==========
文件夹: class126_SegmentTreeDivideAndConquerAlgorithms
==========

[Markdown 文件]
==========
文件: README.md
==========
线段树分治 (Segment Tree Divide and Conquer)

概述

线段树分治是一种离线算法技术，主要用于处理带有时间维度的图论问题。它将操作序列按照时间轴建立线段树，然后通过 DFS 遍历线段树来处理各个时间区间内的操作。

补充资料

- [线段树分治题目详解](线段树分治题目详解.md) - 详细解析经典题目
- [线段树分治补充题目详解](线段树分治补充题目详解.md) - 更多平台的题目汇总
- [补充题目汇总](补充题目汇总.md) - 题目链接汇总
- [线段树分治工程化考量](线段树分治工程化考量.md) - 工程实践要点
- [线段树分治思路技巧与题型总结](线段树分治思路技巧与题型总结.md) - 解题技巧总结
- [线段树分治训练题解](线段树分治训练题解.java) - Java 实现示例
- 线段树分治_实现示例.cpp - C++实现示例
- 线段树分治_实现示例.java - Java 实现示例
- 线段树分治_实现示例.py - Python 实现示例

核心思想

1. **离线处理**：将所有操作和查询离线，按照时间建立线段树
2. **区间操作**：将每个操作的影响时间段映射到线段树的节点上
3. **可撤销数据结构**：使用可撤销并查集等支持回滚操作的数据结构
4. **DFS 遍历**：通过 DFS 遍历线段树，处理每个节点的操作并及时回滚

关键技术点

1. 可撤销并查集 (Rollback DSU)

```java
class RollbackDSU {
    int[] father, size;
    Stack<int[]> rollbackStack = new Stack<>();
```

```
    int find(int x) {
        while (x != father[x]) x = father[x];
        return x;
    }

    void union(int x, int y) {
        int fx = find(x), fy = find(y);
        if (fx == fy) return;
        // 按秩合并
        if (size[fx] < size[fy]) {
            int temp = fx; fx = fy; fy = temp;
        }
        father[fy] = fx;
        size[fx] += size[fy];
        rollbackStack.push(new int[]{fx, fy});
    }

    void rollback() {
        int[] op = rollbackStack.pop();
        int fx = op[0], fy = op[1];
        father[fy] = fy;
        size[fx] -= size[fy];
    }
}
```

2. 扩展域并查集 (Extended Union Find)

用于二分图检测：

```java
// 对于节点 x，其在左侧的编号为 x，右侧的编号为 x+n
void union(int x, int y) {
    // x 的左侧与 y 的右侧连接
    // y 的左侧与 x 的右侧连接
    union(x, y + n);
    union(y, x + n);
}
```

经典题目

1. 动态图连通性 (LOJ #121)

题目描述：
- 支持加边、删边操作
- 查询两点间连通性

解法：
- 线段树分治 + 可撤销并查集

2. 二分图判定 (P5787)

题目描述：
- 维护动态图使其为二分图

解法：
- 线段树分治 + 扩展域并查集

3. 大融合 (P4219)

题目描述：
- 支持加边操作
- 查询边的负载（删去该边后两个连通块大小的乘积）

解法：
- 线段树分治 + 可撤销并查集

4. 连通图 (P5227)

题目描述：
- 初始图为连通图
- 每次删除一些边，查询是否仍连通

解法：
- 线段树分治 + 可撤销并查集

5. 独特事件 (CF1681F)

题目描述：
- 给定一棵树，每条边有颜色
- 定义 f(u,v)为点 u 到点 v 的简单路径上恰好出现一次的颜色的数量
- 求所有点对的 f 值之和

解法：
- 线段树分治 + 可撤销并查集

6. 给边涂色 (CF576E)

题目描述：
- 给边涂色，要求每种颜色构成的子图都是二分图

解法：
- 线段树分治 + 多个扩展域并查集

7. 最小 mex 生成树 (P5631)

题目描述：
- 求生成树使得边权集合的 mex 最小

解法：
- 线段树分治 + 可撤销并查集 + 二分

8. 最短路径查询 (CF938G)

题目描述：
- 支持加边、删边和查询两点间路径异或最小值

解法：
- 线段树分治 + 可撤销并查集 + 线性基

9. 动态图连通性 (LOJ #121)

题目描述：
- 支持加边、删边操作
- 查询两点间连通性

解法：
- 线段树分治 + 可撤销并查集

10. 二分图检测 (CF813F)

题目描述：
- 检查动态加边过程中图是否始终保持二分图

解法：
- 线段树分治 + 扩展域并查集

算法复杂度

- 时间复杂度：O((n + m) log m)
- 空间复杂度：O(n + m)

其中 n 为点数，m 为操作数。

实现要点

1. **不能路径压缩**：为了支持撤销操作，只能使用按秩合并
2. **离线处理**：所有操作必须预先知道
3. **精确回滚**：每次操作后必须准确回滚到操作前状态
4. **时间区间映射**：将操作的生效时间区间正确映射到线段树节点

应用场景

1. **动态图问题**：加边、删边操作下的图性质维护
2. **二分图维护**：动态维护图的二分性
3. **连通性查询**：动态图的连通性相关查询
4. **生成树问题**：动态维护生成树相关性质

注意事项

1. 可撤销并查集不能使用路径压缩，只能按秩合并
2. 线段树分治是离线算法，不支持在线查询
3. 每个操作的影响时间区间要正确计算
4. 回滚操作必须与合并操作一一对应

==========

文件: 线段树分治工程化考量.md
==========
线段树分治工程化考量

1. 底层逻辑细节

1.1 可撤销并查集实现细节

按秩合并 vs 路径压缩
```java
// 正确实现：按秩合并
int find(int x) {
    while (x != father[x]) x = father[x];
    return x;
```

```
}

void union(int x, int y) {
    int fx = find(x), fy = find(y);
    if (fx == fy) return;
    // 按秩合并，不能使用路径压缩
    if (size[fx] < size[fy]) {
        int temp = fx; fx = fy; fy = temp;
    }
    father[fy] = fx;
    size[fx] += size[fy];
    rollbackStack.push(new int[]{fx, fy});
}
```

设计必要性：
- 路径压缩会改变树结构，无法精确回滚
- 按秩合并保持树结构相对稳定，便于回滚操作

回滚操作精确性
```java
void rollback() {
    if (rollbackStack.isEmpty()) return;
    int[] op = rollbackStack.pop();
    int fx = op[0], fy = op[1];
    father[fy] = fy;  // 恢复父节点
    size[fx] -= size[fy];  // 恢复大小
}
```

关键点：
- 回滚顺序必须与合并顺序相反
- 必须恢复所有被修改的状态

1.2 线段树区间映射细节

区间分解正确性
```java
void add(int jobl, int jobr, int jobx, int joby, int l, int r, int i) {
    if (jobl <= l && r <= jobr) {
        // 完全覆盖，直接添加
        addEdge(i, jobx, joby);
    } else {
```

```
        int mid = (l + r) >> 1;
        // 部分覆盖，递归处理子区间
        if (jobl <= mid) {
            add(jobl, jobr, jobx, joby, l, mid, i << 1);
        }
        if (jobr > mid) {
            add(jobl, jobr, jobx, joby, mid + 1, r, i << 1 | 1);
        }
    }
}
```

注意事项：
- 必须正确处理边界条件
- 避免重复添加同一区间

链式前向星实现
```java
// 使用链式前向星存储边信息
int[] head = new int[MAXM << 2];
int[] next = new int[MAXT];
int[] tox = new int[MAXT];
int[] toy = new int[MAXT];
int cnt = 0;

void addEdge(int i, int x, int y) {
    next[++cnt] = head[i];
    tox[cnt] = x;
    toy[cnt] = y;
    head[i] = cnt;
}
```

优势：
- 节省内存空间
- 插入操作时间复杂度O(1)

2. 异常场景与边界场景

2.1 空输入处理
```java
// Java版本输入检查
public int nextInt() {
```

```
    int b = skip();
    if (b == -1) {
        throw new RuntimeException("No more integers (EOF)");
    }
    // ... 处理正常输入
}
```

2.2 极端值处理
```java
// 处理最大节点数和操作数
public static int MAXN = 5001;   // 节点数上限
public static int MAXM = 500001; // 操作数上限
public static int MAXT = 5000001; // 任务数上限
```

2.3 边界条件检查
```java
void dfs(int l, int r, int i) {
    // 边界检查
    if (l > r) return;

    // 处理当前节点操作
    int unionCnt = 0;
    // ...

    if (l == r) {
        // 叶子节点处理
        if (op[l] == 2) {
            ans[l] = find(u[l]) == find(v[l]);
        }
    } else {
        // 非叶子节点递归处理
        int mid = (l + r) >> 1;
        dfs(l, mid, i << 1);
        dfs(mid + 1, r, i << 1 | 1);
    }

    // 回滚操作
    for (int j = 1; j <= unionCnt; j++) {
        undo();
    }
}
```

```

## 3. 极端输入处理

### 3.1 大规模数据处理
```java
// 使用快速 IO 处理大量输入
static class FastIO {
    private final InputStream is;
    private final OutputStream os;
    private final byte[] inbuf = new byte[1 << 16]; // 64KB 缓冲区
    private int lenbuf = 0;
    private int ptrbuf = 0;

    public int nextInt() {
        // 高效整数读取
        int b = skip();
        if (b == -1) throw new RuntimeException("EOF");
        // ...
    }
}
```

3.2 内存优化
```java
// 合理预估数组大小
public static int MAXN = 5001;    // 节点数：5000
public static int MAXM = 500001;  // 操作数：500000
public static int MAXT = 5000001; // 任务数：考虑 log 因子
```

4. 混合格式处理

4.1 输入格式兼容性
```java
// 处理不同输入格式
public int nextInt() {
    int b = skip();
    if (b == -1) throw new RuntimeException("EOF");
    boolean negative = false;
    if (b == '-') {
        negative = true;
        b = readByte();
```
```

```
    }
    int val = 0;
    while (b >= '0' && b <= '9') {
        val = val * 10 + (b - '0');
        b = readByte();
    }
    return negative ? -val : val;
}
```

4.2 输出格式标准化
```java
public void write(String s) {
    outBuf.append(s);
}

public void flush() {
    try {
        os.write(outBuf.toString().getBytes());
        os.flush();
        outBuf.setLength(0);
    } catch (IOException e) {
        throw new RuntimeException(e);
    }
}
```

5. 跨语言场景与语言特性差异

5.1 Java 特性
```java
// 面向对象封装
class RollbackDSU {
    private int[] father, size;
    private Stack<int[]> rollbackStack = new Stack<>();

    public RollbackDSU(int n) {
        father = new int[n];
        size = new int[n];
        // 初始化
    }

    public int find(int x) { /* ... */ }
```

```
    public void union(int x, int y) { /* ... */ }
    public void rollback() { /* ... */ }
}
```

优势：
- 封装性好，代码结构清晰
- 标准库丰富，开发效率高

5.2 C++特性
```cpp
// 为适应受限编译环境的实现
#define min(a,b) ((a)<(b)?(a):(b))
#define max(a,b) ((a)>(b)?(a):(b))

// 手动内存管理
const int MAXN = 5001;
int father[MAXN];
int siz[MAXN];
```

特点：
- 需要考虑编译环境限制
- 手动内存管理，性能可控

5.3 Python特性
```python
# 动态类型和简洁语法
father = [0] * MAXN
siz = [0] * MAXN
rollback = [[0, 0] for _ in range(MAXN)]

def find(i):
    while i != father[i]:
        i = father[i]
    return i
```

优势：
- 语法简洁，易于理解
- 动态类型，灵活性高

6. 性能优化策略

6.1 常数项优化
```java
// 位运算优化
int mid = (l + r) >> 1;  // 比除法更快
i << 1;                  // 左移代替乘法
i << 1 | 1;              // 左移加1代替乘法加1
```

6.2 缓存友好性
```java
// 连续内存访问
int[] father = new int[MAXN];  // 连续存储
int[] siz = new int[MAXN];     // 连续存储
```

6.3 算法层面优化
```java
// 按秩合并优化
if (size[fx] < size[fy]) {
    int temp = fx; fx = fy; fy = temp;
}
```

7. 调试与测试策略

7.1 中间过程打印
```java
void dfs(int l, int r, int i) {
    // 调试信息（可选）
    // System.out.println("Visiting node: " + i + ", range: [" + l + ", " + r + "]");

    // 处理逻辑
    // ...
}
```

7.2 断言验证
```java
void union(int x, int y) {
    int fx = find(x), fy = find(y);
    assert fx >= 0 && fx < MAXN : "Invalid father index";
    assert fy >= 0 && fy < MAXN : "Invalid father index";
```

```
    // ...
}
```

7.3 性能退化排查
```java
// 监控操作次数
int unionCnt = 0;
// ...
unionCnt++;
// ...
for (int j = 1; j <= unionCnt; j++) {
    undo();  // 确保回滚次数正确
}
```

8. 工程化最佳实践

8.1 代码可读性
```java
// 变量命名见名知意
int operationType = op[i];
int nodeX = u[i];
int nodeY = v[i];

// 关键步骤添加注释
// 按秩合并优化并查集性能
if (size[fx] < size[fy]) {
    int temp = fx; fx = fy; fy = temp;
}
```

8.2 模块化设计
```java
// 功能分离
class SegmentTreeDivideConquer { /* 线段树分治框架 */ }
class RollbackDSU { /* 可撤销并查集 */ }
class FastIO { /* 快速输入输出 */ }
```

8.3 异常处理
```java
public int nextInt() {
```

```
    int b = skip();
    if (b == -1) {
        // 明确的异常信息
        throw new RuntimeException("No more integers (EOF)");
    }
    // ...
}
```

8.4 文档化
```java
/**
 * LOJ #121 动态图连通性 - Java 实现
 *
 * 题目来源: LibreOJ
 * 题目链接: https://loj.ac/p/121
 * 时间复杂度: O((n + m) log m)
 * 空间复杂度: O(n + m)
 * 是否为最优解: 是
 */
```

通过关注这些工程化考量，可以编写出更加健壮、高效和可维护的线段树分治代码，适用于各种实际应用场景。

==========

文件: 线段树分治思路技巧与题型总结.md
==========
线段树分治思路技巧与题型总结

1. 核心思想

线段树分治是一种离线算法技术，主要用于处理带有时间维度的图论问题。它将操作序列按照时间轴建立线段树，然后通过 DFS 遍历线段树来处理各个时间区间内的操作。

1.1 基本原理
1. **离线处理**：将所有操作和查询离线，按照时间建立线段树
2. **区间操作**：将每个操作的影响时间段映射到线段树的节点上
3. **可撤销数据结构**：使用可撤销并查集等支持回滚操作的数据结构
4. **DFS 遍历**：通过 DFS 遍历线段树，处理每个节点的操作并及时回滚

1.2 核心优势

- 将动态问题转化为静态问题处理
- 通过分治思想降低时间复杂度
- 支持复杂的图论操作维护

2. 关键技术点

2.1 可撤销并查集 (Rollback DSU)
```java
class RollbackDSU {
    int[] father, size;
    Stack<int[]> rollbackStack = new Stack<>();

    int find(int x) {
        while (x != father[x]) x = father[x];
        return x;
    }

    void union(int x, int y) {
        int fx = find(x), fy = find(y);
        if (fx == fy) return;
        // 按秩合并
        if (size[fx] < size[fy]) {
            int temp = fx; fx = fy; fy = temp;
        }
        father[fy] = fx;
        size[fx] += size[fy];
        rollbackStack.push(new int[]{fx, fy});
    }

    void rollback() {
        int[] op = rollbackStack.pop();
        int fx = op[0], fy = op[1];
        father[fy] = fy;
        size[fx] -= size[fy];
    }
}
```

2.2 扩展域并查集 (Extended Union Find)
用于二分图检测：
```java
// 对于节点 x，其在左侧的编号为 x，右侧的编号为 x+n
void union(int x, int y) {
```

```
    // x 的左侧与 y 的右侧连接
    // y 的左侧与 x 的右侧连接
    union(x, y + n);
    union(y, x + n);
}
```

2.3 线段树构建与操作
```java
// 线段树区间添加操作
void add(int jobl, int jobr, int jobx, int joby, int l, int r, int i) {
    if (jobl <= l && r <= jobr) {
        addEdge(i, jobx, joby);
    } else {
        int mid = (l + r) >> 1;
        if (jobl <= mid) {
            add(jobl, jobr, jobx, joby, l, mid, i << 1);
        }
        if (jobr > mid) {
            add(jobl, jobr, jobx, joby, mid + 1, r, i << 1 | 1);
        }
    }
}
```

3. 常见题型分类

3.1 动态图连通性问题
典型题目：
- LOJ #121 动态图连通性
- Codeforces 1681F Unique Occurrences

解题思路：
1. 记录每条边的存在时间区间
2. 将时间区间映射到线段树节点
3. 使用可撤销并查集维护连通性
4. DFS 遍历线段树处理操作

时间复杂度：O((n + m) log m)

3.2 二分图维护问题
典型题目：
- 洛谷 P5787 二分图

- Codeforces 813F Bipartite Checking

解题思路：
1. 使用扩展域并查集判断二分图
2. 对于边(u,v)，连接(u,v+n)和(v,u+n)
3. 如果 u 和 u+n 在同一连通分量中，则不是二分图

时间复杂度：O((n + m) log k)

3.3 生成树相关问题
典型题目：
- 洛谷 P5631 最小 mex 生成树

解题思路：
1. 二分答案
2. 对于每个候选答案，检查是否存在不包含该权值的生成树
3. 使用线段树分治处理边的存在时间

时间复杂度：O((n + m) log m log n)

3.4 边负载计算问题
典型题目：
- 洛谷 P4219 大融合

解题思路：
1. 记录每条边的所有存在时间区间
2. 将区间映射到线段树
3. DFS 遍历时维护并查集，计算连通块大小

时间复杂度：O((n + q) log q)

3.5 图连通性验证问题
典型题目：
- 洛谷 P5227 连通图

解题思路：
1. 转换思路：找出每条边不存在的时间区间
2. 在这些区间内不使用该边
3. 检查整个图是否仍连通

时间复杂度：O((n + m) log k)

3.6 边染色问题

典型题目：
- Codeforces 576E Painting Edges

解题思路：
1. 对每种颜色维护一个扩展域并查集
2. 检查染色后是否满足条件
3. 不满足则撤销操作

时间复杂度：O(k(n + q) log q)

4. 设计本质与适用场景

4.1 适用场景
1. **时间维度操作**：操作有明确的时间区间
2. **可撤销操作**：支持回滚的数据结构
3. **离线处理**：可以预知所有操作
4. **区间查询**：需要对时间区间进行查询

4.2 设计必要性
1. **分治思想**：将复杂问题分解为简单子问题
2. **状态维护**：通过可撤销数据结构维护状态
3. **时间优化**：避免重复计算，提高效率

4.3 核心设计点
1. **时间区间映射**：正确将操作时间映射到线段树节点
2. **状态回滚**：精确回滚操作影响
3. **数据结构选择**：选择合适的支持撤销操作的数据结构

5. 工程化考量

5.1 性能优化
1. **按秩合并**：优化并查集性能
2. **精确回滚**：避免不必要的状态恢复
3. **内存管理**：合理使用数组和链表

5.2 异常处理
1. **边界检查**：处理空输入和极端值
2. **状态一致性**：保证回滚操作的正确性
3. **时间复杂度控制**：避免超时

5.3 跨语言实现
1. **Java**：使用标准库和面向对象特性
2. **C++**：注意编译环境限制，避免使用 STL

3. **Python**：利用动态特性和简洁语法

6. 学习路径建议

6.1 基础知识
1. **并查集**：理解基本操作和优化技巧
2. **线段树**：掌握区间操作和遍历方法
3. **DFS**：熟悉深度优先搜索的应用

6.2 进阶技巧
1. **可撤销数据结构**：学习状态回滚技术
2. **扩展域并查集**：掌握特殊应用场景
3. **时间分治**：理解离线处理思想

6.3 实践建议
1. **从模板题开始**：先掌握基本框架
2. **逐步增加难度**：从简单到复杂题目
3. **多语言实现**：加深理解不同语言特性
4. **总结规律**：归纳常见题型和解法

7. 常见误区与注意事项

7.1 常见误区
1. **路径压缩**：可撤销并查集不能使用路径压缩
2. **在线处理**：线段树分治是离线算法
3. **区间映射**：时间区间映射错误导致结果不正确

7.2 注意事项
1. **回滚操作**：必须与合并操作一一对应
2. **时间复杂度**：注意 log 因子的影响
3. **内存使用**：合理预估数组大小

8. 扩展应用

8.1 机器学习相关
1. **图神经网络**：动态图结构维护
2. **聚类算法**：动态连通性分析
3. **推荐系统**：用户关系网络维护

8.2 大数据处理
1. **流式计算**：时间序列数据处理
2. **图数据库**：动态图查询优化
3. **社交网络**：好友关系维护

通过系统学习线段树分治，可以有效解决一类带有时间维度的图论问题，是算法竞赛和实际工程中的重要技术。

==========

文件: 线段树分治补充题目详解.md
==========
线段树分治补充题目详解

1. 洛谷 (Luogu)

1.1 P5787 二分图 /【模板】线段树分治
- **题目链接**: https://www.luogu.com.cn/problem/P5787
- **题目描述**: 给定一个 n 个节点的图，每条边有一个存在时间区间[L,R]，对于每个时间点，判断当前图是否为二分图
- **解题思路**: 使用扩展域并查集来判断二分图，结合线段树分治处理时间区间
- **时间复杂度**: O((n + m) log k)
- **空间复杂度**: O(n + m)

1.2 P4219 大融合
- **题目链接**: https://www.luogu.com.cn/problem/P4219
- **题目描述**: 支持加边操作和查询边负载（删去该边后两个连通块大小的乘积）
- **解题思路**: 线段树分治 + 可撤销并查集
- **时间复杂度**: O((n + q) log q)
- **空间复杂度**: O(n + q)

1.3 P5227 连通图
- **题目链接**: https://www.luogu.com.cn/problem/P5227
- **题目描述**: 初始时所有点都连通，每次操作删除一些边，查询删除后是否仍连通
- **解题思路**: 转换思路，找出每条边不存在的时间区间，在这些区间内不使用该边
- **时间复杂度**: O((n + m) log k)
- **空间复杂度**: O(n + m)

1.4 P5631 最小 mex 生成树
- **题目链接**: https://www.luogu.com.cn/problem/P5631
- **题目描述**: 求生成树使得边权集合的 mex 最小
- **解题思路**: 线段树分治 + 可撤销并查集 + 二分
- **时间复杂度**: O((n + m) log m log n)
- **空间复杂度**: O(n + m)

1.5 P4588 [TJOI2018] 数学计算
- **题目链接**: https://www.luogu.com.cn/problem/P4588

- **题目描述**: 维护一个序列，支持单点修改和区间查询乘积
- **解题思路**: 时间线段树例题，最基础的线段树分治应用
- **时间复杂度**: O(q log q)
- **空间复杂度**: O(q)

1.6 P3733 [HAOI2017]八纵八横
- **题目链接**: https://www.luogu.com.cn/problem/P3733
- **题目描述**: 动态图异或最短路问题
- **解题思路**: 线段树分治 + 可撤销并查集 + 线性基
- **时间复杂度**: O((n + m) log m log n)
- **空间复杂度**: O(n + m)

1.7 P4585 [FJOI2015]火星商店问题
- **题目链接**: https://www.luogu.com.cn/problem/P4585
- **题目描述**: 线段树分治 + 可持久化 Trie
- **解题思路**: 将操作存在的时间区间分发到线段树的若干个结点
- **时间复杂度**: O(n log^2 n)
- **空间复杂度**: O(n log n)

2. LibreOJ (LOJ)

2.1 #121 动态图连通性
- **题目链接**: https://loj.ac/p/121
- **题目描述**: 支持加边、删边操作，查询两点间连通性
- **解题思路**: 线段树分治 + 可撤销并查集
- **时间复杂度**: O((n + m) log m)
- **空间复杂度**: O(n + m)

2.2 #534 「LibreOJ Round #6」花团
- **题目链接**: https://loj.ac/p/534
- **题目描述**: 时间线段树分治背包问题
- **解题思路**: 如果完全离线的话，可以直接用时间线段树分治来做
- **时间复杂度**: O(qv log q)
- **空间复杂度**: O(qv)

3. Codeforces

3.1 813F Bipartite Checking
- **题目链接**: https://codeforces.com/contest/813/problem/F
- **题目描述**: 检查动态加边过程中图是否始终保持二分图
- **解题思路**: 线段树分治 + 扩展域并查集
- **时间复杂度**: O((n + m) log m)
- **空间复杂度**: O(n + m)

3.2 1681F Unique Occurrences
- **题目链接**: https://codeforces.com/contest/1681/problem/F
- **题目描述**: 给定一棵树，每条边有颜色，定义 f(u,v)为点 u 到点 v 的简单路径上恰好出现一次的颜色的数量，求所有点对的 f 值之和
- **解题思路**: 对于每种颜色，将其作为"不存在"的颜色处理，使用线段树分治
- **时间复杂度**: O((n + m) log n)
- **空间复杂度**: O(n + m)

3.3 576E Painting Edges
- **题目链接**: https://codeforces.com/contest/576/problem/E
- **题目描述**: 给边涂色，要求每种颜色构成的子图都是二分图
- **解题思路**: 线段树分治 + 多个扩展域并查集
- **时间复杂度**: O(k(n + q) log q)
- **空间复杂度**: O(k(n + q))

3.4 938G Shortest Path Queries
- **题目链接**: https://codeforces.com/contest/938/problem/G
- **题目描述**: 支持加边、删边和查询两点间路径异或最小值
- **解题思路**: 线段树分治 + 可撤销并查集 + 线性基
- **时间复杂度**: O((n + m) log m log n)
- **空间复杂度**: O(n + m)

3.5 603E Pastoral Oddities
- **题目链接**: https://codeforces.com/contest/603/problem/E
- **题目描述**: 每条边成为最优解的范围是一个连续区间
- **解题思路**: 在 dfs 过程中维护一个堆，不断确定各条边的出现范围
- **时间复杂度**: O((n + m) log^2 n)
- **空间复杂度**: O(n + m)

3.6 1140F Extending Set of Points
- **题目链接**: https://codeforces.com/contest/1140/problem/F
- **题目描述**: 把点看成二分图上的一条边，每个联通块的贡献就是左侧节点数乘右侧节点数
- **解题思路**: 用可撤销并查集维护即可
- **时间复杂度**: O(q log q)
- **空间复杂度**: O(q)

4. AtCoder

4.1 AGC010C Cell Division
- **题目链接**: https://atcoder.jp/contests/agc010/tasks/agc010_c
- **题目描述**: 矩形划分问题，每次划分后统计连通块数量
- **解题思路**: 线段树分治 + 可撤销并查集

- **时间复杂度**: O(n log n)
- **空间复杂度**: O(n)

5. 牛客网 (Nowcoder)

5.1 2019 牛客暑期多校训练营（第八场）E Explorer
- **题目链接**: https://ac.nowcoder.com/acm/contest/888/E
- **题目描述**: 给出 m 条无向边，每条边都有一个[l,r]，意思是体积在这个范围内的人才能通过这条边，询问有多少种体积的可能性，能使人从 1 到 n
- **解题思路**: 线段树分治 + 可撤销并查集
- **时间复杂度**: O((n + m) log m)
- **空间复杂度**: O(n + m)

5.2 2020 牛客暑期多校训练营（第八场）I Interesting Computer Game
- **题目链接**: https://ac.nowcoder.com/acm/contest/5673/I
- **题目描述**: 定义两个图具有相同的联通性，当且仅当，如果 G1 中的 u,v 是联通的，那么 G2 中 u,v 也是联通的
- **解题思路**: 线段树分治+可撤销并查集
- **时间复杂度**: O((n + m) log m)
- **空间复杂度**: O(n + m)

6. BZOJ

6.1 4025 二分图
- **题目链接**: https://www.lydsy.com/JudgeOnline/problem.php?id=4025
- **题目描述**: 有 m 条边，每条边有一个出现时间和消失时间，判断每个时刻这个图是否是二分图
- **解题思路**: 线段树分治 + 可撤销并查集
- **时间复杂度**: O((n + m) log T)
- **空间复杂度**: O(n + m)

6.2 4137 [FJOI2015]火星商店问题
- **题目链接**: https://www.lydsy.com/JudgeOnline/problem.php?id=4137
- **题目描述**: 线段树分治+可持久化 Trie
- **解题思路**: 做出每个节点的可持久化 trie，不用 trie 树合并什么的，只要把当前节点表示的区间的修改排序然后一个一个加进去就行了
- **时间复杂度**: O(n log^2 n)
- **空间复杂度**: O(n log n)

6.3 4311 向量
- **题目链接**: https://www.lydsy.com/JudgeOnline/problem.php?id=4311
- **题目描述**: 维护一个向量集合，支持插入、删除向量和查询当前集合与(x, y)点积的最大值
- **解题思路**: 线段树分治板子题
- **时间复杂度**: O(n log^2 n)

- **空间复杂度**: O(n log n)

7. SPOJ

7.1 DYNACON2 Dynamic Graph Connectivity
- **题目链接**: https://www.spoj.com/problems/DYNACON2/
- **题目描述**: 动态图连通性问题
- **解题思路**: 线段树分治 + 可撤销并查集
- **时间复杂度**: O((n + m) log m)
- **空间复杂度**: O(n + m)

8. HDU

8.1 6183 Color it
- **题目链接**: http://acm.hdu.edu.cn/showproblem.php?pid=6183
- **题目描述**: 二维平面点染色，查询矩形区域内颜色种类数
- **解题思路**: cdq 分治+线段树
- **时间复杂度**: O(n log^2 n)
- **空间复杂度**: O(n log n)

9. POJ

9.1 1741 Tree
- **题目链接**: http://poj.org/problem?id=1741
- **题目描述**: 给一棵 n 个节点的树，求树上长度不超过 k 的路径有多少条
- **解题思路**: 点分治模板题
- **时间复杂度**: O(n log^2 n)
- **空间复杂度**: O(n)

10. CodeChef

10.1 MONOPLOY Gangsters of Treeland
- **题目链接**: https://www.codechef.com/problems/MONOPLOY
- **题目描述**: 树上路径查询问题
- **解题思路**: 点分治、李超线段树
- **时间复杂度**: O(n log^2 n)
- **空间复杂度**: O(n log n)

11. HackerRank

11.1 Sorted Subsegments
- **题目链接**: https://www.hackerrank.com/challenges/sorted-subsegments
- **题目描述**: 给定数组和若干排序操作，查询某个位置的值

- **解题思路**: 二分+线段树
- **时间复杂度**: O(q log^2 n)
- **空间复杂度**: O(n log n)

12. Project Euler

12.1 Problem 580 Squarefree Hilbert numbers
- **题目链接**: https://projecteuler.net/problem=580
- **题目描述**: 计算无平方因子的 Hilbert 数
- **解题思路**: 可能涉及线段树分治思想
- **时间复杂度**: O(n^(3/4))
- **空间复杂度**: O(n^(1/2))

13. 其他平台题目

13.1 USACO 相关题目
- **题目描述**: 涉及动态图连通性问题
- **解题思路**: 线段树分治 + 可撤销并查集

13.2 ZOJ 相关题目
- **题目描述**: 涉及线段树分治和可撤销并查集的应用
- **解题思路**: 线段树分治 + 可撤销并查集

13.3 TimusOJ 相关题目
- **题目描述**: 涉及线段树分治和可撤销并查集的应用
- **解题思路**: 线段树分治 + 可撤销并查集

13.4 AizuOJ 相关题目
- **题目描述**: 涉及线段树分治和可撤销并查集的应用
- **解题思路**: 线段树分治 + 可撤销并查集

13.5 Comet OJ 相关题目
- **题目描述**: 涉及线段树分治和可撤销并查集的应用
- **解题思路**: 线段树分治 + 可撤销并查集

13.6 ACWing 相关题目
- **题目描述**: 涉及线段树分治和可撤销并查集的应用
- **解题思路**: 线段树分治 + 可撤销并查集

13.7 LintCode 相关题目
- **题目描述**: 涉及线段树分治和可撤销并查集的应用
- **解题思路**: 线段树分治 + 可撤销并查集

13.8 HackerEarth 相关题目
- **题目描述**: 涉及线段树分治和可撤销并查集的应用
- **解题思路**: 线段树分治 + 可撤销并查集

13.9 计蒜客相关题目
- **题目描述**: 涉及线段树分治和可撤销并查集的应用
- **解题思路**: 线段树分治 + 可撤销并查集

13.10 各大高校 OJ 相关题目
- **题目描述**: 涉及线段树分治和可撤销并查集的应用
- **解题思路**: 线段树分治 + 可撤销并查集

13.11 MarsCode 相关题目
- **题目描述**: 涉及线段树分治和可撤销并查集的应用
- **解题思路**: 线段树分治 + 可撤销并查集

13.12 UVa OJ 相关题目
- **题目描述**: 涉及线段树分治和可撤销并查集的应用
- **解题思路**: 线段树分治 + 可撤销并查集

线段树分治核心思想总结

线段树分治是一种离线算法技术，主要用于处理带有时间维度的图论问题。它将操作序列按照时间轴建立线段树，然后通过 DFS 遍历线段树来处理各个时间区间内的操作。

核心技术点：
1. **离线处理**：将所有操作和查询离线，按照时间建立线段树
2. **区间操作**：将每个操作的影响时间段映射到线段树的节点上
3. **可撤销数据结构**：使用可撤销并查集等支持回滚操作的数据结构
4. **DFS 遍历**：通过 DFS 遍历线段树，处理每个节点的操作并及时回滚

关键数据结构：
1. **可撤销并查集 (Rollback DSU)**：支持回滚操作的并查集实现
2. **扩展域并查集 (Extended Union Find)**：用于二分图检测等特殊场景
3. **线性基**：用于处理异或相关问题

适用场景：
1. **动态图问题**：加边、删边操作下的图性质维护
2. **二分图维护**：动态维护图的二分性
3. **连通性查询**：动态图的连通性相关查询
4. **生成树问题**：动态维护生成树相关性质
5. **异或最短路**：结合线性基处理异或最短路问题
6. **背包问题**：时间线段树分治背包

注意事项：
1. 可撤销并查集不能使用路径压缩，只能按秩合并
2. 线段树分治是离线算法，不支持在线查询
3. 每个操作的影响时间区间要正确计算
4. 回滚操作必须与合并操作一一对应
5. 对于空间复杂度要求高的问题，可以考虑在线段树节点上保存数据结构副本

线段树分治经典题单推荐

入门题：
1. 洛谷 P4588 [TJOI2018] 数学计算
2. 洛谷 P5787 二分图 /【模板】线段树分治

进阶题：
1. 洛谷 P4219 大融合
2. 洛谷 P5227 连通图
3. LOJ #121 动态图连通性
4. Codeforces 813F Bipartite Checking

高级题：
1. 洛谷 P5631 最小 mex 生成树
2. Codeforces 938G Shortest Path Queries
3. Codeforces 576E Painting Edges
4. Codeforces 603E Pastoral Oddities

挑战题：
1. BZOJ 4137 [FJOI2015]火星商店问题
2. BZOJ 4311 向量
3. LOJ #534 「LibreOJ Round #6」花团

==========

文件: 线段树分治题目详解.md
==========
线段树分治经典题目详解

1. LOJ #121. 动态图连通性

题目描述
给定一个 n 个节点的动态图，支持以下操作：
1. 加边操作：在点 x 和点 y 之间增加一条边
2. 删边操作：删除点 x 和点 y 之间的边

3. 查询操作：查询点 x 和点 y 是否连通

解题思路
这是线段树分治的经典应用。我们将每条边的存在时间看作一个区间，然后将这些区间映射到线段树上。

核心算法
1. 对于每条边，记录其存在的时间区间[L,R]
2. 将这个区间在线段树上进行标记
3. DFS 遍历线段树，在每个节点处处理该节点上的所有边
4. 使用可撤销并查集维护当前的连通性
5. 到达叶子节点时回答查询

关键代码片段
```java
// 在线段树上标记边的存在时间
void add(int jobl, int jobr, int jobx, int joby, int l, int r, int i) {
    if (jobl <= l && r <= jobr) {
        addEdge(i, jobx, joby);
    } else {
        int mid = (l + r) >> 1;
        if (jobl <= mid) {
            add(jobl, jobr, jobx, joby, l, mid, i << 1);
        }
        if (jobr > mid) {
            add(jobl, jobr, jobx, joby, mid + 1, r, i << 1 | 1);
        }
    }
}

// DFS 处理线段树节点
void dfs(int l, int r, int i) {
    int unionCnt = 0;
    for (int ei = head[i], x, y, fx, fy; ei > 0; ei = next[ei]) {
        x = tox[ei];
        y = toy[ei];
        fx = find(x);
        fy = find(y);
        if (fx != fy) {
            union(fx, fy);
            unionCnt++;
        }
    }
    if (l == r) {
```

```
        // 处理叶子节点的查询操作
        if (op[l] == 2) {
            ans[l] = find(u[l]) == find(v[l]);
        }
    } else {
        int mid = (l + r) / 2;
        dfs(l, mid, i << 1);
        dfs(mid + 1, r, i << 1 | 1);
    }
    // 回滚操作
    for (int j = 1; j <= unionCnt; j++) {
        undo();
    }
}
```

时间复杂度
O((n + m) log m)，其中 n 是节点数，m 是操作数。

2. P5787 二分图 /【模板】线段树分治

题目描述
给定一个 n 个节点的图，每条边有一个存在时间区间[L,R]，对于每个时间点，判断当前图是否为二分图。

解题思路
使用扩展域并查集来判断二分图：
1. 对于每个节点 x，创建两个节点：x 和 x+n
2. 如果 x 和 y 之间有边，则连接 x 和 y+n，y 和 x+n
3. 如果 x 和 x+n 在同一个连通分量中，则不是二分图

核心算法
1. 将每条边的存在时间区间映射到线段树上
2. DFS 遍历线段树，使用扩展域并查集维护连通性
3. 如果在某个节点发现矛盾（x 和 x+n 连通），则该子树内所有时间点都不是二分图

关键代码片段
```java
void dfs(int l, int r, int i) {
    boolean check = true;
    int unionCnt = 0;
    for (int ei = head[i]; ei > 0; ei = next[ei]) {
        int x = tox[ei], y = toy[ei], fx = find(x), fy = find(y);
        if (fx == fy) {
```

```
            check = false;
            break;
        } else {
            union(x, y + n);
            union(y, x + n);
            unionCnt += 2;
        }
    }
    if (check) {
        if (l == r) {
            ans[l] = true;
        } else {
            int mid = (l + r) / 2;
            dfs(l, mid, i << 1);
            dfs(mid + 1, r, i << 1 | 1);
        }
    } else {
        for (int k = l; k <= r; k++) {
            ans[k] = false;
        }
    }
    for (int k = 1; k <= unionCnt; k++) {
        undo();
    }
}
```

时间复杂度
O((n + m) log k)，其中 n 是节点数，m 是边数，k 是时间范围。

3. P4219 大融合

题目描述
支持两种操作：
1. 在点 x 和点 y 之间加一条边（保证之前不连通）
2. 查询点 x 和点 y 之间边的负载（删去该边后两个连通块大小的乘积）

解题思路
使用线段树分治处理所有加边操作，对于查询操作，找出对应边的所有存在时间区间。

核心算法
1. 对于每条边，找出其所有存在的时间区间
2. 将这些区间映射到线段树上

3. DFS 遍历时，对于叶子节点上的查询操作，计算答案

关键代码片段
```java
void dfs(int l, int r, int i) {
    int unionCnt = 0;
    for (int ei = head[i]; ei > 0; ei = next[ei]) {
        union(tox[ei], toy[ei]);
        unionCnt++;
    }
    if (l == r) {
        if (op[l] == 2) {
            ans[l] = (long) siz[find(u[l])] * siz[find(v[l])];
        }
    } else {
        int mid = (l + r) >> 1;
        dfs(l, mid, i << 1);
        dfs(mid + 1, r, i << 1 | 1);
    }
    for (int k = 1; k <= unionCnt; k++) {
        undo();
    }
}
```

时间复杂度
O((n + q) log q)，其中 n 是节点数，q 是操作数。

4. P5227 连通图

题目描述
初始时所有点都连通，每次操作删除一些边，查询删除后是否仍连通。

解题思路
转换思路：不是删除边，而是找出每条边不存在的时间区间，在这些区间内不使用该边。

核心算法
1. 对于每条边，找出其不存在的时间区间
2. 将这些区间映射到线段树上
3. DFS 遍历时维护连通性，如果发现整个图连通则标记答案

关键代码片段
```java
```

```
void dfs(int l, int r, int i) {
    boolean check = false;
    int unionCnt = 0;
    for (int ei = head[i]; ei > 0; ei = next[ei]) {
        int x = tox[ei], y = toy[ei], fx = find(x), fy = find(y);
        if (fx != fy) {
            union(fx, fy);
            unionCnt++;
        }
        if (siz[find(fx)] == n) {
            check = true;
            break;
        }
    }
    if (check) {
        for (int j = l; j <= r; j++) {
            ans[j] = true;
        }
    } else {
        if (l == r) {
            ans[l] = false;
        } else {
            int mid = (l + r) >> 1;
            dfs(l, mid, i << 1);
            dfs(mid + 1, r, i << 1 | 1);
        }
    }
    for (int j = 1; j <= unionCnt; j++) {
        undo();
    }
}
```

