

# CSCI-GA.1170-001 Homework 7

Ankit Sati

TOTAL POINTS

**70.5 / 57**

QUESTION 1

1 Maximize Your Scores 36 / 26

- ✓ - 0 pts Correct
- ✓ - 1 pts Missing/Incorrect base case in part (f)
- ✓ + 1 pts Correct definition of  $\text{Best}[i,j]$  in part (g)
- ✓ + 2 pts Correct base case in part (g)
- ✓ + 2 pts Correct recursive formulation in part (g)
- ✓ + 3 pts Correct top down algorithm in part (g)
- ✓ + 1 pts Correct invocation call in part (g)
- ✓ + 1 pts Correct recursive solution to original problem given best in part (g)
- ✓ + 1 pts Correct run time in part (g)

① Best[2] = max(A[1], A[2])

QUESTION 2

2 Spacing Out LCS 23 / 19

- ✓ - 0.5 pts Q 7-2 a) running time incorrect/missing
- ✓ + 3 pts Q 7-2 c) Correct (Extra Credit)
- ✓ - 0.5 pts Q 7-2 f) Incorrect/missing space complexity
- ✓ - 1 pts Q 7-2 f) Incorrect/missing correctness
- ✓ + 3 pts Q 7-2 g) algorithm correct (extra credit)

QUESTION 3

3 Counting BSTs 11.5 / 12

part d

- ✓ - 0.5 pts Incorrect/Missing running time

## Problem Set -7.

7-1.

a) Solution :-

- 1) Initialise array B which stores set of indices to be returned
- 2) Iterate array A, ( $i = 1$  to  $n$ )  
if  $A[i] > 0$ :  
append  $i$  to B.
- 3) Return B.

Iterate over the array A and return only those indices  $i$  where the value of  $A[i]$  is positive.

If all values in Array A are negative, then it will be the least to pick none of the score.

$\Rightarrow$  We pick none of the scores if  $\forall i \ A[i] \leq 0$ . The total sum in this case is 0 as we cannot add any -ve scores to our sum. (Hence sum would be  $< 0$ )

(ii)  $\forall i, A[i] \geq 0$ .

$\Rightarrow$  We need to return start & end indices of  $A[1 \dots n]$

$\Rightarrow$  Since all the scores are non-negative, they won't reduce our sum total.

They will either increase it ( $A[i] > 0$ ) or don't affect it ( $A[i] = 0$ ) but they ~~will not~~ will not reduce our sum.

$\therefore$  All scores must be returned

Soln  $\Rightarrow$

Return  $(1, n)$  of array  $A[1 \dots n]$ .

(c) GFT MVCS. (A, m)

maxSum = 0;  
for. i=1 to n.

current = 0

for j=1 to n.

current = current + A[j]

if maxSum < current

then maxSum = current.

return maxSum.

- Best[m] will be a one-dimensional array where  $\text{Best}[i]$  will tell the maximum sum of continuous scores ending at index  $i$ .

- Base Call  $\text{E1} = A[1]$

- Recursive Formula  $\Rightarrow \text{Best}[i] = \max(\text{Best}[i-1] + A[i], A[i])$ .

- Recursive Formula  $\Rightarrow \text{Best}[i] = \max(A[i], A[i] + \text{Best}[i-1]) \quad \{ i=2, \dots, n \}$   
 Justification  $\rightarrow$  At each iteration we decide whether we include the previous best value or not with the current value of Array A. If we want to include  $A[i]$  or  $A[i] + \text{BEST}[i-1]$ .  
 $\Rightarrow$  We choose whichever is maximum.

- Algorithm

$\rightarrow$  Compute-BEST(A, m)

1.  $\text{Best}[1] = A[1]$ .
2. for  $i = 2$  to  $m$ .
3.  $\text{Best}[i] = \max(\text{Best}[i-1] + A[i], A[i])$ .

Runtime  $O(n)$   
 Correctness  $\rightarrow$  Recursive formulation and loops through the length of Array A to compute BEST values at each index.  
 Base case is correctly defined.

$\rightarrow$  Required Sum Calculation (BEST) Runtime  $O(n)$

1. ~~max = 0~~ max = 0
  2. for  $i = 1$  to  $m$ .
  3. if  $m < \text{Best}[i]$   
 $m = \text{Best}[i]$
  4. Return  $m$ .
- Compute-MaxSum(A, m) } The Max value corresponds to maximum sum possible over a continuous sub-array.

- Compute-MaxSum(A, m) will return the continuous maximum sum for Array A.

