# 1443. Minimum Time to Collect All Apples in a Tree

Medium | Tree | Depth-First Search | Breadth-First Search | Hash Table | Leetcode Link

## Problem Description

The problem presents an undirected tree with `n` vertices. Each vertex in the tree is numbered from `0` to `n-1` and can contain zero or more apples. You start at vertex `0` and must find the shortest amount of time to collect all the apples in the tree and return to the starting vertex.

Walking from one vertex to another over an edge takes 1 second. The collection of edges forming the tree is given in the `edges` array, where each element is a pair `[a, b]` indicating a bidirectional connection between vertices `a` and `b`.

A separate boolean array `hasApple` indicates whether a particular vertex contains an apple. Specifically, `hasApple[i] = true` means that vertex `i` has an apple.

The goal is to calculate the minimum time in seconds needed to traverse the tree, collect all the apples, and return to vertex 0. Note that additional time should not be spent traversing to vertices that do not lead to an apple or are not on the path to collect an apple.

## Intuition

The intuition behind the solution to this problem involves performing a Depth-First Search (DFS) from the starting vertex. DFS is particularly useful for this scenario as it allows to explore the tree in a path-oriented manner, visiting all vertices necessary to collect the apples.

The key is to eliminate unnecessary movements. If a subtree does not contain any apples, visiting it would be a waste of time. Therefore, while performing DFS, only traverse to children vertices that have an apple or lead to one.

By using DFS, we start at the root (vertex 0) and traverse down each branch, only continuing as long as there are apples in that direction. If we reach a vertex without an apple and none of its children have apples, we do not need to go further down that path.

When moving back up the tree (after visiting all descendants of a vertex), collect the time spent unless it is the root vertex or the branch did not lead to any apples.

The algorithm also keeps track of visited vertices to avoid redundant traversals, as a vertex may connect to multiple others, creating potential cycles that would not exist in a tree structure.

In summary, the DFS traverses down to collect apples and adds the time cost accordingly, while avoiding traversals of paths that do not contribute to the goal of apple collection.

## Solution Approach

The solution to our problem uses Depth-First Search (DFS), a common algorithm for traversing or searching tree or graph data structures. The specific implementation begins with the root of the tree, vertex `0`, and explores as far as possible along each branch before backtracking.

Here are the main components of the implementation:

- A graph representation `g` (using a `defaultdict` of lists) of the undirected tree that enables easy access to connected vertices.
- A list `vis` to keep track of visited vertices, which is necessary for DFS to prevent revisiting the same vertex, causing infinite loops.
- A recursive `dfs` function that encapsulates the logic for traversal and time calculation.

The `dfs` function performs the following:

1. Accepts a vertex `u` and a `cost` parameter (the time taken to travel to this vertex from its parent).
2. Checks if `u` has already been visited to prevent re-traversal. If visited, it returns `0` because no additional cost should be incurred.
3. If not visited, marks `u` as visited.
4. Initiates a variable `nxt_cost` to aggregate the cost of DFS in the subtree rooted at `u`.
5. Iterates over all the children `v` of `u` (accessible directly via an edge) and adds the result of the recursive DFS for child `v` to `nxt_cost`. The cost for moving to any child is `2` seconds (1 second to go and 1 second to possibly return).
6. After exploring all children, decides whether to include the cost for `u`. If `u` has no apple and `nxt_cost` is `0` (meaning neither `u` nor any of its descendants have apples), it returns `0`. Otherwise, it includes the cost to travel to `u` (`cost`) and any costs from its descendants (`nxt_cost`).

Finally, the root call to `dfs` for vertex `0` does not need any cost associated with it—it is the starting point, so the initial cost is `0`. The total time to collect all the apples is computed by the accumulation of costs from each DFS call.

### Example Walkthrough

Let's illustrate the solution approach with a small example. Suppose we have `n = 5` vertices and the following tree structure with apples:

```
Edges: [[0, 1], [0, 2], [1, 3], [1, 4]]
hasApple: [false, true, false, true, false]
```

There is an apple at vertex `1` and at vertex `3`. Vertex `0` is where we start and do not have any apples.

