Homework#6.5

Raja Kantheti

November 12, 2024

1 Largest subs sequence

1.1 Psuedocode:

```
Function longestCommonSubsequence(text1, text2):
m = length of text1
n = length of text2

# Initialize a 2D array dp with dimensions (m+1) x (n+1) and fill with 0
dp = array of size (m+1) x (n+1) filled with 0

# Iterate over each character in text1
for i from 1 to m:
    # Iterate over each character in text2
    for j from 1 to n:
        # If characters match
        if text1[i-1] == text2[j-1]:
            dp[i][j] = dp[i-1][j-1] + 1
        # If characters do not match
        else:
            dp[i][j] = max(dp[i-1][j], dp[i][j-1])

# The value at dp[m][n] contains the length of the LCS
return dp[m][n]
```

1.2 Explanation:

- Subproblems: The problem can be divided into smaller subproblems by considering prefixes of the two strings.
- Optimal Substructure: The length of the LCS for the two strings is constructed from the lengths of the LCSs of their substrings.
- State Definition: Let dp[i][j] be the length of the LCS of the first i characters of text1 and the first j characters of text2.
- Recurrence Relation: If the last characters of the current substrings are the same, they contribute to the LCS: dp[i][j] = dp[i-1][j-1] + 1. If the last characters are different, the LCS is the maximum of excluding either character: dp[i][j] = max(dp[i-1][j], dp[i][j-1]).

1.3 Problem Instance:

Take text1 = "ABCBDAB" and text2 = "BDCAB" of length 7 and 5 respectively. The final table wouldd like Table 1

		B	D	C	A	B
	0	0	0	0	0	0
A	0	0	0	0	1	1
B	0	1	1	1	1	2
C	0	1	1	2	2	2
B	0	1	1	2	2	3
D	0	1	2	2	2	3
A	0	1	2	2	3	3
B	0	1	2	2	3	4

Table 1: LCS table for text1 and text2

2 Edit distance

2.1 Psuedocode:

```
Function editDistance(str1, str2):
m = length of str1
n = length of str2
Initialize dp array of size (m+1) x (n+1) filled with 0

For i from 0 to m:
    dp[i][0] = i

For j from 0 to n:
    dp[0][j] = j

For i from 1 to m:
    For j from 1 to n:
        If str1[i-1] == str2[j-1]:
            dp[i][j] = dp[i-1][j-1]
        Else:
            dp[i][j] = min(dp[i-1][j] + 1, dp[i][j-1] + 1, dp[i-1][j-1] + 1)

Return dp[m][n]
```

2.2 Explanation:

- Subproblems: The problem can be broken down into smaller subproblems where we consider the prefixes of the two strings.
- Optimal Substructure: The solution to the problem can be constructed from the solutions to its subproblems. Specifically, the edit distance for two strings is built from the edit distances of their substrings.
- State Definition: Let dp[i][j] be the edit distance between the first i characters of word1 and the first j characters of word2.
- Recurrence Relation: If the last characters of the current substrings are the same, no new operation is needed: dp[i][j] = dp[i-1][j-1]. If the last characters are different, consider the cost of each operation (insertion, deletion, substitution) and take the minimum: dp[i][j] = min(dp[i-1][j] + 1, dp[i][j-1] + 1, dp[i-1][j-1] + 1).

2.3 Problem Instance:

word1: horse
word2: ros
The dp array would be be of dimensions 6 x 4
The minimum operationns we have to perform is 3 as seen in the table.

0	1	2	3
1	1	2	3
2	2	1	2
3	2	2	2
4	3	3	2
5	4	4	3

Table 2: Edit distance table

3 Minimum Partition:

3.1 Psuedocode:

```
Function findMin(vec):
    n = length of vec
    total_sum = sum(vec)
    dp = array of size (n+1) x (total_sum+1) filled with False

    for i from 0 to n:
        dp[i][0] = True

    for j from 1 to total_sum:
        dp[0][j] = False

    for i from 1 to n:
        for j from 1 to total_sum:
            dp[i][j] = dp[i-1][j]
            if vec[i-1] <= j:
                dp[i][j] = dp[i][j] OR dp[i-1][j-vec[i-1]]

    diff = INFINITY

    for j from total_sum // 2 to 0:
        if dp[n][j]:
            diff = total_sum - 2 * j
            break

    return diff
```

3.2 Explanation:

- Optimal Substructure: The problem can be broken down into smaller subproblems. The main goal is to find the minimum difference between two subset sums of the array. The problem has optimal substructure because, at each step, we decide whether to include or exclude an element from a subset and we can build the solution incrementally by solving smaller subproblems. These smaller subproblems (whether a subset sum can be achieved or not) are combined to construct the final solution, which is the minimum difference of partition sums.
- State Definition:dp[i][j] represents whether it's possible to form a subset with a sum of j using the first i elements from the array. The state is defined as a 2D array where: The rows (i) represent the number of elements considered (from 0 to n). The columns j represent the possible subset sums (from 0 to $total\_sum$).
- Recurrence Relation: For each element vec[i-1] and each possible sum j: If we exclude the current element vec[i-1], the subset sum remains the same as without the element, i.e., dp[i][j] = dp[i-1][j]. If we include the current element vec[i-1], the subset sum becomes j - vec[i-1], so dp[i][j] = dp[i-1][j-vec[i-1]] if j ¿= vec[i-1].