Analysis & Design of Algorithms

Semester III 2018-19

Lab - 6

Topics: To solve the problem using Dynamic Programming (DP)

INTRODUCTION

In this lab we would be finding longest common subsequence (LCS) of given two sequences using Dynamic Programming.

DYNAMIC PROGRAMMING

DP is another technique for problems with optimal substructure. An optimal solution to a problem contains optimal solutions to subproblems. This doesn't necessarily mean that every optimal solution to a subproblem will contribute to the main solution.

- For divide and conquer (top down), the subproblems are independent so we can solve them in any order.
- For greedy algorithms (bottom up), we can always choose the "right" subproblem by a greedy choice.
- In dynamic programming, we solve many subproblems and store the results: not all of them will contribute to solving the larger problem. Because of optimal substructure, we can be sure that at least some of the subproblems will be useful

SOLVING DYNAMIC PROGRAMMING

Steps to solve a DP problem

1. Define subproblems
2. Write down the recurrence that relates subproblems
3. Recognize and solve the base cases

EXAMPLE

To compute the n^{th} term of fibonnaci series.

Native Approach

```
fib(n)
        if n \leq 2
                f = 1
        else f= fib(n-1)+fib(n-2)
return f
```

Memorization

```
memo={}
fib(n)
        if n is in memo
                return memo[n];
        if n \leq 2
                f = 1;
        else f= fib(n-1)+fib(n-2);
        memo[n]=f;
return f
```

Using Dynamic Programming

```
int fib(int n){
        if n \eq 1
                return n
        F[0]=0; F[1]=1;
        for i=2 to n
                F[i]= F[i-2]+F[i-3];
        return F[n]
}
```

EXERCISE : LONGEST COMMON SUBSEQUENCE (LCS)

Definition: The Longest Common Subsequence (LCS) problem is as follows. We are given two strings: string *S* of length *n*, and string *T* of length *m*. Our goal is to produce their longest common subsequence: the longest sequence of characters that appear left-to-right (but not necessarily in a contiguous block) in both strings. For example, consider:

S = ABAZDC

T = BACBAD

In this case, the LCS has length **4** and is the string **ABAD**. Another way to look at it is we are finding a 1-1 matching between some of the letters in *S* and some of the letters in *T* such that none of the edges in the matching cross each other.

LCS using native approach

```
int LCS (i,j){
        if( A[i] == '\o' || B[i] == '\o' )
                return 0;
        else if (A[i] == B[j])
                return 1+ LCS (i+1,j+1);
        else
                return MAX (LCS(i+1,j), LCS(i,j+1));
}
```

LCS using DP

```
int LCS ( i , j ){
        for i=0 to m;
                for j=0 to n
                        if (i == 0 || j == 0)
                                L[i][j] = 0;
                        else if (A[i] == B[j])
                                LCS[ i , j ] = 1+ LCS (i - 1,j - 1);
                        else
                                LCS[ i , j ] = MAX( LCS (i - 1,j ), LCS ( i ,j - 1)
}
```