- Since we are only doing a linear traversal over the array A.  
 Therefore the run time of Algorithm is  $O(m)$ .

- Basically in our recursive formulation we are either taking ~~Best[i]~~ Best[i] or not. That is why we have 2 choices of  $\text{Best}[i]$  &  $\text{Best}[i] + A[i]$  to choose amongst.

### (c). Print-SOLUTION(BEST).

```
1. max = -∞  
2. end = -1.  
3. for i = 1 to n  
    if max < best[i]  
        max = best[i].  
        end = i.  
7. Start = -1.  
8. for (i = end, down to 1.  
9.     if BEST[i] = A[i]  
10.         start = i  
11.         break.  
12. return Start, end.
```

Runtime  $O(n)$   
the algorithm has 2 linear loops  
& successfully runs in  $O(n)$  trip.

### Approach.

- First we need to find the index  $i$  such that  $\text{Best}[i]$  is maximum and this index will be the ending index ~~of~~ of the Contiguous Subarray.
- Now, we will traverse back from this index and will search for an index where  $\text{Best}[i]$  equals  $A[i]$  and this index will be the starting index of the Contiguous Subarray.

- [f) .  $\text{Best}[n]$  will be a one dimensional array where  $\text{Best}[i]$  will tell the required answer for the prefix of length  $i$  of  $A$ .  
i.e  $A[0 \dots i]$
- Base Cases.

$$\begin{aligned}\text{Best}[1] &= A[1] \\ \text{Best}[2] &= A[2]\end{aligned}$$

Justification  $\Rightarrow$  if  $BEST[i-2]$  is non-negative then we compare between  $BEST[i-2] + A[i]$  and  $BEST[i-1]$   
 otherwise we compare between  $A[i]$  or  $BEST[i-1]$   
 this way we only look at adjacent scores. I include only true best scores.

Recursive Formula  $\rightarrow BEST[i] = \max_{i=3 \dots n} (A[i], A[i] + BEST[i-2], BEST[i-1])$

$$BEST[i] = \begin{cases} \max(BEST[i-1], BEST[i-2] + A[i]) & \text{if } BEST[i-2] \geq 0 \\ \max(BEST[i-1], A[i]) & \text{if } BEST[i-2] < 0. \end{cases}$$

- In our recursive formulation, if  $BEST[i-2]$  is non-negative then we have two choices.

- Take sum of  $BEST[i-2] + A[i]$
- $BEST[i-1]$

Otherwise again we have two choices to take either  $A[i]$  or  $BEST[i-1]$ .

Basically, we are doing this thing so that no adjacent are taken together. like if  $A[i]$  is taken then we are not taking  $BEST[i-2]$  but not  $BEST[i-1]$ .

- COMPUTE -  $BEST(A, m)$ .

if  $BEST[m]! = -\infty$   
 return  $BEST[m]$

if  $m=1$   
 return  $A[1]$

if  $m=2$   
 return  $A[2]$

$BEST[m-2] = \text{Complete-BEST}(A, m-2)$

if  $BEST[m-2] \geq 0$ .

8

$$BEST[m] = \max (\text{Compute-BEST}(A, m-1), BEST[m-1] + A[i]).$$

return ~~BEST~~  $BEST[m]$

- Invocation (call.  $\rightarrow$ )  
 $\text{COMPUTE-BEST}(A, n)$

- Required Sum Calculation  $\rightarrow$  (Finding Maximum Sum)

$\text{COMPUTE-MAXsum}(A, n)$       Runtime =  $O(1)$   
 1. RETURN  $BEST[m]$       Stored at =  $O(n)$

Runtime. Our Algorithm uses dynamic programming, which has only one state. So the sum time will be linear i.e.  $O(1)(n)$ . Values are stored in a top-down manner to we finish in linear time.

- (g) •  $BEST[i, j]$  represents the maximum sum for a prefix of length  $i$  of  $A$  i.e.,  $A[1 \dots i]$  such that exactly ~~exactly~~  $j$  different indices are selected.

- Base Case  $\rightarrow$ .

for column  $j=1$ , the value of  $BEST[i, 1]$  will be maximum element upto prefix of length  $i$  in  $A$ .

$$BEST[1, 1] = A[1]$$

$$BEST[i, 1] = \max [A[i], BEST[i-1, 1]].$$

- Recursive Formulation  $\rightarrow$

$$BEST[i, j] = \max (A[i] + BEST[i-1, j-1], BEST[i-1, j]).$$

$$i=1 \dots m$$

$$j=2 \dots k.$$

We have two choices, either to take  $A[i]$  or not. If we are-

taking  $A[i]$ , then will sum  $A[i]$  and best  $[i-1, j-1]$  otherwise we will take  $BEST[i-1, j]$  and we will find the maximum out of these two values.