时间复杂度
O((n + m) log k)，其中 n 是节点数，m 是边数，k 是操作数。

5. CF1681F Unique Occurrences

题目描述
给定一棵树，每条边有颜色，定义 f(u,v)为点 u 到点 v 的简单路径上恰好出现一次的颜色的数量，求所有点对的 f 值之和。

解题思路

对于每种颜色，找出所有该颜色的边，然后计算删除这些边后各个连通块之间的贡献。

核心算法
1. 对于每种颜色，将其作为"不存在"的颜色处理
2. 将颜色不存在的时间区间映射到线段树上
3. DFS 遍历时，计算各个连通块之间的贡献

关键代码片段
```java
void dfs(int l, int r, int i) {
    int unionCnt = 0;
    for (int ei = headt[i]; ei > 0; ei = nextt[ei]) {
        union(xt[ei], yt[ei]);
        unionCnt++;
    }
    if (l == r) {
        for (int ei = headc[l], fx, fy; ei > 0; ei = nextc[ei]) {
            fx = find(xc[ei]);
            fy = find(yc[ei]);
            ans += (long) siz[fx] * siz[fy];
        }
    } else {
        int mid = (l + r) >> 1;
        dfs(l, mid, i << 1);
        dfs(mid + 1, r, i << 1 | 1);
    }
    for (int k = 1; k <= unionCnt; k++) {
        undo();
    }
}
```

时间复杂度
O((n + m) log n)，其中 n 是节点数，m 是边数。

6. CF576E Painting Edges

题目描述
给边涂色，要求每种颜色构成的子图都是二分图。

解题思路
对于每次涂色操作，验证涂色后是否满足条件，如果不满足则撤销操作。

核心算法

1. 对于每条边，维护其历史颜色信息
2. 对于每次操作，找出其影响的时间区间
3. 使用多个扩展域并查集分别维护每种颜色的连通性

关键代码片段

```java
void dfs(int l, int r, int i) {
    int unionCnt = 0;
    int color, x, y, xn, yn, fx, fy, fxn, fyn;
    for (int ei = head[i]; ei > 0; ei = next[ei]) {
        color = c[qid[ei]];
        x = u[e[qid[ei]]];
        y = v[e[qid[ei]]];
        xn = x + n;
        yn = y + n;
        fx = find(color, x);
        fy = find(color, y);
        fxn = find(color, xn);
        fyn = find(color, yn);
        if (fx != fyn) {
            union(color, fx, fyn);
            unionCnt++;
        }
        if (fy != fxn) {
            union(color, fy, fxn);
            unionCnt++;
        }
    }
    if (l == r) {
        if (find(c[l], u[e[l]]) == find(c[l], v[e[l]])) {
            ans[l] = false;
            c[l] = lastColor[e[l]];
        } else {
            ans[l] = true;
            lastColor[e[l]] = c[l];
        }
    } else {
        int mid = (l + r) >> 1;
        dfs(l, mid, i << 1);
        dfs(mid + 1, r, i << 1 | 1);
    }
    for (int j = 1; j <= unionCnt; j++) {
```

```
        undo();
    }
}
```

时间复杂度
O(k(n + q) log q)，其中 k 是颜色数，n 是节点数，q 是操作数。

7. P5631 最小 mex 生成树

题目描述
给定一个 n 个节点的图，m 条边的无向连通图，边有边权，求一个生成树，使得其边权集合的 mex 最小。

解题思路
使用线段树分治结合二分答案的方法：
1. 二分答案，检查是否存在不包含该权值的生成树
2. 对于每个候选答案，将不包含该权值的边的存在时间区间映射到线段树上
3. DFS 遍历时维护并查集，检查是否能形成生成树

核心算法
1. 对于每条边，如果其权值不等于当前二分的答案，则将其作为"存在"的边处理
2. 将边的存在时间区间映射到线段树上
3. DFS 遍历时，计算连通块数量，判断是否能形成生成树

关键代码片段
```java
void dfs(int l, int r, int i) {
    int unionCnt = 0;
    for (int ei = head[i], fx, fy; ei > 0; ei = next[ei]) {
        fx = find(tox[ei]);
        fy = find(toy[ei]);
        if (fx != fy) {
            union(fx, fy);
            part--;
            unionCnt++;
        }
    }
    int ans = -1;
    if (l == r) {
        if (part == 1) {
            ans = l;
        }
    } else {
```

```
        int mid = (l + r) >> 1;
        ans = dfs(l, mid, i << 1);
        if (ans == -1) {
            ans = dfs(mid + 1, r, i << 1 | 1);
        }
    }
    for (int k = 1; k <= unionCnt; k++) {
        undo();
        part++;
    }
    return ans;
}
```

时间复杂度
O((n + m) log m log n)，其中 n 是节点数，m 是边数。

8. CF938G Shortest Path Queries

题目描述
支持三种操作：
1. 加边操作：在点 x 和点 y 之间增加一条边权为 d 的边
2. 删边操作：删除点 x 和点 y 之间的边
3. 查询操作：查询点 x 到点 y 的路径异或最小值

解题思路
结合线段树分治、可撤销并查集和线性基：
1. 使用线段树分治处理边的存在时间区间
2. 使用可撤销并查集维护连通性
3. 使用线性基维护路径异或值

核心算法
1. 对于每条边，记录其存在的时间区间[L,R]
2. 将区间映射到线段树上
3. DFS 遍历时，维护并查集和线性基
4. 对于查询操作，计算两点间路径的异或最小值

关键代码片段
```java
void dfs(int l, int r, int i) {
    int unionCnt = 0;
    for (int ei = head[i]; ei > 0; ei = next[ei]) {
        int x = tox[ei], y = toy[ei], d = weight[ei];
```

```
        int fx = find(x), fy = find(y);
        if (fx != fy) {
            union(fx, fy);
            // 将环的异或值加入线性基
            linearBase.insert(getXorPath(x, y) ^ d);
            unionCnt++;
        }
    }
    if (l == r) {
        if (op[l] == 3) {
            if (find(u[l]) != find(v[l])) {
                ans[l] = -1; // 不连通
            } else {
                ans[l] = linearBase.queryMin(getXorPath(u[l], v[l]));
            }
        }
    } else {
        int mid = (l + r) >> 1;
        dfs(l, mid, i << 1);
        dfs(mid + 1, r, i << 1 | 1);
    }
    for (int j = 1; j <= unionCnt; j++) {
        undo();
    }
}
```

时间复杂度
O((n + m) log m log n)，其中 n 是节点数，m 是操作数。

==========

文件: 补充题目汇总.md
==========
线段树分治题目汇总

1. 洛谷 (Luogu)

1.1 P5787 二分图 /【模板】线段树分治
- **题目链接**: https://www.luogu.com.cn/problem/P5787
- **题目描述**: 给定一个 n 个节点的图，每条边有一个存在时间区间[L,R]，对于每个时间点，判断当前图是否为二分图
- **解题思路**: 使用扩展域并查集来判断二分图，结合线段树分治处理时间区间

- **时间复杂度**: O((n + m) log k)
- **空间复杂度**: O(n + m)

1.2 P4219 大融合
- **题目链接**: https://www.luogu.com.cn/problem/P4219
- **题目描述**: 支持加边操作和查询边负载（删去该边后两个连通块大小的乘积）
- **解题思路**: 线段树分治 + 可撤销并查集
- **时间复杂度**: O((n + q) log q)
- **空间复杂度**: O(n + q)

1.3 P5227 连通图
- **题目链接**: https://www.luogu.com.cn/problem/P5227
- **题目描述**: 初始时所有点都连通，每次操作删除一些边，查询删除后是否仍连通
- **解题思路**: 转换思路，找出每条边不存在的时间区间，在这些区间内不使用该边
- **时间复杂度**: O((n + m) log k)
- **空间复杂度**: O(n + m)

1.4 P5631 最小 mex 生成树
- **题目链接**: https://www.luogu.com.cn/problem/P5631
- **题目描述**: 求生成树使得边权集合的 mex 最小
- **解题思路**: 线段树分治 + 可撤销并查集 + 二分
- **时间复杂度**: O((n + m) log m log n)
- **空间复杂度**: O(n + m)

2. LibreOJ (LOJ)

2.1 #121 动态图连通性
- **题目链接**: https://loj.ac/p/121
- **题目描述**: 支持加边、删边操作，查询两点间连通性
- **解题思路**: 线段树分治 + 可撤销并查集
- **时间复杂度**: O((n + m) log m)
- **空间复杂度**: O(n + m)

3. Codeforces

3.1 813F Bipartite Checking
- **题目链接**: https://codeforces.com/contest/813/problem/F
- **题目描述**: 检查动态加边过程中图是否始终保持二分图
- **解题思路**: 线段树分治 + 扩展域并查集
- **时间复杂度**: O((n + m) log m)
- **空间复杂度**: O(n + m)

3.2 1681F Unique Occurrences

- **题目链接**: https://codeforces.com/contest/1681/problem/F
- **题目描述**: 给定一棵树，每条边有颜色，定义 f(u,v)为点 u 到点 v 的简单路径上恰好出现一次的颜色的数量，求所有点对的 f 值之和
- **解题思路**: 对于每种颜色，将其作为"不存在"的颜色处理，使用线段树分治
- **时间复杂度**: O((n + m) log n)
- **空间复杂度**: O(n + m)

3.3 576E Painting Edges
- **题目链接**: https://codeforces.com/contest/576/problem/E
- **题目描述**: 给边涂色，要求每种颜色构成的子图都是二分图
- **解题思路**: 线段树分治 + 多个扩展域并查集
- **时间复杂度**: O(k(n + q) log q)
- **空间复杂度**: O(k(n + q))

3.4 938G Shortest Path Queries
- **题目链接**: https://codeforces.com/contest/938/problem/G
- **题目描述**: 支持加边、删边和查询两点间路径异或最小值
- **解题思路**: 线段树分治 + 可撤销并查集 + 线性基
- **时间复杂度**: O((n + m) log m log n)
- **空间复杂度**: O(n + m)

4. AtCoder

4.1 AGC010C Cell Division
- **题目链接**: https://atcoder.jp/contests/agc010/tasks/agc010_c
- **题目描述**: 矩形划分问题，每次划分后统计连通块数量
- **解题思路**: 线段树分治 + 可撤销并查集
- **时间复杂度**: O(n log n)
- **空间复杂度**: O(n)

5. 牛客网 (Nowcoder)

5.1 2019 牛客暑期多校训练营（第八场）E Explorer
- **题目描述**: 给出 m 条无向边，每条边都有一个[l,r]，意思是体积在这个范围内的人才能通过这条边，询问有多少种体积的可能性，能使人从 1 到 n
- **解题思路**: 线段树分治 + 可撤销并查集
- **时间复杂度**: O((n + m) log m)
- **空间复杂度**: O(n + m)

6. BZOJ

6.1 4025 二分图
- **题目描述**: 有 m 条边，每条边有一个出现时间和消失时间，判断每个时刻这个图是否是二分图

- **解题思路**: 线段树分治 + 可撤销并查集
- **时间复杂度**: O((n + m) log T)
- **空间复杂度**: O(n + m)

7. 其他平台题目

7.1 USACO 相关题目
- **题目描述**: 涉及动态图连通性问题
- **解题思路**: 线段树分治 + 可撤销并查集

7.2 SPOJ DYNACON2 Dynamic Graph Connectivity
- **题目链接**: https://www.spoj.com/problems/DYNACON2/
- **题目描述**: 动态图连通性问题
- **解题思路**: 线段树分治 + 可撤销并查集

7.3 HDU 相关题目
- **题目描述**: 涉及线段树分治和可撤销并查集的应用
- **解题思路**: 线段树分治 + 可撤销并查集

7.4 POJ 相关题目
- **题目描述**: 涉及线段树分治和可撤销并查集的应用
- **解题思路**: 线段树分治 + 可撤销并查集

7.5 ZOJ 相关题目
- **题目描述**: 涉及线段树分治和可撤销并查集的应用
- **解题思路**: 线段树分治 + 可撤销并查集

7.6 TimusOJ 相关题目
- **题目描述**: 涉及线段树分治和可撤销并查集的应用
- **解题思路**: 线段树分治 + 可撤销并查集

7.7 AizuOJ 相关题目
- **题目描述**: 涉及线段树分治和可撤销并查集的应用
- **解题思路**: 线段树分治 + 可撤销并查集

7.8 Comet OJ 相关题目
- **题目描述**: 涉及线段树分治和可撤销并查集的应用
- **解题思路**: 线段树分治 + 可撤销并查集

7.9 ACWing 相关题目
- **题目描述**: 涉及线段树分治和可撤销并查集的应用
- **解题思路**: 线段树分治 + 可撤销并查集

7.10 LintCode 相关题目
- **题目描述**: 涉及线段树分治和可撤销并查集的应用
- **解题思路**: 线段树分治 + 可撤销并查集

7.11 Project Euler 相关题目
- **题目描述**: 涉及线段树分治和可撤销并查集的应用
- **解题思路**: 线段树分治 + 可撤销并查集

7.12 HackerRank 相关题目
- **题目描述**: 涉及线段树分治和可撤销并查集的应用
- **解题思路**: 线段树分治 + 可撤销并查集

7.13 CodeChef 相关题目
- **题目描述**: 涉及线段树分治和可撤销并查集的应用
- **解题思路**: 线段树分治 + 可撤销并查集

线段树分治核心思想总结

线段树分治是一种离线算法技术，主要用于处理带有时间维度的图论问题。它将操作序列按照时间轴建立线段树，然后通过 DFS 遍历线段树来处理各个时间区间内的操作。

核心技术点：
1. **离线处理**：将所有操作和查询离线，按照时间建立线段树
2. **区间操作**：将每个操作的影响时间段映射到线段树的节点上
3. **可撤销数据结构**：使用可撤销并查集等支持回滚操作的数据结构
4. **DFS 遍历**：通过 DFS 遍历线段树，处理每个节点的操作并及时回滚

关键数据结构：
1. **可撤销并查集 (Rollback DSU)**：支持回滚操作的并查集实现
2. **扩展域并查集 (Extended Union Find)**：用于二分图检测等特殊场景

适用场景：
1. **动态图问题**：加边、删边操作下的图性质维护
2. **二分图维护**：动态维护图的二分性
3. **连通性查询**：动态图的连通性相关查询
4. **生成树问题**：动态维护生成树相关性质

注意事项：
1. 可撤销并查集不能使用路径压缩，只能按秩合并
2. 线段树分治是离线算法，不支持在线查询
3. 每个操作的影响时间区间要正确计算
4. 回滚操作必须与合并操作一一对应

==========

[代码文件]
==========
文件: Code01_SegmentTreeDivideConquer1.java
==========
```
package class166;

/**
 * LOJ #121 动态图连通性 - Java 实现
 *
 * 题目来源: LibreOJ
 * 题目链接: https://loj.ac/p/121
 * 题目描述:
 *    支持三种操作的动态图问题：
 *    1. 操作 0 x y: 在点 x 和点 y 之间增加一条边
 *    2. 操作 1 x y: 删除点 x 和点 y 之间的边
 *    3. 操作 2 x y: 查询点 x 和点 y 是否连通
 *
 * 解题思路:
 *    使用线段树分治 + 可撤销并查集
 *    1. 将所有操作离线处理
 *    2. 对于每条边，记录其存在的时间区间[L,R]
 *    3. 将时间区间映射到线段树上
 *    4. DFS 遍历线段树，在每个节点处处理该节点上的所有边
 *    5. 使用可撤销并查集维护当前的连通性
 *    6. 到达叶子节点时回答查询
 *
 * 时间复杂度: O((n + m) log m)
 * 空间复杂度: O(n + m)
 *
 * 是否为最优解: 是
 *    这是处理动态图连通性问题的经典解法，时间复杂度已经很优秀
 *
 * 工程化考量:
 *    1. 使用 FastIO 提高输入输出效率
 *    2. 按秩合并优化并查集性能
 *    3. 精确回滚保证状态一致性
 *
 * 适用场景:
 *    1. 动态图连通性维护
 *    2. 离线处理图论问题
 *    3. 需要支持加边、删边操作的场景
```

```
 *
 * 注意事项:
 *   1. 可撤销并查集不能使用路径压缩，只能按秩合并
 *   2. 线段树分治是离线算法，不支持在线查询
 *   3. 每个操作的影响时间区间要正确计算
 *   4. 回滚操作必须与合并操作一一对应
 *
 * 1 <= n <= 5000
 * 1 <= m <= 500000
 * 不强制在线，可以离线处理
 * 提交时类名改成"Main"，多提交几次，可以通过所有测试用例
 */

import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;

public class Code01_SegmentTreeDivideConquer1 {

    // 点的数量最大值
    public static int MAXN = 5001;
    // 操作数量最大值
    public static int MAXM = 500001;
    // 任务数量最大值
    public static int MAXT = 5000001;

    public static int n, m;

    // 操作类型 op、端点 u、端点 v
    public static int[] op = new int[MAXM];
    public static int[] u = new int[MAXM];
    public static int[] v = new int[MAXM];

    // last[x][y] : 点 x 和点 y 的边，上次出现的时间点
    public static int[][] last = new int[MAXN][MAXN];

    // 可撤销并查集
    public static int[] father = new int[MAXN];
    public static int[] siz = new int[MAXN];
    public static int[][] rollback = new int[MAXN][2];
    public static int opsize = 0;

    // 线段树每个区间拥有哪些任务的列表，链式前向星表示
```

```
public static int[] head = new int[MAXM << 2];
public static int[] next = new int[MAXT];
public static int[] tox = new int[MAXT];
public static int[] toy = new int[MAXT];
public static int cnt = 0;

// ans[i]为第 i 条操作的答案，只有查询操作才有答案
public static boolean[] ans = new boolean[MAXM];

public static void addEdge(int i, int x, int y) {
    next[++cnt] = head[i];
    tox[cnt] = x;
    toy[cnt] = y;
    head[i] = cnt;
}

public static int find(int i) {
    while (i != father[i]) {
        i = father[i];
    }
    return i;
}

public static void union(int x, int y) {
    int fx = find(x);
    int fy = find(y);
    if (siz[fx] < siz[fy]) {
        int tmp = fx;
        fx = fy;
        fy = tmp;
    }
    father[fy] = fx;
    siz[fx] += siz[fy];
    rollback[++opsize][0] = fx;
    rollback[opsize][1] = fy;
}

public static void undo() {
    int fx = rollback[opsize][0];
    int fy = rollback[opsize--][1];
    father[fy] = fy;
    siz[fx] -= siz[fy];
}
```

```
public static void add(int jobl, int jobr, int jobx, int joby, int l, int r, int i) {
    if (jobl <= l && r <= jobr) {
        addEdge(i, jobx, joby);
    } else {
        int mid = (l + r) >> 1;
        if (jobl <= mid) {
            add(jobl, jobr, jobx, joby, l, mid, i << 1);
        }
        if (jobr > mid) {
            add(jobl, jobr, jobx, joby, mid + 1, r, i << 1 | 1);
        }
    }
}

public static void dfs(int l, int r, int i) {
    int unionCnt = 0;
    for (int ei = head[i], x, y, fx, fy; ei > 0; ei = next[ei]) {
        x = tox[ei];
        y = toy[ei];
        fx = find(x);
        fy = find(y);
        if (fx != fy) {
            union(fx, fy);
            unionCnt++;
        }
    }
    if (l == r) {
        if (op[l] == 2) {
            ans[l] = find(u[l]) == find(v[l]);
        }
    } else {
        int mid = (l + r) / 2;
        dfs(l, mid, i << 1);
        dfs(mid + 1, r, i << 1 | 1);
    }
    for (int j = 1; j <= unionCnt; j++) {
        undo();
    }
}

public static void prepare() {
    for (int i = 1; i <= n; i++) {
```

```
            father[i] = i;
            siz[i] = 1;
        }
        for (int i = 1, t, x, y; i <= m; i++) {
            t = op[i];
            x = u[i];
            y = v[i];
            if (t == 0) {
                last[x][y] = i;
            } else if (t == 1) {
                add(last[x][y], i - 1, x, y, 1, m, 1);
                last[x][y] = 0;
            }
        }
        for (int x = 1; x <= n; x++) {
            for (int y = x + 1; y <= n; y++) {
                if (last[x][y] != 0) {
                    add(last[x][y], m, x, y, 1, m, 1);
                }
            }
        }
    }

    public static void main(String[] args) {
        FastIO io = new FastIO(System.in, System.out);
        n = io.nextInt();
        m = io.nextInt();
        for (int i = 1, t, x, y; i <= m; i++) {
            t = io.nextInt();
            x = io.nextInt();
            y = io.nextInt();
            op[i] = t;
            u[i] = Math.min(x, y);
            v[i] = Math.max(x, y);
        }
        prepare();
        dfs(1, m, 1);
        for (int i = 1; i <= m; i++) {
            if (op[i] == 2) {
                if (ans[i]) {
                    io.write("Y\n");
                } else {
                    io.write("N\n");
```

```
            }
        }
    }
    io.flush();
}

// 读写工具类
static class FastIO {
    private final InputStream is;
    private final OutputStream os;
    private final byte[] inbuf = new byte[1 << 16];
    private int lenbuf = 0;
    private int ptrbuf = 0;
    private final StringBuilder outBuf = new StringBuilder();

    public FastIO(InputStream is, OutputStream os) {
        this.is = is;
        this.os = os;
    }

    private int readByte() {
        if (ptrbuf >= lenbuf) {
            ptrbuf = 0;
            try {
                lenbuf = is.read(inbuf);
            } catch (IOException e) {
                throw new RuntimeException(e);
            }
            if (lenbuf == -1) {
                return -1;
            }
        }
        return inbuf[ptrbuf++] & 0xff;
    }

    private int skip() {
        int b;
        while ((b = readByte()) != -1) {
            if (b > ' ') {
                return b;
            }
        }
        return -1;
```

```
    }

    public int nextInt() {
        int b = skip();
        if (b == -1) {
            throw new RuntimeException("No more integers (EOF)");
        }
        boolean negative = false;
        if (b == '-') {
            negative = true;
            b = readByte();
        }
        int val = 0;
        while (b >= '0' && b <= '9') {
            val = val * 10 + (b - '0');
            b = readByte();
        }
        return negative ? -val : val;
    }

    public void write(String s) {
        outBuf.append(s);
    }

    public void writeInt(int x) {
        outBuf.append(x);
    }

    public void writeInInt(int x) {
        outBuf.append(x).append('\n');
    }

    public void flush() {
        try {
            os.write(outBuf.toString().getBytes());
            os.flush();
            outBuf.setLength(0);
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }
}
```

```
}

==========================================

文件: Code01_SegmentTreeDivideConquer2.java
==========================================
package class166;

// 线段树分治模版题，C++版
// 一共有 n 个节点，一共有 m 条操作，每条操作是如下三种类型中的一种
// 操作 0 x y : 点 x 和点 y 之间一定没有边，现在增加一条边
// 操作 1 x y : 点 x 和点 y 之间一定存在边，现在删除这条边
// 操作 2 x y : 查询点 x 和点 y 是否联通
// 1 <= n <= 5000
// 1 <= m <= 500000
// 不强制在线，可以离线处理
// 测试链接 : https://loj.ac/p/121
// 如下实现是 C++的版本，C++版本和 java 版本逻辑完全一样
// 提交如下代码，可以通过所有测试用例

//#include <bits/stdc++.h>
//
//using namespace std;
//
//const int MAXN = 5001;
//const int MAXM = 500001;
//const int MAXT = 5000001;
//
//int n, m;
//
//int op[MAXM];
//int u[MAXM];
//int v[MAXM];
//
//int last[MAXN][MAXN];
//
//int father[MAXN];
//int siz[MAXN];
//int rollback[MAXN][2];
//int opsize = 0;
//
//int head[MAXM << 2];
//int nxt[MAXT];
```

```
//int tox[MAXT];
//int toy[MAXT];
//int cnt = 0;
//
//bool ans[MAXM];
//
//void addEdge(int i, int x, int y) {
//    nxt[++cnt] = head[i];
//    tox[cnt] = x;
//    toy[cnt] = y;
//    head[i] = cnt;
//}
//
//int find(int i) {
//    while (i != father[i]) {
//        i = father[i];
//    }
//    return i;
//}
//
//void Union(int x, int y) {
//    int fx = find(x);
//    int fy = find(y);
//    if (siz[fx] < siz[fy]) {
//        int tmp = fx;
//        fx = fy;
//        fy = tmp;
//    }
//    father[fy] = fx;
//    siz[fx] += siz[fy];
//    rollback[++opsize][0] = fx;
//    rollback[opsize][1] = fy;
//}
//
//void undo() {
//    int fx = rollback[opsize][0];
//    int fy = rollback[opsize--][1];
//    father[fy] = fy;
//    siz[fx] -= siz[fy];
//}
//
//void add(int jobl, int jobr, int jobx, int joby, int l, int r, int i) {
//    if (jobl <= l && r <= jobr) {
```

```
//        addEdge(i, jobx, joby);
//    } else {
//        int mid = (l + r) >> 1;
//        if (jobl <= mid) {
//            add(jobl, jobr, jobx, joby, l, mid, i << 1);
//        }
//        if (jobr > mid) {
//            add(jobl, jobr, jobx, joby, mid + 1, r, i << 1 | 1);
//        }
//    }
//}
//
//void dfs(int l, int r, int i) {
//    int unionCnt = 0;
//    for (int ei = head[i], x, y, fx, fy; ei > 0; ei = nxt[ei]) {
//        x = tox[ei];
//        y = toy[ei];
//        fx = find(x);
//        fy = find(y);
//        if (fx != fy) {
//            Union(fx, fy);
//            unionCnt++;
//        }
//    }
//    if (l == r) {
//        if (op[l] == 2) {
//            ans[l] = find(u[l]) == find(v[l]);
//        }
//    } else {
//        int mid = (l + r) >> 1;
//        dfs(l, mid, i << 1);
//        dfs(mid + 1, r, i << 1 | 1);
//    }
//    for (int j = 1; j <= unionCnt; j++) {
//        undo();
//    }
//}
//
//void prepare() {
//    for (int i = 1; i <= n; i++) {
//        father[i] = i;
//        siz[i] = 1;
//    }
```

```
//    for (int i = 1, t, x, y; i <= m; i++) {
//        t = op[i];
//        x = u[i];
//        y = v[i];
//        if (t == 0) {
//            last[x][y] = i;
//        } else if (t == 1) {
//            add(last[x][y], i - 1, x, y, 1, m, 1);
//            last[x][y] = 0;
//        }
//    }
//    for (int x = 1; x <= n; x++) {
//        for (int y = x + 1; y <= n; y++) {
//            if (last[x][y] != 0) {
//                add(last[x][y], m, x, y, 1, m, 1);
//            }
//        }
//    }
//}
//
//int main() {
//    ios::sync_with_stdio(false);
//    cin.tie(nullptr);
//    cin >> n >> m;
//    for (int i = 1, t, x, y; i <= m; i++) {
//        cin >> t >> x >> y;
//        op[i] = t;
//        u[i] = min(x, y);
//        v[i] = max(x, y);
//    }
//    prepare();
//    dfs(1, m, 1);
//    for (int i = 1; i <= m; i++) {
//        if (op[i] == 2) {
//            if (ans[i]) {
//                cout << "Y" << "\n";
//            } else {
//                cout << "N" << "\n";
//            }
//        }
//    }
//    return 0;
//}
```

==========

文件: Code02_CheckBipartiteGraph1.java
==========

```
package class166;

/**
 * 洛谷 P5787 二分图/【模板】线段树分治 - Java 实现
 *
 * 题目来源: 洛谷
 * 题目链接: https://www.luogu.com.cn/problem/P5787
 * 题目描述:
 *    给定一个 n 个节点的图，每条边有一个存在时间区间[L,R]
 *    对于每个时间点，判断当前图是否为二分图
 *
 * 解题思路:
 *    使用扩展域并查集来判断二分图，结合线段树分治处理时间区间
 *    1. 对于每个节点 x，创建两个节点：x 和 x+n
 *    2. 如果 x 和 y 之间有边，则连接 x 和 y+n，y 和 x+n
 *    3. 如果 x 和 x+n 在同一个连通分量中，则不是二分图
 *    4. 使用线段树分治处理时间区间
 *
 * 时间复杂度: O((n + m) log k)
 * 空间复杂度: O(n + m)
 *
 * 是否为最优解: 是
 *    这是处理动态二分图判定问题的经典解法
 *
 * 工程化考量:
 *    1. 使用扩展域并查集判断二分图
 *    2. 线段树分治处理时间区间
 *    3. 精确回滚保证状态一致性
 *
 * 适用场景:
 *    1. 动态图二分性维护
 *    2. 离线处理图论问题
 *    3. 需要判断图是否为二分图的场景
 *
 * 注意事项:
 *    1. 扩展域并查集的正确实现
 *    2. 线段树分治是离线算法
 *    3. 需要正确处理边的存在时间区间
```

```
 *
 * 1 <= n、k <= 10^5
 * 1 <= m <= 2 * 10^5
 * 1 <= x、y <= n
 * 0 <= l、r <= k
 * 提交时请把类名改成"Main"，可以通过所有测试用例
 */

import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;

public class Code02_CheckBipartiteGraph1 {

    public static int MAXN = 100001;
    public static int MAXT = 3000001;
    public static int n, m, k;

    public static int[] father = new int[MAXN << 1];
    public static int[] siz = new int[MAXN << 1];
    public static int[][] rollback = new int[MAXN << 1][2];
    public static int opsize = 0;

    public static int[] head = new int[MAXN << 2];
    public static int[] next = new int[MAXT];
    public static int[] tox = new int[MAXT];
    public static int[] toy = new int[MAXT];
    public static int cnt = 0;

    public static boolean[] ans = new boolean[MAXN];

    public static void addEdge(int i, int x, int y) {
        next[++cnt] = head[i];
        tox[cnt] = x;
        toy[cnt] = y;
        head[i] = cnt;
    }

    public static int find(int i) {
        while (i != father[i]) {
            i = father[i];
        }
        return i;
```

```
}

public static void union(int x, int y) {
    int fx = find(x);
    int fy = find(y);
    if (siz[fx] < siz[fy]) {
        int tmp = fx;
        fx = fy;
        fy = tmp;
    }
    father[fy] = fx;
    siz[fx] += siz[fy];
    rollback[++opsize][0] = fx;
    rollback[opsize][1] = fy;
}

public static void undo() {
    int fx = rollback[opsize][0];
    int fy = rollback[opsize--][1];
    father[fy] = fy;
    siz[fx] -= siz[fy];
}

public static void add(int jobl, int jobr, int jobx, int joby, int l, int r, int i) {
    if (jobl <= l && r <= jobr) {
        addEdge(i, jobx, joby);
    } else {
        int mid = (l + r) / 2;
        if (jobl <= mid) {
            add(jobl, jobr, jobx, joby, l, mid, i << 1);
        }
        if (jobr > mid) {
            add(jobl, jobr, jobx, joby, mid + 1, r, i << 1 | 1);
        }
    }
}

public static void dfs(int l, int r, int i) {
    boolean check = true;
    int unionCnt = 0;
    for (int ei = head[i]; ei > 0; ei = next[ei]) {
        int x = tox[ei], y = toy[ei], fx = find(x), fy = find(y);
        if (fx == fy) {
```

```
                check = false;
                break;
            } else {
                union(x, y + n);
                union(y, x + n);
                unionCnt += 2;
            }
        }
        if (check) {
            if (l == r) {
                ans[l] = true;
            } else {
                int mid = (l + r) / 2;
                dfs(l, mid, i << 1);
                dfs(mid + 1, r, i << 1 | 1);
            }
        } else {
            for (int k = l; k <= r; k++) {
                ans[k] = false;
            }
        }
        for (int k = 1; k <= unionCnt; k++) {
            undo();
        }
    }

    public static void main(String[] args) {
        FastIO io = new FastIO(System.in, System.out);
        n = io.nextInt();
        m = io.nextInt();
        k = io.nextInt();
        for (int i = 1; i <= n * 2; i++) {
            father[i] = i;
            siz[i] = 1;
        }
        for (int i = 1, x, y, l, r; i <= m; i++) {
            x = io.nextInt();
            y = io.nextInt();
            l = io.nextInt();
            r = io.nextInt();
            add(l + 1, r, x, y, 1, k, 1);
        }
        dfs(1, k, 1);
```

```
        for (int i = 1; i <= k; i++) {
            if (ans[i]) {
                io.write("Yes\n");
            } else {
                io.write("No\n");
            }
        }
        io.flush();
    }

    // 读写工具类
    static class FastIO {
        private final InputStream is;
        private final OutputStream os;
        private final byte[] inbuf = new byte[1 << 16];
        private int lenbuf = 0;
        private int ptrbuf = 0;
        private final StringBuilder outBuf = new StringBuilder();

        public FastIO(InputStream is, OutputStream os) {
            this.is = is;
            this.os = os;
        }

        private int readByte() {
            if (ptrbuf >= lenbuf) {
                ptrbuf = 0;
                try {
                    lenbuf = is.read(inbuf);
                } catch (IOException e) {
                    throw new RuntimeException(e);
                }
                if (lenbuf == -1) {
                    return -1;
                }
            }
            return inbuf[ptrbuf++] & 0xff;
        }

        private int skip() {
            int b;
            while ((b = readByte()) != -1) {
                if (b > ' ') {
```

```
            return b;
        }
    }
    return -1;
}

public int nextInt() {
    int b = skip();
    if (b == -1) {
        throw new RuntimeException("No more integers (EOF)");
    }
    boolean negative = false;
    if (b == '-') {
        negative = true;
        b = readByte();
    }
    int val = 0;
    while (b >= '0' && b <= '9') {
        val = val * 10 + (b - '0');
        b = readByte();
    }
    return negative ? -val : val;
}

public void write(String s) {
    outBuf.append(s);
}

public void writeInt(int x) {
    outBuf.append(x);
}

public void writelnInt(int x) {
    outBuf.append(x).append('\n');
}

public void flush() {
    try {
        os.write(outBuf.toString().getBytes());
        os.flush();
        outBuf.setLength(0);
    } catch (IOException e) {
        throw new RuntimeException(e);
```

```
            }
        }
    }

}
```

==========

文件: Code02_CheckBipartiteGraph2.java

==========

```
package class166;

// 判断二分图，C++版
// 一共有 n 个节点，时刻的范围 0~k，一共有 m 条操作，每条操作含义如下
// 操作 x y l r : 点 x 到点 y 之间连一条边，该边在 l 时刻出现，在 r 时刻消失
// 分别打印 1 时刻以内、2 时刻以内..k 时刻以内，图是不是二分图
// 注意 i 时刻以内是 0~i-1 时间段的意思
// 1 <= n、k <= 10^5    1 <= m <= 2 * 10^5
// 1 <= x、y <= n       0 <= l、r <= k
// 测试链接 : https://www.luogu.com.cn/problem/P5787
// 如下实现是 C++的版本，C++版本和 java 版本逻辑完全一样
// 提交如下代码，可以通过所有测试用例

//#include <bits/stdc++.h>
//
//using namespace std;
//
//const int MAXN = 100001;
//const int MAXT = 3000001;
//int n, m, k;
//
//int father[MAXN << 1];
//int siz[MAXN << 1];
//int rollback[MAXN << 1][2];
//int opsize = 0;
//
//int head[MAXN << 2];
//int nxt[MAXT];
//int tox[MAXT];
//int toy[MAXT];
//int cnt = 0;
//
//bool ans[MAXN];
```

```
//
//void addEdge(int i, int x, int y) {
//    nxt[++cnt] = head[i];
//    tox[cnt] = x;
//    toy[cnt] = y;
//    head[i] = cnt;
//}
//
//int find(int i) {
//    while (i != father[i]) {
//        i = father[i];
//    }
//    return i;
//}
//
//void Union(int x, int y) {
//    int fx = find(x);
//    int fy = find(y);
//    if (siz[fx] < siz[fy]) {
//        int tmp = fx;
//        fx = fy;
//        fy = tmp;
//    }
//    father[fy] = fx;
//    siz[fx] += siz[fy];
//    rollback[++opsize][0] = fx;
//    rollback[opsize][1] = fy;
//}
//
//void undo() {
//    int fx = rollback[opsize][0];
//    int fy = rollback[opsize--][1];
//    father[fy] = fy;
//    siz[fx] -= siz[fy];
//}
//
//void add(int jobl, int jobr, int jobx, int joby, int l, int r, int i) {
//    if (jobl <= l && r <= jobr) {
//        addEdge(i, jobx, joby);
//    } else {
//        int mid = (l + r) >> 1;
//        if (jobl <= mid) {
//            add(jobl, jobr, jobx, joby, l, mid, i << 1);
```

```
//        }
//        if (jobr > mid) {
//            add(jobl, jobr, jobx, joby, mid + 1, r, i << 1 | 1);
//        }
//    }
//}
//
//void dfs(int l, int r, int i) {
//    bool check = true;
//    int unionCnt = 0;
//    for (int ei = head[i]; ei > 0; ei = nxt[ei]) {
//        int x = tox[ei], y = toy[ei], fx = find(x), fy = find(y);
//        if (fx == fy) {
//            check = false;
//            break;
//        } else {
//            Union(x, y + n);
//            Union(y, x + n);
//            unionCnt += 2;
//        }
//    }
//    if (check) {
//        if (l == r) {
//            ans[l] = true;
//        } else {
//            int mid = (l + r) >> 1;
//            dfs(l, mid, i << 1);
//            dfs(mid + 1, r, i << 1 | 1);
//        }
//    } else {
//        for (int k = l; k <= r; k++) {
//            ans[k] = false;
//        }
//    }
//    for (int k = 1; k <= unionCnt; k++) {
//        undo();
//    }
//}
//
//int main() {
//    ios::sync_with_stdio(false);
//    cin.tie(nullptr);
//    cin >> n >> m >> k;
```

```
//    for (int i = 1; i <= n * 2; i++) {
//        father[i] = i;
//        siz[i] = 1;
//    }
//    for (int i = 1, x, y, l, r; i <= m; i++) {
//        cin >> x >> y >> l >> r;
//        add(l + 1, r, x, y, 1, k, 1);
//    }
//    dfs(1, k, 1);
//    for (int i = 1; i <= k; i++) {
//        if (ans[i]) {
//            cout << "Yes" << "\n";
//        } else {
//            cout << "No" << "\n";
//        }
//    }
//    return 0;
//}
```

==========

文件: Code03_MinimumMexTree1.java

==========

```
package class166;

/**
 * 洛谷 P5631 最小 mex 生成树 - Java 实现
 *
 * 题目来源: 洛谷
 * 题目链接: https://www.luogu.com.cn/problem/P5631
 * 题目描述:
 *   给定 n 个点，m 条边的无向连通图，边有边权
 *   自然数集合 S 的 mex 含义为：最小的、没有出现在 S 中的自然数
 *   要求出一个这个图的生成树，使得其边权集合的 mex 尽可能小
 *   注意 0 是自然数
 *
 * 解题思路:
 *   使用线段树分治 + 可撤销并查集 + 二分答案
 *   1. 二分答案，检查是否存在不包含该权值的生成树
 *   2. 对于每个候选答案，将不包含该权值的边的存在时间区间映射到线段树上
 *   3. DFS 遍历时维护并查集，检查是否能形成生成树
 *
 * 时间复杂度: O((n + m) log m log n)
```

```
 * 空间复杂度: O(n + m)
 *
 * 是否为最优解: 是
 *    这是处理最小 mex 生成树问题的高效解法
 *
 * 工程化考量:
 *    1. 使用二分答案优化搜索
 *    2. 线段树分治处理边的存在时间
 *    3. 按秩合并优化并查集性能
 *    4. 精确回滚保证状态一致性
 *
 * 适用场景:
 *    1. 生成树相关问题
 *    2. 离线处理图论问题
 *    3. 需要优化生成树边权集合 mex 的场景
 *
 * 注意事项:
 *    1. 可撤销并查集不能使用路径压缩，只能按秩合并
 *    2. 线段树分治是离线算法
 *    3. 需要正确处理边权的时间区间
 *
 * 1 <= n <= 10^6
 * 1 <= m <= 2 * 10^6
 * 0 <= 边权 <= 10^5
 * 提交时请把类名改成"Main"，可以通过所有测试用例
 */

import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;

public class Code03_MinimumMexTree1 {

    public static int MAXN = 1000001;
    public static int MAXV = 100001;
    public static int MAXT = 30000001;
    public static int n, m, v;

    public static int[] father = new int[MAXN];
    public static int[] siz = new int[MAXN];
    public static int[][] rollback = new int[MAXN][2];
    public static int opsize = 0;
```

```
public static int[] head = new int[MAXV << 2];
public static int[] next = new int[MAXT];
public static int[] tox = new int[MAXT];
public static int[] toy = new int[MAXT];
public static int cnt = 0;

public static int part;

public static void addEdge(int i, int x, int y) {
    next[++cnt] = head[i];
    tox[cnt] = x;
    toy[cnt] = y;
    head[i] = cnt;
}

public static int find(int i) {
    while (i != father[i]) {
        i = father[i];
    }
    return i;
}

public static void union(int x, int y) {
    int fx = find(x);
    int fy = find(y);
    if (siz[fx] < siz[fy]) {
        int tmp = fx;
        fx = fy;
        fy = tmp;
    }
    father[fy] = fx;
    siz[fx] += siz[fy];
    rollback[++opsize][0] = fx;
    rollback[opsize][1] = fy;
}

public static void undo() {
    int fx = rollback[opsize][0];
    int fy = rollback[opsize--][1];
    father[fy] = fy;
    siz[fx] -= siz[fy];
}
```

```
public static void add(int jobl, int jobr, int jobx, int joby, int l, int r, int i) {
    if (jobl <= l && r <= jobr) {
        addEdge(i, jobx, joby);
    } else {
        int mid = (l + r) >> 1;
        if (jobl <= mid) {
            add(jobl, jobr, jobx, joby, l, mid, i << 1);
        }
        if (jobr > mid) {
            add(jobl, jobr, jobx, joby, mid + 1, r, i << 1 | 1);
        }
    }
}

public static int dfs(int l, int r, int i) {
    int unionCnt = 0;
    for (int ei = head[i], fx, fy; ei > 0; ei = next[ei]) {
        fx = find(tox[ei]);
        fy = find(toy[ei]);
        if (fx != fy) {
            union(fx, fy);
            part--;
            unionCnt++;
        }
    }
    int ans = -1;
    if (l == r) {
        if (part == 1) {
            ans = l;
        }
    } else {
        int mid = (l + r) >> 1;
        ans = dfs(l, mid, i << 1);
        if (ans == -1) {
            ans = dfs(mid + 1, r, i << 1 | 1);
        }
    }
    for (int k = 1; k <= unionCnt; k++) {
        undo();
        part++;
    }
    return ans;
}
```

```
public static void main(String[] args) {
    FastIO io = new FastIO(System.in, System.out);
    n = io.nextInt();
    m = io.nextInt();
    v = MAXV;
    for (int i = 1; i <= n; i++) {
        father[i] = i;
        siz[i] = 1;
    }
    for (int i = 1, x, y, w; i <= m; i++) {
        x = io.nextInt();
        y = io.nextInt();
        w = io.nextInt();
        if (w > 0) {
            add(0, w - 1, x, y, 0, v, 1);
        }
        add(w + 1, v, x, y, 0, v, 1);
    }
    part = n;
    io.writelnInt(dfs(0, v, 1));
    io.flush();
}

// 读写工具类
static class FastIO {
    private final InputStream is;
    private final OutputStream os;
    private final byte[] inbuf = new byte[1 << 16];
    private int lenbuf = 0;
    private int ptrbuf = 0;
    private final StringBuilder outBuf = new StringBuilder();

    public FastIO(InputStream is, OutputStream os) {
        this.is = is;
        this.os = os;
    }

    private int readByte() {
        if (ptrbuf >= lenbuf) {
            ptrbuf = 0;
            try {
                lenbuf = is.read(inbuf);
```

```
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
        if (lenbuf == -1) {
            return -1;
        }
    }
    return inbuf[ptrbuf++] & 0xff;
}

private int skip() {
    int b;
    while ((b = readByte()) != -1) {
        if (b > ' ') {
            return b;
        }
    }
    return -1;
}

public int nextInt() {
    int b = skip();
    if (b == -1) {
        throw new RuntimeException("No more integers (EOF)");
    }
    boolean negative = false;
    if (b == '-') {
        negative = true;
        b = readByte();
    }
    int val = 0;
    while (b >= '0' && b <= '9') {
        val = val * 10 + (b - '0');
        b = readByte();
    }
    return negative ? -val : val;
}

public void write(String s) {
    outBuf.append(s);
}

public void writeInt(int x) {
```

```
            outBuf.append(x);
        }

        public void writelnInt(int x) {
            outBuf.append(x).append('\n');
        }

        public void flush() {
            try {
                os.write(outBuf.toString().getBytes());
                os.flush();
                outBuf.setLength(0);
            } catch (IOException e) {
                throw new RuntimeException(e);
            }
        }
    }

}
```

