1 二叉树消消乐

二叉树消消乐

时间限制： 1000MS
内存限制： 256MB

题目描述：

给定原始二叉树和参照二叉树 (输入的二叉树均为满二叉树，二叉树节点的值范围为 [1,1000]，二叉树的深度不超过 1000)，现对原始二叉树和参照二叉树中相同层级目值相同的节点进行消除，消除规则为原始二叉树和参照二叉中存在多个值相同的节点只能消除等数量的，消除后的节点变为无效节点，请按节点值出现频率从高到低输出消除后原始二叉树中有效节点的值 (如果原始二叉树消除后没有有效节点返回 0)。

输入描述：

原始二叉树中的节点个数
原始二叉树
参照二叉树中的节点个数
参照二叉树

输出描述：

原始二叉树中有效节点的值，按出现频率从高到低排序 (相同频率的值按大小排序)，相同频率的值按降序排列。

样例输入 1：

```
7
1 3 3 3 4 5 6
3
2 3 4
```

样例输出 1：

```
36541
```

样例 1 解释：原始二叉树 A 消除参照二叉树 B 中的重复元素后，有效节点剩余 2 个 3，1 个 6，1 个 5，1 个 4，1 个 1，3 出现的频率 2, 6、5、4、1 出现的频率为 1，按值从大到小排序，所以排序结果为 36541。

样例输入 2：

```
15
5 6 6 6 7 7 7 8 8 9 9 7 7 5 6
7
5 6 6 7 7 8 8
```

样例输出 2：

```
79865
```

思路分析：

实现代码：

```
```

2 好友推荐系统

好友推荐系统

时间限制： 1000MS
内存限制： 256MB

题目描述：

你正在为一个社交网络平台开发好友推荐功能。

平台上有 N 个用户 (每个用户使用 1 到 N 的整数编号)，同时系统中维护了用户之间的好友关系。

为了推荐新朋友，平台决定采用“共同好友数量”作为衡量两个用户之间相似度的标准。

系统根据输入用户编号 K，输出与此用户 K 相似度最高的前 L 个用户 ID，来推荐给用户 K

相似度定义：两个用户非好友，两个用户的相似度为拥有的共同好友数 (例如用户 A 和用户 B，只有共同好友 C 和 D，相似度 =2)

输入描述：

第一行输入一个整数 $n(1 \leq n \leq 100)$，表示物品的数量。

第二行输入 n 个整数 $a_1, a_2, \cdots, a_n (a_i \in [1, 2])$ 代表物品类别。其中 $a_i = 1$ 表示物品是第一类，$a_i = 2$ 表示物品是第二类。

第三行输入 n 个整数 $b_i, b_2, \cdots, b_n (b_i \in [0, 1])$ 代表物品是否可被移动，其中 $b_i = 1$ 表示第 i 个物品可以移动，$b_i = 0$ 表示第 i 个物品不可以移动。

输出描述：

在一行上输出一个正整数，表示不美观程度的最小值.

样例输入 1：

```
6 7 3 2
1 2
1 3
2 3
3 4
3 5
4 5
5 6
```

样例输出 1：

```
6 0
```

样例 1 解释：

输入包含了 6 个用户，7 条好友记录，给用户 ID 编号为 3 的用户推荐 2 个好友。

输出只有编号为 6 的用户可能是编号 3 用户的可能好友；

尝试推荐与编号 3 用户无共同好友的其他用户，由于除编号为 6 的用户之外，其他用户和编号 3 用户都是好友，所以找不到陌生人作为推荐的第二个用户：推荐结果不足 2 个用户，所以推荐的第二个用户编码使用 0 来占位补足。

