二分

二分中，需要看答案是否满足单调性，满足单调性即可二分，然后根据二分答案判断是否符合题目条件。

```
int bisearch(int a[],int n,int x)
{
    int left = 0,right = n-1;
    while(left<=right)
    {
        int mid = (left+right)>>1;
        if(a[mid] == x)
            return mid;
        if(x > a[mid])
            left = mid+1;
        else
            right = mid- 1;
    }
    return -1;
}

```

```
while (l < r)//往左，最大找最小
{
    int mid = l + r >> 1;
    if (check(mid))
        r = mid;
    else
        l = mid + 1;
}
while (l < r)//往右，最小找最大
{
    int mid = l + r + 1 >> 1;
    if (check(mid))
        l = mid;
    else
        r = mid - 1;
}
```

记录法

后继(最大找最小)

```
while (l<=r)
{
    int mid=(l+r)/2;
    if (check(mid))
    {
        ans=mid;
        r=mid-1;
    }
    else l=mid+1;
}printf("%d",ans);
```

前驱（最小找最大）

```
while (l<=r)
{
    int mid=(l+r)/2;
    if (check(mid))
    {
        l = mid + 1;
        ans =mid;
    }
    else
        r=  mid - 1;
}printf("%d",ans);
```

二分答案常见步骤

1、证明问题单调性。

2、确定上下界

3、设计check()函数。

4、上下界之内二分答案。

异或线性基

把对n个数的组合求异或，缩小到对m个数组合求异或。设原定数字的集合为$A=\{a_1,a_2,a_3,\ldots\ldots,a_n\}$，求得线性基结果为$P=\{p_1,p_2,p_3,\ldots\ldots，p_k\}$。在A和P上面分别对任意组合求异或结果一样。

线性基构造

1、基本原理

规则：P中的每个元素的二进制位数均不同。P中元素的最少位数为，最大位数为m，P中的元素个数不会超过m。

设$A=\{a_1,a_2\}$，且两个元素位数相同（即首位都为1）。那么他的一个线性基为$P=\{a_1,a_1\oplus a_2\}$。

证明：$a_1\oplus a_2$比a_1、a_2的长度短，因为首位异或之后为0。那么$\{a_1,a_2\}$与$\{a_1,a_1\oplus a_2\}$异或结果相同。

当A中的位数超过两个时候，连续处理即可。比如$A=\{1000,1101,1111\}$。先把1000放入P中，然后把1101和P中的1000异或，结果为101，放入P中，此时P中有1000，101，然后把1111与1000异或的结果与101异或 放入P中，结果为10。P中为1000 101 10。

2.高斯消元

把A中的每个数写成n*m的0/1矩阵，然后简化成阶梯矩阵。

$$\begin{matrix}1000 & 1000 & 1000\\ 1101 \Rightarrow & 0101 \Rightarrow & 0101\\ 1111 & 0111 & 0010\end{matrix}$$

线性基应用

最小异或和

有全0行，则最小为0，否则为P中最小的元素。

最大异或和

异或所有的即可。

```cpp
#include <bits/stdc++.h>
using namespace std;

long long p[63];
bool zero;
void Insert(long long x)
{
    for(int i = 63;i >= 0;i--)
    {
        if(x>>i == 1)
        {
            if(p[i] == 0)
            {
                p[i] = x;
                return ;
            }
            else
            {
                x ^=p[i];
            }
        }
    }
    zero = true;
}
long long qmax()
{
    long long ans = 0;
    for(int i = 63;i >= 0;i--)
    {
        ans = max(ans,ans^p[i]);
    }
    return ans;
}
int main ()
```

```
{
    long long x;
    int n;
    cin>>n;
    for(int i =1;i <= n;i++)
    {
        cin>>x;
        Insert(x);
    }
    cout<<qmax()<<endl;
    return 0;
}
```

第k大异或和/第k小异或和

k = 1，取1111，k = 2,取1110，k = 3,取1101。

所以是选$2^t - k$的二进制对应的那些行。t为P中元素个数。

```
#include <bits/stdc++.h>
using namespace std;
int n;
bool zero;
long long a[10100];
void Gauss()//高斯消元法求线性基
{
    int i,k = 1;
    long long j = 1ll<<62;
    for(;j;j >>=1)
    {
        for(i = k;i <= n;i++)
            if(a[i] & j)
                break;
        if(i > n)
            continue;
        swap(a[i],a[k]);
        for(i = 1;i <= n;i++)
            if(i != k&&a[i]&j)
                a[i]^=a[k];
        k++;
    }
    k--;
    if(k != n)
        zero = true;
    else
        zero = false;
    n = k;
}
long long Query(long long k)
{
    long long ans = 0;
    if(zero)
        k--;
    if(!k)
        return 0;
```

```
    for(int i = n;i;i--)
    {
        if(k&1)
            ans ^= a[i];
        k>>=1;
    }
    if(k)
        return -1;
    return ans;
}
int main ()
{
    int cnt = 0;
    int t;
    cin>>t;
    while(t--)
    {
        printf("Case # %d:\n",++cnt);
        cin>>n;
        for(int i = 1;i <= n;i++)
        {
            cin>>a[i];
        }
        Gauss();
        int q;
        cin>>q;
        while(q--)
        {
            long long k;
            cin>>k;
            cout<<Query(k)<<endl;
        }
    }
}
```

