数据结构

```
/**
 * 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
 * 平衡树 Splay
 *
*/
#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 { // 树状数组
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

    */
// 非封装
```

```cpp
namespace plain {
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
    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;
        }
    }
}

}
// 算竞常用封装，下标从0开始，很不熟悉，使用体验：不如自己封装的
    // update：这个板子是越来越适用了（
namespace Fenwick_class{
template <class T>
struct Fenwick {
    int n;
    vector<T> a;
    Fenwick(const int &n = 0) : n(n), a(n, T()) {}
    void modify(int i, T x) {
        for (i++; i <= n; i += i & -i) {
            a[i - 1] += x;
        }
    }
    T get(int i) {
        T res = T();
        for (; i > 0; i -= i & -i) {
            res += a[i - 1];
        }
        return res;
    }
    T sum(int l, int r) { // [l, r] *这里已经改过
        return get(r + 1) - get(l);
    }
    int kth(T k) {
```

```
        int x = 0;
        for (int i = 1 << __lg(n); i; i >>= 1) {
            if (x + i <= n && k >= a[x + i - 1]) {
                x += i;
                k -= a[x - 1];
            }
        }
        return x;
    }
};
}

// 个人封装
namespace class__ {

class BIT {
private:
    int N = 0;
    vector<int> tr;
    int lowbit(int x) {return x & -x; }
public:
    BIT() {;}
    BIT(int sz) {
        sz = sz + 121;
        N = sz;
        assign(sz);
    }
    void assign(int sz) {
        tr.assign(sz,0);
    }
    void add(int x, int c) {
        for(; x < N; x += lowbit(x)) tr[x] += c;
    }
    int sum(int x) {
        int res = 0;
        for(; x; x -= lowbit(x)) res += tr[x];
        return res;
    }
    int sum(int l, int r) {
        return sum(r) - sum(l - 1);
    }
};

class BIT2d {
private:
    vector<vector<int>> tr;
    // int N;
    int n,m;
    int lowbit(int x) {return x & -x; }
public:
    BIT2d() {; }
    BIT2d(int nn, int mm) {
        assign(nn,mm);
    }
    void assign(int nn, int mm) {
        n = nn, m = mm;
        tr.assign(n + 1, vector<int> (m + 1));
    }
```

```
    void add(int x, int y, int d) {
        for(int i = x; i <= n; i += lowbit(i)) {
            for(int j = y; j <= m; j += lowbit(j)) tr[i][j] += d;
        }
    }
    int query(int x, int y) {
        int res = 0;
        for(int i = x; i; i -= lowbit(i)) {
            for(int j = y; j; j -= lowbit(j)) {
                res += tr[i][j];
            }
        }
        return res;
    }
    int query(int x1, int y1, int x2, int y2) {
        return query(x2, y2) - query(x1 - 1, y2) - query(x2, y1 - 1) + query(x1
- 1, y1 - 1);
    }
};


}

}}

namespace golitter { // 线段树
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，
*/

// 区间查询，线段树简单封装版
namespace SG {

struct SegTree {
    static const int N = 2e5 + 21;
    struct node {
        int l, r, mi;
        LL sum,add;
    }tr[N << 2];
    int w[N];
    // 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;
        tr[u].mi = min(tr[ls(u)].mi, tr[rs(u)].mi);
    }
    // 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; right.mi -= root.add;
            left.add += root.add; left.sum += (LL)(left.r - left.l +
1)*root.add; left.mi -= root.add;
            root.add = 0;
        }

    }
    // 建树
    void build(int u, int l, int r) {
