609. Find Duplicate File in System

Medium Array Hash Table String   Leetcode Link

Problem Description

The problem presents a scenario where we have a file system that consists of several directories and files. Each directory can have multiple files with their contents. We are given a list of directory info strings, where each string has the following format: a path to the directory followed by one or more files, with each file's name and content enclosed in parentheses.

Our task is to find all the duplicate files in the file system. A duplicate file is defined as a file that has the exact same content as one or more other files, regardless of the file or directory names. The output should be a list of groups, with each group containing the file paths of all files that have identical content.

Here's a breakdown of a directory info string:

- `"root/d1/d2/.../dm"` represents the path to the directory.
- `f1.txt(f1_content)` represents a file within the directory, where `f1.txt` is the file name, and `f1_content` is the content of the file. There can be multiple such file representations in a single directory info string.

The solution should return groups of file paths for all sets of duplicate files. Each file path in the output is formatted as `"directory_path/file_name.txt"`.

Intuition

To solve this problem, we need to efficiently group the files based on their contents. Here's a straightforward approach to tackle this:

1. Iterate through each directory info string in the given `paths` list.
2. For each directory info string, separate the directory path from the files. Then, for each file, extract the name and content.
3. Use a data structure that allows easy grouping of files with the same content. A dictionary (or hashmap) is a good choice for this task. We can use the file content as the key and a list of file paths as the value.
4. For each file, append the full file path (directory path + '/' + file name) to the list in the dictionary corresponding to the file's content.
5. Once all files are processed, we only need the lists that have more than one file path, since these represent duplicate files.

With this approach, we're using the file contents as a unique identifier to find duplicates, which satisfies the condition that duplicate files consist of at least two files that have the content.

The solution code creates a `defaultdict` of lists, goes through each file, extracts the content, and appends the full path to the corresponding list in the dictionary. Finally, it builds the result by including only the lists with more than one file path, as these are the duplicates.

The intuition behind this approach is that by mapping file contents to file paths, we can quickly identify which files are duplicates without having to compare the contents of every file with every other file, which would be less efficient.

Solution Approach

The solution uses a simple yet effective approach, leveraging a `defaultdict` to group files by their contents. The implementation steps are as follows:

1. **Initialize a defaultdict**: We start by creating a `defaultdict` of lists. This particular data structure is chosen because it simplifies the process of appending to a list for a given key without first checking if the key exists in the dictionary.

2. **Split the input**: Iterate through each entry in the `paths` list. For every path string, we split it into components using the `split()` method. The first component is the directory path, and the rest are files with content.

3. **Process each file**: For each file, find the index of the opening parenthesis (`(`) to separate the file name from its content.

4. **Extracting name and content**: The substring before the parenthesis is the file name, and the substring within the parenthesis is the file content.

5. **Construct the full path and group by content**: Use the extracted name and content to form the full file path (concatenating the directory path, `'/'`, and the file name). Append this path to the list in the `defaultdict` associated with the file content.

6. **Filter out unique files**: After all paths have been processed, we filter the `defaultdict` to only include lists with more than one path, meaning these paths have duplicate contents.

7. **Return the result**: The final step is to return the lists of file paths that have duplicates.

The algorithm's complexity is efficient as it consists mostly of linear operations: splitting strings, indexing, and dictionary operations. The use of `defaultdict` and list appends are constant time operations on average, making the solution scalable for a large number of files and directories.

The solution code completes this process with a concise script that translates the conceptual steps into Python code fluently, leveraging Python's built-in string manipulation and dictionary methods to create an elegant and efficient solution.

Here is the main part of the code with comments to demonstrate the implementation of the approach:

```python
class Solution:
    def findDuplicate(self, paths: List[str]) -> List[List[str]]:
        d = defaultdict(list)  # Step 1: Initialize a defaultdict
        for p in paths:  # Step 2: Split the input
            ps = p.split()
            for f in ps[1:]:
                i = f.find('(')  # Step 3: Process each file
                # Step 4: Extracting name and content
                name, content = f[:i], f[i + 1 : -1]
                # Step 5: Construct the full path and group by content
                d[content].append(ps[0] + '/' + name)
        # Step 6: Filter out unique files
        return [v for v in d.values() if len(v) > 1]
```

The key to this solution is the grouping of files by their content using a dictionary, making the task of finding duplicates straightforward.

Example Walkthrough

Let's illustrate the solution approach with a small example.

