97. Interleaving String String] Medium **Dynamic Programming**

Problem Description

The interleaving of s1 and s2 should create a sequence where characters from s1 and s2 are alternated, but maintain their relative order from the original strings. There can be more than one character from either string coming in sequence, but overall, the characters from s1 and s2 should come in the exact order they appear in their respective strings. For example, if s1 = "abc" and s2 = "def", then s3 = "adbcef" would be a valid interleaving. However, s3 = "abcdef" or s3 =

Given three strings s1, s2, and s3, we need to determine if s3 can be constructed by interleaving characters from s1 and s2.

"badcfe" would not be valid interleavings, because the relative ordering of the characters from s1 and s2 isn't respected in s3.

match the current character in \$3, we cannot proceed further by that route.

stores the results of those subproblems to avoid redundant computations.

s1[i-1]. This is indicated by the operation f[j] &= s1[i-1] == s3[k].

Intuition The intuition for solving this problem lies in dynamic programming. We can imagine forming s3 character by character, making

choices at each step of taking the next character from either s1 or s2. If at any point, the characters from s1 or s2 do not

The key insight for the solution is to construct a table (or array) that represents whether it is possible to form the prefix s3[0...k] from prefixes of s1 and s2. We initialize an array f of length n+1 (where n is the length of s2), which will help us track if s3 can be formed up to the j-th character of s2. We start by assuming that an empty s3 can be formed without any

characters from either string (which is always true). We then iterate over all characters in both s1 and s2, updating the array at each step. The condition f[j] &= s1[i - 1] ==

s3[k] checks if we can form the string by taking the next character from s1, while the condition f[j] |= f[j - 1] and s2[j -1] == s3[k] checks if we can form the string by taking the next character from s2. This way, we fill up the table iteratively, making sure at every step that we satisfy the condition of interleaving without violating the original order in the given strings. If, by the end of the iteration, we find that f[n] is True, it means that we can form \$3 by

Solution Approach The solution uses dynamic programming, a method that solves problems by breaking them down into simpler subproblems and

the sum of the lengths of s1 and s2 (m + n). If not, s3 cannot be an interleaving of s1 and s2, and we return False.

We then create an array f of length n+1. f[j] will hold a boolean value indicating whether s3[0...i+j-1] is a valid interleaving of s1[0...i-1] and s2[0...j-1]. Here is a critical observation: f[0] is initialized to True because an empty string is considered an interleaving of two other empty strings by default.

We begin by defining the lengths of s1 and s2 as m and n, respectively. Immediately, we check if the length of s3 is equal to

We iterate through both strings s1 and s2 using two nested loops, one for index i ranging from 0 to m, and another for

interleaving s1 and s2. Otherwise, we cannot.

Here's a step-by-step breakdown of the algorithm:

index k = i + j - 1. For each pair (i, j), if i is not zero, we update f[j] to keep track of whether s3[k] can still match with the character in

index j ranging from 0 to n. Each iteration represents an attempt to match the character in s3 at the current combined

Similarly, if j is not zero, we update f[j] to see if s3[k] can match with the character s2[j - 1], while also taking into

account the value of f[j - 1], which indicates whether the interleaving was possible without considering the current

character of s2. This happens with the operation f[j] = f[j-1] and s2[j-1] == s3[k]. After both loops terminate, we return the value of f[n]. If f[n] is True, it means that s3 can be formed by interleaving s1 and s2. Otherwise, it means that it is not possible.

The above steps summarize the dynamic programming approach to solving the interleaving string problem. It's worth noting that

the space complexity is optimized to O(n) since we are only using a single one-dimensional array to store the results.

Let's take a simple example to illustrate the solution approach. Consider the following strings: • s1 = "ax"

First, we determine the lengths of s1 (length m = 2), and s2 (length n = 2). The length of s3 is 4, which is equal to m + n(2 + 2), so it's possible for s3 to be an interleaving of s1 and s2.

∘ For i = 0 (considering an empty string for s1), we check each character of s2. We update the array f where indexes correspond to

1, we look at s3[1], which is b. Because f[j - 1] (which is f[0]) is True and s2[j - 1] (which is s2[0] = 'b') matches s3[1], we

We create an array f with n+1 elements, which includes the initial condition (f[0] = True). Initially f = [True, False,

We begin iterating over the strings:

s1 and s2.

Python

class Solution:

Solution Implementation

if len s1 + len s2 != len(s3):

dp = [True] + [False] * len s2

for i in range(len s1 + 1):

k = i + i - 1

for i in range(len s2 + 1):

return False

substrings of s3.

update f[j] to True.