        if(l == r) tr[u] = {l, r, w[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].mi;
        }
        pushdown(u);
        int mid = tr[u].l + tr[u].r >> 1;
        LL sum = INF;
        if(l <= mid) sum = query(ls(u), l, r);
        if(r > mid ) sum = min(sum, query(rs(u), l, r));
        return sum;
    }
}tree;

}

namespace plain{

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
```

```cpp
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) {
    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 xor_template {

// 更新 2023年10月17日 21点30分

struct Adt {
    int l,r;
    int sum,lz;
};

class SegTree {
private:
    vector<int> w;
    vector<Adt> tr;
    inline int ls(int p) {return p << 1; }
    inline int rs(int p) {return p << 1 | 1; }

    void pushup(int u) {
        tr[u].sum = tr[ls(u)].sum + tr[rs(u)].sum;
    }
    void pushdown(int u) {
        auto &root = tr[u], &right = tr[rs(u)], &left = tr[ls(u)];
        if(root.lz) {
            right.lz ^= 1; right.sum = (right.r - right.l + 1 - right.sum);
            left.lz ^= 1; left.sum = (left.r - left.l + 1 - left.sum);
            root.lz = 0;
        }
    }
public:
    SegTree() {
        ;
    }
    SegTree(int N) {
        assign(N);
    }
    void assign(int N) {
        N = N + 21;
        int wn = N;
        tr.assign(N << 2, Adt{});
```

```
        w.assign(wn + 1, 0);
    }
    void atw(int idx, int val) {
        w[idx] = val;
    }
    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 d) {
        if(tr[u].l >= l && tr[u].r <= r) {
            tr[u].lz ^= 1;
            tr[u].sum = (tr[u].r - tr[u].l + 1 - tr[u].sum);
        } 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);
        }
    }
    int query(int u, int l, int r) {
        if(tr[u].l >= l && tr[u].r <= r) {
            return tr[u].sum;
        } else {
            pushdown(u);
            int mid = tr[u].l + tr[u].r >> 1;
            int ans = 0;
            if(l <= mid) ans = query(ls(u), l, r);
            if(r > mid) ans += query(rs(u), l, r);
            return ans;
        }
    }
};

}

namespace seg_merge {

const int N = 2e5 + 21;
struct SegTree {
    int l,r,cover; // cover统计该区间的覆盖次数
}tr[N << 2];
inline int ls(int u) { return u << 1; }
inline int rs(int u) { return u << 1 | 1; }
void pushup(int u) {
    int mi = min(tr[ls(u)].cover, tr[rs(u)].cover);
    tr[ls(u)].cover -= mi; tr[rs(u)].cover -= mi;
    tr[u].cover += mi;
}
void pushdown(int u) {
    auto &root = tr[u], &right = tr[rs(u)], &left = tr[ls(u)];
    if(root.cover) {
```

```
        left.cover += root.cover;
        right.cover += root.cover;
        root.cover = 0;
    }
}
void build(int u, int l, int r) {
    if(l == r) tr[u] = {l,r,0};
    else {
        int mid = l + r >> 1;
        tr[u] = {l,r};
        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].cover += k;
        return ;
    }
    pushdown(u);
    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 query(int u, int l, int r) {
    LL res = 0;
    if(tr[u].l >= l && tr[u].r <= r) {
        if(tr[u].cover > 0) {
            res += tr[u].r - tr[u].l + 1;
        } else if(l != r) {
            int mid = tr[u].l + tr[u].r >> 1;
            pushdown(u);
            if(l <= mid) res += query(ls(u),l,mid);
            if(r > mid) res += query(rs(u),mid+1,r);
        }
