Algorithm__Template\Data__Structure.cpp

```cpp
/**
 * 树状数组 BIT
 * 线段树  SegTree
 * 并查集 DisjointSet
 * 最近公共祖先 LCA
 * 分块 block
 * ST表 st
 *
*/
#include <iostream>
#include <vector>
#include <string>
#include <cstring>
#include <set>
#include <map>
#include <queue>
#include <ctime>
#include <random>
#include <sstream>
#include <numeric>
#include <stdio.h>
#include <algorithm>
using namespace std;

#define rep(i,x,n) for(int i = x; i <= n; i++)

typedef long long LL;
typedef pair<int,int> PII;

const int INF = 0x3f3f3f3f;
const int N = 1e5 + 21;

namespace golitter {
namespace BIT {
    /**  url: https://ac.nowcoder.com/acm/contest/61132/L
        底部确定，顶部无穷大
        最外面的结点是2的n次方，如上图1，2，4，8的结点
        奇数的结点一定是叶子结点
        数组一定要从1开始


        树状数组 离线处理
            查询种类数，维护区间内仅有一个或两个此种元素
            一般都考虑离线询问
            将询问的区间按照右端点小在前排序

    */
const int N = 5e5 + 21;
int n,m;
int tr[N];
int lowbit(int x) {
    return x & -x;
}
void add(int x, int c) {
    // if(!x) return; // 如果 树状数组 离线处理 记得**
https://www.luogu.com.cn/problem/P4113
```

```cpp
    for(; x < N; x += lowbit(x)) tr[x] += c;
}
LL sum(int x) {
    LL res = 0;
    for(; x; x -= lowbit(x)) res += tr[x];
    return res;
}
void solve() {
    cin>>n>>m;
    rep(i,1,n) {
        int x; cin>>x;
        add(i, x);
    }
    rep(i,1,m) {
        int a,b,c; cin>>a>>b>>c;
        if(a == 1) {
            add(b,c);
        } else {
            cout<<sum(c) - sum(b-1)<<endl;
        }
    }
}

}}

namespace golitter {
namespace SegTree {
    /**
 * 小区间的值更新大区间的值
 * 问题满足：区间加法： [l, r] 可以用 [l, mid] 和 [mid+1, r]的值产生
 *      不满足的问题：区间的众数，区间最长连续问题，最长不下降问题
 *
 * 解题步骤： 建树，
*/

int w[N],n,m; // 注意 w[N] 开LL ( https://www.luogu.com.cn/problem/P2357
struct adt {
    int l,r;
    LL sum,add;
}tr[N << 2];
// 左子树
inline int ls(int p) {return p<<1; }
// 右子树
inline int rs(int p) {return p<<1|1; }
// 向上更新
void pushup(int u) {
    tr[u].sum = tr[ls(u)].sum + tr[rs(u)].sum;
}
// 向下回溯时，先进行更新
void pushdown(int u) { // 懒标记，该节点曾经被修改，但其子节点尚未被更新。
    auto &root = tr[u], &right = tr[rs(u)], &left = tr[ls(u)];
    if(root.add) {
        right.add += root.add; right.sum += (LL)(right.r - right.l + 1)*root.add;
        left.add += root.add; left.sum += (LL)(left.r - left.l + 1)*root.add;
        root.add = 0;
    }

}
// 建树
void build(int u, int l, int r) {
```

```cpp
    if(l == r) tr[u] = {l, r, w[r], 0};
    else {
        tr[u] = {l,r}; // 容易忘
        int mid = l + r >> 1;
        build(ls(u), l, mid), build(rs(u), mid + 1, r);
        pushup(u);
    }
}
// 修改
void modify(int u, int l, int r, int d) {
    if(tr[u].l >= l && tr[u].r <= r) {
        tr[u].sum += (LL)(tr[u].r - tr[u].l + 1)*d;
        tr[u].add += d;
    }
    else {
        pushdown(u);
        int mid = tr[u].l + tr[u].r >> 1;
        if(l <= mid) modify(ls(u), l ,r, d);
        if(r > mid) modify(rs(u), l, r, d);
        pushup(u);
    }
}
// 查询
LL query(int u, int l, int r) {
    if(tr[u].l >= l && tr[u].r <= r) {
        return tr[u].sum;
    }
