Planets Cycles

You are playing a game consisting of n planets. Each planet has a teleporter to another planet (or the planet itself).

You start on a planet and then travel through teleporters until you reach a planet that you have already visited before.

Your task is to calculate for each planet the number of teleportations there would be if you started on that planet.

Input

The first input line has an integer n: the number of planets.
The planets are numbered 1,2,...,n.

The second line has n integers t_1,t_2,...,t_n: for each planet, the destination of the teleporter.
It is possible that t_i=i.

Output

Print n integers according to the problem statement.

Constraints

1 <= n <= 2 . 10^5
1 <= t_i <= n

Example

Input:
5
2 4 3 1 4

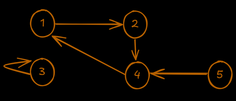
Output:
3 3 1 3 4

Problem Breakdown

Lets understand the problem first

We have to tell starting from each planet when will we find a planet that we have already visited

Or we can say,
For every planet `i`, find out the length of the cycle it eventually ends up in (including any path taken to reach it).



Algorithm

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

// Data structures to store graph and computation results
vector<int> arr;      // arr[u] = the planet teleportation target from planet u
vector<int> dp;       // dp[u] = cycle length if u is in cycle, else distance to cycle + cycle length
vector<int> visited;  // Tracks visited nodes during DFS
vector<int> dist;     // dist[u] = distance from DFS start node to u

// Variables to track cycle information during DFS
int cycle_start = -1; // Marks the start node of a detected cycle
int cycle_len = 0;    // Length of the detected cycle

void dfs(int u, int current_dist) {
    // Skip if node was already visited
    if (visited[u]) return;

    // Mark node as visited and record its distance from DFS start
    visited[u] = true;
    dist[u] = current_dist;

    int v = arr[u]; // The planet this one teleports to

    if (!visited[v]) {
        // If successor hasn't been visited, continue DFS
        dfs(v, current_dist + 1);
    } else {
        // We've found a visited node - cycle detection
        if (dp[v] == 0) {
            // If successor's dp isn't set, this is a new cycle
            cycle_len = dist[u] - dist[v] + 1; // Calculate cycle length
            cycle_start = v;                   // Mark where cycle begins
        } else {
            // If successor's dp is set, use its cycle length
            cycle_len = dp[v];
        }
    }

    // Update dp for current node
    if (cycle_start != -1) {
        // If we're in or after a cycle start
        dp[u] = cycle_len; // All nodes in cycle get cycle_len

        // If we've completed the cycle, reset the marker
        if (u == cycle_start) {
            cycle_start = -1;
        }
    } else {
        // For nodes before the cycle starts
        dp[u] = dp[v] + 1; // Distance increases by 1 from successor
    }
}

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

    int n;
    cin >> n;

    // Initialize data structures
    arr.resize(n + 1);
    dp.resize(n + 1, 0);
    visited.resize(n + 1, false);
    dist.resize(n + 1, 0);

    // Read input graph
    for (int i = 1; i <= n; ++i) {
        cin >> arr[i];
    }

    // Process each unvisited node
    for (int i = 1; i <= n; i++) {
        if (!visited[i]) {
            // Reset cycle tracking for new component
            cycle_start = -1;
            cycle_len = 0;
            dfs(i, 1); // Start DFS with distance 1
        }
    }

    // Output results
    for (int i = 1; i <= n; i++) {
        cout << dp[i] << " ";
    }
    cout << endl;

    return 0;
}
```

Time Complexity

DFS traversal -> O(n)

Space Complexity

Space for arrays -> O(n)