CS6363 - Design and Analysis of Algorithms:

HOMEWORK-3

Name: HIMA SRI .T
UTDID: 2021624644

1) Exercise 15.4-2 on page 396 from CLRS:
Give pseudocode to reconstruct an LCS from the completed c table and the original sequences $X = \langle x_1, x_2, ---, x_m \rangle$ and $Y = \langle y_1, y_2, --- y_n \rangle$ in $O(m+n)$ time without using the b table.

Sol: Input:

$X = \langle x_1, x_2, ---, x_m \rangle$
$Y = \langle y_1, y_2, ---, y_n \rangle$

Output: To print the longest common subsequence (LCS)

Subproblem space:

$c[m, n]$ → A matrix of m×n dimensions to store the length of the longest common subsequence of X & Y.

$c[i, j]$ → contains length of LCS of X_i & Y_j

Recursive formula:

i) If $x_m = y_n$
$z_k = x_m = y_n$

Z = result LCS = $\{z_1, ---, z_k\}$

Scanned with CamScanner

(ii) Else if $x_m \neq y_n$
z_k is LCS of x_{m-1} & y_n or

z_k is LCS of x_m & y_{n-1}.

Hence we can deduce recursive formula as follows:

$$c[i,j] = \begin{cases} 0 & \text{if } i=j=0 \\ 1+c[i-1,j-1] & \text{if } x_i = y_j \\ \max\{c[i-1,j], c[i,j-1]\} & \text{if } x_i \neq y_j \end{cases}$$

Pseudo code for finding LCS_length(X,Y):

```
LCS_length(X,Y):
1) m = X.length;
2) n = Y.length;
3) Let c[0,---,m, 0,---,n] be the table to store the
                                             length.
4) for i=1 to m:
        c[i,0]=0;
5) for j=1 to n:
        c[0,j] = 0;
6) for i = 1 to m:
       for j=1 to n:
7)         If xi == yj
               c[i,j]= c[i-1,j-1] +1
8)         Else if
               c[i-1,j] >= c[i,j-1]
                   c[i,j] = c[i-1,j]
9)         else
               c[i,j] = c[i,j-1]
10) return c;
```

Pseudo code to print the LCS using C-table and X & Y sequences:

```
1) index = c[m,n];
2) Initialise char-array LCS[0,---,index+1]; i=m, j=n
3) lcs[index+1] = '\0';
4) while i>0 && j>0
5)        If xi==yj
          {lcs[index] = xi
           i = i-1;
           j = j-1;
           index = index-1; }
6)        else
          if c[i-1,j] > c[i,j-1]
           { i = i-1; }
7)        else
           { j = j-1; }
```

~~It~~ its runs in a single while loop which is executed m+n times.
Hence, time complexity is O(m+n).

2) Given a set of coin values $C = \{c_1, c_2, \dots c_m\}$. Find minimum number of coins to represent n cents with coins in C. For instance, if the coin value set is $C = \{1, 4, 9\}$, then best way to make $n = 16$ cents is to use 4 four-cent coins. Therefore, the optimal solution is 4. Design an $O(mn)$-time DP algorithm to solve this problem.

Sol: Input: $C = \{c_1, c_2, ---, c_m\} \rightarrow$ Set of coin Values
A value n to be represented.
Eg. Here $C = \{1, 4, 9\}$, $n = 16$.

Output: Minimum number of coins to represent n cents with coins in C.
Eg. Here Output = 4 which is the optimal solution

Subproblem space:

This above problem has an optimal substructure and overlapping subproblems.

~~Here, for the subproblem space is calculating the number of coins needed to~~

Optimal substructure:
The optimal solution for a particular change can be found out by making the optimal choice in the subresults obtained.
Overlapping subproblems:
In the recursive solutions there are many subproblems which occur repeatedly. Hence, it has overlapping subproblems

Scanned with CamScanner

Hence, DP approach can be used.
Let us consider an array result[0, -- n].

result[n] → stores the minimum number of coins needed to make the change n.
For all the coins in $C = \{C_1, C_2 ---, C_m\}$, let us start by picking a first coin C_1.
The value needed to calculate now is $n - C_1$.
Now we need to calculate minimum ~~sss~~ number of coins needed to make the change $n - C_1$.
At this stage; the total no of coins needed are:
$result[n] = 1 + result[n - C_1]$ // Since already 1 coin is picked

Similarly, we can pick second coin C_2 and the value needed to calculate now is $n - C_2$.
$result[n] = 1 + result[n - C_2]$.