    pushdown(u);
    int mid = tr[u].l + tr[u].r >> 1;
    LL sum(0);
    if(l <= mid) sum = query(ls(u), l, r);
    if(r > mid ) sum += query(rs(u), l, r);
    return sum;
}
void solve() {
    cin>>n>>m;
    for(int i = 1; i <= n; ++i) cin>>w[i];
    build(1, 1, n);
    // cout<<tr[1].sum<<endl;
    int xx, yy;
    int op;
    while(m--) {
        cin>>op>>xx>>yy;
        if(op == 1) {
            cin>>op;
            modify(1, xx, yy, op);
        } else {
            cout<<query(1, xx, yy); puts("");
        }
    }
}
/**
 * 建树时 向下回溯对父节点进行更新
 * modily时 向下回溯要先对左右节点进行更新，之后再对父节点进行更新
 * query时 向下回溯要先对左右节点进行更新
 *
*/

namespace mul_and_add {

```

```cpp

    // 先 * 后 +

const int N = 2e5 + 21;
int w[N];
int mod;
struct SegTree {
    int l, r;
    LL sum,mul,add;
}tr[N << 2];
inline int ls(int r) {return r << 1; }
inline int rs(int r) {return r << 1 | 1; }
void pushup(int u) {
    tr[u].sum = (tr[ls(u)].sum + tr[rs(u)].sum) % mod;
}
void pushdown(int u) {
    auto &root = tr[u], &right = tr[rs(u)], &left = tr[ls(u)];
    // 先计算子节点和 子.sum * 根.mul + 根.add * (子节点区间范围)
    right.sum = (right.sum * root.mul + root.add * (right.r - right.l + 1)) % mod;
    left.sum = (left.sum * root.mul + root.add * (left.r - left.l + 1)) % mod;
    // 之后更新子节点lazy # mul  子.mul = 子.mul * 根.mul
    right.mul = (right.mul * root.mul) % mod;
    left.mul = (left.mul * root.mul) % mod;
    // 最后更新子节点lazy # add 子.add = 子.add * 根.mul + 根.add
    left.add = (left.add * root.mul + root.add) % mod;
    right.add = (right.add * root.mul + root.add) % mod;
    // 根.add = 0 根.mul = 1
    root.add = 0;
    root.mul = 1;
}
void build(int u, int l, int r) {
    if(l == r) tr[u] = {l,r,w[r],1,0};
    else {
        int mid = l + r >> 1;
        tr[u] = {l,r,0,1,0};
        build(ls(u),l, mid); build(rs(u), mid + 1, r);
        pushup(u);
    }
}
void modify_mul(int u, int l, int r, int k) {
    if(tr[u].l >= l && tr[u].r <= r) {
        // 加 = 加 * k
        // 乘 = 乘 * k
        // 和 = 和 * k
        tr[u].add = (tr[u].add * k) % mod;
        tr[u].mul = (tr[u].mul * k) % mod;
        tr[u].sum = (tr[u].sum * k) % mod;
        return ;
    }
    pushdown(u);
    int mid = tr[u].l + tr[u].r >> 1;
    if(l <= mid) modify_mul(ls(u), l,r,k);
    if(r > mid) modify_mul(rs(u),l,r,k);
    pushup(u);
}
void modify_add(int u, int l, int r, int k) {
    if(tr[u].l >= l && tr[u].r <= r) {
        // 加 = 加 + k
        // 和 = 和 + k * (区间长度)
        tr[u].add = (tr[u].add + k) % mod;
```

```
236         tr[u].sum = (tr[u].sum + k * (tr[u].r - tr[u].l + 1)) % mod;
237         return ;
238     }
239     pushdown(u);
240     int mid = tr[u].l + tr[u].r >> 1;
241     if(l <= mid) modify_add(ls(u), l,r,k);
242     if(r > mid) modify_add(rs(u),l,r,k);
243     pushup(u);
244 }
245 LL query(int u, int l, int r) {
246     if(tr[u].l >= l && tr[u].r <= r) {
247         return tr[u].sum;
248     }
249     LL sum = 0;
250     pushdown(u);
251     int mid = tr[u].l + tr[u].r >> 1;
252     if(l <= mid) sum = (sum + query(ls(u),l,r) ) % mod;
