Longest Repeating Subsequence

Harsh Kedia
IIB2019028

Milap Anwani
IIB2019029

Kandagatla Meghana Santhoshi
IIB2019030

***Abstract*—In this paper, we have devised an algorithm which when given a string, prints the longest repeating subse- quence such that the two subsequences don't have same string character at same position, i.e., any j'th character in the two subsequences will not have the same location index in the given original string. We have used dynamic programming approach to solve the problem and also analysed the time and space complexity of the same.**

***Index Terms*—Subsequence,String,Dynamic Programming**

I. Introduction

This document describes the procedure followed to find longest repeating sub-sequence of given string using dynamic programming.

II. Algorithm Design

A. Dynamic Programming

Dynamic programming is mainly used in optimising solutions.Similarly like Divide and Conquer, Dynamic Programming also solves problems by combining solutions of sub-problems. Unlike Recursion it stores solutions of sub-problems dynamically for later use to avoid solving the sub-problems again and again.

We prefer solving a problem using Dynamic programming if we identify Overlapping Sub-problems and optimal substructure of its sub-problems.This approach reduces time complexity to linear from exponential by storing solutions of sub-problems to avoid solving sub-problems again and again.

B. Dynamic Programming Components

This technique can be categorised into these steps:

Stages - Analyse and identify the structure of an optimal solution of the problem and its sub problems.Each sub-problem can be called as stage

States - Each stage has many states associated with it. A state of the problem usually represents a sub-solution, i.e. a partial solution or a solution based on a subset of the given input. And the states are built one by one, based on the previously built states.

Decision - At each stage, there can be multiple choices. From these choices , we can choose best decisions according to the problems.The decision taken at every stage should be best according to the problem , this is called as stage decision.

Optimal Policy - This determines the decision at each stage , a rule is called an optimal policy if it is globally optimal.

C. Steps to solve Dynamic Programming Problems

This technique uses four steps :

Step-1 : Identify the problem as dynamic problem , by checking the properties i.e; Overlapping Sub-problems and optimal substructure.

Step-2 : Decide a state expression containing least number of parameters.

Step-3 : Formulate the relationship between the states.

Step-4 : Add tabulation or memoization to store the values of the sub-problems which have already been solved.

III. Algorithmic Analysis

A. Algorithmic Steps for calculating Longest repeating sub-sequence :

To find the Longest repeating sub-sequence of a given string:

1) We input the string from the user to find longest repeating sub-sequence and pass the string to the function findLRS.
2) In this function , we initialise DP table locally. Here we are developing a bottom-up solution to find Longest common repeating sub-sequence.
3) Check if the characters match and are at different indexes , say i and j.
4) If yes then store the value the value in DP table at index i,j by adding one to the DP table[i-1][j-1].
5) Else store the DP table at index i,j as the maximum value of DP table at index i-1,j and i,j-1 i.e; store the maximum of top and left as the value of DP table[i][j].
6) We do this till we reach the end of the DP table.
7) After the table is done, now we try to print the longest repeating sub-sequence of a given string.
8) If this cell is greater than diagonally adjacent cell just above it, then same characters are present at Input[i-1] and Input[j-1]. Append any of that to result.If left cell is greater than top, then move left else top.
9) Print the output result and to obtain the length of the result we can return DP table[n][n].
    - In this way, We can Print Longest repeating sub-sequence and the length of it recursively.

IV. Pseudo Code

table[n+1][n+1] stores the solutions of the sub-problems calculated.n is a variable to store the length of the string Input.String output stores the result. count variable stores length of the longest repeating sub-sequence.

```
findLRS Function:

    n <- size of Input
    for i <- 0 to n
        for j <- 0 to n
       initialise  table[i][j] <- 0
    for i <- 1 to n  for j <- 1 to n
    if Input[i-1] == Input[j-1] and if i!=j
        table[i][j] =  1 + table[i-1][j-1]
    else
    table[i][j]=
            max(table[i][j-1],table[i-1][j])

    count <- table[n][n]
    Assignment i <- n and j <- n
    out <- "\0"
    while count !=0 and i > 0 and j > 0
        if table[i][j]
              >max(table[i][j-1],table[i-1][j])
            output[count-1]=Input[i-1];
            i=i-1
            j=j-1
            count = count -1
        else if table[i][j-1] > table[i-1][j]
            j=j-1
        else
            i=i-1

    for i <- 0 to table[n][n]
        print output[i]
```

V. Priori Analysis of LRS

This section explains Priori Analysis of finding the Longest Repeating sub-sequence of the given string.