        return res;
    } else {
        int mid = tr[u].l + tr[u].r >> 1;
        pushdown(u);
        if(l <= mid) res += query(ls(u),l,r);
        if(r > mid) res += query(rs(u),l,r);
        return res;
    }
}
void inpfile();
void solve() {
    int n,L; cin>>n>>L;
    build(1,1,L);
    set<PII> s;
    while(n--) {
        int opt,l,r; cin>>opt>>l>>r;
        if(opt == 1) {
            if(s.find({l,r}) != s.end()) continue;
            s.insert({l,r});
            modify(1,l,r,1);
        } else if(opt == 2) {
            if(s.find({l,r}) == s.end()) continue;
```

```
            s.erase({l,r});
            modify(1,l,r,-1);
        } else if(opt == 3) {
            cout<<query(1,1,L)<<endl;
        }
    }
}
}

namespace mul_and_add {


    // 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;
        tr[u].sum = (tr[u].sum + k * (tr[u].r - tr[u].l + 1)) % mod;
        return ;
    }
    pushdown(u);
    int mid = tr[u].l + tr[u].r >> 1;
    if(l <= mid) modify_add(ls(u), l,r,k);
    if(r > mid) modify_add(rs(u),l,r,k);
    pushup(u);
}
LL query(int u, int l, int r) {
    if(tr[u].l >= l && tr[u].r <= r) {
        return tr[u].sum;
    }
    LL sum = 0;
    pushdown(u);
    int mid = tr[u].l + tr[u].r >> 1;
    if(l <= mid) sum = (sum + query(ls(u),l,r) ) % mod;
    if(r > mid) sum = (sum + query(rs(u),l,r)) % mod;
    return sum;
}
void solve() {
    int n,q; cin>>n>>q>>mod;
    rep(i,1,n) cin>>w[i];
    build(1,1,n);
    while(q--) {
        int op,l,r; cin>>op>>l>>r;
        int k;
        if(op == 1) {
            cin>>k;
            modify_mul(1,l,r,k);
        } else if(op == 2) {
            cin>>k;
            modify_add(1,l,r,k);
        } else {
            cout<<query(1,l,r)<<endl;
        }
    }
}

}

/**
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）。
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。
```

```
*/
namespace dynamic_maximum {

const int INF = 0x3f3f3f3f;
    // https://www.luogu.com.cn/problem/P6011
    // https://ac.nowcoder.com/acm/problem/208250
    /**
     * 动态删点求区间最值
    */
const int N = 2e6 + 21;
int w[N];
struct SegTree {
    int l,r,num,mi,ma; // num 记录这一段区间内的有效长度
}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].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();
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};
    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 { // 并查集
namespace DisjointSet {
#include <unordered_map>

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

// 朴素并查集
int find(int x) {
    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], sz[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, sz[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(sz[pa] > sz[pb]) swap(pa, pb); // 保证小的合到大的里
```

```
    fa[pa] = pb;
    sz[pb] += sz[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
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 { // LCA
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
```

```cpp
// 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;
}

}

}
}

// 重剖和长剖唯一不同的是：重链剖分中一个点的重儿子是子树最大（管辖节点最多）的儿子，而长链剖分选择的是 子树深度最大的那个儿子（子树深度：一个点的子树中深度最大的点的深度）。
// https://blog.csdn.net/weixin_34138521/article/details/94081891?ops_request_misc=&request_id=&biz_id=102&utm_term=%E9%87%8D%E9%93%BE%20%E9%95%BF%E9%93%BE%20%E7%AE%97%E6%B3%95&utm_medium=distribute.pc_search_result.none-task-blog-2~all~sobaiduweb~default-2-94081891.142^v94^chatsearchT3_1&spm=1018.2226.3001.4187

