Problem 609: Find Duplicate File in System

Problem Information

Difficulty: Medium
Acceptance Rate: 0.00%
Paid Only: No

Problem Description

Given a list

paths

of directory info, including the directory path, and all the files with contents in this directory, return

all the duplicate files in the file system in terms of their paths

. You may return the answer in

any order

.

A group of duplicate files consists of at least two files that have the same content.

A single directory info string in the input list has the following format:

"root/d1/d2/.../dm f1.txt(f1_content) f2.txt(f2_content) ... fn.txt(fn_content)"

It means there are

n

files

(f1.txt, f2.txt ... fn.txt)

with content

(f1_content, f2_content ... fn_content)

respectively in the directory "

root/d1/d2/.../dm"

. Note that

n >= 1

and

m >= 0

. If

m = 0

, it means the directory is just the root directory.

The output is a list of groups of duplicate file paths. For each group, it contains all the file paths of the files that have the same content. A file path is a string that has the following format:

"directory_path/file_name.txt"

Example 1:

Input:

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

Output:

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

Example 2:

Input:

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

Output:

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

Constraints:

1 <= paths.length <= 2 * 10

4

1 <= paths[i].length <= 3000

1 <= sum(paths[i].length) <= 5 * 10

5

paths[i]

consist of English letters, digits,

'/'

,

'.'

,

'('

,

')'

, and

' '

.

You may assume no files or directories share the same name in the same directory.

You may assume each given directory info represents a unique directory. A single blank space separates the directory path and file info.

Follow up:

Imagine you are given a real file system, how will you search files? DFS or BFS?

If the file content is very large (GB level), how will you modify your solution?

If you can only read the file by 1kb each time, how will you modify your solution?

What is the time complexity of your modified solution? What is the most time-consuming part and memory-consuming part of it? How to optimize?

How to make sure the duplicated files you find are not false positive?

Code Snippets

C++:

```
class Solution {
public:
vector<vector<string>> findDuplicate(vector<string>& paths) {

}
};
```

Java:

```
class Solution {
public List<List<String>> findDuplicate(String[] paths) {

}
}
```

Python3:

```
class Solution:
def findDuplicate(self, paths: List[str]) -> List[List[str]]:
```

Python:

```
class Solution(object):
def findDuplicate(self, paths):
"""
:type paths: List[str]
:rtype: List[List[str]]
"""
```

JavaScript:

```
/**
* @param {string[]} paths
* @return {string[][]}
*/
var findDuplicate = function(paths) {

};
```

TypeScript:

```
function findDuplicate(paths: string[]): string[][] {

};
```

C#:

```
public class Solution {
public IList<IList<string>> FindDuplicate(string[] paths) {

}
}
```

C:

```
/**
* Return an array of arrays of size *returnSize.
* The sizes of the arrays are returned as *returnColumnSizes array.
* Note: Both returned array and *columnSizes array must be malloced, assume
caller calls free().
*/
char*** findDuplicate(char** paths, int pathsSize, int* returnSize, int**
returnColumnSizes) {

}
```

Go:

```
func findDuplicate(paths []string) [][]string {

}
```

Kotlin:

```
class Solution {
fun findDuplicate(paths: Array<String>): List<List<String>> {

}
}
```

Swift:

```
class Solution {
func findDuplicate(_ paths: [String]) -> [[String]] {

}
}
```

Rust:

```
impl Solution {
pub fn find_duplicate(paths: Vec<String>) -> Vec<Vec<String>> {

}
}
```

Ruby:

```
# @param {String[]} paths
# @return {String[][]}
def find_duplicate(paths)

end
```

PHP:

```
class Solution {

/**
* @param String[] $paths
* @return String[][]
*/
function findDuplicate($paths) {

}
}
```

Dart:

```
class Solution {
List<List<String>> findDuplicate(List<String> paths) {

}
}
```

Scala:

```
object Solution {
def findDuplicate(paths: Array[String]): List[List[String]] = {

}
}
```

Elixir:

```
defmodule Solution do
@spec find_duplicate(paths :: [String.t]) :: [[String.t]]
def find_duplicate(paths) do
```

```
end
end
```

Erlang:

```
-spec find_duplicate(Paths :: [unicode:unicode_binary()]) ->
[[unicode:unicode_binary()]].
find_duplicate(Paths) ->
.
```