Following the algorithm's steps:

1. Start DFS at vertex `0`. It doesn't have an apple, but we don't consider the cost here, as it's our starting point.
2. Look at the children of `0`, which are `1` and `2`.
3. Move to vertex `1` (cost is now `2`, 1 to go and 1 to potentially return).
4. At vertex `1`, we have an apple, so we will include this cost.
5. Continue DFS from `1` to its children, which are `3` and `4`.
6. Visit `3` (cost at `3` is `2`), there is an apple, so we collect it and return this cost.
7. Visit `4` (cost at `4` is `2`), no apple, and no children with apples, so ignore this path and return `0` cost.
8. We backtrack to `1` with the total cost from its subtree: `2` (to `3` and back) + `0` (ignoring `4`).
9. Vertex `2` has no apples and no children with apples, so ignore this path.
10. The total minimum time used is the cost to travel to `1` (`2` seconds) and collect the apple, plus the cost to and from `3` (`2` seconds). Thus, the total time is `4` seconds to collect all apples and return to `0`.

Here are the visualization steps:

```
Step: Vertex - Action - Cost Accumulated
0: Start at root
1: 0 -> 1 - Move - Cost: 0
2: Collect apple at 1 - Include Cost: 2 (1 go + 1 return)
3: 1 -> 3 - Move - Cost: 2 (previous cost)
4: Collect apple at 3 - Include Cost: 4 (2 go + 2 return)
5: 3 -> 1 - Return - Cost: 4
6: 1 -> 0 - Return to start - Cost: 4
7: 0 -> 2 - Check path - Cost: 4
8: No apple at 2 and no children, ignore
9: Finish - Total Minimum Time: 4 seconds
```

This walk through demonstrates how the DFS method efficiently guides movements towards only the necessary vertices, avoiding wasted time on paths without apples.

## Python Solution

```python
from collections import defaultdict

class Solution:
    def minTime(self, n: int, edges: List[List[int]], has_apple: List[bool]) -> int:
        # Helper function to perform depth-first search from the current node.
        def dfs(current_node, cost_to_come_back):
            # If the current node has been visited, no need to do anything.
            if visited[current_node]:
                return 0
            visited[current_node] = True

            # Accumulated cost of visiting children nodes.
            total_cost = 0
            for child_node in graph[current_node]:
                # Perform DFS on child nodes with the cost of coming back (2 units).
                total_cost += dfs(child_node, 2)

            # If there's no apple at the current node, and no cost accumulated from children,
            # it means we don't need to collect any apples on this path.
            if not has_apple[current_node] and total_cost == 0:
                return 0

            # Return the total cost of picking apples from this subtree (including the cost to come back).
            return cost_to_come_back + total_cost

        # Construct the graph using adjacency lists.
        graph = defaultdict(list)
        for u, v in edges:
            graph[u].append(v)
            graph[v].append(u)

        # Initialize a visited list to avoid revisiting nodes.
        visited = [False] * n

        # Perform a DFS starting at node 0 with no initial cost.
        # Since we start at the root and don't need to return, we pass 0 as the cost.
        return dfs(0, 0)

```

## Java Solution

```java
class Solution {

    public int minTime(int n, int[][] edges, List<Boolean> hasApple) {
        // Visited array to keep track of visited nodes
        boolean[] visited = new boolean[n];

        // Graph representation using adjacency lists
        List<Integer>[] graph = new List[n];
        Arrays.setAll(graph, x -> new ArrayList<>());

        // Building the undirected graph from the edges
        for (int[] edge : edges) {
            int from = edge[0], to = edge[1];
            graph[from].add(to);
            graph[to].add(from);
        }

        // Starting DFS from node 0 with initial cost as 0
        return dfs(0, 0, graph, hasApple, visited);
    }

    private int dfs(int currentNode, int accumulatedCost, List<Integer>[] graph,
                    List<Boolean> hasApple, boolean[] visited) {
        // If the current node is already visited, we return 0 as no additional cost is needed
        if (visited[currentNode]) {
            return 0;
        }

        // Mark the current node as visited
        visited[currentNode] = true;

        // Variable to keep track of the total cost from child nodes
        int childrenCost = 0;

        // Traverse through all the adjacent nodes
        for (int adjacentNode : graph[currentNode]) {
            childrenCost += dfs(adjacentNode, 2, graph, hasApple, visited);
        }

        // If the current node does not have an apple and none of its children have one,
        // we need not take any actions; hence return 0 cost.
        if (!hasApple.get(currentNode) && childrenCost == 0) {
            return 0;
        }

        // Accumulated cost includes the cost from child nodes and possibly the cost for visiting current node
        return accumulatedCost + childrenCost;
    }
}