==========

文件: Code03_MinimumMexTree2.java

==========

```
package class166;

// 最小 mex 生成树，C++版
// 给定 n 个点，m 条边的无向连通图，边有边权
// 自然数集合 S 的 mex 含义为：最小的、没有出现在 S 中的自然数
// 现在你要求出一个这个图的生成树，使得其边权集合的 mex 尽可能小
// 对本题来说，注意 0 是自然数
// 1 <= n <= 10^6
// 1 <= m <= 2 * 10^6
// 0 <= 边权 <= 10^5
// 测试链接 : https://www.luogu.com.cn/problem/P5631
// 如下实现是 C++的版本，C++版本和 java 版本逻辑完全一样
// 提交如下代码，可以通过所有测试用例

//#include <bits/stdc++.h>
//
//using namespace std;
//
//const int MAXN = 1000001;
```

```
//const int MAXV = 100001;
//const int MAXT = 30000001;
//
//int n, m, v;
//
//int father[MAXN];
//int siz[MAXN];
//int rollback[MAXN][2];
//int opsize = 0;
//
//int head[MAXV << 2];
//int nxt[MAXT];
//int tox[MAXT];
//int toy[MAXT];
//int cnt = 0;
//
//int part;
//
//void addEdge(int i, int x, int y) {
//    nxt[++cnt] = head[i];
//    tox[cnt] = x;
//    toy[cnt] = y;
//    head[i] = cnt;
//}
//
//int find(int i) {
//    while (i != father[i]) {
//        i = father[i];
//    }
//    return i;
//}
//
//void Union(int x, int y) {
//    int fx = find(x);
//    int fy = find(y);
//    if (siz[fx] < siz[fy]) {
//        int tmp = fx;
//        fx = fy;
//        fy = tmp;
//    }
//    father[fy] = fx;
//    siz[fx] += siz[fy];
//    rollback[++opsize][0] = fx;
```

```
//    rollback[opsize][1] = fy;
//}
//
//void undo() {
//    int fx = rollback[opsize][0];
//    int fy = rollback[opsize--][1];
//    father[fy] = fy;
//    siz[fx] -= siz[fy];
//}
//
//void add(int jobl, int jobr, int jobx, int joby, int l, int r, int i) {
//    if (jobl <= l && r <= jobr) {
//        addEdge(i, jobx, joby);
//    } else {
//        int mid = (l + r) >> 1;
//        if (jobl <= mid) {
//            add(jobl, jobr, jobx, joby, l, mid, i << 1);
//        }
//        if (jobr > mid) {
//            add(jobl, jobr, jobx, joby, mid + 1, r, i << 1 | 1);
//        }
//    }
//}
//
//int dfs(int l, int r, int i) {
//    int unionCnt = 0;
//    for (int ei = head[i]; ei > 0; ei = nxt[ei]) {
//        int fx = find(tox[ei]);
//        int fy = find(toy[ei]);
//        if (fx != fy) {
//            Union(fx, fy);
//            part--;
//            unionCnt++;
//        }
//    }
//    int ans = -1;
//    if (l == r) {
//        if (part == 1) {
//            ans = l;
//        }
//    } else {
//        int mid = (l + r) >> 1;
//        ans = dfs(l, mid, i << 1);
```

```
//        if (ans == -1) {
//            ans = dfs(mid + 1, r, i << 1 | 1);
//        }
//    }
//    for (int k = 1; k <= unionCnt; k++) {
//        undo();
//        part++;
//    }
//    return ans;
//}
//
//int main() {
//    ios::sync_with_stdio(false);
//    cin.tie(nullptr);
//    cin >> n >> m;
//    v = MAXV;
//    for (int i = 1; i <= n; i++) {
//        father[i] = i;
//        siz[i] = 1;
//    }
//    for (int i = 1; i <= m; i++) {
//        int x, y, w;
//        cin >> x >> y >> w;
//        if (w > 0) {
//            add(0, w - 1, x, y, 0, v, 1);
//        }
//        add(w + 1, v, x, y, 0, v, 1);
//    }
//    part = n;
//    cout << dfs(0, v, 1) << '\n';
//    return 0;
//}
```

==========

文件: Code04_UniqueOccurrences1.java

==========

```
package class166;

/**
 * Codeforces 1681F Unique Occurrences - Java 实现
 *
 * 题目来源: Codeforces
```

```
 * 题目链接: https://codeforces.com/problemset/problem/1681/F
 * 洛谷链接: https://www.luogu.com.cn/problem/CF1681F
 * 题目描述:
 *    给定一棵 n 个节点的树，每条边有一个颜色值
 *    定义 f(u, v)为点 u 到点 v 的简单路径上恰好出现一次的颜色的数量
 *    求∑(u = 1..n) ∑(v = u + 1..n) f(u, v) 的结果
 *
 * 解题思路:
 *    使用线段树分治 + 可撤销并查集
 *    1. 对于每种颜色，找出所有该颜色的边
 *    2. 对于每种颜色 c，将其作为"不存在"的颜色处理
 *    3. 将颜色不存在的时间区间映射到线段树上
 *    4. DFS 遍历时，计算各个连通块之间的贡献
 *
 * 时间复杂度: O((n + m) log n)
 * 空间复杂度: O(n + m)
 *
 * 是否为最优解: 是
 *    这是处理树上路径颜色计数问题的高效解法
 *
 * 工程化考量:
 *    1. 使用 FastIO 提高输入输出效率
 *    2. 按秩合并优化并查集性能
 *    3. 精确回滚保证状态一致性
 *
 * 适用场景:
 *    1. 树上路径颜色计数问题
 *    2. 离线处理树论问题
 *    3. 需要统计恰好出现一次元素的场景
 *
 * 注意事项:
 *    1. 可撤销并查集不能使用路径压缩，只能按秩合并
 *    2. 线段树分治是离线算法
 *    3. 需要正确处理颜色不存在的时间区间
 *
 * 1 <= 颜色值 <= n <= 2 * 10^5
 * 提交时请把类名改成"Main"，可以通过所有测试用例
 */

import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
```

```
public class Code04_UniqueOccurrences1 {

    public static int MAXN = 500001;
    public static int MAXT = 10000001;
    public static int n, v;

    public static int[] father = new int[MAXN];
    public static int[] siz = new int[MAXN];
    public static int[][] rollback = new int[MAXN][2];
    public static int opsize = 0;

    // 每种颜色拥有哪些边的列表
    public static int[] headc = new int[MAXN];
    public static int[] nextc = new int[MAXN];
    public static int[] xc = new int[MAXN];
    public static int[] yc = new int[MAXN];
    public static int cntc = 0;

    // 颜色轴线段树的区间任务列表
    public static int[] headt = new int[MAXN << 2];
    public static int[] nextt = new int[MAXT];
    public static int[] xt = new int[MAXT];
    public static int[] yt = new int[MAXT];
    public static int cntt = 0;

    public static long ans = 0;

    public static void addEdgeC(int i, int x, int y) {
        nextc[++cntc] = headc[i];
        xc[cntc] = x;
        yc[cntc] = y;
        headc[i] = cntc;
    }

    public static void addEdgeS(int i, int x, int y) {
        nextt[++cntt] = headt[i];
        xt[cntt] = x;
        yt[cntt] = y;
        headt[i] = cntt;
    }

    public static int find(int i) {
        while (i != father[i]) {
```

```
        i = father[i];
    }
    return i;
}

public static void union(int x, int y) {
    int fx = find(x);
    int fy = find(y);
    if (siz[fx] < siz[fy]) {
        int tmp = fx;
        fx = fy;
        fy = tmp;
    }
    father[fy] = fx;
    siz[fx] += siz[fy];
    rollback[++opsize][0] = fx;
    rollback[opsize][1] = fy;
}

public static void undo() {
    int fx = rollback[opsize][0];
    int fy = rollback[opsize--][1];
    father[fy] = fy;
    siz[fx] -= siz[fy];
}

public static void add(int jobl, int jobr, int jobx, int joby, int l, int r, int i) {
    if (jobl <= l && r <= jobr) {
        addEdgeS(i, jobx, joby);
    } else {
        int mid = (l + r) >> 1;
        if (jobl <= mid) {
            add(jobl, jobr, jobx, joby, l, mid, i << 1);
        }
        if (jobr > mid) {
            add(jobl, jobr, jobx, joby, mid + 1, r, i << 1 | 1);
        }
    }
}

public static void dfs(int l, int r, int i) {
    int unionCnt = 0;
    for (int ei = headt[i]; ei > 0; ei = nextt[ei]) {
```

```
            union(xt[ei], yt[ei]);
            unionCnt++;
        }
        if (l == r) {
            for (int ei = headc[l], fx, fy; ei > 0; ei = nextc[ei]) {
                fx = find(xc[ei]);
                fy = find(yc[ei]);
                ans += (long) siz[fx] * siz[fy];
            }
        } else {
            int mid = (l + r) >> 1;
            dfs(l, mid, i << 1);
            dfs(mid + 1, r, i << 1 | 1);
        }
        for (int k = 1; k <= unionCnt; k++) {
            undo();
        }
    }

    public static void main(String[] args) {
        FastIO io = new FastIO(System.in, System.out);
        n = io.nextInt();
        v = n;
        for (int i = 1, x, y, c; i < n; i++) {
            x = io.nextInt();
            y = io.nextInt();
            c = io.nextInt();
            addEdgeC(c, x, y);
            if (c > 1) {
                add(1, c - 1, x, y, 1, v, 1);
            }
            if (c < v) {
                add(c + 1, v, x, y, 1, v, 1);
            }
        }
        for (int i = 1; i <= n; i++) {
            father[i] = i;
            siz[i] = 1;
        }
        dfs(1, v, 1);
        io.writelnLong(ans);
        io.flush();
    }
```

```
// 读写工具类
static class FastIO {
    private final InputStream is;
    private final OutputStream os;
    private final byte[] inbuf = new byte[1 << 16];
    private int lenbuf = 0;
    private int ptrbuf = 0;
    private final StringBuilder outBuf = new StringBuilder();

    public FastIO(InputStream is, OutputStream os) {
        this.is = is;
        this.os = os;
    }

    private int readByte() {
        if (ptrbuf >= lenbuf) {
            ptrbuf = 0;
            try {
                lenbuf = is.read(inbuf);
            } catch (IOException e) {
                throw new RuntimeException(e);
            }
            if (lenbuf == -1) {
                return -1;
            }
        }
        return inbuf[ptrbuf++] & 0xff;
    }

    private int skip() {
        int b;
        while ((b = readByte()) != -1) {
            if (b > ' ') {
                return b;
            }
        }
        return -1;
    }

    public int nextInt() {
        int b = skip();
        if (b == -1) {
```

```
                throw new RuntimeException("No more integers (EOF)");
            }
            boolean negative = false;
            if (b == '-') {
                negative = true;
                b = readByte();
            }
            int val = 0;
            while (b >= '0' && b <= '9') {
                val = val * 10 + (b - '0');
                b = readByte();
            }
            return negative ? -val : val;
        }

        public void writelnLong(long x) {
            outBuf.append(x).append('\n');
        }

        public void flush() {
            try {
                os.write(outBuf.toString().getBytes());
                os.flush();
                outBuf.setLength(0);
            } catch (IOException e) {
                throw new RuntimeException(e);
            }
        }
    }

}
```

==========

文件: Code04_UniqueOccurrences2.java

==========

```
package class166;

// 独特事件，C++版
// 一共有 n 个节点，n-1 条无向边，边给定颜色值，所有节点连成一棵树
// 定义 f(u, v) : 点 u 到点 v 的简单路径上恰好出现一次的颜色的数量
// 打印 Σ(u = 1..n) Σ(v = u + 1..n) f(u, v) 的结果
// 1 <= 颜色值 <= n <= 2 * 10^5
```

```
// 测试链接 : https://www.luogu.com.cn/problem/CF1681F
// 测试链接 : https://codeforces.com/problemset/problem/1681/F
// 如下实现是C++的版本，C++版本和java版本逻辑完全一样
// 提交如下代码，可以通过所有测试用例

//#include <bits/stdc++.h>
//
//using namespace std;
//
//const int MAXN = 500001;
//const int MAXT = 10000001;
//int n, v;
//
//int father[MAXN];
//int siz[MAXN];
//int rollback[MAXN][2];
//int opsize = 0;
//
//int headc[MAXN];
//int nxtc[MAXN];
//int xc[MAXN];
//int yc[MAXN];
//int cntc = 0;
//
//int headt[MAXN << 2];
//int nxtt[MAXT];
//int xt[MAXT];
//int yt[MAXT];
//int cntt = 0;
//
//long long ans = 0;
//
//void addEdgeC(int i, int x, int y) {
//    nxtc[++cntc] = headc[i];
//    xc[cntc] = x;
//    yc[cntc] = y;
//    headc[i] = cntc;
//}
//
//void addEdgeS(int i, int x, int y) {
//    nxtt[++cntt] = headt[i];
//    xt[cntt] = x;
//    yt[cntt] = y;
```

```
//    headt[i] = cntt;
//}
//
//int find(int i) {
//    while (i != father[i]) {
//        i = father[i];
//    }
//    return i;
//}
//
//void Union(int x, int y) {
//    int fx = find(x);
//    int fy = find(y);
//    if (siz[fx] < siz[fy]) {
//        int tmp = fx;
//        fx = fy;
//        fy = tmp;
//    }
//    father[fy] = fx;
//    siz[fx] += siz[fy];
//    rollback[++opsize][0] = fx;
//    rollback[opsize][1] = fy;
//}
//
//void undo() {
//    int fx = rollback[opsize][0];
//    int fy = rollback[opsize--][1];
//    father[fy] = fy;
//    siz[fx] -= siz[fy];
//}
//
//void add(int jobl, int jobr, int jobx, int joby, int l, int r, int i) {
//    if (jobl <= l && r <= jobr) {
//        addEdgeS(i, jobx, joby);
//    } else {
//        int mid = (l + r) >> 1;
//        if (jobl <= mid) {
//            add(jobl, jobr, jobx, joby, l, mid, i << 1);
//        }
//        if (jobr > mid) {
//            add(jobl, jobr, jobx, joby, mid + 1, r, i << 1 | 1);
//        }
//    }
```

```
//}
//
//void dfs(int l, int r, int i) {
//    int unionCnt = 0;
//    for (int ei = headt[i]; ei > 0; ei = nxtt[ei]) {
//        Union(xt[ei], yt[ei]);
//        unionCnt++;
//    }
//    if (l == r) {
//        for (int ei = headc[l], fx, fy; ei > 0; ei = nxtc[ei]) {
//            fx = find(xc[ei]);
//            fy = find(yc[ei]);
//            ans += 1LL * siz[fx] * siz[fy];
//        }
//    } else {
//        int mid = (l + r) >> 1;
//        dfs(l, mid, i << 1);
//        dfs(mid + 1, r, i << 1 | 1);
//    }
//    for (int k = 1; k <= unionCnt; k++) {
//        undo();
//    }
//}
//
//int main() {
//    ios::sync_with_stdio(false);
//    cin.tie(nullptr);
//    cin >> n;
//    v = n;
//    for (int i = 1, x, y, c; i < n; i++) {
//        cin >> x >> y >> c;
//        addEdgeC(c, x, y);
//        if (c > 1) {
//            add(1, c - 1, x, y, 1, v, 1);
//        }
//        if (c < v) {
//            add(c + 1, v, x, y, 1, v, 1);
//        }
//    }
//    for (int i = 1; i <= n; i++) {
//        father[i] = i;
//        siz[i] = 1;
//    }
```

```
//    dfs(1, v, 1);
//    cout << ans << '\n';
//    return 0;
//}
```

==========

文件: Code05_GreatIntegration1.java

==========

```
package class166;

/**
 * 洛谷 P4219 大融合 - Java 实现
 *
 * 题目来源: 洛谷
 * 题目链接: https://www.luogu.com.cn/problem/P4219
 * 题目描述:
 *    小强要在 n 个孤立的星球上建立起一套通信系统，这个系统就是连接 n 个点的一棵树
 *    这个树的边是一条一条添加上去的
 *    在某个时刻，一条边的负载就是它所在的当前联通块上经过它的简单路径的数量
 *    支持两种操作：
 *    1. 操作 A x y : 点 x 和点 y 之间连一条边，保证之前 x 和 y 是不联通的
 *    2. 操作 Q x y : 打印点 x 和点 y 之间这条边的负载，保证 x 和 y 之间有一条边
 *    边负载定义为，这条边两侧端点各自连通区大小的乘积
 *
 * 解题思路:
 *    使用线段树分治 + 可撤销并查集
 *    1. 对于加边操作，记录其存在的时间区间
 *    2. 将时间区间映射到线段树上
 *    3. DFS 遍历时，对于叶子节点上的查询操作，计算答案
 *    4. 边的负载定义为删去该边后两个连通块大小的乘积
 *
 * 时间复杂度: O((n + q) log q)
 * 空间复杂度: O(n + q)
 *
 * 是否为最优解: 是
 *    这是处理动态树边负载计算问题的高效解法
 *
 * 工程化考量:
 *    1. 使用 FastIO 提高输入输出效率
 *    2. 按秩合并优化并查集性能
 *    3. 精确回滚保证状态一致性
 *
```

```
 * 适用场景:
 *    1. 动态树边负载计算问题
 *    2. 离线处理树论问题
 *    3. 需要计算边重要性的场景
 *
 * 注意事项:
 *    1. 可撤销并查集不能使用路径压缩，只能按秩合并
 *    2. 线段树分治是离线算法
 *    3. 需要正确处理边的存在时间区间
 *
 * 1 <= n、q <= 10^5
 * 提交时请把类名改成"Main"，可以通过所有测试用例
 */

import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.util.Arrays;

public class Code05_GreatIntegration1 {

    public static int MAXN = 100001;
    public static int MAXT = 3000001;
    public static int n, q;

    public static int[] op = new int[MAXN];
    public static int[] u = new int[MAXN];
    public static int[] v = new int[MAXN];

    // 端点 x、端点 y、操作序号 t
    public static int[][] event = new int[MAXN][3];

    public static int[] father = new int[MAXN];
    public static int[] siz = new int[MAXN];
    public static int[][] rollback = new int[MAXN][2];
    public static int opsize = 0;

    public static int[] head = new int[MAXN << 2];
    public static int[] next = new int[MAXT];
    public static int[] tox = new int[MAXT];
    public static int[] toy = new int[MAXT];
    public static int cnt = 0;
```

```
public static long[] ans = new long[MAXN];

public static void addEdge(int i, int x, int y) {
    next[++cnt] = head[i];
    tox[cnt] = x;
    toy[cnt] = y;
    head[i] = cnt;
}

public static int find(int i) {
    while (i != father[i]) {
        i = father[i];
    }
    return i;
}

public static void union(int x, int y) {
    int fx = find(x);
    int fy = find(y);
    if (siz[fx] < siz[fy]) {
        int tmp = fx;
        fx = fy;
        fy = tmp;
    }
    father[fy] = fx;
    siz[fx] += siz[fy];
    rollback[++opsize][0] = fx;
    rollback[opsize][1] = fy;
}

public static void undo() {
    int fx = rollback[opsize][0];
    int fy = rollback[opsize--][1];
    father[fy] = fy;
    siz[fx] -= siz[fy];
}

public static void add(int jobl, int jobr, int jobx, int joby, int l, int r, int i) {
    if (jobl <= l && r <= jobr) {
        addEdge(i, jobx, joby);
    } else {
        int mid = (l + r) >> 1;
```

```
            if (jobl <= mid) {
                add(jobl, jobr, jobx, joby, l, mid, i << 1);
            }
            if (jobr > mid) {
                add(jobl, jobr, jobx, joby, mid + 1, r, i << 1 | 1);
            }
        }
    }

    public static void dfs(int l, int r, int i) {
        int unionCnt = 0;
        for (int ei = head[i]; ei > 0; ei = next[ei]) {
            union(tox[ei], toy[ei]);
            unionCnt++;
        }
        if (l == r) {
            if (op[l] == 2) {
                ans[l] = (long) siz[find(u[l])] * siz[find(v[l])];
            }
        } else {
            int mid = (l + r) >> 1;
            dfs(l, mid, i << 1);
            dfs(mid + 1, r, i << 1 | 1);
        }
        for (int k = 1; k <= unionCnt; k++) {
            undo();
        }
    }

    public static void prepare() {
        for (int i = 1; i <= n; i++) {
            father[i] = i;
            siz[i] = 1;
        }
        for (int i = 1; i <= q; i++) {
            event[i][0] = u[i];
            event[i][1] = v[i];
            event[i][2] = i;
        }
        Arrays.sort(event, 1, q + 1, (a, b) -> a[0] != b[0] ? a[0] - b[0] : a[1] != b[1] ? a[1] -
b[1] : a[2] - b[2]);
        for (int l = 1, r = 1; l <= q; l = ++r) {
            int x = event[l][0], y = event[l][1], t = event[l][2];
```

```
            while (r + 1 <= q && event[r + 1][0] == x && event[r + 1][1] == y) {
                r++;
            }
            for (int i = l + 1; i <= r; i++) {
                add(t, event[i][2] - 1, x, y, 1, q, 1);
                t = event[i][2] + 1;
            }
            if (t <= q) {
                add(t, q, x, y, 1, q, 1);
            }
        }
    }

    public static void main(String[] args) throws IOException {
        FastReader in = new FastReader();
        PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));
        n = in.nextInt();
        q = in.nextInt();
        char t;
        int x, y;
        for (int i = 1; i <= q; i++) {
            t = in.nextChar();
            x = in.nextInt();
            y = in.nextInt();
            op[i] = t == 'A' ? 1 : 2;
            u[i] = Math.min(x, y);
            v[i] = Math.max(x, y);
        }
        prepare();
        dfs(1, q, 1);
        for (int i = 1; i <= q; i++) {
            if (op[i] == 2) {
                out.println(ans[i]);
            }
        }
        out.flush();
        out.close();
    }

    // 读写工具类
    static class FastReader {
        final private int BUFFER_SIZE = 1 << 16;
        private final InputStream in;
```

```
private final byte[] buffer;
private int ptr, len;

public FastReader() {
    in = System.in;
    buffer = new byte[BUFFER_SIZE];
    ptr = len = 0;
}

private boolean hasNextByte() throws IOException {
    if (ptr < len)
        return true;
    ptr = 0;
    len = in.read(buffer);
    return len > 0;
}

private byte readByte() throws IOException {
    if (!hasNextByte())
        return -1;
    return buffer[ptr++];
}

public char nextChar() throws IOException {
    byte c;
    do {
        c = readByte();
        if (c == -1)
            return 0;
    } while (c <= ' ');
    char ans = 0;
    while (c > ' ') {
        ans = (char) c;
        c = readByte();
    }
    return ans;
}

public int nextInt() throws IOException {
    int num = 0;
    byte b = readByte();
    while (isWhitespace(b))
        b = readByte();
```

```
            boolean minus = false;
            if (b == '-') {
                minus = true;
                b = readByte();
            }
            while (!isWhitespace(b) && b != -1) {
                num = num * 10 + (b - '0');
                b = readByte();
            }
            return minus ? -num : num;
        }

        private boolean isWhitespace(byte b) {
            return b == ' ' || b == '\n' || b == '\r' || b == '\t';
        }
    }

}
```

==========

文件: Code05_GreatIntegration2.java

==========

```
package class166;

// 大融合，C++版
// 一共有 n 个点，一共有 q 条操作，每条操作是如下两种类型中的一种
// 操作 A x y : 点 x 和点 y 之间连一条边，保证之前 x 和 y 是不联通的
// 操作 Q x y : 打印点 x 和点 y 之间这条边的负载，保证 x 和 y 之间有一条边
// 边负载定义为，这条边两侧端点各自连通区大小的乘积
// 1 <= n、q <= 10^5
// 测试链接 : https://www.luogu.com.cn/problem/P4219
// 如下实现是 C++的版本，C++版本和 java 版本逻辑完全一样
// 提交如下代码，可以通过所有测试用例

//#include <bits/stdc++.h>
//
//using namespace std;
//
//struct Event {
//    int x, y, t;
//};
//
```

```
//bool EventCmp(Event a, Event b) {
//    if (a.x != b.x) {
//        return a.x < b.x;
//    } else if (a.y != b.y) {
//        return a.y < b.y;
//    } else {
//        return a.t < b.t;
//    }
//}
//
//const int MAXN = 100001;
//const int MAXT = 3000001;
//int n, q;
//
//int op[MAXN];
//int u[MAXN];
//int v[MAXN];
//
//Event event[MAXN];
//
//int father[MAXN];
//int siz[MAXN];
//int rollback[MAXN][2];
//int opsize = 0;
//
//int head[MAXN << 2];
//int nxt[MAXT];
//int tox[MAXT];
//int toy[MAXT];
//int cnt = 0;
//
//long long ans[MAXN];
//
//void addEdge(int i, int x, int y) {
//    nxt[++cnt] = head[i];
//    tox[cnt] = x;
//    toy[cnt] = y;
//    head[i] = cnt;
//}
//
//int find(int i) {
//    while (i != father[i]) {
//        i = father[i];
```

```
//    }
//    return i;
//}
//
//void Union(int x, int y) {
//    int fx = find(x);
//    int fy = find(y);
//    if (siz[fx] < siz[fy]) {
//        int tmp = fx;
//        fx = fy;
//        fy = tmp;
//    }
//    father[fy] = fx;
//    siz[fx] += siz[fy];
//    rollback[++opsize][0] = fx;
//    rollback[opsize][1] = fy;
//}
//
//void undo() {
//    int fx = rollback[opsize][0];
//    int fy = rollback[opsize--][1];
//    father[fy] = fy;
//    siz[fx] -= siz[fy];
//}
//
//void add(int jobl, int jobr, int jobx, int joby, int l, int r, int i) {
//    if (jobl <= l && r <= jobr) {
//        addEdge(i, jobx, joby);
//    } else {
//        int mid = (l + r) >> 1;
//        if (jobl <= mid) {
//            add(jobl, jobr, jobx, joby, l, mid, i << 1);
//        }
//        if (jobr > mid) {
//            add(jobl, jobr, jobx, joby, mid + 1, r, i << 1 | 1);
//        }
//    }
//}
//
//void dfs(int l, int r, int i) {
//    int unionCnt = 0;
//    for (int ei = head[i]; ei > 0; ei = nxt[ei]) {
//        Union(tox[ei], toy[ei]);
```

```
//        unionCnt++;
//    }
//    if (l == r) {
//        if (op[l] == 2) {
//            ans[l] = 1LL * siz[find(u[l])] * siz[find(v[l])];
//        }
//    } else {
//        int mid = (l + r) >> 1;
//        dfs(l, mid, i << 1);
//        dfs(mid + 1, r, i << 1 | 1);
//    }
//    for (int k = 1; k <= unionCnt; k++) {
//        undo();
//    }
//}
//
//void prepare() {
//    for (int i = 1; i <= n; i++) {
//        father[i] = i;
//        siz[i] = 1;
//    }
//    for (int i = 1; i <= q; i++) {
//        event[i].x = u[i];
//        event[i].y = v[i];
//        event[i].t = i;
//    }
//    sort(event + 1, event + q + 1, EventCmp);
//    for (int l = 1, r = 1; l <= q; l = ++r) {
//        int x = event[l].x, y = event[l].y, t = event[l].t;
//        while (r + 1 <= q && event[r + 1].x == x && event[r + 1].y == y) {
//            r++;
//        }
//        for (int j = l + 1; j <= r; j++) {
//            add(t, event[j].t - 1, x, y, 1, q, 1);
//            t = event[j].t + 1;
//        }
//        if (t <= q) {
//            add(t, q, x, y, 1, q, 1);
//        }
//    }
//}
//
//int main() {
```

```
//    ios::sync_with_stdio(false);
//    cin.tie(nullptr);
//    cin >> n >> q;
//    char t;
//    int x, y;
//    for (int i = 1; i <= q; i++) {
//        cin >> t >> x >> y;
//        op[i] = (t == 'A') ? 1 : 2;
//        u[i] = min(x, y);
//        v[i] = max(x, y);
//    }
//    prepare();
//    dfs(1, q, 1);
//    for (int i = 1; i <= q; i++) {
//        if (op[i] == 2) {
//            cout << ans[i] << '\n';
//        }
//    }
//    return 0;
//}
```

==========

文件: Code06_ConnectedGraph1.java

==========

```
package class166;

/**
 * 洛谷 P5227 连通图 - Java 实现
 *
 * 题目来源: 洛谷
 * 题目链接: https://www.luogu.com.cn/problem/P5227
 * 题目描述:
 *   给定一个无向连通图和若干个小集合，每个小集合包含一些边
 *   对于每个集合，需要确定将集合中的边删掉后图是否保持联通
 *   集合间的询问相互独立
 *
 * 解题思路:
 *   使用线段树分治 + 可撤销并查集
 *   1. 转换思路：找出每条边不存在的时间区间，在这些区间内不使用该边
 *   2. 将每条边不存在的时间区间映射到线段树上
 *   3. DFS 遍历时维护连通性，如果发现整个图连通则标记答案
 *
```

 * 时间复杂度: O((n + m) log k)
 * 空间复杂度: O(n + m)
 *
 * 是否为最优解: 是
 *   这是处理动态图连通性验证问题的高效解法
 *
 * 工程化考量:
 *   1. 使用 FastIO 提高输入输出效率
 *   2. 按秩合并优化并查集性能
 *   3. 精确回滚保证状态一致性
 *
 * 适用场景:
 *   1. 动态图连通性验证问题
 *   2. 离线处理图论问题
 *   3. 需要验证删除边后图连通性的场景
 *
 * 注意事项:
 *   1. 可撤销并查集不能使用路径压缩，只能按秩合并
 *   2. 线段树分治是离线算法
 *   3. 需要正确处理边不存在的时间区间
 *
 * 1 <= n、k <= 10^5
 * 1 <= m <= 2 * 10^5
 * 1 <= c <= 4
 * 提交时请把类名改成"Main"，可以通过所有测试用例
 */

```
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.util.Arrays;

public class Code06_ConnectedGraph1 {

    public static int MAXN = 100001;
    public static int MAXM = 200001;
    public static int MAXE = 400001;
    public static int MAXT = 10000001;
    public static int n, m, k;

    public static int[] u = new int[MAXM];
    public static int[] v = new int[MAXM];
```

```
public static int[][] event = new int[MAXE][2];
public static int ecnt = 0;
public static boolean[] visit = new boolean[MAXM];

public static int[] father = new int[MAXN];
public static int[] siz = new int[MAXN];
public static int[][] rollback = new int[MAXN][2];
public static int opsize = 0;

public static int[] head = new int[MAXN << 2];
public static int[] next = new int[MAXT];
public static int[] tox = new int[MAXT];
public static int[] toy = new int[MAXT];
public static int cnt = 0;

public static boolean[] ans = new boolean[MAXN];

public static void addEdge(int i, int x, int y) {
    next[++cnt] = head[i];
    tox[cnt] = x;
    toy[cnt] = y;
    head[i] = cnt;
}

public static int find(int i) {
    while (i != father[i]) {
        i = father[i];
    }
    return i;
}

public static void union(int x, int y) {
    int fx = find(x);
    int fy = find(y);
    if (siz[fx] < siz[fy]) {
        int tmp = fx;
        fx = fy;
        fy = tmp;
    }
    father[fy] = fx;
    siz[fx] += siz[fy];
    rollback[++opsize][0] = fx;
    rollback[opsize][1] = fy;
```

```
}

public static void undo() {
    int fx = rollback[opsize][0];
    int fy = rollback[opsize--][1];
    father[fy] = fy;
    siz[fx] -= siz[fy];
}

public static void add(int jobl, int jobr, int jobx, int joby, int l, int r, int i) {
    if (jobl <= l && r <= jobr) {
        addEdge(i, jobx, joby);
    } else {
        int mid = (l + r) >> 1;
        if (jobl <= mid) {
            add(jobl, jobr, jobx, joby, l, mid, i << 1);
        }
        if (jobr > mid) {
            add(jobl, jobr, jobx, joby, mid + 1, r, i << 1 | 1);
        }
    }
}

public static void dfs(int l, int r, int i) {
    boolean check = false;
    int unionCnt = 0;
    for (int ei = head[i]; ei > 0; ei = next[ei]) {
        int x = tox[ei], y = toy[ei], fx = find(x), fy = find(y);
        if (fx != fy) {
            union(fx, fy);
            unionCnt++;
        }
        if (siz[find(fx)] == n) {
            check = true;
            break;
        }
    }
    if (check) {
        for (int j = l; j <= r; j++) {
            ans[j] = true;
        }
    } else {
        if (l == r) {
```

```
            ans[l] = false;
        } else {
            int mid = (l + r) >> 1;
            dfs(l, mid, i << 1);
            dfs(mid + 1, r, i << 1 | 1);
        }
    }
    for (int j = 1; j <= unionCnt; j++) {
        undo();
    }
}

public static void prepare() {
    for (int i = 1; i <= n; i++) {
        father[i] = i;
        siz[i] = 1;
    }
    Arrays.sort(event, 1, ecnt + 1, (a, b) -> a[0] != b[0] ? a[0] - b[0] : a[1] - b[1]);
    int eid, t;
    for (int l = 1, r = 1; l <= ecnt; l = ++r) {
        eid = event[l][0];
        visit[eid] = true;
        while (r + 1 <= ecnt && event[r + 1][0] == eid) {
            r++;
        }
        t = 1;
        for (int i = l; i <= r; i++) {
            if (t <= event[i][1] - 1) {
                add(t, event[i][1] - 1, u[eid], v[eid], 1, k, 1);
            }
            t = event[i][1] + 1;
        }
        if (t <= k) {
            add(t, k, u[eid], v[eid], 1, k, 1);
        }
    }
    for (int i = 1; i <= m; i++) {
        if (!visit[i]) {
            add(1, k, u[i], v[i], 1, k, 1);
        }
    }
}
```

```
public static void main(String[] args) {
    FastIO io = new FastIO(System.in, System.out);
    n = io.nextInt();
    m = io.nextInt();
    for (int i = 1; i <= m; i++) {
        u[i] = io.nextInt();
        v[i] = io.nextInt();
    }
    k = io.nextInt();
    for (int i = 1, c; i <= k; i++) {
        c = io.nextInt();
        for (int j = 1; j <= c; j++) {
            event[++ecnt][0] = io.nextInt();
            event[ecnt][1] = i;
        }
    }
    prepare();
    dfs(1, k, 1);
    for (int i = 1; i <= k; i++) {
        if (ans[i]) {
            io.write("Connected\n");
        } else {
            io.write("Disconnected\n");
        }
    }
    io.flush();
}

// 读写工具类
static class FastIO {
    private final InputStream is;
    private final OutputStream os;
    private final byte[] inbuf = new byte[1 << 16];
    private int lenbuf = 0;
    private int ptrbuf = 0;
    private final StringBuilder outBuf = new StringBuilder();

    public FastIO(InputStream is, OutputStream os) {
        this.is = is;
        this.os = os;
    }

    private int readByte() {
```

```
        if (ptrbuf >= lenbuf) {
            ptrbuf = 0;
            try {
                lenbuf = is.read(inbuf);
            } catch (IOException e) {
                throw new RuntimeException(e);
            }
            if (lenbuf == -1) {
                return -1;
            }
        }
        return inbuf[ptrbuf++] & 0xff;
    }

    private int skip() {
        int b;
        while ((b = readByte()) != -1) {
            if (b > ' ') {
                return b;
            }
        }
        return -1;
    }

    public int nextInt() {
        int b = skip();
        if (b == -1) {
            throw new RuntimeException("No more integers (EOF)");
        }
        boolean negative = false;
        if (b == '-') {
            negative = true;
            b = readByte();
        }
        int val = 0;
        while (b >= '0' && b <= '9') {
            val = val * 10 + (b - '0');
            b = readByte();
        }
        return negative ? -val : val;
    }

    public void write(String s) {
```

```
            outBuf.append(s);
        }

        public void writeInt(int x) {
            outBuf.append(x);
        }

        public void writelnInt(int x) {
            outBuf.append(x).append('\n');
        }

        public void flush() {
            try {
                os.write(outBuf.toString().getBytes());
                os.flush();
                outBuf.setLength(0);
            } catch (IOException e) {
                throw new RuntimeException(e);
            }
        }
    }

}
```

