87. Scramble String

Hard String Dynamic Programming Leetcode Link

Problem Description

The problem gives us a possible operation on a string where we can take any non-empty string (other than a single character) and split it into two non-empty substrings. We then have the option to either swap these two substrings or leave them as they are. This process can be applied recursively to the new substrings generated from the split. With these rules in mind, we are asked to determine if a given string `s2` can be made from another string `s1` through one or more of the mentioned scrambling operations, assuming `s1` and `s2` are of the same length.

Intuition

The core of the solution lies in understanding that the problem can be broken down into smaller subproblems. This is a classic example of a divide and conquer algorithm. Given two strings `s1` and `s2`, our goal is to see if there is a way to split both strings into substrings such that, after potentially swapping the substrings, the smaller pieces are equal or can themselves be further scrambled to be equal.

Taking this intuition, we can design a recursive function that tries to split the strings into two parts in various ways and then checks if after swapping or not swapping, the resulting substrings could be made equal through further recursion. To optimize the recursive solution and eliminate redundant calculations, we use memorization, which stores the results of subproblems that have already been solved.

The function `dfs(i, j, k)` represents a recursive depth-first search with memorization that checks if the substring of `s1` starting at index `i` and of length `k` can be transformed into the substring of `s2` starting at index `j` and of the same length. This is done by checking, for each possible split, both cases: keeping the order of the substrings untouched, and swapping them. If any of these cases yields `true`, we can conclude that `s2` is a scrambled version of `s1`.

To implement this efficiently, we use the `cache` decorator (from Python's `functools` module) to memorize the results of subproblems, thus ensuring that our algorithm runs in polynomial time. The recursive function will stop splitting once it reaches substrings of length 1, as those cannot be split further, and a simple character comparison will suffice to continue the recursion. Without memorization, the solution would be prohibitively slow due to the exponential growth of possible splits as the string length increases.

Solution Approach

The solution to this problem utilizes a technique known as memorized recursion, which is a subset of dynamic programming. The idea is that we build our solution from the bottom up by solving all possible subproblems and storing their results to avoid redundant computation for overlapping subproblems.

Here's a breakdown of the implementation:

- **Memorized Recursion (`dfs` function)**: A depth-first search is performed recursively by the function `dfs(i, j, k)` which attempts to determine whether a substring of `s1` starting at `i` and having a length `k` can be transformed into a substring of `s2` starting at `j` of the same length (`k`). We use Python's `@cache` decorator to store the results of subproblems in memory, essentially creating a memoization table that the `dfs` function accesses before computing a new result. This significantly speeds up the algorithm by avoiding recomputation.

- **Subproblem Conditions**: For each call to `dfs(i, j, k)`, two cases are possible depending on whether the substrings are swapped or not:

  1. When there is no swap, we check if `dfs(i, j, h)` and `dfs(i + h, j + h, k - h)` return true for some split length `h`, meaning that the first `h` characters of the two substrings and the remaining `k - h` characters of the two substrings can make a scramble respectively.
  2. When there is a swap, we check if `dfs(i, j + k - h, h)` and `dfs(i + h, j, k - h)` return true for some split length `h`, meaning that the last `h` characters of the first substring and the first `h` characters of the second substring, and the remaining characters of both substrings can make a scramble, respectively.
- **Base Case**: `dfs` includes a base case for when the length of the substring (`k`) is 1, at which point it simply compares the individual characters at the given indices `i` and `j` in `s1` and `s2`, respectively.

- **Return Value**: The overall result comes from the initial call to `dfs(0, 0, len(s1))`, which checks if the entire `s1` can be scrambled to result in `s2`.

- **Complexity**: The memoization search has a time complexity of `O(n^4)` and a space complexity of `O(n^3)`, where `n` is the length of the strings. Time complexity is such because, in the worst case, we may need to check each possible split for each possible substring length `k` at every possible starting index `i` and `j`. Space complexity comes from the amount of information we need to store in our cache.

This recursive and memoization approach ensures we only compute what's necessary, making the problem solvable within a reasonable amount of time for larger strings.

Example Walkthrough

Let's consider a small example to illustrate the solution approach. Suppose we have two strings `s1 = "great"` and `s2 = "rgeat"`. We want to find out if `s2` can be obtained from `s1` by applying the scrambling operations described in the problem.

Step 1: Check if `s1` can be scrambled to become `s2` by calling `dfs(0, 0, len(s1))`. This is the top-level problem we need to solve.

Step 2: In our `dfs(i, j, k)` function, we start with `i = 0`, `j = 0`, and `k = len("great") = 5`. We need to check every possible split.