- $\text{COMPUTE-BEST}(A, i, j)$

- 1 if  $BEST[i, j] \neq -\infty$   
return  $BEST[i, j]$ .
- 2 if  $i = 1 \& j = 1$   
return  $A[i]$ .

- 3 if  $j = 1$ .  
return  $BEST[i, 1] = \max(A[i], \text{COMPUTE-BEST}(A, i-1, 1))$ .

- 4 if  $i < j$ :  
return  $-\infty$ .
- 5  $BEST[i-1, j-1] = \text{COMPUTE-BEST}(A, i-1, j-1)$ .
- 6  $BEST[i-1, j] = \text{COMPUTE-BEST}(A, i-1, j)$
- 7  $BEST[i, j] = \max(A[i] + \text{BEST}[i-1, j-1], \text{BEST}[i-1, j])$

return  $BEST[i, j]$

- Invocation call.

$\text{COMPUTE-BEST}(A, n, k)$

- Required sum calculation  $\rightarrow$  (Find MaxSum)  
 $\text{COMPUTE-MaxSum}(A, n, k)$

1. return  $BEST[n, k]$

size the best array: stores the maximum possible value. until k picks for n elements in  $BEST[n, k]$ .

- Our algorithm uses two states  $i$  and  $j$  such that ( $1 \leq i \leq m$ ) and ( $1 \leq j \leq k$ ). So it will take runtime of  $O(nk)$ .

### Problem 7-2

(a). PRINT-LCS-INDEX.( $c, x, y, i, j$ )

if ( $i = 0$  or  $j = 0$ )

return "

if  $x[i] == y[j]$

$z = \text{PRINT-LCS-INDEX}(c, x, y, i-1, j-1)$

$z = z + x[i]$

else if  $c[i-1, j] >= c[i, j-1]$

$z = \text{PRINT-LCS-INDEX}(c, x, y, i-1, j)$

else.

$s = \text{PRINT-LCS-Index}(c, x, y, i, j-1)$

return  $s$ :

We are using a recursive algorithm for calculating the LCS of  $x$  &  $y$ . If  $x[i]$  &  $y[j]$  are same, then we are finding the LCS for  $x_{i-1}$  &  $y_{j-1}$  ad. appending  $x[i]$  to this LCS.

If  $c[i-1, j] >= c[i, j-1]$ , then we are finding LCS for  $x_{i-1}$  &  $y_j$  otherwise we are doing the opposite.

At the end we are returning the LCS stored in  $s$ .

Runtime of algorithm is  $O(mn)$  as we have 2 states  
 $1 \leq i \leq m$  &  $1 \leq j \leq n$ .

## 1 Maximize Your Scores 36 / 26

- ✓ - 0 pts Correct
- ✓ - 1 pts Missing/Incorrect base case in part (f)
- ✓ + 1 pts Correct definition of  $\text{Best}[i,j]$  in part (g)
- ✓ + 2 pts Correct base case in part (g)
- ✓ + 2 pts Correct recursive formulation in part (g)
- ✓ + 3 pts Correct top down algorithm in part (g)
- ✓ + 1 pts Correct invocation call in part (g)
- ✓ + 1 pts Correct recursive solution to original problem given best in part (g)
- ✓ + 1 pts Correct run time in part (g)

1 Best[2] = max(A[1], A[2])

- Our algorithm uses two states  $i$  and  $j$  such that ( $1 \leq i \leq m$ ) and ( $1 \leq j \leq k$ ). So it will take runtime of  $O(nk)$ .

### Problem 7-2

(a). PRINT-LCS-INDEX.( $c, x, y, i, j$ )

if ( $i = 0$  or  $j = 0$ )

return "

if  $x[i] == y[j]$

$z = \text{PRINT-LCS-INDEX}(c, x, y, i-1, j-1)$

$z = z + x[i]$

else if  $c[i-1, j] >= c[i, j-1]$

$z = \text{PRINT-LCS-INDEX}(c, x, y, i-1, j)$

else.

$s = \text{PRINT-LCS-Index}(c, x, y, i, j-1)$

return  $s$ :

We are using a recursive algorithm for calculating the LCS of  $x$  &  $y$ . If  $x[i]$  &  $y[j]$  are same, then we are finding the LCS for  $x_{i-1}$  &  $y_{j-1}$  ad. appending  $x[i]$  to this LCS.

