p)].sum + tree[rs(p)].sum - (tree[ls(p)].rc == tree[rs(p)].lc);
            tree[p].lc = tree[ls(p)].lc;
            tree[p].rc = tree[rs(p)].rc;
        }

        void apply_set(int p,int val)
        {
            tree[p].lc = tree[p].rc = val;
            tree[p].sum = 1;
            tree[p].set = val;
            tree[p].change = true;
        }

        void push_down(int p)
        {
            if (tree[p].change)
            {
                apply_set(ls(p), tree[p].set);
                apply_set(rs(p), tree[p].set);
                tree[p].change = false;
            }
        }
        SegTree(const vector<int>& _arr) : arr(_arr)
        {
            build(1, 1, (int)arr.size()-1);
```

```cpp
            //build(1, 0, (int)arr.size()-1);
        }
        void build(int p, int l, int r)
        {
            tree[p].l = l;
            tree[p].r = r;
            if (l == r)
            {
                tree[p].sum = 1;
                tree[p].lc = tree[p].rc = arr[l];
                return;
            }
            int mid = (l + r) >> 1;
            build(ls(p), l, mid);
            build(rs(p), mid+1, r);
            push_up(p);
        }
        void updateset(int p, int ul, int ur, int val)
        {
            if (ul <= tree[p].l && tree[p].r <= ur)
            {
                apply_set(p, val);
                return;
            }
            push_down(p);
            int mid = (tree[p].l + tree[p].r) >> 1;
            if (ul <= mid) updateset(ls(p), ul, ur, val);
            if (ur > mid) updateset(rs(p), ul, ur, val);
            push_up(p);
        }
        int querySum(int p, int ql, int qr)
        {
            if (ql <= tree[p].l && tree[p].r <= qr) return tree[p].sum;
            push_down(p);
            int mid = (tree[p].l + tree[p].r) >> 1;
            if(qr <= mid) return querySum(ls(p), ql, qr);
            if(ql > mid) return querySum(rs(p), ql, qr);
            int res = querySum(ls(p),ql,mid) + querySum(rs(p),mid+1,qr);
            if(tree[ls(p)].rc == tree[rs(p)].lc) res--;
            return res;
        }
        int querycolor(int p,int x)
        {
            if(x == tree[p].l and x == tree[p].r) return tree[p].lc;
            push_down(p);
            int mid = (tree[p].l + tree[p].r) >> 1;
            if(x <= mid) return querycolor(ls(p),x);
            if(x > mid) return querycolor(rs(p),x);
        }
};
struct HLD
{
    ... ...
};
// 线段树合并 + 重链剖分
```

```cpp
void solve()
{
    int n,m;
    cin >> n >> m;
    vector<int> arr(n+1);
    for(int i=1;i<=n;i++) cin >> arr[i];
    HLD hld(n);
    for(int i=1;i<n;i++)
    {
        int u,v;
        cin >> u >> v;
        hld.addedge(u,v);
        hld.addedge(v,u);
    }
    hld.dfs(1,-1);
    hld.dfs2(1,1);
    vector<int> w(n+1); // 对dfn序号建树
    for(int i=1;i<=n;i++)
    {
        w[i] = arr[hld.seg[i]]; // 为dfn序号为i的原始节点赋点权
    }
    SegTree st(w);
    auto querypathsum = [&](int x, int y) // O(logn * logn)
    {
        int res = 0;
        while(hld.top[x] != hld.top[y])
        {
            if(hld.dep[hld.top[x]] < hld.dep[hld.top[y]]) swap(x,y);
            res += st.querySum(1,hld.dfn[hld.top[x]],hld.dfn[x]);
            if(hld.fa[hld.top[x]]) // 如果所在重链头节点父节点存在
            {
                int soncolor = st.querycolor(1,hld.dfn[hld.top[x]]);
                int facolor = st.querycolor(1,hld.dfn[hld.fa[hld.top[x]]]);
                if(soncolor == facolor) res--;
            }
            x = hld.fa[hld.top[x]];
        }
        if(hld.dfn[x] > hld.dfn[y]) swap(x,y);
        res += st.querySum(1,hld.dfn[x],hld.dfn[y]);
        return res;
    };
    auto updatepathcolor = [&](int x,int y,int val)
    {
        while(hld.top[x] != hld.top[y])
        {
            if(hld.dep[hld.top[x]] < hld.dep[hld.top[y]]) swap(x,y);
            st.updateset(1,hld.dfn[hld.top[x]],hld.dfn[x],val);
            x = hld.fa[hld.top[x]];
        }
        if(hld.dfn[x] > hld.dfn[y]) swap(x,y);
        st.updateset(1,hld.dfn[x],hld.dfn[y],val);
    };
    ... ...
}
```

Algorithms

EulerGetPrime_O(N)

```
vector<int> pri;
bitset<100000> not_prime;
void Euler(int n)
{
    for(int i=2;i<=n;i++)
    {
        if(!not_prime[i])
        {
            pri.push_back(i);
        }
        for(int pri_j : pri)
        {
            if(pri_j*i>n) break;
            not_prime[pri_j*i] = true;
            if(i%pri_j==0) break;
        }
    }
}
```

Get_Segment_Prime

```
// O(sqrt(b) loglogsqrt(b) + (b-a)loglogsqrt(b))
const int N = 1e6+100;
// 区间筛法求[a, b)内的质数
bitset<N> not_prime; // 标记区间[a,b)内的合数
vector<int> pri;  // 存储区间[a,b)内的质数
void segprime(ll a,ll b)
{
    pri.clear();
    not_prime.reset();
    // 处理a=0或1的情况
    if (a == 0) a = 2;
    if (a == 1) a = 2;
    // 预处理sqrt(b)以内的质数
    ll sqrtb = sqrt(b) + 1;
    vector<int> primes;
    bitset<N> notprime;
    for (ll i = 2; i < sqrtb; i++)
    {
        if (!notprime[i])
        {
            primes.push_back(i);
            for (ll j = i * i; j < sqrtb; j += i)
                notprime[j] = 1;
        }
    }
    // 筛选区间[a,b)
    for (ll p : primes)
```

```
    {
        // 计算第一个>=a的p的倍数
        ll start = max(p * 2LL, (a + p - 1) / p * p);
        for (ll j = start; j < b; j += p)
            not_prime[j - a] = 1;
    }
    // 收集区间内的质数
    for (ll i = a; i < b; i++)
        if (!not_prime[i - a])
            pri.push_back(i);
}

void solve() {
    ll a, b;
    cin >> a >> b;
    segprime(a, b);
    // 输出区间内的质数个数
    cout << "区间[" << a << "," << b << ")内的质数个数: " << pri.size() << endl;

    // 输出所有质数（可选，根据需要取消注释）
    /*
    cout << "这些质数是: ";
    for (ll p : primes) cout << p << " ";
    cout << endl;
    */
}
```

QuickPow

```
int qpow(int x,int y,int mod)
{
    int res = 1;
    while(y>0)
    {
        if(y&1) res = (res * x) % mod;
        x = (x * x) % mod;
        y >>= 1;
    }
    return res;
}
```

Invmod

```
int inv(int x) // 求x在MOD模状态下的逆元
{
    return qpow(x,MOD-2) % MOD;
}

int C(int n,int m,int MOD) // n 选 m
{
    int res = fac[n] % MOD; // fac[i] : i的阶乘,需要预处理
    res *= invfac[n] % MOD;
    res *= invfac[n-m] % MOD;
    return res;
```

```
}

int calcu(int p,int q) // 计算p / q的同余
{
    return ((p % MOD)*(invfac(q))) % MOD;
}
```

Bitwise Algorithms

```
    unsigned int x = 0b101000;  // 40
    // GCC 内置函数
    cout << "前导零: " << __builtin_clz(x) << endl;      // 26
    cout << "末尾零: " << __builtin_ctz(x) << endl;      // 3
    cout << "1的个数: " << __builtin_popcount(x) << endl; // 2
    // 判断2的幂
    (x != 0 && (x & (x - 1)) == 0)
struct BA {
    // 加法
    int add(int a, int b) {
        while (b != 0) {
            int carry = a & b;
            a = a ^ b;
            b = carry << 1;
        }
        return a;
    }
    // 减法
    int subtract(int a, int b) {
        return add(a, add(~b, 1));
    }

