1233. Remove Sub-Folders from the Filesystem

Medium Depth-First Search Trie Array String

Problem Description

In this problem, we are given a list of folder paths in a file system. Each folder path is represented as a string, which follows a specific format - a concatenation of lowercase English letters, where each directory in the path is preceded by a slash (`'/'`). For example, `"/leetcode"` and `"/leetcode/problems"` are valid folder paths.

Our goal is to eliminate any folder paths that are sub-folders of other folder paths. A folder `A` is considered a sub-folder of folder `B` if the path of `A` starts with the entire path of `B` followed directly by a slash and additional characters.

For instance, given a folder path `"/leetcode"`, any folder that has a path starting with `"/leetcode/"`, such as `"/leetcode/problems"`, is considered a sub-folder of `"/leetcode"`.

We have to identify all the "unique" folders after removing these sub-folders and can return the remaining paths in any order.

Intuition

The provided solution works by sorting the list of folders first. Sorting is key because it ensures that any potential sub-folder paths come immediately after the parent folder path. This is due to lexicographical ordering, where a string that starts with another string will come right after it.

After sorting, we iterate through the list starting with the second folder path (as the first folder cannot be a sub-folder of any other). With each folder path, we compare it to the most recently added folder in our answer list, which is initially the first folder from the sorted list.

During this iteration, if the current path does not start with the path of the last added folder (which would make it a sub-folder), it gets added to the answer list, effectively keeping it as a unique folder.

This is achieved with a check that ensures that the last folder is not a prefix of the current folder followed immediately by a slash (to guarantee it's a sub-folder and not just a folder with a similar prefix).

In this way, we build a collection of folder paths, removing any sub-folders and retaining only the "unique" ones. The final result is a list of the cleaned folder paths.

Solution Approach

The solution begins with sorting the list of folders. This is done using Python's built-in `sort()` method, which arranges the folder paths in lexicographical order. The significance of sorting here is that if one folder is a sub-folder of another, it will appear immediately after its parent folder in the sorted list.

```
folder.sort()
```

After sorting the folders, the algorithm initializes an `ans` list with the first folder path in it, since there's nothing to compare it with, it can't be a sub-folder of any sort:

```
ans = [folder[0]]
```

Next, we loop over the remaining folder paths starting from the second item. The `for` loop determines if the current folder (`f`) is a sub-folder of the most recently added folder in the `ans` list. To check this, we compare the lengths of the current folder (`f`) and the last added folder (`ans[-1]`), and we also ensure that `ans[-1]` is the prefix of `f` followed by a `'/'`:

```
for f in folder[1:]:
    m, n = len(ans[-1]), len(f)
```

The condition inside the loop checks two things:

1. If `m` is not less than `n`, `f` cannot be a sub-folder of `ans[-1]` because a sub-folder must have a longer path than its parent folder.
2. If `ans[-1]` is not a prefix of `f` or if the character in `f` that immediately follows the prefix is not the slash `/`, `f` is not a sub-folder.

If these conditions are not met, it means `f` is a unique folder (not a sub-folder), so it gets appended to `ans`:

```
if m >= n or not (ans[-1] == f[:m] and f[m] == '/'):
    ans.append(f)
```

At the end of the loop, `ans` contains all unique folders, excluding any sub-folders, following the criteria. We return `ans` as the final result:

```
return ans
```

In terms of data structures and patterns, the solution mainly relies on list operations and basic string manipulation. The sorting operation helps utilize the power of lexicographical order to organize and efficiently identify sub-folders based on their path prefixes.

Example Walkthrough

Let's say we have the following list of folder paths:

```
["/a", "/a/b", "/c/d", "/c/d/e", "/c/f"]
```

According to the problem statement, we need to remove any paths that are sub-folders of any other path. We'll illustrate the solution approach using this list.

Firstly, we sort the list of folders:

Before sorting: `["/a", "/a/b", "/c/d", "/c/d/e", "/c/f"]`

After sorting: `["/a", "/a/b", "/c/d", "/c/d/e", "/c/f"]`

In this case, the list remains the same after sorting because it was already in lexicographical order. But the sorting step is crucial because if our list had been unordered, sorting would have arranged any sub-folders immediately after their parent folders.