Once we have completed our iterations:

False].

Example Walkthrough

Here's how we'd walk through the algorithm:

• s2 = "by"

• s3 = "abxy"

characters of s2 that match the start of s3. After checking, f = [True, True, False] (since s3[0] matches s2[0]). ∘ For i = 1 (looking at s1[0] = 'a'), we loop through s2. When j = 0, f[j] remains True since s3[0] = 'a' matches s1[0]. When j =

Continuing in this fashion, we finally get f = [True, True, True].

interleaving of s1 and s2 using the described dynamic programming approach.

it is impossible for s3 to be an interleaving of s1 and s2.

Initialize a list to keep track of the possibility of interleaving up to each position.

Check if substrings of s1 can interleave with an empty s2 to form corresponding

The function isInterleave returns True if s3 is an interleaving of s1 and s2, otherwise False.

k is the current index in s3 that we want to match.

dp[j] = dp[j - 1] and s2[j - 1] == s3[k]

The last element in the dp array contains the answer.

The +1 accounts for the empty substring. True implies that the interleaving is possible.

If we are not at the first row, check if previous character in s1 matches with

current character in s3. Use "&=" to modify the existing dp[j] value.

• For i = 2, we iterate over s2 again, and now s3 matches s1[1] and the interleaving continues with the last character y. The final f array becomes [True, True, True].

At the end, we examine the value of f[n], which in this case is True, indicating that s3 can indeed be formed by interleaving

This completes the example walkthrough, showing that given the strings s1 = "ax", s2 = "by", and s3 = "abxy", s3 is a valid

def isInterleave(self, s1: str, s2: str, s3: str) -> bool: # Get the lengths of the input strings. len_s1 , $len_s2 = len(s1)$, len(s2)# If the combined length of s1 and s2 is not equal to the length of s3,

if i: dp[i] &= s1[i - 1] == s3[k]# If we are not at the first column, check if the above cell or the left cell # can lead to an interleaving. Use "|=" to incorporate this new possibility.

if j:

return dp[len_s2]

```
Java
class Solution {
    public boolean isInterleave(String s1, String s2, String s3) {
        // Get the lengths of the strings
        int m = s1.length(), n = s2.length();
        // If the combined length of s1 and s2 does not equal the length of s3, return false
        if (m + n != s3.length()) {
            return false;
        // Create a boolean array to keep track of the interleavings
        boolean[] dp = new boolean[n + 1];
        dp[0] = true;
        // Iterate over each character of both s1 and s2
        for (int i = 0; i \le m; ++i) {
            for (int j = 0; j <= n; ++j) {
                // Index k for matching characters in s3
                int k = i + j - 1;
                // If there are remaining characters in s1, check if they match s3's characters
                if (i > 0) {
                    dp[j] \&= s1.charAt(i - 1) == s3.charAt(k);
                // If there are remaining characters in s2, check if they match s3's characters
                if (i > 0) {
                    dp[j] = (dp[j - 1] & (s2.charAt(j - 1) == s3.charAt(k)));
        // Return whether it's possible to interleave s1 and s2 to get s3
        return dp[n];
```

};

C++

public:

class Solution {

bool isInterleave(string s1, string s2, string s3) {

if (length1 + length2 != s3.size()) {

// Initialize the dp array to false

// Iterate over characters in s1 and s2

int indexS3 = i + j - 1;

// Return the last element in the dp array,

for (int j = 0; j <= length2; ++j) {</pre>

for (int i = 0; i <= length1; ++i) {</pre>

memset(dp, false, sizeof(dp));

if (i > 0) {

if (i > 0) {

return dp[length2];

return false;

bool dp[length2 + 1];

dp[0] = true;

int length1 = s1.size(), length2 = s2.size();

// then s3 cannot be formed by interleaving s1 and s2

// is an interleaving of s1 and s2 up to certain points

// Base case: empty strings are considered interleaving

// Calculate the corresponding index in s3

dp[j] = dp[j] && (s1[i - 1] == s3[indexS3]);

// If the sum of lengths of s1 and s2 is not equal to length of s3,

// dp array to hold the computed values; indicates if s3 up to a point

// If we can take a character from s1 and it matches the corresponding character in s3,

// we maintain the value of dp[j] (continue to be true or false based on previous value)

// If we can take a character from s2 and it matches the corresponding character in s3,

// if previous element in dp array was true (indicating a valid interleave up to that point)

// we update the value of dp[i] to be true if it's either true already or

 $dp[j] = dp[j] \mid | (s2[j - 1] == s3[indexS3] && dp[j - 1]);$

Check if substrings of s1 can interleave with an empty s2 to form corresponding

The function isInterleave returns True if s3 is an interleaving of s1 and s2, otherwise False.