    // 乘法
    int multiply(int a, int b) {
        int result = 0;
        bool negative = (a < 0) ^ (b < 0);
        a = a < 0 ? add(~a, 1) : a;
        b = b < 0 ? add(~b, 1) : b;

        while (b != 0) {
            if (b & 1) {
                result = add(result, a);
            }
            a <<= 1;
            b >>= 1;
        }

        return negative ? add(~result, 1) : result;
    }

    // 除法
    int divide(int a, int b) {
        if (b == 0) {
```

```
            throw std::runtime_error("Division by zero!");
        }
        bool negative = (a < 0) ^ (b < 0);
        a = a < 0 ? add(~a, 1) : a;
        b = b < 0 ? add(~b, 1) : b;

        int result = 0;
        for (int i = 31; i >= 0; --i) {
            if ((a >> i) >= b) {
                result = add(result, 1 << i);
                a = subtract(a, b << i);
            }
        }

        return negative ? add(~result, 1) : result;
    }
};

int main() {
    BA ba;
    int num1 = 10, num2 = 3;
    std::cout << "Addition: " << ba.add(num1, num2) << std::endl;
    std::cout << "Subtraction: " << ba.subtract(num1, num2) << std::endl;
    std::cout << "Multiplication: " << ba.multiply(num1, num2) << std::endl;
    try {
        std::cout << "Division: " << ba.divide(num1, num2) << std::endl;
    } catch (const std::exception& e) {
        std::cerr << e.what() << std::endl;
    }
    return 0;
}
```

Plane Geometry

```
struct Point
{
    long double x;
    long double y;
    // 点
    Point(long double x_ = 0, long double y_ = 0) : x(x_), y(y_) {}

    Point operator+(const Point &p) const {
        return Point(x + p.x, y + p.y);
    }
    Point operator-(const Point &p) const {
        return Point(x - p.x, y - p.y);
    }
    Point operator*(const long double &v) const {
        return Point(x * v, y * v);
    }
    Point operator/(const long double &v) const {
        return Point(x / v, y / v);
    }
    Point operator-() const {
        return Point(-x, -y);
```

```
    }
    bool operator==(const Point &p) const {
        return x == p.x && y == p.y;
    }
    bool operator!=(const Point &p) const {
        return !(*this == p);
    }
    friend std::istream &operator>>(std::istream &is, Point &p) {
        return is >> p.x >> p.y;
    }
    friend std::ostream &operator<<(std::ostream &os, const Point &p) {
        return os << "(" << p.x << ", " << p.y << ")";
    }
};
    // 直线 AB
struct Line {
    Point a;
    Point b;
    Line(const Point &a_ = Point(), const Point &b_ = Point()) : a(a_), b(b_) {}
};
    //计算两点点乘
long double dot(const Point &a, const Point &b) {
    return a.x * b.x + a.y * b.y;
}
    //计算两点叉乘
long double cross(const Point &a, const Point &b) {
    return a.x * b.y - a.y * b.x;
}
    //计算点平方(x^2 + y^2)
long double square(const Point &p) {
    return dot(p, p);
}
    //计算点到原点的长度
double length(const Point &p) {
    return std::sqrt(square(p));
}
    //计算线的长度
double length(const Line &l) {
    return length(l.a - l.b);
}
    //计算点的单位向量
Point normalize(const Point &p) {
    return p / length(p);
}
    //判断两线是否平行
bool parallel(const Line &l1, const Line &l2) {
    return cross(l1.b - l1.a, l2.b - l2.a) == 0;
}
    //计算两点距离
double distance(const Point &a, const Point &b) {
    return length(a - b);
}
    //计算点到直线距离
double distancePL(const Point &p, const Line &l) {
    return std::abs(cross(l.a - l.b, l.a - p)) / length(l);
```

```cpp
}
    //计算点到线段距离
double distancePS(const Point &p, const Line &l) {
    if (dot(p - l.a, l.b - l.a) < 0) {
        return distance(p, l.a);
    }
    if (dot(p - l.b, l.a - l.b) < 0) {
        return distance(p, l.b);
    }
    return distancePL(p, l);
}
    //将点绕原点逆时针旋转90度
Point rotate(const Point &a) {
    return Point(-a.y, a.x);
}
    //判断点在 x 轴上方还是下方
int sgn(const Point &a) {
    return a.y > 0 || (a.y == 0 && a.x > 0)? 1 : -1;
}
    //判断点是否在线的左侧
bool pointOnLineLeft(const Point &p, const Line &l) {
    return cross(l.b - l.a, p - l.a) > 0;
}
    // 计算两条线的交点
Point lineIntersection(const Line &l1, const Line &l2) {
    return l1.a + (l1.b - l1.a) * (cross(l2.b - l2.a, l1.a - l2.a) / cross(l2.b
- l2.a, l1.a - l1.b));
}
    // 计算两条线的交点
bool pointOnSegment(const Point &p, const Line &l) {
    return cross(p - l.a, l.b - l.a) == 0 && std::min(l.a.x, l.b.x) <= p.x &&
p.x <= std::max(l.a.x, l.b.x)
           && std::min(l.a.y, l.b.y) <= p.y && p.y <= std::max(l.a.y, l.b.y);
}
    // 判断点是否在多边形内 vector存储多边形点集
bool pointInPolygon(const Point &a, const std::vector<Point> &p) {
    int n = p.size();
    for (int i = 0; i < n; i++) {
        if (pointOnSegment(a, Line(p[i], p[(i + 1) % n]))) {
            return true;
        }
    }

    int t = 0;
    for (int i = 0; i < n; i++) {
        auto u = p[i];
        auto v = p[(i + 1) % n];
        if (u.x < a.x && v.x >= a.x && pointOnLineLeft(a, Line(v, u))) {
            t ^= 1;
        }
        if (u.x >= a.x && v.x < a.x && pointOnLineLeft(a, Line(u, v))) {
            t ^= 1;
        }
    }
```

```cpp
    return t == 1;
}
// 判断两条线段是否相交，并返回相交情况和交点
// 0：表示两条线段不相交，即两条线段在平面上没有任何公共点。
// 1：代表两条线段严格相交，意味着两条线段相交且交点不在任何一条线段的端点上，而是在两条线段的内部。
// 2：表示两条线段重叠，说明两条线段完全重合，它们有无数个公共点，即一条线段的所有点都在另一条线段上。
// 3：表示两条线段相交于端点，也就是两条线段相交，且交点是其中一条线段的端点或者两条线段的端点重合。
std::tuple<int, Point, Point> segmentIntersection(const Line &l1, const Line &l2) {
    if (std::max(l1.a.x, l1.b.x) < std::min(l2.a.x, l2.b.x)) {
        return {0, Point(), Point()};
    }
    if (std::min(l1.a.x, l1.b.x) > std::max(l2.a.x, l2.b.x)) {
        return {0, Point(), Point()};
    }
    if (std::max(l1.a.y, l1.b.y) < std::min(l2.a.y, l2.b.y)) {
        return {0, Point(), Point()};
    }
    if (std::min(l1.a.y, l1.b.y) > std::max(l2.a.y, l2.b.y)) {
        return {0, Point(), Point()};
    }
    if (cross(l1.b - l1.a, l2.b - l2.a) == 0) {
        if (cross(l1.b - l1.a, l2.a - l1.a) != 0) {
            return {0, Point(), Point()};
        } else {
            auto maxx1 = std::max(l1.a.x, l1.b.x);
            auto minx1 = std::min(l1.a.x, l1.b.x);
            auto maxy1 = std::max(l1.a.y, l1.b.y);
            auto miny1 = std::min(l1.a.y, l1.b.y);
            auto maxx2 = std::max(l2.a.x, l2.b.x);
            auto minx2 = std::min(l2.a.x, l2.b.x);
            auto maxy2 = std::max(l2.a.y, l2.b.y);
            auto miny2 = std::min(l2.a.y, l2.b.y);
            Point p1(std::max(minx1, minx2), std::max(miny1, miny2));
            Point p2(std::min(maxx1, maxx2), std::min(maxy1, maxy2));
            if (!pointOnSegment(p1, l1)) {
                std::swap(p1.y, p2.y);
            }
            if (p1 == p2) {
                return {3, p1, p2};
            } else {
                return {2, p1, p2};
            }
        }
    }
    auto cp1 = cross(l2.a - l1.a, l2.b - l1.a);
    auto cp2 = cross(l2.a - l1.b, l2.b - l1.b);
    auto cp3 = cross(l1.a - l2.a, l1.b - l2.a);
    auto cp4 = cross(l1.a - l2.b, l1.b - l2.b);

    if ((cp1 > 0 && cp2 > 0) || (cp1 < 0 && cp2 < 0) || (cp3 > 0 && cp4 > 0) ||
(cp3 < 0 && cp4 < 0)) {
```

