E. Anton and Tree

time limit per test: 3 seconds
memory limit per test: 256 megabytes

→ Problem tags

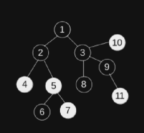
dfs and similar | dp | trees | *2100

No tag edit access

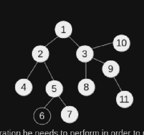
Anton is growing a tree in his garden. In case you forgot, the tree is a connected acyclic undirected graph.

There are n vertices in the tree, each of them is painted black or white. Anton doesn't like multicolored trees, so he wants to change the tree such that all vertices have the same color (black or white).

To change the colors Anton can use only operations of one type. We denote it as $paint(v)$, where v is some vertex of the tree. This operation changes the color of all vertices u such that all vertices on the shortest path from v to u have the same color (including v and u). For example, consider the tree



and apply operation $paint(3)$ to get the following:



Anton is interested in the minimum number of operation he needs to perform in order to make the colors of all vertices equal.

Input

The first line of the input contains a single integer n ($1 \le n \le 200\,000$) — the number of vertices in the tree.

The second line contains n integers $color_i$ ($0 \le color_i \le 1$) — colors of the vertices. $color_i = 0$ means that the i-th vertex is initially painted white, while $color_i = 1$ means it's initially painted black.

Then follow $n - 1$ line, each of them contains a pair of integers u_i and v_i ($1 \le u_i, v_i \le n, u_i \ne v_i$) — indices of vertices connected by the corresponding edge. It's guaranteed that all pairs (u_i, v_i) are distinct, i.e. there are no multiple edges.

Output

Print one integer — the minimum number of operations Anton has to apply in order to make all vertices of the tree black or all vertices of the tree white.

Examples

input Copy
```
11
0 0 0 1 1 0 1 0 0 1 1
1 2
1 3
2 4
2 5
5 6
5 7
3 8
3 9
3 10
9 11
```
output Copy
```
2
```

input Copy
```
4
0 0 0 0
1 2
2 3
3 4
```
output Copy
```
0
```

Note

In the first sample, the tree is the same as on the picture. If we first apply operation $paint(3)$ and then apply $paint(6)$, the tree will become completely black, so the answer is 2.

In the second sample, the tree is already white, so there is no need to apply any operations and the answer is 0.

Problem Breakdown

Lets understand the problem first

We have a tree of n nodes (connected and acyclic).
Each node is colored either:

- 0 → white
- 1 → black

We can perform an operation paint(v) which:

Flips the color of the entire monochromatic connected component that contains v.

We want the minimum number of such operations to make all nodes the same color (all 0 or all 1).

What the operation really does ?

The operation paint(v):

- Finds all vertices reachable from v without crossing an edge that connects two differently-colored vertices,
- and flips all of them (0→1, 1→0).

So it flips an entire connected region of the same color.

-> Every "color boundary" (edge connecting two nodes of different colors) acts as a wall — paint(v) doesn't cross it.

Reduce the problem conceptually

Let's build a new graph where:

- each node represents a monochromatic component (connected region of same color),
- and edges exist between components that were connected by a color boundary in the original tree.

This new graph is:

- also a tree (because merging nodes of same color doesn't create cycles),
- and it's bipartite (because edges always connect 0 ↔ 1 components).

The answer comes from the diameter of the component-tree

This is the main trick.

- Every paint() operation can "push" color propagation one step through this component-tree.
- To unify all components, you need roughly half the length of the longest alternating path — because each operation flips one side at a time.

Formally:

The minimum number of operations = ⌈ (diameter of the component-tree) / 2 ⌉

Why?

- Each operation can merge one level of color alternation.
- The worst case is when colors alternate along the longest path.
- It takes half the diameter (rounded up) to make them all one color.

Code

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

int n;
vector<int> color;
vector<vector<int>> adj;
vector<bool> used;
vector<int> comp;
vector<vector<int>> compGraph;
vector<int> dp;
int compCount = 0;
int answer = 0;

// DFS #1 - find connected components of same color
void dfsColor(int v, int col, int cmp) {
    if (used[v] || color[v] != col) return;
    used[v] = true;
    comp[v] = cmp;

    for (int u : adj[v]) {
        dfsColor(u, col, cmp);
    }
}

// DFS #2 - compute diameter in component graph
void dfsDiameter(int v, int parent = -1) {
    int best1 = 0, best2 = 0; // two largest depths among children

    for (int u : compGraph[v]) {
        if (u == parent) continue;
        dfsDiameter(u, v);
        int depth = dp[u] + 1;
        if (depth > best1) {
            best2 = best1;
            best1 = depth;
        } else if (depth > best2) {
            best2 = depth;
        }
    }

    dp[v] = best1;
    answer = max(answer, best1 + best2);
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    cin >> n;
    color.resize(n);
    adj.assign(n, {});
    used.assign(n, false);
    comp.resize(n);

    for (int i = 0; i < n; ++i)
        cin >> color[i];

    for (int i = 0; i < n - 1; ++i) {
        int a, b;
        cin >> a >> b;
        --a; --b;
        adj[a].push_back(b);
        adj[b].push_back(a);
    }

    // Step 1: group nodes of the same color into components
    for (int i = 0; i < n; ++i) {
        if (!used[i]) {
            dfsColor(i, color[i], compCount++);
        }
    }

    // Step 2: build component graph (edges between different colors)
    compGraph.assign(compCount, {});
    for (int v = 0; v < n; ++v) {
        for (int u : adj[v]) {
            if (comp[v] != comp[u]) {
                compGraph[comp[v]].push_back(comp[u]);
            }
        }
    }

    // Step 3: find diameter of component graph
    dp.assign(compCount, 0);
    dfsDiameter(0);

    cout << (answer + 1) / 2 << "\n";
    return 0;
}
```