==========

文件: Code06_ConnectedGraph2.java

==========

```
package class166;

// 连通图，C++版
// 一共有 n 个点，给定 m 条边，所有点一开始就连通在一起了
// 一共有 k 条操作，每条操作格式如下
// 操作 c ... : 操作涉及 c 条边，这些边的编号 ... 一共 c 个
//              假设删掉这些边，打印整张图是否联通
// 每条操作都是独立的，相互之间没有任何关系
// 1 <= n、k <= 10^5
// 1 <= m <= 2 * 10^5
// 1 <= c <= 4
// 测试链接 : https://www.luogu.com.cn/problem/P5227
// 如下实现是 C++的版本，C++版本和 java 版本逻辑完全一样
// 提交如下代码，可以通过所有测试用例
```

```
//#include <bits/stdc++.h>
//
//using namespace std;
//
//struct Event {
//    int ei, t;
//};
//
//bool EventCmp(Event a, Event b) {
//    if (a.ei != b.ei) {
//        return a.ei < b.ei;
//    } else {
//        return a.t < b.t;
//    }
//}
//
//const int MAXN = 100001;
//const int MAXM = 200001;
//const int MAXE = 400001;
//const int MAXT = 10000001;
//
//int n, m, k;
//
//int u[MAXM];
//int v[MAXM];
//
//Event event[MAXE];
//int ecnt = 0;
//bool vis[MAXM];
//
//int father[MAXN];
//int siz[MAXN];
//int rollback[MAXN][2];
//int opsize = 0;
//
//int head[MAXN << 2];
//int nxt[MAXT];
//int tox[MAXT];
//int toy[MAXT];
//int cnt = 0;
//
//bool ans[MAXN];
//
```

```
//void addEdge(int i, int x, int y) {
//    nxt[++cnt] = head[i];
//    tox[cnt] = x;
//    toy[cnt] = y;
//    head[i] = cnt;
//}
//
//int find(int i) {
//    while (i != father[i]) {
//        i = father[i];
//    }
//    return i;
//}
//
//void Union(int x, int y) {
//    int fx = find(x);
//    int fy = find(y);
//    if (siz[fx] < siz[fy]) {
//        int tmp = fx;
//        fx = fy;
//        fy = tmp;
//    }
//    father[fy] = fx;
//    siz[fx] += siz[fy];
//    rollback[++opsize][0] = fx;
//    rollback[opsize][1] = fy;
//}
//
//void undo() {
//    int fx = rollback[opsize][0];
//    int fy = rollback[opsize--][1];
//    father[fy] = fy;
//    siz[fx] -= siz[fy];
//}
//
//void add(int jobl, int jobr, int jobx, int joby, int l, int r, int i) {
//    if (jobl <= l && r <= jobr) {
//        addEdge(i, jobx, joby);
//    } else {
//        int mid = (l + r) >> 1;
//        if (jobl <= mid) {
//            add(jobl, jobr, jobx, joby, l, mid, i << 1);
//        }
```

```
//        if (jobr > mid) {
//            add(jobl, jobr, jobx, joby, mid + 1, r, i << 1 | 1);
//        }
//    }
//}
//
//void dfs(int l, int r, int i) {
//    bool check = false;
//    int unionCnt = 0;
//    for (int ei = head[i]; ei > 0; ei = nxt[ei]) {
//        int x = tox[ei], y = toy[ei], fx = find(x), fy = find(y);
//        if (fx != fy) {
//            Union(fx, fy);
//            unionCnt++;
//        }
//        if (siz[find(fx)] == n) {
//            check = true;
//            break;
//        }
//    }
//    if (check) {
//        for (int j = l; j <= r; j++) {
//            ans[j] = true;
//        }
//    } else {
//        if (l == r) {
//            ans[l] = false;
//        } else {
//            int mid = (l + r) >> 1;
//            dfs(l, mid, i << 1);
//            dfs(mid + 1, r, i << 1 | 1);
//        }
//    }
//    for (int j = 1; j <= unionCnt; j++) {
//        undo();
//    }
//}
//
//void prepare() {
//    for (int i = 1; i <= n; i++) {
//        father[i] = i;
//        siz[i] = 1;
//    }
```

```
//    sort(event + 1, event + ecnt + 1, EventCmp);
//    for (int l = 1, r = 1, eid; l <= ecnt; l = ++r) {
//        eid = event[l].ei;
//        vis[eid] = true;
//        while (r + 1 <= ecnt && event[r + 1].ei == eid) {
//            r++;
//        }
//        int t = 1;
//        for (int i = l; i <= r; i++) {
//            if (t <= event[i].t - 1) {
//                add(t, event[i].t - 1, u[eid], v[eid], 1, k, 1);
//            }
//            t = event[i].t + 1;
//        }
//        if (t <= k) {
//            add(t, k, u[eid], v[eid], 1, k, 1);
//        }
//    }
//    for (int i = 1; i <= m; i++) {
//        if (!vis[i]) {
//            add(1, k, u[i], v[i], 1, k, 1);
//        }
//    }
//}
//
//int main() {
//    ios::sync_with_stdio(false);
//    cin.tie(nullptr);
//    cin >> n >> m;
//    for (int i = 1; i <= m; i++) {
//        cin >> u[i] >> v[i];
//    }
//    cin >> k;
//    for (int i = 1, c; i <= k; i++) {
//        cin >> c;
//        for (int j = 1; j <= c; j++) {
//            cin >> event[++ecnt].ei;
//            event[ecnt].t = i;
//        }
//    }
//    prepare();
//    dfs(1, k, 1);
//    for (int i = 1; i <= k; i++) {
```

```
//        if (ans[i]) {
//            cout << "Connected" << "\n";
//        } else {
//            cout << "Disconnected" << "\n";
//        }
//    }
//    return 0;
//}
```

==========

文件: Code07_PaintingEdges1.java

==========

```
package class166;

/**
 * Codeforces 576E Painting Edges - Java 实现
 *
 * 题目来源: Codeforces
 * 题目链接: https://codeforces.com/problemset/problem/576/E
 * 洛谷链接: https://www.luogu.com.cn/problem/CF576E
 * 题目描述:
 *   给定一张 n 个点 m 条边的无向图，每条边有一个颜色(初始为无色)
 *   一共有 q 条操作，每次将一条边染为 k 种颜色之一
 *   要求染完后对于任意 i=1...k，当只有颜色为 i 的边存在时，图都是一张二分图
 *   如果执行操作后图仍合法则执行并打印"YES"，否则不执行并打印"NO"
 *
 * 解题思路:
 *   使用线段树分治 + 多个扩展域并查集
 *   1. 对每种颜色维护一个扩展域并查集
 *   2. 检查染色后是否满足条件
 *   3. 不满足则撤销操作
 *
 * 时间复杂度: O(k(n + q) log q)
 * 空间复杂度: O(k(n + q))
 *
 * 是否为最优解: 是
 *   这是处理边染色二分图问题的高效解法
 *
 * 工程化考量:
 *   1. 使用多个扩展域并查集分别维护每种颜色的连通性
 *   2. FastIO 提高输入输出效率
 *   3. 精确回滚保证状态一致性
```

```
 *
 * 适用场景:
 *    1. 边染色二分图问题
 *    2. 离线处理图论问题
 *    3. 需要维护多种颜色约束的场景
 *
 * 注意事项:
 *    1. 扩展域并查集的正确实现
 *    2. 线段树分治是离线算法
 *    3. 需要正确处理操作的时间区间
 *
 * 1 <= n、m、q <= 5 * 10^5
 * 1 <= k <= 50
 * 提交时请把类名改成"Main"，可以通过所有测试用例
 */

import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;

public class Code07_PaintingEdges1 {

    public static int MAXN = 500001;
    public static int MAXK = 51;
    public static int MAXT = 10000001;
    public static int n, m, k, q;

    public static int[] u = new int[MAXN];
    public static int[] v = new int[MAXN];

    public static int[] e = new int[MAXN];
    public static int[] c = new int[MAXN];
    public static int[] post = new int[MAXN];

    public static int[][] father = new int[MAXK][MAXN << 1];
    public static int[][] siz = new int[MAXK][MAXN << 1];
    public static int[][] rollback = new int[MAXN << 1][3];
    public static int opsize = 0;

    // 时间轴线段树的区间上的任务列表
    // 尤其注意 qid 的设置，课上进行了重点解释
    public static int[] head = new int[MAXN << 2];
    public static int[] next = new int[MAXT];
```

```
public static int[] qid = new int[MAXT];
public static int cnt = 0;

// lastColor[i] : 第 i 号边上次成功涂上的颜色
public static int[] lastColor = new int[MAXN];

public static boolean[] ans = new boolean[MAXN];

public static void addEdge(int i, int qi) {
    next[++cnt] = head[i];
    qid[cnt] = qi;
    head[i] = cnt;
}

public static int find(int color, int i) {
    while (i != father[color][i]) {
        i = father[color][i];
    }
    return i;
}

public static void union(int color, int x, int y) {
    int fx = find(color, x);
    int fy = find(color, y);
    if (siz[color][fx] < siz[color][fy]) {
        int tmp = fx;
        fx = fy;
        fy = tmp;
    }
    father[color][fy] = fx;
    siz[color][fx] += siz[color][fy];
    rollback[++opsize][0] = color;
    rollback[opsize][1] = fx;
    rollback[opsize][2] = fy;
}

public static void undo() {
    int color = rollback[opsize][0];
    int fx = rollback[opsize][1];
    int fy = rollback[opsize--][2];
    father[color][fy] = fy;
    siz[color][fx] -= siz[color][fy];
}
```

```
public static void add(int jobl, int jobr, int jobq, int l, int r, int i) {
    if (jobl <= l && r <= jobr) {
        addEdge(i, jobq);
    } else {
        int mid = (l + r) >> 1;
        if (jobl <= mid) {
            add(jobl, jobr, jobq, l, mid, i << 1);
        }
        if (jobr > mid) {
            add(jobl, jobr, jobq, mid + 1, r, i << 1 | 1);
        }
    }
}

public static void dfs(int l, int r, int i) {
    int unionCnt = 0;
    int color, x, y, xn, yn, fx, fy, fxn, fyn;
    for (int ei = head[i]; ei > 0; ei = next[ei]) {
        color = c[qid[ei]];
        x = u[e[qid[ei]]];
        y = v[e[qid[ei]]];
        xn = x + n;
        yn = y + n;
        fx = find(color, x);
        fy = find(color, y);
        fxn = find(color, xn);
        fyn = find(color, yn);
        if (fx != fyn) {
            union(color, fx, fyn);
            unionCnt++;
        }
        if (fy != fxn) {
            union(color, fy, fxn);
            unionCnt++;
        }
    }
    if (l == r) {
        if (find(c[l], u[e[l]]) == find(c[l], v[e[l]])) {
            ans[l] = false;
            c[l] = lastColor[e[l]];
        } else {
            ans[l] = true;
```

```
            lastColor[e[l]] = c[l];
        }
    } else {
        int mid = (l + r) >> 1;
        dfs(l, mid, i << 1);
        dfs(mid + 1, r, i << 1 | 1);
    }
    for (int j = 1; j <= unionCnt; j++) {
        undo();
    }
}

public static void prepare() {
    for (int color = 1; color <= k; color++) {
        for (int i = 1; i <= n; i++) {
            father[color][i] = i;
            father[color][i + n] = i + n;
            siz[color][i] = 1;
            siz[color][i + n] = 1;
        }
    }
    for (int i = 1; i <= m; i++) {
        post[i] = q;
    }
    for (int i = q; i >= 1; i--) {
        if (i + 1 <= post[e[i]]) {
            add(i + 1, post[e[i]], i, 1, q, 1);
        }
        post[e[i]] = i;
    }
}

public static void main(String[] args) {
    FastIO io = new FastIO(System.in, System.out);
    n = io.nextInt();
    m = io.nextInt();
    k = io.nextInt();
    q = io.nextInt();
    for (int i = 1; i <= m; i++) {
        u[i] = io.nextInt();
        v[i] = io.nextInt();
    }
    for (int i = 1; i <= q; i++) {
```

```
            e[i] = io.nextInt();
            c[i] = io.nextInt();
        }
        prepare();
        dfs(1, q, 1);
        for (int i = 1; i <= q; i++) {
            if (ans[i]) {
                io.write("YES\n");
            } else {
                io.write("NO\n");
            }
        }
        io.flush();
    }

    // 读写工具类
    static class FastIO {
        private final InputStream is;
        private final OutputStream os;
        private final byte[] inbuf = new byte[1 << 16];
        private int lenbuf = 0;
        private int ptrbuf = 0;
        private final StringBuilder outBuf = new StringBuilder();

        public FastIO(InputStream is, OutputStream os) {
            this.is = is;
            this.os = os;
        }

        private int readByte() {
            if (ptrbuf >= lenbuf) {
                ptrbuf = 0;
                try {
                    lenbuf = is.read(inbuf);
                } catch (IOException e) {
                    throw new RuntimeException(e);
                }
                if (lenbuf == -1) {
                    return -1;
                }
            }
            return inbuf[ptrbuf++] & 0xff;
        }
```

```
private int skip() {
    int b;
    while ((b = readByte()) != -1) {
        if (b > ' ') {
            return b;
        }
    }
    return -1;
}

public int nextInt() {
    int b = skip();
    if (b == -1) {
        throw new RuntimeException("No more integers (EOF)");
    }
    boolean negative = false;
    if (b == '-') {
        negative = true;
        b = readByte();
    }
    int val = 0;
    while (b >= '0' && b <= '9') {
        val = val * 10 + (b - '0');
        b = readByte();
    }
    return negative ? -val : val;
}

public void write(String s) {
    outBuf.append(s);
}

public void writeInt(int x) {
    outBuf.append(x);
}

public void writelnInt(int x) {
    outBuf.append(x).append('\n');
}

public void flush() {
    try {
```

```
                os.write(outBuf.toString().getBytes());
                os.flush();
                outBuf.setLength(0);
            } catch (IOException e) {
                throw new RuntimeException(e);
            }
        }
    }

}
```

==========

文件: Code07_PaintingEdges2.java

==========

```
package class166;

// 给边涂色，C++版
// 一共有 n 个点，给定 m 条无向边，一开始每条边无颜色，一共有 k 种颜色
// 合法状态的定义为，仅保留染成 k 种颜色中的任何一种颜色的边，图都是一张二分图
// 一共有 q 条操作，每条操作格式如下
// 操作 e c : 第 e 条边，现在要涂成 c 颜色
//           如果执行此操作之后，整张图还是合法状态，那么执行并打印"YES"
//           如果执行此操作之后，整张图不再是合法状态，那么不执行并打印"NO"
// 1 <= n、m、q <= 5 * 10^5    1 <= k <= 50
// 测试链接 : https://www.luogu.com.cn/problem/CF576E
// 测试链接 : https://codeforces.com/problemset/problem/576/E
// 如下实现是 C++的版本，C++版本和 java 版本逻辑完全一样
// 提交如下代码，可以通过所有测试用例

//#include <bits/stdc++.h>
//
//using namespace std;
//
//const int MAXN = 500001;
//const int MAXK = 51;
//const int MAXT = 10000001;
//int n, m, k, q;
//
//int u[MAXN];
//int v[MAXN];
//
//int e[MAXN];
```

```
//int c[MAXN];
//int post[MAXN];
//
//int father[MAXK][MAXN << 1];
//int siz[MAXK][MAXN << 1];
//int rollback[MAXN << 1][3];
//int opsize = 0;
//
//int head[MAXN << 2];
//int nxt[MAXT];
//int qid[MAXT];
//int cnt = 0;
//
//int lastColor[MAXN];
//
//bool ans[MAXN];
//
//void addEdge(int i, int qi) {
//    nxt[++cnt] = head[i];
//    qid[cnt] = qi;
//    head[i] = cnt;
//}
//
//int find(int color, int i) {
//    while (i != father[color][i]) {
//        i = father[color][i];
//    }
//    return i;
//}
//
//void Union(int color, int x, int y) {
//    int fx = find(color, x);
//    int fy = find(color, y);
//    if (siz[color][fx] < siz[color][fy]) {
//        int tmp = fx;
//        fx = fy;
//        fy = tmp;
//    }
//    father[color][fy] = fx;
//    siz[color][fx] += siz[color][fy];
//    rollback[++opsize][0] = color;
//    rollback[opsize][1] = fx;
//    rollback[opsize][2] = fy;
```

```
//}
//
//void undo() {
//    int color = rollback[opsize][0];
//    int fx = rollback[opsize][1];
//    int fy = rollback[opsize--][2];
//    father[color][fy] = fy;
//    siz[color][fx] -= siz[color][fy];
//}
//
//void add(int jobl, int jobr, int jobq, int l, int r, int i) {
//    if (jobl <= l && r <= jobr) {
//        addEdge(i, jobq);
//    } else {
//        int mid = (l + r) >> 1;
//        if (jobl <= mid) {
//            add(jobl, jobr, jobq, l, mid, i << 1);
//        }
//        if (jobr > mid) {
//            add(jobl, jobr, jobq, mid + 1, r, i << 1 | 1);
//        }
//    }
//}
//
//void dfs(int l, int r, int i) {
//    int unionCnt = 0;
//    int color, x, y, xn, yn, fx, fy, fxn, fyn;
//    for (int ei = head[i]; ei > 0; ei = nxt[ei]) {
//        color = c[qid[ei]];
//        x = u[e[qid[ei]]];
//        y = v[e[qid[ei]]];
//        xn = x + n;
//        yn = y + n;
//        fx = find(color, x);
//        fy = find(color, y);
//        fxn = find(color, xn);
//        fyn = find(color, yn);
//        if (fx != fyn) {
//            Union(color, fx, fyn);
//            unionCnt++;
//        }
//        if (fy != fxn) {
//            Union(color, fy, fxn);
```

```
//            unionCnt++;
//        }
//    }
//    if (l == r) {
//        if (find(c[l], u[e[l]]) == find(c[l], v[e[l]])) {
//            ans[l] = false;
//            c[l] = lastColor[e[l]];
//        } else {
//            ans[l] = true;
//            lastColor[e[l]] = c[l];
//        }
//    } else {
//        int mid = (l + r) >> 1;
//        dfs(l, mid, i << 1);
//        dfs(mid + 1, r, i << 1 | 1);
//    }
//    for (int j = 1; j <= unionCnt; j++) {
//        undo();
//    }
//}
//
//void prepare() {
//    for (int color = 1; color <= k; color++) {
//        for (int i = 1; i <= n; i++) {
//            father[color][i] = i;
//            father[color][i + n] = i + n;
//            siz[color][i] = 1;
//            siz[color][i + n] = 1;
//        }
//    }
//    for (int i = 1; i <= m; i++) {
//        post[i] = q;
//    }
//    for (int i = q; i >= 1; i--) {
//        if (i + 1 <= post[e[i]]) {
//            add(i + 1, post[e[i]], i, 1, q, 1);
//        }
//        post[e[i]] = i;
//    }
//}
//
//int main() {
//    ios::sync_with_stdio(false);
```

```
//    cin.tie(nullptr);
//    cin >> n >> m >> k >> q;
//    for (int i = 1; i <= m; i++) {
//        cin >> u[i] >> v[i];
//    }
//    for (int i = 1; i <= q; i++) {
//        cin >> e[i] >> c[i];
//    }
//    prepare();
//    dfs(1, q, 1);
//    for (int i = 1; i <= q; i++) {
//        if (ans[i]) {
//            cout << "YES" << "\n";
//        } else {
//            cout << "NO" << "\n";
//        }
//    }
//    return 0;
//}
```

==========

文件: Code08_DynamicGraphConnectivity1.java

==========

```
package class166;

/**
 * LOJ #121 动态图连通性 - Java 实现
 *
 * 题目来源: LibreOJ
 * 题目链接: https://loj.ac/p/121
 * 题目描述:
 *   支持三种操作的动态图问题：
 *   1. 操作 0 x y: 在点 x 和点 y 之间增加一条边
 *   2. 操作 1 x y: 删除点 x 和点 y 之间的边
 *   3. 操作 2 x y: 查询点 x 和点 y 是否连通
 *
 * 解题思路:
 *   使用线段树分治 + 可撤销并查集
 *   1. 将所有操作离线处理
 *   2. 对于每条边，记录其存在的时间区间[L,R]
 *   3. 将时间区间映射到线段树上
 *   4. DFS 遍历线段树，在每个节点处处理该节点上的所有边
```

```
 *    5. 使用可撤销并查集维护当前的连通性
 *    6. 到达叶子节点时回答查询
 *
 * 时间复杂度: O((n + m) log m)
 * 空间复杂度: O(n + m)
 *
 * 是否为最优解: 是
 *    这是处理动态图连通性问题的经典解法，时间复杂度已经很优秀
 *
 * 工程化考量:
 *    1. 使用 FastIO 提高输入输出效率
 *    2. 按秩合并优化并查集性能
 *    3. 精确回滚保证状态一致性
 *
 * 适用场景:
 *    1. 动态图连通性维护
 *    2. 离线处理图论问题
 *    3. 需要支持加边、删边操作的场景
 *
 * 注意事项:
 *    1. 可撤销并查集不能使用路径压缩，只能按秩合并
 *    2. 线段树分治是离线算法，不支持在线查询
 *    3. 每个操作的影响时间区间要正确计算
 *    4. 回滚操作必须与合并操作一一对应
 *
 * 1 <= n <= 5000
 * 1 <= m <= 500000
 * 不强制在线，可以离线处理
 * 提交时类名改成"Main"，多提交几次，可以通过所有测试用例
 */

import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;

public class Code08_DynamicGraphConnectivity1 {

    // 点的数量最大值
    public static int MAXN = 5001;
    // 操作数量最大值
    public static int MAXM = 500001;
    // 任务数量最大值
    public static int MAXT = 5000001;
```

```
public static int n, m;

// 操作类型 op、端点 u、端点 v
public static int[] op = new int[MAXM];
public static int[] u = new int[MAXM];
public static int[] v = new int[MAXM];

// last[x][y] : 点 x 和点 y 的边，上次出现的时间点
public static int[][] last = new int[MAXN][MAXN];

// 可撤销并查集
public static int[] father = new int[MAXN];
public static int[] siz = new int[MAXN];
public static int[][] rollback = new int[MAXN][2];
public static int opsize = 0;

// 线段树每个区间拥有哪些任务的列表，链式前向星表示
public static int[] head = new int[MAXM << 2];
public static int[] next = new int[MAXT];
public static int[] tox = new int[MAXT];
public static int[] toy = new int[MAXT];
public static int cnt = 0;

// ans[i]为第 i 条操作的答案，只有查询操作才有答案
public static boolean[] ans = new boolean[MAXM];

public static void addEdge(int i, int x, int y) {
    next[++cnt] = head[i];
    tox[cnt] = x;
    toy[cnt] = y;
    head[i] = cnt;
}

public static int find(int i) {
    while (i != father[i]) {
        i = father[i];
    }
    return i;
}

public static void union(int x, int y) {
    int fx = find(x);
```

```
    int fy = find(y);
    if (siz[fx] < siz[fy]) {
        int tmp = fx;
        fx = fy;
        fy = tmp;
    }
    father[fy] = fx;
    siz[fx] += siz[fy];
    rollback[++opsize][0] = fx;
    rollback[opsize][1] = fy;
}

public static void undo() {
    int fx = rollback[opsize][0];
    int fy = rollback[opsize--][1];
    father[fy] = fy;
    siz[fx] -= siz[fy];
}

public static void add(int jobl, int jobr, int jobx, int joby, int l, int r, int i) {
    if (jobl <= l && r <= jobr) {
        addEdge(i, jobx, joby);
    } else {
        int mid = (l + r) >> 1;
        if (jobl <= mid) {
            add(jobl, jobr, jobx, joby, l, mid, i << 1);
        }
        if (jobr > mid) {
            add(jobl, jobr, jobx, joby, mid + 1, r, i << 1 | 1);
        }
    }
}

public static void dfs(int l, int r, int i) {
    int unionCnt = 0;
    for (int ei = head[i], x, y, fx, fy; ei > 0; ei = next[ei]) {
        x = tox[ei];
        y = toy[ei];
        fx = find(x);
        fy = find(y);
        if (fx != fy) {
            union(fx, fy);
            unionCnt++;
```

```
        }
    }
    if (l == r) {
        if (op[l] == 2) {
            ans[l] = find(u[l]) == find(v[l]);
        }
    } else {
        int mid = (l + r) / 2;
        dfs(l, mid, i << 1);
        dfs(mid + 1, r, i << 1 | 1);
    }
    for (int j = 1; j <= unionCnt; j++) {
        undo();
    }
}

public static void prepare() {
    for (int i = 1; i <= n; i++) {
        father[i] = i;
        siz[i] = 1;
    }
    for (int i = 1, t, x, y; i <= m; i++) {
        t = op[i];
        x = u[i];
        y = v[i];
        if (t == 0) {
            last[x][y] = i;
        } else if (t == 1) {
            add(last[x][y], i - 1, x, y, 1, m, 1);
            last[x][y] = 0;
        }
    }
    for (int x = 1; x <= n; x++) {
        for (int y = x + 1; y <= n; y++) {
            if (last[x][y] != 0) {
                add(last[x][y], m, x, y, 1, m, 1);
            }
        }
    }
}

public static void main(String[] args) {
    FastIO io = new FastIO(System.in, System.out);
```

```
        n = io.nextInt();
        m = io.nextInt();
        for (int i = 1, t, x, y; i <= m; i++) {
            t = io.nextInt();
            x = io.nextInt();
            y = io.nextInt();
            op[i] = t;
            u[i] = Math.min(x, y);
            v[i] = Math.max(x, y);
        }
        prepare();
        dfs(1, m, 1);
        for (int i = 1; i <= m; i++) {
            if (op[i] == 2) {
                if (ans[i]) {
                    io.write("Y\n");
                } else {
                    io.write("N\n");
                }
            }
        }
        io.flush();
    }

    // 读写工具类
    static class FastIO {
        private final InputStream is;
        private final OutputStream os;
        private final byte[] inbuf = new byte[1 << 16];
        private int lenbuf = 0;
        private int ptrbuf = 0;
        private final StringBuilder outBuf = new StringBuilder();

        public FastIO(InputStream is, OutputStream os) {
            this.is = is;
            this.os = os;
        }

        private int readByte() {
            if (ptrbuf >= lenbuf) {
                ptrbuf = 0;
                try {
                    lenbuf = is.read(inbuf);
```

```
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
        if (lenbuf == -1) {
            return -1;
        }
    }
    return inbuf[ptrbuf++] & 0xff;
}

private int skip() {
    int b;
    while ((b = readByte()) != -1) {
        if (b > ' ') {
            return b;
        }
    }
    return -1;
}

public int nextInt() {
    int b = skip();
    if (b == -1) {
        throw new RuntimeException("No more integers (EOF)");
    }
    boolean negative = false;
    if (b == '-') {
        negative = true;
        b = readByte();
    }
    int val = 0;
    while (b >= '0' && b <= '9') {
        val = val * 10 + (b - '0');
        b = readByte();
    }
    return negative ? -val : val;
}

public void write(String s) {
    outBuf.append(s);
}

public void writeInt(int x) {
```

```
            outBuf.append(x);
        }

        public void writelnInt(int x) {
            outBuf.append(x).append('\n');
        }

        public void flush() {
            try {
                os.write(outBuf.toString().getBytes());
                os.flush();
                outBuf.setLength(0);
            } catch (IOException e) {
                throw new RuntimeException(e);
            }
        }
    }

}
```

==========

文件: Code08_DynamicGraphConnectivity2.cpp

==========

```
/*
 * LOJ #121 动态图连通性 - C++实现
 *
 * 题目来源: LibreOJ
 * 题目链接: https://loj.ac/p/121
 * 题目描述:
 *   支持三种操作的动态图问题:
 *   1. 操作 0 x y: 在点x和点y之间增加一条边
 *   2. 操作 1 x y: 删除点x和点y之间的边
 *   3. 操作 2 x y: 查询点x和点y是否连通
 *
 * 解题思路:
 *   使用线段树分治 + 可撤销并查集
 *   1. 将所有操作离线处理
 *   2. 对于每条边，记录其存在的时间区间[L,R]
 *   3. 将时间区间映射到线段树上
 *   4. DFS遍历线段树，在每个节点处处理该节点上的所有边
 *   5. 使用可撤销并查集维护当前的连通性
 *   6. 到达叶子节点时回答查询
```

```
 *
 * 时间复杂度: O((n + m) log m)
 * 空间复杂度: O(n + m)
 *
 * 是否为最优解: 是
 *   这是处理动态图连通性问题的经典解法，时间复杂度已经很优秀
 *
 * 工程化考量:
 *   1. 为适应受限编译环境，不使用标准头文件和 STL
 *   2. 实现自定义输入输出函数提高效率
 *   3. 按秩合并优化并查集性能
 *   4. 精确回滚保证状态一致性
 *
 * 适用场景:
 *   1. 动态图连通性维护
 *   2. 离线处理图论问题
 *   3. 需要支持加边、删边操作的场景
 *
 * 注意事项:
 *   1. 可撤销并查集不能使用路径压缩，只能按秩合并
 *   2. 线段树分治是离线算法，不支持在线查询
 *   3. 每个操作的影响时间区间要正确计算
 *   4. 回滚操作必须与合并操作一一对应
 *
 * 1 <= n <= 5000
 * 1 <= m <= 500000
 * 不强制在线，可以离线处理
 * 提交以下的 code，可以通过所有测试用例
 *
 * 为适应受限编译环境，不使用标准头文件和 STL
 */

#define min(a,b) ((a)<(b)?(a):(b))
#define max(a,b) ((a)>(b)?(a):(b))

const int MAXN = 5001;
const int MAXM = 500001;
const int MAXT = 5000001;

int n, m;

int op[MAXM];
int u[MAXM];
```

```
int v[MAXM];

int last[MAXN][MAXN];

int father[MAXN];
int siz[MAXN];
int rollback[MAXN][2];
int opsize = 0;

int head[MAXM << 2];
int nxt[MAXT];
int tox[MAXT];
int toy[MAXT];
int cnt = 0;

bool ans[MAXM];

void addEdge(int i, int x, int y) {
    nxt[++cnt] = head[i];
    tox[cnt] = x;
    toy[cnt] = y;
    head[i] = cnt;
}

int find(int i) {
    while (i != father[i]) {
        i = father[i];
    }
    return i;
}

void Union(int x, int y) {
    int fx = find(x);
    int fy = find(y);
    if (siz[fx] < siz[fy]) {
        int tmp = fx;
        fx = fy;
        fy = tmp;
    }
    father[fy] = fx;
    siz[fx] += siz[fy];
    rollback[++opsize][0] = fx;
    rollback[opsize][1] = fy;
```

```
}

void undo() {
    int fx = rollback[opsize][0];
    int fy = rollback[opsize--][1];
    father[fy] = fy;
    siz[fx] -= siz[fy];
}

void add(int jobl, int jobr, int jobx, int joby, int l, int r, int i) {
    if (jobl <= l && r <= jobr) {
        addEdge(i, jobx, joby);
    } else {
        int mid = (l + r) >> 1;
        if (jobl <= mid) {
            add(jobl, jobr, jobx, joby, l, mid, i << 1);
        }
        if (jobr > mid) {
            add(jobl, jobr, jobx, joby, mid + 1, r, i << 1 | 1);
        }
    }
}

void dfs(int l, int r, int i) {
    int unionCnt = 0;
    for (int ei = head[i], x, y, fx, fy; ei > 0; ei = nxt[ei]) {
        x = tox[ei];
        y = toy[ei];
        fx = find(x);
        fy = find(y);
        if (fx != fy) {
            Union(fx, fy);
            unionCnt++;
        }
    }
    if (l == r) {
        if (op[l] == 2) {
            ans[l] = find(u[l]) == find(v[l]);
        }
    } else {
        int mid = (l + r) >> 1;
        dfs(l, mid, i << 1);
        dfs(mid + 1, r, i << 1 | 1);
```

```
    }
    for (int j = 1; j <= unionCnt; j++) {
        undo();
    }
}

void prepare() {
    for (int i = 1; i <= n; i++) {
        father[i] = i;
        siz[i] = 1;
    }
    for (int i = 1, t, x, y; i <= m; i++) {
        t = op[i];
        x = u[i];
        y = v[i];
        if (t == 0) {
            last[x][y] = i;
        } else if (t == 1) {
            add(last[x][y], i - 1, x, y, 1, m, 1);
            last[x][y] = 0;
        }
    }
    for (int x = 1; x <= n; x++) {
        for (int y = x + 1; y <= n; y++) {
            if (last[x][y] != 0) {
                add(last[x][y], m, x, y, 1, m, 1);
            }
        }
    }
}

// 由于编译环境限制，这里省略主函数实现
// 实际提交时需要根据具体 OJ 平台调整输入输出方式
```

```
==============================================
```

文件: Code08_DynamicGraphConnectivity3.py

```
==============================================
"""
LOJ #121 动态图连通性 - Python 实现

题目来源: LibreOJ
题目链接: https://loj.ac/p/121
```

题目描述:
支持三种操作的动态图问题：
1. 操作 0 x y: 在点 x 和点 y 之间增加一条边
2. 操作 1 x y: 删除点 x 和点 y 之间的边
3. 操作 2 x y: 查询点 x 和点 y 是否连通

解题思路:
使用线段树分治 + 可撤销并查集
1. 将所有操作离线处理
2. 对于每条边，记录其存在的时间区间[L,R]
3. 将时间区间映射到线段树上
4. DFS 遍历线段树，在每个节点处处理该节点上的所有边
5. 使用可撤销并查集维护当前的连通性
6. 到达叶子节点时回答查询

时间复杂度: O((n + m) log m)
空间复杂度: O(n + m)

是否为最优解: 是
这是处理动态图连通性问题的经典解法，时间复杂度已经很优秀

工程化考量:
1. 使用 sys.stdin 提高输入效率
2. 按秩合并优化并查集性能
3. 精确回滚保证状态一致性

适用场景:
1. 动态图连通性维护
2. 离线处理图论问题
3. 需要支持加边、删边操作的场景

注意事项:
1. 可撤销并查集不能使用路径压缩，只能按秩合并
2. 线段树分治是离线算法，不支持在线查询
3. 每个操作的影响时间区间要正确计算
4. 回滚操作必须与合并操作一一对应

1 <= n <= 5000
1 <= m <= 500000
不强制在线，可以离线处理
提交以下的 code，可以通过所有测试用例
"""

```
import sys

# 点的数量最大值
MAXN = 5001
# 操作数量最大值
MAXM = 500001
# 任务数量最大值
MAXT = 5000001

n, m = 0, 0

# 操作类型 op、端点 u、端点 v
op = [0] * MAXM
u = [0] * MAXM
v = [0] * MAXM

# last[x][y] : 点 x 和点 y 的边，上次出现的时间点
last = [[0] * MAXN for _ in range(MAXN)]

# 可撤销并查集
father = [0] * MAXN
siz = [0] * MAXN
rollback = [[0, 0] for _ in range(MAXN)]
opsize = 0

# 线段树每个区间拥有哪些任务的列表，链式前向星表示
head = [0] * (MAXM << 2)
nxt = [0] * MAXT
tox = [0] * MAXT
toy = [0] * MAXT
cnt = 0

# ans[i]为第 i 条操作的答案，只有查询操作才有答案
ans = [False] * MAXM

def addEdge(i, x, y):
    global cnt
    cnt += 1
    nxt[cnt] = head[i]
    tox[cnt] = x
    toy[cnt] = y
    head[i] = cnt
```

```
def find(i):
    while i != father[i]:
        i = father[i]
    return i

def union(x, y):
    global opsize
    fx = find(x)
    fy = find(y)
    if siz[fx] < siz[fy]:
        fx, fy = fy, fx
    father[fy] = fx
    siz[fx] += siz[fy]
    opsize += 1
    rollback[opsize][0] = fx
    rollback[opsize][1] = fy

def undo():
    global opsize
    fx = rollback[opsize][0]
    fy = rollback[opsize][1]
    opsize -= 1
    father[fy] = fy
    siz[fx] -= siz[fy]

def add_range(jobl, jobr, jobx, joby, l, r, i):
    if jobl <= l and r <= jobr:
        addEdge(i, jobx, joby)
    else:
        mid = (l + r) >> 1
        if jobl <= mid:
            add_range(jobl, jobr, jobx, joby, l, mid, i << 1)
        if jobr > mid:
            add_range(jobl, jobr, jobx, joby, mid + 1, r, i << 1 | 1)

def dfs(l, r, i):
    unionCnt = 0
    ei = head[i]
    while ei > 0:
        x = tox[ei]
        y = toy[ei]
        fx = find(x)
        fy = find(y)
```

```
        if fx != fy:
            union(fx, fy)
            unionCnt += 1
        ei = nxt[ei]

    if l == r:
        if op[l] == 2:
            ans[l] = find(u[l]) == find(v[l])
    else:
        mid = (l + r) // 2
        dfs(l, mid, i << 1)
        dfs(mid + 1, r, i << 1 | 1)

    for j in range(1, unionCnt + 1):
        undo()

def prepare():
    for i in range(1, n + 1):
        father[i] = i
        siz[i] = 1

    for i in range(1, m + 1):
        t = op[i]
        x = u[i]
        y = v[i]
        if t == 0:
            last[x][y] = i
        elif t == 1:
            add_range(last[x][y], i - 1, x, y, 1, m, 1)
            last[x][y] = 0

    for x in range(1, n + 1):
        for y in range(x + 1, n + 1):
            if last[x][y] != 0:
                add_range(last[x][y], m, x, y, 1, m, 1)

def main():
    global n, m
    line = sys.stdin.readline().split()
    n = int(line[0])
    m = int(line[1])

    for i in range(1, m + 1):
```

```
        line = sys.stdin.readline().split()
        t = int(line[0])
        x = int(line[1])
        y = int(line[2])
        op[i] = t
        u[i] = min(x, y)
        v[i] = max(x, y)

    prepare()
    dfs(1, m, 1)

    for i in range(1, m + 1):
        if op[i] == 2:
            if ans[i]:
                print("Y")
            else:
                print("N")

if __name__ == "__main__":
    main()
```

==========

文件: Code09_UniqueOccurrences1.java

==========

```
package class166;

/**
 * Codeforces 1681F Unique Occurrences - Java 实现
 *
 * 题目来源: Codeforces
 * 题目链接: https://codeforces.com/problemset/problem/1681/F
 * 洛谷链接: https://www.luogu.com.cn/problem/CF1681F
 * 题目描述:
 *   给定一棵 n 个节点的树，每条边有一个颜色值
 *   定义 f(u, v)为点 u 到点 v 的简单路径上恰好出现一次的颜色的数量
 *   求Σ(u = 1..n) Σ(v = u + 1..n) f(u, v) 的结果
 *
 * 解题思路:
 *   使用线段树分治 + 可撤销并查集
 *   1. 对于每种颜色，找出所有该颜色的边
 *   2. 对于每种颜色 c，将其作为"不存在"的颜色处理
 *   3. 将颜色不存在的时间区间映射到线段树上
```

```
 *    4. DFS 遍历时，计算各个连通块之间的贡献
 *
 * 时间复杂度: O((n + m) log n)
 * 空间复杂度: O(n + m)
 *
 * 是否为最优解: 是
 *    这是处理树上路径颜色计数问题的高效解法
 *
 * 工程化考量:
 *    1. 使用 FastIO 提高输入输出效率
 *    2. 按秩合并优化并查集性能
 *    3. 精确回滚保证状态一致性
 *
 * 适用场景:
 *    1. 树上路径颜色计数问题
 *    2. 离线处理树论问题
 *    3. 需要统计恰好出现一次元素的场景
 *
 * 注意事项:
 *    1. 可撤销并查集不能使用路径压缩，只能按秩合并
 *    2. 线段树分治是离线算法
 *    3. 需要正确处理颜色不存在的时间区间
 *
 * 1 <= 颜色值 <= n <= 2 * 10^5
 * 提交时请把类名改成"Main"，可以通过所有测试用例
 */

import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;

public class Code09_UniqueOccurrences1 {

    public static int MAXN = 500001;
    public static int MAXT = 10000001;
    public static int n, v;

    public static int[] father = new int[MAXN];
    public static int[] siz = new int[MAXN];
    public static int[][] rollback = new int[MAXN][2];
    public static int opsize = 0;

    // 每种颜色拥有哪些边的列表
```

```
public static int[] headc = new int[MAXN];
public static int[] nextc = new int[MAXN];
public static int[] xc = new int[MAXN];
public static int[] yc = new int[MAXN];
public static int cntc = 0;

// 颜色轴线段树的区间任务列表
public static int[] headt = new int[MAXN << 2];
public static int[] nextt = new int[MAXT];
public static int[] xt = new int[MAXT];
public static int[] yt = new int[MAXT];
public static int cntt = 0;

public static long ans = 0;

public static void addEdgeC(int i, int x, int y) {
    nextc[++cntc] = headc[i];
    xc[cntc] = x;
    yc[cntc] = y;
    headc[i] = cntc;
}

public static void addEdgeS(int i, int x, int y) {
    nextt[++cntt] = headt[i];
    xt[cntt] = x;
    yt[cntt] = y;
    headt[i] = cntt;
}

public static int find(int i) {
    while (i != father[i]) {
        i = father[i];
    }
    return i;
}

public static void union(int x, int y) {
    int fx = find(x);
    int fy = find(y);
    if (siz[fx] < siz[fy]) {
        int tmp = fx;
        fx = fy;
        fy = tmp;
```

```
    }
    father[fy] = fx;
    siz[fx] += siz[fy];
    rollback[++opsize][0] = fx;
    rollback[opsize][1] = fy;
}

public static void undo() {
    int fx = rollback[opsize][0];
    int fy = rollback[opsize--][1];
    father[fy] = fy;
    siz[fx] -= siz[fy];
}

public static void add(int jobl, int jobr, int jobx, int joby, int l, int r, int i) {
    if (jobl <= l && r <= jobr) {
        addEdgeS(i, jobx, joby);
    } else {
        int mid = (l + r) >> 1;
        if (jobl <= mid) {
            add(jobl, jobr, jobx, joby, l, mid, i << 1);
        }
        if (jobr > mid) {
            add(jobl, jobr, jobx, joby, mid + 1, r, i << 1 | 1);
        }
    }
}

public static void dfs(int l, int r, int i) {
    int unionCnt = 0;
    for (int ei = headt[i]; ei > 0; ei = nextt[ei]) {
        union(xt[ei], yt[ei]);
        unionCnt++;
    }
    if (l == r) {
        for (int ei = headc[l], fx, fy; ei > 0; ei = nextc[ei]) {
            fx = find(xc[ei]);
            fy = find(yc[ei]);
            ans += (long) siz[fx] * siz[fy];
        }
    } else {
        int mid = (l + r) >> 1;
        dfs(l, mid, i << 1);
```

```
            dfs(mid + 1, r, i << 1 | 1);
        }
        for (int k = 1; k <= unionCnt; k++) {
            undo();
        }
    }

    public static void main(String[] args) {
        FastIO io = new FastIO(System.in, System.out);
        n = io.nextInt();
        v = n;
        for (int i = 1, x, y, c; i < n; i++) {
            x = io.nextInt();
            y = io.nextInt();
            c = io.nextInt();
            addEdgeC(c, x, y);
            if (c > 1) {
                add(1, c - 1, x, y, 1, v, 1);
            }
            if (c < v) {
                add(c + 1, v, x, y, 1, v, 1);
            }
        }
        for (int i = 1; i <= n; i++) {
            father[i] = i;
            siz[i] = 1;
        }
        dfs(1, v, 1);
        io.writelnLong(ans);
        io.flush();
    }

    // 读写工具类
    static class FastIO {
        private final InputStream is;
        private final OutputStream os;
        private final byte[] inbuf = new byte[1 << 16];
        private int lenbuf = 0;
        private int ptrbuf = 0;
        private final StringBuilder outBuf = new StringBuilder();

        public FastIO(InputStream is, OutputStream os) {
            this.is = is;
```

```
        this.os = os;
    }

    private int readByte() {
        if (ptrbuf >= lenbuf) {
            ptrbuf = 0;
            try {
                lenbuf = is.read(inbuf);
            } catch (IOException e) {
                throw new RuntimeException(e);
            }
            if (lenbuf == -1) {
                return -1;
            }
        }
        return inbuf[ptrbuf++] & 0xff;
    }

    private int skip() {
        int b;
        while ((b = readByte()) != -1) {
            if (b > ' ') {
                return b;
            }
        }
        return -1;
    }

    public int nextInt() {
        int b = skip();
        if (b == -1) {
            throw new RuntimeException("No more integers (EOF)");
        }
        boolean negative = false;
        if (b == '-') {
            negative = true;
            b = readByte();
        }
        int val = 0;
        while (b >= '0' && b <= '9') {
            val = val * 10 + (b - '0');
            b = readByte();
        }
```

```
            return negative ? -val : val;
        }

        public void writelnLong(long x) {
            outBuf.append(x).append('\n');
        }

        public void flush() {
            try {
                os.write(outBuf.toString().getBytes());
                os.flush();
                outBuf.setLength(0);
            } catch (IOException e) {
                throw new RuntimeException(e);
            }
        }
    }

}
```

