Algorithms: Assignment #9

Tanvi Magdum
magdum.t@northeastern.edu

March 18, 2023

Answers

1 Problem 1 Solution :

To find the length of the longest common subsequence of given two strings, we can use dynamic programming approach. We can follow the below algorithm -

1. Create a 2D array 'dp' of size (n+1)*(m+1), where n and m are the lengths of input strings x and y. dp[i][j] will represent the length of longest common subsequence between first i characters of x and first j characters of y.
2. Initialize the first row and first column of dp to 0, as the length of common subsequence between an empty string and any string is always 0.
3. Iterate over remaining elements of dp.
   (a) If the ith character of x is equal to the jth character of y, then, the length of longest common subsequence between first i characters of x and first j characters of y is same as the length of longest common subsequence between first (i-1) characters of x and first (j-1) characters of y plus 1, as we have a new common character present in both strings.
   (b) Else, we take the maximum of longest common subsequences between the first i characters of x and first (j-1) characters of y, and longest common subsequences between the first (i-1) characters of x and first j characters of y.
4. The final answer is stored in dp[n][m], which represents the length of the longest common subsequence between the entire strings x and y.
5. Return dp[n][m].

```
Psuedocode :

x : first input string
y : second input string

Longest_Common_Subsequence(String x, String y):
    n = length(x)
    m = length(y)
    dp = new matrix[n+1][m+1]

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

    for j = 0 to m:
        dp[0][j] = 0

    for i = 1 to n:
```

```
        for j = 1 to m:
            if x[i-1] == y[j-1]:
                dp[i][j] = dp[i-1][j-1] + 1
            else
                dp[i][j] = Max(dp[i][j-1], dp[i-1][j])

    return dp[n][m]
```

Let us consider an example. Suppose we have 2 string x = "ACBDEA" and y = "ABCDA". Create a matrix dp[n+1][m+1], i.e dp[7][6]. We have to fill in the first row and first column with zeros as the length of longest common subsequence between and empty string and any string is always going to be 0.

		A	B	C	D	A
	0	0	0	0	0	0
A	0					
C	0					
B	0					
D	0					
E	0					
A	0					

Now iterate over the rest of the cells in dp. For (1,1) we see that the 0th element of x and 0th element of y is the same, character A. So, now the length of the longest common subsequence between the first character of x and first character of y is 1. So, dp[1][1] = 1.

		A	B	C	D	A
	0	0	0	0	0	0
A	0	1				
C	0					
B	0					
D	0					
E	0					
A	0					

For (1,2), no change as we already have a common subsequence and we are comparing the same element of string x to other elements in string y, which will not make any difference in length of common subsequence. We can directly move to cell (2,2) where the range of comparison changes for both strings. The second element of x is C and that of y is B, which are not similar. Again the length remains same since until now considering sequences AC and AB, the longest subsequence is A with length 1.

		A	B	C	D	A
	0	0	0	0	0	0
A	0	1	1	1	1	1
C	0	1	1			
B	0					
D	0					
E	0					
A	0					

Moving forward to (2,3) we see that we have second element in x as C and 3rd element in y also C. So, the length of common subsequence increases by 1, as we have found length of a sequence greater than the one we already have. For (2,4), (2,5) no changes as we found a match.

		A	B	C	D	A
	0	0	0	0	0	0
A	0	1	1	1	1	1
C	0	1	1	2	2	2
B	0	1	2			
D	0					
E	0					
A	0					

We might find the next longest subsequence from cell (3,3) and after. At (3,3) we don't have a common subsequence with length greater than 2. In the whole row, we cannot find a larger subsequence, so move to next row. Take (4,4). At this cell, we have 2 same elements in x and y which is D. So the longest length is 3 now. The common subsequence at this point is 1 of 2 choices, either ABD or ACD, both are correct. However, the longest length is 3 in both cases. For (4,5) it remains same.

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

For (5,5), we don't have a common character. So, the length remains same. If we would have found a common character at this stage, then there was no need of comparing further elements as we have exhausted elements of string y and thus, there wouldn't have been a longer subsequence. But, since we didn't find any common element here, we have to compare the last element of x with all elements of y, especially the last element to find whether we get a longer common subsequence. And yes, we do. We have a common character A and at cell (6,5) length of longest common subsequence becomes 4.

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

The final answer 4 is stored in dp[6][5]. This means that the longest common subsequence between "ACBDEA" and "ABCDA" has length 4. The time complexity of this algorithm is $O(nm)$ where n is length of first string and m is length of second string. The algorithm tries to fill an $n \times m$ matrix and each cell takes a constant time to compute the result.

2 Problem 2 Solution :

The space complexity for the above algorithm of finding the longest common subsequence is $O(nm)$, where n is the length of first string and m is the length of second string. We have used dynamic programming to find the expected subsequence. From the pseudocode, we can see that the operations take place on the dp matrix. The final result is also stored in the same matrix. Therefore, it is clear that the space complexity of the algorithm primarily depends upon how much space this dp matrix requires in the computer memory.