Data Structure 数据结构

线段树

普通线段树

```
/**
 * @see https://www.luogu.com.cn/problem/P2068
 */
#include <bits/stdc++.h>

using ll = long long;
const int N = 1e5 + 3;
ll a[N];

struct node {
    int l, r;
    ll val, sum;
} tr[N << 2];
```

```
14
15  inline int left_child(int p) {
16      return p << 1;
17  }
18
19  inline int right_child(int p) {
20      return p << 1 | 1;
21  }
22
23  inline void push_up(int p) {
24      tr[p].val = tr[left_child(p)].val + tr[right_child(p)].val;
25  }
26
27  inline void push_down(int p) {
28      if (tr[p].sum) {
29          tr[left_child(p)].val += tr[p].sum * (tr[left_child(p)].r -
    tr[left_child(p)].l + 1);
30          tr[right_child(p)].val += tr[p].sum * (tr[right_child(p)].r -
    tr[right_child(p)].l + 1);
31          tr[left_child(p)].sum += tr[p].sum;
32          tr[right_child(p)].sum += tr[p].sum;
33          tr[p].sum = 0;
34      }
35  }
36
37  inline void build(int p, int s, int t) {
38      tr[p] = {s, t, 0, 0};
39      if(s == t) {
40          tr[p].val = a[s];
41          return;
42      }
43
44      int mid = (s + t) >> 1;
45      build(left_child(p), s, mid);
46      build(right_child(p), mid + 1, t);
47      push_up(p);
48  }
49
50  inline void update(int p, int l, int r, ll k) {
51      if(l <= tr[p].l && tr[p].r <= r) {
52          tr[p].val += k * (tr[p].r - tr[p].l + 1);
53          tr[p].sum += k;
54          return;
55      }
56
57      push_down(p);
58      int mid = (tr[p].l + tr[p].r) >> 1;
59      if(l <= mid) {
60          update(left_child(p), l, r, k);
61      }
62      if(mid < r) {
63          update(right_child(p), l, r, k);
64      }
65      push_up(p);
66  }
67
```

```cpp
inline ll query(int p, int l, int r) {
    if(l <= tr[p].l && tr[p].r <= r) {
        return tr[p].val;
    }

    push_down(p);
    int mid = (tr[p].l + tr[p].r) >> 1;
    ll ans = 0;
    if(l <= mid) {
        ans += query(left_child(p), l, r);
    }
    if(mid < r) {
        ans += query(right_child(p), l, r);
    }
    return ans;
}

int main() {
    int n, w;
    std::cin >> n >> w;
    build(1, 1, N);
    while (w--) {
        char op;
        int a, b;
        std::cin >> op >> a >> b;
        if (op == 'x') {
            update(1, a, a, b);
        } else {
            std::cout << query(1, a, b) << std::endl;
        }
    }
}
```

加乘线段树

```cpp
/**
 * @see https://www.luogu.com.cn/problem/P3373
 */
#include <bits/stdc++.h>

using ll = long long;
const int N = 1e5 + 7;
ll a[N];
int n, q, m;

struct node {
    ll val, mul, add;
} t[N << 2];

inline int left_child(int p) {
    return p << 1;
}

inline int right_child(int p) {
    return p << 1 | 1;
```

```
}

void push_up(int p) {
    t[p].val = (t[left_child(p)].val + t[right_child(p)].val) % m;
}

void push_down(int p, int l, int r) {
    int mid = (l + r) >> 1;
    t[left_child(p)].val = (t[left_child(p)].val * t[p].mul + t[p].add * (mid - l + 1)) % m;
    t[right_child(p)].val = (t[right_child(p)].val * t[p].mul + t[p].add * (r - mid)) % m;
    t[left_child(p)].add = (t[left_child(p)].add * t[p].mul + t[p].add) % m;
    t[right_child(p)].add = (t[right_child(p)].add * t[p].mul + t[p].add) % m;
    t[left_child(p)].mul = (t[left_child(p)].mul * t[p].mul) % m;
    t[right_child(p)].mul = (t[right_child(p)].mul * t[p].mul) % m;
    t[p].mul = 1;
    t[p].add = 0;
}

void build(int p, int l, int r) {
    t[p] = {0, 1, 0};
    if(l == r) {
        t[p].val = a[l] % m;
        return;
    }

    int mid = (l + r) >> 1;
    build(left_child(p), l, mid);
    build(right_child(p), mid + 1, r);
    push_up(p);
}

void update1(int p, int std_l, int std_r, int l, int r, ll k) {
    if(r < std_l || std_r < l) {
        return;
    }
    if(l <= std_l && std_r <= r) {
        t[p].val = (t[p].val * k) % m;
        t[p].add = (t[p].add * k) % m;
        t[p].mul = (t[p].mul * k) % m;
        return;
    }

    push_down(p, std_l, std_r);
    int mid = (std_l + std_r) >> 1;
    update1(left_child(p), std_l, mid, l, r, k);
    update1(right_child(p), mid + 1, std_r, l, r, k);
    push_up(p);
}

/**
 * [l, r] 区间乘以 k
 */
void update1(int l, int r, ll k) {
```

```
    update1(1, 1, n, l, r, k);
}

void update2(int p, int std_l, int std_r, int l, int r, ll k) {
    if(r < std_l || std_r < l) {
        return;
    }
    if(l <= std_l && std_r <= r) {
        t[p].val = (t[p].val + k * (std_r - std_l + 1)) % m;
        t[p].add = (t[p].add + k) % m;
        return;
    }

    push_down(p, std_l, std_r);
    int mid = (std_l + std_r) >> 1;
    update2(left_child(p), std_l, mid, l, r, k);
    update2(right_child(p), mid + 1, std_r, l, r, k);
    push_up(p);
}

/**
 * [l, r] 区间加上 k
 */
void update2(int l, int r, ll k) {
    update2(1, 1, n, l, r, k);
}

ll rangeSum(int p, int std_l, int std_r, int l, int r) {
    if(r < std_l || std_r < l) {
        return 0;
    }
    if(l <= std_l && std_r <= r) {
        return t[p].val;
    }

    push_down(p, std_l, std_r);
    int mid = (std_l + std_r) >> 1;
    return (rangeSum(left_child(p), std_l, mid, l, r) +
rangeSum(right_child(p), mid + 1, std_r, l, r)) % m;
}

ll rangeSum(int l, int r) {
    return rangeSum(1, 1, n, l, r);
}

int main() {
    std::cin >> n >> q >> m;
    for (int i = 1; i <= n; ++i) {
        std::cin >> a[i];
    }
    build(1, 1, n);
    while(q--) {
        int op, x, y;
        ll k;
        std::cin >> op;
        if(op == 1) {
```

```
            std::cin >> x >> y >> k;
            update1(x, y, k);
        } else if(op == 2) {
            std::cin >> x >> y >> k;
            update2(x, y, k);
        } else {
            std::cin >> x >> y;
            std::cout << rangeSum(x, y) << std::endl;
        }
    }
}
```