Likewise, we choose all the m coins and we need to calculate the minimum no. of coins from all the results; Hence

$$result[n] = \begin{cases} \min \left\{ result[n - C_d] + 1 \text{ where } d = 1 \text{ to } m \right\} & \text{if } n > 0. \\ 0 & \text{if } n = 0. \end{cases}$$

Here, calculating the optimal solution for subproblem result[n - Cd] gives the optimal solution of problem.
Hence, optimal substructure property holds.

There are many subproblems which are calculated multiple times. Hence, overlapping subproblems property also holds.

The subproblem space is
result[0, -- n] → which stores the minimum number of coins required to make a value i.

Recursive formula:

$$result[i] = \begin{cases} 0 & \text{if } n == 0. \\ \min\limits_{i=0 \text{ to } m,\ c[i] \le n} \{1 + result[n - c[i]]\} & \text{if } n > 0 \end{cases}$$

PseudoCode:

```
1) Initialise result[0, ... n] to INT_MAX.
2) for i = 1 to n
3)     for j = 1 to m
4)         If (c[j] <= i)
               sub_result = result[i - c[j]]
5)             If (sub_result != INT_MAX &&
                                 sub_result + 1 ≤ result[i])
6)                 result[i] = sub_result + 1.

7) return result[n];
```

Time Complexity: The outer for loop runs n times and the inner loop runs m (no. of coins) times. Hence, the time complexity is O(mn).

③ Tribonacci numbers are a generalisation of Fibonacci numbers and defined as:

$$T(n) = \begin{cases} 0 & \text{if } n=0 \\ 0 & \text{if } n=1 \\ 1 & \text{if } n=2 \\ T(n-1)+T(n-2)+T(n-3) & \text{if } n=3. \end{cases}$$

To compute n^{th} Tribonacci number;

→ Design a recursive algorithm

Sol: Pseudocode for recursive algorithm:

Tribonacci (n) : // Function to calculate n^{th} Tribonacci number

```
1) If (n == 0 or n == 1):
       return 0;
2) If (n == 2)
       return 1;
3) else
       return Tribonacci(n-1) + Tribonacci(n-2) +
                                Tribonacci(n-3);
```

→ Design a bottom-up DP algorithm and Top-down DP algorithm.

I/P: N

O/P: N^{th} Tribonacci number.

Scanned with CamScanner

Recursive formula:

$$T(n) = \begin{cases} 0 & \text{if } n=0 \\ 0 & \text{if } n=1 \\ 1 & \text{if } n=2 \\ T(n-1)+T(n-2)+T(n-3) & \text{if } n \geq 3 \end{cases}$$

$T(n) \rightarrow$ stores the n^{th} Tribonacci number.

Subproblem space:

$T[n] \rightarrow$ An array which stores the ~~fibon~~ Tribonacci numbers so far.

Bottom-up DP algorithm:

Tribonacci(n):

1) If n==0 or n==1:
   return 0;
2) If n==2
   return 1;
3) Initialise T[n]; // array to store tribonacci numbers
4) T[0]=0; T[1]=0; T[2]=1;
5) for i=3 to n:
   T[i] = T[i-1]+T[i-2]+T[i-3];
6) return T[n];

Bottom-up algorithm builds from bottom in the following way:

$T[1] \rightarrow T[2] \rightarrow T[3] \rightarrow \cdots \rightarrow T[n]$.

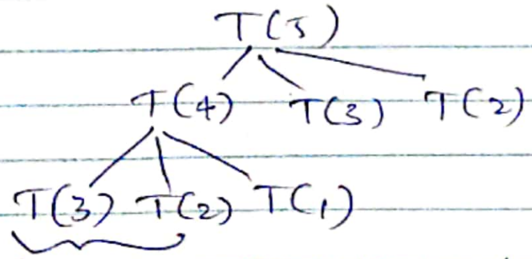
Time Complexity for calculating Tribonacci numbers is : O(n). since there is only 1 for loop where the numbers are calculated.

Top-down DP algorithm:

Tribonacci(n, T) : // T is an array to store Tribonacci numbers;

1) Initialize T[ ] = -1 for all values of i = 1 to n;
2) If n == 0 or n == 1
   T[n] = 0;
   return T[n];
3) If n == 2
   T[n] = 1;
   return T[n];
4) else If T[n] ≠ -1
   return T[n];
5) else
   T[n] = Tribonacci(n-1, T) + Tribonacci(n-2, T) + Tribonacci(n-3, T);
6) return T[n];

Top down approach calculates the numbers from n and uses the stored values when needed.



already calculated values are used.