```cpp
        return {0, Point(), Point()};
    }

    Point p = lineIntersection(l1, l2);
    if (cp1 != 0 && cp2 != 0 && cp3 != 0 && cp4 != 0) {
        return {1, p, p};
    } else {
        return {3, p, p};
    }
}
    // 计算两条线段之间的距离
double distanceSS(const Line &l1, const Line &l2) {
    if (std::get<0>(segmentIntersection(l1, l2)) != 0) {
        return 0.0;
    }
    return std::min({distancePS(l1.a, l2), distancePS(l1.b, l2),
distancePS(l2.a, l1), distancePS(l2.b, l1)});
}
    // 判断线段是否在多边形内
bool segmentInPolygon(const Line &l, const std::vector<Point> &p) {
    int n = p.size();
    if (!pointInPolygon(l.a, p)) {
        return false;
    }
    if (!pointInPolygon(l.b, p)) {
        return false;
    }
    for (int i = 0; i < n; i++) {
        auto u = p[i];
        auto v = p[(i + 1) % n];
        auto w = p[(i + 2) % n];
        auto [t, p1, p2] = segmentIntersection(l, Line(u, v));

        if (t == 1) {
            return false;
        }
        if (t == 0) {
            continue;
        }
        if (t == 2) {
            if (pointOnSegment(v, l) && v != l.a && v != l.b) {
                if (cross(v - u, w - v) > 0) {
                    return false;
                }
            }
        } else {
            if (p1 != u && p1 != v) {
                if (pointOnLineLeft(l.a, Line(v, u))
                    || pointOnLineLeft(l.b, Line(v, u))) {
                    return false;
                }
            } else if (p1 == v) {
                if (l.a == v) {
                    if (pointOnLineLeft(u, l)) {
                        if (pointOnLineLeft(w, l)
```

```cpp
                            && pointOnLineLeft(w, Line(u, v))) {
                            return false;
                        }
                    } else {
                        if (pointOnLineLeft(w, l)
                            || pointOnLineLeft(w, Line(u, v))) {
                            return false;
                        }
                    }
                } else if (l.b == v) {
                    if (pointOnLineLeft(u, Line(l.b, l.a))) {
                        if (pointOnLineLeft(w, Line(l.b, l.a))
                            && pointOnLineLeft(w, Line(u, v))) {
                            return false;
                        }
                    } else {
                        if (pointOnLineLeft(w, Line(l.b, l.a))
                            || pointOnLineLeft(w, Line(u, v))) {
                            return false;
                        }
                    }
                } else {
                    if (pointOnLineLeft(u, l)) {
                        if (pointOnLineLeft(w, Line(l.b, l.a))
                            || pointOnLineLeft(w, Line(u, v))) {
                            return false;
                        }
                    } else {
                        if (pointOnLineLeft(w, l)
                            || pointOnLineLeft(w, Line(u, v))) {
                            return false;
                        }
                    }
                }
            }
        }
    }
    return true;
};
using P = Point; // 使用 Point 类型，保持与结构体定义一致
void solve() {
    int n;
    std::cin >> n;
    std::vector<P> p(n);
    for (int i = 0; i < n; i++) {
        std::cin >> p[i].x >> p[i].y;
    }
    int ans = 0;
    for (int i = 0; i < n; i++) {
        std::map<std::pair<long double, long double>, int> mp; // 为 long double 类型
        for (int j = 0; j < n; j++) {
            for (int k = 0; k < n; k++) {
                if (i == j || i == k || j == k) continue;
                if (square(p[j] - p[i]) != square(p[k] - p[i])) continue;
```

```
                if (cross(p[j] - p[i], p[k] - p[i]) <= 0) continue;
                auto d = p[k] - p[j];
                ans += mp[{d.x, d.y}]++;
            }
        }
    }
    std::cout << ans;
}
```

Solid Geometry

```
using P = long double; // 定义默认数据类型(double / long double)
struct Point {
    P x = 0;
    P y = 0;
    P z = 0;
};
// 重载三维点/向量的加减乘除
Point operator+(const Point &a, const Point &b) {
    return {a.x + b.x, a.y + b.y, a.z + b.z};
}

Point operator-(const Point &a, const Point &b) {
    return {a.x - b.x, a.y - b.y, a.z - b.z};
}

Point operator*(const Point &a, P b) {
    return {a.x * b, a.y * b, a.z * b};
}

Point operator/(const Point &a, P b) {
    return {a.x / b, a.y / b, a.z / b};
}
// 返回向量模长
P length(const Point &a) {
    return hypot(a.x, a.y, a.z);
}
// 返回单位向量
Point normalize(const Point &a) {
    P l = length(a);
    return {a.x / l, a.y / l, a.z / l};
}
// 返回 a 和 b 的夹角
P getAng(P a, P b, P c) {
    return acos((a * a + b * b - c * c) / 2 / a / b);
}

ostream &operator<<(ostream &os, const Point &a) {
    return os << "(" << a.x << ", " << a.y << ", " << a.z << ")";
}

istream &operator>>(istream &is, Point &a)
{
    return is >> a.x >> a.y >> a.z;
}
```

```
// 点乘
P dot(const Point &a, const Point &b) {
    return a.x * b.x + a.y * b.y + a.z * b.z;
}
// 叉乘
Point cross(const Point &a, const Point &b) {
    return {
        a.y * b.z - a.z * b.y,
        a.z * b.x - a.x * b.z,
        a.x * b.y - a.y * b.x
    };
}
```

KMP Algorithm

```
// KMP 字符串匹配算法，找出 str 中所有与 pat 匹配的子串，并返回这些子串开头的位置
// 实现相同的算法可以使用 std::str.find(匹配字串pat,开始匹配的位置pos)库函数，但仅会返回第一
次出现的位置
struct KMP {
    // next数组，第i个数字反映了字符串pat中前i个字符构成的子串中真前缀和真后缀相等的最大长度。
    vector<int> next;
    string pat;
    KMP(const string& p) : pat(p) {
        int m = pat.length();
        next.resize(m + 1, 0);
        int j = 0;
        for (int i = 1; i < m; ++i) {
            while (j > 0 && pat[i] != pat[j]) {
                j = next[j];
            }
            if (pat[i] == pat[j]) {
                ++j;
            }
            next[i + 1] = j;
        }
    }
    vector<int> search(const string& str) {
        vector<int> matches;
        int n = str.length();
        int m = pat.length();
        int j = 0;
        for (int i = 0; i < n; ++i) {
            while (j > 0 && str[i] != pat[j]) {
                j = next[j];
            }
            if (str[i] == pat[j]) {
                ++j;
            }
            if (j == m) {
                matches.push_back(i - m + 1);
                j = next[j];
            }
        }
```

```
        return matches;
    }
};
signed main()
{
    string str;
    string pat;
    cin >> str >> pat;
    KMP kmp(pat);
    vector<int> matches = kmp.search(str);
    for(auto pos : matches)
    {
        cout << pos+1 << endl;
    }
    for(int i=1;i<=(int)pat.size();i++)
    {
        cout << kmp.next[i] << ' ';
    }
    return 0;
}
```

Dijkstra(Heap Promoted)

```
//最短路计数
vector<int> head(N);
int cnt = 0 ;
vector<int> dis(N,INF);
vector<int> ans(N,0);
vector<bool> vis(N,false);
struct node
{
    int to,next,w;
}edge[M<<1];
void add(int u,int v,int w)
{
    edge[++cnt].to = v;
    edge[cnt].w = w;
    edge[cnt].next = head[u];
    head[u] = cnt;
}
priority_queue<pii,vector<pii>,greater<pii>> q;
void dijkstra(int start)
{
    dis[start] = 0;
    ans[start] = 1;
    q.push({0,start});// distance & node
    while(!q.empty())
    {
        int u = q.top().second;
        q.pop();
        if(vis[u]) continue;
        vis[u] = true;
        for(int i=head[u];i;i=edge[i].next)
```

```
        {
            int v = edge[i].to;
            int w = edge[i].w;
            if(dis[v] > dis[u] + w)
            {
                dis[v] = dis[u] + w;
                ans[v] = ans[u];
                ans[v] %= MOD;
                q.push({dis[v],v});
            }
            else
            if(dis[v] == dis[u] + w)
            {
                ans[v] += ans[u];
                ans[v] %= MOD;
            }
        }
    }
}
void solve()
{
    int n,m;
    cin >> n >> m;
    for(int i=1;i<=m;i++)
    {
        int u,v;
        cin >> u >> v;
        add(u,v,1);
        add(v,u,1);
    }
    dijkstra(1);
    for(int i=1;i<=n;i++)
    {
        cout << ans[i] << endl;
    }
}
```

Get phi

