727. Minimum Window Subsequence

Hard | String | Dynamic Programming | Sliding Window | Leetcode Link

Problem Description

In this problem, you are given two strings `s1` and `s2`. Your task is to find the shortest contiguous substring in `s1` such that the entire string `s2` is a subsequence of that substring. A subsequence is a sequence that can be derived from the other sequence by deleting some or no elements without changing the order of the remaining elements.

For example, if `s1` is "abcdebdde" and `s2` is "bde", you have to find the shortest part of `s1` that contains all the characters in `s2` in the same order.

Here are the conditions you need to satisfy:

- If no such window exists, return an empty string "".
- If there are multiple substrings of `s1` that satisfy the condition, return the one with the left-most starting index.

In the example given, the minimum window in `s1` where `s2` is a subsequence is "bcde".

Intuition

The key to solving this problem is understanding dynamic programming and the process of subsequence matching. We must scan through `s1` and `s2` to find all possible match positions in a way that allows us to efficiently calculate the minimum window length where `s2` is a subsequence.

The intuition behind the solution is to first find all the matchings between characters of `s2` with `s1`, and keep track of the starting index of the sequence in `s1` that matches up to a certain point in `s2`.

To solve this, we create a 2D array `f` where `f[i][j]` represents the starting index in `s1` from which we have a matching sequence for `s2` up to its `j-th` character at `i-th` index of `s1`.

Once we have this information stored in the `f` array, we iterate through `s1` to find the character that matches the last character of `s2`. Each time we find such a match, we look into our `f` array for the starting index and calculate the window size.

The minimum window size is kept updated during the scan, and finally, we have the starting index and the size of our minimum window which we use to return our result.

The entire algorithm takes into account:

- Dynamic Programming to solve the subsequence matching.
- Iterating `s1` and `s2` smartly to minimize unnecessary comparisons.
- Keeping track of the minimum window during the iteration.
- Handling edge cases effectively, like when no window exists.

Solution Approach

The solution provided uses a two-dimensional dynamic programming approach:

1. **Initialization**:
   - The 2D array `f` is created with a size of `(m+1) x (n+1)` where `m` is the length of `s1` and `n` is the length of `s2`. This array will help track matches between `s1` and `s2`.
2. **Filling the DP array**:
   - We iterate over both strings starting from index 1 (because we've initialized from 0 as part of the dynamic programming table setup), comparing each character of `s1` with each character of `s2`.
   - When a match (`a == b`) is found:
     - If it is the first character of `s2` (`j == 1`), we record this position `i` as a starting point of a matching sequence because it could potentially start a new subsequence.
     - For other characters, we copy the starting index from `f[i - 1][j - 1]`, which means we extend the subsequence found until the previous character of `s2`.
   - When there is no match, we get the starting index from the previous value `f[i - 1][j]` because we want to retain the starting position of the best match found so far for `s2` up to `j`.
3. **Identifying the minimum window**:
   - Now, we look for the last character of `s2` in `s1` to try and close a potential window.
   - For each matching position `i` in `s1` where `s1[i] == s2[n - 1]` and a subsequence match has been found (`f[i][n]` is not zero), we calculate the window size by subtracting the starting index `j` (which is `f[i][n] - 1`) from `i`. We keep track of the smallest window found in variables `k` for size and `p` for the starting index.
4. **Returning the result**:
   - If we have not found any window (`k > m`), we return an empty string.
   - Otherwise, we return the substring of `s1` starting from `p` with the length `k`.

The choice of dynamic programming in this solution is crucial as it eliminates redundant comparisons and stores intermediate results, allowing efficient computation of the final minimum length substring. This algorithm has an `O(m * n)` time complexity due to the nested loops required to fill the DP table, where `m` and `n` are the lengths of `s1` and `s2`, respectively.

The solution is concise and the use of dynamic programming provides optimal substructure and overlapping subproblems, two key characteristics exploited by this paradigm to achieve efficiency. Each entry in the DP table only depends on previously computed values, making the implementation straightforward and logical once the table relationships are understood.

Example Walkthrough

Let's illustrate the solution approach using a small example.

Suppose `s1` is "axbxcxdx" and `s2` is "bcd".