253     if(r > mid) sum = (sum + query(rs(u),l,r)) % mod;
254     return sum;
255 }
256 void solve() {
257     int n,q; cin>>n>>q>>mod;
258     rep(i,1,n) cin>>w[i];
259     build(1,1,n);
260     while(q--) {
261         int op,l,r; cin>>op>>l>>r;
262         int k;
263         if(op == 1) {
264             cin>>k;
265             modify_mul(1,l,r,k);
266         } else if(op == 2) {
267             cin>>k;
268             modify_add(1,l,r,k);
269         } else {
270             cout<<query(1,l,r)<<endl;
271         }
272     }
273 }
274
275 }
276
277 /**
278 eat k：吃掉当前的第k个零食。右边的零食全部往左移动一位（编号减一）。
279 query i j：查询当前第i个零食到第j个零食里面美味度最高的和最低的零食的美味度。
280 */
281 namespace dynamic_maximum {
282
283 const int INF = 0x3f3f3f3f;
284     // https://www.luogu.com.cn/problem/P6011
285     // https://ac.nowcoder.com/acm/problem/208250
286     /**
287      * 动态删点求区间最值
288     */
289 const int N = 2e6 + 21;
290 int w[N];
291 struct SegTree {
292     int l,r,num,mi,ma; // num 记录这一段区间内的有效长度
293 }tr[N << 2];
294 inline int ls(int u) {return u << 1; }
295 inline int rs(int u) {return u << 1 | 1; }
```

```cpp

void pushup(int u) {
    tr[u].ma = max(tr[ls(u)].ma, tr[rs(u)].ma);
    tr[u].mi = min(tr[ls(u)].mi, tr[rs(u)].mi);
    tr[u].num = tr[ls(u)].num + tr[rs(u)].num;
}

void build(int u, int l, int r) {
    if(l == r) tr[u] = {l,r,1,w[r],w[r]};
    else {
        tr[u] = {l,r};
        int mid = l + r >> 1;
        build(ls(u), l, mid), build(rs(u), mid + 1, r);
        pushup(u);
    }
}

void del(int u, int l, int r, int x) {
    if(l == r) { // 删点，将该点 经过最大值min，最小值max，num = 0，进行忽略（删除
        tr[u].mi = INF;
        tr[u].ma = -INF;
        tr[u].num = 0;
        return ;
    }
    int mid = l + r >> 1;
    // 如果左区间实际长度大于x，表示x在左
    if(tr[ls(u)].num >= x) del(ls(u), l, mid, x);
    // 否则在右，在右时，需要将x 删除掉左区间实际长度
    else del(rs(u), mid + 1, r, x - tr[ls(u)].num);
    pushup(u);
}
int query_mi(int u, int l, int r) {
    if(l <= 1 && r >= tr[u].num) {
        return tr[u].mi;
    }
    int lnum = tr[ls(u)].num;
    int tmp = INF;
    if(l <= lnum) {
        tmp = query_mi(ls(u), l,r);
    }
    if(r > lnum) {
        tmp = min(tmp, query_mi(rs(u), l - lnum, r - lnum));
    }
    return tmp;
}
int query_ma(int u, int l, int r) { // 同理
    if(l <= 1 && r >= tr[u].num) {
        return tr[u].ma;
    }
    int lnum = tr[ls(u)].num;
    int tmp = -INF;
    if(l <= lnum) {
        tmp = query_ma(ls(u),l,r);
    }
    if(r > lnum) {
        tmp = max(tmp, query_ma(rs(u),l - lnum, r - lnum));
    }
    return tmp;
}
void inpfile();
```

```cpp
void solve() {
    int n,m; cin>>n>>m;
    rep(i,1,n) w[i] = fread();
    build(1,1,n);
    while(m--) {
        int opt; cin>>opt;
        if(opt == 1) {
            int k; cin>>k;
            del(1,1,n,k);
        } else {
            int l,r; l = fread(), r = fread();
            // cout<<query_mi(1,l,r)<<" "<<query_ma(1,l,r)<<endl;
            printf("%d %d\n", query_mi(1,l,r), query_ma(1,l,r));
