Design and Analysis of Algorithms
DAA432C
Assignment 2
Group - 14

Mukul Mohmare **IIT2019226** **Anshuman Bharadwaj** **IIT2019227** **KishanTripathi** **IIT2019225**

Abstract* — *Given a string, print 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 shouldn't have the same location index in the given original string. Solve using dynamic programming.

I. INTRODUCTION

We are given a string we need to find a subsequence which is repeating and is the longest one in the given string.

And we are going to use dynamic programming to solve this question.

Dynamic programming is a very powerful technique to solve a particular class of problems. It demands very elegant formulation of the approach and simple thinking and the coding part is very easy. The idea is very simple, If you have solved a problem with the given input, then save the result for future reference, so as to avoid solving the same problem again.. shortly *'Remember your Past'* :) . If the given problem can be broken up into smaller subproblems and these smaller subproblems are in turn divided in to still-smaller ones, and in this process, if we observe some overlapping subproblems, then it's a big hint for Dynamic Programming.There are two ways of doing this.

1.) Top-Down : Start solving the given problem by breaking it down. If you see that the problem has been solved already, then just return the saved answer. If it has not been solved, solve it and save the answer. This is usually easy to think of and very intuitive. This is referred to as ***Memoization***.

2.) Bottom-Up : Analyze the problem and see the order in which the sub-problems are solved and start solving from the trivial subproblem, up towards the given problem. In this process, it is guaranteed that the subproblems are solved before solving the problem. This is referred to as ***Dynamic Programming***.

Subsequence is a sequence that can be derived from another sequence by deleting some or no elements without changing the order of the remaining elements.

Now suppose we choose a string S
S = "ATACTCGGA"
Output: "ATCG"
Here we observe that "ATCG" is repeating twice and is the longest among all the repeating sequences in this string.

let's take another example with string S
S = "AABCBDC"
Output: "ABC"

A	A	B	C	B	D	C
0	1	2	3	4	5	6

First

A	B	C
0	2	3

Second

A	B	C
1	4	6

For this example our dynamic programming solution will give output in O(n^2) complexity.

Where ,
n =length of given string

This report further contains -
II. Algorithm Design
III. Code And Illustration
IV. Algorithm Analysis
V. Conclusion
VI.References

II. ALGORITHM DESIGN

For this problem , the main idea is to find the subsequence which is repeating for the longest length and **with the restriction that when both the characters are the same , they shouldn't be on the same index in the two strings.**

We first prepare a table which stores the longest common subsequence between the given string and a copy of the same string but this time we need to keep

a check that **when both the characters are the same , they shouldn't be on the same index in the two strings.**

Now Initialize a resulting string
s1 = "";
Traverse t[][] from bottom right
i = n;
j = n;
while (i > 0 && j > 0)
{
If this cell is the same as the diagonally adjacent cell just above it, then the same characters are present at str[i-1] and str[j-1]. Append any of them to string s1.

```
if (t[i][j] == t[i-1][j-1] + 1)
{
  s1= s1+ str[i-1];
  i--;
  j--;
}
```

Otherwise we move to the side that that gave us maximum result else if (t[i][j] ==
t[i-1][j])
i--;
else
j--;
}
Since we traverse t[][] from bottom, we get the result in reverse order.
reverse(s1.begin(), res.end());
return s1;

III. CODE AND ILLUSTRATION

Approach -1 Using Tabulation (Bottom Up)

```
#include <bits/stdc++.h>
using namespace std;

string longestRepeatedSubSeq(string str)
{

    int n = str.length();
    int dp[n+1][n+1];
    for (int i=0; i<=n; i++)
        for (int j=0; j<=n; j++)
            dp[i][j] = 0;
    for (int i=1; i<=n; i++)
        for (int j=1; j<=n; j++)
            if (str[i-1] == str[j-1] && i != j)
                dp[i][j] =  1 + dp[i-1][j-1];
            else
                dp[i][j] = max(dp[i][j-1], dp[i-1][j]);


    string res = "";

    int i = n, j = n;
    while (i > 0 && j > 0)
    {

        if (dp[i][j] == dp[i-1][j-1] + 1)
        {
            res = res + str[i-1];
            i--;
            j--;
        }

        else if (dp[i][j] == dp[i-1][j])
            i--;
        else
            j--;
    }


    reverse(res.begin(), res.end());

    return res;
}

// Driver Program
int main()
{
    string str;
    cin >> str;
    cout << longestRepeatedSubSeq(str);
    return 0;
}
```