If we are not at the first row, check if previous character in s1 matches with

If we are not at the first column, check if the above cell or the left cell

can lead to an interleaving. Use "|=" to incorporate this new possibility.

current character in s3. Use "&=" to modify the existing dp[j] value.

k is the current index in s3 that we want to match.

dp[j] = dp[j - 1] and s2[j - 1] == s3[k]

0 to n), so the combined iterations for the inner loop are actually m * (n + 1).

// which represents whether the whole s3 is an interleaving of s1 and s2

```
TypeScript
function isInterleave(s1: string, s2: string, s3: string): boolean {
    const s1Length = s1.length;
    const s2Length = s2.length;
    // If the lengths of s1 and s2 together don't add up to the length of s3, they can't interleave to form s3.
    if (s1Length + s2Length !== s3.length) {
        return false;
    // Initialize an array to hold the interim results of the dynamic programming solution.
    // dp[i] will hold the truth value of whether s1 up to i characters can interleave with s2 up to j characters to form s3 up to i+
    const dp: boolean[] = new Array(s2Length + 1).fill(false);
    // Initialize the first value to true as an empty string is considered an interleave of two other empty strings.
    dp[0] = true;
    // Iterate over each character in s1 and s2 to build up the solution in dp.
    for (let i = 0; i <= s1Length; ++i) {
        for (let j = 0; j <= s2Length; ++j) {</pre>
            // Calculate the corresponding index in s3.
            const s3Index = i + j - 1;
            // If we are not at the start of s1, determine if the current character of s1 is equal to the current character of s3
            // and whether the result up to the previous character of s1 was true.
            if (i > 0) {
                dp[j] = dp[j] && s1[i - 1] === s3[s3Index];
            // If we are not at the start of s2, determine if the current character of s2 is equal to the current character of s3
            // and whether the result up to the previous character of s2 was true.
            if (j > 0) {
                dp[j] = dp[j] \mid | (dp[j - 1] && s2[j - 1] === s3[s3Index]);
    // The final result would be if s1 up to its full length can interleave with s2 up to its full length to form s3.
    return dp[s2Length];
class Solution:
    def isInterleave(self, s1: str, s2: str, s3: str) -> bool:
        # Get the lengths of the input strings.
        len_s1, len_s2 = len(s1), len(s2)
        # If the combined length of s1 and s2 is not equal to the length of s3,
        # it is impossible for s3 to be an interleaving of s1 and s2.
        if len s1 + len s2 != len(s3):
            return False
        # Initialize a list to keep track of the possibility of interleaving up to each position.
        # The +1 accounts for the empty substring. True implies that the interleaving is possible.
        dp = [True] + [False] * len_s2
```

The given Python code provides a solution to determine if a string s3 is formed by the interleaving of two other strings s1 and s2. Let's analyze both the time complexity and space complexity of the code.

Time and Space Complexity

substrings of s3.

if i:

if j:

return dp[len_s2]

for i in range(len s1 + 1):

k = i + j - 1

for j in range(len s2 + 1):

dp[i] &= s1[i - 1] == s3[k]

The last element in the dp array contains the answer.

Time Complexity The outer loop in the code runs m + 1 times, where m is the length of s1. Within this loop, there's an inner loop that runs m + 1

times, where n is the length of s2. However, observe that for each outer iteration, the inner loop starts at 1 (since j ranges from

Each iteration of the inner loop consists of constant time checks and assignment operations, so its time complexity is 0(1). Thus,

the total time complexity for the double loop structure is 0(m * (n + 1)). Simplifying, this is equivalent to 0(m * n) since the addition of a constant 1 does not change the order of growth.

Therefore, the overall time complexity of the code is 0(m * n). **Space Complexity**

Space complexity considers the additional space used by the algorithm excluding the input sizes. In the code, a one-dimensional

Boolean array f is initialized with n + 1 elements. The space usage of this array dominates the space complexity. There is no

other data structure that grows with the input size. Thus, the space complexity is based on the size of f, which is 0(n).

To summarize:

- Time complexity: 0(m * n) Space complexity: 0(n)