1598. Crawler Log Folder

Easy Stack Array String

Leetcode Link

Problem Description

The problem is about simulating a user navigating through a file system by performing operations that move them between folders. The file system is represented as a series of operations, and our aim is to find out the minimum number of operations needed for the user to return to the main folder after executing a given sequence of operations.

There are three types of operations:

- `"../"`: This operation moves the user to the parent folder of the current folder. If the user is already in the main folder, this operation will not change the current folder.
- `"./"`: This operation means that the user will stay in the current folder.
- `"x/"`: Represents moving to a child folder named `x`. It is stated that such a folder will always exist when this operation is performed.

Given a list of strings `logs` where each string represents one of the three operations above, the task is to calculate the minimum steps needed to navigate back to the main folder.

Intuition

To find the solution, we need to keep track of the user's location relative to the main folder. This can be done by treating the main folder as the starting point (i.e., the "root") and using a counter to represent how deep into the file system hierarchy the user has moved.

- When the user moves to a child folder (operation `"x/"`), we increment the counter because the user is moving one level deeper into the hierarchy.
- When the `"../"` operation is encountered, the user attempts to move up one level in the hierarchy. We should decrement the counter to represent going up one level, but with a condition: if the user is already at the main folder, the counter should not go below zero since we can't go above the main folder.
- The operation `"./"` doesn't change the user's depth in the directory hierarchy, so the counter remains unchanged when this operation is applied.

By iterating through each operation in the `logs` array and applying this logic, we can maintain an accurate count of how deep the user is into the folder system. After processing all operations, the value of the counter will be the minimum number of operations needed to go back to the main folder because it represents the depth of the folder the user is currently in, and hence, the number of `"../"` operations needed to return to the main folder.

Solution Approach

The implementation of the solution involves a straightforward approach without the need for complex data structures or algorithms.

Here's how the given solution works:

1. Initialize a variable `ans` to `0`. This will serve as a counter to keep track of the user's depth in the file system, where `0` corresponds to the main folder.

2. Iterate through each operation in the `logs` list. We use a `for` loop to check each value, referred to as `v`, in the list.

3. For each iteration, check if `v` equals `"../"`. If so, decrement the `ans` counter by `1`, but ensure that the counter does not go below `0`. This is done using the `max` function:

```
ans = max(0, ans - 1)
```

The `max` function ensures that `ans` will stay at `0` even if `ans - 1` yields a negative number, which can happen if we're already in the main folder and encounter a move to the parent folder.

4. If the operation is not `"../"` and the operation does not start with a dot (which indicates it's `"./"`), assume it's an operation moving to a child folder (`"x/"`), and increment `ans` by `1`.

```
elif v[0] != ".":
    ans += 1
```

5. After processing all the operations in the `logs` list, return `ans`. The value of `ans` now represents the minimum number of steps to return to the root (main folder) since it counts how deep we've gone into the file system.

The simplicity of this solution lies in the fact that we only need to keep track of one variable (`ans`) and that the operations `"../"` and `"./"` have straightforward effects on this variable. There's no need for a stack or other data structures because we only care about the depth, not the actual path taken.

By the end of the iteration through `logs`, `ans` indicates the fewest number of steps to get back to the main folder. If `ans` is `3`, it means we're three folders deep, and it would take three `"../"` operations to return to the main folder. The solution efficiently computes this with a time complexity of O(n), where n is the number of operations in the `logs` list, and a constant space complexity O(1) because it uses a fixed amount of additional space.

Example Walkthrough

Let's illustrate the solution approach with a small example. Consider the following `logs` list, which represents a sequence of operations in the file system:

```
logs = ["x/", "y/", "../", "z/", "./"]
```

Now, let's walk through the operations one by one using the solution approach:

1. We start with `ans = 0`, which means we are at the main folder.

2. We encounter `"x/"`. This is not `"../"` and not `"./"`, so we are moving to a child folder. We increment `ans`:

```
ans = 0 + 1 -> ans = 1
```

Now, we are one level deep into the file system.

3. Next, we have `"y/"`. As before, this represents moving to another child folder. We increment `ans` again:

```
ans = 1 + 1 -> ans = 2
```

We are now two levels deep.

4. The following operation is `"../"`, which means moving up to the parent folder. We decrement `ans` but ensure it doesn't go below `0`:

```
ans = max(0, 2 - 1) -> ans = 1
```

We have moved one level up, so we are back to being one level deep.