Given the following input list of directory info strings:

```
paths = ["root/a 1.txt(abcd) 2.txt(efgh)", "root/c 3.txt(abcd)", "root/c/d 4.txt(efgh)", "root 4.txt(efgh)"]
```

This list tells us we have 4 paths:

1. "root/a" contains "1.txt" with content "abcd" and "2.txt" with content "efgh".
2. "root/c" contains "3.txt" with content "abcd".
3. "root/c/d" contains "4.txt" with content "efgh".
4. "root" contains "4.txt" with content "efgh".

We need to find duplicate files (files with the same content) and group them.

- Step 1: Initialize a defaultdict called "d".
- Step 2: Split each path string based on space to separate the directory path and the files within it.
- Step 3: For each file in the directory string, find the parenthesis to separate the file name and content.
- Step 4: Extract the file name which is before the parenthesis and the file content which is within the parentheses.
- Step 5: Form the full file path by concatenating the directory path with the file name and append this path to the corresponding list in the defaultdict indicated by the file content.
- Step 6: After processing all paths, we filter the defaultdict to include only those lists that have more than one file path, which indicate duplicates.

For the given example, the process will work as follows:

- `root/a` is split into `["root/a", "1.txt(abcd)", "2.txt(efgh)"]`.
  - The file `"1.txt(abcd)"` is processed to get the name "1.txt" and content "abcd".
  - The full path becomes `"root/a/1.txt"` and is appended to the list for content "abcd".
  - Similarly for `"2.txt(efgh)"`, producing the full path `"root/a/2.txt"` and appending to the list for content "efgh".
- The same process follows for each entry in `paths`. After processing, the defaultdict `d` looks like this:

```
{
    "abcd": ["root/a/1.txt", "root/c/3.txt"],  # Duplicate content
    "efgh": ["root/a/2.txt", "root/c/d/4.txt", "root/4.txt"]  # Duplicate content
}
```

- Step 6: We only want lists with more than one file path:

  Our final output will be:

```
[
    ["root/a/1.txt", "root/c/3.txt"],
    ["root/a/2.txt", "root/c/d/4.txt", "root/4.txt"]
]
```

These are the groups of duplicate files. The list for "abcd" content shows that "1.txt" in the "root/a" directory is a duplicate of "3.txt" in the "root/c" directory. Similarly, for the "efgh" content group.

Python Solution

```python
from collections import defaultdict
from typing import List

class Solution:
    def findDuplicate(self, paths: List[str]) -> List[List[str]]:
        # Initialize a dictionary which maps file content to a list of file paths
        content_to_paths = defaultdict(list)

        # Iterate over each path in the input list
        for path_string in paths:
            # Split the path string into the directory and the file detail parts
            path_parts = path_string.split()
            directory = path_parts[0]

            # For each file in the current directory path
            for file_detail in path_parts[1:]:
                # Find the index of the opening bracket '(' which starts the file content
                content_start_idx = file_detail.find('(')
                # Extract the file name and content
                file_name = file_detail[:content_start_idx]
                content = file_detail[content_start_idx + 1 : -1]  # Exclude the closing bracket

                # Append the full file path to the list of paths corresponding to the content
                full_path = directory + '/' + file_name
                content_to_paths[content].append(full_path)

        # Filter and return only the lists of file paths with more than one file path
        # This indicates that files are duplicates as per their content
        return [paths_list for paths_list in content_to_paths.values() if len(paths_list) > 1]

```

Java Solution

```java
class Solution {
    public List<List<String>> findDuplicate(String[] paths) {
        // Create a map to hold file content as key and a list of file paths as value
        Map<String, List<String>> contentToPathsMap = new HashMap<>();

        // Loop over each path in the input array
        for (String path : paths) {
            // Split the current path into directory and files with content
            String[] parts = path.split(" ");

            // Skip the first part as it is the directory path and iterate over the files
            for (int i = 1; i < parts.length; ++i) {
                // Find the index of the first '(' to separate file name and content
                int contentStartIndex = parts[i].indexOf('(');
                // Extract the content between parentheses
                String content = parts[i].substring(contentStartIndex + 1, parts[i].length() - 1);
                // Construct the full file path using the directory and the file name
                String fullPath = parts[0] + '/' + parts[i].substring(0, contentStartIndex);
                // Add the full file path to the list of paths for the current content
                contentToPathsMap.computeIfAbsent(content, k -> new ArrayList<>()).add(fullPath);
            }
        }

        // Prepare the result list for duplicate files
        List<List<String>> duplicates = new ArrayList<>();

        // Loop over the entries in the content to paths map
        for (List<String> pathsList : contentToPathsMap.values()) {
            // Only consider lists with more than one file path (indicating duplicates)
            if (pathsList.size() > 1) {
                duplicates.add(pathsList);
            }
        }

        // Return the list of duplicates
        return duplicates;
    }
}