DSU 并查集

普通并查集

```
const int N = 2e4 + 3;
int fa[N];

void init(int n) {
    for(int i = 1; i <= n; ++i) {
        fa[i] = i;
    }
}

int query(int x) {
    while(x != fa[x]) {
        x = fa[x] = fa[fa[x]];
    }
    return x;
}

void merge(int x, int y) {
    int r1 = query(x);
    int r2 = query(y);
    if(r1 != r2) {
        fa[r2] = r1;
    }
}

bool same(int x, int y) {
    return query(x) == query(y);
}
```

带权并查集

1. 封装版

```
/**
 * @see https://codeforces.com/edu/course/2/lesson/7/1/practice/contest/289390/problem/B
 * 封装为结构体
 */
```

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

struct Info {
    int min, max, cnt;

    Info() {}

    Info(int id)
        : min(id + 1), max(id + 1), cnt(1) {}

    void apply(const Info& fy) {
        min = std::min(min, fy.min);
        max = std::max(max, fy.max);
        cnt += fy.cnt;
    }
};

template <typename Info>
struct DSU {
    std::vector<int> fa;
    std::vector<Info> info;

    DSU() {}

    DSU(int n) {
        init(n);
    }

    void init(int n) {
        fa.resize(n);
        info.resize(n);
        for(int i = 0; i < n; ++i) {
            fa[i] = i;
            info[i] = Info{i};
        }
    }

    int query(int x) {
        while (x != fa[x]) {
            x = fa[x] = fa[fa[x]];
        }
        return x;
    }

    bool merge(int x, int y) {
        int fx = query(x);
        int fy = query(y);
        if (fx == fy) {
            return false;
        }
        fa[fy] = fx;
        info[fx].apply(info[fy]);
        return true;
    }

    bool same(int x, int y) {
```

```
        return query(x) == query(y);
    }

    Info get(int x) {
        return info[x];
    }
};

int main() {
    int n, m;
    std::cin >> n >> m;
    DSU<Info> dsu(n + 1);
    while (m--) {
        std::string op;
        int u, v;
        std::cin >> op;
        if (op == "get") {
            std::cin >> v;
            Info fv = dsu.get(dsu.query(v - 1));
            std::cout << fv.min << ' ' << fv.max << ' ' << fv.cnt <<
std::endl;
        } else {
            std::cin >> u >> v;
            dsu.merge(u - 1, v - 1);
        }
    }
}
```

2. 简化版（下标从0开始）

```
/**
 * @see
https://codeforces.com/edu/course/2/lesson/7/1/practice/contest/289390/pr
oblem/B
 * 下标从 0 开始
 */
#include <bits/stdc++.h>

const int N = 3e5 + 7;

struct Info {
    int max, min, cnt;
} info[N];

int fa[N];

void init(int n) {
    for (int i = 0; i < n; ++i) {
        fa[i] = i;
        info[i] = {i + 1, i + 1, 1};
    }
}

int query(int x) {
    while (x != fa[x]) {
        x = fa[x] = fa[fa[x]];
```

```
    }
    return x;
}

bool merge(int x, int y) {
    int fx = query(x);
    int fy = query(y);
    if (fx == fy) {
        return false;
    }
    fa[fy] = fx;
    info[fx].max = std::max(info[fx].max, info[fy].max);
    info[fx].min = std::min(info[fx].min, info[fy].min);
    info[fx].cnt += info[fy].cnt;
    return true;
}

bool same(int x, int y) {
    return query(x) == query(y);
}

int main() {
    int n, m;
    std::cin >> n >> m;
    init(n);
    while (m--) {
        std::string op;
        int u, v;
        std::cin >> op;
        if (op == "get") {
            std::cin >> v;
            int fv = query(v - 1);
            std::cout << info[fv].min << ' ' << info[fv].max << ' ' <<
info[fv].cnt << std::endl;
        } else {
            std::cin >> u >> v;
            merge(u - 1, v - 1);
        }
    }
}
```

3. 简化版（下标从1开始）