```

## C++ Solution

```cpp
class Solution {
public:
    // The main function that calculates the minimum time to collect all apples.
    int minTime(int n, vector<vector<int>>& edges, vector<bool>& hasApple) {
        vector<bool> visited(n, false);          // Keep track of visited nodes
        vector<vector<int>> graph(n);            // Adjacency list representation of the graph

        // Building the undirected graph
        for (auto& edge : edges) {
            int u = edge[0], v = edge[1];
            graph[u].push_back(v);
            graph[v].push_back(u);
        }

        // Start DFS from the node 0 (root) with an initial cost of 0
        return dfs(0, 0, graph, hasApple, visited);
    }

private:
    // Depth-first search function to navigate the tree and calculate the cost.
    int dfs(int node, int cost, vector<vector<int>>& graph,
            vector<bool>& hasApple, vector<bool>& visited) {

        // If the node is already visited, skip it to prevent cycles
        if (visited[node]) return 0;
        visited[node] = true;    // Mark the current node as visited

        int totalCost = 0;       // Total cost to collect apples from child nodes

        // Explore all adjacent nodes (children)
        for (int& child : graph[node]) {
            // Add to the total cost the cost of collecting apples from the child subtree
            totalCost += dfs(child, 2, graph, hasApple, visited);
        }

        // If the node does not have an apple and there is no cost incurred from its children,
        // it means we don't need to collect apples from this path, hence return 0.
        if (!hasApple[node] && totalCost == 0) return 0;

        // Otherwise, return the cost of the current path plus the totalCost from children.
        return cost + totalCost;
    }
};

```

## Typescript Solution

```typescript
// Define the type for the edges as an array of number arrays.
type Edges = number[][];

// The main function calculates the minimum time to collect all apples.
function minTime(n: number, edges: Edges, hasApple: boolean[]): number {
    const visited: boolean[] = new Array(n).fill(false); // Keep track of visited nodes
    const graph: number[][] = new Array(n).fill(0).map(() => []); // Adjacency list representation of the graph

    // Building the undirected graph
    edges.forEach(edge => {
        const [u, v] = edge;
        graph[u].push(v);
        graph[v].push(u);
    });

    // Start DFS from the node 0 (root) with an initial cost of 0
    return dfs(0, 0, graph, hasApple, visited);
}

// Depth-first search function to navigate the tree and calculate the cost.
function dfs(
    node: number,
    cost: number,
    graph: number[][],
    hasApple: boolean[],
    visited: boolean[]
): number {
    // If the node is already visited, skip it to prevent cycles
    if (visited[node]) return 0;
    visited[node] = true; // Mark the current node as visited

    let totalCost = 0; // Total cost to collect apples from child nodes

    // Explore all adjacent nodes (children)
    for (const child of graph[node]) {
        // Add to the total cost the cost of collecting apples from the child subtree
        totalCost += dfs(child, 2, graph, hasApple, visited);
    }

    // If the node does not have an apple and there is no cost incurred from its children,
    // it means we don't need to collect apples from this path, hence return 0.
    if (!hasApple[node] && totalCost === 0) return 0;

    // Otherwise, return the cost of the current path plus the totalCost from children.
    return cost + totalCost;
}

```

## Time and Space Complexity

### Time Complexity

The given code implements a Depth-First Search (DFS) algorithm. Each node in the tree is traversed exactly once. For each node, the `dfs` function is called, which goes through all of the connected child nodes (via the edges).

The DFS algorithm visits each node once and examines each edge once. In the worst case, there would be `n-1` edges for `n` nodes in a tree (as it is an undirected acyclic graph, i.e., a tree structure). Therefore, the time complexity of the function is `O(n)` because every edge is looked at once, and each node is processed in a single pass.

### Space Complexity

The space complexity of the code is based on the space used by the recurrence stack during the DFS calls, the `vis` array used to mark visited nodes, and the adjacency list representation of the graph `g`.

1. **Recurrence Stack**: In the worst case scenario, where the tree is skewed, the DFS could go as deep as `n-1` calls, making the space complexity due to the recursion stack `O(n)`.

2. **Visited Array**: An array of size `n` is used to mark visited nodes, which consumes `O(n)` space.

3. **Adjacency List**: The adjacency list `g` stores all the edges in a dictionary of lists. In the worst case, a binary tree structure would require about `2*(n-1)` space (since each edge `{u, v}` is stored twice—once for `u`'s list and once for `v`'s list). Therefore, `g` also occupies `O(n)` space.

Because these all are additive contributions to the total space used, and none of them dominate the others in the `O(n)` notation, the overall space complexity of the function is `O(n)`.