We initialize the `ans` list with the first folder:

```
ans = ["/a"]
```

We then loop over the remaining folders in the list starting from the second one.

- We compare `"/a/b"` with the last element in `ans`, which is `"/a"`. Since `"/a/b"` starts with `"/a"` and has additional characters after `"/a"`, it is a sub-folder, and we do not add it to `ans`.
- Next, we compare `"/c/d"` with the last element in `ans` which is still `"/a"`. It does not start with `"/a"` so we add it to `ans`.

```
ans = ["/a", "/c/d"]
```

- We move to `"/c/d/e"` and compare it with the last element in `ans`, which is now `"/c/d"`. Since `"/c/d/e"` starts with `"/c/d"`, it is a sub-folder, and we do not add it.
- Lastly, we compare `"/c/f"` with `"/c/d"`. It does not start with `"/c/d/"`, so we add it to `ans`.

```
ans = ["/a", "/c/d", "/c/f"]
```

Now, the loop has ended, and the `ans` list contains all the unique folder paths, excluding any sub-folders that were removed:

Result: `["/a", "/c/d", "/c/f"]`

Finally, we return this list as the solution. The whole process demonstrates how the combination of sorting and prefix comparison efficiently filters out the sub-folders.

Solution Implementation

Python

```
class Solution:
    def removeSubfolders(self, folders: List[str]) -> List[str]:
        # Sort the list of folder paths to ensure that parent directories are before their subfolders
        folders.sort()

        # Initialize the result list with the first folder since it is guaranteed not to be a subfolder
        result = [folders[0]]

        # Iterate through the sorted folders starting from the second folder
        for folder in folders[1:]:
            last_added_folder_length = len(result[-1])  # Length of the last added folder to result
            current_folder_length = len(folder)         # Length of the current folder

            # Check if the current folder is not a subfolder of the last added folder
            # This is determined by checking if the last added folder does not match the prefix of the current folder and there is a
            if last_added_folder_length >= current_folder_length or not (result[-1] == folder[:last_added_folder_length] and folder[
                result.append(folder)  # The current folder is not a subfolder, so add it to the result

        # Return the list of folders without subfolders
        return result
```

Java

```
class Solution {
    public List<String> removeSubfolders(String[] folders) {
        // Sort the array of folder paths to ensure that parent folders come before their subfolders.
        Arrays.sort(folders);

        // Initialize the answer list with the first folder path since it can not be a subfolder.
        List<String> filteredFolders = new ArrayList<>();
        filteredFolders.add(folders[0]);

        // Iterate through the sorted folder paths starting from the index 1.
        for (int i = 1; i < folders.length; ++i) {
            // Get the length of the last added folder path in the answer list.
            int lastAddedFolderPathLength = filteredFolders.get(filteredFolders.size() - 1).length();

            // Get the current folder path length.
            int currentFolderPathLength = folders[i].length();

            // Check if the last added folder is a prefix of the current folder and if there is a '/' right after it.
            // If the last added folder path is not a prefix of the current folder (or it is the complete current folder),
            // or the character just after the prefix is not '/', then it is not a subfolder.
            if (lastAddedFolderPathLength >= currentFolderPathLength
                || !(filteredFolders.get(filteredFolders.size() - 1).equals(folders[i].substring(0, lastAddedFolderPathLength))
                    && folders[i].charAt(lastAddedFolderPathLength) == '/')) {
                // If the current folder is not a subfolder, add it to the filtered list.
                filteredFolders.add(folders[i]);
            }
        }

        // Return the list of filtered folders with subfolders removed.
        return filteredFolders;
    }
}
```

C++

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

class Solution {
public:
    // This function takes a list of folder paths and removes any subfolders
    // from the list since a subfolder is implicitly included when its parent
    // folder is included.
    vector<string> removeSubfolders(vector<string>& folders) {
        // Sort the folder list to ensure that subfolders follow their parents
        sort(folders.begin(), folders.end());

        // Initialize the answer list with the first (smallest) folder
        vector<string> filteredFolders = {folders[0]};