```
/**
 * @see
https://codeforces.com/edu/course/2/lesson/7/1/practice/contest/289390/pr
oblem/B
 * 下标从 1 开始
 */
#include <bits/stdc++.h>

const int N = 3e5 + 7;

struct Info {
    int max, min, cnt;
} info[N];
```

```cpp

int fa[N];

void init(int n) {
    for (int i = 1; i <= n; ++i) {
        fa[i] = i;
        info[i] = {i, i, 1};
    }
}

int query(int x) {
    while (x != fa[x]) {
        x = fa[x] = fa[fa[x]];
    }
    return x;
}

bool merge(int x, int y) {
    int fx = query(x);
    int fy = query(y);
    if (fx == fy) {
        return false;
    }
    fa[fy] = fx;
    info[fx].max = std::max(info[fx].max, info[fy].max);
    info[fx].min = std::min(info[fx].min, info[fy].min);
    info[fx].cnt += info[fy].cnt;
    return true;
}

bool same(int x, int y) {
    return query(x) == query(y);
}

int main() {
    int n, m;
    std::cin >> n >> m;
    init(n);
    while (m--) {
        std::string op;
        int u, v;
        std::cin >> op;
        if (op == "get") {
            std::cin >> v;
            int fv = query(v);
            std::cout << info[fv].min << ' ' << info[fv].max << ' ' << info[fv].cnt << std::endl;
        } else {
            std::cin >> u >> v;
            merge(u, v);
        }
    }
}
```

heap

```
std::priority_queue<int, std::vector<int>, std::less<int>> pq; // less表示按照递减(从大到小)的顺序插入元素
std::priority_queue<int, std::vector<int>, std::greater<int>> pq; // greater表示按照递增（从小到大）的顺序插入元素
```

Graph 图论

链式前向星

```
const int N = 1e5 + 3;

int head[N], tot;

struct node {
    int to, next;
} edge[N << 1];

void addEdge(int u, int v) {
    edge[++tot] = {v, head[u]};
    head[u] = tot;
}

// 遍历x的邻接节点
for(int i = head[x]; i; i = edge[i].next) {

}
```

二分图最大权匹配

```
/**
 * @see https://www.luogu.com.cn/problem/B3605
 */
#include <bits/stdc++.h>

const int N = 2.5e5 + 3;

int head[N], tot;

struct node {
    int to, next;
} edge[N << 1];

void addEdge(int u, int v) {
    edge[++tot] = {v, head[u]};
    head[u] = tot;
}
```

```cpp

bool vis[N];
int match[N];

bool dfs(int x) {
    for(int i = head[x]; i; i = edge[i].next) {
        int to = edge[i].to;
        if(!vis[to]) {
            vis[to] = true;
            if(!match[to] || dfs(match[to])) {
                match[to] = x;
                return true;
            }
        }
    }
    return false;
}

int main() {
    int nl, nr, m;
    std::cin >> nl >> nr >> m;
    while(m--) {
        int u, v;
        std::cin >> u >> v;
        addEdge(u, v);
    }

    int ans = 0;
    for(int i = 1; i <= nl; ++i) {
        ans += dfs(i);
        memset(vis, 0, sizeof(vis));
    }
    std::cout << ans << std::endl;
}
```

最近公共组先LCA

树链剖分求LCA

```cpp
/**
 * 树链剖分求LCA
 * @see https://blog.csdn.net/qq_41418281/article/details/108220247
 */
#include <bits/stdc++.h>

struct HLD {
    int n;                                  // 节点个数
    std::vector<std::vector<int>> edge;  // 邻接矩阵
    std::vector<int> siz;                // siz[u]: 存以u为根的子树的结点数
    std::vector<int> dep;                // dep[u]: 存u的深度
    std::vector<int> top;                // top[u]: 存u所在重链的顶点
    std::vector<int> son;                // son[u]: 存u的重儿子
    std::vector<int> fa;                 // fa[u]: 存u的父节点

```

```
    HLD(int n) {
        this->n = n;
        edge.resize(n + 1);
        siz.resize(n + 1);
        dep.resize(n + 1);
        top.resize(n + 1);
        son.resize(n + 1);
        fa.resize(n + 1);
    }

    /**
     * 添加边
     */
    void addEdge(int u, int v) {
        edge[u].push_back(v);
    }

    /**
     * 遍历u的邻接结点
     */
    void forEach(int u, const std::function<void(int)>& func) {
        for (auto& n : edge[u]) {
            func(n);
        }
    }

    /**
     * 初始化
     * O(n)
     */
    void init(int root = 1) {
        dfs1(root, 0);
        dfs2(root, root);
    }

    /**
     * 求解lca
     * O(log n)
     */
    int lca(int u, int v) {
        while (top[u] != top[v]) {
            if (dep[top[u]] < dep[top[v]]) {
                std::swap(u, v);
            }
            u = fa[top[u]];
        }
        return dep[u] < dep[v] ? u : v;
    }

    /**
     * 查询两点间的距离
     */
    int distance(int u, int v) {
        return dep[u] + dep[v] - (dep[lca(u, v)] << 1);
    }

```

```cpp
    /**
     * 第一次dfs
     */
    void dfs1(int u, int father) {
        fa[u] = father;
        dep[u] = dep[fa[u]] + 1;
        siz[u] = 1;
        for (const auto& neighbor : edge[u]) {
            if (neighbor == fa[u]) {
                continue;
            }

            dfs1(neighbor, u);
            // 更新当前节点的子树大小
            siz[u] += siz[neighbor];
            // 寻找重儿子
            if (siz[neighbor] > siz[son[u]]) {
                son[u] = neighbor;
            }
        }
    }

