1487. Making File Names Unique

Medium Array Hash Table String

Leetcode Link

Problem Description

The problem requires creating a file system with folders named according to the input array `names`. Whenever a folder is created, it should have a unique name. If a name has been used before, a suffix in the format of `(k)` must be appended, where `k` is the smallest positive integer that makes the name unique again. The task is to simulate this process and return the final list of names used for the folders.

For instance, if `names` is ["doc", "doc", "image", "doc(1)", "doc"], the output should be ["doc", "doc(1)", "image", "doc(1)(1)", "doc(2)"]. This array represents the names of the folders after every minute following the naming rules defined by the problem.

Intuition

The intuition behind the solution involves tracking each folder name and the suffixes applied to resolve the name conflicts. We use a dictionary (hashmap) to keep track of the names and their corresponding suffixes to achieve this efficiently.

We iterate over the input list of names. For each name, we perform the following steps:

1. *Check if the name already exists:* If the name exists in the dictionary, it means we've encountered a conflict and we need to find the next suitable suffix to make the name unique.
2. *Resolve the conflict:* Increase the numeral suffix starting from the one recorded in the dictionary until we find a name that is not already used.
3. *Update the name with the suffix:* Once a unique name is constructed, we update the current name in the list of names with the newly formed unique name.
4. *Update the dictionary:* Store the changed name with a suffix of 1 if it's a new name. If the name was altered and had a suffix added, update the original conflicting name's suffix for future conflict resolution.

The logic behind incrementing the suffix numeral ensures that we will always find the smallest possible integer `k` to append while maintaining uniqueness. The defaultdict from the collections module provides convenience for this process since it will automatically assume the default integer value of 0 when accessing a new name not yet in the dictionary.

By the end of the input list iteration, we have a list representing the actual names the system has assigned to each folder.

Solution Approach

The implementation of the provided solution uses a combination of a loop and a dictionary (hashmap) to achieve its objectives. Here's a step-by-step explanation of how the code works:

- **Data Structure:** We use a `defaultdict` from the `collections` module with integer values that defaults to 0. This helps in easily tracking the next available integer suffix that should be used for each unique base name.
- **Iterate Through Names:** We iterate over the input list of names with the help of a `for` loop. Each name encountered is evaluated to determine if it's unique or if a suffix is needed.
- **Check and Resolve Name Conflicts:**
  - If the current name is present in the dictionary, it indicates a conflict. We then initiate a while loop to find a unique name by incrementing the integer suffix.
  - Inside the loop (`while f'{name}({k})' in d`), we keep appending a suffix(`k`) to the original name and increment `k` to find a non-conflicting name.
  - Once a unique name is found, we assign this new name to the correct position in the `names` list (using `names[i] = f'{name}({k})'`) and update the dictionary with the new suffix (`d[name] = k + 1`) to indicate the next available suffix for this base name.
- **Final Names Update and Dictionary Maintenance:**
  - If the current name isn't in the dictionary, it's already unique, and we simply store it with a suffix of 1 (which means `d[names[i]] = 1`) in the dictionary for potential future conflicts.
  - If the name was modified by adding a suffix, we've already updated the entry in the dictionary with the next suffix to be used.
- **End of Loop:** Each iteration of the loop either adds a unique name directly to the dictionary or resolves a conflict by appending the appropriate suffix. By the end of the loop, all conflicts are resolved.
- **Return the Modified List:** The `names` list, which now contains all the unique names assigned by the system, is returned at the end of the function.

This solution's complexity lies in the loop and the operations within it. The average case time complexity is O(n*k), where `n` is the length of the `names` list and `k` represents the time taken to find a unique name through the incremental suffix process. However, the worst-case time complexity can increase if there is a large number of conflicts that result in longer searches for a unique name.

Example Walkthrough

Let's use a small example to illustrate the solution approach. Consider the input array `names` as `["file", "file", "file", "file"]`. The aim is to create a list where all folder names are unique following the rules described in the problem statement.

1. Start with an empty `defaultdict` and iterate through the `names` list.
2. First, encounter "file". "file" is not in the dictionary, so we add "file" to the dictionary with a value of 1 (meanwhile it's also added to our final names list).

   Updated names list: `["file"]` Dictionary: `{"file": 1}`
3. Next, see another "file". Since "file" is in the dictionary (with a value `1`), there's a conflict. We append `(1)` to "file" to resolve this, resulting in "file(1)".

   Updated names list: `["file", "file(1)"]` Dictionary before incrementing: `{"file": 1}` Increment the suffix in the dictionary: `{"file": 2}` (for potential future conflicts)
4. Encounter another "file". Check the dictionary. Since "file" is there with the value `2`, append `(2)` and increment the suffix.

   Updated names list: `["file", "file(1)", "file(2)"]` Dictionary: `{"file": 3}`
5. We see "file" again. Repeat the check and update steps, the suffix this time is `(3)`.

   Final names list: `["file", "file(1)", "file(2)", "file(3)"]` Final dictionary: `{"file": 4}`
6. The iteration ends as there are no more names in the `names` array.

Our final list is `["file", "file(1)", "file(2)", "file(3)"]`, where each "file" is unique. This shows how the aforementioned algorithm successfully solves the file naming problem by systematically ensuring each new folder name has not been used before, appending the smallest possible unique suffix when necessary.

