1593. Split a String Into the Max Number of Unique Substrings

Medium | Hash Table | String | Backtracking

Leetcode Link

Problem Description

Given a string `s`, we are tasked to find the maximum number of non-empty, unique substrings into which the string can be split. The goal is to make sure that after splitting, each substring is distinct, and their concatenation would result in the original string `s`. It's important to note that a substring in this context is defined as a contiguous block of characters within the string `s`.

Intuition

To solve this problem, we can use a Depth First Search (DFS) approach. The intuition behind a DFS is to explore each possible substring starting from the beginning of the string and recursively split the rest of the string in a similar manner. For each recursive call, we track the index at which we're currently splitting the string and maintain a count of the unique substrings created so far. In addition, we keep a record of visited substrings in a set to ensure uniqueness.

Here's the step-by-step thought process:

1. Begin at the start of the string and consider every possible substring that can be formed starting from this position.
2. At each step, choose a substring and check if it is unique (i.e., not already in the set of visited substrings). If it's unique, add it to the set to track its inclusion.
3. Recurse with the next part of the string, starting immediately after the selected substring, while incrementing the count of unique substrings.
4. If you reach the end of the string, compare the count of unique substrings with a global variable that maintains the maximum count seen thus far, updating it if necessary.
5. After the recursion call, backtrack by removing the last chosen substring from the visited set, allowing it to be considered in different combinations.
6. By the end of the DFS exploration, the global variable, `ans`, will contain the maximum number of unique substrings we can obtain by splitting the string.

The use of DFS ensures we explore all possible combinations of substrings, and the use of a set (`vis`) makes sure that all chosen substrings are unique throughout the recursion.

Solution Approach

The solution utilizes a recursive Depth First Search (DFS) algorithm to traverse through all possibilities, and uses a set as a data structure to ensure the uniqueness of substrings.

Here's a detailed walkthrough of the implementation:

- Define a recursive function `dfs` that takes two parameters: `i`, which represents the current starting index of the substring, and `t`, the count of unique substrings formed so far.
- If `i` reaches the length of string `s`, it means we have reached the end. We compare and update the global `ans` variable with `t` if it represents a higher count of unique substrings.
- Otherwise, iterate over the string `s` starting from index `i + 1` to the end of the string. This loop represents all possible ends of the substring that starts at index `i`.
- In each iteration, we first check if the current substring `s[i:j]` hasn't been visited (is not in the set `vis`). If it is unique:
  - Add the current substring to the `vis` set.
  - Recurse with the new index `j`, which is the end of the current substring, and increment the count of unique substrings `t + 1`.
  - After recursion, remove the current substring `s[i:j]` from the set `vis` to allow for other potential unique substrings that could use the same segment of `s`.
- Initialize the `vis` set that holds the unique substrings and declare a variable `ans` which initially holds the value `1`, since the string itself can be considered as a unique substring of length one.
- Finally, start the DFS with the initial call `dfs(0, 0)`, and after recursive exploration of all possible splits, return the maximum count stored in `ans`.

The key patterns used here are recursion for exhaustive search and backtracking for exploring different paths without repeating the visited combinations. The set data structure is essential for assuring the uniqueness constraint for substrings.

Here's a snippet of the key function from the code:

```python
def dfs(i, t):
    if i >= len(s):
        nonlocal ans
        ans = max(ans, t)
        return
    for j in range(i + 1, len(s) + 1):
        if s[i:j] not in vis:
            vis.add(s[i:j])
            dfs(j, t + 1)
            vis.remove(s[i:j])
```

This method ensures that all potential substring splits are considered, and the maximum number of unique substrings that can be obtained from `s` is found.

Example Walkthrough

Let's take a small example to illustrate the solution approach using the provided problem content. Consider the string `s = "abab"`.

Following the steps of the depth-first search (DFS) and backtracking algorithm:

1. We begin with an empty set `vis` for maintaining unique substrings and a variable `ans` initialized to `1`.
2. We start the DFS by calling `dfs(0, 0)`, signifying that we start from the beginning of the string and currently have `0` unique substrings.

First, consider `i = 0`:

- We take the substring `s[0:1]` which is `"a"` and since it's not in `vis`, we add it to `vis` and call `dfs(1, 1)`.

  For the recursive call `dfs(1, 1)`:

  - Now `i = 1`; we take `s[1:2]` which is `"b"`, add it to `vis`, and call `dfs(2, 2)`.