namespace golitter { // 链剖分
namespace dfn { // dfs序建立
// https://ac.nowcoder.com/acm/problem/204871
// https://ac.nowcoder.com/acm/problem/23051?&headNav=acm
// https://codeforces.com/contest/1891/problem/F
void build() {

    vector<vector<int>> g(n+1);
    for(int i = 1; i < n; ++i) {
        // u,v 建图
        int u,v; u = fread(); v = fread();
        g[u].push_back(v);
        g[v].push_back(u);
    }
    // dfs序的左右端点
    // 表示以x为根的子树的左右端点位置
    vector<int> l(n + 1), r(n + 1);
    int cnt = 0;
    // 一个dfs找dfs序
    auto dfs = [&](auto &&self, int u, int fa) -> void {
        l[u] = ++cnt;
        for(auto y: g[u]) {
            if(y == fa) continue;
            self(self, y,u);
        }
        r[u] = cnt;
    };
    dfs(dfs, k,-1);
}
}

namespace LCS { // 长链剖分
        // https://codeforces.com/gym/104077/problem/L
```

```cpp
// 树链剖分改
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;
void inpfile();
vector<int> lgh;
void add(int u, int v) {
    e[idx] = v, ne[idx] = h[u], h[u] = idx++;
}
void dfs1(int u) {
    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
            fa[y] = u;
            dfs1(y); // 递归 y
            siz[u] += siz[ y]; // 当前节点u增加子节点个数
            if(dep[ y] > dep[ son[u]]) son[u] = y; // 更新重儿子
        }
    }
    dep[u] = dep[son[u]] + 1;
}

void dfs2(int u, int len) {

    if(son[u] == 0) {
        lgh.push_back(len);
        return ;
    } // 如果son[u] = -1，表示是叶子节点
    dfs2(son[u], len+1); // 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, 1);
    }
}

}}
namespace golitter {
/// @brief 换根操作 https://www.luogu.com.cn/blog/Farkas/guan-yu-shu-lian-pou-fen-huan-gen-cao-zuo-bi-ji
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;
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);
    if(r > mid) modify(rs(u), l, r,k);
```

```
    pushup(u);
}
LL query(int u, int l, int r) {
    if(tr[u].l >= l && tr[u].r <= r) {
        return tr[u].sum;
    }
    int mid = tr[u].l + tr[u].r >> 1;
    LL sum = 0;
    pushdown(u);
    if(l <= mid) sum = query(ls(u), l,r);
    if(r > mid) sum += query(rs(u),l,r);
    return sum;
}
/* -------------------- 树链剖分 -----------------------------------------------
------------------------*/
// 求树 从 x 到 y 结点最短路径上所有节点的值之和
LL treesum(int x, int y) {
    LL ans = 0;
    // 如果x 和 y两个点对应重链顶点不一样，就向上跳
    while(top[x] != top[y]) {
        // 让 x 向上跳
        if(dep[ top[x]] < dep[ top[y]]) swap(x,y);
        // 查询这条重链的和
        // dfn -- 对应 树中点在线段树中的映射
        // top -- 对应重链顶点
        ans = (ans + query(1,dfn[ top[x]], dfn[x])) % p;
        // 让 x等于它重链顶点的父亲节点
        x = fa[ top[x]];
    }
    // 让 x 在左边
    if(dep[x] > dep[y]) swap(x,y);
    // 处理 x 和 y 在同一条重链的区间和
    ans = (ans + query(1, dfn[x], dfn[y])) % p;
    return ans;
}

// 将树从 x 到 y 结点 最短路径上所有节点的值都加上k
// 同上 treeadd
void treeadd(int x, int y, int k) {
    while(top[x] != top[y]) {
        if(dep[ top[x]] < dep[ top[y]]) swap(x,y);
        modify(1, dfn[ top[x]], dfn[x], k);
        x = fa[ top[x]];
    }
    if(dep[x] > dep[y]) swap(x,y);
    modify(1, dfn[x], dfn[y], k);
}
int a[N];
void solve() {
    memset(h, -1, sizeof(h));
    int n,m,r; cin>>n>>m>>r>>p;
    for(int i = 1; i <= n; ++i) cin>>a[i];
