FA assignment 7

Q2:

2.(a)

```
Print-LCS-Index ( c, X, Y, i, j )
{
  if i == 0 or j == 0 :
          return

  if X[i] == Y[j] :
      Print_LCS_Index (c, X, Y, i-1, j-1)
      print ( X[i])

  else if ( c[i-1, j] ≥ c[i, j-1]):
      Print_LCS_Index (c, X, Y, i-1, j)

  else :
      Print_LCS_Index ( c, X, Y, i, j-1)
}
```

→ Call Print_LCS_Index (c, X, Y, m, n)

→ We have the matrix c.
→ In the base case, if i and j are 0, we return from our procedure.
→ When we call for m and n, if X[m] equals Y[n], we call the procedure with m-1, n-1; after which we print the value.
→ If X[m] does not equal Y[n], we check c for understanding how c[m,n] has been filled. i.e. with respect to which of the previous cells.
→ We know that X[m] isn't part of the LCS, we then call the print procedure for the corresponding previous cell.
→ We repeat this procedure recursively until the base case.

$$T(n, m) = O(n + m)$$

We decrement by 1 either n or m or both in each recursive call.

2. (b)

LCS- Length (X, Y, m, n)

LCS - Length (X, Y, m, n)

{

$c[0][0, \_\_, n]$ and $c[1][0, \_\_, n]$ be two orrays.

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

```
for i = 1 to m:
    for j = 1 to n:
        if X[i] = Y[j]:
            c[i mod 2][j] =
            c[1 - i mod 2][j-1] + 1

        else:

            c[i mod 2][j] =

            max(c[1 - i mod 2][j],
                c[i mod 2][j-1])


return c[m mod 2]
```

}

→ For some iteration i in order to fill

- for some iteration i, in order to fill in the j values for Y; we only need the current row i and the previous row (i-1).

→ i mod 2 will give us the current array of c under consideration.

→ When we are filling in the current value of c[j] at iteration i, we check if X[i] equals Y[j], if true, we add one to the prevous array value (j-1) corresponding to c[i-1][j-1] in the last question. We assign this to the current array value c[j].

→ Else, we compare current array value [j-1] and previous array value [j], assigning the max to current array value [j]. This is in accordance with the conditions in standard LCS.

$$T(n, m) = \Theta(mn)$$
$$\text{Space complexity} = O(n)$$

Q2.(c)

LCS_Length (X, Y, m, n)

```
LCS_Length (X, Y, m, n)
{
    let c[0, —, n] be array

    c[0] = 0

    for i = 1 to m :
        prev = i-1              # c[i-1][j-1]

        temp = 0

        for j = 1 to n :

            temp = c[j]         # for next j

            if ( X[i] == Y[j] ) :

                c[j] = prev + 1

            else :

                c[j] = max ( c[j-1] ,
                             c[j] )

            prev = temp
                        # store the previous
                          diagonal value
```

return c

}

→ For some i, as we enter iteration j, our $c[j]$ will contain $c[i-1][j]$ with respect to the original c matrix.

→ We store this in temp. Prev denotes $c[i-1][j-1]$ from original matrix. We maintain the prev value for each iteration.

→ Based on the condition for equality, we update $c[j]$ to reflect $c[i][j]$.

→ Else, $c[j]$ will be max of $c[i-1][j]$ # which is essentially the $c[j]$ value bedore update, and, $c[i][j-1]$, # this is the updated value for $c[j-1]$.

→ Thus for each i, c will redlect the LCS for X_i and Y_j where j goes from 1 to n.

$$T(n, m) = \Theta(mn)$$

Space complexity - single array
- $O(n)$

Q2.d.

(i) Theorem:

Suffix based LCS of X^i and Y^j is equal to the suffix based LCS of X^{i+1} and Y^{j+1} ($i+1$ and ahead, $j+1$ and ahead) plus one, if $X[i]$ equals $Y[j]$.
Else; LCS if the maximum of the LCS of X^{i+1} and Y^j or X^i and Y^{j+1}.

(ii)

$$\therefore \quad c[i,j] = \begin{cases} c[i+1, j+1] + 1 & \text{if } X[i] = Y[j] \\ \max(c[i+1, j], c[i, j+1]) & \text{if } X[i] \mathrel{!=} Y[j] \end{cases}$$

(iii)

```
LCS - Length - Suffix (X, Y, m, n)
{
    let c [1, ___ , (m+1),  1, ___ , (n+1)]
    be new table

    for i = 1 to m :
        c [i, n+1] = 0

    for j = 1 to n :
        c [i, m+1] = 0

    for i = m downto 1 :
        for j = n down to 1 :
            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] )
```

$- , j+1)$

return c

}

$$T(n, m) = \theta(mn)$$

2. (e)

$X[1, \_\_, m/2, \_\_ m]$

$Y[1, \_\_\_\_\_\_\_\_, n]$

Case I:

$m/2$ is a part of the LCS.