    For the recursive call `dfs(2, 2)`:

    - `i = 2`; we can't choose `"a"` again (as `s[2:3]`) because it is already in `vis`. Hence, we move to `s[2:4]`, which is `"ab"`, add it to `vis`, and call `dfs(4, 3)`.

      For the recursive call `dfs(4, 3)`:

      - `i = 4`; we have reached the end of the string. We compare and update `ans` with `3` which is greater than the initial value `1`.
      - We backtrack and remove `"ab"` from `vis`.
      - The calls `dfs(3, 3)` and previous layers would not produce any further unique substrings, so we backtrack completely to the first call after `"b"` and `s[1:2]` is removed from `vis`.
  - Back in `dfs(1, 1)`, we continue looking for substrings and try `s[1:3]` which is `"ab"`. It's unique at this point, so we add it to `vis` and call `dfs(3, 2)`.

    For the recursive call `dfs(3, 2)`:

    - `i = 3`; we choose `s[3:4]` which is `"b"`, add it to `vis`, and call `dfs(4, 3)`.

      For the recursive call `dfs(4, 3)`:

      - `i = 4`; we are at the end of the string. `ans` is again compared and updated if necessary, but it remains `3`.

In each recursive call, after we reach the end, we backtrack by removing the last inserted substring from `vis`. This allows us to explore different combinations.

At the end of the DFS, `ans` will contain the value `3`, which is the maximum number of non-empty, unique substrings that `"abab"` can be split into - these are `"a"`, `"b"`, and `"ab"`.

Python Solution

```python
class Solution:
    def maxUniqueSplit(self, s: str) -> int:
        def backtrack(start_index, unique_count):
            # Base Case: If the starting index reaches or exceeds the length of the string
            if start_index >= len(s):
                # Update the maximum number of unique splits
                nonlocal max_unique_splits
                max_unique_splits = max(max_unique_splits, unique_count)
                return
            # Try to split the string from the current index to all possible subsequent indexes
            for end_index in range(start_index + 1, len(s) + 1):
                # If this substring has not been seen before
                if s[start_index:end_index] not in seen_substrings:
                    # Add the substring to the set of seen substrings
                    seen_substrings.add(s[start_index:end_index])
                    # Continue splitting the rest of the string with one additional unique substring found
                    backtrack(end_index, unique_count + 1)
                    # Backtrack: remove the substring from the set to try other possibilities
                    seen_substrings.remove(s[start_index:end_index])

        # Initialize a set to keep track of seen substrings
        seen_substrings = set()
        # Initialize the maximum number of unique splits variable
        max_unique_splits = 1
        # Start the recursive backtracking process from the beginning of the string
        backtrack(0, 0)
        # Return the maximum number of unique splits found
        return max_unique_splits

```

Java Solution

```java
class Solution {
    // HashSet to keep track of visited substrings
    private Set<String> visited = new HashSet<>();

    // Variable to store the maximum number of unique splittings
    private int maxSplitCount = 1;

    // The input string on which splitting is performed
    private String inputString;

    // Main method that is called to get the max unique split count
    public int maxUniqueSplit(String s) {
        // Initialize inputString with the input parameter
        this.inputString = s;

        // Start recursive depth-first search from index 0 with count 0
        dfs(0, 0);

        // Return the maximum number of unique splittings found
        return maxSplitCount;
    }

    // Recursive Depth-First Search method to split the string and track maximum unique splitting
    private void dfs(int startIndex, int splitCount) {
        // Base case: if we have reached the end of the string
        if (startIndex >= inputString.length()) {
            // Update the maxSplitCount with the maximum value between current max and current split count
            maxSplitCount = Math.max(maxSplitCount, splitCount);
            return; // End the current branch of recursion
        }
        // Recursive case: try to split the string for all possible next positions
        for (int endIndex = startIndex + 1; endIndex <= inputString.length(); ++endIndex) {
            // Get the current substring to be considered
            String currentSubstring = inputString.substring(startIndex, endIndex);

            // Check and add the current substring to the visited set if it's not already present
            if (visited.add(currentSubstring)) {
                // Continue search with the next part of the string, increasing the split count
                dfs(endIndex, splitCount + 1);

                // Backtrack and remove the current substring from the visited set
                visited.remove(currentSubstring);
            }
        }
    }
}