        }
    }
}

}

namespace scanning_line {

const int N = 2e5 + 21;
/**
 * 操作一：将某个区间 [l,r] + k
 * 操作二：整个区间中，长度大于0的区间总长度是多少
 *
 * 线段树中的节点信息：
 *      1. cnt 当前区间整个被覆盖次数
 *      2. 不考虑祖先节点cnt的前提下， cnt > 0 的区间总长
*/
int w[N];
int n;
struct Segment { // 线段
    double x, y1, y2;
    int k;
    // 按横坐标进行排序
    bool operator<(const Segment& rhs) const {
        return x < rhs.x;
    }
}seg[N << 1];
vector<double> native;
int find(double y) {
    return lower_bound(all(native), y) - native.begin();
}
struct SegTree {
    //线段树的节点tr[u]表示的线段树Node区间[tr[u].l,tr[u].r]维护离散化后的区间 --> [y_l, y_r + 1]
    int l,r,cnt;
    double len;
}tr[N << 3];
inline int ls(int u) {return u << 1; }
inline int rs(int u) {return u << 1 | 1; }
void pushup(int u) {
    if(tr[u].cnt) tr[u].len = (native[tr[u].r + 1] - native[tr[u].l]);
    else if(tr[u].l != tr[u].r) {
        tr[u].len = tr[ls(u)].len + tr[rs(u)].len;
    } else tr[u].len = 0;
}
void build(int u, int l, int r) {
    if(l == r) tr[u] = {l,r,0,0};
```

```cpp
    else {
        tr[u] = {l,r};
        int mid = l + r >> 1;
        build(ls(u),l,mid), build(rs(u),mid+1,r);
    }
}
void modify(int u, int l, int r, int k) {
    if(tr[u].l >= l && tr[u].r <= r) {
        tr[u].cnt += k;
        pushup(u);
    } else {
        int mid = tr[u].l + tr[u].r >> 1;
        if(l <= mid) modify(ls(u), l,r,k);
        if(r > mid) modify(rs(u), l,r,k);
        pushup(u);
    }
}
int T = 1;
void inpfile();
void solve() {
    native.clear();
    int seglen = 0;
    rep(i,1,n) {
        double x1,x2,y1,y2; cin>>x1>>y1>>x2>>y2;
        seg[seglen++] = {x1,y1,y2,1};
        seg[seglen++] = {x2,y1,y2,-1};
        native.push_back(y1), native.push_back(y2);
    }
    sort(all(native));
    native.erase(unique(all(native)), native.end());
    // 离散化后纵坐标有2n个点， 2n-1个区间，构建线段树，线段树的节点维护这些区间tr[i] --> [y_i, y_i+1]，所以线段树的节点个数与区间个数相同2n-1
    //从1号点开始建线段树，对应的离散化后的坐标的取值范围是0~ys.size()-2 --> 2n-1个
    build(1,0, native.size() - 2);
    sort(seg, seg + n * 2);
    double ans = 0;
    rep(i,0,n * 2 - 1) {
        ans += tr[1].len  * (seg[i].x - seg[i-1].x);
        modify(1, find(seg[i].y1), find(seg[i].y2) - 1, seg[i].k);
    }
    printf("Test case #%d\n",T++);
    printf("Total explored area: %.2lf\n\n",ans);
}
}

}}




namespace golitter {
namespace DisjointSet {
#include <unordered_map>

const int N = 234;
int fa[N];

// 朴素并查集
int find(int x) {
```

```cpp
    return fa[x] == x ? x : fa[x] = find(fa[x]);
}
void solve() {
    int n,m; cin>>n>>m;
    for(int i = 1; i <= n; ++i) fa[i] = i;
    while(m--) {
        int a,b; char opt[2];
        scanf("%s%d%d", opt, &a, &b);
        if(opt[0] == 'M') {
            fa[find(a)] = find(b); // 合并
        } else {
            if(find(a) == find(b)) {
                puts("Yes");
            } else puts("No");
        }
    }
}
// url: https://blog.csdn.net/m0_63794226/article/details/126697871
// 维护size的并查集 | 按秩合并
int fa[N], size[N],n;
// p[]存储每个点的祖宗节点, size[]只有祖宗节点的有意义，表示祖宗节点所在集合中的点的数量

// 返回x的祖宗节点