A. Time Complexity Analysis

We can observe that, we are using nested for loops in the findLRS function to store solution of the sub-problems.Hence , Complexity can be calculated as $O(n^2)$, where n is the length of the string Input.

B. Space Complexity

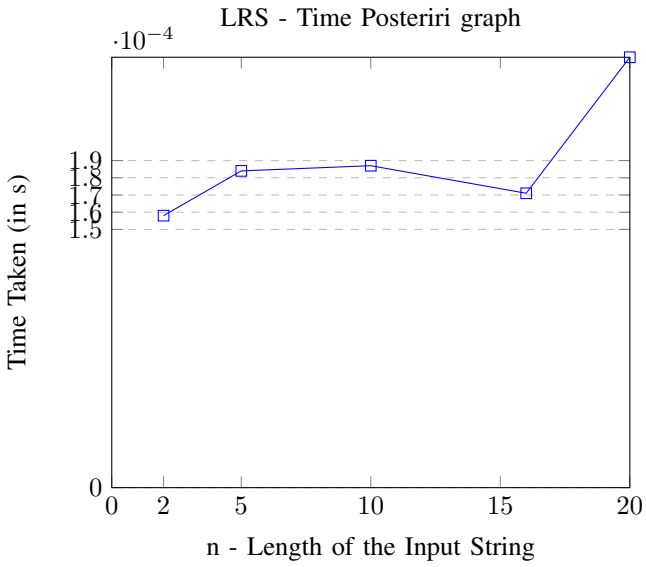
The space complexity of the Program is $O(n^2)$ Because of the space allocation for resultant LRS in each subproblem.We solve every sub-problem till the end , that can be produced/

VI. Experimental Analysis

A. Time Complexity

In the following table some cases are plotted

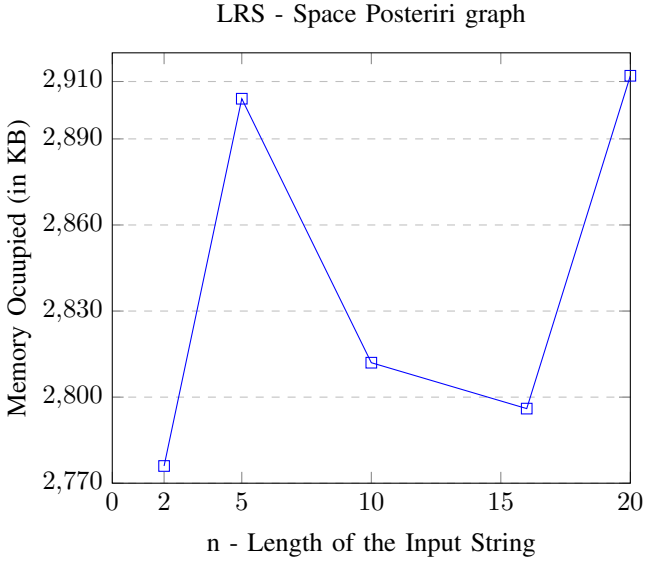
Input	n	Time Taken (in s)
ab	2	0.000158
aabaa	5	0.000184
aaaaaaaaaa	10	0.000187
abaacddefrferghh	16	0.000171
aabaddefrghhuvdsused	20	0.000250



B. Space Complexity

In the following table some cases are plotted

Input	n	Space Occupied (in KB)
ab	2	2776
aabaa	5	2904
aaaaaaaaaa	10	2812
abaacddefrferghh	16	2796
aabaddefrghhuvdsused	20	2912



VII. Conclusion

Using Dynamic Programming, we have reduced our time complexity to $O(n^2)$ from the complexity of $O(2^n)$ obtained by using brute force or naive solution from using recursion.

VIII. Acknowledgment

We are very much grateful to our Course instructor Mr.Rahul kala and our mentor, Mr.Md Meraz, who have

provided the great opportunity to do this wonderful work on the subject of Data Structure and Algorithm Analysis specifically on methodologies of Dynamic Programming.

IX. References

We have referred [1] and [2] to clear the basic concepts of Dynamic programming.Reference [3] helped us , to develop solution of longest repeating subsequence.

References

[1] https://en.wikipedia.org/wiki/Dynamic-programming
[2] http://web.mit.edu/15.053/www/AMP-Chapter-11.pdf
[3] https://www.geeksforgeeks.org/printing-longest-common-subsequence

X. Appendix

A. *Project link on Github:*

https://github.com/maggi2k19/DaaAssignments