样例输入 2：

```
8 11 1 3
1 2
1 3
2 3
3 4
3 5
4 5
5 6
6 7
```

```
7 8
1 8
2 7
```

样例输出 2：

```
7 4 5
```

样例 2 解释：

输入包含了 8 个用户，11 条好友记录，给用户 ID 编号为 1 的用户推荐 3 个好友。

按照相似度排序推荐给用户 1 的相关好友：7 4 5

思路分析：

实现代码：

```cpp
#include <iostream>
#include <vector>
#include <set>
#include <unordered_map>
#include <algorithm>
using namespace std;

struct User {
    int id, similarity;
    User(int id, int similarity) : id(id), similarity(similarity) {}
    // 自定义排序规则：相似度高优先，相似度相同时编号小优先
    bool operator<(const User& other) const {
        if (similarity == other.similarity)
        return id < other.id;
        return similarity > other.similarity;
    }
};

int main() {
    int N, M, K, L;
    cin >> N >> M >> K >> L;



    vector<set<int>> friends(N + 1); // 邻接表，存储每个用户的好友列表

    // 输入好友关系
    for (int i = 0; i < M; ++i) {
        int X, Y;
        cin >> X >> Y;
        friends[X].insert(Y);
        friends[Y].insert(X);
    }

    unordered_map<int, int> similarity; // 用于存储与用户K的相似度

    // 遍历K的所有好友，找出与K有共同好友的用户
    if (friends[K].size() > 0) { // 特殊情况2：如果K没有好友，跳过这一步
        for (int friendOfK : friends[K]) {
            // 计算K的每个好友的好友
            for (int potentialFriend : friends[friendOfK]) {
                // 如果这个用户不是K本身且不是K的好友，则增加相似度
                if (potentialFriend != K && !friends[K].count(potentialFriend)) {
                    similarity[potentialFriend]++;
                }
            }
        }
    }
```

```

    // 将结果放入一个用户结构体数组中
    vector<User> potentialFriends;
    for (int i = 1; i <= N; ++i) {
        if (i != K && !friends[K].count(i)) { // 不是自己且不是好友
            int sim = similarity.count(i) ? similarity[i] : 0;
            potentialFriends.push_back(User(i, sim));
        }
    }

    // 对所有潜在好友进行排序
    sort(potentialFriends.begin(), potentialFriends.end());

    vector<int> result;
    int recommended = 0;

    // 按照相似度高低输出潜在好友
    for (const User& user : potentialFriends) {
        if (recommended < L) {
            result.push_back(user.id);
            recommended++;
        } else {
            break;
        }
    }

    // 如果推荐不足L个，补充陌生人（没有与用户K共同好友的非好友）
    for (int i = 1; i <= N && recommended < L; ++i) {
        if (i != K && !friends[K].count(i) && similarity.count(i) == 0) {
            if (find(result.begin(), result.end(), i) == result.end()) { // 避免重复推荐
                result.push_back(i);
                recommended++;
            }
        }
    }

    // 如果仍不足L个，补充0
    while (recommended < L) {
        result.push_back(0);
        recommended++;
    }

    // 输出结果
    for (int i = 0; i < result.size(); ++i) {
        if (i > 0) cout << " ";
        cout << result[i];
    }
    cout << endl;

    return 0;
}
```

3 维修工

维修工

时间限制： 1000MS
内存限制： 256MB

题目描述：

小红有一棵有 n 个节点的树，其中每个节点是红色或者黑色，她想知道，删除一个红色节点以及与它相连的全部边后，剩余的连通块 † 中黑色节点数量的最大值是多少。

†：对于树上的两个点，如果它们相互连通，则称他们位于同一个连通块里；显然，在执行删除操作后，剩余部分至多构成两个连通块。

输入描述：

第一行输入一个整数 $n(1 \leq n \leq 10^5)$ 代表节点的数量。

第二行输入一个长度为 n 的字符串 $s_1, s_2, \cdots, s_n(s_1 \epsilon$ `'R','B'`$)$ 代表第 i 个节点的颜色为 s_i。若 s_i 为 `'B'` 表示节点的颜色为黑色，若 s_i 为 `'R'` 则表示节点的颜色为红色。保证 s 中至少有一个红色节点。

此后 $n-1$ 行，第 i 行输入两个整数 u_i 和 $v_i(1 \leq u_i, v_i \leq n; u_i \neq v_i)$ 表示树上第 i 条边连接节点 u_i 和 v_i。