Let $l = \alpha_1, \_\_ \alpha_p, x_{m/2}, \beta_1, \_\_, \beta_r$

be the LCS. This is without loss of generality. $|LCS| = p + 1 + r$

There will be a correspondence between the α's, $x_{m/2}$, β's in X and elements in Y.

For the element which corresponds with $x_{m/2}$, let it be y_j.

We know that the LCS is l, therefore, elements in Y with indices less than L will have a one to one correspondence with the α's. Similarly, elements in Y that are greater than j will have a one to one correspondence with the β values.

$$\therefore \text{Prefix\_LCS}(X_{m/2}, Y_{j-1}) = k$$

$$\text{Suffix\_LCS}(X^{m/2}, Y^{j-1}) = 1 + r$$

$$\therefore LCS(X_{m/2}, Y_{j-1}) + LCS(X^{m/2}, Y^{j-1})$$

$$= 1 + k + r$$

$$= |LCS(X, Y)|$$

This can be shown for Y_{j+1} as well.

Case II:

$x_{m/2}$ is not a part of the LCS.

let the LCS String be,

$l = \alpha_1 — \alpha_k \; B_1 — B_r$

$$|LCS| = k + r$$

$\rightarrow \alpha_k < x_{m/2} < \beta_1$

Let y_j be the element which corresponds to α_k.

$$LCS(X_{m/2}, Y_j) = k$$

$$LCS(X^{m/2}, Y^j) = r$$

$$\therefore LCS(X_{m/2}, Y_j) + LCS(X^{m/2}, Y^j)$$

$$= k + r$$

$$= |LCS|$$

$\rightarrow$ We have shown without loss of generality that there exists j which satisfies the conditions for j^*. Therefore, j^* exists.

2.f

```
Find_J (X, Y, m, n)
{
    let cp, cs, l be arrays from [0,_,n]

    cp [0,__,n] = LCS_length_Prefix
                      (X, Y, m/2, n)

    cs [0,__,n] = LCS_length_Suffix
                      (X, Y, m/2, n)

    l [0,__,n] = LCS_length (X, Y, m, n)

    val = l [n]

    for i from 1 to n:

        l [i] = cp [i] + cs [i]

    for j from 1 to n:

        if l [j] == val:
            j* = j
```

return j^*, val

}

→ The running time of this algorithm is $\Theta(mn)$
→ The find length predix step takes $\Theta(mn)$ time.
→ The find length suffiv step takes $\Theta(mn)$ time.
→ We find $LCS(X_{m/2}, Y_j)$ in array C_p and $LCS(X^{m/2}, Y^{n-j})$ in array C_s.
→ We find the value of $LCS(X, Y)$ in $\Theta(mn)$ time.
→ The next steps take linear time as we check for j^*.

$$T(m,n) = \Theta(mn)$$

2.g.

LCS (X, Y, m, n)

{

```
if (m == 1):

    for i = 1 to n:
        if Y[i] == X[1]:
            return X[1]


if (n == 1):

    for i = 1 to m:
        if X[i] == Y[1]
            return Y[1]

if (m == 0 or n == 0):
    return Nil
```

$j = \text{Find\_J}(X, Y, m, n)$

$c_1 = LCS(X_{m/2-1}, Y_{j-1}, m/2-1, j-1)$

$c_2 = LCS(X^{m/2+1}, Y^{n-(j+1)}, m/2, n-j)$

return str($c_1 \cdot j \cdot c_2$)

}

→ When we find j^*, we know that elements in the LCS before j^* are also before $m/2$ and elements in the LCS after j^* are also after $m/2$.

→ Hence we can call the procedure recursively on the two subarrays.

The find-J procedure takes $\theta(mn)$ time

The appending of strings takes $O(n)$ time. $\theta(mn)$ term will dominate.