    for(int i = 1; i < n; ++i) {
        int u,v; cin>>u>>v;
        add(u,v), add(v,u);
    }
    /* ------- dfs * 2--------------------------------------*/
    dfs1(r);
```

```
    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 change_root {

/**
 * 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 = 5e5 + 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);
    }
}

// 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;
}
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;
        right.add += root.add; right.sum += (right.r - right.l + 1) * root.add;
        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].sum += (tr[u].r - tr[u].l + 1) * k;
        return ;
    }
    pushdown(u);
    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);
}
LL query(int u, int l, int r) {
    if(tr[u].l >= l && tr[u].r <= r) {
        return tr[u].sum;
    }
    int mid = tr[u].l + tr[u].r >> 1;
    LL sum = 0;
    pushdown(u);
    if(l <= mid) sum = query(ls(u), l,r);
    if(r > mid) sum += query(rs(u),l,r);
    return sum;
}
/* -------------------- 树链剖分 ---------------------------------------------
-----------------------*/
// 求树 从 x 到 y 结点最短路径上所有节点的值之和
```

```
LL treesum(int x, int y) {
    LL ans = 0;
    // 如果x 和 y两个点对应重链顶点不一样，就向上跳
    while(top[x] != top[y]) {
        // 让 x 向上跳
        if(dep[ top[x]] < dep[ top[y]]) swap(x,y);
        // 查询这条重链的和
        // dfn -- 对应 树中点在线段树中的映射
        // top -- 对应重链顶点
        ans = (ans + query(1,dfn[ top[x]], dfn[x]));
        // 让 x等于它重链顶点的父亲节点
        x = fa[ top[x]];
    }
    // 让 x 在左边
    if(dep[x] > dep[y]) swap(x,y);
    // 处理 x 和 y 在同一条重链的区间和
    ans = (ans + query(1, dfn[x], dfn[y]));
    return ans;
}

// 将树从 x 到 y 结点 最短路径上所有节点的值都加上k
// 同上 treeadd
void treeadd(int x, int y, int k) {
    while(top[x] != top[y]) {
        if(dep[ top[x]] < dep[ top[y]]) swap(x,y);
        modify(1, dfn[ top[x]], dfn[x], k);
        x = fa[ top[x]];
    }
    if(dep[x] > dep[y]) swap(x,y);
    modify(1, dfn[x], dfn[y], k);
}
/*------------------------------ 换根操作 ----------------------------------------
--------------------*/
int root, n,m;

/*---------------------------------- 在以root为根的lca ----------------------------
-------------------*/
// https://www.luogu.com.cn/blog/Farkas/guan-yu-shu-lian-pou-fen-huan-gen-cao-
zuo-bi-ji
int LCA(int x, int y) {
    if(dep[x] > dep[y]) swap(x,y);
    int xr = lca(x,root), yr = lca(y,root), xy = lca(x,y);
    if(xy == x) { // 当 lca(x,y) == x时
        if(xr == x) {
            // 情况1：root 在x的子树中，也在y的子树中，即
                // lca(x,root) == x && lca(y,root) == y 此时 LCA(x,y)是y
            if(yr == y) return y;
            // 情况2：root在x的子树中，但不在y的子树中，即lca(x,root)，此时 LCA(x,y) 是
lca(y,root)
            return yr;
        }
        // 情况3：是x
        return x;
    }
    if(xr == x) {
        return x;
    }
    if(yr == y) {
```

```
        return y;
    }
    if(xr == root && xy == yr || (yr == root && xy == xy)) {
        return root;
    }
    if(xr == yr) return xy;
    if(xy != xr) return xr;
    return yr;
}
/*--------------------------------以root为根的增添，查询 ----------------------------
--------------------*/
// u 到 root路径上 与u相挨着的节点v的子树
int find_adj(int u, int rt) {
    // 从深度大的开始跳，往上跳
    while(top[u] != top[rt]) {
        if(dep[ top[u]] < dep[ top[rt]]) swap(u,rt);