```

C++ Solution

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

class Solution {
public:
    // Finds duplicates in the file system given list of directory paths
    vector<vector<string>> findDuplicate(vector<string>& paths) {
        // This map will store the content of the file as the key and a list of file paths with that content as the value
        unordered_map<string, vector<string>> contentToFilePathsMap;

        // Iterate over each path provided in the input
        for (auto& path : paths) {
            // Split the path string based on spaces. The first element is the directory path and the remaining elements are files
            vector<string> pathAndFiles = split(path, ' ');
            // Iterate over the files
            for (size_t i = 1; i < pathAndFiles.size(); ++i) {
                // Find the position of the first parenthesis which marks the start of the file content
                size_t contentStartPos = pathAndFiles[i].find('(');
                // Extract the file content (excluding the enclosing parentheses)
                string content = pathAndFiles[i].substr(contentStartPos + 1, pathAndFiles[i].size() - contentStartPos - 2);
                // Build the full file path by concatenating directory path and the file name
                string fullPath = pathAndFiles[0] + '/' + pathAndFiles[i].substr(0, contentStartPos);
                // Store the full path in the map under the corresponding content key
                contentToFilePathsMap[content].push_back(fullPath);
            }
        }

        // Vector to store the final result
        vector<vector<string>> duplicates;
        // Iterate over the map to check for duplicate contents
        for (auto& mapEntry : contentToFilePathsMap) {
            // If there is more than one file with the same content, it is considered a duplicate
            if (mapEntry.second.size() > 1) {
                // Add the list of file paths with the same content to the result
                duplicates.push_back(mapEntry.second);
            }
        }

        // Return the list of duplicate file groups
        return duplicates;
    }

    // A helper function to split a string by a given delimiter
    vector<string> split(const string& str, char delimiter) {
        vector<string> tokens;
        stringstream ss(str);
        string token;

        // Split the string by the delimiter and push each token into the vector
        while (getline(ss, token, delimiter)) {
            tokens.push_back(token);
        }
        return tokens;
    }
};

```

Typescript Solution

```typescript
function findDuplicate(paths: string[]): string[][] {
    // Map to store the content of the files and their paths
    const fileContentMap = new Map<string, string[]>();

    // Iterate over each path provided
    for (const path of paths) {
        // Split the path into root directory and files
        const [rootDirectory, ...files] = path.split(' ');

        // Iterate over each file in the current path
        for (const file of files) {
            // Split the file information into name and content (the Boolean filter removes empty strings caused by splitting)
            const [fileName, fileContent] = file.split(/\(|\)/g).filter(Boolean);

            // Get the existing list of file paths with the same content, or initialize an empty one
            const existingPaths = fileContentMap.get(fileContent) ?? [];

            // Add the new file path to the list of paths for this content
            existingPaths.push(`${rootDirectory}/${fileName}`);

            // Update the map with the new list of paths for this content
            fileContentMap.set(fileContent, existingPaths);
        }
    }

    // Filter out the file contents that have more than one path and return the values of the map
    return [...fileContentMap.values()].filter(paths => paths.length > 1);
}

```

Time and Space Complexity

Time Complexity

The time complexity of the function is determined by several factors:

1. Splitting each path string, which takes `O(N)` time where `N` is the total length of the string.
2. Iterating through each file within each path, which takes `O(M)` time where `M` is the number of files across all paths.
3. Finding the index of the first parenthesis in each file name, which takes `O(K)` time where `K` is the length of the file name.
4. Inserting the path and file name into the dictionary, which takes `O(1)` for each operation assuming average case scenario for the hash table insert.

Let's assume `P` is the number of paths and each path contains at most `F` files, and the longest file is of `L` characters. The split operation initially takes `O(P*L)`, and then each file of each path will be accessed and splitted which is `O(P*F*L)`.

Thus, overall the time complexity is `O(P*L + P*F*L)` which simplifies to `O(P*F*L)` - as we assume that each file is also bounded by the same `L`.

Space Complexity

The space complexity includes:

1. The dictionary storing all the file contents as keys, and the associated file paths as values. In the worst-case scenario, each file has unique content, so the space complexity would be `O(M)` where `M` is the number of files.
2. Temporary variables for processing, such as storing the name and content of each file. This does not significantly add to the space complexity, as they're overwritten for each file.

Thus, the space complexity is primarily dictated by the size of the dictionary, which in the worst case is `O(P*F)` to store all unique file content-path pairs. Therefore, the space complexity is `O(P*F)`.

Note: If file content is very large, it could dominate the space complexity, and we might express it as `O(P*F*C)` where `C` is the average content size.