Following the solution approach:

1. **Initialization**:
   - We set up a 2D array `f` with a size of `(8+1) x (3+1)` since `s1` is of length 8 and `s2` is of length 3.
2. **Filling the DP array**:
   - We iterate over `s1` and `s2`, filling up the `f` array.
   - Let's iterate over `s1`: "axbxcxdx" and `s2`: "bcd".
     - When `i = 1` and `j = 1`, we find that `s1[i] != s2[j]`, so we don't update `f[1][1]`.
     - When `i = 2` and `j = 1`, we find `s1[i] == s2[j]` ('b' == 'b'), so we update `f[2][1]` to 2. This marks the start of a possible subsequence.
     - Continuing this process, we find the 'c' of `s2` in `s1` at position 4, so `f[4][2]` is updated to 2, indicating that from position 2 in `s1`, we have 'bc' of `s2`.
     - We find the 'd' of `s2` in `s1` at position 6, so `f[6][3]` is updated to 2, meaning from position 2 in `s1`, we have the full 'bcd' of `s2`.
3. **Identifying the minimum window**:
   - We scan `f` looking for the smallest window where `s1[i] == s2[3]` ('d' in this case) and `f[i][3]` is not zero.
   - We find this at `i = 6` for `s2[3]` ('d'), where `f[i][3]` is 2. The window size is `6 - 2 + 1 = 5`.
4. **Returning the result**:
   - The smallest window has a size of 5 starting at index 2 in `s1`. So, the result is the substring "bxcxd".

Following this process, we've identified the subsequence 'bcd' within `s1` and found the minimum window. The algorithm efficiently computes this by tracking possible starting points for subsequences in `s1` and keeping track of these starting points as we match characters of `s2`. This allows us to quickly compute the length of potential windows without redundant comparisons.

Python Solution

```python
class Solution:
    def minWindow(self, s1: str, s2: str) -> str:
        # Length of input strings
        len_s1, len_s2 = len(s1), len(s2)

        # Initialize a DP table with dimensions (len_s1+1) x (len_s2+1)
        dp = [[0] * (len_s2 + 1) for _ in range(len_s1 + 1)]

        # Fill the DP table
        for i, char_s1 in enumerate(s1, 1):
            for j, char_s2 in enumerate(s2, 1):
                # If characters match, propagate the match information
                if char_s1 == char_s2:
                    dp[i][j] = i if j == 1 else dp[i - 1][j - 1]
                else:  # If not, inherit the value from previous s1 character
                    dp[i][j] = dp[i - 1][j]

        # Variables to keep track of the start position and length of the minimum window
        min_window_start_pos = 0
        min_window_length = len_s1 + 1

        # Find the minimum window in s1 which contains s2
        for i, char_s1 in enumerate(s1, 1):
            # When the last character of s2 is matched in s1 and a match sequence exists
            if char_s1 == s2[len_s2 - 1] and dp[i][len_s2]:
                match_start = dp[i][len_s2] - 1
                window_length = i - match_start
                # Update the minimum window if a smaller one is found
                if window_length < min_window_length:
                    min_window_length = window_length
                    min_window_start_pos = match_start

        # Check if a valid window was ever found, if not return an empty string
        return "" if min_window_length > len_s1 else s1[min_window_start_pos: min_window_start_pos + min_window_length]
```

Java Solution

```java
class Solution {
    public String minWindow(String s1, String s2) {
        int s1Length = s1.length(), s2Length = s2.length();

        // table to store the start index of the window in s1 that ends at i and has s2.charAt(j)
        int[][] windowStartAtIndex = new int[s1Length + 1][s2Length + 1];

        // initialize the table with default values
        for (int i = 0; i <= s1Length; i++) {
            Arrays.fill(windowStartAtIndex[i], -1);
        }

        // fill the table based on the input strings s1 and s2
        for (int i = 1; i <= s1Length; ++i) {
            for (int j = 1; j <= s2Length; ++j) {
                // On matching characters, update the table with the start index of the current window
                // If it's the first character of s2, the start is the current index in s1
                // Otherwise, it's the index stored in the previous position of the table
                if (s1.charAt(i - 1) == s2.charAt(j - 1)) {
                    windowStartAtIndex[i][j] = (j == 1) ? i : windowStartAtIndex[i - 1][j - 1];
                } else {
                    // If there's no match, inherit the value from the previous index of s1
                    windowStartAtIndex[i][j] = windowStartAtIndex[i - 1][j];
                }
            }
        }

        // position and length of the minimum window
        int startPosition = 0, minLength = s1Length + 1;

        // find the smallest window in s1 that has all characters of s2
        for (int i = 1; i <= s1Length; ++i) {
            // check if the current position is the end of a valid window, i.e., it matches last character of s2
            if (s1.charAt(i - 1) == s2.charAt(s2Length - 1) && windowStartAtIndex[i][s2Length] > 0) {
                int j = windowStartAtIndex[i][s2Length] - 1; // the window's start position in s1
                int currentLength = i - j; // the length of the current window
                // update minimum length window, if the current window is smaller
                if (currentLength < minLength) {
                    minLength = currentLength;
                    startPosition = j;
                }
            }
        }

        // if no valid window is found, return an empty string
        // otherwise, return the substring from startPosition with minLength
        return minLength > s1Length ? "" : s1.substring(startPosition, startPosition + minLength);
    }
}
```

