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.

Traverse over the array A and return only those indices i where the value of A[i] is positive.
If all value in Array A are negative, then it will be the best to pick none of the score.

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

(b) ∀ i , A[i] ≥ 0.
⇒ We need to return start & end indices of A[1 . . . n]
⇒ Since all the scores are non negative, they wont reduce our sum total.
they will either increase it (A[i] > 0) or dont affect it (A[i] = 0)
but they ~~are not~~ will not reduce our sum.
∴ All scores must be returned

Soln ⇒.
Return (1, n) of array A[1 . . . n].

(c) GET MVCS. (A, n.)

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.

(d). Best[n] will be a one-dimensional array where Best[i] will tell the maximum sum of continous scores ending at index i.

- Base Case E1] =- A[1]
- Recursive Formula → Best[i] =. max (Best[i-1]. + A[i], A[i]).

- Recursive Formula → $Best[i] = max(A[i], A[i] + Best[i-1])$ } $i = 2 \ldots n$.

Justification →. 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 $A[i] + BEST[i-1]$
⇒ We choose whichever is maximum.

- Algorithm

→ Compute_BEST (A, n)

1. Best [1] = A[1].
2. for i = 2 to n.
3. Best [i] = max(Best[i-1] + A[i] ; A[i])

Runtime O(n).
Correctness →. Recursive formula and loops through the length of Array A to compute BEST value at each index
Base case is correctly defined.

→ Required Sum Calculation

Compute_MaxSum (BEST, n)

1. ~~max~~ = 0
2. for i = 1 to n.
3. if m < Best [i]
   m = Best [i]
5. Return m.

Runtime O(n)
The max value corresponds to maximum sum possible over a continuous sub-array.

- Compute_MaxSum (A, n) 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(n).

- Basically in our recursive formulation we are either taking ~~best~~ Best [i] or not that is why we have 2 choices of Best [i] & Best [i] + A[i] to choose amongst.

(e). Print_SOLUTION (BEST).

```
1.  max = -∞
2.  end = -1.
3.    for i = 1 to n
4.          if max ≤ Best [i]
5.              max. = Best[i]
6.                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 $O(n)$ time.

Approach.

→ First we need to find the index i such that Best [i] is maximum and this index will be the ending index of the Contigous Subarray.

→ Now, we will traverse back from this index and will search for an index where Best [i] equals A[i] and this index will be the starting index of the Contigous Subarray.

(f) • Best [n] will be a one dimensional array where Best [i] will tell the required answer. for the prefix of length i of A.
i.e A[1.....i]

• Base Cases.

Best [1] = A[1]
Best [2] = A[2]

Justification → 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 & include only true & best scores.

Recursive Formula → BEST[i] = max.(A[i], A[i] + BEST[i-2], BEST[i-1]) / i = 3 ..... n.

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

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

(i) Take Sum of BEST[i-2] + A[i]
(ii) BEST[i-1]

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

Basically, we are doing this this so that no adjacent are taken from the sum. like if A[i] is taken then we are taking BEST[i-2] but not Best[i-1]

- COMPUTE - BEST (A, n).

```
if BEST[n] != -∞
        return Best[n]
if n == 1
        return A[1]
if n == 2
        return A[2]
Best[n-2] = Compute_BEST(A, n-2)

if BEST[n-2] ≥ 0.
```

BEST[n] = max(Compute_BEST(A, n-1), BEST[n-2] + A[i])

return ~~Best~~ BEST[n]

- Invocation call. →
  COMPUTE_BEST(A, n)

- Required Sum Calculation →. (Finding Maximum Sum)

```
COMPUTE_MAXSUM(A,n)        Runtime = O(1)
1. RETURN BEST[n]          Stored at = O(n)
```

Runtime. • Our Algorithm uses dynamic programming which has only one state. So the run time will be linear i.e O(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...i] such that exactly ~~one~~ j different indices are selected.

- Base Case →.
  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]

  DEST[i,1] = max[A[i], BEST[i-1, i]).

- Recursive Formulation →

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

  i = 1 ..... n

  j = 2 .... 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.

- COMPUTE_BEST (A, i, j)

```
1        if BEST[i,j] != -∞
2                return BEST[i,j]
3        if i == 1 & j == 1
4                return A[1]

5        if j == 1
6                return BEST[i,1] = max(A[i], COMPUTE_BEST(A, i-1, 1))

7        if i < j
8                return -∞
```

BEST[i-1, j-1] = COMPUTE_BEST(A, i-1, j-1)
BEST[i-1, j] = " " (A, i-1, j)
BEST[i, j] = max(A[i] + BEST[i-1, j-1], BEST[i-1, j])

return BEST[i,j]

- Invocation call.
  COMPUTE_BEST(A, n, k)

- Required sum Calculation → (Find MaxSum)
  COMPUTE_maxSum(A, n, k)

1. return BEST[n, k]

Since the best array stores the maximum possible value with k picks for n element in BEST[n, k]

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

Problem 7-2.

(a). PRINT_LCS_INDEX(C, X, Y, i, j)

```
if (i == 0 or j == 0)
        return " "
if X[i] == Y[i]
    s = PRINT_LCS_INDEX(C, X, Y, i-1, j-1)
    s = s + X[i]

else if C[i-1, j] >= C[i, j-1]
        s = PRINT_LCS_INDEX(C, X, Y, i-1, j)
else
    S = 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} and 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 $1 \le i \le m$ & $1 \le j \le n$.

