Salutations fellow coders!

In this article, we explore the solution to LeetCode problem No. 97 "Interleaving String". The problem requires of us to determine whether a given string can be created by interleaving the characters of two other given strings. We are to compose a function that accepts three parameters: String S1, String S2, and String S3. Our function returns a Boolean value of true if string S3 can be created by interleaving the characters from strings S1 & S2; otherwise return false. We present the following two examples from Leetcode to illustrate the problem and its solution.

Example 1:

S1 = "aabcc", S2 = "dbbca", S3 = "aadbbcbcac" output = **true**

Explanation: Our function outputs **true** here because the characters in S3 appear in the *exact* order as they do in strings S1 and S2. It is therefore true that string S3 is created by interleaving strings S1 and S2.

Example 2:

S1 = "aabcc", S2 = "dbbca", S3 = "aadbbbaccc" output = **false**

Explanation: Here our function outputs **false** because of the 7th character ('a') that appears to have come from string S2 is out of place relative to the other characters in S2.

So how do we solve this problem? Upon initial inspection, one might be tempted to implement the naive solution of generating the set of all strings interleaved from S1 and S2 and checking if S3 is among this set. As the string sizes grow, so does the set, consequently requiring more time to generate it. This method is therefore prohibitively slow and we must do better. So what do we do now? In generating the set of all possible strings, the keen observer will notice that we are performing some computions more than once! This suggests the use of Dynamic Programming as a candidate method for the design of the algorithm that will solve this problem, so we will proceed to use it. Before we begin, we must first observe the notion that **in order for a string to be interleaved from two other strings, the last character MUST be one of the last characters from any of the other two strings!** With this fact in mind, we construct an algorithm similar to the algorithm used to determine the Longest Common Subsequence between two strings (it is strongly suggested the reader familiarize themselves with that problem before continuing). We will solve this problem using recursion with memoization, the steps of which are outlined below.

Step 1: We will maintain three pointers: *i, j, k* which point to the characters in strings S1, S2, and S3 respectively. They will be initialized to point to the last character in each string.

Step 2: We will maintain a memo table of Boolean values where memo[i+1][j+1] is **true** if S3.charAt(k) is one of the letters from S1.charAt(i) or S2.charAt(j); it is **false** otherwise. We add one to the values of i and j to account for the fact that memo[0][0] accounts for an empty string and NOT the first two characters of S1 and S2 respectively.

Step 3: We will define a recursive function (call it *dfs* since this is "depth-first search" on a tree) that will take in as input our three pointers and the memo table which will compare the characters, populate the memo table, and output a Boolean value as to whether the strings are valid up to points *i, j, k*.

When comparing the characters, there are four possible cases that will arise:

- Case 1: S3.charAt(k) matches S1.charAt(i), we decrement both i and k by one and call our recursive function on the new pointers. Decrementing i and k implies that we found a character match and now we must check the remainder of the strings to see if it is a valid interleave. The value placed in the memo table will be the result of the new function call we make.

- Case 2: S3.charAt(k) matches S2.charAt(j) same procedure as above except we decrement j by one instead if i and proceed.
- Case 3: S3.charAt(k) matches both S1.charAt(i) *AND* S2.charAt(j). In this situation, we first decrement i by one and call the recursive function on i-1 and k-1 while leaving j untouched. We then do the same for j-1 and k-1. The value of memo[i][j] in this case is the logical **OR** of the two function calls. Doing this means that up to now, we have no way of determining from which string character S3.charAt(k) came from, so we have no choice but to try the two options of first claiming it came from S1 or second claiming it came from S2.
- Case 4: S3.charAt(k) matches neither character from S1.charAt(i) and S2.charAt(j). This means that String S3 cannot be created by interleaving strings S1 and S2, therefore we enter a value of **false** at memo[i+1][j+1] (remember the empty string) and return **false**. We no longer continue to test whether S3 is interleaved from S1 and S2 so we do not call the *dfs* function again on the remainder of the string.

In preprocessing the input variables we consider the following edge cases and their solutions:

- Edge Case 1: String S1 or S2 is empty. We therefore check if S3 is an exact copy of the non-empty string.
- Edge Case 2: The length of S3 is not equal to the sum of the lengths of S1 and S2. We simply return **false** as it is not logically possible to interleave two strings and have the interleaved string not have the same number of characters as the initial two strings.

The code to solve this problem is written in Java and presented on the following page...

```
class Solution {
    String S1, S2, S3; //Global variables so as to not bloat the signature of the dfs function
    public boolean isInterleave(String s1, String s2, String s3) {
        S1 = s1;
        S2 = s2;
        S3 = s3;
        int l1 = s1.length();
        int l2 = s2.length();
        int l3 = s3.length();
        if(l1 == 0)return s2.equals(s3); //Edge case S1 is empty. Check if S3 is a copy of S2
        if(l2 == 0)return s1.equals(s3); //Edge case same as above with S1 and S2 switched
        Boolean[][] memo = new Boolean[l1+1][l2+1];
        memo[0][0] = true; //An empty string ALWAYS matches an empty string
        if(l1 + l2 != l3)return false; //Sanity Check: S3.length() MUST be equal to S2.length()
            plus S1.length()
        return dfs(l1 - 1, l2 - 1, l3 -1, memo);
    }
    //The recursion function
    public boolean dfs(int i, int j, int k, Boolean[][] memo){
        if(memo[i+1][j+1] != null)return memo[i+1][j+1]; //Base case: if this value has been
            computed before, DO NOT recompute, just return it
        //If we have consumed all of the characters from S1, we no longer attempt to use it to test
        //We only use S2 from now on
        if(i < 0){
            memo[i+1][j+1] = S2.charAt(j) == S3.charAt(k) && dfs(i, j-1, k-1, memo);
            return memo[i+1][j+1];
        }
        //Same as last if-block except S2 is out of characters now
        //So we just use S1
        if(j < 0){
            memo[i+1][j+1] = S1.charAt(i) == S3.charAt(k) && dfs(i-1, j, k-1, memo);
            return memo[i+1][j+1];
        }
        //Case 4
        if(S1.charAt(i) != S3.charAt(k) && S2.charAt(j) != S3.charAt(k)){
            memo[i+1][j+1] = false;
            return memo[i+1][j+1];
        }
        if(i >= 0 && S1.charAt(i) == S3.charAt(k) ){
            if(j >= 0 && S2.charAt(j) == S3.charAt(k)){
                memo[i+1][j+1] = dfs(i-1, j, k-1, memo) || dfs(i, j-1, k-1, memo); //Case 3
            }else{
                memo[i+1][j+1] = dfs(i-1, j, k-1, memo); //Case 1
            }
        }else if(j >= 0 && S2.charAt(j) == S3.charAt(k)){
            memo[i+1][j+1] = dfs(i, j-1, k-1, memo); //Case 2
        }else{
            memo[i+1][j+1] = false;
        }
        return memo[i+1][j+1];

    }
}
```