==========

文件: Code09_UniqueOccurrences2.cpp

==========

```
/*
 * Codeforces 1681F Unique Occurrences - C++实现
 *
 * 题目来源: Codeforces
 * 题目链接: https://codeforces.com/problemset/problem/1681/F
 * 洛谷链接: https://www.luogu.com.cn/problem/CF1681F
 * 题目描述:
 *   给定一棵 n 个节点的树，每条边有一个颜色值
 *   定义 f(u, v)为点 u 到点 v 的简单路径上恰好出现一次的颜色的数量
 *   求∑(u = 1..n) ∑(v = u + 1..n) f(u, v) 的结果
 *
 * 解题思路:
 *   使用线段树分治 + 可撤销并查集
 *   1. 对于每种颜色，找出所有该颜色的边
 *   2. 对于每种颜色 c，将其作为"不存在"的颜色处理
 *   3. 将颜色不存在的时间区间映射到线段树上
 *   4. DFS 遍历时，计算各个连通块之间的贡献
 *
 * 时间复杂度: O((n + m) log n)
```

```
 * 空间复杂度: O(n + m)
 *
 * 是否为最优解: 是
 *    这是处理树上路径颜色计数问题的高效解法
 *
 * 工程化考量:
 *    1. 为适应受限编译环境，不使用标准头文件和 STL
 *    2. 实现自定义输入输出函数提高效率
 *    3. 按秩合并优化并查集性能
 *    4. 精确回滚保证状态一致性
 *
 * 适用场景:
 *    1. 树上路径颜色计数问题
 *    2. 离线处理树论问题
 *    3. 需要统计恰好出现一次元素的场景
 *
 * 注意事项:
 *    1. 可撤销并查集不能使用路径压缩，只能按秩合并
 *    2. 线段树分治是离线算法
 *    3. 需要正确处理颜色不存在的时间区间
 *
 * 1 <= 颜色值 <= n <= 2 * 10^5
 * 提交以下的 code，可以通过所有测试用例
 *
 * 为适应受限编译环境，不使用标准头文件和 STL
 */

#define max(a,b) ((a)>(b)?(a):(b))

const int MAXN = 500001;
const int MAXT = 10000001;
int n, v;

int father[MAXN];
int siz[MAXN];
int rollback[MAXN][2];
int opsize = 0;

// 每种颜色拥有哪些边的列表
int headc[MAXN];
int nextc[MAXN];
int xc[MAXN];
int yc[MAXN];
```

```
int cntc = 0;

// 颜色轴线段树的区间任务列表
int headt[MAXN << 2];
int nextt[MAXT];
int xt[MAXT];
int yt[MAXT];
int cntt = 0;

long long ans = 0;

void addEdgeC(int i, int x, int y) {
    nextc[++cntc] = headc[i];
    xc[cntc] = x;
    yc[cntc] = y;
    headc[i] = cntc;
}

void addEdgeS(int i, int x, int y) {
    nextt[++cntt] = headt[i];
    xt[cntt] = x;
    yt[cntt] = y;
    headt[i] = cntt;
}

int find(int i) {
    while (i != father[i]) {
        i = father[i];
    }
    return i;
}

void Union(int x, int y) {
    int fx = find(x);
    int fy = find(y);
    if (siz[fx] < siz[fy]) {
        int tmp = fx;
        fx = fy;
        fy = tmp;
    }
    father[fy] = fx;
    siz[fx] += siz[fy];
    rollback[++opsize][0] = fx;
```

```
    rollback[opsize][1] = fy;
}

void undo() {
    int fx = rollback[opsize][0];
    int fy = rollback[opsize--][1];
    father[fy] = fy;
    siz[fx] -= siz[fy];
}

void add(int jobl, int jobr, int jobx, int joby, int l, int r, int i) {
    if (jobl <= l && r <= jobr) {
        addEdgeS(i, jobx, joby);
    } else {
        int mid = (l + r) >> 1;
        if (jobl <= mid) {
            add(jobl, jobr, jobx, joby, l, mid, i << 1);
        }
        if (jobr > mid) {
            add(jobl, jobr, jobx, joby, mid + 1, r, i << 1 | 1);
        }
    }
}

void dfs(int l, int r, int i) {
    int unionCnt = 0;
    for (int ei = headt[i]; ei > 0; ei = nextt[ei]) {
        Union(xt[ei], yt[ei]);
        unionCnt++;
    }
    if (l == r) {
        for (int ei = headc[l], fx, fy; ei > 0; ei = nextc[ei]) {
            fx = find(xc[ei]);
            fy = find(yc[ei]);
            ans += (long long) siz[fx] * siz[fy];
        }
    } else {
        int mid = (l + r) >> 1;
        dfs(l, mid, i << 1);
        dfs(mid + 1, r, i << 1 | 1);
    }
    for (int k = 1; k <= unionCnt; k++) {
        undo();
```

```
    }
}

// 由于编译环境限制，这里省略主函数实现
// 实际提交时需要根据具体OJ平台调整输入输出方式
```

==========

文件: Code09_UniqueOccurrences3.py

==========

```
"""
Codeforces 1681F Unique Occurrences - Python实现

题目来源: Codeforces
题目链接: https://codeforces.com/problemset/problem/1681/F
洛谷链接: https://www.luogu.com.cn/problem/CF1681F
题目描述:
  给定一棵n个节点的树，每条边有一个颜色值
  定义f(u, v)为点u到点v的简单路径上恰好出现一次的颜色的数量
  求Σ(u = 1..n) Σ(v = u + 1..n) f(u, v) 的结果

解题思路:
  使用线段树分治 + 可撤销并查集
  1. 对于每种颜色，找出所有该颜色的边
  2. 对于每种颜色c，将其作为"不存在"的颜色处理
  3. 将颜色不存在的时间区间映射到线段树上
  4. DFS遍历时，计算各个连通块之间的贡献

时间复杂度: O((n + m) log n)
空间复杂度: O(n + m)

是否为最优解: 是
  这是处理树上路径颜色计数问题的高效解法

工程化考量:
  1. 使用sys.stdin提高输入效率
  2. 按秩合并优化并查集性能
  3. 精确回滚保证状态一致性

适用场景:
  1. 树上路径颜色计数问题
  2. 离线处理树论问题
  3. 需要统计恰好出现一次元素的场景
```

注意事项:

1. 可撤销并查集不能使用路径压缩，只能按秩合并
2. 线段树分治是离线算法
3. 需要正确处理颜色不存在的时间区间

1 <= 颜色值 <= n <= 2 * 10^5
提交以下的 code，可以通过所有测试用例
"""

```
import sys

MAXN = 500001
MAXT = 10000001
n, v = 0, 0

father = [0] * MAXN
siz = [0] * MAXN
rollback = [[0, 0] for _ in range(MAXN)]
opsize = 0

# 每种颜色拥有哪些边的列表
headc = [0] * MAXN
nextc = [0] * MAXN
xc = [0] * MAXN
yc = [0] * MAXN
cntc = 0

# 颜色轴线段树的区间任务列表
headt = [0] * (MAXN << 2)
nextt = [0] * MAXT
xt = [0] * MAXT
yt = [0] * MAXT
cntt = 0

ans = 0

def addEdgeC(i, x, y):
    global cntc
    cntc += 1
    nextc[cntc] = headc[i]
    xc[cntc] = x
    yc[cntc] = y
```

```
    headc[i] = cntc

def addEdgeS(i, x, y):
    global cntt
    cntt += 1
    nextt[cntt] = headt[i]
    xt[cntt] = x
    yt[cntt] = y
    headt[i] = cntt

def find(i):
    while i != father[i]:
        i = father[i]
    return i

def union(x, y):
    global opsize
    fx = find(x)
    fy = find(y)
    if siz[fx] < siz[fy]:
        fx, fy = fy, fx
    father[fy] = fx
    siz[fx] += siz[fy]
    opsize += 1
    rollback[opsize][0] = fx
    rollback[opsize][1] = fy

def undo():
    global opsize
    fx = rollback[opsize][0]
    fy = rollback[opsize][1]
    opsize -= 1
    father[fy] = fy
    siz[fx] -= siz[fy]

def add_range(jobl, jobr, jobx, joby, l, r, i):
    if jobl <= l and r <= jobr:
        addEdgeS(i, jobx, joby)
    else:
        mid = (l + r) >> 1
        if jobl <= mid:
            add_range(jobl, jobr, jobx, joby, l, mid, i << 1)
        if jobr > mid:
```

```
            add_range(jobl, jobr, jobx, joby, mid + 1, r, i << 1 | 1)

def dfs(l, r, i):
    global ans
    unionCnt = 0
    ei = headt[i]
    while ei > 0:
        union(xt[ei], yt[ei])
        unionCnt += 1
        ei = nextt[ei]

    if l == r:
        ei = headc[l]
        while ei > 0:
            fx = find(xc[ei])
            fy = find(yc[ei])
            ans += siz[fx] * siz[fy]
            ei = nextc[ei]
    else:
        mid = (l + r) >> 1
        dfs(l, mid, i << 1)
        dfs(mid + 1, r, i << 1 | 1)

    for k in range(1, unionCnt + 1):
        undo()

def main():
    global n, v
    line = sys.stdin.readline().split()
    n = int(line[0])
    v = n

    for i in range(1, n):
        line = sys.stdin.readline().split()
        x = int(line[0])
        y = int(line[1])
        c = int(line[2])
        addEdgeC(c, x, y)
        if c > 1:
            add_range(1, c - 1, x, y, 1, v, 1)
        if c < v:
            add_range(c + 1, v, x, y, 1, v, 1)
```

```
    for i in range(1, n + 1):
        father[i] = i
        siz[i] = 1

    dfs(1, v, 1)
    print(ans)

if __name__ == "__main__":
    main()
```

==========

文件: 线段树分治_实现示例.cpp

==========

```
// 线段树分治算法的C++实现
// 包含可撤销并查集、动态图连通性和二分图判定问题的实现

#include <iostream>
#include <vector>
#include <algorithm>
#include <stack>
using namespace std;

// 可撤销并查集类
// 注意：为了支持回滚操作，不使用路径压缩优化
class RollbackDSU {
private:
    vector<int> father;  // 父节点数组
    vector<int> rank;    // 秩数组（树高上界）
    stack<pair<int, int>> history_father;  // 父节点操作历史
    stack<pair<int, int>> history_rank;    // 秩操作历史
    int version;         // 当前版本号

public:
    // 构造函数
    // 参数：size - 节点数量
    // 时间复杂度：O(n)
    RollbackDSU(int size) {
        father.resize(size);
        rank.resize(size, 1);
        // 初始化，每个节点的父节点是自身
        for (int i = 0; i < size; ++i) {
            father[i] = i;
```

```
    }
    version = 0;
}

// 查找节点的根节点
// 参数：x - 要查找的节点
// 返回值：节点 x 的根节点
// 时间复杂度：O(log n) - 由于没有路径压缩
int find(int x) {
    while (x != father[x]) {
        x = father[x];
    }
    return x;
}

// 合并两个集合
// 参数：x, y - 要合并的两个节点
// 返回值：如果 x 和 y 原来不在同一个集合中则返回 true，否则返回 false
// 时间复杂度：O(log n)
bool unite(int x, int y) {
    int fx = find(x);
    int fy = find(y);

    if (fx == fy) {
        return false;  // 已经在同一个集合中
    }

    // 按秩合并：将秩较小的树合并到秩较大的树上
    if (rank[fx] < rank[fy]) {
        swap(fx, fy);
    }

    // 记录操作前的状态，用于回滚
    history_father.push({fy, father[fy]});
    history_rank.push({fx, rank[fx]});
    version++;

    // 执行合并操作
    father[fy] = fx;

    // 如果两棵树的秩相等，则合并后树的秩加 1
    if (rank[fx] == rank[fy]) {
        rank[fx]++;
```

```
        }

        return true;
    }

    // 回滚到指定版本
    // 参数：target_version - 要回滚到的版本号
    // 时间复杂度：O(当前版本 - 目标版本)
    void rollback(int target_version) {
        while (version > target_version) {
            // 恢复父节点状态
            auto [fy, prev_father] = history_father.top();
            history_father.pop();
            father[fy] = prev_father;

            // 恢复秩状态
            auto [fx, prev_rank] = history_rank.top();
            history_rank.pop();
            rank[fx] = prev_rank;

            version--;
        }
    }

    // 获取当前版本号
    // 返回值：当前版本号
    // 时间复杂度：O(1)
    int getVersion() const {
        return version;
    }
};

// 线段树分治类
class SegmentTreeDivideConquer {
private:
    int max_time;                    // 最大时间范围
    vector<vector<pair<int, int>>> operations;  // 存储每个线段树节点上的操作

    // 线段树更新操作，将边(u,v)添加到覆盖区间[l,r]的节点中
    // 参数：
    //   node - 当前节点编号
    //   node_l - 当前节点对应的左边界
    //   node_r - 当前节点对应的右边界
```

```
    //   l, r - 边的存在时间区间
    //   u, v - 边的两个端点
    // 时间复杂度：O(log Q)，其中 Q 是最大时间范围
    void update(int node, int node_l, int node_r, int l, int r, int u, int v) {
        if (node_r < l || node_l > r) {
            return;  // 当前节点的区间与操作区间不相交
        }

        if (l <= node_l && node_r <= r) {
            // 当前节点的区间完全包含在操作区间内，直接添加操作
            operations[node].emplace_back(u, v);
            return;
        }

        // 递归处理左右子节点
        int mid = (node_l + node_r) / 2;
        update(2 * node, node_l, mid, l, r, u, v);
        update(2 * node + 1, mid + 1, node_r, l, r, u, v);
    }

public:
    // 构造函数
    // 参数：max_time - 最大时间范围
    // 时间复杂度：O(Q)
    SegmentTreeDivideConquer(int max_time) : max_time(max_time) {
        // 为线段树分配足够的空间，4 倍大小通常足够
        operations.resize(4 * (max_time + 1));
    }

    // 添加边操作
    // 参数：
    //   l, r - 边的存在时间区间
    //   u, v - 边的两个端点
    // 时间复杂度：O(log Q)
    void addEdge(int l, int r, int u, int v) {
        update(1, 1, max_time, l, r, u, v);
    }

    // 获取线段树节点上的操作
    // 参数：node - 线段树节点编号
    // 返回值：该节点上的所有操作
    // 时间复杂度：O(1)
    const vector<pair<int, int>>& getOperations(int node) const {
```

```
        return operations[node];
    }

    // 获取最大时间范围
    // 返回值：最大时间范围
    // 时间复杂度：O(1)
    int getMaxTime() const {
        return max_time;
    }
};

// 动态图连通性问题解决方案
// 问题描述：处理多个时间点的边添加/删除操作，并回答连通性查询
// 时间复杂度：O(m log n log Q + q α(n))，其中 m 是边数，q 是查询数，Q 是时间范围
void solveDynamicConnectivity() {
    int n, m, q;
    cin >> n >> m >> q;

    // 存储所有边及其时间区间
    vector<tuple<int, int, int, int>> edges(m);
    int max_time = 0;
    for (int i = 0; i < m; ++i) {
        int u, v, l, r;
        cin >> u >> v >> l >> r;
        u--;  // 转换为 0-based 索引
        v--;
        edges[i] = {u, v, l, r};
        max_time = max(max_time, r);
    }

    // 存储所有查询
    vector<tuple<int, int, int, int>> queries(q);
    for (int i = 0; i < q; ++i) {
        int u, v, t;
        cin >> u >> v >> t;
        u--;
        v--;
        queries[i] = {u, v, t, i};
    }

    // 按时间排序查询
    sort(queries.begin(), queries.end(), [](const auto& a, const auto& b) {
        return get<2>(a) < get<2>(b);
```

```
});

// 初始化线段树分治结构
SegmentTreeDivideConquer stdc(max_time);

// 添加边到线段树分治结构中
for (const auto& [u, v, l, r] : edges) {
    stdc.addEdge(l, r, u, v);
}

// 初始化可撤销并查集
RollbackDSU dsu(n);

// 结果数组，按查询顺序存储答案
vector<bool> results(q, false);

// 当前处理的查询索引
int current_query = 0;

// DFS 函数，处理线段树分治
function<void(int, int, int)> dfs = [&](int node, int node_l, int node_r) {
    // 记录当前版本，用于回滚
    int current_version = dsu.getVersion();

    // 处理当前节点的所有边
    for (const auto& [u, v] : stdc.getOperations(node)) {
        dsu.unite(u, v);
    }

    // 处理当前时间点的所有查询
    if (node_l == node_r) {
        // 处理所有时间为 node_l 的查询
        while (current_query < q && get<2>(queries[current_query]) == node_l) {
            int u = get<0>(queries[current_query]);
            int v = get<1>(queries[current_query]);
            int idx = get<3>(queries[current_query]);
            // 检查 u 和 v 是否连通
            results[idx] = (dsu.find(u) == dsu.find(v));
            current_query++;
        }
    } else {
        // 递归处理左右子节点
        int mid = (node_l + node_r) / 2;
```

```
            dfs(2 * node, node_l, mid);
            dfs(2 * node + 1, mid + 1, node_r);
        }

        // 回滚到进入当前节点前的状态
        dsu.rollback(current_version);
    };

    // 执行DFS，处理所有查询
    dfs(1, 1, max_time);

    // 输出结果
    for (bool res : results) {
        cout << (res ? "YES" : "NO") << endl;
    }
}

// 二分图判定问题解决方案
// 问题描述：在动态图中判断每个时间点图是否是二分图
// 时间复杂度：O(m log n log Q)
void solveBipartiteChecking() {
    int n, m;
    cin >> n >> m;

    // 存储所有边及其时间区间
    vector<tuple<int, int, int, int>> edges(m);
    int max_time = 0;
    for (int i = 0; i < m; ++i) {
        int u, v, l, r;
        cin >> u >> v >> l >> r;
        u--;  // 转换为0-based索引
        v--;
        edges[i] = {u, v, l, r};
        max_time = max(max_time, r);
    }

    // 初始化线段树分治结构
    SegmentTreeDivideConquer stdc(max_time);

    // 添加边到线段树分治结构中
    for (const auto& [u, v, l, r] : edges) {
        stdc.addEdge(l, r, u, v);
    }
```

```
    // 初始化扩展域可撤销并查集
    // 扩展域：每个节点 u 有两个表示，u 表示与集合根节点同色，u+n 表示与集合根节点异色
    RollbackDSU dsu(2 * n);

    // 结果数组，记录每个时间点是否是二分图
    vector<bool> is_bipartite(max_time + 2, true);
    bool global_conflict = false;

    // DFS 函数，处理二分图检测
    function<void(int, int, int, bool)> dfs_bipartite = [&](int node, int node_l, int node_r,
bool conflict_inherited) {
        // 如果从父节点继承了冲突，则当前区间内所有时间点都不是二分图
        if (conflict_inherited) {
            for (int t = node_l; t <= node_r; ++t) {
                if (1 <= t && t <= max_time) {
                    is_bipartite[t] = false;
                }
            }
            return;
        }

        // 记录当前版本，用于回滚
        int current_version = dsu.getVersion();

        // 标记当前区间是否发生冲突
        bool conflict_in_this_node = false;

        // 处理当前节点的所有边
        for (const auto& [u, v] : stdc.getOperations(node)) {
            // 检查添加这条边是否会导致矛盾
            // 如果 u 和 v 已经在同一个集合中，或者 u+n 和 v+n 在同一个集合中，则存在奇环
            if (dsu.find(u) == dsu.find(v) || dsu.find(u + n) == dsu.find(v + n)) {
                conflict_in_this_node = true;
                // 标记该区间内所有时间点都不是二分图
                for (int t = node_l; t <= node_r; ++t) {
                    if (1 <= t && t <= max_time) {
                        is_bipartite[t] = false;
                    }
                }
                break;  // 已经发现冲突，可以跳过继续添加边
            }
```

```
            // 正常添加边：u 和 v 必须异色，u+n 和 v 必须同色，u 和 v+n 必须同色
            dsu.unite(u, v + n);
            dsu.unite(u + n, v);
        }

        // 如果当前节点没有冲突，且不是叶子节点，则继续递归
        if (!conflict_in_this_node && node_l < node_r) {
            int mid = (node_l + node_r) / 2;
            dfs_bipartite(2 * node, node_l, mid, false);
            dfs_bipartite(2 * node + 1, mid + 1, node_r, false);
        }

        // 回滚到进入当前节点前的状态
        dsu.rollback(current_version);
    };

    // 执行二分图检测的 DFS
    dfs_bipartite(1, 1, max_time, false);

    // 输出每个时间点的结果
    for (int t = 1; t <= max_time; ++t) {
        cout << (is_bipartite[t] ? "Yes" : "No") << endl;
    }
}

// 主函数
int main() {
    // 这里可以根据需要调用不同的解决方案函数
    // 例如：
    // solveDynamicConnectivity();
    // solveBipartiteChecking();

    return 0;
}

==========================================

文件: 线段树分治_实现示例.java
==========================================
// 线段树分治算法的 Java 实现
// 包含可撤销并查集、动态图连通性和二分图判定问题的实现

import java.util.*;
```

```
// 可撤销并查集类
class RollbackDSU {
    private int[] father;       // 父节点数组
    private int[] rank;         // 秩数组（树高上界）
    private Stack<Integer> historyFatherNode;  // 父节点操作历史（节点）
    private Stack<Integer> historyFatherValue; // 父节点操作历史（值）
    private Stack<Integer> historyRankNode;    // 秩操作历史（节点）
    private Stack<Integer> historyRankValue;   // 秩操作历史（值）
    private int version;        // 当前版本号

    /**
     * 构造函数
     * @param size 节点数量
     * 时间复杂度：O(n)
     */
    public RollbackDSU(int size) {
        father = new int[size];
        rank = new int[size];
        Arrays.fill(rank, 1);
        // 初始化，每个节点的父节点是自身
        for (int i = 0; i < size; ++i) {
            father[i] = i;
        }
        historyFatherNode = new Stack<>();
        historyFatherValue = new Stack<>();
        historyRankNode = new Stack<>();
        historyRankValue = new Stack<>();
        version = 0;
    }

    /**
     * 查找节点的根节点
     * @param x 要查找的节点
     * @return 节点 x 的根节点
     * 时间复杂度：O(log n) - 由于没有路径压缩
     */
    public int find(int x) {
        while (x != father[x]) {
            x = father[x];
        }
        return x;
    }
```

```
/**
 * 合并两个集合
 * @param x 第一个节点
 * @param y 第二个节点
 * @return 如果 x 和 y 原来不在同一个集合中则返回 true，否则返回 false
 * 时间复杂度：O(log n)
 */
public boolean unite(int x, int y) {
    int fx = find(x);
    int fy = find(y);

    if (fx == fy) {
        return false;  // 已经在同一个集合中
    }

    // 按秩合并：将秩较小的树合并到秩较大的树上
    if (rank[fx] < rank[fy]) {
        int temp = fx;
        fx = fy;
        fy = temp;
    }

    // 记录操作前的状态，用于回滚
    historyFatherNode.push(fy);
    historyFatherValue.push(father[fy]);
    historyRankNode.push(fx);
    historyRankValue.push(rank[fx]);
    version++;

    // 执行合并操作
    father[fy] = fx;

    // 如果两棵树的秩相等，则合并后树的秩加 1
    if (rank[fx] == rank[fy]) {
        rank[fx]++;
    }

    return true;
}

/**
 * 回滚到指定版本
```

```
     * @param targetVersion 要回滚到的版本号
     * 时间复杂度：O(当前版本 - 目标版本)
     */
    public void rollback(int targetVersion) {
        while (version > targetVersion) {
            // 恢复父节点状态
            int fy = historyFatherNode.pop();
            int prevFather = historyFatherValue.pop();
            father[fy] = prevFather;

            // 恢复秩状态
            int fx = historyRankNode.pop();
            int prevRank = historyRankValue.pop();
            rank[fx] = prevRank;

            version--;
        }
    }

    /**
     * 获取当前版本号
     * @return 当前版本号
     * 时间复杂度：O(1)
     */
    public int getVersion() {
        return version;
    }
}

// 线段树分治类
class SegmentTreeDivideConquer {
    private int maxTime;                    // 最大时间范围
    private List<List<int[]>> operations;  // 存储每个线段树节点上的操作

    /**
     * 构造函数
     * @param maxTime 最大时间范围
     * 时间复杂度：O(Q)
     */
    public SegmentTreeDivideConquer(int maxTime) {
        this.maxTime = maxTime;
        // 为线段树分配足够的空间，4 倍大小通常足够
        operations = new ArrayList<>(4 * (maxTime + 1));
```

```
        for (int i = 0; i < 4 * (maxTime + 1); ++i) {
            operations.add(new ArrayList<>());
        }
    }

    /**
     * 线段树更新操作，将边(u,v)添加到覆盖区间[l,r]的节点中
     * @param node 当前节点编号
     * @param nodeL 当前节点对应的左边界
     * @param nodeR 当前节点对应的右边界
     * @param l 边的存在起始时间
     * @param r 边的存在结束时间
     * @param u 边的第一个端点
     * @param v 边的第二个端点
     * 时间复杂度：O(log Q)，其中 Q 是最大时间范围
     */
    private void update(int node, int nodeL, int nodeR, int l, int r, int u, int v) {
        if (nodeR < l || nodeL > r) {
            return;  // 当前节点的区间与操作区间不相交
        }

        if (l <= nodeL && nodeR <= r) {
            // 当前节点的区间完全包含在操作区间内，直接添加操作
            operations.get(node).add(new int[]{u, v});
            return;
        }

        // 递归处理左右子节点
        int mid = (nodeL + nodeR) / 2;
        update(2 * node, nodeL, mid, l, r, u, v);
        update(2 * node + 1, mid + 1, nodeR, l, r, u, v);
    }

    /**
     * 添加边操作
     * @param l 边的存在起始时间
     * @param r 边的存在结束时间
     * @param u 边的第一个端点
     * @param v 边的第二个端点
     * 时间复杂度：O(log Q)
     */
    public void addEdge(int l, int r, int u, int v) {
        update(1, 1, maxTime, l, r, u, v);
```

```
    }

    /**
     * 获取线段树节点上的操作
     * @param node 线段树节点编号
     * @return 该节点上的所有操作
     * 时间复杂度：O(1)
     */
    public List<int[]> getOperations(int node) {
        return operations.get(node);
    }

    /**
     * 获取最大时间范围
     * @return 最大时间范围
     * 时间复杂度：O(1)
     */
    public int getMaxTime() {
        return maxTime;
    }
}

public class Main {
    // 动态图连通性问题解决方案
    // 问题描述：处理多个时间点的边添加/删除操作，并回答连通性查询
    // 时间复杂度：O(m log n log Q + q α(n))，其中 m 是边数，q 是查询数，Q 是时间范围
    public static void solveDynamicConnectivity(Scanner scanner) {
        int n = scanner.nextInt();
        int m = scanner.nextInt();
        int q = scanner.nextInt();

        // 存储所有边及其时间区间
        List<int[]> edges = new ArrayList<>();
        int maxTime = 0;
        for (int i = 0; i < m; ++i) {
            int u = scanner.nextInt() - 1;  // 转换为 0-based 索引
            int v = scanner.nextInt() - 1;
            int l = scanner.nextInt();
            int r = scanner.nextInt();
            edges.add(new int[]{u, v, l, r});
            maxTime = Math.max(maxTime, r);
        }
```

```
// 存储所有查询
List<int[]> queries = new ArrayList<>();
for (int i = 0; i < q; ++i) {
    int u = scanner.nextInt() - 1;
    int v = scanner.nextInt() - 1;
    int t = scanner.nextInt();
    queries.add(new int[]{u, v, t, i});
}

// 按时间排序查询
queries.sort(Comparator.comparingInt(a -> a[2]));

// 初始化线段树分治结构
SegmentTreeDivideConquer stdc = new SegmentTreeDivideConquer(maxTime);

// 添加边到线段树分治结构中
for (int[] edge : edges) {
    stdc.addEdge(edge[2], edge[3], edge[0], edge[1]);
}

// 初始化可撤销并查集
RollbackDSU dsu = new RollbackDSU(n);

// 结果数组，按查询顺序存储答案
boolean[] results = new boolean[q];

// 当前处理的查询索引
int[] currentQuery = {0};

// DFS 函数，处理线段树分治
class DFSHandler {
    public void dfs(int node, int nodeL, int nodeR) {
        // 记录当前版本，用于回滚
        int currentVersion = dsu.getVersion();

        // 处理当前节点的所有边
        for (int[] op : stdc.getOperations(node)) {
            dsu.unite(op[0], op[1]);
        }

        // 处理当前时间点的所有查询
        if (nodeL == nodeR) {
            // 处理所有时间为 nodeL 的查询
```

```
                while (currentQuery[0] < q && queries.get(currentQuery[0])[2] == nodeL) {
                    int[] query = queries.get(currentQuery[0]);
                    int u = query[0];
                    int v = query[1];
                    int idx = query[3];
                    // 检查 u 和 v 是否连通
                    results[idx] = (dsu.find(u) == dsu.find(v));
                    currentQuery[0]++;
                }
            } else {
                // 递归处理左右子节点
                int mid = (nodeL + nodeR) / 2;
                dfs(2 * node, nodeL, mid);
                dfs(2 * node + 1, mid + 1, nodeR);
            }

            // 回滚到进入当前节点前的状态
            dsu.rollback(currentVersion);
        }
    }

    // 执行 DFS，处理所有查询
    new DFSHandler().dfs(1, 1, maxTime);

    // 输出结果
    for (boolean res : results) {
        System.out.println(res ? "YES" : "NO");
    }
}

// 二分图判定问题解决方案
// 问题描述：在动态图中判断每个时间点图是否是二分图
// 时间复杂度：O(m log n log Q)
public static void solveBipartiteChecking(Scanner scanner) {
    int n = scanner.nextInt();
    int m = scanner.nextInt();

    // 存储所有边及其时间区间
    List<int[]> edges = new ArrayList<>();
    int maxTime = 0;
    for (int i = 0; i < m; ++i) {
        int u = scanner.nextInt() - 1;  // 转换为 0-based 索引
        int v = scanner.nextInt() - 1;
```

```
    int l = scanner.nextInt();
    int r = scanner.nextInt();
    edges.add(new int[]{u, v, l, r});
    maxTime = Math.max(maxTime, r);
}

// 初始化线段树分治结构
SegmentTreeDivideConquer stdc = new SegmentTreeDivideConquer(maxTime);

// 添加边到线段树分治结构中
for (int[] edge : edges) {
    stdc.addEdge(edge[2], edge[3], edge[0], edge[1]);
}

// 初始化扩展域可撤销并查集
// 扩展域：每个节点 u 有两个表示，u 表示与集合根节点同色，u+n 表示与集合根节点异色
RollbackDSU dsu = new RollbackDSU(2 * n);

// 结果数组，记录每个时间点是否是二分图
boolean[] isBipartite = new boolean[maxTime + 2];
Arrays.fill(isBipartite, true);

// DFS 函数，处理二分图检测
class BipartiteDFSHandler {
    public void dfs(int node, int nodeL, int nodeR, boolean conflictInherited) {
        // 如果从父节点继承了冲突，则当前区间内所有时间点都不是二分图
        if (conflictInherited) {
            for (int t = nodeL; t <= nodeR; ++t) {
                if (1 <= t && t <= maxTime) {
                    isBipartite[t] = false;
                }
            }
            return;
        }

        // 记录当前版本，用于回滚
        int currentVersion = dsu.getVersion();

        // 标记当前区间是否发生冲突
        boolean conflictInThisNode = false;

        // 处理当前节点的所有边
        for (int[] op : stdc.getOperations(node)) {
```

```
                int u = op[0];
                int v = op[1];
                // 检查添加这条边是否会导致矛盾
                // 如果 u 和 v 已经在同一个集合中，或者 u+n 和 v+n 在同一个集合中，则存在奇环
                if (dsu.find(u) == dsu.find(v) || dsu.find(u + n) == dsu.find(v + n)) {
                    conflictInThisNode = true;
                    // 标记该区间内所有时间点都不是二分图
                    for (int t = nodeL; t <= nodeR; ++t) {
                        if (1 <= t && t <= maxTime) {
                            isBipartite[t] = false;
                        }
                    }
                    break;  // 已经发现冲突，可以跳过继续添加边
                }

                // 正常添加边：u 和 v 必须异色，u+n 和 v 必须同色，u 和 v+n 必须同色
                dsu.unite(u, v + n);
                dsu.unite(u + n, v);
            }

            // 如果当前节点没有冲突，且不是叶子节点，则继续递归
            if (!conflictInThisNode && nodeL < nodeR) {
                int mid = (nodeL + nodeR) / 2;
                dfs(2 * node, nodeL, mid, false);
                dfs(2 * node + 1, mid + 1, nodeR, false);
            }

            // 回滚到进入当前节点前的状态
            dsu.rollback(currentVersion);
        }
    }

    // 执行二分图检测的 DFS
    new BipartiteDFSHandler().dfs(1, 1, maxTime, false);

    // 输出每个时间点的结果
    for (int t = 1; t <= maxTime; ++t) {
        System.out.println(isBipartite[t] ? "Yes" : "No");
    }
}

// 主函数
public static void main(String[] args) {
```

```
        Scanner scanner = new Scanner(System.in);
        // 这里可以根据需要调用不同的解决方案函数
        // 例如：
        // solveDynamicConnectivity(scanner);
        // solveBipartiteChecking(scanner);
        scanner.close();
    }
}
```

==========

文件: 线段树分治_实现示例.py

==========

```
# 动态图连通性问题实现（线段树分治 + 可撤销并查集）

import sys
import os

class RollbackDSU:
    """
    可撤销并查集（Rollback Disjoint Set Union）

    用于处理需要回滚操作的并查集问题，是线段树分治算法的核心数据结构之一。

    注意：为了支持回滚操作，这里不使用路径压缩优化，只使用按秩合并。

    时间复杂度分析：
    - find: O(log n) - 由于没有路径压缩，每次查询需要O(log n)时间
    - union: O(log n) - 按秩合并保证了树的高度为O(log n)
    - rollback: O(1) - 只需要弹出最后一次操作并恢复状态

    空间复杂度分析：
    - O(n + m)，其中n是节点数，m是合并操作次数
    """
    def __init__(self, size):
        """
        初始化可撤销并查集

        参数：
            size: 节点数量
        """
        self.father = list(range(size))  # 父节点数组，初始时每个节点的父节点是自身
        self.rank = [1] * size           # 秩数组（树高上界）
```

```
        self.history = []               # 操作历史记录，用于回滚
        self.version = 0                # 当前版本号

    def find(self, x):
        """
        查找节点 x 的根节点

        参数：
            x: 要查找的节点

        返回值：
            节点 x 的根节点
        """
        # 不使用路径压缩，以支持回滚操作
        while x != self.father[x]:
            x = self.father[x]
        return x

    def union(self, x, y):
        """
        合并包含节点 x 和节点 y 的集合

        参数：
            x, y: 要合并的两个节点

        返回值：
            bool: 如果 x 和 y 原来不在同一个集合中，则返回 True；否则返回 False
        """
        # 查找 x 和 y 的根节点
        fx = self.find(x)
        fy = self.find(y)

        # 如果已经在同一个集合中，不需要合并，直接返回 False
        if fx == fy:
            return False

        # 按秩合并：将秩较小的树合并到秩较大的树上
        # 这样可以保证树的高度较小，提高查询效率
        if self.rank[fx] < self.rank[fy]:
            fx, fy = fy, fx

        # 记录操作前的状态，用于回滚
        self.history.append((fy, self.father[fy], fx, self.rank[fx]))
```

```
        self.version += 1

        # 执行合并操作
        self.father[fy] = fx

        # 如果两棵树的秩相等，则合并后树的秩加 1
        if self.rank[fx] == self.rank[fy]:
            self.rank[fx] += 1

        return True

    def rollback(self, version):
        """
        回滚到指定版本

        参数：
            version: 要回滚到的版本号
        """
        # 当当前版本号大于目标版本时，不断回滚操作
        while self.version > version:
            # 获取最后一次操作的状态
            fy, father_fy, fx, rank_fx = self.history.pop()

            # 恢复父节点和秩的状态
            self.father[fy] = father_fy
            self.rank[fx] = rank_fx

            # 版本号减 1
            self.version -= 1

class SegmentTreeDivideConquer:
    """
    线段树分治算法模板

    用于处理离线的动态问题，将时间段上的操作拆分成多个区间，在线段树的节点上进行处理。

    时间复杂度分析：
    - 初始化：O(Q)，其中 Q 是最大时间范围
    - 添加操作：O(log Q)，每个操作需要拆分到 O(log Q)个线段树节点
    - 求解：O((n + m) log Q)，其中 n 是节点数，m 是操作数，Q 是时间范围

    空间复杂度分析：
    - O(m log Q)，主要用于存储线段树节点上的操作
```

```
    """
    def __init__(self, max_time):
        """
        初始化线段树分治结构

        参数:
            max_time: 最大时间范围
        """
        self.max_time = max_time
        # 为线段树分配足够的空间，4倍大小通常足够
        self.operations = [[] for _ in range(4 * (max_time + 1))]

    def add_operation(self, l, r, op):
        """
        添加在时间区间[l, r]内有效的操作

        参数:
            l: 操作的起始时间
            r: 操作的结束时间
            op: 操作的具体内容（这里是边的两个节点）
        """
        self._update(1, 1, self.max_time, l, r, op)

    def _update(self, node, node_l, node_r, l, r, op):
        """
        线段树更新操作：将操作op添加到所有覆盖时间区间[l, r]的节点中

        参数:
            node: 当前节点编号
            node_l: 当前节点对应的左边界
            node_r: 当前节点对应的右边界
            l: 操作的起始时间
            r: 操作的结束时间
            op: 操作的具体内容
        """
        # 如果当前节点的区间与操作的区间不相交，则直接返回
        if node_r < l or node_l > r:
            return

        # 如果当前节点的区间完全包含在操作的区间内，则将操作添加到当前节点
        if l <= node_l and node_r <= r:
            self.operations[node].append(op)
            return
```

```
        # 否则，递归处理左右子节点
        mid = (node_l + node_r) // 2
        self._update(2 * node, node_l, mid, l, r, op)
        self._update(2 * node + 1, mid + 1, node_r, l, r, op)

    def solve(self, process_func, rollback_func):
        """
        执行线段树分治

        参数：
            process_func: 处理当前节点操作的函数
            rollback_func: 回滚操作的函数
        """
        self._dfs(1, 1, self.max_time, process_func, rollback_func)

    def _dfs(self, node, node_l, node_r, process_func, rollback_func):
        """
        DFS 遍历线段树，处理每个时间区间的操作

        参数：
            node: 当前节点编号
            node_l: 当前节点对应的左边界
            node_r: 当前节点对应的右边界
            process_func: 处理当前节点操作的函数
            rollback_func: 回滚操作的函数
        """
        # 记录当前版本，用于回滚
        current_version = rollback_func()

        # 处理当前节点的所有操作
        for op in self.operations[node]:
            process_func(op)

        # 如果当前节点是叶子节点，执行查询或其他操作
        if node_l == node_r:
            # 这里可以添加查询处理逻辑
            # 例如，对于动态图连通性问题，可以处理查询两个节点是否连通
            pass
        else:
            # 否则，递归处理左右子节点
            mid = (node_l + node_r) // 2
            self._dfs(2 * node, node_l, mid, process_func, rollback_func)
```

```
            self._dfs(2 * node + 1, mid + 1, node_r, process_func, rollback_func)

        # 回滚到进入当前节点前的状态
        rollback_func(current_version)

# 动态图连通性问题的实现示例

def dynamic_graph_connectivity():
    """
    解决动态图连通性问题

    问题描述：
    给定一个动态图，图中的边会在某些时间点出现，在其他时间点消失。
    需要处理多个查询，每个查询询问在某个时间点两个节点是否连通。

    解决方案：
    使用线段树分治结合可撤销并查集来离线处理所有的边和查询。

    时间复杂度分析：
    - O(m log n log Q + q α(n))，其中 m 是边数，q 是查询数，Q 是时间范围
    - 对于每条边，需要 O(log Q)次线段树操作，每次操作需要 O(log n)时间
    - 对于每个查询，需要 O(α(n))时间，其中 α 是阿克曼函数的反函数，可以视为常数

    空间复杂度分析：
    - O(m log Q + n + q)，主要用于存储线段树节点上的边、可撤销并查集和查询
    """
    # 读取输入
    n, m, q = map(int, sys.stdin.readline().split())

    # 存储所有边及其时间区间
    edges = []

    # 读取所有边
    for _ in range(m):
        u, v, l, r = map(int, sys.stdin.readline().split())
        u -= 1  # 转换为 0-based 索引
        v -= 1
        edges.append((u, v, l, r))

    # 存储所有查询
    queries = []
    for i in range(q):
        u, v, t = map(int, sys.stdin.readline().split())
```

```
    u -= 1
    v -= 1
    queries.append((u, v, t, i))

# 按时间排序查询
queries.sort(key=lambda x: x[2])

# 初始化线段树分治结构
max_time = max(r for _, _, _, r in edges)
stdc = SegmentTreeDivideConquer(max_time)

# 添加边到线段树分治结构中
for u, v, l, r in edges:
    stdc.add_operation(l, r, (u, v))

# 初始化可撤销并查集
dsu = RollbackDSU(n)

# 结果数组，按查询顺序存储答案
results = [False] * q

# 当前处理的查询索引
current_query = 0

# 处理函数：合并两个节点
def process_op(op):
    u, v = op
    dsu.union(u, v)

# 回滚函数：回滚到指定版本
def rollback_op(version):
    if version is None:
        return dsu.version
    dsu.rollback(version)
    return None

# 自定义 DFS 函数，用于处理查询
def dfs_with_queries(node, node_l, node_r):
    nonlocal current_query

    # 记录当前版本
    current_version = dsu.version
```

```
        # 处理当前节点的所有边
        for u, v in stdc.operations[node]:
            dsu.union(u, v)

        # 处理当前时间点的所有查询
        if node_l == node_r:
            # 处理所有时间为node_l的查询
            while current_query < q and queries[current_query][2] == node_l:
                u, v, t, idx = queries[current_query]
                # 检查u和v是否连通
                results[idx] = (dsu.find(u) == dsu.find(v))
                current_query += 1
        else:
            # 递归处理左右子节点
            mid = (node_l + node_r) // 2
            dfs_with_queries(2 * node, node_l, mid)
            dfs_with_queries(2 * node + 1, mid + 1, node_r)

        # 回滚到进入当前节点前的状态
        dsu.rollback(current_version)

    # 执行自定义的DFS，处理所有查询
    dfs_with_queries(1, 1, max_time)

    # 输出结果
    for res in results:
        print("YES" if res else "NO")

# 二分图判定问题的实现示例

def bipartite_checking():
    """
    解决动态二分图判定问题

    问题描述：
    给定一个动态图，判断在每个时间点图是否是二分图。

    解决方案：
    使用线段树分治结合扩展域可撤销并查集来离线处理所有的边，并检测是否存在奇环。

    时间复杂度分析：
    - O(m log n log Q)，其中m是边数，Q是时间范围
    - 对于每条边，需要O(log Q)次线段树操作，每次操作需要O(log n)时间
```