Racket:

```
(define/contract (find-duplicate paths)
(-> (listof string?) (listof (listof string?)))
)
```

Solutions

C++ Solution:

```
/*
* Problem: Find Duplicate File in System
* Difficulty: Medium
* Tags: array, string, tree, hash, search
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(h) for recursion stack where h is height
*/

class Solution {
public:
vector<vector<string>> findDuplicate(vector<string>& paths) {

}
};
```

Java Solution:

```
/**
* Problem: Find Duplicate File in System
* Difficulty: Medium
* Tags: array, string, tree, hash, search
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(h) for recursion stack where h is height
*/

class Solution {
public List<List<String>> findDuplicate(String[] paths) {

}
}
```

Python3 Solution:

```
"""
Problem: Find Duplicate File in System
Difficulty: Medium
Tags: array, string, tree, hash, search

Approach: Use two pointers or sliding window technique
Time Complexity: O(n) or O(n log n)
Space Complexity: O(h) for recursion stack where h is height
"""

class Solution:
def findDuplicate(self, paths: List[str]) -> List[List[str]]:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def findDuplicate(self, paths):
"""
:type paths: List[str]
:rtype: List[List[str]]
"""
```

JavaScript Solution:

```
/**
* Problem: Find Duplicate File in System
* Difficulty: Medium
* Tags: array, string, tree, hash, search
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(h) for recursion stack where h is height
*/

/**
* @param {string[]} paths
* @return {string[][]}
*/
var findDuplicate = function(paths) {

};
```

TypeScript Solution:

```
/**
* Problem: Find Duplicate File in System
* Difficulty: Medium
* Tags: array, string, tree, hash, search
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(h) for recursion stack where h is height
*/

function findDuplicate(paths: string[]): string[][] {

};
```

C# Solution:

```
/*
* Problem: Find Duplicate File in System
* Difficulty: Medium
* Tags: array, string, tree, hash, search
*
```

```
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(h) for recursion stack where h is height
*/

public class Solution {
public IList<IList<string>> FindDuplicate(string[] paths) {

}
}
```

C Solution:

```
/*
* Problem: Find Duplicate File in System
* Difficulty: Medium
* Tags: array, string, tree, hash, search
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(h) for recursion stack where h is height
*/

/**
* Return an array of arrays of size *returnSize.
* The sizes of the arrays are returned as *returnColumnSizes array.
* Note: Both returned array and *columnSizes array must be malloced, assume
caller calls free().
*/
char*** findDuplicate(char** paths, int pathsSize, int* returnSize, int**
returnColumnSizes) {

}
```

Go Solution:

```
// Problem: Find Duplicate File in System
// Difficulty: Medium
// Tags: array, string, tree, hash, search
//
// Approach: Use two pointers or sliding window technique
```

```
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(h) for recursion stack where h is height

func findDuplicate(paths []string) [][]string {

}
```

Kotlin Solution:

```
class Solution {
fun findDuplicate(paths: Array<String>): List<List<String>> {

}
}
```

Swift Solution:

```
class Solution {
func findDuplicate(_ paths: [String]) -> [[String]] {

}
}
```

Rust Solution:

```
// Problem: Find Duplicate File in System
// Difficulty: Medium
// Tags: array, string, tree, hash, search
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(h) for recursion stack where h is height

impl Solution {
pub fn find_duplicate(paths: Vec<String>) -> Vec<Vec<String>> {

}
}
```

Ruby Solution:

```
# @param {String[]} paths
# @return {String[][]}
def find_duplicate(paths)

end
```

PHP Solution:

```
class Solution {

/**
* @param String[] $paths
* @return String[][]
*/
function findDuplicate($paths) {

}
}
```

Dart Solution:

```
class Solution {
List<List<String>> findDuplicate(List<String> paths) {

}
}
```

Scala Solution:

```
object Solution {
def findDuplicate(paths: Array[String]): List[List[String]] = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec find_duplicate(paths :: [String.t]) :: [[String.t]]
def find_duplicate(paths) do

end
```

```
end
```

Erlang Solution:

```
-spec find_duplicate(Paths :: [unicode:unicode_binary()]) ->
[[unicode:unicode_binary()]].
find_duplicate(Paths) ->
.
```

Racket Solution:

```
(define/contract (find-duplicate paths)
(-> (listof string?) (listof (listof string?)))
)
```