If  $c[i-1, j] >= c[i, j-1]$ , then we are finding LCS for  $x_{i-1}$  &  $y_j$  otherwise we are doing the opposite.

At the end we are returning the LCS stored in  $s$ .

Runtime of algorithm is  $O(mn)$  as we have 2 states  
 $1 \leq i \leq m$  &  $1 \leq j \leq n$ .

(a).  $LCS\_LENGTH(x, y, m, n)$ .

let  $c[0][0..m]$  &  $c[1][1..n]$  be two array.

$$c[0][0] = c[1][0] = c[0][1] = 0$$

for  $i = 1$  to  $m$ .

for  $j = 1$  to  $n$ .

if  $x[i] == y[j]$

$$c[i \bmod 2][j] = c[(i-1) \bmod 2][j-1] + 1$$

else  $c[i \bmod 2][j] = \max(c[i \bmod 2][j-1],$   
 $c[(i-1) \bmod 2][j])$ .

return  $c[m \bmod 2]$

(c)  $LCS\_LENGTH(x, y, m, n)$

let  $c[0..m]$  be array.

$$c[0] = 0$$

for  $i = 1$  to  $m$

$$\text{prev} = 0$$

$$\text{len} = 0$$

for  $j = 1$  to  $n$ .

$$\text{len} = c[j]$$

if  $x[i] == y[j]$

$$c[i] = \text{prev} + 1$$

else  $c[i] = \max(c[i], c[i-1])$

$$\text{prev} = \text{len}$$

return  $c$ .

(d)

Theorem:

- If  $x$  &  $y$  are of length  $m$  and  $n$  respectively, then if  $x[i]$  and  $y[i]$  are same, then the LCS. of  $x$  and  $y$  will be the same as LCS of suffix of length  $m-1$  of  $x$  and suffix of length  $n-1$  of  $y$ , appended to  $x[i]$ .

i.e.

$$\text{LCS}(x, y) = x[i] + \text{LCS}(x^{m-1}, y^{n-1})$$

*if  $x[i] = y[i]$*

otherwise if  $x[i]$  &  $y[i]$  are not the same the LCS of  $x$  &  $y$  will be the maximum out of LCS of  $x^{m-1}, y$  and LCS of  $x, y^{n-1}$

$$\text{LCS}(x, y) = \max (\text{LCS}(x^{m-1}, y), \text{LCS}(x, y^{n-1}))$$

*if  $x[i] \neq y[i]$*

Recursive Formula.

$$\text{if } x[i] = y[j]$$
$$c[i, j] = c[i+1, j+1] + 1$$

$$\text{else. } c[i, j] = \max (c[i+1, j], c[i, j+1])$$

LCS-LENGTH( $x, y, m, n$ )

Let  $c[1 \dots m+1, 1 \dots n+1]$  be the new table.

for  $i = 1$  to  $m$

$$c[i, m+1] = 0$$

for  $j = 1$  to  $n$

$$c[m+1, j] = 0$$

for  $i = m$  down to 1.

for  $j = m$  down to 1.

$$\text{if } x_i == y_j \\ c[i, j] = c[i+1, j+1] + 1.$$

else

$$c[i, j] = \max(c[i+1, j], c[i, j+1])$$

return.

(e)  $L(x, y)$  represents the length of  $x$  and  $y$ .

Now for a prefix of  $x$  of size  $m/2$  we will always get same.

$$j^* \text{ such that } 0 \leq j^* \leq m, \text{ and} \\ L(x_{m/2}, y_{j^*}) + L(x_{m/2}, y^{m-j^*}) = L(x, y).$$

So basically, if we divide  $x$  into 2 equal parts and divide  $y$  into a prefix of length  $j^*$  and suffix of length  $m-j^*$ . Then the sum of the length of the LCS. for  $x_{m/2}$ ,  $y_{j^*}$  and  $x_{m/2}$ ,  $y^{m-j^*}$  will be the same as the length of LCS for  $x, y$ .

(f)  $L(x, y, m, n)$

$$\text{if } m = 1 \text{ and } x[m] == y[m] \\ \text{return } 0, 1$$

$$\max = -\infty$$

$$j^* = -1$$

for  $i = 0$  to  $n$

$$\text{if } \max < (p[i]).$$

$$\max = p[i]$$

$$j^* = j$$

$\text{tmp}, S_1 = L(x_{m/2}, Y_{j^*}, m/2, j^*)$   
 $\text{tmp}, S_2 = L(x^{m/2}, Y^{m-j^*}, m/2, m-j^*)$

return  $j^*, S_1 + S_2$ .