    /**
     * 第二次dfs
     */
    void dfs2(int u, int t) {
        top[u] = t;
        if (!son[u]) {
            return;
        }
        dfs2(son[u], t);
        for(const auto& neighbor : edge[u]) {
            if (neighbor == fa[u] || neighbor == son[u]) {
                continue;
            }
            dfs2(neighbor, neighbor);
        }
    }
};

int main() {
    int n, m, s;
    std::cin >> n >> m >> s;

    HLD h(n);
    for (int i = 1; i <= n - 1; ++i) {
        int u, v;
        std::cin >> u >> v;
        h.addEdge(u, v);
        h.addEdge(v, u);
    }

    h.init(s);

    while (m--) {
        int a, b;
```

```
        std::cin >> a >> b;
        std::cout << h.lca(a, b) << std::endl;
    }
}
```

树链剖分HLD + 树上修改（HLD + 线段树）

```
/**
 * @see https://www.luogu.com.cn/problem/P3384
 * 树上修改与查询
 * 闭区间，下标从1开始
 */
#include <bits/stdc++.h>

// 树链剖分
template <typename T>
struct HLD {
    // 线段树节点
    struct Node {
        int l, r;
        T val, sum;
    };

    // 树链剖分相关数据
    int n;                                  // 节点个数
    std::vector<T> w;                       // 节点权值
    std::vector<std::vector<int>> edge;     // 树边
    std::vector<int> fa;                    // fa[u]: 存u的父节点
    std::vector<int> dep;                   // dep[u]: 存u的深度
    std::vector<int> son;                   // son[u]: 存u的重儿子
    std::vector<int> siz;                   // siz[u]: 存以u为根的子树的结点数
    std::vector<int> top;                   // top[u]: 存u所在重链的顶点

    // 树上修改相关数据
    int cnt;                    // 新编号计数
    std::vector<int> id;        // id[u]: 存u剖分后的新编号
    std::vector<T> nw;          // 存新编号在树中所对应节点的权值
    std::vector<Node> tree;     // 线段树

    HLD(int n, const std::vector<T>& w) {
        this->n = n;
        this->w = w;
        edge.resize(n + 1);
        fa.resize(n + 1);
        dep.resize(n + 1);
        son.resize(n + 1);
        siz.resize(n + 1);
        top.resize(n + 1);

        this->cnt = 0;
        id.resize(n + 1);
        nw.resize(n + 1);
        tree.resize(n << 2);  // 4n
    }

```

```cpp
    /**
     * 添加边
     */
    void addEdge(int u, int v) {
        edge[u].push_back(v);
    }

    /**
     * 初始化
     * O(n)
     */
    void init(int root = 1) {
        dfs1(root, 0);
        dfs2(root, root);
        build(1, 1, n);
    }

    /**
     * 求解lca
     * O(log n)
     */
    int lca(int u, int v) {
        while (top[u] != top[v]) {
            if (dep[top[u]] < dep[top[v]]) {
                std::swap(u, v);
            }
            u = fa[top[u]];
        }
        return dep[u] < dep[v] ? u : v;
    }

    /**
     * 查询两点间的距离
     */
    int distance(int u, int v) {
        return dep[u] + dep[v] - (dep[lca(u, v)] << 1);
    }

    /**
     * 第一次dfs
     * 搞出 fa dep siz son
     */
    void dfs1(int u, int father) {
        fa[u] = father;
        dep[u] = dep[fa[u]] + 1;
        siz[u] = 1;
        for (const auto& neighbor : edge[u]) {
            if (neighbor == father) {  // 只准往下走
                continue;
            }

            dfs1(neighbor, u);
            siz[u] += siz[neighbor];            // 更新当前节点的子树大小
            if (siz[son[u]] < siz[neighbor]) {  // 寻找重儿子
                son[u] = neighbor;
            }
```

```
        }
    }

    /**
     * 第二次dfs
     * 搞出 top id nw
     */
    void dfs2(int u, int t) {
        top[u] = t;
        id[u] = ++cnt;
        nw[cnt] = w[u];
        if (!son[u]) {
            return;
        }
        dfs2(son[u], t);
        for (const auto& neighbor : edge[u]) {
            if (neighbor == fa[u] || neighbor == son[u]) {  // 只准往下走 && 不能选刚才走的重儿子
                continue;
            }
            dfs2(neighbor, neighbor);
        }
    }

    inline int lc(int p) {
        return p << 1;
    }

    inline int rc(int p) {
        return p << 1 | 1;
    }

    void pushUp(int p) {
        tree[p].sum = tree[lc(p)].sum + tree[rc(p)].sum;
    }

    void pushDown(int p) {
        if (tree[p].val) {
            tree[lc(p)].sum += tree[p].val * (tree[lc(p)].r - tree[lc(p)].l + 1);
            tree[rc(p)].sum += tree[p].val * (tree[rc(p)].r - tree[rc(p)].l + 1);
            tree[lc(p)].val += tree[p].val;
            tree[rc(p)].val += tree[p].val;
            tree[p].val = 0;
        }
    }

    void build(int p, int s, int t) {
        tree[p] = {s, t, 0, nw[t]};
        if (s == t) {
            return;
        }
        int mid = s + t >> 1;
        build(lc(p), s, mid);
        build(rc(p), mid + 1, t);
```