int find(int x) {
    if(fa[x] != x) fa[x] = find(fa[x]);
    return fa[x];
}
// 初始化，假定节点编号是1~n
void init() {
    for(int i = 1; i <= n; ++i) fa[i] = i, size[i] = 1;
}
// 合并a和b所在的两个集合
// 合并时的小优化 -- 将一棵点数与深度都较小的集合树连接到一棵更大的集合树下。
// 在实际代码中，即便不使用启发式合并，代码也能够在规定时间内完成任务。
void merge(int a, int b) {
    int pa = find(a), pb = find(b);
    if(pa == pb) return ;
    if(size[pa] > size[pb]) swap(pa, pb); // 保证小的合到大的里
    fa[pa] = pb;
    size[pb] += size[pa];
}

// 维护祖先节点距离的并查集 | 带权并查集
// problem: https://codeforces.com/contest/1850/problem/H
int fa[N], d[N];
//p[]存储每个点的祖宗节点, d[x]存储x到p[x]的距离
// 返回x的祖宗节点
int find(int x) {
    if(fa[x] != x) {
        int r = find(fa[x]);
        d[x] += d[fa[x]];
        fa[x] = r;
    }
    return fa[x];
}
// 初始化，假定节点编号是1~n
void init() {
    for(int i = 1; i <= n; ++i) fa[i] = i, d[i] = 0;
}
// 合并a和b所在的两个集合
```

```cpp
int distance;
void merge(int a, int b, int t) {
    int pa = find(a), pb = find(b);
    fa[pa] = pb;
    d[pa] = d[b] - d[a] + t;
    /**
     * 解释：
     *  a .. pa         b .. pb
     * a ---> pa       b ---> pb
     * a -->pa  --> pb
     *          b --|
     * dist(pa -> pb) == d[b] + t - dist(a -> pa) = d[a];
     * d[pa] = d[b] - d[a] + t;
    */
    // fa[find(a)] = find(b);
    // d[find(a)] = distance; // 根据具体问题，初始化find(a)的偏移量
}

/**
 * 离散化处理：
 *      https://www.acwing.com/solution/content/112727/
 * int find(int x ,unordered<int,int>& umii) {
 *      return umii[x] == x ? x : umii[x] = find(umii[x], umii);
 * }
 * 合并：  umii[ find(x, umii)] = umii[ find(y, umii)];
 * 判断是否联通：  find(x, umii) == find(y, umii)
*/

}}



namespace golitter {
namespace LCA { // https://www.luogu.com.cn/problem/P8805#submit 加前缀和 求树上两点之间距离
// 板子
// https://www.acwing.com/problem/content/1173/
// https://www.luogu.com.cn/problem/P3379
namespace beizeng {

const int N = 10e5 + 21;
const int M = 2*N;
int h[M],e[M],ne[M],idx;
int dep[N],root,n,m,t;
int fa[N][20];
void add(int u,int v) { //
    e[idx] = v, ne[idx] = h[u], h[u] = idx++;
}
void bfs() { // 找深度 + 预处理
    memset(dep, 0x3f, sizeof(dep));
    dep[0] = 0, dep[root] = 1;
    queue<int> q;
    q.push(root);
    while (q.size())
    {
        int x = q.front(); q.pop();
        for(int i = h[x]; ~i; i = ne[i]) {
            int y = e[i];
            if(dep[y] > dep[x]) {
                dep[y] = dep[x] + 1;
```

```
                q.push(y);
                fa[y][0] = x;
                for(int k = 1; k <= t; k++) {
                    fa[y][k] = fa[ fa[y][k-1]][k-1];
                }
            }
        }
    }

}
int lca(int x, int y) {
    if(dep[y] > dep[x]) swap(x,y); // 让x深度最大，从x到y找
    for(int k = t; k >= 0; --k) {
        if(dep[ fa[x][k]] >= dep[y]) x = fa[x][k];
    }
    if(x == y) return x;
    for(int k = t; k >= 0; --k) {
        if(fa[x][k] != fa[y][k]) {
            x = fa[x][k], y = fa[y][k];
        }
    }
    return fa[x][0];
}
void solve() {
    t = 15;
    cin>>n>>m>>root;
    memset(h, -1, sizeof(h));
    for(int i = 1; i < n; ++i) {