```
//欧拉函数线性筛
//---------------------------------------------------
vector<int> pri;
bitset<N> not_prime;
int phi[N];
void Euler_Phi(int n)
{
  phi[1] = 1;
  for (int i = 2; i <= n; i++)
  {
    if (!not_prime[i])
    {
      pri.push_back(i);
      phi[i] = i - 1;
```

```
    }
    for (int pri_j : pri)
    {
      if (i * pri_j > n) break;
      not_prime[i * pri_j] = true;
      if (i % pri_j == 0)
      {
        phi[i * pri_j] = phi[i] * pri_j;
        break;
      }
      phi[i * pri_j] = phi[i] * phi[pri_j];//积性函数性质
    }
  }
}
//----------------------------------------------------
```

0-1 BFS

```
// 0-1BFS
// 适用于图中所有边的权重仅有0和1两种值的情况 O（N+M)
// 过程:
// 1.准备一个双端队列和dis[N]数组，dis[i]表示从源点到达i点的最短距离，初始化dis[N]为INF
// 2.源点进入双端队列，dis[源点] = 0;
// 3.双端队列 头部弹出 X
//      A.如果X是目标点,返回dis[X]即为最短距离
//      B.考察从X出发的每一条边,假设某边去往Y点,边权为W
//          1) 若 dis[x] > dis[y] + w 此时处理该边,否则忽略
//          2) 处理时更新 dis[x] = dis[y] + w
//              -1- 若W==0 Y从头部进入双端队列,继续重复步骤3
//              -2- 若W==1 Y从头部进入双端队列,继续重复步骤3
// 4.双端队列为空时停止
//--------------------------------------------------------------------------------
----------
const int N = 1e4+1;
const int M = 0;
int dx[] = {1,-1,0,0};
int dy[] = {0,0,-1,1};
int dis[N][N];
int mp[N][N];
int n,m;
//从(1,1)到达(n,m)的途中需要移除多少障碍物
bool check(int x,int y)
{
    return x>=1 && x<=n && y>=1 && y<=m;
}
void solve()
{
    memset(dis,0x3f,sizeof dis);
    cin >> n >> m;
    for(int i=1;i<=n;i++)
    {
        for(int j=1;j<=m;j++)
        {
            cin >> mp[i][j];
        }
```

```
    }
    deque<pii> q;//(x,y)
    q.push_front({1,1});
    dis[1][1] = 0;
    auto bfs = [&]()
    {
        while(!q.empty())
        {
            auto [x,y] = q.front();
            q.pop_front();
            if(x==n && y==m)
            {
                return dis[x][y];
            }
            for(int i=0;i<4;i++)
            {
                int nx = x + dx[i];
                int ny = y + dy[i];
                if(check(nx,ny) && dis[x][y] + mp[nx][ny] < dis[nx][ny])
                {
                    dis[nx][ny] = dis[x][y] + mp[nx][ny];
                    if(mp[nx][ny]==0)
                    {
                        q.push_front({nx,ny});
                    }
                    else
                    {
                        q.push_back({nx,ny});
                    }
                }
            }
        }
        return -1;//无法找到
    };
    cout << bfs();
}
```

Lucas

```
// 快速幂函数，用于计算乘法逆元
int qpow(int a,int b,int mod)
{
    int res = 1;
    while(b)
    {
        if(b&1) res = (res * a) % mod;
        a = (a * a) % mod;
        b >>= 1;
    }
    return res;
}
// 计算组合数取模的函数
int comb(int n, int k, int p)
```

```
{
    if (k>n) return 0;
    if (k==0) return 1;
    int up = 1,down = 1;
    for (int i = n-k+1;i<=n;i++) up = (up*i) % p;
    for (int i = 1;i<=k;i++) down = (down*i) % p;
    return (up*qpow(down,p-2,p)) % p;
}
// 卢卡斯定理函数
int lucas(int n, int k, int p)
{
    if(k==0) return 1;
    return (comb(n%p,k%p,p) * lucas(n/p,k/p,p)) % p;
}
//Lucas(n,k,p) = C(n,k) % p
```

BigInt

```
struct BigInt
{
    int a[N];
    BigInt(int x = 0) : a{}
     {
        for (int i = 0; x; i++)
        {
            a[i] = x % 10;
            x /= 10;
        }
    }
    BigInt(const std::string& s) : a{}
    {
        for (size_t i = 0; i < s.size(); i++) a[s.size() - 1 - i] = s[i] - '0';
    }
    BigInt &operator*=(int x)
    {
        for (int i = 0; i < N; i++) a[i] *= x;
        for (int i = 0; i < N - 1; i++)
        {
            a[i + 1] += a[i] / 10;
            a[i] %= 10;
        }
        return *this;
    }
    BigInt &operator/=(int x)
    {
        for (int i = N - 1; i >= 0; i--)
        {
            if (i) a[i - 1] += a[i] % x * 10;
            a[i] /= x;
        }
        return *this;
    }
    BigInt &operator+=(const BigInt &x)
    {
        for (int i = 0; i < N; i++)
```

```
        {
            a[i] += x.a[i];
            if (a[i] >= 10)
            {
                a[i + 1] += 1;
                a[i] -= 10;
            }
        }
        return *this;
    }
};

std::ostream &operator<<(std::ostream &o, const BigInt &a)
{
    int t = N - 1;
    while (a.a[t] == 0) t--;
    for (int i = t; i >= 0; i--) o << a.a[i];
    return o;
}
```

Checker

```
#include <bits/stdc++.h>
using namespace std;
int randint(int l, int r)
{
    int k = l;
    r++;
    k+=(1.0*rand()/RAND_MAX) * (r-l);
    return k;
}
bool compareFiles(const string &file1, const string &file2)
{
    ifstream f1(file1, ios::binary);
    ifstream f2(file2, ios::binary);
    return equal(istreambuf_iterator<char>(f1.rdbuf()),
                 istreambuf_iterator<char>(),
                 istreambuf_iterator<char>(f2.rdbuf()));
}
int main()
{
    int testcase = 1;
    while (true) {
        int seed = time(nullptr) + testcase;
        string gen_cmd = "gen.exe " + to_string(seed) + " > input.txt";
        system(gen_cmd.c_str());
        system("test.exe < input.txt > output1.txt");
        system("brute.exe < input.txt > output2.txt");
        if (!compareFiles("output1.txt", "output2.txt")) {
            cerr << "\nWrong Answer!\nError found in test case " << testcase <<
endl;
            cerr << "Input:\n"; system("type input.txt");
            cerr << "Test output:\n"; system("type output1.txt");
```

```
            cerr << "Brute output:\n"; system("type output2.txt");
            break;
        } else {
            cout << "Test case " << testcase++ << " accepted." << endl;
        }
    }
    return 0;
}
```

Vscode On Linux

```
// 下载编译器
sudo apt install g++/gcc/pypy3/python3
// 编译环境
g++ -std=c++17 -o2 -Wall demo.cpp
// out文件运行
./demo.out
// Bash脚本快速编译 run.sh
#/bin/bash
g++ -std=c++17 -o2 -Wall $1.cpp -o $1.main
./$1.main < input.txt > output.txt # 输入input.txt中的样例，将结果输出在output.txt中
cat output.txt
# 给定权限 chmod +x run.sh
```

Kruscal

```
const int N = 2e5+10;
int cnt = 0;
int n,m;
struct node
{
    int u,v,w;
    bool operator<(const node &another) const
    {
        return w < another.w;
    }
}edge[N];
//无向边加边
void add(int u,int v,int w)
{
    cnt++;
    edge[cnt].u = u;
    edge[cnt].v = v;
    edge[cnt].w = w;
}
struct DSU
{ ... ...  };
void Kruscal(DSU& dsu)
{
    int count = 0;//已经加入树的边数
    int ans = 0;//已经加入的总最小边权和
    for(int i=1;i<=cnt;i++)
    {
        int xr = dsu.find(edge[i].u);
```