C++ Solution

```cpp
#include <cstring> // include this to use memset

class Solution {
public:
    string minWindow(string s1, string s2) {
        int mainStrSize = s1.size(), subStrSize = s2.size();
        int lastIndex[subStrSize + 1]; // lastIndex[i] will store the last index of subStr's ith character in mainStr
        memset(lastIndex, 0, sizeof(lastIndex)); // initializes lastIndex array with 0

        for (int i = 1; i <= mainStrSize; ++i) {
            for (int j = 1; j <= subStrSize; ++j) {
                if (s1[i - 1] == s2[j - 1]) {
                    // If characters match, store the index of start of the subsequence
                    lastIndex[j] = (j == 1) ? i : lastIndex[j - 1];
                } else {
                    // If characters don't match, carry forward the last index
                    lastIndex[j] = lastIndex[j - 1];
                }
            }
        }

        // Initialize variables for storing the start position and the length of the minimum window
        int startPosition = 0, minLength = mainStrSize + 1;

        // Loop to find the minimum window in s1 which has s2 as a subsequence
        for (int i = 1; i <= mainStrSize; ++i) {
            if (s1[i - 1] == s2[subStrSize - 1] && lastIndex[subStrSize]) {
                int start = lastIndex[subStrSize] - 1; // Find the start position of the subsequence
                int length = i - start; // Calculate the length of the window
                if (length < minLength) { // If this is smaller than the previously found minimum
                    minLength = length; // Update minLength with the new smaller length
                    startPosition = start; // Update the start position of the minimum window
                }
            }
        }

        // Check if a valid window was found. If minLength is still greater than mainStrSize, no valid window was found
        return minLength > mainStrSize ? "" : s1.substr(startPosition, minLength);
    }
};
```

Typescript Solution

```typescript
function minWindow(source: string, target: string): string {
    // Lengths of the source and target strings
    const sourceLength = source.length;
    const targetLength = target.length;

    // Initialize a 2D array to store the start index of the substring
    const startIndex: number[][] = Array(sourceLength + 1)
        .fill(0)
        .map(() => Array(targetLength + 1).fill(0));

    // Populate the 2D array with the start index of the substring ending with s1[i] and s2[j]
    for (let i = 1; i <= sourceLength; ++i) {
        for (let j = 1; j <= targetLength; ++j) {
            // When characters match, store the start index if it starts with the first character of s2,
            // else get the index from the previous character in source and target
            if (source[i - 1] === target[j - 1]) {
                startIndex[i][j] = j === 1 ? i : startIndex[i - 1][j - 1];
            } else {
                // If characters do not match, carry over the start index from the previous character in source
                startIndex[i][j] = startIndex[i - 1][j];
            }
        }
    }

    // Variables to store the starting point and smallest window size found so far
    let startingPoint = 0;
    let smallestWindowSize = sourceLength + 1;

    // Find the smallest window in source that contains all characters of target in order
    for (let i = 1; i <= sourceLength; ++i) {
        // Check for the last character match and if there is a valid starting index
        if (source[i - 1] === target[targetLength - 1] && startIndex[i][targetLength]) {
            // Calculate the starting index and window size
            const j = startIndex[i][targetLength] - 1;
            if (i - j < smallestWindowSize) {
                smallestWindowSize = i - j;
                startingPoint = j;
            }
        }
    }

    // If smallest window size is larger than source length, target is not found; return an empty string
    return smallestWindowSize > sourceLength ? '' : source.slice(startingPoint, startingPoint + smallestWindowSize);
}
```

Time and Space Complexity

The given Python code snippet is designed to find the smallest window in string `s1` which contains all the characters of string `s2` in the same order. It utilizes dynamic programming to achieve this.

Time Complexity

The time complexity of the code can be analyzed by looking at the nested loops:

1. The first pair of nested loops, where the outer loop runs for `m` iterations (`m` being the length of `s1`) and the inner loop runs for `n` iterations (`n` being the length of `s2`), establishes a time complexity of `O(m * n)` for the dynamic programming table population.

2. The second pair of nested loops also runs up to `m` times,and the inner operations are constant time since they're only comparing and updating values based on previously computed results. Therefore, the second nested loop does not exceed `O(m)` in complexity.

When combined, the time complexity is dictated by the larger of these loops, which is the first one. Therefore, the overall time complexity of the algorithm is `O(m * n)`.

Space Complexity

The space complexity of the code is determined by the size of the dynamic programming table `f`. Since the table is of size `(m + 1) * (n + 1)`, where `m` is the length of `s1` and `n` is the length of `s2`, the space complexity is `O(m * n)`. No additional significant space is consumed since only integer variables for bookkeeping purposes are used outside the table.

In summary, the time complexity is `O(m * n)` and the space complexity is `O(m * n)`.