```
        pushUp(p);
    }

    /**
     * 区间加
     */
    void update(int p, int l, int r, const T& k) {
        if (l <= tree[p].l && tree[p].r <= r) {
            tree[p].val += k;
            tree[p].sum += k * (tree[p].r - tree[p].l + 1);
            return;
        }
        pushDown(p);
        int mid = tree[p].l + tree[p].r >> 1;
        if (l <= mid) {
            update(lc(p), l, r, k);
        }
        if (r > mid) {
            update(rc(p), l, r, k);
        }
        pushUp(p);
    }

    /**
     * 将树从u到v结点最短路径上所有节点的值都加上k
     */
    void updatePath(int u, int v, const T& k) {
        while (top[u] != top[v]) {
            if (dep[top[u]] < dep[top[v]]) {
                std::swap(u, v);
            }
            update(1, id[top[u]], id[u], k);
            u = fa[top[u]];
        }
        if (dep[u] < dep[v]) {
            std::swap(u, v);
        }
        update(1, id[v], id[u], k);  // 最后一段
    }

    /**
     * 将以u为根节点的子树内所有节点值都加上k
     */
    void updateTree(int u, const T& k) {
        update(1, id[u], id[u] + siz[u] - 1, k);
    }

    /**
     * 查询区间和
     */
    T query(int p, int l, int r) {
        if (l <= tree[p].l && tree[p].r <= r) {
            return tree[p].sum;
        }
        pushDown(p);
        int mid = tree[p].l + tree[p].r >> 1;
```

```
        T ans{};
        if (l <= mid) {
            ans += query(lc(p), l, r);
        }
        if (r > mid) {
            ans += query(rc(p), l, r);
        }
        return ans;
    }

    /**
     * 求树从u到v结点最短路径上所有节点的值之和
     */
    T queryPath(int u, int v) {
        T ans{};
        while (top[u] != top[v]) {
            if (dep[top[u]] < dep[top[v]]) {
                std::swap(u, v);
            }
            ans += query(1, id[top[u]], id[u]);
            u = fa[top[u]];
        }
        if (dep[u] < dep[v]) {
            std::swap(u, v);
        }
        ans += query(1, id[v], id[u]);  // 最后一段
        return ans;
    }

    /**
     * 以u为根节点的子树内所有节点值之和
     */
    T queryTree(int u) {
        return query(1, id[u], id[u] + siz[u] - 1);
    }
};

using ll = long long;

int main() {
    int n, m, r;
    ll p;
    std::cin >> n >> m >> r >> p;
    std::vector<ll> w(n + 1);
    for (int i = 1; i <= n; ++i) {
        std::cin >> w[i];
    }
    HLD<ll> hld(n, w);
    for (int i = 1; i < n; ++i) {
        int u, v;
        std::cin >> u >> v;
        hld.addEdge(u, v);
        hld.addEdge(v, u);
    }

    hld.init(r);
```

```
270
271     while (m--) {
272         int op, x, y;
273         ll z;
274         std::cin >> op;
275         if (op == 1) {
276             std::cin >> x >> y >> z;
277             hld.updatePath(x, y, z);
278         } else if (op == 2) {
279             std::cin >> x >> y;
280             std::cout << hld.queryPath(x, y) % p << std::endl;
281         } else if (op == 3) {
282             std::cin >> x >> z;
283             hld.updateTree(x, z);
284         } else if (op == 4) {
285             std::cin >> x;
286             std::cout << hld.queryTree(x) % p << std::endl;
287         }
288     }
289 }
```

Set

有序性，所有操作是log^n，红黑树实现。只能使用迭代器访问

作用：去重，升序排序。

```
1 set<int> s;//声明
2 s.clear();//清空
3 s.insert(x);//插入元素，如果之前没有，则插入后排序，否则不插入。
4 int hav = s.count(x);//查找是否有x，返回0或1
5 set<int>::iterator it = s.find(x);//查找x并返回迭代器
6 bool isempty = s.empty();//判断空集
7 int n = s.size();//元素个数
8 s.erase(x);//删除
```

访问

使用迭代器访问，注意set不支持 `it < st.end()` 的写法

```
1  set<int> st;
2  for(set<int>::iterator it = st.begin();it != st.end();it++)
3  {
4      cout<<*it;
5  }
6
7  // since c++ 11
8  std::set<int> st = {1, 2, 3, 4, 5};
9  for(const auto& s : st) {
10     std::cout << s << ' ';
11 }
```

降序排列

```
set<int,greater<int> > st;
set<int,greater<int> >::iterator it;
st.insert(1);
for(it = st.begin();it != st.end();it++)
{
    cout<<*it;
}
```

如果是结构体类型，需要在结构体中重载小于运算符。

- 仿函数定义

```
template<class T>
struct Less {
    bool operator()(const T& x, const T& y) const {
        return x < y;
    }
};

template<class T>
struct Greater {
    bool operator()(const T& x, const T& y) const {
        return x > y;
    }
};

int main() {
    std::set<int, Greater<int>> st = {1, 2, 3, 4, 5};

    for (const auto& s : st) {
        std::cout << s << ' ';
    }
    std::cout << std::endl;

    return 0;
}
```

unordered_set

底层实现 hash table，存储唯一对象集合

遍历

```
/**
 * 遍历 since c++ 11
 * 可以看出 unordered_set 是无序，不重复的
 */
template <class Os, class K>
Os& operator<<(Os& os, const std::unordered_set<K>& v) {
    os << '[' << v.size() << "] {";
    bool o{};
    for (const auto& e : v) {
```