(a). LCS_LENGTH (X, Y, m, n).

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

c[0][0] = c[1][0] = c[0][1] = 0.

for i = 1 ton 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[i mod 2][j-1], c[1-i mod 2][j]).

return c[m mod 2]

(c) LCS_LENGTH. (X, Y, m, n)

let c[0.... n] be array.

c[0] = 0.

for i=1 to m

prev = 0

temp = 0.

for j=1 to n.

temp = c[j]

if X[i] == Y[j]

c[j] = prev + 1.

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

prev = temp.

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.

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

  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}

  $$LCS(x,y) = \max(LCS(x^{m-1}, y), LCS(x, y^{n-1}))$$
  if $x[i] \neq y[i]$

- Recursive Formula.

  if $x[i] == y[j]$
  $c[i,j] = c[i+1, j+1] + 1$

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

- LCS-LENGTH $(X; Y\ m, n)$.

  let $c[1 \ldots m+1, 1 \ldots n+1]$ be the new table.
  for $i = 1$ to m
  $c[i, n+1] = 0$

  for $j = 1$ to n
  $c[m+1, j] = 0$.

  for $i = m$ down to 1.

for $j = n$ down to 1.

if $x_i == x_j$.
$c[i,j] = c[i+1, j+1] + 1$.
else.
$c[i,j] = \max(c[i+1,j], c[i, j+1])$
return c.

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

Now for a prefix of x. of length $m/2$ we will always get some.

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

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

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

if $m == 1$ and $x[m] == y[n]$
return 0,1

$\max = -\infty$
$j^* = -1$.
for $j = 0$ to n

if $\max < c_p[j]$.

$\max = c_p[j]$

$j^* = j$

temp, $S1 = L(x_{m/2}, y_{j^*}, m/2, j^*)$
temp, $S2 = L(x^{m/2}, y^{n-j^*}, m/2, n-j^*)$

return j^*; $S1 + S2$.

Runtime $= O(mn)$
space Complexity $\Rightarrow O(1)$ excluding the space for recursion stack

(g) LCS (X, Y, m, n)

if $m == 1$ and $X[n] == Y[n]$
return $X[m]$

$j =$ FIND_J (X, Y, m, n)
str 1 $=$ LCS $(x_{m/2}, y_j, m/2, j)$
str 2 $=$ LCS $(x^{m/2}, y^{n-j}, m/2, n-j)$

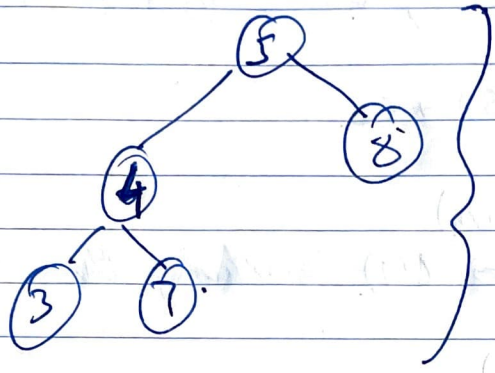
return Str 1 + Str 2.

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

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

Q Problem 7-3.

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



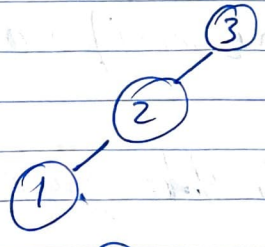
This leads to (5) not checking that (1) is to its right, hence violating BST property.

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.

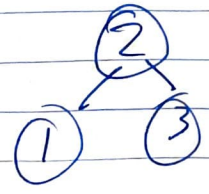
Therefore the algorithm is wrong.

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

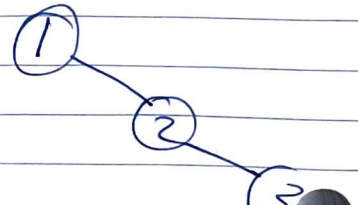
(1)



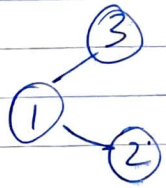
(2)



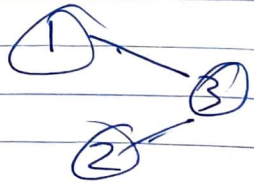
(3)



4.



5.



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

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

Recursive formulation ⇒ (Catalan BST)

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

Base Case. $n = 0$, $F_n = 1$
$n = 1$, $F_n = 1$ (root)

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

(d) Calculate - Fn (n):

1). If. n == 0 or n == 1.
RETURN 1
2). If. M[n] ! = -1.
RETURN M[n]
3) f = 0
4. For i = 1. to n.
f = f +. [Calculate - Fn(i-1). Calculate - Fn.(n-i)]
5. M[n] = f
6. RETURN M[n].

→. Function
Initialize M[1.... n] = -1.
Fn = Calculate - Fn (n)

Runtime O(n).