Illustration - Approach -1
Input : String = "AABCBDC"
Output : ABC
n = string length // 7
Here length of string is 7
Creating dp table of size 8x8
If String character matches and indexes are different then
dp[i][j] = 1 + dp[i-1][j-1];
Else
dp[i][j] = max(dp[i][j-1] , dp[i-1][j]);

After generating Dp table we initialise an empty string (res="") to store the ans and print the string
Traverse the dp table in reverse order
Int this case we will traverse from dp[8][8]
if (dp[i][j] == dp[i-1][j-1] + 1)
Then res = res + str[i-1];
First character 'c' is added to res

Then character ‘b’ is added and at the last character ’a’ is added to res.

Finally reversing the res string and printing the output .

Approach-2

Using Recursion with Memoization (Top-Down)

```
#include <bits/stdc++.h>
using namespace std;
 int dp[1001][1001];
int longestRepeatedSubSeq(string str,int n,int m)
{
    if (dp[n][m]!=0)
    return dp[n][m];
    if (n==0||m==0)
    return 0;
    else if (str[n-1]==str[m-1]&& n != m)
   return dp[n][m]=1+longestRepeatedSubSeq(str,n-1,m-1);
    else
   return dp[n][m]=max(longestRepeatedSubSeq(str,n-1,m),longestRepeatedSubSeq(str,n,m-1));



}


int main()
{
    string str;
    cin >> str;
    memset(dp,0,sizeof(dp));
    int n=str.length();
    longestRepeatedSubSeq(str,n,n);

   string res = "";

    int i = n, j = n;
    while (i > 0 && j > 0)
    {

        if (dp[i][j] == dp[i-1][j-1] + 1)
        {
            res = res + str[i-1];
            i--;
            j--;
        }

        else if (dp[i][j] == dp[i-1][j])
            i--;
        else
            j--;
    }


    reverse(res.begin(), res.end());


    cout<<res;
}
```

Illustration - Approach -2

Same as the previous approach only difference here is we use recursion instead of iteration.

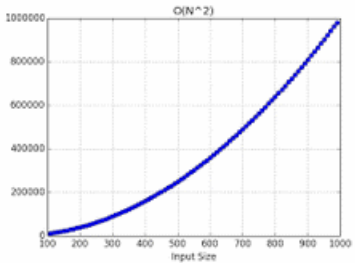
```
if (dp[n][m]!=0)
    return dp[n][m];
    if (n==0||m==0)
    return 0;
    else if (str[n-1]==str[m-1]&& n != m)
   return dp[n][m]=1+longestRepeatedSubSeq(str,n-1,m-1);
    else
   return dp[n][m]=max(longestRepeatedSubSeq(str,n-1,m),longestRepeatedSubSeq(str,n,m-1));
```

Dp table after recursive calls

IV. ALGORITHM ANALYSIS

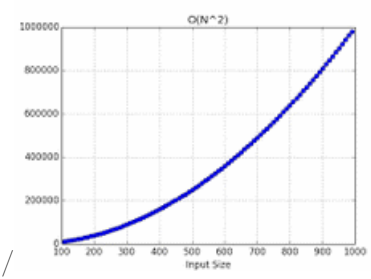
Time Complexity - Time complexity is the number of operations an algorithm performs to complete its task (considering that each operation takes the same amount of time). The algorithm that performs the task in the smallest number of operations is considered the most efficient one in terms of the time complexity.

Time Complexity – O(n^2)



Space Complexity - The space complexity of an algorithm is the amount of memory space required to solve an instance of the computational problem as a function of characteristics of the input. It is the memory required by an algorithm until it executes completely.

Space Complexity – O(n^2)



//

V. CONCLUSION

Here we have seen booth approaches of dynamic programming to solve this question with great ease . Memoization approach is really very easy to think and once a recursive solution is framed iterative code is also generated

Using Dynamic programming we solved this problem in quadratic complexity which proves the efficiency of Dynamic Programming solution.

VI.REFERENCES

https://www.geeksforgeeks.org/longest-repeated-subsequence/
https://www.techiedelight.com/longest-repeated-subsequence-problem/
https://www.geeksforgeeks.org/longest-repeating-subsequence/