        // 如果 root是u所在重链的父亲节点，那么直接返回即可
        if(fa[ top[u]] == rt) return top[u];
        u = fa[ top[u]];
    }
    // 让root深度最浅
    if(dep[u] < dep[rt]) swap(u, rt);
    return son[rt];
}


// 设u为要查的子树的根节点。
// - 如果root = u，那么子树即为整棵树
// - 设lca为root和u的LCA。如果 `lca != u`，那么对于查询没有影响
// - 如果 `lca = u`，那么u节点的子树就是整棵树减去u - root这个路径上与u相挨着的节点v的子树
即可。
void nodeadd(int u, int k) {
    if(root == u) modify(1,1,n, k); // 子树就是整树
    else {
        int lac = lca(u, root);
        if(lac != u) modify(1, dfn[u], dfn[u] + siz[u] - 1, k); // 对查询没有影响
        else {
            // 否则就是  u节点的子树就是整棵树减去u - root这个路径上与u相挨着的节点v的子树
            int adju = find_adj(u, root);
            modify(1,1,n, k);
          modify(1, dfn[adju], dfn[adju] + siz[adju] - 1, -k);
        }
    }
}
// 同上
LL nodesum(int u) {
    if(root == u) return query(1,1,n);
    else {
        int lac = lca(u, root);
        if(lac != u) return query(1, dfn[u], dfn[u] + siz[u] - 1);
        else {
            int adju = find_adj(u, root);
            return query(1, 1,n) - query(1, dfn[adju], dfn[adju] + siz[adju] -
1);
        }
    }
}
int a[N];
```

```
void solve() {
    memset(h, -1, sizeof(h));
    cin>>n;
    root = 1;
    for(int i = 1; i <= n; ++i) cin>>a[i];
    for(int i = 1; i < n; ++i) {
        int u; cin>>u;
        add(u,i+1); add(i+1,u);
    }
    // cin>>m;
    // dbgtt
    /* ------- dfs * 2-----------------------------------*/
    dfs1(1);
    dfs2(1,1);
    /* ------- 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];
    // for(int i = 1; i <= n; ++i) cout<<dfn[i]<<endl;
    /* ------- 建树 ---------*/
    build(1,1,n);
    /* --------- 查询 ----------------------------------*/
    cin>>m;
    while(m--) {
        int opt; cin>>opt;
        int x,y,z;
        if(opt == 1) {
            cin>>x;
            root = x;
        } else if(opt == 2) {
            cin>>x>>y>>z;
            treeadd(x,y,z);
        } else if(opt == 3) {
            cin>>x>>z;
            nodeadd(x,z);
        } else if(opt == 4) {
            cin>>x>>y;
            cout<<treesum(x,y)<<endl;
        } else {
            cin>>x;
            cout<<nodesum(x)<<endl;
        }
    }
}
}

}
namespace golitter { // 分块
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 { // st表
// 算竞常用封装
namespace SparseTable {
template <class T>
struct SparseTable {
    int n;
    vector<vector<T>> a;
    function<T(T, T)> func = [](const T &a, const T &b) { // 套模板时修改此函数 代表查询的性质
        return dis[a]<dis[b]?a:b;
    };
    SparseTable(const vector<T> &init) : n(init.size()) {
        int lg = __lg(n);
        a.assign(lg + 1, vector<T>(n));
        a[0] = init;
        for (int i = 1; i <= lg; i++) {
            for (int j = 0; j <= n - (1 << i); j++) {
                a[i][j] = func(a[i - 1][j], a[i - 1][(1 << (i - 1)) + j]);
            }
        }
    }
    T get(int l, int r) { // [l, r) 下标从0开始
        if(l > r)swap(l,r);
        r++;
        int lg = __lg(r - l);
        return func(a[lg][l], a[lg][r - (1 << lg)]);
    }
};
}
namespace st {

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]);
}

}}


namespace golitter { // splay
// 普通Splay
namespace Splay {
// https://zhuanlan.zhihu.com/p/556896902
// https://www.luogu.com.cn/problem/P3369
// https://www.luogu.com.cn/problem/P5076