输出描述：

在一行上输出一个整数代表最大值。

样例输入 1：

```
10
RRBBBBBBBB
1 2
1 3
1 4
1 5
1 7
2 8
4 6
4 10
6 9
```

样例输出 1：

```
7
```

思路分析：

1. 输入读取与初始化：
   - 首先，读取节点的数量 n，再读取每个节点的颜色信息（由 `'R'` 表示红色，`'B'` 表示黑色）。
   - 然后，读取 n-1 条边的信息，并构建无向图，这个图表示的是树的结构。
2. DFS 计算每个子树中的黑色节点数量：
   - 利用 DFS（深度优先搜索）从树的根节点开始（这里选择 1 号节点作为根节点），计算每个节点的子树中黑色节点的数量。
   - `sub_black[node]` 记录了以 `node` 为根节点的子树中黑色节点的数量。如果当前节点是黑色节点，`sub_black[node]` 增加 1，否则保持不变。
   - 对于每个节点的所有相邻节点（子节点），递归地计算其子树中的黑色节点数量并累加到当前节点。

3. 寻找删除红色节点后黑色节点最多的连通块；

   - `total_black` 表示整棵树中所有的黑色节点总数。
   - 对于每一个红色节点，我们检查它的邻居节点。如果邻居节点的子树中的黑色节点数量小于当前红色节点的子树，那么说明这个邻居节点的子树在删除红色节点后将成为独立的连通块，因此可以更新最大黑色连通块。
   - 如果邻居节点的子树黑色节点数量大于等于当前红色节点的子树，那么我们计算整个树减去红色节点子树后剩余的黑色节点，并更新最大值。

4. 输出结果。

实现代码：

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

const int MAX_N = 100005;  // 最大节点数

vector<int> adj[MAX_N];    // 邻接表表示树的结构
int sub_black[MAX_N];      // 记录每个节点的子树中黑色节点的数量
char col[MAX_N];           // 记录每个节点的颜色，'R' 表示红色，'B' 表示黑色
int n;                     // 节点数

// DFS 计算每个子树中黑色节点的数量
void dfs(int node, int parent) {
    sub_black[node] = (col[node] == 'B') ? 1 : 0;  // 如果当前节点是黑色，计数+1
    for (int nei : adj[node]) {  // 遍历相邻节点
        if (nei != parent) {     // 避免回到父节点，防止死循环
            dfs(nei, node);      // 递归处理子节点
            sub_black[node] += sub_black[nei];  // 累加子树中的黑色节点数量
        }
    }
}

int main() {
    cin >> n;  // 输入节点数
    string colors;
    cin >> colors;  // 输入每个节点的颜色信息

    // 将颜色信息存入 col 数组中，节点编号从1开始
    for (int i = 0; i < n; ++i) {
        col[i + 1] = colors[i];
    }

    // 读取树的边信息，构建无向图
    for (int i = 1; i < n; ++i) {
        int u, v;
        cin >> u >> v;
        adj[u].push_back(v);
        adj[v].push_back(u);
    }

    // 计算以每个节点为根的子树中黑色节点的数量
    dfs(1, -1);  // 从节点1开始，父节点设为-1（表示无父节点）

    int total_black = sub_black[1];  // 整棵树中黑色节点的总数
    int max_black_comp = 0;  // 记录删除某个红色节点后最大的黑色连通块的数量

```

```
        // 遍历所有节点，找出红色节点
        for (int i = 1; i <= n; ++i) {
            if (col[i] == 'R') {  // 只有红色节点需要考虑删除的情况
                for (int nei : adj[i]) {  // 遍历该红色节点的所有邻居节点
                    if (sub_black[nei] < sub_black[i]) {
                        // 邻居节点的子树形成一个独立的黑色连通块
                        max_black_comp = max(max_black_comp, sub_black[nei]);
                    } else {
                        // 其他部分形成一个连通块
                        max_black_comp = max(max_black_comp, total_black - sub_black[i]);
                    }
                }
            }
        }

        // 输出删除红色节点后最大黑色连通块的数量
        cout << max_black_comp << endl;

        return 0;
    }


```