1 最长山峰数组

最长山峰数组

时间限制： 3000MS
内存限制： 589824KB

题目描述：

小红定义一个长度为 n 的数组为“山峰数组”：存在这样的一个位置，使得这个位置左右两边的全部元素均依次严格递减；使用数学的语言来描述，即存在 $x \in (1,n)$ 使得 $a_{i-1} < a_i (i \in (x,n])$ 且 $a_i > a_{i+1} (i \in [1,x))$。例如 $[1,2,5,4,2]$ 和 $[1,3,2]$ 是“山峰数组”，而 $[1],[1,2,3],[1,2,1,2]$ 和 $[1,2,1,1]$ 不是“山峰数组”。

小红有一个长度为 n 的数组 $a_1, a_2, \cdots, a_n$，她想知道在全部的子数组 † 中，是“山峰数组”的子数组的长度最大值是多少。

†：如果数组 a 可以通过从数组 b 的开头删除若干 (可能为零或全部) 元素以及从结尾删除若干 (可能为零或全部) 元素得到，则数组 a 是数组 b 的子数组。

输入描述：

第一行输入一个整数 $n(1 \le n \le 10^5)$ 代表数组长度。

第二行输入 n 个整数 $a_1, a_2, \cdots, a_n (1 \le a_i \le 10^9)$ 代表数组的值。

输出描述：

在一行上输出一个正整数，代表长度的最大值。

样例输入 1：

```
6
1 1 4 5 1 4
```

样例输出 1：

```
4
```

样例 1 提示：最长的“山峰函数”是 [1,4,5,1]。

样例输入 2：

```
4
1 2 2 1
```

样例输出 2：

```
0
```

思路分析：

1. 输入与初步判断：
    - 首先读取数组的长度 n 和数组内容。
    - 如果数组的长度小于 3，直接返回 0，因为数组必须至少包含 3 个元素才可能形成一个山峰数组。
2. 严格递增与递减的计算：
    - 定义两个辅助数组 `inc` 和 `dec`，分别存储从左到右和从右到左的严格递增与严格递减序列的长度，初始时都设为 1。
    - 遍历数组，计算每个位置 `i` 的从左到右的严格递增序列长度 `inc[i]`。如果当前位置的元素大于前一个元素（即 `a[i] > a[i - 1]`），则该位置的递增长度为前一个位置的递增长度加 1。

- 类似地，计算每个位置 i 的从右到左的严格递减序列长度 dec[i]。如果当前位置的元素大于后一个元素（即 a[i] > a[i + 1]），则该位置的递减长度为后一个位置的递减长度加 1。

3. 寻找最长的山峰数组：
   - 对于每个位置 i，如果它满足左侧有一个严格递增的子序列（inc[i] > 1）且右侧有一个严格递减的子序列（dec[i] > 1），那么它是一个山峰，计算山峰数组的长度为 inc[i] + dec[i] - 1。
   - 遍历所有符合条件的位置，找出最大的山峰数组长度。
4. 输出结果：
   - 输出找到的最长山峰数组的长度。如果没有找到山峰数组，输出 0。

实现代码：

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

using namespace std;

int main() {
    int n;
    cin >> n;  // 读取数组长度
    vector<int> a(n);
    for (int i = 0; i < n; i++) {
        cin >> a[i];  // 读取数组元素
    }

    // 如果数组长度小于3，不可能成为山峰数组
    if (n < 3) {
        cout << 0 << endl;
        return 0;
    }

    vector<int> inc(n, 1); // inc[i] 表示从左到右，以 i 结尾的最长严格递增序列的长度
    vector<int> dec(n, 1); // dec[i] 表示从右到左，以 i 开始的最长严格递减序列的长度

    // 计算从左到右的严格递增序列长度
    for (int i = 1; i < n; i++) {
        if (a[i] > a[i - 1]) {
            inc[i] = inc[i - 1] + 1;  // 如果当前元素大于前一个元素，则递增长度+1
        }
    }

    // 计算从右到左的严格递减序列长度
    for (int i = n - 2; i >= 0; i--) {
        if (a[i] > a[i + 1]) {
            dec[i] = dec[i + 1] + 1;  // 如果当前元素大于后一个元素，则递减长度+1
        }
    }

    int max_len = 0;  // 记录找到的最长山峰数组的长度

    // 寻找满足山峰数组条件的子数组长度最大值
    for (int i = 1; i < n - 1; i++) {
        // 要成为山峰，左侧必须有严格递增，右侧必须有严格递减
        if (inc[i] > 1 && dec[i] > 1) {
            max_len = max(max_len, inc[i] + dec[i] - 1);  // 更新最长山峰数组长度
        }
    }

```