        int u,v; cin>>u>>v;
        add(u,v); add(v,u);
    }
    bfs();
    for(int i  = 0; i < m; ++i) {
        int u,v; cin>>u>>v;
        cout<<lca(u,v)<<endl;
    }
}
}

namespace tarjan {
    // 离线 O(n + m)

const int N = 1e6 + 21;
int fa[N], e[N], h[N], ne[N], w[N],idx,dist[N],vis[N],ans[N];
vector<PII> ask[N];
int find(int x) {return x == fa[x] ? x : fa[x] = find(fa[x]); }
void add(int u, int v, int a) {
    e[idx] = v, w[idx] = a, ne[idx] = h[u], h[u] = idx++;
}
void dfs(int u, int fu) {
    for(int i = h[u]; ~i; i = ne[i]) {
        int y = e[i];
        if(y == fu) continue;
        dist[y] = dist[u] + w[i];
        dfs(y,u);
    }
}
/**
 * 树中节点分为三类：
 * 1. 已经访问完毕并且回溯的节点。在这些节点上标记一个整数
```

```
 * 2. 已经开始递归，但尚未回溯的节点。这些节点就是当前正在访问的节点x以及x的祖先。标记为1
 * 3. 尚未访问过的节点。没有标记。
 * 对于正在访问的节点x，它到根节点的路径已经标记为1.若y是已经访问完毕并且回溯的节点，则 LCA(x,y) 就是y向上走到根，第一个遇到的标记为1的节点。
*/
void tarjan(int u) {
    vis[u] = 1;
    for(int i = h[u]; ~i; i = ne[i]) {
        int y = e[i];
        if(vis[y]) continue;
        tarjan(y);
        fa[y] = u;
    }
    for(auto t: ask[u]) {
        int y = t.vf, id = t.vs;
        if(vis[y] == 2) {
            int lca = find(y); // 最近的公共祖先
            // ans[id] = dist[u] + dist[y] - 2 * dist[anc]; // 求距离
            ans[id] = lca;
        }
    }
    vis[u] = 2;
}
void inpfile();
void solve() {
    memset(h, -1, sizeof(h));
    int n,m; cin>>n>>m; int s; cin>>s;
    for(int i = 1; i < n; ++i) {
        int u,v,a; cin>>u>>v; a = 1;
        add(u,v,a); add(v,u,a);
    }
    for(int i = 0; i < m; ++i) {
        int u,v; cin>>u>>v;
        if(u != v) {
            ask[u].pb({v,i});
            ask[v].pb({u,i});
        } else ans[i] = u; // 如果求的是最近公共祖先，重复的话就是本身咯
    }
    rep(i,1,n) fa[i] = i;
    dfs(s,-1);
    tarjan(s);
    rep(i,0,m-1) cout<<ans[i]<<endl;
}

}

}
}

namespace TCS { // 树链剖分

/**
 * url: https://www.luogu.com.cn/problem/solution/P3384
 * 树链剖分的思想是:对于两个不在同一重链内的节点,让他们不断地跳,使得他们处于同一重链上
 *
 * 如何跳:
 *  用第二次dfs中记录的top数组，** x 到 top[x] 中的节点在线段树上是连续的。
 * 结合dep数组即可。
 *
```

```
 * 选择x 和 y点dep较大的点开始跳（假设较大点是x），让x节点直接跳到 top[x]，然后在线段树上更新。
 * 最后两个节点一定是处于同一条重链的，再直接在线段树上处理即可。
 *
 *
*/
/* ------------------------------------------------------------------------------- */
const int N = 2e6 + 21;

// - `fa(x)`：表示节点`x`在树上的父亲
// - `dep(x)`：表示节点`x`在树上的深度
// - `siz(x)`：表示节点`x`的子树的节点个数
// - `son(x)`：表示节点`x`的重儿子
// - `top(x)`：表示节点`x`所在**重链**的顶部节点（深度最小
// - `dfn(x)`：表示节点`x`的**DFS序**，也是其在线段树中的编号
// - `rnk(x)`：表示DFS序所对应的节点编号，有`rnk(dfn(x)) = x`

int fa[N], dep[N], siz[N], son[N], top[N], dfn[N], rnk[N];
int h[N], e[N], ne[N], w[N], dist[N], idx,cnt;
int p;
void inpfile();

void add(int u, int v) {
    e[idx] = v, ne[idx] = h[u], h[u] = idx++;
}
/* -------------------- 树链剖分 两次dfs ---------------------------------------------- */

// 找出 fa dep siz son
void dfs1(int u) {
    // if(dep[u])
    son[u] = -1; // 重儿子设置为-1
    siz[u] = 1; // 当前u节点大小为1（它本身
    for(int i = h[u]; ~i; i = ne[i]) {
        int y = e[i];
        if(y == fa[u]) continue; // **