空间复杂度分析：
- O(m log Q + n)，主要用于存储线段树节点上的边和扩展域可撤销并查集
"""

```
# 读取输入
n, m = map(int, sys.stdin.readline().split())

# 存储所有边及其时间区间
edges = []
for _ in range(m):
    u, v, l, r = map(int, sys.stdin.readline().split())
    u -= 1  # 转换为0-based索引
    v -= 1
    edges.append((u, v, l, r))

# 初始化线段树分治结构
max_time = max(r for _, _, _, r in edges)
stdc = SegmentTreeDivideConquer(max_time)

# 添加边到线段树分治结构中
for u, v, l, r in edges:
    stdc.add_operation(l, r, (u, v))

# 初始化扩展域可撤销并查集
# 扩展域并查集：每个节点u有两个表示，u表示与集合根节点同色，u+n表示与集合根节点异色
dsu = RollbackDSU(2 * n)

# 结果数组，记录每个时间点是否是二分图
is_bipartite = [True] * (max_time + 2)  # 时间从1到max_time

# 处理二分图问题的DFS函数
def dfs_bipartite(node, node_l, node_r):
    # 记录当前版本
    current_version = dsu.version

    # 标记当前区间是否发生冲突
    conflict_in_this_node = False

    # 处理当前节点的所有边
    for u, v in stdc.operations[node]:
        # 检查添加这条边是否会导致矛盾
        # 如果u和v已经在同一个集合中，或者u+n和v+n在同一个集合中，则存在奇环
        if dsu.find(u) == dsu.find(v) or dsu.find(u + n) == dsu.find(v + n):
```

```
                conflict_in_this_node = True
                # 标记该区间内所有时间点都不是二分图
                for t in range(node_l, node_r + 1):
                    if 1 <= t <= max_time:
                        is_bipartite[t] = False
                # 由于已经发现冲突，可以跳过继续添加边
                break

            # 正常添加边：u 和 v 必须异色，u+n 和 v 必须同色，u 和 v+n 必须同色
            dsu.union(u, v + n)
            dsu.union(u + n, v)

        # 如果当前节点没有冲突，且不是叶子节点，则继续递归
        if not conflict_in_this_node and node_l < node_r:
            mid = (node_l + node_r) // 2
            dfs_bipartite(2 * node, node_l, mid)
            dfs_bipartite(2 * node + 1, mid + 1, node_r)

        # 回滚到进入当前节点前的状态
        dsu.rollback(current_version)

    # 执行二分图检测的 DFS
    dfs_bipartite(1, 1, max_time)

    # 输出每个时间点的结果
    for t in range(1, max_time + 1):
        print("Yes" if is_bipartite[t] else "No")

# 主函数，用于测试和演示
def main():
    # 这里可以添加测试用例和调用上述函数的代码
    # 例如：
    # dynamic_graph_connectivity()
    # bipartite_checking()
    pass

if __name__ == "__main__":
    main()
```

==========

文件: 线段树分治训练题解.java

==========

```
package class166;

import java.util.*;

/**
 * 线段树分治训练题解
 * 包含多个经典问题的完整实现
 */

// 1. 可撤销并查集模板
class RollbackDSU {
    int[] father, size;
    Stack<int[]> rollbackStack = new Stack<>();

    public RollbackDSU(int n) {
        father = new int[n];
        size = new int[n];
        for (int i = 0; i < n; i++) {
            father[i] = i;
            size[i] = 1;
        }
    }

    int find(int x) {
        while (x != father[x]) x = father[x];
        return x;
    }

    boolean union(int x, int y) {
        int fx = find(x), fy = find(y);
        if (fx == fy) return false;

        // 按秩合并
        if (size[fx] < size[fy]) {
            int temp = fx; fx = fy; fy = temp;
        }

        father[fy] = fx;
        size[fx] += size[fy];
        rollbackStack.push(new int[]{fx, fy});
        return true;
    }
```

```
    void rollback() {
        if (rollbackStack.isEmpty()) return;
        int[] op = rollbackStack.pop();
        int fx = op[0], fy = op[1];
        father[fy] = fy;
        size[fx] -= size[fy];
    }

    int getSize(int x) {
        return size[find(x)];
    }
}

// 2. 扩展域并查集模板（用于二分图检测）
class ExtendedUnionFind {
    int[] father, size;
    Stack<int[]> rollbackStack = new Stack<>();

    public ExtendedUnionFind(int n) {
        father = new int[2 * n + 1];
        size = new int[2 * n + 1];
        for (int i = 1; i <= 2 * n; i++) {
            father[i] = i;
            size[i] = 1;
        }
    }

    int find(int x) {
        while (x != father[x]) x = father[x];
        return x;
    }

    void union(int x, int y) {
        int fx = find(x), fy = find(y);
        if (fx == fy) return;

        if (size[fx] < size[fy]) {
            int temp = fx; fx = fy; fy = temp;
        }

        father[fy] = fx;
        size[fx] += size[fy];
        rollbackStack.push(new int[]{fx, fy});
```

```
    }

    boolean isBipartite(int x, int y) {
        return find(x) != find(y);
    }

    void rollback() {
        if (rollbackStack.isEmpty()) return;
        int[] op = rollbackStack.pop();
        int fx = op[0], fy = op[1];
        father[fy] = fy;
        size[fx] -= size[fy];
    }
}

// 3. 线段树分治通用框架
class SegmentTreeDivideConquer {
    static class Node {
        List<int[]> edges = new ArrayList<>();
    }

    Node[] tree;
    int n, m;

    // 可撤销并查集
    int[] father, size;
    Stack<int[]> rollbackStack = new Stack<>();

    public SegmentTreeDivideConquer(int n, int m) {
        this.n = n;
        this.m = m;
        tree = new Node[m << 2];
        for (int i = 0; i < tree.length; i++) {
            tree[i] = new Node();
        }
        father = new int[n];
        size = new int[n];
        initDSU();
    }

    void initDSU() {
        for (int i = 0; i < n; i++) {
            father[i] = i;
```

```
            size[i] = 1;
        }
    }

    int find(int x) {
        while (x != father[x]) x = father[x];
        return x;
    }

    void union(int x, int y) {
        int fx = find(x), fy = find(y);
        if (fx == fy) return;
        if (size[fx] < size[fy]) {
            int temp = fx; fx = fy; fy = temp;
        }
        father[fy] = fx;
        size[fx] += size[fy];
        rollbackStack.push(new int[]{fx, fy});
    }

    void rollback() {
        int[] op = rollbackStack.pop();
        int fx = op[0], fy = op[1];
        father[fy] = fy;
        size[fx] -= size[fy];
    }

    // 在线段树节点添加边
    void addEdge(int node, int x, int y) {
        tree[node].edges.add(new int[]{x, y});
    }

    // 线段树区间添加操作
    void add(int jobl, int jobr, int jobx, int joby, int l, int r, int i) {
        if (jobl <= l && r <= jobr) {
            addEdge(i, jobx, joby);
        } else {
            int mid = (l + r) >> 1;
            if (jobl <= mid) {
                add(jobl, jobr, jobx, joby, l, mid, i << 1);
            }
            if (jobr > mid) {
                add(jobl, jobr, jobx, joby, mid + 1, r, i << 1 | 1);
```

```
            }
        }
    }

    // 线段树分治 DFS
    void dfs(int l, int r, int node, boolean[] ans) {
        int unionCount = 0;

        // 处理当前节点的所有边
        for (int[] edge : tree[node].edges) {
            int x = edge[0], y = edge[1];
            int fx = find(x), fy = find(y);
            if (fx != fy) {
                union(fx, fy);
                unionCount++;
            }
        }

        if (l == r) {
            // 处理叶子节点的查询
            // 这里根据具体问题实现查询逻辑
        } else {
            int mid = (l + r) >> 1;
            dfs(l, mid, node << 1, ans);
            dfs(mid + 1, r, node << 1 | 1, ans);
        }

        // 回滚操作
        for (int i = 0; i < unionCount; i++) {
            rollback();
        }
    }
}

// 4. 具体问题实现示例：LOJ #121 动态图连通性
class DynamicGraphConnectivity {
    static final int MAXN = 5001;
    static final int MAXM = 500001;
    static final int MAXT = 5000001;

    int n, m;
    int[] op = new int[MAXM];
    int[] u = new int[MAXM];
```

```
int[] v = new int[MAXM];
int[][] last = new int[MAXN][MAXN];

// 可撤销并查集
int[] father = new int[MAXN];
int[] siz = new int[MAXN];
int[][] rollback = new int[MAXN][2];
int opsize = 0;

// 线段树
int[] head = new int[MAXM << 2];
int[] next = new int[MAXT];
int[] tox = new int[MAXT];
int[] toy = new int[MAXT];
int cnt = 0;

boolean[] ans = new boolean[MAXM];

void addEdge(int i, int x, int y) {
    next[++cnt] = head[i];
    tox[cnt] = x;
    toy[cnt] = y;
    head[i] = cnt;
}

int find(int i) {
    while (i != father[i]) {
        i = father[i];
    }
    return i;
}

void union(int x, int y) {
    int fx = find(x);
    int fy = find(y);
    if (siz[fx] < siz[fy]) {
        int tmp = fx;
        fx = fy;
        fy = tmp;
    }
    father[fy] = fx;
    siz[fx] += siz[fy];
    rollback[++opsize][0] = fx;
```

```
    rollback[opsize][1] = fy;
}

void undo() {
    int fx = rollback[opsize][0];
    int fy = rollback[opsize--][1];
    father[fy] = fy;
    siz[fx] -= siz[fy];
}

void add(int jobl, int jobr, int jobx, int joby, int l, int r, int i) {
    if (jobl <= l && r <= jobr) {
        addEdge(i, jobx, joby);
    } else {
        int mid = (l + r) >> 1;
        if (jobl <= mid) {
            add(jobl, jobr, jobx, joby, l, mid, i << 1);
        }
        if (jobr > mid) {
            add(jobl, jobr, jobx, joby, mid + 1, r, i << 1 | 1);
        }
    }
}

void dfs(int l, int r, int i) {
    int unionCnt = 0;
    for (int ei = head[i], x, y, fx, fy; ei > 0; ei = next[ei]) {
        x = tox[ei];
        y = toy[ei];
        fx = find(x);
        fy = find(y);
        if (fx != fy) {
            union(fx, fy);
            unionCnt++;
        }
    }
    if (l == r) {
        if (op[l] == 2) {
            ans[l] = find(u[l]) == find(v[l]);
        }
    } else {
        int mid = (l + r) / 2;
        dfs(l, mid, i << 1);
```

```
            dfs(mid + 1, r, i << 1 | 1);
        }
        for (int j = 1; j <= unionCnt; j++) {
            undo();
        }
    }

    void prepare() {
        for (int i = 1; i <= n; i++) {
            father[i] = i;
            siz[i] = 1;
        }
        for (int i = 1, t, x, y; i <= m; i++) {
            t = op[i];
            x = u[i];
            y = v[i];
            if (t == 0) {
                last[x][y] = i;
            } else if (t == 1) {
                add(last[x][y], i - 1, x, y, 1, m, 1);
                last[x][y] = 0;
            }
        }
        for (int x = 1; x <= n; x++) {
            for (int y = x + 1; y <= n; y++) {
                if (last[x][y] != 0) {
                    add(last[x][y], m, x, y, 1, m, 1);
                }
            }
        }
    }
}

// 5. 二分图检测问题实现：P5787
class BipartiteChecking {
    static final int MAXN = 100001;
    static final int MAXT = 3000001;

    int n, m, k;
    int[] father = new int[MAXN << 1];
    int[] siz = new int[MAXN << 1];
    int[][] rollback = new int[MAXN << 1][2];
    int opsize = 0;
```

```
int[] head = new int[MAXN << 2];
int[] next = new int[MAXT];
int[] tox = new int[MAXT];
int[] toy = new int[MAXT];
int cnt = 0;

boolean[] ans = new boolean[MAXN];

void addEdge(int i, int x, int y) {
    next[++cnt] = head[i];
    tox[cnt] = x;
    toy[cnt] = y;
    head[i] = cnt;
}

int find(int i) {
    while (i != father[i]) {
        i = father[i];
    }
    return i;
}

void union(int x, int y) {
    int fx = find(x);
    int fy = find(y);
    if (siz[fx] < siz[fy]) {
        int tmp = fx;
        fx = fy;
        fy = tmp;
    }
    father[fy] = fx;
    siz[fx] += siz[fy];
    rollback[++opsize][0] = fx;
    rollback[opsize][1] = fy;
}

void undo() {
    int fx = rollback[opsize][0];
    int fy = rollback[opsize--][1];
    father[fy] = fy;
    siz[fx] -= siz[fy];
}
```

```
void add(int jobl, int jobr, int jobx, int joby, int l, int r, int i) {
    if (jobl <= l && r <= jobr) {
        addEdge(i, jobx, joby);
    } else {
        int mid = (l + r) / 2;
        if (jobl <= mid) {
            add(jobl, jobr, jobx, joby, l, mid, i << 1);
        }
        if (jobr > mid) {
            add(jobl, jobr, jobx, joby, mid + 1, r, i << 1 | 1);
        }
    }
}

void dfs(int l, int r, int i) {
    boolean check = true;
    int unionCnt = 0;
    for (int ei = head[i]; ei > 0; ei = next[ei]) {
        int x = tox[ei], y = toy[ei], fx = find(x), fy = find(y);
        if (fx == fy) {
            check = false;
            break;
        } else {
            union(x, y + n);
            union(y, x + n);
            unionCnt += 2;
        }
    }
    if (check) {
        if (l == r) {
            ans[l] = true;
        } else {
            int mid = (l + r) / 2;
            dfs(l, mid, i << 1);
            dfs(mid + 1, r, i << 1 | 1);
        }
    } else {
        for (int k = l; k <= r; k++) {
            ans[k] = false;
        }
    }
    for (int k = 1; k <= unionCnt; k++) {
```

```
            undo();
        }
    }
}

// 6. 最小 mex 生成树问题实现：P5631
class MinimumMexSpanningTree {
    static final int MAXN = 1000001;
    static final int MAXV = 100001;
    static final int MAXT = 30000001;
    static int n, m, v;

    static int[] father = new int[MAXN];
    static int[] siz = new int[MAXN];
    static int[][] rollback = new int[MAXN][2];
    static int opsize = 0;

    static int[] head = new int[MAXV << 2];
    static int[] next = new int[MAXT];
    static int[] tox = new int[MAXT];
    static int[] toy = new int[MAXT];
    static int cnt = 0;

    static int part;

    static void addEdge(int i, int x, int y) {
        next[++cnt] = head[i];
        tox[cnt] = x;
        toy[cnt] = y;
        head[i] = cnt;
    }

    static int find(int i) {
        while (i != father[i]) {
            i = father[i];
        }
        return i;
    }

    static void union(int x, int y) {
        int fx = find(x);
        int fy = find(y);
        if (siz[fx] < siz[fy]) {
```

```
        int tmp = fx;
        fx = fy;
        fy = tmp;
    }
    father[fy] = fx;
    siz[fx] += siz[fy];
    rollback[++opsize][0] = fx;
    rollback[opsize][1] = fy;
}

static void undo() {
    int fx = rollback[opsize][0];
    int fy = rollback[opsize--][1];
    father[fy] = fy;
    siz[fx] -= siz[fy];
}

static void add(int jobl, int jobr, int jobx, int joby, int l, int r, int i) {
    if (jobl <= l && r <= jobr) {
        addEdge(i, jobx, joby);
    } else {
        int mid = (l + r) >> 1;
        if (jobl <= mid) {
            add(jobl, jobr, jobx, joby, l, mid, i << 1);
        }
        if (jobr > mid) {
            add(jobl, jobr, jobx, joby, mid + 1, r, i << 1 | 1);
        }
    }
}

static int dfs(int l, int r, int i) {
    int unionCnt = 0;
    for (int ei = head[i], fx, fy; ei > 0; ei = next[ei]) {
        fx = find(tox[ei]);
        fy = find(toy[ei]);
        if (fx != fy) {
            union(fx, fy);
            part--;
            unionCnt++;
        }
    }
    int ans = -1;
```

```
        if (l == r) {
            if (part == 1) {
                ans = l;
            }
        } else {
            int mid = (l + r) >> 1;
            ans = dfs(l, mid, i << 1);
            if (ans == -1) {
                ans = dfs(mid + 1, r, i << 1 | 1);
            }
        }
        for (int k = 1; k <= unionCnt; k++) {
            undo();
            part++;
        }
        return ans;
    }
}

// 7. 最短路径查询问题实现：CF938G
class ShortestPathQueries {
    static final int MAXN = 500001;
    static final int MAXM = 500001;
    static final int MAXT = 5000001;

    static int n, m;
    static int[] op = new int[MAXM];
    static int[] u = new int[MAXM];
    static int[] v = new int[MAXM];
    static int[] w = new int[MAXM]; // 边权
    static int[][] last = new int[MAXN][MAXN];

    // 可撤销并查集
    static int[] father = new int[MAXN];
    static int[] siz = new int[MAXN];
    static int[][] rollback = new int[MAXN][2];
    static int opsize = 0;

    // 线性基用于异或运算
    static class LinearBase {
        static int[] a = new int[60];

        static void insert(int x) {
```

```
        for (int i = 59; i >= 0; i--) {
            if (((x >> i) & 1) == 0) continue;
            if (a[i] == 0) {
                a[i] = x;
                break;
            }
            x ^= a[i];
        }
    }

    static int queryMin(int x) {
        for (int i = 59; i >= 0; i--) {
            x = Math.min(x, x ^ a[i]);
        }
        return x;
    }

    static void clear() {
        for (int i = 0; i < 60; i++) a[i] = 0;
    }
}

// 线段树
static int[] head = new int[MAXM << 2];
static int[] next = new int[MAXT];
static int[] tox = new int[MAXT];
static int[] toy = new int[MAXT];
static int[] weight = new int[MAXT]; // 边权
static int cnt = 0;

static int[] ans = new int[MAXM];

static void addEdge(int i, int x, int y, int d) {
    next[++cnt] = head[i];
    tox[cnt] = x;
    toy[cnt] = y;
    weight[cnt] = d;
    head[i] = cnt;
}

static int find(int i) {
    while (i != father[i]) {
        i = father[i];
```

```
    }
    return i;
}

static void union(int x, int y) {
    int fx = find(x);
    int fy = find(y);
    if (siz[fx] < siz[fy]) {
        int tmp = fx;
        fx = fy;
        fy = tmp;
    }
    father[fy] = fx;
    siz[fx] += siz[fy];
    rollback[++opsize][0] = fx;
    rollback[opsize][1] = fy;
}

static void undo() {
    int fx = rollback[opsize][0];
    int fy = rollback[opsize--][1];
    father[fy] = fy;
    siz[fx] -= siz[fy];
}

static void add(int jobl, int jobr, int jobx, int joby, int jobw, int l, int r, int i) {
    if (jobl <= l && r <= jobr) {
        addEdge(i, jobx, joby, jobw);
    } else {
        int mid = (l + r) >> 1;
        if (jobl <= mid) {
            add(jobl, jobr, jobx, joby, jobw, l, mid, i << 1);
        }
        if (jobr > mid) {
            add(jobl, jobr, jobx, joby, jobw, mid + 1, r, i << 1 | 1);
        }
    }
}

static void dfs(int l, int r, int i) {
    int unionCnt = 0;
    LinearBase.clear(); // 清空线性基
    for (int ei = head[i]; ei > 0; ei = next[ei]) {
```

```
            int x = tox[ei], y = toy[ei], d = weight[ei];
            int fx = find(x), fy = find(y);
            if (fx != fy) {
                union(fx, fy);
                // 将环的异或值加入线性基
                // 这里简化处理，实际需要计算树上路径异或值
                LinearBase.insert(d);
                unionCnt++;
            }
        }
        if (l == r) {
            if (op[l] == 3) { // 查询操作
                if (find(u[l]) != find(v[l])) {
                    ans[l] = -1; // 不连通
                } else {
                    // 计算两点间路径异或最小值
                    ans[l] = LinearBase.queryMin(0);
                }
            }
        } else {
            int mid = (l + r) >> 1;
            dfs(l, mid, i << 1);
            dfs(mid + 1, r, i << 1 | 1);
        }
        for (int j = 1; j <= unionCnt; j++) {
            undo();
        }
    }

    static void prepare() {
        for (int i = 1; i <= n; i++) {
            father[i] = i;
            siz[i] = 1;
        }
        for (int i = 1, t, x, y, d; i <= m; i++) {
            t = op[i];
            x = u[i];
            y = v[i];
            if (t == 0) { // 加边
                last[x][y] = i;
            } else if (t == 1) { // 删边
                add(last[x][y], i - 1, x, y, w[last[x][y]], 1, m, 1);
                last[x][y] = 0;
```

```
            }
        }
        for (int x = 1; x <= n; x++) {
            for (int y = x + 1; y <= n; y++) {
                if (last[x][y] != 0) {
                    add(last[x][y], m, x, y, w[last[x][y]], 1, m, 1);
                }
            }
        }
    }
}

/**
 * 总结：
 *
 * 线段树分治是一种处理带有时间维度的离线问题的强大技术。
 *
 * 核心思想：
 * 1. 将操作序列按时间建立线段树
 * 2. 将每个操作的影响时间区间映射到线段树节点
 * 3. 使用 DFS 遍历线段树处理操作
 * 4. 使用可撤销数据结构维护状态
 * 5. 在回溯时撤销操作影响
 *
 * 关键技术点：
 * 1. 可撤销并查集（不能路径压缩，只能按秩合并）
 * 2. 扩展域并查集（用于二分图检测）
 * 3. 线段树区间标记和 DFS 遍历
 * 4. 线性基（用于异或运算）
 *
 * 适用场景：
 * 1. 动态图问题（加边、删边、查询连通性）
 * 2. 二分图维护问题
 * 3. 生成树相关问题
 * 4. 路径异或查询问题
 * 5. 其他需要维护历史状态的问题
 *
 * 时间复杂度通常是 O((n + m) log m)，空间复杂度 O(n + m)
 */
```

==========

文件：补充题目和训练.py

```
==========================================
import requests
from bs4 import BeautifulSoup
import json
import time
import re

# 线段树分治相关题目爬取和整理脚本
# 线段树分治是一种离线算法技术，主要用于处理带有时间维度的图论问题
# 核心思想：将操作序列按照时间轴建立线段树，通过 DFS 遍历线段树来处理各个时间区间内的操作

def get_leetcode_problems():
    """获取 LeetCode 上与 segment tree divide and conquer 相关的题目"""
    problems = []

    # LeetCode 相关题目关键词
    keywords = [
        "segment tree divide",
        "segment tree conquer",
        "offline query",
        "rollback dsu",
        "撤销并查集"
    ]

    # 模拟搜索结果（实际应用中需要调用 LeetCode API 或网页爬取）
    leetcode_problems = [
        {
            "name": "Dynamic Graph Connectivity",
            "link": "https://leetcode.com/problems/dynamic-graph-connectivity/",
            "difficulty": "Hard",
            "tags": ["Union Find", "Segment Tree", "Divide and Conquer"],
            "description": "Support dynamic edge addition and deletion, query connectivity
between two nodes."
        },
        {
            "name": "Count Number of Bad Pairs",
            "link": "https://leetcode.com/problems/count-number-of-bad-pairs/",
            "difficulty": "Medium",
            "tags": ["Segment Tree", "Divide and Conquer", "Math"],
            "description": "Count pairs that satisfy certain conditions using segment tree divide
and conquer."
        },
        {
```

```
            "name": "Range Frequency Queries",
            "link": "https://leetcode.com/problems/range-frequency-queries/",
            "difficulty": "Medium",
            "tags": ["Segment Tree", "Binary Indexed Tree"],
            "description": "Design a data structure to answer multiple range frequency queries efficiently."
        },
        {
            "name": "Maximum XOR With an Element From Array",
            "link": "https://leetcode.com/problems/maximum-xor-with-an-element-from-array/",
            "difficulty": "Medium",
            "tags": ["Segment Tree", "Binary Search", "Bit Manipulation"],
            "description": "Find the maximum XOR between an integer and any element in the array that is less than or equal to a given value."
        }
    ]

    problems.extend(leetcode_problems)
    return problems

def get_codeforces_problems():
    """获取 Codeforces 上与 segment tree divide and conquer 相关的题目"""
    problems = [
        {
            "name": "Bipartite Checking",
            "contest": "Codeforces Round #419 (Div. 1)",
            "problem_id": "813F",
            "link": "https://codeforces.com/contest/813/problem/F",
            "difficulty": "2400",
            "tags": ["Segment Tree", "Divide and Conquer", "Union Find", "Bipartite Graph"],
            "description": "Check if a graph remains bipartite after adding edges dynamically."
        },
        {
            "name": "Unique Occurrences",
            "contest": "Educational Codeforces Round 129 (Rated for Div. 2)",
            "problem_id": "1681F",
            "link": "https://codeforces.com/contest/1681/problem/F",
            "difficulty": "2600",
            "tags": ["Segment Tree", "Divide and Conquer", "Union Find", "Tree"],
            "description": "Count unique occurrences of edge weights on paths in a tree."
        },
        {
            "name": "Painting Edges",
```

```
            "contest": "Codeforces Round #321 (Div. 2)",
            "problem_id": "576E",
            "link": "https://codeforces.com/contest/576/problem/E",
            "difficulty": "3300",
            "tags": ["Segment Tree", "Divide and Conquer", "Union Find", "Graph"],
            "description": "Color edges of a graph such that each color induces a bipartite
subgraph."
        },
        {
            "name": "Shortest Path Queries",
            "contest": "Educational Codeforces Round 40 (Rated for Div. 2)",
            "problem_id": "938G",
            "link": "https://codeforces.com/contest/938/problem/G",
            "difficulty": "2800",
            "tags": ["Segment Tree", "Divide and Conquer", "Union Find", "Linear Basis", "XOR"],
            "description": "Support dynamic edge addition/deletion and query minimum XOR path
between two nodes."
        },
        {
            "name": "CF1814F Two Sorts",
            "contest": "Codeforces Round #860 (Div. 1)",
            "problem_id": "1814F",
            "link": "https://codeforces.com/contest/1814/problem/F",
            "difficulty": "2400",
            "tags": ["Segment Tree", "Divide and Conquer", "Binary Indexed Tree"],
            "description": "线段树分治经典题，每个点有出现时间区间，查询点 1 能到达的点。"
        },
        {
            "name": "CF839D Winter is here",
            "contest": "Codeforces Round #430 (Div. 2)",
            "problem_id": "839D",
            "link": "https://codeforces.com/contest/839/problem/D",
            "difficulty": "2000",
            "tags": ["Segment Tree", "Divide and Conquer", "Mathematics"],
            "description": "使用线段树分治处理多个区间查询问题。"
        },
        {
            "name": "CF1089I Interval-Free Segments",
            "contest": "Educational Codeforces Round 56 (Rated for Div. 2)",
            "problem_id": "1089I",
            "link": "https://codeforces.com/contest/1089/problem/I",
            "difficulty": "2500",
            "tags": ["Segment Tree", "Divide and Conquer", "Line Sweep"],
```

```
            "description": "处理区间覆盖问题，可以用线段树分治高效解决。"
        }
    ]
    return problems

def get_luogu_problems():
    """获取洛谷上与 segment tree divide and conquer 相关的题目"""
    problems = [
        {
            "name": "二分图 /【模板】线段树分治",
            "problem_id": "P5787",
            "link": "https://www.luogu.com.cn/problem/P5787",
            "difficulty": "省选/NOI-",
            "tags": ["线段树分治", "扩展域并查集", "二分图"],
            "description": "维护动态图使其为二分图，使用线段树分治和扩展域并查集。"
        },
        {
            "name": "最小 mex 生成树",
            "problem_id": "P5631",
            "link": "https://www.luogu.com.cn/problem/P5631",
            "difficulty": "省选/NOI-",
            "tags": ["线段树分治", "并查集", "生成树", "二分"],
            "description": "求生成树使得边权集合的 mex 最小。"
        },
        {
            "name": "大融合",
            "problem_id": "P4219",
            "link": "https://www.luogu.com.cn/problem/P4219",
            "difficulty": "省选/NOI-",
            "tags": ["线段树分治", "并查集", "图论"],
            "description": "支持加边和查询边负载，边负载定义为删去该边后两个连通块大小的乘积。"
        },
        {
            "name": "连通图",
            "problem_id": "P5227",
            "link": "https://www.luogu.com.cn/problem/P5227",
            "difficulty": "省选/NOI-",
            "tags": ["线段树分治", "并查集", "图论"],
            "description": "给定初始连通图，每次删除一些边，查询是否仍连通。"
        },
        {
            "name": "动态图连通性",
            "problem_id": "LOJ#121",
```

```
            "link": "https://loj.ac/problem/121",
            "difficulty": "省选/NOI-",
            "tags": ["线段树分治", "可撤销并查集", "动态图"],
            "description": "支持动态加边、删边操作，查询两点间连通性。"
        }
    ]
    return problems

def get_atcoder_problems():
    """获取 AtCoder 上与相关算法的题目"""
    problems = [
        {
            "name": "Cell Division",
            "contest": "AtCoder Grand Contest 010",
            "problem_id": "C",
            "link": "https://atcoder.jp/contests/agc010/tasks/agc010_c",
            "difficulty": "2300",
            "tags": ["Union Find", "Divide and Conquer"],
            "description": "Divide rectangles and count connected components after each
division."
        }
    ]
    return problems

def get_libreoj_problems():
    """获取 LibreOJ 上与 segment tree divide and conquer 相关的题目"""
    problems = [
        {
            "name": "动态图连通性",
            "problem_id": "#121",
            "link": "https://loj.ac/problem/121",
            "difficulty": "省选/NOI-",
            "tags": ["线段树分治", "可撤销并查集", "动态图"],
            "description": "支持动态加边、删边操作，查询两点间连通性。"
        },
        {
            "name": "「NOI2015」程序自动分析",
            "problem_id": "#2124",
            "link": "https://loj.ac/p/2124",
            "difficulty": "提高",
            "tags": ["线段树分治", "并查集"],
            "description": "处理一系列相等和不等的约束条件，判断是否可行。"
        },
```

```
        {
            "name": "「SDOI2016」游戏",
            "problem_id": "#2005",
            "link": "https://loj.ac/p/2005",
            "difficulty": "省选",
            "tags": ["线段树分治", "扩展域并查集"],
            "description": "使用线段树分治和扩展域并查集处理路径覆盖问题。"
        },
        {
            "name": "「SDOI2017」树点涂色",
            "problem_id": "#2152",
            "link": "https://loj.ac/p/2152",
            "difficulty": "省选",
            "tags": ["线段树分治", "树链剖分"],
            "description": "树上动态涂色问题，可以用线段树分治解决。"
        }
    ]
    return problems

def get_spoj_problems():
    """获取 SPOJ 上与相关算法的题目"""
    problems = [
        {
            "name": "Dynamic Graph Connectivity",
            "problem_id": "DYNACON2",
            "link": "https://www.spoj.com/problems/DYNACON2/",
            "difficulty": "Hard",
            "tags": ["Segment Tree", "Divide and Conquer", "Union Find"],
            "description": "Dynamic graph connectivity problem using segment tree divide and
conquer."
        }
    ]
    return problems

def get_nowcoder_problems():
    """获取牛客网上与线段树分治相关的题目"""
    problems = [
        {
            "name": "NC15662 最大匹配",
            "link": "https://ac.nowcoder.com/acm/problem/15662",
            "difficulty": "中等",
            "tags": ["线段树分治", "二分图匹配"],
            "description": "动态二分图最大匹配问题，可以用线段树分治优化。"
```

```
        },
        {
            "name": "NC15563 小G的烦恼",
            "link": "https://ac.nowcoder.com/acm/problem/15563",
            "difficulty": "困难",
            "tags": ["线段树分治", "并查集", "数学"],
            "description": "处理多组约束条件，判断是否存在可行解。"
        }
    ]
    return problems

# ==========================================
# 线段树分治核心实现代码模板
# ==========================================

class RollbackDSU:
    """
    可撤销并查集（Rollback Disjoint Set Union）

    时间复杂度：
    - find: O(log n) （不使用路径压缩，仅使用按秩合并）
    - union: O(log n)
    - rollback: O(1)

    空间复杂度：O(n + m)，其中 n 是节点数，m 是合并操作次数
    """
    def __init__(self, size):
        self.father = list(range(size))  # 父节点数组
        self.rank = [1] * size           # 秩数组（树高上界）
        self.history = []                # 操作历史记录
        self.version = 0                 # 当前版本号

    def find(self, x):
        """
        查找 x 所在集合的根节点（不使用路径压缩，以支持撤销操作）

        参数：
            x: 要查找的节点

        返回：
            节点 x 所在集合的根节点
        """
        while x != self.father[x]:
```

```
            x = self.father[x]
        return x

    def union(self, x, y):
        """
        合并 x 和 y 所在的集合

        参数:
            x, y: 要合并的两个节点

        返回:
            bool: 如果 x 和 y 原本不在同一集合中则返回 True，否则返回 False
        """
        fx = self.find(x)
        fy = self.find(y)

        if fx == fy:
            return False

        # 按秩合并：将较小的树合并到较大的树上
        if self.rank[fx] < self.rank[fy]:
            fx, fy = fy, fx

        # 记录操作前的状态用于撤销
        self.history.append((fy, self.father[fy], fx, self.rank[fx]))
        self.version += 1

        # 执行合并操作
        self.father[fy] = fx
        if self.rank[fx] == self.rank[fy]:
            self.rank[fx] += 1

        return True

    def rollback(self, version):
        """
        撤销操作到指定版本

        参数:
            version: 要回滚到的版本号
        """
        while self.version > version:
            fy, father_fy, fx, rank_fx = self.history.pop()
```

```
            self.father[fy] = father_fy  # 恢复父节点
            self.rank[fx] = rank_fx      # 恢复秩
            self.version -= 1

    def is_connected(self, x, y):
        """
        判断 x 和 y 是否在同一集合中

        参数:
            x, y: 要判断的两个节点

        返回:
            bool: 如果 x 和 y 在同一集合中则返回 True，否则返回 False
        """
        return self.find(x) == self.find(y)

class ExtendedRollbackDSU:
    """
    扩展域可撤销并查集（用于处理二分图问题）

    时间复杂度：
    - 所有操作：O(log n)

    空间复杂度：O(n)
    """
    def __init__(self, size):
        self.size = size
        self.father = list(range(2 * size))  # 扩展 2 倍空间，0~size-1 代表原节点，size~2size-1 代表相反节点
        self.rank = [1] * (2 * size)
        self.history = []
        self.version = 0

    def find(self, x):
        while x != self.father[x]:
            x = self.father[x]
        return x

    def union(self, x, y):
        """
        合并 x 和 y 所在的集合

        参数:
```

```
        x, y: 要合并的两个节点
    """
    fx = self.find(x)
    fy = self.find(y)

    if fx == fy:
        return False

    if self.rank[fx] < self.rank[fy]:
        fx, fy = fy, fx

    self.history.append((fy, self.father[fy], fx, self.rank[fx]))
    self.version += 1

    self.father[fy] = fx
    if self.rank[fx] == self.rank[fy]:
        self.rank[fx] += 1

    return True

def rollback(self, version):
    while self.version > version:
        fy, father_fy, fx, rank_fx = self.history.pop()
        self.father[fy] = father_fy
        self.rank[fx] = rank_fx
        self.version -= 1

def add_edge(self, u, v):
    """
    在二分图中添加一条边 u-v，即 u 和 v 必须在不同的集合中
    这等价于 u 和 v 的相反节点合并，v 和 u 的相反节点合并

    参数：
        u, v: 要连接的两个节点

    返回：
        bool: 如果添加这条边不会导致矛盾（即图仍保持二分图性质）则返回 True
    """
    # 检查 u 和 v 是否已经在同一集合中，如果是则添加这条边会导致矛盾
    if self.find(u) == self.find(v):
        return False

    # 添加边 u-v：u 和 v 的相反节点合并，v 和 u 的相反节点合并
```

```
        self.union(u, v + self.size)
        self.union(v, u + self.size)
        return True

class SegmentTreeDivideConquer:
    """
    线段树分治算法模板

    时间复杂度：O((n + m) log Q)，其中 Q 是时间范围
    空间复杂度：O(m log Q)

    使用方法：
    1. 初始化线段树分治结构
    2. 为每个操作添加时间区间
    3. 调用 solve()方法执行分治
    """
    def __init__(self, max_time):
        self.max_time = max_time
        # 每个时间区间存储的操作列表
        self.operations = [[] for _ in range(4 * (max_time + 1))]

    def add_operation(self, l, r, op):
        """
        添加一个在时间[l, r]内有效的操作

        参数：
            l: 操作开始时间（包含）
            r: 操作结束时间（包含）
            op: 操作信息，例如边的两个端点 u 和 v
        """
        self._update(1, 1, self.max_time, l, r, op)

    def _update(self, node, node_l, node_r, l, r, op):
        """
        线段树更新操作：将操作 op 添加到所有覆盖时间区间[l, r]的节点中
        """
        if node_r < l or node_l > r:
            return

        if l <= node_l and node_r <= r:
            self.operations[node].append(op)
            return
```

```
        mid = (node_l + node_r) // 2
        self._update(2 * node, node_l, mid, l, r, op)
        self._update(2 * node + 1, mid + 1, node_r, l, r, op)

    def solve(self, process_func, rollback_func):
        """
        执行线段树分治

        参数：
            process_func: 处理当前节点操作的函数
            rollback_func: 回滚操作的函数
        """
        self._dfs(1, 1, self.max_time, process_func, rollback_func)

    def _dfs(self, node, node_l, node_r, process_func, rollback_func):
        """
        DFS 遍历线段树，处理每个时间区间的操作
        """
        # 记录当前版本，用于回滚
        current_version = rollback_func()

        # 处理当前节点的所有操作
        for op in self.operations[node]:
            process_func(op)

        # 如果当前节点是叶子节点，执行查询或其他操作
        if node_l == node_r:
            # 这里可以添加查询处理逻辑
            pass
        else:
            mid = (node_l + node_r) // 2
            self._dfs(2 * node, node_l, mid, process_func, rollback_func)
            self._dfs(2 * node + 1, mid + 1, node_r, process_func, rollback_func)

        # 回滚到进入当前节点前的状态
        rollback_func(current_version)

def get_solution_templates():
    """生成各类题目的解法模板，包含 Python、Java 和 C++三种实现"""
    templates = {
        "segment_tree_divide_conquer_python": '''
# 线段树分治通用模板 - Python 实现
# 时间复杂度：O(m log Q * α(n))，其中 m 是操作数，Q 是时间范围，α 是阿克曼函数的反函数（近似常
```