$$\therefore T(m,n) = T(m/2 - 1, j-1) + T(m/2, n-j) + \theta(mn)$$

→ Inductive hypothesis, $T(n,m) = \theta(mn)$

→ Induction base case $T(1 \cdot 1) = \theta(1 \cdot 1)$

$= \theta(1)$

let $T(m/2 - 1,\ j-1) = \theta((m/2 - 1)\ (j-1))$

$= \theta(mj)$

$T(m/2\ ,\ n-j) = \theta(m/2 \cdot (n-j))$

$= \theta(\frac{mn}{2} - \frac{mj}{2})$

$= \theta(mn) - \theta(mj)$

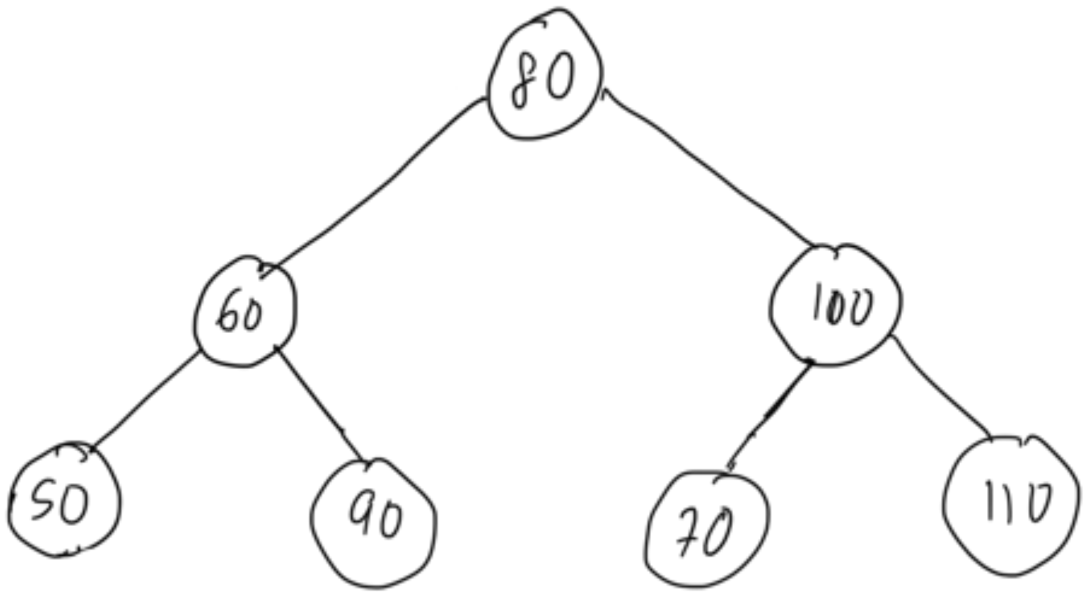
$\therefore T(n,m) = \theta(mj) + \theta(mn) - \theta(mj)$

$+ \theta(mn)$

$\therefore T(n,m) = \theta(mn)$

Q3.

3. a.

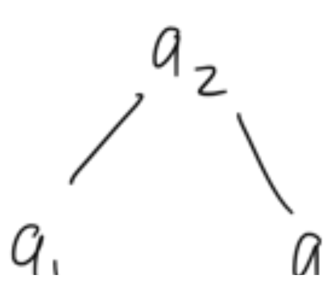


$80 > 60$ and $80 < 100$
recursion passes to trees rooted at 60 and 100.
Again subtrees are BST's but the tree as a whole isn't. $90 > 80$ and in the left sub tree. $70 < 80$ and in the right sub tree

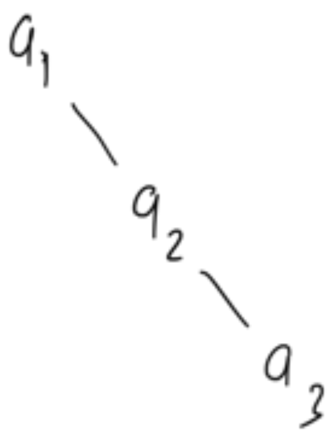
3. b.

$n = 3$
$A = [a_1, a_2, a_3]$

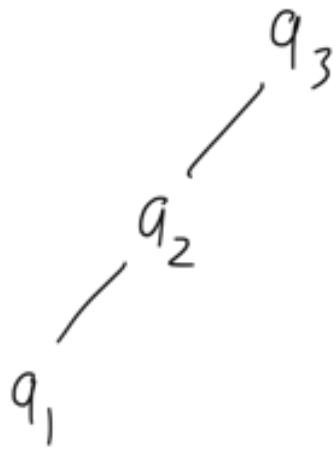
1.



2.

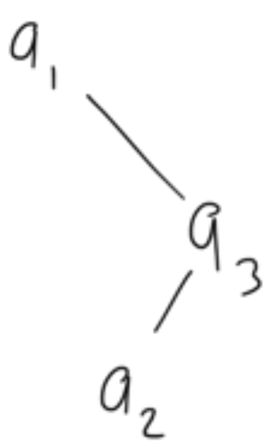


3.



4.

5.



3. c

F_n is the number of possible BSTs of size n.

With i as root, we will have F_{i-1} BSTs on the ledt and F_{n-i} BST's on the right.