```
        int yr = dsu.find(edge[i].v);
        if(xr!=yr)
        {
            count++;//边数+1
            ans += edge[i].w;
            dsu.merge(xr,yr);
        }
        if(count == n-1)//已建成最小生成树
        {
            cout << ans << endl;
            return;
        }
    }
    //否则则说明该图不连通
    cout << "orz" << endl;
}
void solve()
{
    cin >> n >> m;
    DSU dsu(n);
    for(int i=1;i<=m;i++)
    {
        int x,y,w;
        cin >> x >> y >> w;
        add(x,y,w);//无向边建图
    }
    sort(edge+1,edge+1+m);
    Kruscal(dsu);
}
```

Topological Sort and Longest Path on DAG

```
const int N = 1550;
const int M = 5e4+100;
int head[N];
int indegree[N];
int cnt = 0;
struct node
{
    int to,next,w;
}edge[M];
void add(int u,int v,int w)
{
    cnt++;
    edge[cnt].to = v;
    edge[cnt].w = w;
    edge[cnt].next = head[u];
    head[u] = cnt;
}
void solve()
{
    int n,m;
    cin >> n >> m;
    for(int i=1;i<=m;i++)
    {
```

```
        int u,v,w;
        cin >> u >> v >> w;
        indegree[v]++;
        add(u,v,w);
    }
    queue<int> q;
    for(int i=1;i<=n;i++) if(indegree[i]==0) q.push(i);
    vector<int> topo;//用于存储拓扑序列
    topo.push_back(0);
    //拓扑排序
    while(!q.empty())
    {
        int t = q.front();
        q.pop();
        topo.push_back(t);
        for(int i=head[t];i;i=edge[i].next)
        {
            int v = edge[i].to;
            indegree[v]--;
            if(indegree[v]==0)
            {
                q.push(v);
            }
        }
    }
    int dp[n+1];
    //可求解存在负边权的情况
    for(int i=1;i<=n;i++)
    {
        dp[i] = -INF;
    }
    dp[1] = 0;
    for(int i=1;i<=n;i++)
    {
        int u = topo[i];
        for(int i=head[u];i;i=edge[i].next)
        {
            int v = edge[i].to;
            int w = edge[i].w;
            dp[v] = max(dp[v],dp[u] + w);
        }
    }
    if(dp[n]==-INF)
    {
        cout << -1;
    }
    else
    cout << dp[n];
}
```

Bellman-Ford

Bellman-Ford算法，用于解决可以有负边权，但没有负环的单源最短路问题

时间复杂度 O(V*E)

适用于顶点较少的情况

松弛操作轮数一定小于等于 n - 1 若大于此轮次，说明从某个点出发有负环

```
//求解从1到n号节点的、最多经过k条边的最短距离，图中存在负边权(有向图)
const int N = 505;
const int M = 1e5+100;
//此算法无需存图，只需存边
struct node
{
    int u,v,w;
}edge[M];
int cnt = 0;
void add(int u,int v,int w)
{
    edge[++cnt].u = u;
    edge[cnt].v = v;
    edge[cnt].w = w;
}
int dis[N],backup[N];//dis[i]表示从1开始到达第i点的最短距离
int n,m,k;
void bellmanford()
{
    for(int i=1;i<=n;i++)
    {
        dis[i] = INF;
    }
    dis[1] = 0;
    //最多经过K轮松弛操作
    for(int i=1;i<=k;i++)
    {
        memcpy(backup,dis,sizeof dis);
        for(int j=1;j<=m;j++)
        {
            auto [u,v,w] = edge[j];
            dis[v] = min(dis[v],backup[u]+w);
        }
    }
}
void solve()
{
    cin >> n >> m >> k;
    for(int i=1;i<=m;i++)
    {
        int u,v,w;
        cin >> u >> v >> w;
        add(u,v,w);
    }
```

```
    bellmanford();
    if(dis[n]>INF/2)
    {
        cout << -1;
        return;
    }
    cout << dis[n];
}
```

Tarjan - SCC

```
const int M = 1e5+100; // 使用前需初始化最大边数
struct Edge
{
    int to,next;
};
// Tarjan 算法来寻找有向图中的强连通分量，并进行缩点。
struct SCC {
    int n;
    vector<int> head;
    vector<Edge> edge;
    int edge_cnt;

    vector<int> stk;            // 栈，用于存储当前DFS路径上的节点
    vector<int> dfn, low, bel;  // dfn：访问时间戳；low：能追溯到的最早时间戳；bel：所属强连通分量编号
    int cur, cnt;               // cur：时间戳计数器；cnt：强连通分量计数器 计数范围[1,scc.cnt]
    SCC() {}
    SCC(int n) {init(n);}

    // 初始化图和变量
    void init(int n) {
        this->n = n;
        head.assign(n + 1, 0); // 初始化为0表示没有边
        edge.resize(M);        // M 为边的最大数量 为防止超时需手动定义
        edge.clear();
        edge_cnt = 0;

        dfn.assign(n + 1, -1);  // -1表示未访问
        low.resize(n + 1);
        bel.assign(n + 1, -1);  // -1表示未分配分量
        stk.clear();
        cur = 0;
        cnt = 1;
    }
    void addEdge(int u, int v) {
        edge[++edge_cnt].to = v;
        edge[edge_cnt].next = head[u];
        head[u] = edge_cnt;
    }
    // 深度优先搜索，计算强连通分量
    void dfs(int x) {
        dfn[x] = low[x] = cur++;  // 初始化时间戳
        stk.push_back(x);         // 将当前节点压入栈中
```

```
        // 遍历所有邻接节点
        for (int i = head[x]; i ; i = edge[i].next) {
            int y = edge[i].to;
            if (dfn[y] == -1) {   // 如果y未被访问
                dfs(y);           // 递归访问y
                low[x] = min(low[x], low[y]);  // 更新x的low值
            } else if (bel[y] == -1) {  // 如果y已访问但还未分配分量（在栈中）
                low[x] = min(low[x], dfn[y]);  // 更新x的low值
            }
        }

        // 如果当前节点的dfn等于low，说明它是一个强连通分量的根
        if (dfn[x] == low[x]) {
            int y;
            do {
                y = stk.back();
                bel[y] = cnt;     // 将栈中节点弹出并分配到当前分量
                stk.pop_back();
            } while (y != x);     // 直到弹出当前节点x为止
            cnt++;                // 分量计数器加1
        }
    }
    // 计算并返回每个节点所属的强连通分量
    vector<int> work() {
        for (int i = 1; i <= n; i++) {  // 从1开始遍历到n
            if (dfn[i] == -1) {         // 如果节点i未被访问
                dfs(i);                 // 从节点i开始DFS
            }
        }
        return bel;
    }
};

void solve()
{
    // 例题 : 给定原始图的点权，求缩点后各强连通分量组成的DAG的最长路
    int n,m;
    cin >> n >> m;
    SCC scc(n);
    vector<int> val(n+1);
    for(int i=1;i<=n;i++) cin >> val[i];
    for(int i=1;i<=m;i++)
    {
        int u,v;
        cin >> u >> v;
        scc.addEdge(u,v);
    }
    // 得到原图各点属于哪个SCC
    vector<int> comp = scc.work();
    // 用于存储各个强连通分量的权值，即原图中属于各个强连通分量的权值总和
    // 由于SCC下标从1开始，scc.cnt中已经添加了0下标时默认的comp[-1],即scc.cnt =
scc.cnt(real) + 1
    vector<int> comp_val(scc.cnt,0);
    for(int i=1;i<=n;i++)
    {
```

```
        comp_val[comp[i]] += val[i];
    }

    vector<int> head(scc.cnt,0);
    int cnt = 0;
    vector<Edge> edge(M);
    auto add = [&](int u,int v)
    {
        edge[++cnt].to = v;
        edge[cnt].next = head[u];
        head[u] = cnt;
    };
    // 构建新图 DAG
    vector<vector<bool>> added(scc.cnt,vector<bool>(scc.cnt,false));
    vector<int> ind(scc.cnt,0);
    for(int u=1;u<=n;u++)
    {
        for(int i=scc.head[u];i;i=scc.edge[i].next)
        {
            int v = scc.edge[i].to;
            int cu = comp[u];
            int cv = comp[v];
            if(cu != cv && !added[cu][cv])
            {
                add(cu,cv);
                ind[cv]++;
                added[cu][cv] = true;
            }
        }
    }
    // Toposort + DP 求解最长路
    queue<int> q;
    for(int i=1;i<=scc.cnt;i++)
    {
        if(ind[i]==0) q.push(i);
    }
    vector<int> dp(scc.cnt,0);
    while(!q.empty())
    {
        int u = q.front();
        q.pop();
        dp[u] += comp_val[u];
        for(int i=head[u];i;i=edge[i].next)
        {
            int v = edge[i].to;
            dp[v] = max(dp[v],dp[u]);
            if(--ind[v]==0)
            {
                q.push(v);
            }
        }
    }
    int ans = *max_element(dp.begin(),dp.end());
    cout << ans << endl;
}
```

EBCC