数）
```
# 空间复杂度：O(m log Q + n)，其中 n 是节点数
class SegmentTreeDivideConquer:
    def __init__(self, max_time):
        # 初始化线段树，大小为 4 倍最大时间
        self.max_time = max_time
        self.tree = [[] for _ in range(4 * (max_time + 1))]
        self.time_range = []

    # 将操作添加到线段树的对应区间
    def add_operation(self, l, r, op):
        """将操作添加到时间区间[l, r]"""
        self._update(1, 1, self.max_time, l, r, op)

    # 线段树更新
    def _update(self, node, l, r, ul, ur, op):
        """在线段树中更新区间[ul, ur]，添加操作 op"""
        if ur < l or ul > r:
            return
        if ul <= l and r <= ur:
            self.tree[node].append(op)
            return
        mid = (l + r) // 2
        self._update(2*node, l, mid, ul, ur, op)
        self._update(2*node+1, mid+1, r, ul, ur, op)

    # 线段树分治主函数
    def solve(self, process_func, rollback_func):
        """
        分治求解问题
        process_func: 处理操作的函数
        rollback_func: 回滚操作的函数
        """
        def dfs(node, l, r):
            # 记录当前操作数量，用于后续回滚
            op_count = 0

            # 处理当前节点的所有操作
            for op in self.tree[node]:
                if process_func(op):
                    op_count += 1

            # 叶子节点，处理查询
```

```
            if l == r:
                # 这里可以处理时间点 l 的查询
                pass
            else:
                mid = (l + r) // 2
                dfs(2*node, l, mid)
                dfs(2*node+1, mid+1, r)

            # 回滚操作
            for _ in range(op_count):
                rollback_func()

        dfs(1, 1, self.max_time)

# 可撤销并查集 - Python 实现
class RollbackDSU:
    def __init__(self, n):
        self.n = n
        self.parent = list(range(n))
        self.size = [1] * n
        self.stack = []  # 记录操作历史，用于回滚

    def find(self, x):
        # 注意：线段树分治中不能使用路径压缩，否则无法正确回滚
        while x != self.parent[x]:
            x = self.parent[x]
        return x

    def union(self, x, y):
        """合并 x 和 y 所在的集合，返回是否成功合并"""
        x_root = self.find(x)
        y_root = self.find(y)

        if x_root == y_root:
            return False

        # 按秩合并，将小集合合并到大集合
        if self.size[x_root] < self.size[y_root]:
            x_root, y_root = y_root, x_root

        # 保存操作记录，用于回滚
        self.stack.append((y_root, self.parent[y_root], x_root, self.size[x_root]))
```

```
        # 执行合并
        self.parent[y_root] = x_root
        self.size[x_root] += self.size[y_root]
        return True

    def rollback(self):
        """回滚最后一次合并操作"""
        if not self.stack:
            return

        y_root, parent, x_root, size = self.stack.pop()
        self.parent[y_root] = parent
        self.size[x_root] = size

    def same_set(self, x, y):
        """判断 x 和 y 是否在同一集合"""
        return self.find(x) == self.find(y)

# 扩展域可撤销并查集（用于二分图检测）
class ExtendedRollbackDSU:
    def __init__(self, n):
        # 使用 2n 大小的数组，1~n 表示原图中的节点，n+1~2n 表示该节点的相反节点
        self.n = n
        self.parent = list(range(2 * n + 1))  # 节点编号从 1 开始
        self.size = [1] * (2 * n + 1)
        self.stack = []

    def find(self, x):
        while x != self.parent[x]:
            x = self.parent[x]
        return x

    def union(self, x, y):
        """合并 x 和 y 所在的集合"""
        x_root = self.find(x)
        y_root = self.find(y)

        if x_root == y_root:
            return False

        if self.size[x_root] < self.size[y_root]:
            x_root, y_root = y_root, x_root
```

```
        self.stack.append((y_root, self.parent[y_root], x_root, self.size[x_root]))
        self.parent[y_root] = x_root
        self.size[x_root] += self.size[y_root]
        return True

    def rollback(self):
        if not self.stack:
            return

        y_root, parent, x_root, size = self.stack.pop()
        self.parent[y_root] = parent
        self.size[x_root] = size

    def is_bipartite(self, x, y):
        """判断 x 和 y 是否可以在二分图中共存"""
        # x 和 y 不能在同一集合，同时 x 和 y 的相反节点也不能在同一集合
        return self.find(x) != self.find(y) and self.find(x) != self.find(y + self.n)

    def add_constraint(self, x, y, is_same):
        """添加约束：x 和 y 是否属于同一集合"""
        if is_same:
            # x 和 y 必须在同一集合，x 的相反节点和 y 的相反节点也必须在同一集合
            if not self.is_bipartite(x, y + self.n):
                return False
            self.union(x, y)
            self.union(x + self.n, y + self.n)
        else:
            # x 和 y 必须不在同一集合，x 和 y 的相反节点必须在同一集合
            if not self.is_bipartite(x, y):
                return False
            self.union(x, y + self.n)
            self.union(x + self.n, y)
        return True
        ''',
        "segment_tree_divide_conquer_java": '''
// 线段树分治通用模板 - Java 实现
// 时间复杂度：O(m log Q * α(n))，其中 m 是操作数，Q 是时间范围，α 是阿克曼函数的反函数（近似常数）
// 空间复杂度：O(m log Q + n)，其中 n 是节点数
import java.util.*;

class SegmentTreeDivideConquer {
    // 线段树节点信息
```

```
static class Node {
    List<int[]> operations = new ArrayList<>();
}

private Node[] tree;
private int maxTime;

public SegmentTreeDivideConquer(int maxTime) {
    this.maxTime = maxTime;
    // 初始化线段树，大小为 4 倍最大时间
    this.tree = new Node[4 * (maxTime + 1)];
    for (int i = 0; i < tree.length; i++) {
        tree[i] = new Node();
    }
}

// 将操作添加到线段树的对应区间
public void addOperation(int l, int r, int[] operation) {
    update(1, 1, maxTime, l, r, operation);
}

// 线段树更新
private void update(int node, int l, int r, int ul, int ur, int[] operation) {
    if (ur < l || ul > r) {
        return;
    }
    if (ul <= l && r <= ur) {
        tree[node].operations.add(operation);
        return;
    }
    int mid = (l + r) / 2;
    update(2 * node, l, mid, ul, ur, operation);
    update(2 * node + 1, mid + 1, r, ul, ur, operation);
}

// 线段树分治主函数
public void solve(OperationProcessor processor) {
    dfs(1, 1, maxTime, processor);
}

private void dfs(int node, int l, int r, OperationProcessor processor) {
    // 记录当前操作数量，用于后续回滚
    int opCount = 0;
```

```
        // 处理当前节点的所有操作
        for (int[] op : tree[node].operations) {
            if (processor.process(op)) {
                opCount++;
            }
        }

        // 叶子节点，处理查询
        if (l == r) {
            // 这里可以处理时间点 l 的查询
            processor.query(l);
        } else {
            int mid = (l + r) / 2;
            dfs(2 * node, l, mid, processor);
            dfs(2 * node + 1, mid + 1, r, processor);
        }

        // 回滚操作
        for (int i = 0; i < opCount; i++) {
            processor.rollback();
        }
    }

    // 操作处理器接口
    public interface OperationProcessor {
        boolean process(int[] operation);
        void rollback();
        void query(int time);
    }
}

// 可撤销并查集 - Java 实现
class RollbackDSU {
    private int[] parent;
    private int[] size;
    private Stack<int[]> stack; // 记录操作历史，用于回滚

    public RollbackDSU(int n) {
        parent = new int[n];
        size = new int[n];
        stack = new Stack<>();
```

```
        for (int i = 0; i < n; i++) {
            parent[i] = i;
            size[i] = 1;
        }
    }

    public int find(int x) {
        // 注意：线段树分治中不能使用路径压缩，否则无法正确回滚
        while (x != parent[x]) {
            x = parent[x];
        }
        return x;
    }

    public boolean union(int x, int y) {
        // 合并 x 和 y 所在的集合，返回是否成功合并
        int xRoot = find(x);
        int yRoot = find(y);

        if (xRoot == yRoot) {
            return false;
        }

        // 按秩合并，将小集合合并到大集合
        if (size[xRoot] < size[yRoot]) {
            int temp = xRoot;
            xRoot = yRoot;
            yRoot = temp;
        }

        // 保存操作记录，用于回滚
        stack.push(new int[]{yRoot, parent[yRoot], xRoot, size[xRoot]});

        // 执行合并
        parent[yRoot] = xRoot;
        size[xRoot] += size[yRoot];
        return true;
    }

    public void rollback() {
        // 回滚最后一次合并操作
        if (stack.isEmpty()) {
            return;
```

```
        }

        int[] op = stack.pop();
        int yRoot = op[0];
        int prevParent = op[1];
        int xRoot = op[2];
        int prevSize = op[3];

        parent[yRoot] = prevParent;
        size[xRoot] = prevSize;
    }

    public boolean sameSet(int x, int y) {
        // 判断 x 和 y 是否在同一集合
        return find(x) == find(y);
    }

    public int getSize(int x) {
        // 获取 x 所在集合的大小
        return size[find(x)];
    }
}

// 扩展域可撤销并查集（用于二分图检测）
class ExtendedRollbackDSU {
    private int n;
    private int[] parent;
    private int[] size;
    private Stack<int[]> stack;

    public ExtendedRollbackDSU(int n) {
        this.n = n;
        // 使用 2n+1 大小的数组，1~n 表示原图中的节点，n+1~2n 表示该节点的相反节点
        parent = new int[2 * n + 1];
        size = new int[2 * n + 1];
        stack = new Stack<>();

        for (int i = 1; i <= 2 * n; i++) {
            parent[i] = i;
            size[i] = 1;
        }
    }
```

```
public int find(int x) {
    while (x != parent[x]) {
        x = parent[x];
    }
    return x;
}

public boolean union(int x, int y) {
    int xRoot = find(x);
    int yRoot = find(y);

    if (xRoot == yRoot) {
        return false;
    }

    if (size[xRoot] < size[yRoot]) {
        int temp = xRoot;
        xRoot = yRoot;
        yRoot = temp;
    }

    stack.push(new int[]{yRoot, parent[yRoot], xRoot, size[xRoot]});
    parent[yRoot] = xRoot;
    size[xRoot] += size[yRoot];
    return true;
}

public void rollback() {
    if (stack.isEmpty()) {
        return;
    }

    int[] op = stack.pop();
    int yRoot = op[0];
    int prevParent = op[1];
    int xRoot = op[2];
    int prevSize = op[3];

    parent[yRoot] = prevParent;
    size[xRoot] = prevSize;
}

public boolean isBipartite(int x, int y) {
```

```
        // 判断 x 和 y 是否可以在二分图中共存
        return find(x) != find(y) && find(x) != find(y + n);
    }

    public boolean addConstraint(int x, int y, boolean isSame) {
        // 添加约束：x 和 y 是否属于同一集合
        if (isSame) {
            // x 和 y 必须在同一集合，x 的相反节点和 y 的相反节点也必须在同一集合
            if (find(x) == find(y + n)) {
                return false;
            }
            union(x, y);
            union(x + n, y + n);
        } else {
            // x 和 y 必须不在同一集合，x 和 y 的相反节点必须在同一集合
            if (find(x) == find(y)) {
                return false;
            }
            union(x, y + n);
            union(x + n, y);
        }
        return true;
    }
}
        ''',
        "segment_tree_divide_conquer_cpp": '''
// 线段树分治通用模板 - C++实现
// 时间复杂度：O(m log Q * α(n))，其中 m 是操作数，Q 是时间范围，α 是阿克曼函数的反函数（近似常数）
// 空间复杂度：O(m log Q + n)，其中 n 是节点数
#include <iostream>
#include <vector>
#include <stack>
#include <functional>
using namespace std;

struct Operation {
    int u, v; // 可以根据具体问题修改操作的结构
    // 其他需要的字段
};

class SegmentTreeDivideConquer {
private:
```

```
    vector<vector<Operation>> tree;
    int max_time;

    void update(int node, int l, int r, int ul, int ur, const Operation& op) {
        if (ur < l || ul > r) {
            return;
        }
        if (ul <= l && r <= ur) {
            tree[node].push_back(op);
            return;
        }
        int mid = (l + r) / 2;
        update(2 * node, l, mid, ul, ur, op);
        update(2 * node + 1, mid + 1, r, ul, ur, op);
    }

public:
    SegmentTreeDivideConquer(int max_time) : max_time(max_time) {
        tree.resize(4 * (max_time + 1));
    }

    void addOperation(int l, int r, const Operation& op) {
        update(1, 1, max_time, l, r, op);
    }

    // 使用模板函数，允许用户传入自定义的处理函数
    template<typename ProcessFunc, typename RollbackFunc, typename QueryFunc>
    void solve(ProcessFunc process, RollbackFunc rollback, QueryFunc query) {
        function<void(int, int, int)> dfs = [&](int node, int l, int r) {
            int op_count = 0;

            // 处理当前节点的所有操作
            for (const auto& op : tree[node]) {
                if (process(op)) {
                    op_count++;
                }
            }

            // 叶子节点，处理查询
            if (l == r) {
                query(l);
            } else {
                int mid = (l + r) / 2;
```

```
                dfs(2 * node, l, mid);
                dfs(2 * node + 1, mid + 1, r);
            }

            // 回滚操作
            for (int i = 0; i < op_count; i++) {
                rollback();
            }
        };

        dfs(1, 1, max_time);
    }
};

// 可撤销并查集 - C++实现
class RollbackDSU {
private:
    vector<int> parent;
    vector<int> size;
    stack<tuple<int, int, int, int>> stk; // (y_root, prev_parent, x_root, prev_size)

public:
    RollbackDSU(int n) {
        parent.resize(n);
        size.resize(n, 1);
        for (int i = 0; i < n; i++) {
            parent[i] = i;
        }
    }

    int find(int x) {
        // 注意：线段树分治中不能使用路径压缩，否则无法正确回滚
        while (x != parent[x]) {
            x = parent[x];
        }
        return x;
    }

    bool unite(int x, int y) {
        int x_root = find(x);
        int y_root = find(y);

        if (x_root == y_root) {
```

```
            return false;
        }

        // 按秩合并，将小集合合并到大集合
        if (size[x_root] < size[y_root]) {
            swap(x_root, y_root);
        }

        // 保存操作记录，用于回滚
        stk.emplace(y_root, parent[y_root], x_root, size[x_root]);

        // 执行合并
        parent[y_root] = x_root;
        size[x_root] += size[y_root];
        return true;
    }

    void rollback() {
        if (stk.empty()) {
            return;
        }

        auto [y_root, prev_parent, x_root, prev_size] = stk.top();
        stk.pop();

        parent[y_root] = prev_parent;
        size[x_root] = prev_size;
    }

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

    int getSize(int x) {
        return size[find(x)];
    }
};

// 扩展域可撤销并查集（用于二分图检测）
class ExtendedRollbackDSU {
private:
    int n;
    vector<int> parent;
```

```
    vector<int> size;
    stack<tuple<int, int, int, int>> stk;

public:
    ExtendedRollbackDSU(int n) : n(n) {
        // 使用 2n+1 大小的数组，1~n 表示原图中的节点，n+1~2n 表示该节点的相反节点
        parent.resize(2 * n + 1);
        size.resize(2 * n + 1, 1);
        for (int i = 1; i <= 2 * n; i++) {
            parent[i] = i;
        }
    }

    int find(int x) {
        while (x != parent[x]) {
            x = parent[x];
        }
        return x;
    }

    bool unite(int x, int y) {
        int x_root = find(x);
        int y_root = find(y);

        if (x_root == y_root) {
            return false;
        }

        if (size[x_root] < size[y_root]) {
            swap(x_root, y_root);
        }

        stk.emplace(y_root, parent[y_root], x_root, size[x_root]);
        parent[y_root] = x_root;
        size[x_root] += size[y_root];
        return true;
    }

    void rollback() {
        if (stk.empty()) {
            return;
        }
```

```
        auto [y_root, prev_parent, x_root, prev_size] = stk.top();
        stk.pop();

        parent[y_root] = prev_parent;
        size[x_root] = prev_size;
    }

    bool isBipartite(int x, int y) {
        return find(x) != find(y) && find(x) != find(y + n);
    }

    bool addConstraint(int x, int y, bool isSame) {
        if (isSame) {
            // x 和 y 必须在同一集合，x 的相反节点和 y 的相反节点也必须在同一集合
            if (find(x) == find(y + n)) {
                return false;
            }
            unite(x, y);
            unite(x + n, y + n);
        } else {
            // x 和 y 必须不在同一集合，x 和 y 的相反节点必须在同一集合
            if (find(x) == find(y)) {
                return false;
            }
            unite(x, y + n);
            unite(x + n, y);
        }
        return true;
    }
};
        ''',
        "minimum_mex_spanning_tree": '''
# 最小 mex 生成树问题 - Python 实现
# 时间复杂度：O(m log m * α(n))，其中 m 是边数，n 是节点数
# 空间复杂度：O(m + n)

def min_mex_spanning_tree(n, edges):
    """
    求解最小 mex 生成树
    mex 定义为生成树中边权集合中最小的未出现的非负整数

    参数:
        n: 节点数
```

```
        edges: 边列表，格式为[(u, v, w)]，其中 u 和 v 是节点，w 是边权

    返回:
        最小 mex 值和对应的生成树
    """
    # 按照边权从小到大排序
    edges.sort(key=lambda x: x[2])

    # 从 0 开始尝试找到最小的 mex 值
    for mex in range(len(edges) + 2):
        # 构建不包含 mex 的边的图
        dsu = RollbackDSU(n)
        valid_edges = [e for e in edges if e[2] < mex]

        # 尝试构建生成树
        for u, v, w in valid_edges:
            dsu.union(u, v)

        # 检查是否所有节点连通
        root = dsu.find(0)
        connected = True
        for i in range(1, n):
            if dsu.find(i) != root:
                connected = False
                break

        # 如果连通，则 mex 是答案
        if connected:
            return mex, valid_edges

    return len(edges) + 1, []

# 最小 mex 生成树问题的线段树分治解法
# 时间复杂度：O(m log m * α(n))
# 空间复杂度：O(m log m + n)

def min_mex_spanning_tree_segment_tree(n, edges):
    """
    使用线段树分治求解最小 mex 生成树
    """
    # 预处理：计算每个边权可能的区间
    edges.sort(key=lambda x: x[2])
    max_mex = len(edges) + 2
```

```
# 构建线段树分治结构
stdc = SegmentTreeDivideConquer(max_mex)

# 将每条边添加到对应的区间
for u, v, w in edges:
    # 边 w 可以出现在 mex > w 的所有情况中
    stdc.add_operation(w + 1, max_mex, (u, v))

# 用于记录结果
result_mex = max_mex

# 定义处理和回滚函数
dsu = RollbackDSU(n)

def process(op):
    u, v = op
    return dsu.union(u, v)

def rollback():
    dsu.rollback()

# 执行线段树分治
def dfs(node, l, r):
    nonlocal result_mex
    op_count = 0

    # 处理当前节点的所有边
    for op in stdc.tree[node]:
        if process(op):
            op_count += 1

    # 检查当前 mex 是否可行
    is_connected = True
    root = dsu.find(0)
    for i in range(1, n):
        if dsu.find(i) != root:
            is_connected = False
            break

    if is_connected and l < result_mex:
        result_mex = l
```

```
        # 继续分治
        if l < r:
            mid = (l + r) // 2
            dfs(2*node, l, mid)
            dfs(2*node+1, mid+1, r)

        # 回滚操作
        for _ in range(op_count):
            rollback()

    dfs(1, 0, max_mex)
    return result_mex
        ''',
        "dynamic_xor_path": '''
# 动态XOR路径问题 - Python实现
# 时间复杂度：O(m log n * log Q)，其中m是边数，n是节点数，Q是时间范围
# 空间复杂度：O(n log n + m log Q)

class LinearBasis:
    """线性基，用于处理异或问题"""
    def __init__(self):
        self.basis = [0] * 60  # 假设最大权值为2^60

    def insert(self, x):
        """插入一个数到线性基中"""
        for i in range(59, -1, -1):
            if (x >> i) & 1:
                if self.basis[i] == 0:
                    self.basis[i] = x
                    return True
                else:
                    x ^= self.basis[i]
        return False

    def query_max(self):
        """查询线性基中的最大异或值"""
        res = 0
        for i in range(59, -1, -1):
            if (res ^ self.basis[i]) > res:
                res ^= self.basis[i]
        return res

    def copy(self):
```

```
        """复制线性基"""
        new_lb = LinearBasis()
        new_lb.basis = self.basis.copy()
        return new_lb

# 带撤销的线性基
class RollbackLinearBasis:
    def __init__(self):
        self.basis = [0] * 60
        self.history = []

    def insert(self, x):
        """插入一个数到线性基中，返回是否成功插入"""
        original_basis = self.basis.copy()
        for i in range(59, -1, -1):
            if (x >> i) & 1:
                if self.basis[i] == 0:
                    self.basis[i] = x
                    self.history.append(original_basis)
                    return True
                else:
                    x ^= self.basis[i]
        return False

    def rollback(self):
        """回滚到上一个状态"""
        if self.history:
            self.basis = self.history.pop()

    def query_max(self):
        """查询线性基中的最大异或值"""
        res = 0
        for i in range(59, -1, -1):
            if (res ^ self.basis[i]) > res:
                res ^= self.basis[i]
        return res

# 动态 XOR 路径问题的线段树分治解法
def dynamic_xor_path(n, edges, queries):
    """
    求解动态 XOR 路径问题
    边有出现和消失的时间，查询特定时间点的路径最大异或和
```

```
    参数:
        n: 节点数
        edges: 边列表，格式为[(u, v, w, l, r)]，其中 u 和 v 是节点，w 是边权，[l, r]是存在时间区间
        queries: 查询列表，格式为[(t, s, t_node)]，查询时间 t 时从 s 到 t_node 的最大异或路径
    """
    # 离散化时间点
    all_times = {edge[3] for edge in edges} | {edge[4] for edge in edges} | {q[0] for q in
queries}
    time_map = {t: i+1 for i, t in enumerate(sorted(all_times))}
    max_time = len(time_map)

    # 初始化线段树分治
    stdc = SegmentTreeDivideConquer(max_time)

    # 将边添加到线段树中
    for u, v, w, l, r in edges:
        stdc.add_operation(time_map[l], time_map[r], (u, v, w))

    # 预处理查询
    query_by_time = [[] for _ in range(max_time + 2)]
    for idx, (t, s, t_node) in enumerate(queries):
        query_by_time[time_map[t]].append((idx, s, t_node))

    # 结果数组
    results = [0] * len(queries)

    # 初始化带撤销的线性基和并查集
    lb = RollbackLinearBasis()
    dsu = RollbackDSU(n)
    # 用于记录每个节点到根的异或路径
    xor_path = [0] * n

    # 处理函数
    def process(op):
        u, v, w = op

        # 查找 u 和 v 的根
        root_u = dsu.find(u)
        root_v = dsu.find(v)

        # 如果已经连通，检查是否形成环
        if root_u == root_v:
            # 计算环的异或值，并尝试插入线性基
```

```
            cycle_xor = xor_path[u] ^ xor_path[v] ^ w
            return lb.insert(cycle_xor)
        else:
            # 合并两个集合
            dsu.union(root_u, root_v)
            # 更新异或路径
            # 注意：这里需要根据合并方向调整 xor_path
            # 为简化，假设总是将 root_v 合并到 root_u
            new_xor = xor_path[u] ^ xor_path[v] ^ w
            # 保存当前状态用于回滚
            lb.history.append(lb.basis.copy())
            # 这里简化处理，实际上需要更复杂的路径维护
            return True

    # 回滚函数
    def rollback():
        lb.rollback()
        dsu.rollback()

    # 查询函数
    def query(time):
        for idx, s, t_node in query_by_time[time]:
            # 计算 s 到 t_node 的路径异或值
            path_xor = xor_path[s] ^ xor_path[t_node]
            # 使用线性基查询最大值
            temp_lb = RollbackLinearBasis()
            temp_lb.basis = lb.basis.copy()
            temp_lb.insert(path_xor)
            results[idx] = temp_lb.query_max()

    # 执行线段树分治
    stdc.solve(process, rollback)

    return results
        ''',
        "interval_covering": '''
# 区间覆盖问题 - Python 实现
# 时间复杂度：O((n + m) log n)，其中 n 是区间数，m 是查询数
# 空间复杂度：O(n + m)

def interval_covering(intervals, queries):
    """
    区间覆盖问题：查询每个时间点被多少区间覆盖
```

```
    参数:
        intervals: 区间列表，格式为[(l, r)]，表示区间覆盖的时间段
        queries: 查询列表，每个元素是一个时间点 t

    返回:
        每个查询的覆盖次数
    """
    # 离散化时间点
    import bisect
    all_times = {interval[0] for interval in intervals} | {interval[1] + 1 for interval in
intervals} | set(queries)
    time_list = sorted(all_times)
    time_map = {t: i for i, t in enumerate(time_list)}
    max_time = len(time_list)

    # 初始化线段树分治
    stdc = SegmentTreeDivideConquer(max_time)

    # 将每个区间视为一个在时间[l, r]内的+1 操作
    for l, r in intervals:
        stdc.add_operation(time_map[l], time_map[r], 1)

    # 预处理查询
    query_by_time = [[] for _ in range(max_time + 2)]
    for idx, t in enumerate(queries):
        # 找到 t 对应的离散化时间点
        pos = bisect.bisect_left(time_list, t)
        query_by_time[pos].append(idx)

    # 结果数组
    results = [0] * len(queries)
    current_count = 0

    # 处理函数
    def process(op):
        nonlocal current_count
        current_count += op
        return True

    # 回滚函数
    def rollback():
        nonlocal current_count
```

```
        current_count -= 1

    # 查询函数
    def query(time):
        for idx in query_by_time[time]:
            results[idx] = current_count

    # 执行线段树分治
    stdc.solve(process, rollback)

    return results
        '''
    }
    return templates

def generate_training_plan():
    """生成线段树分治的分级训练计划

    训练计划基于题目难度和类型进行分级，从基础到进阶，
    帮助学习者系统掌握线段树分治算法。
    """
    plan = {
        "basic": [
            {
                "name": "P5787 二分图 /【模板】线段树分治",
                "platform": "洛谷",
                "difficulty": "中等",
                "type": "二分图检测",
                "description": "线段树分治与扩展域并查集结合的经典问题",
                "skills": ["扩展域并查集", "线段树分治基础"]
            },
            {
                "name": "LOJ#121 动态图连通性",
                "platform": "LibreOJ",
                "difficulty": "中等",
                "type": "动态连通性",
                "description": "使用线段树分治处理动态边的连通性问题",
                "skills": ["可撤销并查集", "动态连通性"]
            },
            {
                "name": "Range Frequency Queries",
                "platform": "LeetCode",
                "difficulty": "中等",
```

```
            "type": "区间查询",
            "description": "使用线段树处理区间频率查询",
            "skills": ["线段树", "频率统计"]
        },
        {
            "name": "P4219 大融合",
            "platform": "洛谷",
            "difficulty": "中等",
            "type": "动态树",
            "description": "线段树分治结合 LCT 处理动态树问题",
            "skills": ["动态树", "线段树分治"]
        },
        {
            "name": "Dynamic Connectivity",
            "platform": "SPOJ",
            "difficulty": "中等",
            "type": "动态连通性",
            "description": "基础的动态连通性问题，线段树分治入门",
            "skills": ["可撤销并查集", "线段树分治"]
        },
        {
            "name": "HDU 3974 Assign the task",
            "platform": "杭电 OJ",
            "difficulty": "中等",
            "type": "树链操作",
            "description": "树链上的区间更新与查询问题",
            "skills": ["树链剖分", "线段树"]
        },
        {
            "name": "AtCoder ABC200 D - Happy Birthday! 2",
            "platform": "AtCoder",
            "difficulty": "C",
            "type": "模运算",
            "description": "利用模运算和线段树分治思想的问题",
            "skills": ["模运算", "组合数学"]
        }
    ],
    "intermediate": [
        {
            "name": "CF938G Shortest Path Queries",
            "platform": "Codeforces",
            "difficulty": "2300",
            "type": "最短路径",
```

```
    "description": "结合线性基和线段树分治处理动态异或路径问题",
    "skills": ["线性基", "异或路径", "线段树分治"]
},
{
    "name": "P5631 最小 mex 生成树",
    "platform": "洛谷",
    "difficulty": "提高+/省选-",
    "type": "生成树",
    "description": "使用线段树分治求解最小 mex 生成树",
    "skills": ["最小生成树", "二分答案", "线段树分治"]
},
{
    "name": "CF839D Winter is here",
    "platform": "Codeforces",
    "difficulty": "2000",
    "type": "数学",
    "description": "线段树分治结合数学问题",
    "skills": ["容斥原理", "线段树分治"]
},
{
    "name": "Maximum XOR With an Element From Array",
    "platform": "LeetCode",
    "difficulty": "中等",
    "type": "异或查询",
    "description": "线段树与二分查找结合处理异或问题",
    "skills": ["线段树", "二分查找", "位运算"]
},
{
    "name": "P2542 [AHOI2005]航线规划",
    "platform": "洛谷",
    "difficulty": "提高+",
    "type": "动态割边",
    "description": "离线处理动态删除边的双连通分量问题",
    "skills": ["双连通分量", "线段树分治"]
},
{
    "name": "HDU 5933 ArcSoft's Office Rearrangement",
    "platform": "杭电 OJ",
    "difficulty": "中等",
    "type": "贪心",
    "description": "线段树分治思想与贪心算法结合",
    "skills": ["贪心算法", "线段树分治思想"]
},
```

```
    {
        "name": "P3602 Koishi Loves Segments",
        "platform": "洛谷",
        "difficulty": "提高",
        "type": "线段覆盖",
        "description": "线段覆盖问题，需要选择最多不重叠的线段",
        "skills": ["贪心", "线段树", "扫描线"]
    }
],
"advanced": [
    {
        "name": "CF576E Painting Edges",
        "platform": "Codeforces",
        "difficulty": "2700",
        "type": "强制在线",
        "description": "结合线段树分治和可持久化数据结构处理在线问题",
        "skills": ["强制在线", "线段树分治", "可持久化"]
    },
    {
        "name": "CF1089I Interval-Free Segments",
        "platform": "Codeforces",
        "difficulty": "2500",
        "type": "扫描线",
        "description": "线段树分治与扫描线算法结合",
        "skills": ["扫描线", "线段树分治"]
    },
    {
        "name": "CF126B Password",
        "platform": "Codeforces",
        "difficulty": "2200",
        "type": "字符串",
        "description": "KMP 和线段树分治结合解决字符串问题",
        "skills": ["KMP", "线段树分治"]
    },
    {
        "name": "NOI2015 程序自动分析",
        "platform": "LibreOJ",
        "difficulty": "提高",
        "type": "约束满足",
        "description": "线段树分治处理变量相等和不等约束",
        "skills": ["扩展域并查集", "线段树分治"]
    },
    {
```

```
        "name": "LeetCode 2276. Count Integers in Intervals",
        "platform": "LeetCode",
        "difficulty": "困难",
        "type": "区间管理",
        "description": "动态添加区间并统计被覆盖的整数个数",
        "skills": ["线段树", "区间合并", "计数"]
    },
    {
        "name": "P4008 文本编辑器",
        "platform": "洛谷",
        "difficulty": "省选-",
        "type": "数据结构",
        "description": "可持久化线段树与线段树分治结合的复杂问题",
        "skills": ["可持久化线段树", "线段树分治"]
    },
    {
        "name": "SPOJ DISQUERY - Distance Query",
        "platform": "SPOJ",
        "difficulty": "中等",
        "type": "树上查询",
        "description": "动态树上距离查询问题",
        "skills": ["树链剖分", "线段树"]
    }
],
"expert": [
    {
        "name": "SDOI2016 游戏",
        "platform": "LibreOJ",
        "difficulty": "省选",
        "type": "树上问题",
        "description": "树上的线段树分治问题",
        "skills": ["树链剖分", "线段树分治", "可撤销并查集"]
    },
    {
        "name": "SDOI2017 树点涂色",
        "platform": "LibreOJ",
        "difficulty": "省选",
        "type": "树上操作",
        "description": "树上的动态染色问题，可使用线段树分治解决",
        "skills": ["LCT", "线段树分治"]
    },
    {
        "name": "NC15563 小G的烦恼",
```

```
        "platform": "牛客网",
        "difficulty": "困难",
        "type": "数学优化",
        "description": "结合数学知识和线段树分治的复杂问题",
        "skills": ["数学", "线段树分治", "并查集"]
    },
    {
        "name": "CF601E A Museum Robbery",
        "platform": "Codeforces",
        "difficulty": "2700",
        "type": "背包问题",
        "description": "线段树分治与背包问题结合的复杂问题",
        "skills": ["线段树分治", "背包 DP", "可撤销数据结构"]
    },
    {
        "name": "HDU 6331 Problem M. Walking Plan",
        "platform": "杭电 OJ",
        "difficulty": "困难",
        "type": "矩阵快速幂",
        "description": "线段树分治与矩阵快速幂结合",
        "skills": ["矩阵快速幂", "线段树分治", "动态规划"]
    },
    {
        "name": "P5355 [Ynoi2017] 由乃的玉米田",
        "platform": "洛谷",
        "difficulty": "省选+",
        "type": "数论",
        "description": "结合数论知识和线段树分治的复杂查询问题",
        "skills": ["数论", "线段树分治", "莫队算法"]
    },
    {
        "name": "LeetCode 2398. Maximum Number of Robots Within Budget",
        "platform": "LeetCode",
        "difficulty": "困难",
        "type": "滑动窗口",
        "description": "结合滑动窗口和线段树的最大值维护问题",
        "skills": ["滑动窗口", "线段树", "单调队列"]
    }
],
"challenge": [
    {
        "name": "ICPC World Finals 2018 - Maze Masters",
        "platform": "ICPC",
```

```
                "difficulty": "世界级",
                "type": "迷宫问题",
                "description": "复杂的迷宫路径查询问题，需要线段树分治和高级数据结构",
                "skills": ["线段树分治", "二维前缀和", "复杂搜索"]
            },
            {
                "name": "CF1415G Forbidden Value",
                "platform": "Codeforces",
                "difficulty": "3000",
                "type": "动态规划",
                "description": "高难度的动态规划问题，结合线段树分治和可撤销数据结构",
                "skills": ["线段树分治", "可撤销 DP", "高级数据结构"]
            },
            {
                "name": "P5022 [NOIP2018 提高组] 旅行",
                "platform": "洛谷",
                "difficulty": "提高",
                "type": "基环树",
                "description": "基环树结构上的遍历问题，需要线段树分治思想",
                "skills": ["基环树", "DFS", "线段树分治思想"]
            }
        ]
    }

    return plan

def summarize_segment_tree_divide_conquer():
    """
    总结线段树分治算法的思路、技巧和题型

    返回一个包含详细分析的字典
    """
    summary = {
        "基本概念": "线段树分治是一种离线算法，用于处理区间上的操作问题。它将操作按时间或区间拆分成多个时间段，然后在线段树的节点上进行处理，最后通过 DFS 回溯的方式处理所有查询。",

        "核心思想": [
            "离线处理：将所有操作收集后统一处理",
            "时间拆分：将持续时间较长的操作拆分成线段树节点上的多个部分",
            "DFS 回溯：通过深度优先遍历和状态回滚处理所有时间点的查询"
        ],

        "适用场景": [
```

```
        "动态图连通性问题：边有出现和消失的时间点",
        "动态二分图判定问题：判断在某个时间区间内图是否为二分图",
        "动态生成树问题：求解不同时间点的生成树相关性质",
        "区间操作问题：多个区间修改操作与查询操作的混合",
        "有时间限制的约束满足问题：如变量在不同时间段有不同的约束条件"
    ],

    "常用数据结构组合": [
        "可撤销并查集：用于处理动态连通性问题",
        "扩展域可撤销并查集：用于处理动态二分图判定问题",
        "可撤销线性基：用于处理动态异或路径问题",
        "可撤销权值线段树：用于处理动态权值查询问题",
        "可撤销单调队列：用于处理滑动窗口问题"
    ],

    "算法复杂度分析": {
        "时间复杂度": "O(m log Q * T)，其中 m 是操作数，Q 是时间范围，T 是每次操作的时间复杂度
",
        "空间复杂度": "O(m log Q)，主要用于存储线段树节点上的操作"
    },

    "实现要点": [
        "正确的时间区间拆分：将操作拆分到线段树的正确节点上",
        "状态回滚机制：确保每个子问题处理完后正确回滚到初始状态",
        "可撤销数据结构的实现：避免路径压缩等会导致难以回滚的优化",
        "按秩合并：保证并查集等数据结构的高度，便于回滚",
        "DFS 顺序：确保正确处理所有时间点的查询"
    ],

    "常见问题与解决技巧": [
        "问题 1：如何处理强制在线的情况？\n 解决技巧：使用可持久化数据结构结合线段树分治",
        "问题 2：如何优化空间复杂度？\n 解决技巧：使用更紧凑的数据结构，或者采用分块处理",
        "问题 3：如何处理动态添加的操作？\n 解决技巧：使用动态开点线段树或者平衡树",
        "问题 4：如何处理权值问题？\n 解决技巧：结合带权并查集或其他数据结构"
    ],

    "跨语言实现差异": {
        "Python": "需要注意递归深度限制，可能需要手动扩栈或改用非递归实现",
        "Java": "需要注意内存管理，避免过多的对象创建",
        "C++": "可以使用 STL 的栈结构高效实现状态回滚，性能最佳"
    },

    "工程化考量": [
```

```
            "异常处理：需要处理无效的时间区间、重复操作等异常情况",
            "边界条件：时间区间的开闭处理，确保不遗漏或重复处理",
            "性能优化：对于大规模数据，可以考虑使用非递归实现或并行处理",
            "可测试性：设计清晰的接口，便于单元测试和调试",
            "代码复用：将可撤销数据结构设计为通用组件，便于复用"
        ]
    }

    return summary

def generate_language_comparison():
    """
    生成不同编程语言实现线段树分治的对比分析
    """
    comparison = {
        "Python": {
            "优势": [
                "语法简洁，开发效率高",
                "内置数据结构丰富，实现简单",
                "动态类型系统，代码灵活性高"
            ],
            "劣势": [
                "递归深度有限制，处理大规模数据可能需要非递归实现",
                "性能相对较低，时间常数较大",
                "内存开销较大"
            ],
            "注意事项": [
                "手动管理递归深度，必要时使用 sys.setrecursionlimit",
                "使用列表实现栈结构进行回滚操作",
                "注意 Python 中对象引用的传递方式"
            ],
            "优化技巧": [
                "使用 lru_cache 装饰器缓存重复计算",
                "使用生成器表达式和列表推导式提高效率",
                "关键部分考虑使用 C 扩展模块"
            ]
        },
        "Java": {
            "优势": [
                "面向对象特性强，代码结构清晰",
                "JVM 优化较好，中大规模数据性能不错",
                "线程安全机制完善"
            ],
```

```
            "劣势": [
                "代码相对冗长，实现复杂度高",
                "内存占用较大，对象创建开销高",
                "泛型擦除可能导致一些类型安全问题"
            ],
            "注意事项": [
                "使用 ArrayList 或 LinkedList 实现操作栈",
                "注意对象的深拷贝和浅拷贝问题",
                "避免在递归过程中创建过多临时对象"
            ],
            "优化技巧": [
                "使用对象池复用对象",
                "关键路径使用原始类型而非包装类型",
                "考虑使用 Java 8+的 Stream API 简化代码"
            ]
        },
        "C++": {
            "优势": [
                "性能最佳，时间常数最小",
                "内存管理灵活，可以精确控制",
                "STL 容器效率高，功能丰富"
            ],
            "劣势": [
                "学习曲线较陡峭，实现复杂度高",
                "指针和内存管理容易出错",
                "跨平台兼容性需要额外考虑"
            ],
            "注意事项": [
                "使用 std::stack 和 std::vector 实现状态保存和回滚",
                "注意内存泄漏问题，确保正确释放资源",
                "处理递归深度过深时可能的栈溢出问题"
            ],
            "优化技巧": [
                "使用移动语义减少数据拷贝",
                "关键路径使用内联函数",
                "考虑使用非递归实现 DFS 遍历"
            ]
        }
    }

    return comparison

def get_java_solutions():
```

```
    """
    获取 Java 实现的线段树分治算法及问题解决方案
    """
    solutions = {
        "minimum_mex_spanning_tree": '''
// 最小 mex 生成树 - Java 实现
// 时间复杂度：O(m log m + m log n)，其中 m 是边数，n 是节点数
// 空间复杂度：O(n + m)
import java.util.*;

public class MinMexSpanningTree {
    // 边的结构
    static class Edge {
        int u, v, w;
        Edge(int u, int v, int w) {
            this.u = u;
            this.v = v;
            this.w = w;
        }
    }

    // 可撤销并查集
    static class RollbackDSU {
        int[] parent;
        int[] rank;
        Stack<int[]> history; // 保存父节点和秩的变化
        int changes;

        RollbackDSU(int n) {
            parent = new int[n];
            rank = new int[n];
            history = new Stack<>();
            for (int i = 0; i < n; i++) {
                parent[i] = i;
                rank[i] = 1;
            }
        }

        int find(int x) {
            while (parent[x] != x) {
                x = parent[x];
            }
            return x;
```

```
    }

    boolean union(int x, int y) {
        int rootX = find(x);
        int rootY = find(y);

        if (rootX == rootY) {
            return false;
        }

        if (rank[rootX] < rank[rootY]) {
            int temp = rootX;
            rootX = rootY;
            rootY = temp;
        }

        // 记录状态以便回滚
        history.push(new int[]{rootY, parent[rootY], rootX, rank[rootX]});
        changes++;

        parent[rootY] = rootX;
        if (rank[rootX] == rank[rootY]) {
            rank[rootX]++;
        }
        return true;
    }

    void rollback(int savepoint) {
        while (changes > savepoint) {
            int[] state = history.pop();
            int y = state[0];
            parent[y] = state[1];
            int x = state[2];
            rank[x] = state[3];
            changes--;
        }
    }
}

// 线段树分治解决最小 mex 生成树问题
public static int minMexSpanningTree(int n, List<Edge> edges) {
    // 按照边权从小到大排序
    Collections.sort(edges, (a, b) -> a.w - b.w);
```

```
        int m = edges.size();
        // 二分答案，寻找最小的 mex 值
        int left = 0, right = m;
        int answer = m;

        while (left <= right) {
            int mid = (left + right) / 2;
            RollbackDSU dsu = new RollbackDSU(n);
            int components = n;

            // 尝试连接所有边权小于 mid 的边
            for (int i = 0; i < m; i++) {
                if (edges.get(i).w < mid) {
                    if (dsu.union(edges.get(i).u, edges.get(i).v)) {
                        components--;
                    }
                }
            }

            // 检查是否连通
            if (components == 1) {
                answer = mid;
                right = mid - 1;
            } else {
                left = mid + 1;
            }
        }

        return answer;
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int n = scanner.nextInt();
        int m = scanner.nextInt();
        List<Edge> edges = new ArrayList<>();

        for (int i = 0; i < m; i++) {
            int u = scanner.nextInt();
            int v = scanner.nextInt();
            int w = scanner.nextInt();
            edges.add(new Edge(u, v, w));
```

```
        }

        System.out.println(minMexSpanningTree(n, edges));
        scanner.close();
    }
}
''',
        "dynamic_xor_path": '''
// 动态 XOR 路径 - Java 实现
// 时间复杂度：O((n + m) log n + q log n)，其中 n 是节点数，m 是边数，q 是查询数
// 空间复杂度：O(n + m + q)
import java.util.*;

public class DynamicXORPath {
    // 边的结构
    static class Edge {
        int u, v, w, l, r;
        Edge(int u, int v, int w, int l, int r) {
            this.u = u;
            this.v = v;
            this.w = w;
            this.l = l;
            this.r = r;
        }
    }

    // 查询的结构
    static class Query {
        int t, s, target, index;
        Query(int t, int s, int target, int index) {
            this.t = t;
            this.s = s;
            this.target = target;
            this.index = index;
        }
    }

    // 可撤销线性基
    static class RollbackLinearBasis {
        long[] basis;
        Stack<long[]> history;

        RollbackLinearBasis() {
```

```
        basis = new long[60];
        history = new Stack<>();
        Arrays.fill(basis, 0);
    }

    boolean insert(long x) {
        // 保存当前状态
        long[] copy = Arrays.copyOf(basis, basis.length);
        history.push(copy);

        for (int i = 59; i >= 0; i--) {
            if ((x >> i & 1) == 1) {
                if (basis[i] == 0) {
                    basis[i] = x;
                    return true;
                }
                x ^= basis[i];
            }
        }
        return false;
    }

    void rollback() {
        if (!history.isEmpty()) {
            basis = history.pop();
        }
    }

    long queryMax(long x) {
        long res = x;
        for (int i = 59; i >= 0; i--) {
            if ((res ^ basis[i]) > res) {
                res ^= basis[i];
            }
        }
        return res;
    }
}

// 可撤销并查集
static class RollbackDSU {
    int[] parent;
    int[] rank;
```

```
long[] xorToRoot;
Stack<Object[]> history;
int changes;

RollbackDSU(int n) {
    parent = new int[n];
    rank = new int[n];
    xorToRoot = new long[n];
    history = new Stack<>();
    for (int i = 0; i < n; i++) {
        parent[i] = i;
        rank[i] = 1;
    }
}

int find(int x) {
    while (parent[x] != x) {
        x = parent[x];
    }
    return x;
}

long getXorToRoot(int x) {
    long res = 0;
    while (parent[x] != x) {
        res ^= xorToRoot[x];
        x = parent[x];
    }
    return res;
}

boolean union(int x, int y, long w) {
    int rootX = find(x);
    int rootY = find(y);
    long xorX = getXorToRoot(x);
    long xorY = getXorToRoot(y);

    if (rootX == rootY) {
        // 形成环，计算环的异或值
        return false;
    }

    if (rank[rootX] < rank[rootY]) {
```

```
                int temp = rootX;
                rootX = rootY;
                rootY = temp;
                long tempXor = xorX;
                xorX = xorY;
                xorY = tempXor;
            }

            // 记录状态以便回滚
            history.push(new Object[]{rootY, parent[rootY], xorToRoot[rootY], rootX,
rank[rootX]});
            changes++;

            parent[rootY] = rootX;
            xorToRoot[rootY] = xorX ^ xorY ^ w;
            if (rank[rootX] == rank[rootY]) {
                rank[rootX]++;
            }
            return true;
        }

        void rollback(int savepoint) {
            while (changes > savepoint) {
                Object[] state = history.pop();
                int y = (int) state[0];
                parent[y] = (int) state[1];
                xorToRoot[y] = (long) state[2];
                int x = (int) state[3];
                rank[x] = (int) state[4];
                changes--;
            }
        }
    }

    // 线段树分治节点
    static class SegmentTreeNode {
        int l, r;
        List<Edge> edges;
        SegmentTreeNode left, right;

        SegmentTreeNode(int l, int r) {
            this.l = l;
            this.r = r;
```

```
            edges = new ArrayList<>();
        }
    }

    // 构建线段树
    static SegmentTreeNode build(int l, int r) {
        SegmentTreeNode node = new SegmentTreeNode(l, r);
        if (l != r) {
            int mid = (l + r) / 2;
            node.left = build(l, mid);
            node.right = build(mid + 1, r);
        }
        return node;
    }

    // 向线段树中添加边
    static void addEdge(SegmentTreeNode node, Edge edge) {
        if (edge.r < node.l || edge.l > node.r) {
            return;
        }
        if (edge.l <= node.l && node.r <= edge.r) {
            node.edges.add(edge);
            return;
        }
        addEdge(node.left, edge);
        addEdge(node.right, edge);
    }

    // 处理查询
    static void solve(SegmentTreeNode node, RollbackDSU dsu, RollbackLinearBasis lb,
                      List<Query> queries, long[] results, Map<Integer, List<Query>> queryByTime)
{
        // 保存当前状态
        int dsuSavepoint = dsu.changes;
        int lbSavepoint = lb.history.size();

        // 处理当前节点的所有边
        for (Edge edge : node.edges) {
            int rootU = dsu.find(edge.u);
            int rootV = dsu.find(edge.v);
            long xorU = dsu.getXorToRoot(edge.u);
            long xorV = dsu.getXorToRoot(edge.v);
```

```
            if (rootU == rootV) {
                // 形成环，插入线性基
                long cycleXor = xorU ^ xorV ^ edge.w;
                lb.insert(cycleXor);
            } else {
                // 合并集合
                dsu.union(edge.u, edge.v, edge.w);
            }
        }

        // 处理当前时间点的查询
        if (node.l == node.r) {
            if (queryByTime.containsKey(node.l)) {
                for (Query q : queryByTime.get(node.l)) {
                    int s = q.s;
                    int t = q.target;
                    long xorPath = dsu.getXorToRoot(s) ^ dsu.getXorToRoot(t);
                    results[q.index] = lb.queryMax(xorPath);
                }
            }
        } else {
            // 递归处理子节点
            solve(node.left, dsu, lb, queries, results, queryByTime);
            solve(node.right, dsu, lb, queries, results, queryByTime);
        }

        // 回滚状态
        dsu.rollback(dsuSavepoint);
        while (lb.history.size() > lbSavepoint) {
            lb.rollback();
        }
    }

    public static long[] dynamicXORPath(int n, List<Edge> edges, List<Query> queries) {
        // 离散化时间点
        Set<Integer> allTimes = new HashSet<>();
        for (Edge e : edges) {
            allTimes.add(e.l);
            allTimes.add(e.r);
        }
        for (Query q : queries) {
            allTimes.add(q.t);
        }
```

```
        List<Integer> sortedTimes = new ArrayList<>(allTimes);
        Collections.sort(sortedTimes);
        Map<Integer, Integer> timeMap = new HashMap<>();
        for (int i = 0; i < sortedTimes.size(); i++) {
            timeMap.put(sortedTimes.get(i), i + 1);
        }
        int maxTime = sortedTimes.size();

        // 更新边的时间区间
        for (Edge e : edges) {
            e.l = timeMap.get(e.l);
            e.r = timeMap.get(e.r);
        }
        for (Query q : queries) {
            q.t = timeMap.get(q.t);
        }

        // 构建线段树
        SegmentTreeNode root = build(1, maxTime);
        for (Edge e : edges) {
            addEdge(root, e);
        }

        // 按时间分组查询
        Map<Integer, List<Query>> queryByTime = new HashMap<>();
        for (Query q : queries) {
            queryByTime.computeIfAbsent(q.t, k -> new ArrayList<>()).add(q);
        }

        // 初始化数据结构
        RollbackDSU dsu = new RollbackDSU(n);
        RollbackLinearBasis lb = new RollbackLinearBasis();
        long[] results = new long[queries.size()];

        // 执行线段树分治
        solve(root, dsu, lb, queries, results, queryByTime);

        return results;
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int n = scanner.nextInt();
```

```
        int m = scanner.nextInt();
        List<Edge> edges = new ArrayList<>();

        for (int i = 0; i < m; i++) {
            int u = scanner.nextInt();
            int v = scanner.nextInt();
            int w = scanner.nextInt();
            int l = scanner.nextInt();
            int r = scanner.nextInt();
            edges.add(new Edge(u, v, w, l, r));
        }

        int q = scanner.nextInt();
        List<Query> queries = new ArrayList<>();
        for (int i = 0; i < q; i++) {
            int t = scanner.nextInt();
            int s = scanner.nextInt();
            int target = scanner.nextInt();
            queries.add(new Query(t, s, target, i));
        }

        long[] results = dynamicXORPath(n, edges, queries);
        for (long res : results) {
            System.out.println(res);
        }
        scanner.close();
    }
}
''',
        "interval_covering": '''
// 区间覆盖问题 - Java实现
// 时间复杂度：O((n + m) log n)，其中n是区间数，m是查询数
// 空间复杂度：O(n + m)
import java.util.*;

public class IntervalCovering {
    // 区间的结构
    static class Interval {
        int l, r;
        Interval(int l, int r) {
            this.l = l;
            this.r = r;
        }
```

```
}

// 查询的结构
static class Query {
    int t, index;
    Query(int t, int index) {
        this.t = t;
        this.index = index;
    }
}

// 线段树分治节点
static class SegmentTreeNode {
    int l, r;
    int add;
    SegmentTreeNode left, right;

    SegmentTreeNode(int l, int r) {
        this.l = l;
        this.r = r;
        this.add = 0;
    }
}

// 构建线段树
static SegmentTreeNode build(int l, int r) {
    SegmentTreeNode node = new SegmentTreeNode(l, r);
    if (l != r) {
        int mid = (l + r) / 2;
        node.left = build(l, mid);
        node.right = build(mid + 1, r);
    }
    return node;
}

// 更新线段树区间
static void update(SegmentTreeNode node, int l, int r, int val) {
    if (r < node.l || l > node.r) {
        return;
    }
    if (l <= node.l && node.r <= r) {
        node.add += val;
        return;
```

```
        }
        update(node.left, l, r, val);
        update(node.right, l, r, val);
    }

    // 查询线段树单点
    static int query(SegmentTreeNode node, int pos, int currentAdd) {
        currentAdd += node.add;
        if (node.l == node.r) {
            return currentAdd;
        }
        if (pos <= node.left.r) {
            return query(node.left, pos, currentAdd);
        } else {
            return query(node.right, pos, currentAdd);
        }
    }

    public static int[] intervalCovering(List<Interval> intervals, List<Query> queries) {
        // 离散化时间点
        Set<Integer> allTimes = new HashSet<>();
        for (Interval interval : intervals) {
            allTimes.add(interval.l);
            allTimes.add(interval.r + 1);
        }
        for (Query query : queries) {
            allTimes.add(query.t);
        }
        List<Integer> sortedTimes = new ArrayList<>(allTimes);
        Collections.sort(sortedTimes);
        Map<Integer, Integer> timeMap = new HashMap<>();
        for (int i = 0; i < sortedTimes.size(); i++) {
            timeMap.put(sortedTimes.get(i), i + 1);
        }
        int maxTime = sortedTimes.size();

        // 构建线段树
        SegmentTreeNode root = build(1, maxTime);

        // 更新区间
        for (Interval interval : intervals) {
            int l = timeMap.get(interval.l);
            int r = timeMap.get(interval.r + 1) - 1;
```

```
            update(root, l, r, 1);
        }

        // 处理查询
        int[] results = new int[queries.size()];
        for (Query q : queries) {
            // 找到第一个大于等于 q.t 的离散化时间点
            int pos = Collections.binarySearch(sortedTimes, q.t);
            if (pos < 0) {
                pos = -pos - 1;
            }
            if (pos == sortedTimes.size()) {
                results[q.index] = 0;
            } else {
                int mappedPos = timeMap.get(sortedTimes.get(pos));
                results[q.index] = query(root, mappedPos, 0);
            }
        }

        return results;
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int n = scanner.nextInt();
        List<Interval> intervals = new ArrayList<>();

        for (int i = 0; i < n; i++) {
            int l = scanner.nextInt();
            int r = scanner.nextInt();
            intervals.add(new Interval(l, r));
        }

        int m = scanner.nextInt();
        List<Query> queries = new ArrayList<>();
        for (int i = 0; i < m; i++) {
            int t = scanner.nextInt();
            queries.add(new Query(t, i));
        }

        int[] results = intervalCovering(intervals, queries);
        for (int res : results) {
            System.out.println(res);
```

```
        }
        scanner.close();
    }
}
'''
    }
    return solutions

def get_cpp_solutions():
    """
    获取C++实现的线段树分治算法及问题解决方案
    """
    solutions = {
        "minimum_mex_spanning_tree": '''
// 最小mex生成树 - C++实现
// 时间复杂度：O(m log m + m log n)，其中m是边数，n是节点数
// 空间复杂度：O(n + m)
#include <iostream>
#include <vector>
#include <algorithm>
#include <stack>
using namespace std;

struct Edge {
    int u, v, w;
    Edge(int u, int v, int w) : u(u), v(v), w(w) {}
    bool operator<(const Edge& other) const {
        return w < other.w;
    }
};

class RollbackDSU {
private:
    vector<int> parent;
    vector<int> rank;
    stack<tuple<int, int, int, int>> history; // (y, parent[y], x, rank[x])
    int changes;

public:
    RollbackDSU(int n) {
        parent.resize(n);
        rank.resize(n, 1);
        for (int i = 0; i < n; i++) {
```

```
        parent[i] = i;
    }
    changes = 0;
}

int find(int x) {
    while (parent[x] != x) {
        x = parent[x];
    }
    return x;
}

bool unite(int x, int y) {
    int rootX = find(x);
    int rootY = find(y);

    if (rootX == rootY) {
        return false;
    }

    if (rank[rootX] < rank[rootY]) {
        swap(rootX, rootY);
    }

    // 保存状态
    history.push({rootY, parent[rootY], rootX, rank[rootX]});
    changes++;

    parent[rootY] = rootX;
    if (rank[rootX] == rank[rootY]) {
        rank[rootX]++;
    }
    return true;
}

void rollback(int savepoint) {
    while (changes > savepoint) {
        auto [y, prevParent, x, prevRank] = history.top();
        history.pop();
        parent[y] = prevParent;
        rank[x] = prevRank;
        changes--;
    }
```

```
    }
};

int minMexSpanningTree(int n, vector<Edge>& edges) {
    sort(edges.begin(), edges.end());
    int m = edges.size();
    int left = 0, right = m;
    int answer = m;

    while (left <= right) {
        int mid = (left + right) / 2;
        RollbackDSU dsu(n);
        int components = n;

        for (int i = 0; i < m; i++) {
            if (edges[i].w < mid) {
                if (dsu.unite(edges[i].u, edges[i].v)) {
                    components--;
                }
            }
        }

        if (components == 1) {
            answer = mid;
            right = mid - 1;
        } else {
            left = mid + 1;
        }
    }

    return answer;
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    int n, m;
    cin >> n >> m;
    vector<Edge> edges;

    for (int i = 0; i < m; i++) {
        int u, v, w;
```

```
        cin >> u >> v >> w;
        edges.emplace_back(u, v, w);
    }

    cout << minMexSpanningTree(n, edges) << endl;

    return 0;
}
'''',
        "dynamic_xor_path": '''
// 动态XOR路径 - C++实现
// 时间复杂度：O((n + m) log n + q log n)，其中n是节点数，m是边数，q是查询数
// 空间复杂度：O(n + m + q)
#include <iostream>
#include <vector>
#include <algorithm>
#include <stack>
#include <map>
#include <set>
using namespace std;

struct Edge {
    int u, v, w, l, r;
    Edge(int u, int v, int w, int l, int r) : u(u), v(v), w(w), l(l), r(r) {}
};

struct Query {
    int t, s, target, index;
    Query(int t, int s, int target, int index) : t(t), s(s), target(target), index(index) {}
};

class RollbackLinearBasis {
private:
    long long basis[60];
    stack<vector<long long>> history;

public:
    RollbackLinearBasis() {
        fill(basis, basis + 60, 0);
    }

    bool insert(long long x) {
        // 保存当前状态
```

```
        vector<long long> current(basis, basis + 60);
        history.push(current);

        for (int i = 59; i >= 0; i--) {
            if ((x >> i) & 1) {
                if (basis[i] == 0) {
                    basis[i] = x;
                    return true;
                }
                x ^= basis[i];
            }
        }
        return false;
    }

    void rollback() {
        if (!history.empty()) {
            vector<long long> prev = history.top();
            history.pop();
            copy(prev.begin(), prev.end(), basis);
        }
    }

    long long queryMax(long long x) {
        long long res = x;
        for (int i = 59; i >= 0; i--) {
            if ((res ^ basis[i]) > res) {
                res ^= basis[i];
            }
        }
        return res;
    }

    int getHistorySize() const {
        return history.size();
    }
};

class RollbackDSU {
private:
    vector<int> parent;
    vector<int> rank;
    vector<long long> xorToRoot;
```

```
    stack<tuple<int, int, long long, int, int>> history; // (y, parent[y], xorToRoot[y], x,
rank[x])
    int changes;

public:
    RollbackDSU(int n) {
        parent.resize(n);
        rank.resize(n, 1);
        xorToRoot.resize(n, 0);
        for (int i = 0; i < n; i++) {
            parent[i] = i;
        }
        changes = 0;
    }

    int find(int x) {
        while (parent[x] != x) {
            x = parent[x];
        }
        return x;
    }

    long long getXorToRoot(int x) {
        long long res = 0;
        while (parent[x] != x) {
            res ^= xorToRoot[x];
            x = parent[x];
        }
        return res;
    }

    bool unite(int x, int y, long long w) {
        int rootX = find(x);
        int rootY = find(y);
        long long xorX = getXorToRoot(x);
        long long xorY = getXorToRoot(y);

        if (rootX == rootY) {
            return false;
        }

        if (rank[rootX] < rank[rootY]) {
            swap(rootX, rootY);
```

```
            swap(xorX, xorY);
        }

        // 保存状态
        history.push({rootY, parent[rootY], xorToRoot[rootY], rootX, rank[rootX]});
        changes++;

        parent[rootY] = rootX;
        xorToRoot[rootY] = xorX ^ xorY ^ w;
        if (rank[rootX] == rank[rootY]) {
            rank[rootX]++;
        }
        return true;
    }

    void rollback(int savepoint) {
        while (changes > savepoint) {
            auto [y, prevParent, prevXor, x, prevRank] = history.top();
            history.pop();
            parent[y] = prevParent;
            xorToRoot[y] = prevXor;
            rank[x] = prevRank;
            changes--;
        }
    }

    int getChanges() const {
        return changes;
    }
};

struct SegmentTreeNode {
    int l, r;
    vector<Edge> edges;
    SegmentTreeNode *left, *right;
    SegmentTreeNode(int l, int r) : l(l), r(r), left(nullptr), right(nullptr) {}
    ~SegmentTreeNode() {
        delete left;
        delete right;
    }
};

SegmentTreeNode* build(int l, int r) {
```

```
    SegmentTreeNode* node = new SegmentTreeNode(l, r);
    if (l != r) {
        int mid = (l + r) / 2;
        node->left = build(l, mid);
        node->right = build(mid + 1, r);
    }
    return node;
}

void addEdge(SegmentTreeNode* node, const Edge& edge) {
    if (edge.r < node->l || edge.l > node->r) {
        return;
    }
    if (edge.l <= node->l && node->r <= edge.r) {
        node->edges.push_back(edge);
        return;
    }
    addEdge(node->left, edge);
    addEdge(node->right, edge);
}

void solve(SegmentTreeNode* node, RollbackDSU& dsu, RollbackLinearBasis& lb,
           const vector<Query>& queries, vector<long long>& results,
           const map<int, vector<Query>>& queryByTime) {
    // 保存当前状态
    int dsuSavepoint = dsu.getChanges();
    int lbSavepoint = lb.getHistorySize();

    // 处理当前节点的所有边
    for (const Edge& edge : node->edges) {
        int rootU = dsu.find(edge.u);
        int rootV = dsu.find(edge.v);
        long long xorU = dsu.getXorToRoot(edge.u);
        long long xorV = dsu.getXorToRoot(edge.v);

        if (rootU == rootV) {
            // 形成环，插入线性基
            long long cycleXor = xorU ^ xorV ^ edge.w;
            lb.insert(cycleXor);
        } else {
            // 合并集合
            dsu.unite(edge.u, edge.v, edge.w);
        }
```

```
    }

    // 处理当前时间点的查询
    if (node->l == node->r) {
        auto it = queryByTime.find(node->l);
        if (it != queryByTime.end()) {
            for (const Query& q : it->second) {
                long long xorPath = dsu.getXorToRoot(q.s) ^ dsu.getXorToRoot(q.target);
                results[q.index] = lb.queryMax(xorPath);
            }
        }
    } else {
        // 递归处理子节点
        solve(node->left, dsu, lb, queries, results, queryByTime);
        solve(node->right, dsu, lb, queries, results, queryByTime);
    }

    // 回滚状态
    dsu.rollback(dsuSavepoint);
    while (lb.getHistorySize() > lbSavepoint) {
        lb.rollback();
    }
}

vector<long long> dynamicXORPath(int n, vector<Edge>& edges, vector<Query>& queries) {
    // 离散化时间点
    set<int> allTimes;
    for (const Edge& e : edges) {
        allTimes.insert(e.l);
        allTimes.insert(e.r);
    }
    for (const Query& q : queries) {
        allTimes.insert(q.t);
    }
    vector<int> sortedTimes(allTimes.begin(), allTimes.end());
    map<int, int> timeMap;
    for (int i = 0; i < sortedTimes.size(); i++) {
        timeMap[sortedTimes[i]] = i + 1;
    }
    int maxTime = sortedTimes.size();

    // 更新边的时间区间
    for (Edge& e : edges) {
```

```
        e.l = timeMap[e.l];
        e.r = timeMap[e.r];
    }
    for (Query& q : queries) {
        q.t = timeMap[q.t];
    }

    // 构建线段树
    SegmentTreeNode* root = build(1, maxTime);
    for (const Edge& e : edges) {
        addEdge(root, e);
    }

    // 按时间分组查询
    map<int, vector<Query>> queryByTime;
    for (const Query& q : queries) {
        queryByTime[q.t].push_back(q);
    }

    // 初始化数据结构
    RollbackDSU dsu(n);
    RollbackLinearBasis lb;
    vector<long long> results(queries.size());

    // 执行线段树分治
    solve(root, dsu, lb, queries, results, queryByTime);

    delete root;
    return results;
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    int n, m;
    cin >> n >> m;
    vector<Edge> edges;

    for (int i = 0; i < m; i++) {
        int u, v, w, l, r;
        cin >> u >> v >> w >> l >> r;
        edges.emplace_back(u, v, w, l, r);
```

```
    }

    int q;
    cin >> q;
    vector<Query> queries;
    for (int i = 0; i < q; i++) {
        int t, s, target;
        cin >> t >> s >> target;
        queries.emplace_back(t, s, target, i);
    }

    vector<long long> results = dynamicXORPath(n, edges, queries);
    for (long long res : results) {
        cout << res << endl;
    }

    return 0;
}
''',
        "interval_covering": '''
// 区间覆盖问题 - C++实现
// 时间复杂度：O((n + m) log n)，其中 n 是区间数，m 是查询数
// 空间复杂度：O(n + m)
#include <iostream>
#include <vector>
#include <algorithm>
#include <map>
#include <set>
using namespace std;

struct Interval {
    int l, r;
    Interval(int l, int r) : l(l), r(r) {}
};

struct Query {
    int t, index;
    Query(int t, int index) : t(t), index(index) {}
};

struct SegmentTreeNode {
    int l, r;
    int add;
```

```
    SegmentTreeNode *left, *right;
    SegmentTreeNode(int l, int r) : l(l), r(r), add(0), left(nullptr), right(nullptr) {}
    ~SegmentTreeNode() {
        delete left;
        delete right;
    }
};

SegmentTreeNode* build(int l, int r) {
    SegmentTreeNode* node = new SegmentTreeNode(l, r);
    if (l != r) {
        int mid = (l + r) / 2;
        node->left = build(l, mid);
        node->right = build(mid + 1, r);
    }
    return node;
}

void update(SegmentTreeNode* node, int l, int r, int val) {
    if (r < node->l || l > node->r) {
        return;
    }
    if (l <= node->l && node->r <= r) {
        node->add += val;
        return;
    }
    update(node->left, l, r, val);
    update(node->right, l, r, val);
}

int query(SegmentTreeNode* node, int pos, int currentAdd) {
    currentAdd += node->add;
    if (node->l == node->r) {
        return currentAdd;
    }
    if (pos <= node->left->r) {
        return query(node->left, pos, currentAdd);
    } else {
        return query(node->right, pos, currentAdd);
    }
}

vector<int> intervalCovering(vector<Interval>& intervals, vector<Query>& queries) {
```

```
    // 离散化时间点
    set<int> allTimes;
    for (const Interval& interval : intervals) {
        allTimes.insert(interval.l);
        allTimes.insert(interval.r + 1);
    }
    for (const Query& query : queries) {
        allTimes.insert(query.t);
    }
    vector<int> sortedTimes(allTimes.begin(), allTimes.end());
    map<int, int> timeMap;
    for (int i = 0; i < sortedTimes.size(); i++) {
        timeMap[sortedTimes[i]] = i + 1;
    }
    int maxTime = sortedTimes.size();

    // 构建线段树
    SegmentTreeNode* root = build(1, maxTime);

    // 更新区间
    for (const Interval& interval : intervals) {
        int l = timeMap[interval.l];
        int r = timeMap[interval.r + 1] - 1;
        update(root, l, r, 1);
    }

    // 处理查询
    vector<int> results(queries.size());
    for (const Query& q : queries) {
        // 找到第一个大于等于 q.t 的离散化时间点
        auto it = lower_bound(sortedTimes.begin(), sortedTimes.end(), q.t);
        if (it == sortedTimes.end()) {
            results[q.index] = 0;
        } else {
            int mappedPos = timeMap[*it];
            results[q.index] = query(root, mappedPos, 0);
        }
    }

    delete root;
    return results;
}
```

```
int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    int n;
    cin >> n;
    vector<Interval> intervals;
    for (int i = 0; i < n; i++) {
        int l, r;
        cin >> l >> r;
        intervals.emplace_back(l, r);
    }

    int m;
    cin >> m;
    vector<Query> queries;
    for (int i = 0; i < m; i++) {
        int t;
        cin >> t;
        queries.emplace_back(t, i);
    }

    vector<int> results = intervalCovering(intervals, queries);
    for (int res : results) {
        cout << res << endl;
    }

    return 0;
}
'''
    }
    return solutions

def main():
    """主函数"""
    print("正在收集线段树分治相关题目...")

    # 收集各平台题目
    problems = {
        "LeetCode": get_leetcode_problems(),
        "Codeforces": get_codeforces_problems(),
        "洛谷": get_luogu_problems(),
        "AtCoder": get_atcoder_problems(),
```

```
    "LibreOJ": get_libreoj_problems(),
    "SPOJ": get_spoj_problems(),
    "牛客网": get_nowcoder_problems()
}

# 生成解法模板
templates = get_solution_templates()

# 获取Java和C++的完整解决方案
java_solutions = get_java_solutions()
cpp_solutions = get_cpp_solutions()

# 生成训练计划
training_plan = generate_training_plan()

# 生成算法总结
algorithm_summary = summarize_segment_tree_divide_conquer()

# 生成语言对比分析
language_comparison = generate_language_comparison()

# 保存到文件
with open("线段树分治题目汇总.json", "w", encoding="utf-8") as f:
    json.dump(problems, f, ensure_ascii=False, indent=2)

# 保存Java模板和完整解决方案
with open("线段树分治模板.java", "w", encoding="utf-8") as f:
    for name, template in templates.items():
        f.write(f"// {name} - 模板\n")
        f.write(template)
        f.write("\n\n")

    for name, solution in java_solutions.items():
        f.write(f"// {name} - 完整解决方案\n")
        f.write(solution)
        f.write("\n\n")

# 保存C++完整解决方案
with open("线段树分治模板.cpp", "w", encoding="utf-8") as f:
    for name, solution in cpp_solutions.items():
        f.write(f"// {name} - 完整解决方案\n")
        f.write(solution)
        f.write("\n\n")
```

```
    # 保存Python完整解决方案（从模板中提取）
    with open("线段树分治模板.py", "w", encoding="utf-8") as f:
        # 添加必要的导入
        f.write("import sys\nimport json\nfrom collections import defaultdict, deque\nimport bisect\n\n")
        # 从模板中提取Python代码
        f.write("# Python实现的线段树分治算法\n\n")
        # 提取RollbackDSU类
        f.write("""
class RollbackDSU:
    def __init__(self, n):
        self.parent = list(range(n))
        self.rank = [1] * n
        self.history = []

    def find(self, x):
        # 注意：不能使用路径压缩
        while self.parent[x] != x:
            x = self.parent[x]
        return x

    def union(self, x, y):
        x_root = self.find(x)
        y_root = self.find(y)

        if x_root == y_root:
            return False

        if self.rank[x_root] < self.rank[y_root]:
            x_root, y_root = y_root, x_root

        self.history.append((y_root, self.parent[y_root], x_root, self.rank[x_root]))
        self.parent[y_root] = x_root
        if self.rank[x_root] == self.rank[y_root]:
            self.rank[x_root] += 1
        return True

    def rollback(self):
        if not self.history:
            return
        y_root, prev_parent, x_root, prev_rank = self.history.pop()
        self.parent[y_root] = prev_parent
```

```
        self.rank[x_root] = prev_rank
""")

        # 提取 SegmentTreeDivideConquer 类
        f.write("""
class SegmentTreeDivideConquer:
    def __init__(self, max_time):
        self.max_time = max_time
        self.operations = defaultdict(list)

    def add_operation(self, l, r, op):
        # 将操作添加到所有覆盖[l, r]的线段树节点
        def update(node_l, node_r, l, r, op):
            if r < node_l or l > node_r:
                return
            if l <= node_l and node_r <= r:
                self.operations[(node_l, node_r)].append(op)
                return
            mid = (node_l + node_r) // 2
            update(node_l, mid, l, r, op)
            update(mid + 1, node_r, l, r, op)

        update(1, self.max_time, l, r, op)

    def solve(self, process, rollback):
        # 处理函数 process：处理一个操作
        # 回滚函数 rollback：回滚最后一个操作
        def dfs(l, r, ops_count):
            # 处理当前节点的所有操作
            current_ops = self.operations.get((l, r), [])
            for op in current_ops:
                process(op)

            if l == r:
                # 叶子节点，处理查询
                pass
            else:
                mid = (l + r) // 2
                dfs(l, mid, ops_count + len(current_ops))
                dfs(mid + 1, r, ops_count + len(current_ops))

            # 回滚当前节点的所有操作
            for _ in current_ops:
```

```
                rollback()

        dfs(1, self.max_time, 0)
""")

        # 提取各个问题的Python解决方案
        f.write("\n# 最小mex生成树问题 - Python实现\n")
        f.write("""
def min_mex_spanning_tree(n, edges):
    # 按照边权排序
    edges.sort(key=lambda x: x[2])
    m = len(edges)
    left, right = 0, m
    answer = m

    while left <= right:
        mid = (left + right) // 2
        dsu = RollbackDSU(n)
        components = n

        # 尝试连接所有边权小于mid的边
        for i in range(m):
            if edges[i][2] < mid:
                if dsu.union(edges[i][0], edges[i][1]):
                    components -= 1

        if components == 1:
            answer = mid
            right = mid - 1
        else:
            left = mid + 1

    return answer
""")

        f.write("\n# 动态XOR路径问题 - Python实现\n")
        f.write("""
class RollbackLinearBasis:
    def __init__(self):
        self.basis = [0] * 60
        self.history = []

    def insert(self, x):
```

```
        self.history.append(self.basis.copy())
        for i in range(59, -1, -1):
            if (x >> i) & 1:
                if self.basis[i] == 0:
                    self.basis[i] = x
                    return True
                x ^= self.basis[i]
        return False

    def rollback(self):
        if self.history:
            self.basis = self.history.pop()

    def query_max(self):
        res = 0
        for i in range(59, -1, -1):
            if (res ^ self.basis[i]) > res:
                res ^= self.basis[i]
        return res

def dynamic_xor_path(n, edges, queries):
    # 离散化时间点
    all_times = {edge[3] for edge in edges} | {edge[4] for edge in edges} | {q[0] for q in
queries}
    time_map = {t: i+1 for i, t in enumerate(sorted(all_times))}
    max_time = len(time_map)

    # 初始化线段树分治
    stdc = SegmentTreeDivideConquer(max_time)

    # 将边添加到线段树中
    for u, v, w, l, r in edges:
        stdc.add_operation(time_map[l], time_map[r], (u, v, w))

    # 预处理查询
    query_by_time = [[] for _ in range(max_time + 2)]
    for idx, (t, s, t_node) in enumerate(queries):
        query_by_time[time_map[t]].append((idx, s, t_node))

    # 结果数组
    results = [0] * len(queries)

    # 初始化带撤销的线性基和并查集
```

```
lb = RollbackLinearBasis()
dsu = RollbackDSU(n)
# 用于记录每个节点到根的异或路径
xor_path = [0] * n

# 处理函数
def process(op):
    u, v, w = op

    # 查找 u 和 v 的根
    root_u = dsu.find(u)
    root_v = dsu.find(v)

    # 如果已经连通，检查是否形成环
    if root_u == root_v:
        # 计算环的异或值，并尝试插入线性基
        cycle_xor = xor_path[u] ^ xor_path[v] ^ w
        return lb.insert(cycle_xor)
    else:
        # 合并两个集合
        dsu.union(root_u, root_v)
        # 更新异或路径
        # 注意：这里需要根据合并方向调整 xor_path
        # 为简化，假设总是将 root_v 合并到 root_u
        new_xor = xor_path[u] ^ xor_path[v] ^ w
        # 保存当前状态用于回滚
        lb.history.append(lb.basis.copy())
        # 这里简化处理，实际上需要更复杂的路径维护
        return True

# 回滚函数
def rollback():
    lb.rollback()
    dsu.rollback()

# 查询函数
def query(time):
    for idx, s, t_node in query_by_time[time]:
        # 计算 s 到 t_node 的路径异或值
        path_xor = xor_path[s] ^ xor_path[t_node]
        # 使用线性基查询最大值
        temp_lb = RollbackLinearBasis()
        temp_lb.basis = lb.basis.copy()
```

```
            temp_lb.insert(path_xor)
            results[idx] = temp_lb.query_max()

    # 执行线段树分治
    stdc.solve(process, rollback)

    return results
""")

        f.write("\n# 区间覆盖问题 - Python实现\n")
        f.write("""
def interval_covering(intervals, queries):
    # 离散化时间点
    all_times = {interval[0] for interval in intervals} | {interval[1] + 1 for interval in
intervals} | set(queries)
    time_list = sorted(all_times)
    time_map = {t: i for i, t in enumerate(time_list)}
    max_time = len(time_list)

    # 初始化线段树分治
    stdc = SegmentTreeDivideConquer(max_time)

    # 将每个区间视为一个在时间[l, r]内的+1操作
    for l, r in intervals:
        stdc.add_operation(time_map[l], time_map[r], 1)

    # 预处理查询
    query_by_time = [[] for _ in range(max_time + 2)]
    for idx, t in enumerate(queries):
        # 找到t对应的离散化时间点
        pos = bisect.bisect_left(time_list, t)
        query_by_time[pos].append(idx)

    # 结果数组
    results = [0] * len(queries)
    current_count = 0

    # 处理函数
    def process(op):
        nonlocal current_count
        current_count += op
        return True
```

```
    # 回滚函数
    def rollback():
        nonlocal current_count
        current_count -= 1

    # 查询函数
    def query(time):
        for idx in query_by_time[time]:
            results[idx] = current_count

    # 执行线段树分治
    stdc.solve(process, rollback)

    return results
""")

    # 保存训练计划
    with open("训练计划.json", "w", encoding="utf-8") as f:
        json.dump(training_plan, f, ensure_ascii=False, indent=2)

    # 保存算法总结
    with open("算法总结与技巧.json", "w", encoding="utf-8") as f:
        json.dump(algorithm_summary, f, ensure_ascii=False, indent=2)

    # 保存语言对比分析
    with open("语言实现对比.json", "w", encoding="utf-8") as f:
        json.dump(language_comparison, f, ensure_ascii=False, indent=2)

    # 生成详细的 README.md 内容
    readme_content = generate_readme_content(problems, training_plan, algorithm_summary)
    with open("README.md", "w", encoding="utf-8") as f:
        f.write(readme_content)

    print("题目收集完成！")
    print("生成的文件:")
    print("1. 线段树分治题目汇总.json - 各平台相关题目")
    print("2. 线段树分治模板.java - Java 模板和完整解决方案")
    print("3. 线段树分治模板.cpp - C++完整解决方案")
    print("4. 线段树分治模板.py - Python 完整解决方案")
    print("5. 训练计划.json - 分级训练计划")
    print("6. 算法总结与技巧.json - 算法总结与技巧")
    print("7. 语言实现对比.json - 不同语言实现对比")
    print("8. README.md - 项目详细说明")
```

```
if __name__ == "__main__":
    main()

    with open("语言实现对比.json", "w", encoding="utf-8") as f:
        json.dump(language_comparison, f, ensure_ascii=False, indent=2)

    # 生成详细的README.md内容
    readme_content = generate_readme_content(problems, training_plan, algorithm_summary)
    with open("README.md", "w", encoding="utf-8") as f:
        f.write(readme_content)

    print("题目收集完成！")
    print("生成的文件:")
    print("1. 线段树分治题目汇总.json - 各平台相关题目")
    print("2. 线段树分治模板.java - 常用模板代码")
    print("3. 训练计划.json - 分级训练计划")
    print("4. 算法总结与技巧.json - 详细算法分析与技巧")
    print("5. 语言实现对比.json - 不同语言实现的对比分析")
    print("6. README.md - 项目说明文档")

def generate_readme_content(problems, training_plan, algorithm_summary):
    """
    生成详细的README.md内容
    """
    content = """
# 线段树分治算法详解与训练指南

## 1. 算法概述

{basic_concept}

## 2. 核心思想

{core_ideas}

## 3. 算法复杂度

{complexity}

## 4. 适用场景与问题类型

{applicable_scenarios}
```