```
        os << (o ? ", " : (o = 1, " ")) << e;
    }
    return os << " }";
}

std::unordered_set<int> us = {2, 7, 1, 8, 2, 8};
std::cout << us << std::endl;  // [4] { 8, 1, 7, 2 }
```

插入元素

O(1)

若重复则不做任何事

如果操作后新的元素数量大于原 max_load_factor() * bucket_count() 则会发生重散列。如果（因插入而）发生了重散列，索引迭代器均会失效。否则（未发生重散列），则迭代器不会失效。

```
us.insert(9);
std::cout << us << std::endl;  // [5] { 9, 8, 1, 7, 2 }
```

移除元素

O(1)

```
us.erase(us.begin());              // 移除开头
us.erase(std::next(us.begin()));  // 移除第二个元素
std::cout << us << std::endl;     // [3] { 8, 7, 2 }

us.erase(7);                     // 移除具体值
std::cout << us << std::endl;  // [2] { 8, 2 }

// iterator erase( const_iterator first, const_iterator last );
```

合并两个集合

O(N)

```
std::unordered_set<char>
    p{'C', 'B', 'B', 'A'},
    q{'E', 'D', 'E', 'C'};

p.merge(q);
std::cout << "p: " << p << std::endl;  // p: [5] { E, D, A, B, C }
std::cout << "q: " << q << std::endl;  // q: [1] { C }
```

返回匹配特定键的元素数量

只要set中存在这个元素，count就返回1，否则返回0

```cpp
std::unordered_set set{2, 7, 1, 8, 2, 8, 1, 8, 2, 8};
std::cout << set << std::endl;  // [4] { 8, 1, 7, 2 }

const auto [min, max] = std::ranges::minmax(set);  // since c++ 20
for (int i = min; i <= max; ++i) {
    if (set.count(i) == 1) {
        std::cout << i << ' ';
    }
}
std::cout << std::endl;
```

查找指定元素

找得到返回迭代器，找不到返回 end()

```cpp
if (auto res = set.find(2); res != set.end()) {
    std::cout << "Find: " << *res << std::endl;
} else {
    std::cout << "Not found!" << std::endl;
}
```

判断集合中是否存在该元素 *since c++ 20*

```cpp
for (int x : {2, 5}) {
    if (set.contains(x)) {
        std::cout << "Find: " << x << std::endl;  // Find: 2
    } else {
        std::cout << "Not found: " << x << std::endl;  // Not found: 5
    }
}
```

清空集合

清空集合

```cpp
set.clear();
std::cout << set << std::endl;  // [0] { }
```

multiset

底层实现：红黑树

允许元素重复

基本操作O(log n)，证明：摊还分析/势函数

```cpp
std::multiset<int> ms = {1, 2, 2, 3, 3};
assert(ms.size() == 5);
for(const auto& it : ms) {
    std::cout << it << ' ';
}
std::cout << std::endl;

```

```cpp
// insert
ms.insert(2);
assert(ms.size() == 6);
for(const auto& it : ms) {
    std::cout << it << ' ';
}
std::cout << std::endl;

// merge since c++ 17
std::multiset<int> ms2 = {2, 2, 3, 3, 4};
ms.merge(ms2);
assert(ms.size() == 11);
for(const auto& it : ms) {
    std::cout << it << ' ';
}
std::cout << std::endl;

// count
// ms: 1 2 2 2 2 2 3 3 3 3 4
assert(ms.count(2) == 5);
assert(ms.count(3) == 4);

// find
assert(ms.find(1) == ms.begin());
assert(ms.find(100) == ms.end());

// contains since c++ 20
assert(ms.contains(1));
assert(ms.contains(2));
assert(ms.contains(3));
assert(ms.contains(4));
assert(!ms.contains(100));

// ==
// ms: 1 2 2 2 2 2 3 3 3 3 4
std::multiset<int> ms3 = {1, 2, 2, 2, 2, 2, 3, 3, 3, 3, 4};
assert(ms == ms3);

// erase
ms.erase(2);
assert(ms.size() == 6);
// ms: 1 3 3 3 3 4
for(const auto& it : ms) {
    std::cout << it << ' ';
}
std::cout << std::endl;

ms.erase(std::next(ms.begin()));
assert(ms.size() == 5);
// ms: 1 3 3 3 4
for(const auto& it : ms) {
    std::cout << it << ' ';
}
std::cout << std::endl;
```

自定义结构体

自定义结构体

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

struct X {
    int a, b;
};

// 按b逆序
struct Cmp {
    bool operator()(const X& o1, const X& o2) const {
        return o1.b > o2.b;
    }
};

int main() {
    std::multiset<X, Cmp> ms = {{1, 1}, {1, 2}, {1, 3}};
    /*
    1 3
    1 2
    1 1
    */
    for(const auto& it : ms) {
        std::cout << it.a << ' ' << it.b << std::endl;
    }
}
```

Map

红黑树实现

键值对(key/value)容器，迭代器可以修改value，不能修改key。Map会根据key自动排序。key不一定是int类型，只要是重载的<操作符的类型均可

```cpp
map<int,string>m;
m.count(k);//返回m中键值等于k的元素个数
m.find(k);//存在返回指向元素的迭代器，否则返回end()
m.erase(k);//删除m中键为k的元素，返回删除元素的个数。
m.erase(p);//删除迭代器p所指向的元素
m.insert(e);//e是一个用在m上的一个pair，如果e.first不在m中，则插入一个值为e.second的新元素；如果该键在m中存在，不做任何操作。
m.clear();//清空
m,empty();//判空
```

