

# 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:

- Begin at the start of the string and consider every possible substring that can be formed starting from this position.
- 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.
- Recurse with the next part of the string, starting immediately after the selected substring, while incrementing the count of unique substrings.
- 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.
- After the recursion call, backtrack by removing the last chosen substring from the visited set, allowing it to be considered in different combinations.
- 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:

```
1 def dfs(i, t):
2     if i >= len(s):
3         nonlocal ans
4         ans = max(ans, t)
5         return
6     for j in range(i + 1, len(s) + 1):
7         if s[i:j] not in vis:
8             vis.add(s[i:j])
9             dfs(j, t + 1)
10            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:

- We begin with an empty set `vis` for maintaining unique substrings and a variable `ans` initialized to 1.
- 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

```
1 class Solution:
2     def maxUniqueSplit(self, s: str) -> int:
3         def backtrack(start_index, unique_count):
4             # Base Case: If the starting index reaches or exceeds the length of the string
5             if start_index >= len(s):
6                 # Update the maximum number of unique splits
7                 nonlocal max_unique_splits
8                 max_unique_splits = max(max_unique_splits, unique_count)
9                 return
10            # Try to split the string from the current index to all possible subsequent indexes
11            for end_index in range(start_index + 1, len(s) + 1):
12                # If this substring has not been seen before
13                if s[start_index:end_index] not in seen_substrings:
14                    # Add the substring to the set of seen substrings
15                    seen_substrings.add(s[start_index:end_index])
16                    # Continue splitting the rest of the string with one additional unique substring found
17                    backtrack(end_index, unique_count + 1)
18                    # Backtrack: remove the substring from the set to try other possibilities
19                    seen_substrings.remove(s[start_index:end_index])
20
21            # Initialize a set to keep track of seen substrings
22            seen_substrings = set()
23            # Initialize the maximum number of unique splits variable
24            max_unique_splits = 1
25            # Start the recursive backtracking process from the beginning of the string
26            backtrack(0, 0)
27            # Return the maximum number of unique splits found
28            return max_unique_splits
29
```

## Java Solution

```
1 class Solution {
2     // HashSet to keep track of visited substrings
3     private Set<String> visited = new HashSet<>();
4
5     // Variable to store the maximum number of unique splittings
6     private int maxSplitCount = 1;
7
8     // The input string on which splitting is performed
9     private String inputString;
10
11    // Main method that is called to get the max unique split count
12    public int maxUniqueSplit(String s) {
13        // Initialize inputString with the input parameter
14        this.inputString = s;
15
16        // Start recursive depth-first search from index 0 with count 0
17        dfs(0, 0);
18
19        // Return the maximum number of unique splittings found
20        return maxSplitCount;
21    }
22
23    // Recursive Depth-First Search method to split the string and track maximum unique splitting
24    private void dfs(int startIndex, int splitCount) {
25        // Base case: if we have reached the end of the string
26        if (startIndex >= inputString.length()) {
27            // Update the maxSplitCount with the maximum value between current max and current split count
28            maxSplitCount = Math.max(maxSplitCount, splitCount);
29            return; // End the current branch of recursion
30        }
31
32        // Recursive case: try to split the string for all possible next positions
33        for (int endIndex = startIndex + 1; endIndex <= inputString.length(); ++endIndex) {
34            // Get the current substring to be considered
35            String currentSubstring = inputString.substring(startIndex, endIndex);
36
37            // Check and add the current substring to the visited set if it's not already present
38            if (!visited.contains(currentSubstring)) {
39                // Continue search with the next part of the string, increasing the split count
40                dfs(endIndex, splitCount + 1);
41
42                // Backtrack and remove the current substring from the visited set
43                visited.remove(currentSubstring);
44            }
45        }
46    }
47}
```

## C++ Solution

```
1 #include <unordered_set>
2 #include <algorithm>
3 #include <string>
4 using namespace std;
5
6 class Solution {
7 public:
8     unordered_set<string> visited; // Set to store visited substrings
9     string inputString;
10    int maxCount = 1; // Store the maximum count of unique splits
11
12    // The main function to be called to find the maximum number of unique splits
13    int maxUniqueSplit(string s) {
14        inputString = s;
15        dfs(0, 0); // Begin the depth-first search (DFS) from the first character
16        return maxCount;
17    }
18
19    // The DFS function to explore all possible unique splits
20    void dfs(int startIndex, int uniqueCount) {
21        // Base case: if the index has reached or exceeded the size of the string
22        if (startIndex >= inputString.size()) {
23            // Update the maximum count of unique splits found so far
24            maxCount = max(maxCount, uniqueCount);
25            return;
26        }
27
28        // Iterate through the string, attempting to split at each index
29        for (int endIndex = startIndex + 1; endIndex <= inputString.size(); ++endIndex) {
30            // Generate the current substring using substr
31            string currentSubstring = inputString.substr(startIndex, endIndex - startIndex);
32
33            // Check if the substring has not been visited (i.e., is unique)
34            if (!visited.contains(currentSubstring)) {
35                // Mark the substring as visited
36                visited.insert(currentSubstring);
37
38                // Recursively call dfs with the updated parameters to proceed with the next part of the string
39                dfs(endIndex, uniqueCount + 1);
40
41                // Backtrack: erase the current substring from visited set as we return from the recursion
42                visited.erase(currentSubstring);
43            }
44        }
45    }
46 };
47
```

## Typescript Solution

```
1 // Importing a set equivalent in TypeScript
2 import { Set } from "typescript-collections";
3
4 // Global variable to store visited substrings
5 const visited = new Set<string>();
6 // Global variable for the input string
7 let inputString: string;
8 // Global variable to store the maximum count of unique splits, initially 1
9 let maxCount: number = 1;
10
11 // Function to find the maximum number of unique splits in a string
12 function maxUniqueSplit(s: string): number {
13     inputString = s;
14     dfs(0, 0); // Begin the depth-first search (DFS) from the first character
15     return maxCount;
16 }
17
18 // Recursive DFS function to explore all possible unique splits of the string
19 function dfs(startIndex: number, uniqueCount: number): void {
20     // If the startIndex reaches or exceeds the length of the inputString, we update maxCount if necessary
21     if (startIndex >= inputString.length) {
22         maxCount = Math.max(maxCount, uniqueCount);
23         return;
24     }
25
26     // Iterate through the string, attempting to split at each possible position
27     for (let endIndex = startIndex + 1; endIndex <= inputString.length; ++endIndex) {
28         // Generate a substring based on the current start and end indices
29         let currentSubstring: string = inputString.substring(startIndex, endIndex);
30
31         // Check if this substring hasn't been seen before
32         if (!visited.contains(currentSubstring)) {
33             // Mark this substring as visited
34             visited.add(currentSubstring);
35
36             // Recursively call dfs on the next segment of the string
37             dfs(endIndex, uniqueCount + 1);
38
39             // Backtrack by removing the current substring from the visited set
40             visited.remove(currentSubstring);
41         }
42     }
43 }
44
45 // Note: The Set collection may not have exactly the same API as the JavaScript Set.
46 // If using 'typescript-collections' doesn't match the expected behavior, you would have to
47 // implement the Set functionality to appropriately reflect the expected behavior or find an
48 // appropriate library that mimics the JavaScript Set API closely.
49
```

## 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  $O(2^n * 2^n)$ , 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  $O(2^n)$ .