Amalia Karaman

Algorithm Analysis Assignment
Questions #2 + #6

#2: Common Substring

Algorithm Analysis (text)

#2 Common Substring

A substring is a new string generated from an original string without changing character order.

Our goal is to create an algorithm that finds the longest common substring (LCS) between two strings. If there's no common substring, we return empty.

~~Initialize 2 strings:~~
~~text1~~
~~text2~~

```
longestCommonSubstring(text1, text2)
    maxLength ← 0
    endIndex ← 0

    2D dp array [m+1][n+1] set at 0
O(m x n) { for i from 1 to m:      //outer loop over text1 (O(m))
         { for j from 1 to n:      //inner loop over text2 (O(n))
        if text1[i-1] = text2[j-1]:
          dp[i][j] > maxLength:    //dp table
            maxLength ← dp[i][j]
            endIndex ← j
          else dp[i][j] ← 0
          end if
        end for
      end for
    return substring(text1, endIndex-maxLength, endIndex)
end
```

Example:
text1 = "esteban"
text2 = "amalia"

To execute we go character by character comparing.

esteban amalia

Check → 'a' match at text1[4] & text2[2]
✓ 'a' match at text1[6] & text2[5]

But no others do, (or are longer) so our output should look like:

Longest common substring: "a"

#6: Algorithm Analysis

For Problem #1, the program calculates the longest common subsequence (LCS) between two strings, text1 and text2. I chose the examples "chocolate" and "latte" because they resonated with me, and it made solving the problem feel more fun and personal. The program compares every character of text1 with every character of text2, using a 2D dptable to track the length of the LCS at each step. If two characters match, it adds 1 to the diagonal value in the table; if they don't, it takes the larger value from either above or left. The length of the subsequence gets saved in the last spot of the table, stored in dp[m][n]. The time complexity is $O(m \times n)$ because of the two nested loops. Even in the best case, where the strings match perfectly, the entire table must be filled, so the best-case complexity is also $\Omega(m \times n)$. For example, with text1 = "chocolate" and text2 = "latte", the LCS is "late", with a length of 4.

For Problem #2, the program determines the longest substring that appears in both text1 and text2. A substring must consist of consecutive characters. The program uses a 2D dp table to track how many characters match in sequence at every step. If two characters are equal, it adds 1 to the diagonal value in the table, representing the length of the substring up to that point. If they don't match, the streak breaks, and the value resets to 0. The algorithm keeps track of the largest value in the table to identify the longest substring. Its time complexity is $O(m \times n)$ because it compares all characters in both strings. Even in the best-case scenario, where the strings match perfectly, the entire table must be filled, so the best-case complexity is also $\Omega(m \times n)$. With inputs text1 = "esteban" and text2 = "amalia", the program finds "a" as the longest common substring, with a length of 1.

Problem #3 generates the NotFibonacci sequence using the formula $n[i] = 3 \times n[i-1] + 2 \times n[i-2]$. The sequence begins with 0 and 1, and each subsequent term is calculated based on the two previous terms. Since the algorithm loops through the sequence only once, its time complexity is $O(n)$, and the best-case complexity is also $\Omega(n)$ because the loop always runs exactly n times. For example, if $n = 10$, the generated sequence is 0, 1, 3, 11, 39, 139, 495, 1763, 6279, 22363.

In Problem #4, the program finds the position of a number in the NotFibonacci sequence or the closest smaller number if the input isn't in the sequence. It first generates the sequence from Problem #3 and then searches through it to locate the number. The algorithm has a time complexity of $O(n)$ because it combines generating the sequence (linear time) with searching for the number (also linear). In the best-case scenario, the input is near the start of the sequence, so the search completes quickly, resulting in a best-case complexity of $\Omega(1)$. For example, when the input is 8 and the sequence is 0, 1, 3, 11, the closest smaller number is 3, and found at position 3.

In Problem #5, the program removes all instances of a given value from an array and returns the count of the remaining elements. It does this in place by iterating through the array and skipping over the target value. Non-matching elements are shifted to the front of the array, effectively "removing" the unwanted values without creating a new array. The algorithm's time complexity is $O(n)$ because it processes each element once, and the best-case complexity is also $\Omega(n)$since every element must still be checked. With an input of nums = [3, 2, 2, 3] and val = 3, the array is modified to [2, 2], with 2 remaining elements.