访问

访问

```
map<int,string>::iterator iter;
for(iter = mp.begin();iter != mp.end();iter++)
{
    cout<<iter->first<<" "<<iter->second<<endl;
}
```

```
auto it = mp.begin();
for(auto &[k,v]:mp)
{

}
```

自定义结构体：只能对键自定义排序

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

struct X {
    int a, b;
};

// 按b逆序
struct Cmp {
    bool operator()(const X& o1, const X& o2) const {
        return o1.b > o2.b;
    }
};

int main() {
    std::map<X, int, Cmp> mp = {{{1, 1}, 1}, {{1, 2}, 1}, {{1, 3}, 1}};
    // 1-3:1 1-2:1 1-1:1
    for(const auto& [k, v] : mp) {
        std::cout << k.a << '-' << k.b << ':' << v << ' ';
    }
    std::cout << std::endl;
}
```

统计出现次数

统计出现次数

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

int main() {
    int n = 10;
    int a[n] = {1, 5, 5, 5, 4, 4, 4, 4, 2, 3};

    std::map<int, int> mp; // k-num v-cnt
    for(int i = 0; i < n; ++i) {
        mp[a[i]]++;
    }

    for(const auto& [num, cnt] : mp) {
        std::cout << num << ':' << cnt << std::endl;
    }
```

```
}
```

unordered_map

哈希表实现，操作时间复杂度O(1)，证明：摊还分析/势函数

```cpp
std::unordered_map<int, int> mp = {{1, 1}, {2, 1}, {3, 1}};
assert(mp.size() == 3);
for(const auto&[k, v] : mp) {
    std::cout << k << ':' << v << ' ';
}
std::cout << std::endl;

// insert
mp.insert({4, 1});
assert(mp.size() == 4);
mp.insert({4, 2}); // 键相同，不会插进去
// mp: 4:1 3:1 2:1 1:1
assert(mp.size() == 4);
assert(mp[4] == 1);

// []
mp[3] = 2; // 修改
assert(mp.size() == 4);
assert(mp[3] == 2);

// erase
mp.erase(4);
assert(mp.size() == 3);
// mp: 3:2 2:1 1:1
for(const auto&[k, v] : mp) {
    std::cout << k << ':' << v << ' ';
}
std::cout << std::endl;

// find
assert(mp.find(3)->second == 2);
assert(mp.find(4) == mp.end());

// count
assert(mp.count(3) == 1);
assert(mp.count(2) == 1);
assert(mp.count(1) == 1);
assert(mp.count(4) == 0);

// contains since c++ 20
assert(mp.contains(3));
assert(mp.contains(2));
assert(mp.contains(1));
assert(!mp.contains(4));

// iterator
// 3:2 2:1 1:1
for(auto it = mp.begin(); it != mp.end(); ++it) {
    std::cout << it->first << ':' << it->second << ' ';
```

```
}
std::cout << std::endl;
```

multimap

红黑树实现，摊还时间O(log n)

允许键重复

```
std::multimap<int, int> mp = {{1, 1}, {1, 2}, {1, 3}};
assert(mp.size() == 3);
// 1:1 1:2 1:3
for (const auto &[k, v] : mp) {
    std::cout << k << ':' << v << ' ';
}
std::cout << std::endl;

// insert
mp.insert({1, 4});
assert(mp.size() == 4);
mp.insert({2, 1});
assert(mp.size() == 5);
// 1:1 1:2 1:3 1:4 2:1
for (const auto &[k, v] : mp) {
    std::cout << k << ':' << v << ' ';
}
std::cout << std::endl;

// erase
mp.erase(1);
assert(mp.size() == 1);
// 2:1
for (const auto &[k, v] : mp) {
    std::cout << k << ':' << v << ' ';
}
std::cout << std::endl;

mp.insert({1, 1});
mp.insert({1, 2});
mp.insert({1, 3});

// count
// 1:1 1:2 1:3 2:1
assert(mp.count(1) == 3);
assert(mp.count(2) == 1);

// find
// 寻找键等于 key 的的元素。若容器中有数个拥有所请求的键的元素，则可能返回任意一个
std::cout << mp.find(1)->second << std::endl;

// contains
assert(mp.contains(1));
assert(mp.contains(2));
assert(!mp.contains(100));

```

```
// 1:1 1:2 1:3 2:1
// lower_bound 返回指向首个不小于（即大于或等于）key 的元素的迭代器。
assert(mp.lower_bound(1)->first == 1);
assert(mp.lower_bound(1)->second == 1);

// upper_bound 返回指向首个大于 key 的元素的迭代器。
assert(mp.upper_bound(1)->first == 2);
assert(mp.upper_bound(1)->second == 1);
```

自定义结构体：只能对键自定义排序

自定义结构体：只能对键自定义排序

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

struct X {
    int a, b;
};

// 按b逆序
struct Cmp {
    bool operator()(const X& o1, const X& o2) const {
        return o1.b > o2.b;
    }
};

int main() {
    std::multimap<X, int, Cmp> mp = {{{1, 1}, 1}, {{1, 2}, 1}, {{1, 3}, 1}};
    // 1-3:1 1-2:1 1-1:1
    for(const auto&[k, v] : mp) {
        std::cout << k.a << '-' << k.b << ':' << v << ' ';
    }
    std::cout << std::endl;
}
```