5. We then see `"z/"`. This is another move to a child folder. We increment `ans`:

```
ans = 1 + 1 -> ans = 2
```

We are two levels deep again after moving to the child folder 'z'.

6. Finally, there's `"./"`. This operation indicates staying in the current folder, so `ans` remains unchanged:

```
ans = 2
```

7. Having processed all operations, we conclude that `ans = 2` is the minimum number of steps to move back to the main folder since we are two levels deep in the file system.

Therefore, for the given sequence of operations, the user needs a minimum of 2 steps to navigate back to the main folder. This is done by performing two `"../"` operations.

Python Solution

```
class Solution:
    # Function to compute the minimum number of operations
    # to get back to the main folder after executing a sequence of operations.
    def minOperations(self, logs: List[str]) -> int:
        # Initialize a variable to keep track of the current folder level.
        folder_level = 0

        # Iterate over each operation in the log.
        for operation in logs:
            if operation == "../":
                # Move up a level in the folder hierarchy, but not above the main folder.
                folder_level = max(0, folder_level - 1)
            elif operation != "./":
                # Move down a level into a new folder.
                folder_level += 1

        # Return the folder level, which represents the minimum number of operations needed.
        return folder_level
```

Here's a breakdown of the modifications made:

- Variable `ans` has been renamed to `folder_level` to better represent its purpose.
- The variable `v` within the loop has been renamed to `operation` for clarity.
- A condition `elif v != "./":` has been modified to `elif operation != "./":` for consistency in variable naming.
- Added comments to explain what each part of the code is doing.
- Noted that we do not modify method names, so `minOperations` remains unchanged.

Remember to import `List` from `typing` before using it in the type annotations:

```
from typing import List

```

Java Solution

```
class Solution {
    public int minOperations(String[] logs) {
        // Initialize the steps counter to 0.
        int steps = 0;

        // Iterate through each log entry.
        for (String log : logs) {
            // If the log entry is "../", move one directory up.
            if ("../".equals(log)) {
                // Ensure that steps cannot be negative.
                steps = Math.max(0, steps - 1);
            // If the log entry is not a "." or "../", move one directory deeper.
            } else if (!"./".equals(log)) {
                // Increment the steps counter.
                ++steps;
            }
            // No action required for "./" as it means stay in the current directory.
        }
        // Return the minimum number of operations to end up back at the main folder.
        return steps;
    }
}
```

C++ Solution

```
#include <vector>
#include <string>
#include <algorithm>

class Solution {
public:
    int minOperations(std::vector<std::string>& logs) {
        int depth = 0; // Initialize 'depth' to track the current depth in the filesystem

        for (const auto& log : logs) { // Iterate through each log entry
            if (log == "../") { // If we encounter a parent directory log, move up unless we are at the root
                depth = std::max(0, depth - 1);
            } else if (log != "./") {
                // If the log is not a 'stay in the current directory' instruction,
                // then we must have navigated to a new directory, so we increase depth
                ++depth;
            }
            // If the log is "./", we ignore it since it means 'stay in the current directory'
        }
        return depth; // Return the final depth value representing the minimum operations
    }
};
```

Typescript Solution

```
// This function calculates the minimum number of operations required to get back to the main folder.
// The input is an array of strings where each element represents a log operation.
function minOperations(logs: string[]): number {
    let currentDepth = 0; // Initialize the current depth to 0, representing the main folder level.

    // Iterate through each log in the log array
    for (const log of logs) {
        if (log === '../') {
            // If the log is '../' it means move one folder up, but not beyond the main folder.
            currentDepth = Math.max(0, currentDepth - 1);
        } else if (log !== './') {
            // If the log is anything other than './' it means move one folder deeper.
            currentDepth++;
        }
        // If the log is './', it means stay in the current folder, and no action is needed.
    }

    // The function returns the final currentDepth as the minimum number of operations to get back to the main folder.
    return currentDepth;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the code is `O(n)`, where `n` is the number of operations in the input list `logs`. This is because there is a single loop that iterates over all elements in the list, and the operations within the loop (checking the string and incrementing/decrementing the counter) are executed in constant time.

Space Complexity

The space complexity of the code is `O(1)` since we only use a fixed number of variables (`ans`) to store the intermediate results, irrespective of the size of the input. There is no use of any auxiliary data structure that would grow with the size of the input.