Runtime =  $O(mn)$

Space Complexity  $\Rightarrow O(1)$ . excluding the space for recursion stack

(g). LCS. ( $x, y, m, n$ )

if  $m == 1$  and  $x[m] == y[m]$   
return  $x[m]$ .

$j = \text{FIND\_J}(x, y, m, n)$   
 $str\ 1 = LCS(x_{m/2}, Y_j, m/2, j)$   
 $str\ 2 = LCS(x^{m/2}, Y^{m-j}, m/2, m-j)$

return  $str\ 1 + str\ 2$ .

$T(m, n) = T(m/2, j) + T(m/2, m-j) + O(n)$

Runtime  $\rightarrow O(nm)$

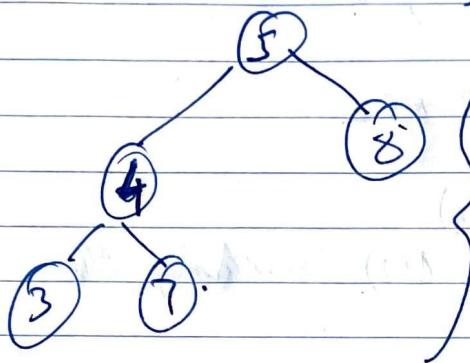
Space  $\rightarrow O(n)$  excluding the space for recursion stack.

2 Spacing Out LCS 23 / 19

- ✓ - 0.5 pts Q 7-2 a) running time incorrect/missing
- ✓ + 3 pts Q 7-2 c) Correct (Extra Credit)
- ✓ - 0.5 pts Q 7-2 f) Incorrect/missing space complexity
- ✓ - 1 pts Q 7-2 f) Incorrect/missing correctness
- ✓ + 3 pts Q 7-2 g) algorithm correct (extra credit)

## Q Problem 7-3.

(a) Example of Binary tree where the algorithm will return True



This leads to (5) not checking that (1) is to its right, hence violating BST property.

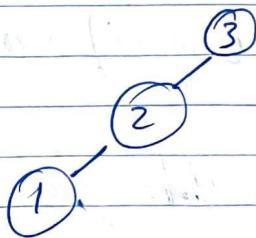
The algorithm is only checking whether left child value is less than its parent or not and right side child's value is greater than parent or not.

Instead it should work upon the ranges in which a value of a node must lie.

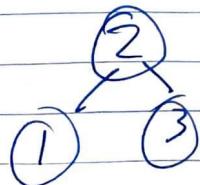
Therefore the algorithm is wrong.

(b) All possible BST, for n=3:

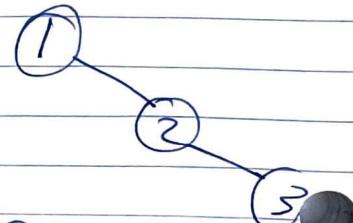
1.



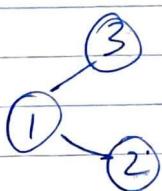
2.



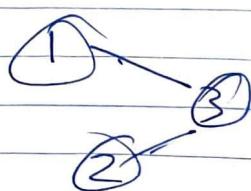
3.



4.



5.



So, a total of 5 BST are possible for  $n=3$ .

(c) No of trees possible with  $i$  as root as  $F_{i-1} F_{m-i}$

Recursive Formulation  $\rightarrow$

$$F_n = \sum_{i=1}^n F_{i-1} F_{m-i}$$

Where  $F_n$  denotes the number of possible BST's with  $n$  elements.

Boundary conditions:

- Empty BST:  $m=0, F_m=1$
- $m=1, F_m=1$  (root)

(d) Calculate -  $F_m(n)$ :

- 1). If  $n = 0$  or  $n = 1$ ,  
RETURN 1
- 2). If  $M[n] \neq -1$ ,  
RETURN  $M[n]$
- 3).  $f = 0$
- 4). For  $i=1$  to  $n$   
 $f = f + [\text{Calculate } F_m(i-1), \text{Calculate } F_m(n-i)]$
- 5).  $M[n] = f$
- 6). RETURN  $M[n]$ .

⇒ Function

Initializ  $M[1 \dots n] = -1$   
 $F_m = \text{Calculate } F_m(n)$

Runtime  $O(n)$ .

3 Counting BSTs 11.5 / 12

part d

✓ - 0.5 pts *Incorrect/Missing running time*