        // Iterate through the sorted folders starting from the second element
        for (int i = 1; i < folders.size(); ++i) {
            int parentSize = filteredFolders.back().size();
            int currentSize = folders[i].size();

            // Check if the current folder is a subfolder of the last parent folder
            // Subfolder check: the parent is a prefix and is followed by a '/' in the current folder
            if (parentSize >= currentSize ||
                !(folders[i].substr(0, parentSize) == filteredFolders.back() &&
                  folders[i][parentSize] == '/')) {
                // If the current folder is not a subfolder, add it to the answer list
                filteredFolders.emplace_back(folders[i]);
            }
        }

        // Return the list of filtered folders
        return filteredFolders;
    }
};
```

TypeScript

```
function removeSubfolders(folders: string[]): string[] {
    // Sort the folder list to ensure that subfolders follow their parent folders.
    folders.sort();

    // Initialize the filtered folder array with the first (lexicographically smallest) folder.
    const filteredFolders: string[] = [folders[0]];

    // Iterate through the sorted folders, starting with the second element.
    for (let i = 1; i < folders.length; i++) {
        const parentSize = filteredFolders[filteredFolders.length - 1].length;
        const currentSize = folders[i].length;

        // If the current folder is a subfolder of the last folder in filteredFolders, we skip it.
        // To determine if it's a subfolder, we check if the last folder in filteredFolders is a prefix
        // of the current folder followed by a '/'.
        if (!(parentSize < currentSize &&
              folders[i].substring(0, parentSize) === filteredFolders[filteredFolders.length - 1] &&
              folders[i][parentSize] === '/')) {
            // If the current folder is not a subfolder, add it to the filtered list.
            filteredFolders.push(folders[i]);
        }
    }

    // Return the list of filtered folders.
    return filteredFolders;
}


class Solution:
    def removeSubfolders(self, folders: List[str]) -> List[str]:
        # Sort the list of folder paths to ensure that parent directories are before their subfolders
        folders.sort()

        # Initialize the result list with the first folder since it is guaranteed not to be a subfolder
        result = [folders[0]]

        # Iterate through the sorted folders starting from the second folder
        for folder in folders[1:]:
            last_added_folder_length = len(result[-1])  # Length of the last added folder to result
            current_folder_length = len(folder)         # Length of the current folder

            # Check if the current folder is not a subfolder of the last added folder
            # This is determined by checking if the last added folder does not match the prefix of the current folder and there is a
            if last_added_folder_length >= current_folder_length or not (result[-1] == folder[:last_added_folder_length] and folder[
                result.append(folder)  # The current folder is not a subfolder, so add it to the result

        # Return the list of folders without subfolders
        return result
```

Time and Space Complexity

Time Complexity

The given algorithm's time complexity mainly consists of two parts: sorting the `folder` list and iterating through the sorted list to check for subfolders.

1. Sorting the `folder` list using the `sort()` method has a time complexity of $O(n \log n)$, where n is the number of elements in the `folder` list. Sorting is necessary to ensure that any potential subfolder appears immediately after its parent folder, which enables the subsequent linear check.
2. The second part of the algorithm iterates through the sorted `folder` list, performing a constant-time string comparison (checking prefix and the following character) for each pair of adjacent folders. This process has a time complexity of $O(n * m)$, where n is the number of folders, and m is the average length of a folder's path because a folder's full path may need to be traversed to perform the comparison.

Overall, the time complexity of the given algorithm can be expressed as $O(n \log n + n * m)$. Since $O(n \log n)$ is dominated by $O(n * m)$ for large values of n, you could consider the overall time complexity to be $O(n * m)$, where m is the average string length and n is the number of strings.

Space Complexity

The space complexity of the algorithm is determined by the additional space used:

1. The `ans` list is used to store the resulting list of folder paths that are not subfolders of any other folders in the list. In the worst case, none of the folders are subfolders of others, and hence, the `ans` list will contain all the folder paths. This gives us a space complexity of $O(n * m)$, where n is the number of folders and m is the average length of a folder's path.
2. No other additional data structures that grow with input size are used, so other space considerations are constant and can be ignored in Big O notation.

Therefore, the overall space complexity is $O(n * m)$, which accounts for the space required to store the list of folder paths in the `ans` list.