        if(!dep[ y]) { // 如果深度没有，则可以接着往下遍历
            dep[ y] = dep[u] + 1; // 求出深度
            fa[ y] = u; // 为y设置父亲节点
            dfs1(y); // 递归 y
            siz[u] += siz[ y]; // 当前节点u增加子节点个数
            if(son[u] == -1 || siz[ y] > siz[ son[u]]) son[u] = y; // 更新重儿子
        }
    }
}

// 求出 top dfn rnk
void dfs2(int u, int t) {

    top[u] = t; // 设置节点u的顶部节点为t
    cnt++;
    dfn[u] = cnt; // 在线段树中的编号
    rnk[cnt] = u; // DFS序对应的节点编号
    if(son[u] == -1) return ; // 如果son[u] = -1，表示是叶子节点
    dfs2(son[u], t); // 优先对重儿子进行DFS，保证同一条重链上的点DFS序连续
    for(int i = h[u]; ~i; i = ne[i]) {
        int y = e[i];
        // 当不是u的重儿子，也不是u的父亲节点
        // 那就是新的重链
        if(y != son[u] && y != fa[u]) dfs2(y, y);
```

```cpp
    }
}

// 求lca
int lca(int u, int v) {
    // 当两个点的重链顶点不一样时，表示是两个不同的重链
    // 深度大的向上跳
    // 跳到重链顶点的父亲节点
    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] ? v : u;
}

/* -------------------- 线段树 [ 区间修改 区间求和 板子 ] ---------------------------------
-----------------*/
// ( 裸线段树：树中点映射到线段树重
struct SegTree {
    int l,r;
    LL sum, add;
}tr[N << 2];
inline int ls(int u) {return u << 1; }
inline int rs(int u) {return u << 1 | 1; }
void pushup(int u) {
    tr[u].sum = (tr[ls(u)].sum + tr[rs(u)].sum) % p;
}
void pushdown(int u) {
    auto &root = tr[u], &left = tr[ls(u)], &right = tr[rs(u)];
    if(root.add) {
        left.add += root.add; left.sum += (left.r - left.l + 1) * root.add;
        left.add %= p; left.sum %= p;
        right.add += root.add; right.sum += (right.r - right.l + 1) * root.add;
        right.add %= p; right.sum %= p;
        root.add = 0;
    }
}
void build(int u, int l, int r) {
    if(l == r) tr[u] = {l,r,w[r],0};
    else {
        tr[u] = {l,r};
        int mid = l + r >> 1;
        build(ls(u), l, mid), build(rs(u), mid + 1, r);
        pushup(u);
    }
}
void modify(int u, int l, int r, int k) {
    if(tr[u].l >= l && tr[u].r <= r) {
        tr[u].add += k;
        tr[u].add %= p;
        tr[u].sum += (tr[u].r - tr[u].l + 1) * k;
        tr[u].sum %= p;
        return ;
    }
    pushdown(u);
    int mid = tr[u].l + tr[u].r >> 1;
    if(l <= mid) modify(ls(u),l,r,k);
```

```
828     if(r > mid) modify(rs(u), l, r,k);
829     pushup(u);
830 }
831 LL query(int u, int l, int r) {
832     if(tr[u].l >= l && tr[u].r <= r) {
833         return tr[u].sum;
834     }
835     int mid = tr[u].l + tr[u].r >> 1;
836     LL sum = 0;
837     pushdown(u);
838     if(l <= mid) sum = query(ls(u), l,r);
839     if(r > mid) sum += query(rs(u),l,r);
840     return sum;
841 }
842 /* -------------------- 树链剖分 ------------------------------------------------------------
    ---------------*/
843 // 求树 从 x 到 y 结点最短路径上所有节点的值之和
844 LL treesum(int x, int y) {
845     LL ans = 0;
846     // 如果x 和 y两个点对应重链顶点不一样，就向上跳
847     while(top[x] != top[y]) {
848         // 让 x 向上跳
849         if(dep[ top[x]] < dep[ top[y]]) swap(x,y);
850         // 查询这条重链的和
851         // dfn -- 对应 树中点在线段树中的映射
852         // top -- 对应重链顶点