```
48      cout << max_len << endl;  // 输出结果
49
50      return 0;
51  }
```

2 最小化物品排列的不美观程度

最小化物品排列的不美观程度

时间限制： 3000MS
内存限制： 589824KB

题目描述：

小红经常从小红书上的笔记中获得灵感。其中一篇笔记认为，如果有 n 个物品，可以把所有物品分为两类，这些物品摆成一排，如果相邻的两个物品属于不同的类别，那么不美观程度就会加一。

现在小红有 n 个物品，其中一些物品不方便移动，另一些物品可以移动。小红想知道，如何移动其中可以移动的物品, 使得不美观程度最小。

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：

```
5
1 2 1 2 1
0 1 1 0 1
```

样例输出 1：

```
1
```

交换第二个和第五个物品，可以使得不美观程度最小。此时物品类别为 $[1, 1, 1, 2, 2]$。

思路分析：

核心问题在于通过移动可移动物品来使相邻物品类别相同的次数尽可能少，从而最小化“不美观程度”。

1. 首先需要定义一个计算不美观程度的函数 `calcUgliness(const vector<int>& arr)` ;
    - 这是一个辅助函数，负责计算当前排列的不美观程度。通过遍历数组 `a`，每当遇到相邻元素不同的情况，增加不美观程度。
2. 需要对所有可移动的物品进行不同的排列，以寻找能够最小化不美观程度的排列；
3. 通过递归枚举所有可移动物品的排列，并在递归过程忠交换位置进行排列生成。每次每次递归结束时，恢复原状；
4. 在暴力法的基础上，加入备忘录，避免重复计算同一排列的不美观程度。我们可以使用一个哈希表记录已经计算过的排列结果，后续遇到相同的排列时直接取值，避免重复计算；
    - 使用 `getKey` 将当前物品排列转换为字符串形式，作为哈希表 `memo` 的键。这是备忘录的关键，避免重复计算相同排列。
    - 这个递归函数用于生成可移动物品的全排列。`idx` 表示当前递归的层次。对于每一个排列，递归到最深处后（即 `idx == mov.size()`），计算该排列的不美观程度。如果这个排列已经存在于 `memo` 中，直接使用它的值；否则，将当前计算结果存入 `memo`。

实现代码：

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

// 计算不美观程度
int calcUgliness(const vector<int>& arr) {
    int ug = 0;
    for (int i = 1; i < arr.size(); ++i) {
        if (arr[i] != arr[i - 1]) {
            ++ug;
        }
    }
    return ug;
}

// 获取排列的键
string getKey(const vector<int>& arr) {
    string key;
    for (int x : arr) {
        key += to_string(x) + ",";
    }
    return key;
}

// 递归函数：生成所有可移动物品的排列
void dfs(int idx, vector<int>& a, const vector<int>& b, vector<int>& mov, int& minUg, unordered_map<string, int>& memo) {
    if (idx == mov.size()) {
        vector<int> curr = a;
        int movIdx = 0;
        for (int i = 0; i < a.size(); ++i) {
            if (b[i] == 1) {
                curr[i] = mov[movIdx++];
            }
        }
        string key = getKey(curr);
        if (memo.find(key) != memo.end()) {
            minUg = min(minUg, memo[key]);
        } else {
            int currUg = calcUgliness(curr);
            minUg = min(minUg, currUg);
            memo[key] = currUg;
        }
        return;
    }

    // 递归+回溯
    for (int i = idx; i < mov.size(); ++i) {
        swap(mov[idx], mov[i]);
        dfs(idx + 1, a, b, mov, minUg, memo);
        swap(mov[idx], mov[i]); // 回溯
    }
}

int main() {
    int n;
    cin >> n;
    vector<int> a(n), b(n), mov;
    unordered_map<string, int> memo;

    // 输入类别
```

```
    for (int i = 0; i < n; ++i) cin >> a[i];
    // 输入可移动性
    for (int i = 0; i < n; ++i) {
        cin >> b[i];
        if (b[i] == 1) mov.push_back(a[i]);
    }

    // 初始不美观度
    int minUg = calcUgliness(a);

    // 递归搜索
    dfs(0, a, b, mov, minUg, memo);

    cout << minUg << endl;
    return 0;
}
```

3 小红的红黑树

小红的红黑树

时间限制： 3000MS
内存限制： 589824KB

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;
}
```