Step 3: Let's split `s1` and `s2` into `h = 2` characters from the start and the remaining `k - h = 3` characters. For `s1` (`"great"`), the two parts are `"gr"` and `"eat"`. For `s2` (`"rgeat"`), the corresponding parts are `"rg"` and `"eat"`.

In the no-swap case:

- For the first `2` characters, a recursive call `dfs(0, 0, 2)` checks if `"gr"` can be scrambled into `"rg"`, which is true. So `"gr"` can be transformed into `"rg"` by swapping the characters.
- For the remaining `3` characters, a recursive call `dfs(2, 2, 3)` checks if `"eat"` can be scrambled into `"eat"`, which is trivially true.

The recursive calls would return `true` since both parts can be made to match. Therefore, `s2` is a valid scramble of `s1` without even needing to check the swap case for this split.

Step 4: The algorithm would then proceed to check all other possible ways of splitting `s1` and `s2`. However, since we have already found a valid scramble, the algorithm doesn't need to continue further for this example.

Step 5: After all the recursion and verification, if any of the recursive calls return `true`, we conclude that `s2` is a scrambled version of `s1`. Since we found that `dfs(0, 0, len(s1))` returns `true`, `s2 = "rgeat"` can indeed be made from `s1 = "great"`.

Step 6: During the process, the results of various calls to `dfs` are stored in the memoization table. Should the same subproblems arise, the algorithm retrieves the result from the table instead of recalculating, effectively reducing computation time and complexity.

In summary, by breaking the problem into smaller subproblems, recursing on those, and memoizing the results, the algorithm can efficiently determine if one string is a scrambled version of the other.

Python Solution

```python
from functools import lru_cache

class Solution:
    def isScramble(self, s1: str, s2: str) -> bool:
        # Decorator to cache the results of recursive calls
        @lru_cache(maxsize=None)
        def search_recursive(i: int, j: int, length: int) -> bool:
            # Base case: if substring length is 1, check equality of chars
            if length == 1:
                return s1[i] == s2[j]

            # Check for each possible split
            for split_point in range(1, length):
                # If a matching scramble is found with a split, return True
                if (search_recursive(i, j, split_point) and
                    search_recursive(i + split_point, j + split_point, length - split_point)):
                    return True

                # Check for a scramble with a swapped second half
                if (search_recursive(i + split_point, j, length - split_point) and
                    search_recursive(i, j + length - split_point, split_point)):
                    return True

            # If no scramble can be formed, return False
            return False

        # Initial call for the recursive search function
        return search_recursive(0, 0, len(s1))

# Example usage
# sol = Solution()
# result = sol.isScramble("great", "rgeat") # Should return True
# print(result)

```

Java Solution

```java
class Solution {
    private Boolean[][][] memoizationCache; // A 3D memoization cache to store the results of subproblems
    private String str1; // First string to compare
    private String str2; // Second string to compare

    // Determines if s2 is a scrambled string of s1
    public boolean isScramble(String s1, String s2) {
        int length = s1.length();
        this.str1 = s1;
        this.str2 = s2;
        memoizationCache = new Boolean[length][length][length + 1]; // Initialize memoization cache
        return dfs(0, 0, length); // Launch the depth-first search starting with the full length
    }

    // Executes the depth-first search to see if two substrings are scrambled equivalents
    private boolean dfs(int index1, int index2, int length) {
        if (memoizationCache[index1][index2][length] != null) {
            // If result is already computed for this subproblem, return the result
            return memoizationCache[index1][index2][length];
        }
        if (length == 1) {
            // If the length to compare is 1, check if characters are equal
            return str1.charAt(index1) == str2.charAt(index2);
        }
        for (int partition = 1; partition < length; ++partition) {
            // Check if swapping the partition leads to a scramble that matches
            if (dfs(index1, index2, partition) && dfs(index1 + partition, index2 + partition, length - partition)) {
                // The first segment of str1 matches the first segment of str2 and the second segment of str1 matches the second segn
                return memoizationCache[index1][index2][length] = true;
            }
            // Check if non-swapped variant leads to a scramble that matches
            if (dfs(index1 + partition, index2, length - partition) && dfs(index1, index2 + length - partition, partition)) {
                // The first segment of str1 matches the second segment of str2 and the second segment of str1 matches the first segn
                return memoizationCache[index1][index2][length] = true;
            }
        }
        // If none of the above conditions lead to a scramble that matches, then return false
        return memoizationCache[index1][index2][length] = false;
    }
}