853         ans = (ans + query(1,dfn[ top[x]], dfn[x])) % p;
854         // 让 x等于它重链顶点的父亲节点
855         x = fa[ top[x]];
856     }
857     // 让 x 在左边
858     if(dep[x] > dep[y]) swap(x,y);
859     // 处理 x 和 y 在同一条重链的区间和
860     ans = (ans + query(1, dfn[x], dfn[y])) % p;
861     return ans;
862 }
863
864 // 将树从 x 到 y 结点 最短路径上所有节点的值都加上k
865 // 同上 treeadd
866 void treeadd(int x, int y, int k) {
867     while(top[x] != top[y]) {
868         if(dep[ top[x]] < dep[ top[y]]) swap(x,y);
869         modify(1, dfn[ top[x]], dfn[x], k);
870         x = fa[ top[x]];
871     }
872     if(dep[x] > dep[y]) swap(x,y);
873     modify(1, dfn[x], dfn[y], k);
874 }
875 int a[N];
876 void solve() {
877     memset(h, -1, sizeof(h));
878     int n,m,r; cin>>n>>m>>r>>p;
879     for(int i = 1; i <= n; ++i) cin>>a[i];
880     for(int i = 1; i < n; ++i) {
881         int u,v; cin>>u>>v;
882         add(u,v), add(v,u);
883     }
884     /* ------- dfs * 2-----------------------------------*/
885     dfs1(r);
886     dfs2(r,r);
```

```
    /* ------- 将对应的在线段树中的位置和值进行设置 ---------*/
    for(int i = 1; i <= n; ++i) w[ dfn[i]] = a[i];
    /* ------- 建树 ---------*/
    build(1,1,n);

    /* --------- 查询 ----------------------------------*/
    while(m--) {
        int opt; cin>>opt;
        int x,y,z;
        if(opt == 1) {
            cin>>x>>y>>z;
            treeadd(x,y,z);
        } else if(opt == 2) {
            cin>>x>>y;
            cout<<treesum(x,y) % p<<endl;
        } else if(opt == 3) {
            cin>>x>>z;
            // 以 x 为根节点的子树内所有节点值都加上z
            modify(1, dfn[x], dfn[x] + siz[x] - 1, z);

        } else {
            cin>>x;
            // 求以 x 为根节点的子树内所有的节点值之和
            cout<<query(1, dfn[x], dfn[x] + siz[x] - 1) % p<<endl;
        }
    }
}
}

namespace golitter {
namespace block {
/**
 *
 * https://www.bilibili.com/video/BV1ms411t7xu/?spm_id_from=333.337.search-card.all.click&vd_source=13dfbe5ed2deada83969fafa995ccff6
 * 范围比线段树广，不必考虑区间可加性~
 * belong 表示这个数在哪个块里面
 * block 块大小
 * lb 数所在的左边界
 * rb 数所在的右边界
 *
*/
const int N = 2e5 + 21;
int belong[N], block, lb[N], rb[N],n,num;
void build() {
    block = sqrt(n);
    num = n / block; if(n%block) num++;
    for(int i = 1; i <= num; ++i) { // 下标从1开始
        lb[i] = (i-1)*block + 1, rb[i] = min(i*block, n);
    }
    for(int i = 1; i <= n; ++i) {
        belong[i] = (i-1)/block + 1;
    }
}

}}

namespace golitter {
namespace st {

```

```cpp
const int N = 1e5 + 21;
int st[N][25];
int a[N],n,m;
int mn[N];

void st_init() {
    for(int i = 1; i <= n; ++i) {
        st[i][0] = a[i];
    }
    for(int j = 1; (1<<j) <= n; ++j) {
        for(int i = 1; i + (1 << j) - 1 <= n; ++i) {
            st[i][j] = max(st[i][j-1], st[i + (1 << (j -1))][j-1]);
        }
    }
    for(int len = 1; len <= n; ++len) {
        int k = 0;
        while(1 <<(k+1) <= len) {
            k++;
        }
        mn[len] = k;
    }
}
int st_query(int l, int r) {
    int k = mn[r - l + 1];
    return max(st[l][k], st[r - (1<<k) + 1][k]);
}

}}

```