// https://www.luogu.com.cn/problem/P1168
// https://www.luogu.com.cn/problem/P1801
struct Splay {
    static const int N = 200005;
    int rt, tot, fa[N], ch[N][2], val[N], cnt[N], siz[N];

    void push_up(int x) { siz[x] = siz[ch[x][0]] + siz[ch[x][1]] + cnt[x]; }

    bool get(int x) { return x == ch[fa[x]][1]; }

    void clear(int x) {
        ch[x][0] = ch[x][1] = fa[x] = val[x] = siz[x] = cnt[x] = 0;
    }

    void rotate(int x) {
        int y = fa[x], z = fa[y], chk = get(x); ch[y][chk] = ch[x][chk ^ 1];
        if (ch[x][chk ^ 1]) fa[ch[x][chk ^ 1]] = y;
        ch[x][chk ^ 1] = y; fa[y] = x; fa[x] = z; if (z) ch[z][y == ch[z][1]] =
x; push_up(y);
    }

    void splay(int x,int goal) {
        for (int f = fa[x]; (f = fa[x]) != goal; rotate(x))
            if (fa[f] != goal) rotate(get(x) == get(f) ? f : x);
        if(goal == 0)rt = x;
    }

    void splay(int x) {
        splay(x, 0);
    }

    void ins(int k) {
        if (!rt) {
            val[++tot] = k; cnt[tot]++; rt = tot; push_up(rt);
```

```
            return;
        }
        int cur = rt, f = 0;
        while (1) {
            if (val[cur] == k) {
                cnt[cur]++; push_up(cur); push_up(f); splay(cur); break;
            }
            f = cur; cur = ch[cur][val[cur] < k];
            if (!cur) {
                val[++tot] = k; cnt[tot]++; fa[tot] = f; ch[f][val[f] < k] =
tot; push_up(tot); push_up(f); splay(tot);
                break;
            }
        }
    }

    int rk(int k) {
        int res = 0, cur = rt;
        while (1) {
            if (k < val[cur]) {
                cur = ch[cur][0];
            }
            else {
                res += siz[ch[cur][0]];
                if (k == val[cur]) {
                    splay(cur); return res + 1;
                }
                res += cnt[cur]; cur = ch[cur][1];
            }
        }
    }

    int kth(int k) {
        int cur = rt;
        while (1) {
            if (ch[cur][0] && k <= siz[ch[cur][0]]) {
                cur = ch[cur][0];
            }
            else {
                k -= cnt[cur] + siz[ch[cur][0]];
                if (k <= 0) {
                    splay(cur); return val[cur];
                }
                cur = ch[cur][1];
            }
        }
    }

    int pre() {
        int cur = ch[rt][0];
        if (!cur) return cur;
        while (ch[cur][1]) cur = ch[cur][1];
        splay(cur); return cur;
    }

    int nxt() {
        int cur = ch[rt][1];
        if (!cur) return cur;
```

```
        while (ch[cur][0]) cur = ch[cur][0];
        splay(cur); return cur;
    }

    void del(int k) {
        rk(k);
        if (cnt[rt] > 1) {
            cnt[rt]--; push_up(rt); return;
        }
        if (!ch[rt][0] && !ch[rt][1]) {
            clear(rt); rt = 0; return;
        }
        if (!ch[rt][0]) {
            int cur = rt; rt = ch[rt][1]; fa[rt] = 0; clear(cur); return;
        }
        if (!ch[rt][1]) {
            int cur = rt; rt = ch[rt][0]; fa[rt] = 0; clear(cur); return;
        }
        int cur = rt; int x = pre(); fa[ch[cur][1]] = x; ch[x][1] = ch[cur][1];
clear(cur); push_up(rt);
    }
} tree;

void solve() {
    int n; cin >> n;
    for (int i = 1; i <= n; i++) {
        int opt, x; cin >> opt >> x;
        if (opt == 1)
            tree.ins(x); // 插入x