```

C++ Solution

```cpp
#include <unordered_set>
#include <algorithm>
#include <string>
using namespace std;

class Solution {
public:
    unordered_set<string> visited;  // Set to store visited substrings
    string inputString;
    int maxCount = 1;               // Store the maximum count of unique splits

    // The main function to be called to find the maximum number of unique splits
    int maxUniqueSplit(string s) {
        inputString = s;
        dfs(0, 0); // Begin the depth-first search (DFS) from the first character
        return maxCount;
    }

    // The DFS function to explore all possible unique splits
    void dfs(int startIndex, int uniqueCount) {
        // Base case: if the index has reached or exceeded the size of the string
        if (startIndex >= inputString.size()) {
            // Update the maximum count of unique splits found so far
            maxCount = max(maxCount, uniqueCount);
            return;
        }

        // Iterate through the string, attempting to split at each index
        for (int endIndex = startIndex + 1; endIndex <= inputString.size(); ++endIndex) {
            // Generate the current substring using substr
            string currentSubstring = inputString.substr(startIndex, endIndex - startIndex);

            // Check if the substring has not been visited (i.e., is unique)
            if (!visited.count(currentSubstring)) {
                // Mark the substring as visited
                visited.insert(currentSubstring);

                // Recursively call dfs with the updated parameters to proceed with the next part of the string
                dfs(endIndex, uniqueCount + 1);

                // Backtrack: erase the current substring from visited set as we return from the recursion
                visited.erase(currentSubstring);
            }
        }
    }
};

```

Typescript Solution

```typescript
// Importing a set equivalent in TypeScript
import { Set } from "typescript-collections";

// Global variable to store visited substrings
const visited = new Set<string>();
// Global variable for the input string
let inputString: string;
// Global variable to store the maximum count of unique splits, initially 1
let maxCount: number = 1;

// Function to find the maximum number of unique splits in a string
function maxUniqueSplit(s: string): number {
    inputString = s;
    dfs(0, 0); // Begin the depth-first search (DFS) from the first character
    return maxCount;
}

// Recursive DFS function to explore all possible unique splits of the string
function dfs(startIndex: number, uniqueCount: number): void {
    // If the startIndex reaches or exceeds the length of the inputString, we update maxCount if necessary
    if (startIndex >= inputString.length) {
        maxCount = Math.max(maxCount, uniqueCount);
        return;
    }

    // Iterate through the string, attempting to split at each possible position
    for (let endIndex = startIndex + 1; endIndex <= inputString.length; ++endIndex) {
        // Generate a substring based on the current start and end indices
        let currentSubstring: string = inputString.substring(startIndex, endIndex);

        // Check if this substring hasn't been seen before
        if (!visited.contains(currentSubstring)) {
            // Mark this substring as visited
            visited.add(currentSubstring);

            // Recursively call dfs on the next segment of the string
            dfs(endIndex, uniqueCount + 1);

            // Backtrack by removing the current substring from the visited set
            visited.remove(currentSubstring);
        }
    }
}

// Note: The Set collection may not have exactly the same API as the JavaScript Set.
// If using 'typescript-collections' doesn't match the expected behavior, you would have to
// implement the Set functionality to appropriately reflect the expected behavior or find an
// appropriate library that mimics the JavaScript Set API closely.

```

Time and Space Complexity

Time Complexity

The provided code implements a depth-first search (DFS) approach to solve the problem of finding the maximum number of unique substrings that can be split from a given string.

To determine the time complexity, consider each step in the code:

- The `dfs` function is called recursively, considering all possible splits starting at different positions in the string until the end of the string is reached.
- In each function call, it performs a loop from `i + 1` to `len(s) + 1`, exploring all the possibilities for the next starting point of a substring.
- A substring is added to the `vis` set only if it's not already seen. This operation takes `O(k)` where `k` is the length of the substring, as strings are immutable in Python and need to be hashed for insertion in a set.

Since we consider every possible split for a string of length `n`, and in each step we could potentially generate all possible substrings, the worst-case is exponential in nature with respect to the input size.

The worst-case time complexity is therefore `O(n * 2^n)`, because at each of the `n` positions, we could make a decision to split or not to split the string, and we are doing this for all `n` positions except the last one which is determined by the choices made for the previous substrings.

Space Complexity

Looking at the space complexity:

- The set `vis` stores the unique substrings we have encountered. In the worst case, it can store all possible substrings of `s`, which amounts to `O(2^n)` substrings if we consider all possible splits.
- The recursion depth can go up to `n` in the case of splitting each character as a separate substring. Hence, the space complexity due to recursive calls is `O(n)`.

Therefore, combining the storage for the set and the recursive call stack space, the overall space complexity is `O(2^n + n)`, where `O(2^n)` is the dominant term, simplifying the space complexity to `O(2^n)`.