The matrix has dimensions (n+1)*(m+1). Each cell in the matrix stores an integer value which represents the longest common sequence between corresponding prefixes of the provided two strings. To compute the values to be filled in each of the cells, we need to keep track of previously computed cell values. Specifically, to find the new longest common subsequence, while finding the maximum we need the access to previously calculated data. This complete data is stored in the dp matrix. The total number

of cells in the matrix is (n+1)*(m+1). Thus the space complexity proportional to the size of matrix. It is $O(nm)$. That means, the space required by the algorithm grows quadratically with respect to input size.

In the above example, we see the filled matrix at the end of the solution. The size of dp matrix here is dp[7][6], that is, 7×6. The end result is stored in cell dp[6][5], which the last cell of the matrix. To reach that result, we need data from all other cells. Firstly, we set the first row and first column of the matrix to 0. Then, we start comparing the characters in both strings and if the similar characters are found, we increase the count by 1. In previous example, cells dp[1][1], dp[2][3], dp[4][4] and dp[6][5] are the checkpoints where the count was increased because a new longest common subsequence was found. The space complexity is $O(nm)$, where n = 6 and m = 5 for this particular example.

3 Problem 3 Solution :

We can solve this problem of finding the longest increasing subsequence using dynamic programming approach. We can follow the below algorithm -

1. Create an array dp of length equal to length of input array 'n'.
2. Initialize all the elements of array to 1.
3. Iterate over all pairs of indices (i,j) such that i < j.
   (a) If the jth element of input array is greater than the ith element of same input array, then update the value of dp[j] as the maximum of jth element of dp and the ith element of dp plus 1.
4. Return the maximum value in dpas the length of longest increasing subsequence.

The idea behind this approach is to use an array dp of size n where dp[i] represents the length of longest increasing subsequence ending at index i. We start by initializing every element of array to 1, since each element is a subsequence of length 1 by itself. Then we iterate over pairs of indices (i,j) so that i < j. For each pair if x[j] > x[i], update dp[j] to max(dp[j], dp[i]+1). This means that we consider all possible increasing subsequences that end at index i and add x[j] to those that can be extended by x[j] to create a new subsequence that can be ended at j. At last, we return the maximum valued element from dp as the length of longest increasing subsequence.

```
Psuedocode :

x : input string

Longest_Increasing_Subsequence(String x):
    n = length(x)
    dp = new array[n]

    for i = 0 to n-1:
        dp[i] = 1

    for j = 0 to n-1:
        for i = 0 to j-1:
            if x[j] > x[i]:
                dp[j] = Max(dp[j], dp[i]+1)

    return Max(dp)
```

Let us consider an example. Suppose we have an input string x = {3,1,4,1,5,9,2,6,5}. First, declare and initialize dp as dp = {1,1,1,1,1,1,1,1,1}. Iterate on pairs of indices. For j=0, the condition for inner loop fails, and programs exits the that loop. Then for j=1 and i=0, since i < j, the program enters the inner loop. Here value of x[j] = 1 and x[i] = 3, that is x[j] < x[i]. So, dp[1] remains intact.

The next pair is j=2, i=0. Here x[j] > x[i] as 4 > 3, so dp[2] = max(dp[2], dp[0]+1) = 2. So, dp = {1,1,2,1,1,1,1,1,1}. Then, j=2 and i=1. So, x[j] > x[i] as 4 > 1, so dp[2] = max(2, 2) = 2. The dp array remains same.

Similarly cases when j=3, i=0, and j=3, i=1, and j=3, i=2 no change in dp occurs. When j=4, i=0, dp[4] = max(dp[4], dp[0]+1) = 2. When j=4, i=1, then also dp[4]=2. When j=4, i=2, dp[4] = max(dp[4], dp[2]+1) = max(2, 3) = 3. so, we got a new dp as dp = {1,1,2,1,3,1,1,1,1}. We keep on calculating further. At j=5 and i=4, the dp changes to dp = {1,1,2,1,3,4,1,1,1}. At j=6, i=1 , dp = {1,1,2,1,3,4,2,1,1}. At j=7, i=4, dp = {1,1,2,1,3,4,2,4,1}. Lastly, at j=7 and i=2, dp = {1,1,2,1,3,4,2,4,3}.

So, the final array dp = {1,1,2,1,3,4,2,4,3}. The max of dp is 4. Hence, the length of longest increasing subsequence in this example is 4. This subsequence could be either of - {3,4,5,9} or {1,4,5,9} or {3,4,5,6} or {1,4,5,6}, regardless the length would be constant.

The time complexity of this algorithm is $O(n^2)$, since we iterate over all pairs of indice (i,j) so that i < j and for each pair we are performing constant amount of work of comparing 2 elements and updating an array. So, the total number of operations is proportional to number of pairs of indices, which is $O(n^2)$.

The space complexity here is $O(n)$, where n is the length of input array. We are using an array dp of size n to store the length of longest increasing sequence ending at each index i. This array is initialized to 1 and then updated iteratively as we process the sequence. Therefore total space required is proportional to the length of input sequence $O(n)$. This algorithm has the optimal space complexity.