```
/*
 * 边双连通分量(EBCC)模板 时间复杂度O(N*M)
 * 用于求解无向图中的边双连通分量
 * 边双连通分量是指任意两点之间至少存在两条边不重复的路径的最大子图，删除任意一条边不会影响连通性
 ** 具体应用题目
 * 割边(桥)的定义是删除后会增加图的连通分量数的边，原图中的边若连接不同分量，则为割边
 * 添加最少的边，使图变为边双连通图(即不存在割边)，求出所有边双连通分量并缩点为树，最少边数 = (叶子节点数 + 1) / 2
 * 判断从点 s 到 t 的所有路径是否必须经过某条边 (u, v) <- 若为割边则必经
*/
struct EBCC
{
    int n;                      // 节点数量
    vector<vector<int>> adj;    // 邻接表存储图结构
    vector<int> stk;            // 栈，用于记录DFS路径
    vector<int> dfn, low, bel;  // dfn: 时间戳; low: 追溯值; bel: 所属分量编号
    int cur, cnt;               // cur: 当前时间戳; cnt: 分量计数器
    // 默认构造函数
    EBCC() {}
    // 带参数的构造函数，初始化图
    EBCC(int n) {
        init(n);
    }
    // 初始化图结构
    void init(int n) {
        this->n = n;
        adj.assign(n, {});
        dfn.assign(n, -1);      // 初始化为-1表示未访问
        low.resize(n);
        bel.assign(n, -1);      // 初始化为-1表示未分配分量
        stk.clear();
        cur = cnt = 0;          // 初始化时间戳和分量计数器
    }
    // 添加无向边
    void addEdge(int u, int v) {
        adj[u].push_back(v);
        adj[v].push_back(u);
    }
    // 深度优先搜索，计算dfn和low数组
    void dfs(int x, int p) {
        dfn[x] = low[x] = cur++;  // 分配时间戳
        stk.push_back(x);          // 将当前节点压入栈中
        for (auto y : adj[x]) {
            if (y == p) {          // 跳过父节点
                continue;
            }
            if (dfn[y] == -1) {  // 处理未访问的邻居
                dfs(y, x);
                low[x] = min(low[x], low[y]);  // 更新low值
            } else if (bel[y] == -1) {  // 处理已访问但未分配分量的邻居(回边)
                low[x] = min(low[x], dfn[y]);  // 更新low值
            }
```

```
        }
        // 找到边双连通分量的根节点
        if (dfn[x] == low[x]) {
            int y;
            do {
                y = stk.back();
                bel[y] = cnt;    // 分配分量编号
                stk.pop_back();
            } while (y != x);
            cnt++;              // 分量编号递增
        }
    }
    // 计算所有边双连通分量
    vector<int> work() {
        for (int i = 0; i < n; i++) {
            if (dfn[i] == -1) {  // 处理所有连通块
                dfs(i, -1);
            }
        }
        return bel;             // 返回每个节点所属的分量
    }
    // compress()方法可生成缩点后的树，树的边表示原图中的割边
    struct Graph {
        int n; // 缩点后的节点数（边双连通分量的数量）
        vector<pair<int,int>> edges;// 缩点后的树的边集（每条边对应原图的一条割边）
        vector<int> siz;// 每个分量的原始节点数
        vector<int> cnte;// 每个分量内部的边数
    };
    Graph compress() {
        Graph g;
        g.n = cnt;
        g.siz.resize(cnt);
        g.cnte.resize(cnt);
        for (int i = 0; i < n; i++) {
            g.siz[bel[i]]++;
            for (auto j : adj[i]) {
                if (bel[i] < bel[j]) {
                    g.edges.emplace_back(bel[i], bel[j]);
                } else if (i < j) {
                    g.cnte[bel[i]]++;
                }
            }
        }
        return g;
    }
};
int main()
{
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    int n, m;
    cin >> n >> m;
    EBCC ebcc(n);
    // 存储所有边用于后续判断
    vector<pair<int, int>> edges(m);
```

```
    for (int i = 0; i < m; i++) {
        int a, b;
        cin >> a >> b;
        // 题目中节点编号从1开始，转换为0开始
        a--; b--;
        edges[i] = {a,b};
        ebcc.addEdge(a,b);
    }
    // 计算每个节点所属的边双连通分量
    vector<int> component = ebcc.work();
    // 收集所有割边
    vector<pair<int, int>> bridges;
    for (auto [u, v] : edges)
    {
        if (component[u] != component[v])
        {
            // 转换回题目中的节点编号(1-based)并排序
            int a = min(u,v) + 1;
            int b = max(u,v) + 1;
            bridges.emplace_back(a, b);
        }
    }
    sort(bridges.begin(), bridges.end());
    for (auto [a, b] : bridges) {
        cout << a << " " << b << endl;
    }
    return 0;
}
```

Find Tree's Centroids

树的重心有如下三种定义，求出的点是一样的：

```
    1. 以某个节点为根，最大子树的节点数最少，此节点为重心
    2.  以某个节点为根，每颗子树的节点数不超过总节点数的一半，此节点是重心，可用于得到重心数量
(通过dfs遍历maxsub <= n/2)
    3.  以某个节点为根，所有节点都走向重心的总边数最少，此节点为重心
```

常用性质:

```
    1. 一棵树最多有两个重心，且此两个节点必相邻
    2.  如果树上增加/删除一个叶节点，转移后的重心最多移动一条边
    3.  如果把两棵树连起来，那么新树的重心一定在原来两棵树重心的路径上
    4.  树上的边权如果都>=0，不管边权怎么分布，所有节点都走向重心的总距离和最小
```

```
struct node
{
    int to,next;
};
// 例题：增删边使得该棵树重心唯一
void solve()
{
```

```
    int n;
    cin >> n;
    int cnt = 0;
    vector<int> center; // 存储重心，最多为两个
    vector<int> head(n+1,0);
    vector<int> maxsub(n+1,0);  // 以 i 节点为根的最大子树节点数量
    vector<int> size(n+1,1); // size[i]表示以节点i为根时，最大子树的节点数
    vector<node> edge(n<<1);
    auto add = [&](int u,int v)
    {
        edge[++cnt].to = v;
        edge[cnt].next = head[u];
        head[u] = cnt;
    };
    for(int i=1;i<=n-1;i++)
    {
        int u,v;
        cin >> u >> v;
        add(u,v);
        add(v,u);
    }
    auto dfs = [&](auto &&self,int u,int fa) -> void
    {
        size[u] = 1; // 默认的子树节点 有时可自定义树枝定义，并非一定为1
        // 以当前节点u为节点，最大的子树有多少节点
        maxsub[u] = 0;
        for(int i = head[u];i;i = edge[i].next)
        {
            int v = edge[i].to;
            if(v != fa)
            {
                self(self,v,u);
                size[u] += size[v];
                maxsub[u] = max(maxsub[u],size[v]);
            }
        }
        maxsub[u] = max(maxsub[u],n-size[u]);
    };
    dfs(dfs,1,-1);
    for(int i=1;i<=n;i++)
    {
        if(maxsub[i] <= n/2)
        {
            center.push_back(i);
        }
    }
    if(center.size()==1)
    {
        cout << center[0] << ' ' <<  edge[head[center[0]]].to << endl;
        cout << center[0] << ' ' << edge[head[center[0]]].to << endl;
    }
    else
    {
        int leaf;
        int leaffa;
```

```
        auto findleaf = [&](auto &&self,int u,int fa) -> void
        {
            for(int i=head[u];i;i=edge[i].next)
            {
                int v = edge[i].to;
                if(v != fa)
                {
                    self(self,v,u);
                    return;
                }
            }
            leaf = u;
            leaffa = fa;
        };
        findleaf(findleaf,center[1],center[0]);
        cout << leaf << ' ' << leaffa << endl;
        cout << center[0] << ' ' << leaf << endl;
    }
}
```

Find Tree's Diameter

树上距离最远的两个点，形成的路径叫做树的直径

如果树上边权都为**正**，有以下直径相关结论：

*如果有多条直径，那么这些直径一定有共同的中间部分，可能是一个公共点或公共路径

*树上任意一点，相隔最远的点的集合，直径的两端点至少有一个在其中

