# Longest Repeating Subsequence (Dynamic Programming)

Group 29 :
Harsh Kedia - IIB2019028
Milap Anwani - IIB2019029
Kandagatla Meghana Santhoshi - IIB2019030

**Department of Information Technology**
**Indian Institute of Information Technology, Allahabad**

## Content

## Abstract

In this paper, we have devised an algorithm which when given a string, prints the longest repeating subsequence 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.

## Introduction

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

# Algorithm description and analysis

1. We input the string from the user to find longest repeating subsequence 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 subsequence.
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 subsequence of a given string.
8. While count is 0,i and j are equal to 0 , do this .If this cell is greater than top and left adjacent cells , 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].

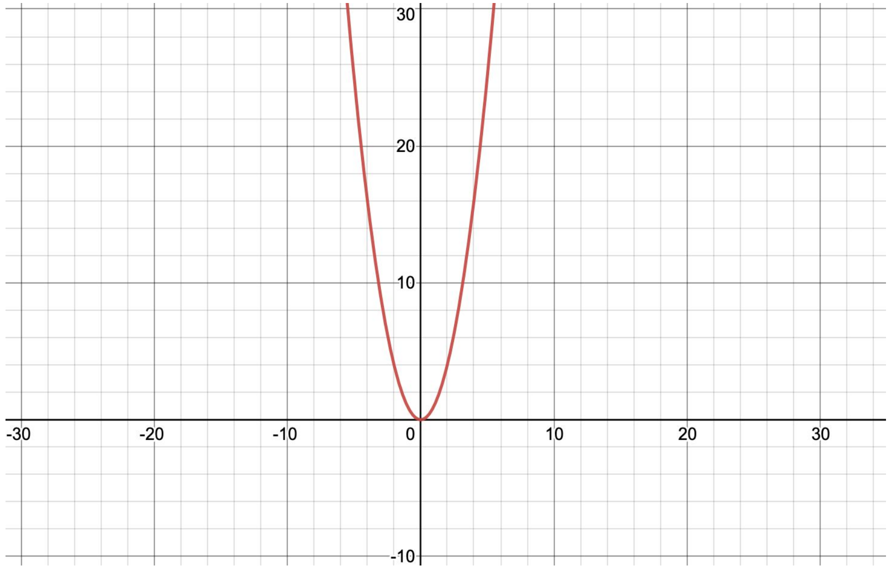
# Pseudo code

**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 ( t a b l e [ i ] [ j – 1 ] , t a b l e [ i – 1 ] [ j ] )
count <– table [n][n]
Assignment i <– n and j <– n
output<– ”\0”
while count !=0 and i > 0 and j > 0
if table[i][j]
>max ( t a b l e [ i ] [ j – 1 ] , t a b l e [ 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 ]
```

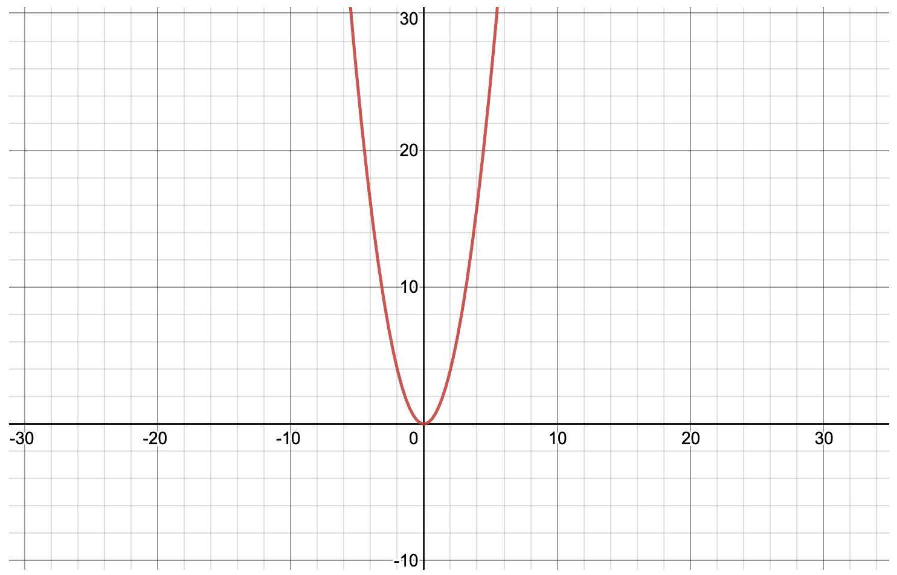
## Time Complexity

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.



## 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



## 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.

## 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

THANK YOU