Python Solution

```python
from collections import defaultdict

class Solution:
    def getFolderNames(self, names: List[str]) -> List[str]:
        # Create a dictionary to store the counts of each folder name.
        name_counts = defaultdict(int)

        # Iterate over all the names in the input list.
        for index, name in enumerate(names):
            # Check if the name already exists in the dictionary.
            if name in name_counts:
                # Start with the current count for this name.
                count = name_counts[name]
                # Check if the name with the appended count exists.
                # If it does, increment the count and check again.
                while f'{name}({count})' in name_counts:
                    count += 1
                # Update the count for this name in the dictionary.
                name_counts[name] = count + 1
                # Modify the current name to include the count in the format "name(count)".
                names[index] = f'{name}({count})'
            # Add or update the name in the dictionary with a count of 1,
            # indicating this modified name is now used once.
            name_counts[names[index]] = 1

        # Return the list with unique folder names.
        return names
```

Java Solution

```java
class Solution {
    public String[] getFolderNames(String[] names) {
        // Create a map to store the highest integer k used for each original name.
        Map<String, Integer> nameMap = new HashMap<>();

        // Iterate through each name in the input array.
        for (int i = 0; i < names.length; ++i) {
            // If the current name has already been encountered,
            // append a unique identifier to make it distinct.
            if (nameMap.containsKey(names[i])) {
                // Get the current highest value of k used for this name.
                int k = nameMap.get(names[i]);
                // Check if the name with the appended "(k)" already exists.
                // If it does, increment k until a unique name is found.
                while (nameMap.containsKey(names[i] + "(" + k + ")")) {
                    ++k;
                }

                // Update the map with the new highest value of k for this name.
                nameMap.put(names[i], k);
                // Modify the current name by appending "(k)" to make it unique.
                names[i] += "(" + k + ")";
            }

            // Insert or update the current name in the map, setting its value to 1.
            // If the name is new, this records it as seen for the first time.
            // If the name has been modified, this ensures it's treated as new.
            nameMap.put(names[i], 1);
        }

        // Return the modified array of names, with each name now unique.
        return names;
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <string>
#include <unordered_map>
using namespace std;

class Solution {
public:
    vector<string> getFolderNames(vector<string>& names) {
        unordered_map<string, int> nameCountMap; // Map to keep a count of folder names

        for (auto& name : names) {
            // Check if 'name' already appeared and get its count
            int count = nameCountMap[name];

            if (count > 0) {
                // If name exists, find a unique name by appending the count in parentheses
                string newName;
                do {
                    count++; // Increment the count
                    newName = name + "(" + to_string(count) + ")";
                } while (nameCountMap[newName] > 0); // Check if the newName is also taken, continue if yes

                nameCountMap[name] = count; // Update count for the original name
                name = newName; // Use the newName for the current iteration
            }

            // If name is original or new unique name is found, set its count to 1
            nameCountMap[name] = 1;
        }

        return names; // Return the vector with unique folder names
    }
};
```

Typescript Solution

```typescript
function getFolderNames(names: string[]): string[] {
    // Initialize a map to keep track of the counts of each folder name
    const nameCounts: Map<string, number> = new Map();

    // Iterate over the list of names
    for (let i = 0; i < names.length; ++i) {
        // Check if the current name already exists in the map
        if (nameCounts.has(names[i])) {
            // Retrieve the count for the current name, defaulting to 0 if it's not found
            let count: number = nameCounts.get(names[i]) || 0;

            // Create a unique folder name by appending count in parentheses
            // Keep incrementing count until a unique name is found
            while (nameCounts.has(`${names[i]}(${count})`)) {
                ++count;
            }

            // Update the count for the current name in the map
            nameCounts.set(names[i], count);

            // Append the count to the current folder name to make it unique
            names[i] += `(${count})`;
        }

        // Set the count for this unique folder name in the map, or update it to 1
        nameCounts.set(names[i], 1);
    }

    // Return the transformed list of names with all unique folders
    return names;
}
```

Time and Space Complexity

The provided code snippet maintains a dictionary to keep track of the number of times a particular folder name has appeared. It then generates unique names by appending a number in parentheses if necessary. Let's analyze the time and space complexity:

Time Complexity

1. The `for` loop iterates through each name in the input list, so it gives us an `O(n)` term, where `n` is the number of names in the list.
2. Inside the loop, the code checks if `name` exists in the dictionary with `if name in d`, which is an `O(1)` operation due to hashing.
3. If there is a conflict for a name, the code enters into a `while` loop to find the correct suffix `k`. In the worst case, it may run as many times as there are names with the same base, let's call this number `m`.
4. Each iteration of the `while` loop involves a string concatenation and a dictionary lookup, both of which are `O(1)` operations.

The `while` loop could contribute significantly to the time complexity if there are many names with the same base. If `m` is the maximum number of duplicates for any name, the worst-case time complexity for `while` loop iterations across the entire input is `O(mem)` because each unique name will go through the loop at most `m` times.

Therefore, the overall worst-case time complexity is `O(n + mem)` which simplifies to `O(mem)`.

Space Complexity

The space complexity of the algorithm can be broken down as:

1. The dictionary `d`, which stores each unique folder name. In the worst case (all folder names are unique, and for each original name, there's a different number of folders), this dictionary could potentially store 2 entries per name (the original and the latest modified version). So, in the worst case, it could hold up to `2n` entries, contributing `O(n)` to the space complexity.
2. Temporary storage for constructing new folder names during the `while` loop does not significantly add to the space complexity because it only holds a single name at a time.

Therefore, the overall space complexity is `O(n)`.

To summarize:

- Time Complexity: `O(mem)`
- Space Complexity: `O(n)`