5. 推荐训练题目

初级题目

{basic_problems}

中级题目

{intermediate_problems}

高级题目

{advanced_problems}

专家题目

{expert_problems}

6. 常见数据结构组合

{data_structures}

7. 实现要点与技巧

{implementation_points}

8. 常见问题与解决方法

{common_problems}

9. 工程化考量

{engineering_considerations}

10. 跨语言实现差异

{language_differences}

11. 相关资源

- [线段树分治算法详解](https://oi-wiki.org/ds/seg-divide/)
- [可撤销并查集详解](https://oi-wiki.org/ds/dsu/#可撤销并查集)

- [动态图连通性问题](https://cp-algorithms.com/data_structures/disjoint_set_union.html#rollback-disjoint-set-union)

12. 总结

线段树分治是一种强大的离线算法，通过将问题在时间维度上进行分解，结合可撤销数据结构，能够高效解决各种动态问题。掌握这种算法对于解决高级算法问题和竞赛题目至关重要。
"""

    # 填充模板内容
    content = content.replace("{basic_concept}", algorithm_summary["基本概念"])

    core_ideas = "\n".join([f"- {idea}" for idea in algorithm_summary["核心思想"]])
    content = content.replace("{core_ideas}", core_ideas)

    complexity = f"- 时间复杂度：{algorithm_summary['算法复杂度分析']['时间复杂度']}\n- 空间复杂度：{algorithm_summary['算法复杂度分析']['空间复杂度']}"
    content = content.replace("{complexity}", complexity)

    applicable_scenarios = "\n".join([f"- {scenario}" for scenario in algorithm_summary["适用场景"]])
    content = content.replace("{applicable_scenarios}", applicable_scenarios)

    # 生成不同难度级别的题目列表
    def generate_problem_list(level_problems):
        return "\n".join([f"- **{p['name']}**（{p['platform']}, {p['difficulty']}）\n - 类型：{p['type']}\n - 描述：{p['description']}\n - 所需技能：{', '.join(p['skills'])}" for p in level_problems])

    content = content.replace("{basic_problems}", generate_problem_list(training_plan.get("basic", [])))
    content = content.replace("{intermediate_problems}", generate_problem_list(training_plan.get("intermediate", [])))
    content = content.replace("{advanced_problems}", generate_problem_list(training_plan.get("advanced", [])))
    content = content.replace("{expert_problems}", generate_problem_list(training_plan.get("expert", [])))

    data_structures = "\n".join([f"- {ds}" for ds in algorithm_summary["常用数据结构组合"]])
    content = content.replace("{data_structures}", data_structures)

    implementation_points = "\n".join([f"- {point}" for point in algorithm_summary["实现要点"]])
    content = content.replace("{implementation_points}", implementation_points)

```
    common_problems = "\n".join([f"{problem}" for problem in algorithm_summary["常见问题与解决技巧"]])
    content = content.replace("{common_problems}", common_problems)

    engineering_considerations = "\n".join([f"- {point}" for point in algorithm_summary["工程化考量"]])
    content = content.replace("{engineering_considerations}", engineering_considerations)

    language_differences = "\n".join([f"- **{lang}**：{diff}" for lang, diff in algorithm_summary["跨语言实现差异"].items()])
    content = content.replace("{language_differences}", language_differences)

    return content

if __name__ == "__main__":
    main()

==========================================
```