```
struct node
{
    int to,next,w;
};
// 求法一，两次DFS，仅使用于边权非负情况，但沿途可以得到更多信息
void solve1()
{
    // 例题：求出树的直径，找到所有直径的公共边数量
    int n;
    cin >> n;
    int cnt = 0;
    vector<int> head(n+1,0);
    vector<node> edge(n<<1);
    auto add = [&](int u,int v,int w) -> void
    {
        edge[++cnt].to = v;
        edge[cnt].w = w;
        edge[cnt].next = head[u];
        head[u] = cnt;
    };
    for(int i=1;i<=n-1;i++)
    {
        int u,v,w;
        cin >> u >> v >> w;
        add(u,v,w);
        add(v,u,w);
```

```
    }
    int start = 1; // 直径开始点
    int end = 1; // 直径结束点
    int diameter; // 直径长度
    vector<int> dis(n+1,0); // 从规定的头节点出发，走到i的距离
    vector<int> last(n+1,0); // 从规定的头节点出发，到达i节点的上一个节点
    auto dfs = [&](auto &&self,int u,int fa) -> void
    {
        last[u] = fa;
        for(int i=head[u];i;i=edge[i].next)
        {
            int v = edge[i].to;
            int w = edge[i].w;
            if(v != fa)
            {
                dis[v] = dis[u] + w;
                self(self,v,u);
            }
        }
    };
    dis[start] = 0; // 初始化起点dis为0
    dfs(dfs,start,-1); // 第一次DFS
    for (int i=1;i<=n;i++)
    {
        if(dis[i] > dis[start]) start = i; // 找到与默认初始点最远的点，此点即直径起点
    }
    dis[start] = 0;
    dfs(dfs,start,-1); // 第二次DFS
    for (int i=1;i<=n;i++)
    {
        if (dis[i] > dis[end]) end = i; // 找到直径终点
    }
    diameter = dis[end]; // 求到直径
    // 求所有直径的公共部分，返回公共部分有几条边
    vector<bool> diameterpath(n+1,false); // 记录哪些点是当前直径经过的点
    diameterpath[start] = true;
    for(int i=end;i!=start;i=last[i]) diameterpath[i] = true;
    int l = start; // 直径公共部分的左边界节点
    int r = end; // 直径公共部分的右边界节点
    int maxDist;
    int ans;
    // 不能走向直径路径上的节点，能走出的最大距离
    auto FindLongestDistance = [&](auto &&self,int u,int fa,int curdis) -> int
    {
        int ans = curdis;
        for(int i=head[u];i;i=edge[i].next)
        {
            int v = edge[i].to;
            int w = edge[i].w;
            if(v != fa && !diameterpath[v])
            {
                ans = max(ans,self(self,v,u,curdis+w));
            }
        }
        return ans;
```

```
    };
    for(int i=last[end];i!=start;i=last[i])
    {
        maxDist = FindLongestDistance(FindLongestDistance,i,0,0);
        if(maxDist == diameter - dis[i]) r = i;
        if(maxDist == dis[i] && l == start) l = i;
        if(l == r) ans = 0;
        else
        {
            ans = 1;
            for(int i=last[r];i!=l;i=last[i]) // 左右边界之间的所有边都是公共边
            {
                ans++;
            }
        }
    }
    cout << diameter << endl;
    cout << ans << endl;
}
// 求法二：一次DFS，利用树形DP思想，但仅能收集到树的直径长度
void solve2()
{
    int n;
    cin >> n;
    int cnt = 0;
    vector<int> head(n+1,0);
    vector<node> edge(n<<1);
    auto add = [&](int u,int v,int w) -> void
    {
        edge[++cnt].to = v;
        edge[cnt].w = w;
        edge[cnt].next = head[u];
        head[u] = cnt;
    };
    for(int i=1;i<=n-1;i++)
    {
        int u,v,w;
        cin >> u >> v >> w;
        add(u,v,w);
        add(v,u,w);
    }
    vector<int> dis(n+1,0); // 从u开始必须往下走，能走出的最大距离，可以不选任何边
    vector<int> ans(n+1,0); //  路径必须包含点u的情况下，最大路径和
    auto dfs = [&](auto &&self,int u,int fa) -> void
    {
        for (int i = head[u];i;i=edge[i].next)
        {
            int v = edge[i].to;
            int w = edge[i].w;
            if (v != fa)
            {
                self(self,v,u);
                ans[u] = max(ans[u], dis[u] + dis[v] + w);
                dis[u] = max(dis[u], dis[v] + w);
            }
```

```
        }
    };
    dfs(dfs,1,-1);
    int diameter = NINF;
    for(int i=1;i<=n;i++) diameter = max(diameter,ans[i]);
    cout << diameter << endl;
}
```

Diff On Tree

```
const int N = 5e4 + 100;
const int M = 0;
int head[N];
int cnt = 0;
struct node
{
    int to,next,w;
}edge[N<<1];
void add(int u,int v,int w)
{
    edge[++cnt].to = v;
    edge[cnt].w = w;
    edge[cnt].next = head[u];
    head[u] = cnt;
}
int st[N][32];
int depth[N];
int maxpow;
int n,m,root;
void dfs(int u,int fa)
{
    depth[u] = (fa == -1) ? 0 : (depth[fa] + 1);
    st[u][0] = fa;
    for(int p = 1;p<=maxpow;p++)
    {
        st[u][p] = (st[u][p-1] == -1) ? -1 : st[st[u][p-1]][p-1];
    }
    for(int i=head[u];i;i=edge[i].next)
    {
        int v = edge[i].to;
        if(v!=fa)
        {
            dfs(v,u);
        }
    }
}
int LCA(int x,int y)
{
    if(depth[x] > depth[y]) swap(x,y);
    while(depth[x] != depth[y]) y = st[y][(int)log2(depth[y]-depth[x])];
    if(x==y) return x;
    for(int i = log2(depth[x]);i>=0;i--)
    {
        if(st[x][i]!=st[y][i])
        {
```

```
            x = st[x][i];
            y = st[y][i];
        }
    }
    return st[x][0];
}
void solve1()
{
    int n,m;
    cin >> n >> m;
    vector<int> val(n+1,0);
    maxpow = log2(n) + 1;
    for (int i = 1; i <= n - 1; i++)
    {
        int u, v, w;
        cin >> u >> v;
        w = 0;
        add(u, v, w);
        add(v, u, w);
    }
    dfs(1,-1);
    for(int i=1;i<=m;i++)
    {
        int x,y,k;
        k = 1;
        cin >> x >> y;
        int lca = LCA(x,y);
        int lcafa = (st[lca][0] == -1) ? 0 : st[lca][0];
        val[x]+=k;
        val[y]+=k;
        val[lca]-=k;
        val[lcafa]-=k;
    }
    auto dfs1 = [&](auto &&self, int u, int fa) -> void
    {
        for(int i=head[u]; i; i=edge[i].next)
        {
            int v = edge[i].to;
            if(v != fa)
            {
                self(self, v, u);
                val[u] += val[v];  // 合并状态转移到递归返回后
            }
        }
    };
    dfs1(dfs1,1,0);
    int ans = 0;
    for(int i=1;i<=n;i++) ans = max(ans,val[i]);
    cout << ans << endl;
}
// 树上边差分
void solve2()
{
    int n,m;
    cin >> n >> m;
```

```cpp
    vector<int> val(n+1,0);
    maxpow = log2(n) + 1;
    cnt = 0;
    for (int i = 1; i <= n - 1; i++)
    {
        int u, v, w;
        cin >> u >> v >> w;
        add(u, v, w);
        add(v, u, w);
    }
    dfs(1, -1);
    for(int i=1;i<=m;i++)
    {
        int x, y, k;
        cin >> x >> y >> k;
        int lca = LCA(x, y);
        val[x] += k;
        val[y] += k;
        val[lca] -= 2 * k;
    }
    auto dfs1 = [&](auto &&self, int u, int fa) -> void
    {
        for(int i = head[u]; i; i = edge[i].next)
        {
            int v = edge[i].to;
            int &w = edge[i].w;
            if(v != fa)
            {
                self(self, v, u);
                w += val[v];
                val[u] += val[v];
            }
        }
    };
    dfs1(dfs1, 1, -1);
    int ans = 0;
    auto dfs2 = [&](auto &&self, int u, int fa) -> void
    {
        for(int i = head[u]; i; i = edge[i].next)
        {
            int v = edge[i].to;
            int w = edge[i].w;
            if(v != fa)
            {
                self(self, v, u);
                ans = max(ans,w);
            }
        }
    };
    dfs2(dfs2,1,-1);
    cout << ans << endl;
}
```

Chunking Algorithms