$\therefore$ No. of possible combinations for trees with i as root $\equiv F_{i-1} * F_{n-i}$

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

d.

Array M [1, ___ , n] initialized to zeros.

M[1] = 1

F(n)

{

if (n == 0) :

```
    return 1

  if M[n] != 0:
        return M[n]

  for i from 1 to n:

      M[n] = M[n] + F(i-1) * F(n-i)

  return M[n]

}

# Call F(n)
```

Q1.

1.a

- In one pass over the array, we select the positive scores.
- We create a buffer array to store the

positive values and return it along with it's length.

1.b.

- Select indices (1, n)
- Since all elements are greater than equal to zero, they will only add to the total score.

1.c.

```
Get Mvcs (A, n)
{

    max sum = 0
    for i = 1 to n :

        current = 0
        for j = i to n :
            current = current + A[j]

            if (current > maxsum):
                max sum = current


    return maxsum
```

2.

}

$$T(n) = O(n^2)$$

1.d.

→ Best [n] - Array [1, —, n] will contain the best sub array sum ending in n.

→ Best [k] - max sum of sub array ending in k.

→ We use this to find the overall max sum sub array.

$$Best[k] = \begin{cases} A[k] + Best[k-1] & \text{if } A[k] + Best[k-1] \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

Best [1, —, n] initialized to $-\infty$.

Compute_Best (A, n)

{

```
if A[1] > 0 :
        Best [1] = A[1]

for i from 2 to n :
    if A[i] + Best [i-1] >= 0 :

          Best [i] = A[i] + Best [i-1]

    else:
       Best [i] = 0

}

Find_ Global_ Best (Best)

{  max = 0
   for i from 1 to n :
        if Best [i] > max :
              max = Best [i]

   return max

}
```

- Computing Best takes a pass over A, finding the global best takes a pass over Best, both of size n.

$$T(n) = O(n)$$

- The maximum sum sub array has to end somewhere. We select the corresponding max value.

i.e.

```
Print_Solution (Best)
{

    # start, end - indices to be returned.

    s = 1
    max = 0

    for i from 1 to n :

        if max < Best[i] :
            max = Best[i] :
            start = s
            end = i

        if Best[i] < 0 :
            s = i + 1
```

return start, end

}

$T(n) = O(n)$

We update the max value and hence the start, end values. If the maximum subarray ending in i is less than zero, we recalibrate our anchor variable s.

Q1-e.

Best [n] - Array [1, —, n]

Best [k] contains the maximum sum value until k based on the conditions.

Base case:

$$\text{Best}[1] = A[1]$$
$$\text{Best}[2] = \max(A[2], A[1])$$

Recursive formulation:

$$Best[k] = \max\left(A[k] + Best[k-2], \; Best[k-1] \right)$$

Initialize array Best to $-\infty$. Adjust base cases $Best[1]$, $Best[2]$ as discussed.

```
Compute_Best (A, n)
{
    if Best[n] != -∞:
        return Best[n]

#   if n == 1:
        return Best[1]

#   if n == 2:
        return Best[2]

    Best[n] = max ( A[n] + Compute_best (A, n-2)
                  ,            )
```

Compute_Best (A, n-1))

```
return Best[n]

}

Invocation call - Compute_Best (A, n)


Find_Global (Best)
{
    max = 0
    for i from 1 to n :
            if max < Best[i] :
                    max = Best[i]

    return max
}
```

$T(n) = O(n)$

The compute Best procedure takes $O(n)$ time as we use dynamic programming as applied to this problem. The find max procedure takes one pass over Best.

1.g.

Best = $n \times k$

Best[i, j] will contain the maximum sum of elements upto i such that j elements are allowed.

$$Best[1,1] = A[1]$$

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

$$\vdots$$

$$Best[n,1] = \max(A[n], Best[n-1,1])$$

$$Best[1,2] = A[1]$$

$$Best[i][j] = \max\{ Best[i-1][j], A[i] + Best[i-1][j-1])$$

```
# Best n×k global array initialized to -∞

# Initialization

Best [1, 1] = A[1]

for i from 2 to n :
        Best [i, 1] = max (Best [i-1, 1],
                           A[i])


Compute_Best (A, i, j)

{
    if Best [i, j] != -∞ :
          return Best [i, j]


    Best [i, j] = max (
                   (Compute_Best (A, i-1, j))
                   ,
               A[i] +
                    Compute_Best
                         (A, i-1, j-1))
                 )

                                   }
```

return Best [i, j]

}

Invocation call : Compute_Best (A, n, k)

- Maximum sum of elements when we have to pick k scores is Best [n, k] obtained by calling Compute_Best (A, n, k)

$T(n, k) = \Theta(nk)$

- Proportional to the size of our table.