        else if (opt == 2)
            tree.del(x); // 删除x
        else if (opt == 3)
            tree.ins(x), printf("%d\n", tree.rk(x)), tree.del(x); // 查询 x 数的排
名(排名定义为比当前数小的数的个数
        else if (opt == 4)
            printf("%d\n", tree.kth(x)); // 查询排名为x的数 (从小到大排)
        else if (opt == 5)
            tree.ins(x), printf("%d\n", tree.val[tree.pre()]), tree.del(x); //
求 x 的前驱（小于x的最大数
        else
            tree.ins(x), printf("%d\n", tree.val[tree.nxt()]), tree.del(x); //
求 x 的后继（大于x的最小数
    }
}

}
// 区间翻转的板子
namespace RevSplay {
// https://www.luogu.com.cn/problem/P3391
// https://codeforces.com/contest/1878/problem/D
struct Splay {
    static const int N = 2e5 + 21;
    int rt, tot, fa[N], ch[N][2], val[N], cnt[N], siz[N],n;
    int tag[N];
    void push_up(int x) {
        siz[x] = siz[ch[x][0]] + siz[ch[x][1]] + cnt[x];
    }
    void init(int len, int root) {n = len; rt = root; }
```

```
    void push_down(int x) {
        if (x && tag[x]) {
            if (ch[x][0])tag[ch[x][0]] ^= 1;
            if (ch[x][1])tag[ch[x][1]] ^= 1;
            swap(ch[x][0], ch[x][1]);
            tag[x] = 0;
        }
    }
    bool get(int x) { return x == ch[fa[x]][1]; }
    void clear(int x) {
        ch[x][0] = ch[x][1] = fa[x] = val[x] = siz[x] = cnt[x] = tag[x] = tot =
rt = 0;
    }

    void rotate(int x) {
        int y = fa[x], z = fa[y], chk = get(x);
        push_down(x);
        push_down(y);
        ch[y][chk] = ch[x][chk ^ 1];
        if (ch[x][chk ^ 1]) fa[ch[x][chk ^ 1]] = y;
        ch[x][chk ^ 1] = y;
        fa[y] = x;
        fa[x] = z;
        if (z) ch[z][y == ch[z][1]] = x;
        push_up(y);
    }
    void splay(int x, int goal) {
        for (int f = fa[x]; (f = fa[x]) != goal; rotate(x))
            if (fa[f] != goal) rotate(get(x) == get(f) ? f : x);
        if (goal == 0)rt = x;
    }
    void splay(int x) {
        splay(x, 0);
    }
    int kth(int k) {

        int cur = rt;
        while (1) {
            push_down(cur);
            if (ch[cur][0] && k <= siz[ch[cur][0]]) {
                cur = ch[cur][0];
            }
            else {
                k -= cnt[cur] + siz[ch[cur][0]];
                if (k <= 0) {
                    splay(cur);
                    return cur;
                }
                cur = ch[cur][1];
            }
        }
    }
    int find(int x) {
        return kth(x + 1);
    }
    int build(int L, int R, int father) {
        if (L > R) { return 0; }
        int x = ++tot;
```

```
        int mid = (L + R) / 2;
        fa[x] = father;
        cnt[x] = 1;
        val[x] = mid;
        ch[x][0] = build(L, mid - 1, x);
        ch[x][1] = build(mid + 1, R, x);
        push_up(x);
        return x;
    }
    void rev(int L, int R) {
        int fl = find(L - 1);
        int fr = find(R + 1);
        splay(fl, 0);
        splay(fr, fl);
        int pos = ch[rt][1]; pos = ch[pos][0];
        tag[pos] ^= 1;
    }
    void dfs(int x) {
        push_down(x);
        if (ch[x][0])dfs(ch[x][0]);
        if (val[x] != 0 && val[x] != (n + 1))cout << val[x] << " ";
        if (ch[x][1])dfs(ch[x][1]);
    }
} tree;

void solve() {
    int n,m; cin>>n>>m;
    tree.init(n,1);
    tree.build(0,n + 1, 0);
    while(m--) {
        int l, r; cin>>l>>r;
        tree.rev(l, r);
    }
    tree.dfs(tree.rt);
}

}

}
```