```
// 利用分块思想实现单点修改，区间和查询的树状数组模板
void solve1()
{
    int n,m;
    cin >> n >> m;
    vector<int> arr(n+1);
    for(int i=1;i<=n;i++) cin >> arr[i];
    vector<int> l(n+1);
    vector<int> r(n+1); // 每个块左右端点
    vector<int> belong(n+1); // 第i个元素属于哪个块
    vector<int> sum(n+1);// 表示每个块块内的和是多少
    vector<int> pre(n+1);// 表示块的前缀和
    int len, bcnt;     // 块的长度和块的个数
    auto init = [&]() -> void //O(n)
    {
        len = sqrtl(n); // 默认块的长度为sqrt(n)
        bcnt = (n - 1) / len + 1; // 块的个数则为sqrt(n) + 1
        for(int i=1;i<=bcnt;i++) // 初始化每个块的左右端点
        {
            l[i] = r[i-1] + 1;
            r[i] = i * len;
        }
        r[bcnt] = n;
        for(int i=1;i<=bcnt;i++)
        {
            for(int j=l[i];j<=r[i];j++)
            {
                belong[j] = i; //初始化每个值属于哪个块
                sum[i] += arr[j];
            }
            pre[i] = pre[i-1] + sum[i];
        }
    };
    // 在第k个元素位置上加上x
    auto update = [&](int k,int x) -> void // O(sqrt(n))
    {
        // 在第 belong[k] 块内加上x，则当前即之后的所有块的前缀和都要加上x
        sum[belong[k]] += x;
        arr[k] += x;
        for(int i=belong[k];i<=bcnt;i++)
        {
            pre[i] += x;
        }
    };
    auto query = [&](int ql,int qr) -> int // O(sqrt(n))
    {
        int left = belong[ql],right = belong[qr]; // 记录询问的左右端点分别在哪一块
        int ans = 0;
        if(left == right) // 如果它们在同一块，直接暴力求值
        {
            for(int i=ql;i<=qr;i++) ans += arr[i];
            return ans;
        }
```

```
        ans = pre[right - 1] - pre[left]; //  右端点左边第一个块 和 左端点右边第一个块
之间的前缀和
        for(int i=ql;i<=r[left];i++) ans += arr[i]; // 暴力更新 剩余左右端点块内的元素
        for(int i=qr;i>=l[right];i--) ans += arr[i];
        return ans;
    };
    init();
}
// 利用分块思想实现区间修改，区间和查询的线段树模板
void solve2()
{
    int n,m;
    cin >> n >> m;
    vector<int> arr(n+1);
    for(int i=1;i<=n;i++) cin >> arr[i];
    vector<int> l(n+1);
    vector<int> r(n+1); // 每个块左右端点
    vector<int> belong(n+1); // 第i个元素属于哪个块
    vector<int> sum(n+1);// 表示每个块块内的和是多少
    vector<int> pre(n+1);// 表示块的前缀和
    vector<int> add(n+1);// 懒标记数组，表示第i个块被加了多少
    int len, bcnt;     // 块的长度和块的个数
    auto init = [&]() -> void //O(n)
    {
        len = sqrtl(n); // 默认块的长度为sqrt(n)
        bcnt = (n - 1) / len + 1; // 块的个数则为sqrt(n) + 1
        for(int i=1;i<=bcnt;i++) // 初始化每个块的左右端点
        {
            l[i] = r[i-1] + 1;
            r[i] = i * len;
        }
        r[bcnt] = n;
        for(int i=1;i<=bcnt;i++)
        {
            for(int j=l[i];j<=r[i];j++)
            {
                belong[j] = i; //初始化每个值属于哪个块
                sum[i] += arr[j];
            }
            pre[i] = pre[i-1] + sum[i];
        }
    };
    // 在第k个元素位置上加上x
    auto update = [&](int ul,int ur,int k) -> void // O(sqrt(n))
    {
        int bl = belong[ul],br = belong[ur];
        if(bl == br)
        {
            for(int i=ul;i<=ur;i++) arr[i] += k,sum[bl] += k;
            for(int i=bl;i<=bcnt;i++) pre[i] = pre[i-1] + sum[i];
            return;
        }
        for(int i=ul;i<=r[bl];i++) arr[i] += k,sum[bl] += k;
        for(int i=bl+1;i<=br-1;i++) add[i] += k,sum[i] += (r[i] - l[i] + 1) * k;
// 代表这个块整体被加上了K
```

```
        for(int i=ur;i>=l[br];i--) arr[i] += k,sum[br] += k;
        for(int i=bl;i<=bcnt;i++) pre[i] = pre[i-1] + sum[i];

    };
    auto query = [&](int ql,int qr) -> int // O(sqrt(n))
    {
        int bl = belong[ql],br = belong[qr]; // 记录询问的左右端点分别在哪一块
        int ans = 0;
        if(bl == br) // 如果它们在同一块，直接暴力求值
        {
            for(int i=ql;i<=qr;i++) ans += arr[i] + add[bl]; // 记得加上懒更新数组
            return ans;
        }
        ans = pre[br - 1] - pre[bl]; //  右端点左边第一个块 和 左端点右边第一个块之间的前缀和
        for(int i=ql;i<=r[bl];i++) ans += arr[i], ans += add[bl]; // 暴力更新 剩余左右端点块内的元素
        for(int i=qr;i>=l[br];i--) ans += arr[i], ans += add[br];
        return ans;
    };
    init();
}
```

Mo's Algorithm

小B 有一个长为 n 的整数序列 a，值域为 $[1,k]$。
他一共有 m 个询问，每个询问给定一个区间 $[l,r]$，求：

$$\sum\limits_{i=1}^k c_i^2$$

其中 c_i 表示数字 i 在 $[l,r]$ 中的出现次数。
小B请你帮助他回答询问。

```
struct query
{
    int l,r,id;
};
void solve()
{
    int n, m, k;
    cin >> n >> m >> k;
    vector<int> arr(n+1);
    vector<int> belong(n+1);
    vector<int> cl(n+1);
    vector<int> cr(n+1);
    vector<query> q(m+1);
    vector<int> ans(m+1);
    for(int i=1;i<=n;i++) cin >> arr[i];
    int len, bcnt;
    auto init = [&]() -> void //O(n)
    {
        len = sqrtl(n); // 默认块的长度为sqrt(n)
        bcnt = (n - 1) / len + 1; // 块的个数则为sqrt(n) + 1
        for(int i=1;i<=bcnt;i++) // 初始化每个块的左右端点
        {
```

```
            cl[i] = cr[i-1] + 1;
            cr[i] = i * len;
        }
        cr[bcnt] = n;
        for(int i=1;i<=bcnt;i++)
        {
            for(int j=cl[i];j<=cr[i];j++)
            {
                belong[j] = i; //初始化每个值属于哪个块
            }
        }
    };
    init();
    for(int i=1;i<=m;i++)
    {
        cin >> q[i].l >> q[i].r;
        q[i].id = i;
    }
    sort(q.begin()+1, q.end(), [&](query& a,query& b)
    {
        if(belong[a.l] != belong[b.l]) return belong[a.l] < belong[b.l];
        return (belong[a.l] % 2 == 1) ? (a.r < b.r) : (a.r > b.r);
    });
    int l=1,r=0; // 初始化双指针
    int res = 0;
    vector<int> cnt(k+1,0); //维护窗口内数字i出现的次数
    auto Add = [&](int x) ->void
    {
        int val = arr[x];
        res -= cnt[val] * cnt[val]; // 减去旧的平方贡献
        cnt[val]++;
        res += cnt[val] * cnt[val]; // 加上新的平方贡献
    };
    auto Sub = [&](int x) ->void
    {
        int val = arr[x];
        res -= cnt[val] * cnt[val]; // 减去旧的平方贡献
        cnt[val]--;
        res += cnt[val] * cnt[val]; // 加上新的平方贡献
    };
    for(int i=1;i<=m;i++)
    {
        while(q[i].l < l) Add(--l); // 向左扩张窗口，并更新贡献
        while(q[i].r > r) Add(++r); // 向右扩张窗口，并更新贡献
        while(q[i].l > l) Sub(l++); // 向右收缩窗口，并更新贡献
        while(q[i].r < r) Sub(r--); // 向左收缩窗口，并更新贡献
        // 记录答案
        ans[q[i].id] = res;
    }
    for(int i=1;i<=m;i++) cout << ans[i] << endl;
}
```