```

C++ Solution

```cpp
#include <cstring>
#include <functional>
#include <string>

using namespace std;

class Solution {
public:
    // Function to determine if two strings are scramble strings
    bool isScramble(string s1, string s2) {
        int length = s1.size();
        // Define a 3D array to store the states of substring scrambles
        int scrambleStates[length][length][length + 1];
        memset(scrambleStates, -1, sizeof(scrambleStates));

        // Recursive lambda function to check for scramble strings
        function<bool(int, int, int)> checkScramble = [&](int index1, int index2, int len) -> bool {
            if (scrambleStates[index1][index2][len] != -1) {
                // Use memoization to return the previously computed result
                return scrambleStates[index1][index2][len] == 1;
            }
            if (len == 1) {
                // If the length is 1, just compare the characters
                return s1[index1] == s2[index2];
            }
            for (int split = 1; split < len; ++split) {
                // Check if the first split part is a scramble and the second split part is a scramble
                if (checkScramble(index1, index2, split) && checkScramble(index1 + split, index2 + split, len - split)) {
                    scrambleStates[index1][index2][len] = 1;
                    return true;
                }
                // Check if swapping the split parts results in a scramble
                if (checkScramble(index1 + split, index2, len - split) && checkScramble(index1, index2 + len - split, split)) {
                    scrambleStates[index1][index2][len] = 1;
                    return true;
                }
            }
            scrambleStates[index1][index2][len] = 0;
            return false;
        };

        // Start the recursive check with the full length of the strings
        return checkScramble(0, 0, length);
    }
};

```

Typescript Solution

```typescript
// A function to determine if two strings are scrambles of each other
function isScramble(str1: string, str2: string): boolean {
    const length = str1.length;

    // The memoization table where -1 indicates uninitialized, 0 for false, and 1 for true
    const memo = new Array(length)
        .fill(0)
        .map(() => new Array(length).fill(0).map(() => new Array(length + 1).fill(-1)));

    // A recursive function to check if substrings are scrambles
    // i is the start index of the substring in str1
    // j is the start index of the substring in str2
    // k is the length of the substrings
    const checkScramble = (i: number, j: number, k: number): boolean => {

        // If this subproblem has already been computed, return the result
        if (memo[i][j][k] !== -1) {
            return memo[i][j][k] === 1;
        }

        // If the length of the substrings is 1, we simply compare the characters
        if (k === 1) {
            return str1[i] === str2[j];
        }

        // Check for all possible splits
        for (let h = 1; h < k; ++h) {
            // If swapping the current parts make them equals, set result to true
            if (checkScramble(i, j, h) && checkScramble(i + h, j + h, k - h)) {
                return Boolean((memo[i][j][k] = 1));
            }
            // If swapping the non-corresponding parts make them equals, set result to true
            if (checkScramble(i + h, j, k - h) && checkScramble(i, j + k - h, h)) {
                return Boolean((memo[i][j][k] = 1));
            }
        }

        // If none of the splits result in matching strings, set result to false
        return Boolean((memo[i][j][k] = 0));
    };

    // Calling the recursive check with the start indices at 0 and full length of the strings
    return checkScramble(0, 0, length);
}

```

Time and Space Complexity

The given Python solution leverages recursion with memoization to determine if one string is a scramble of another. Here's an analysis of its time and space complexities:

Time Complexity:

The `dfs` function is memoized and explores each possible split of the string segments once for every distinct pair of starting indices `(i, j)` and length `k`. Since `i` and `j` can each range from `0` to `n-1` (for an n-character string) and `k` can range from `1` to `n`, there are `O(n^2)` pairs of starting indices and `O(n)` possible lengths for each pair. Thus, the time complexity can seem to be `O(n^3)` considering each possible `(i, j, k)` triplet is calculated once due to memoization.

However, within every call to `dfs(i, j, k)`, there is a loop that runs `k-1` times. Since `k` can be up to `n`, the loop can contribute at most `O(n)` operations per memoized function call. This loop is where the fourth dimension of the time complexity stems from, leading to a total time complexity of `O(n^4)`.

Space Complexity:

The space complexity is primarily dictated by the memoization cache. The space needed for the cache relates directly to the number of distinct arguments passed to the `dfs` function, which, as explained, is the product of the different values `i`, `j`, and `k` can take. Since `i` and `j` range from `0` to `n-1` and `k` ranges from `1` to `n`, the max number of distinct states stored in the cache is `n * n * n`, which is `O(n^3)`.

Besides the cache, the space complexity also includes the space for the call stack due to recursion. In the worst case, the recursive calls could stack up to `O(n)` deep (the maximum possible depth of recursion is when `k` decreases by `1` each time). However, this does not affect the overall space complexity, which remains `O(n^3)` due to the dominating size of the memoization cache.