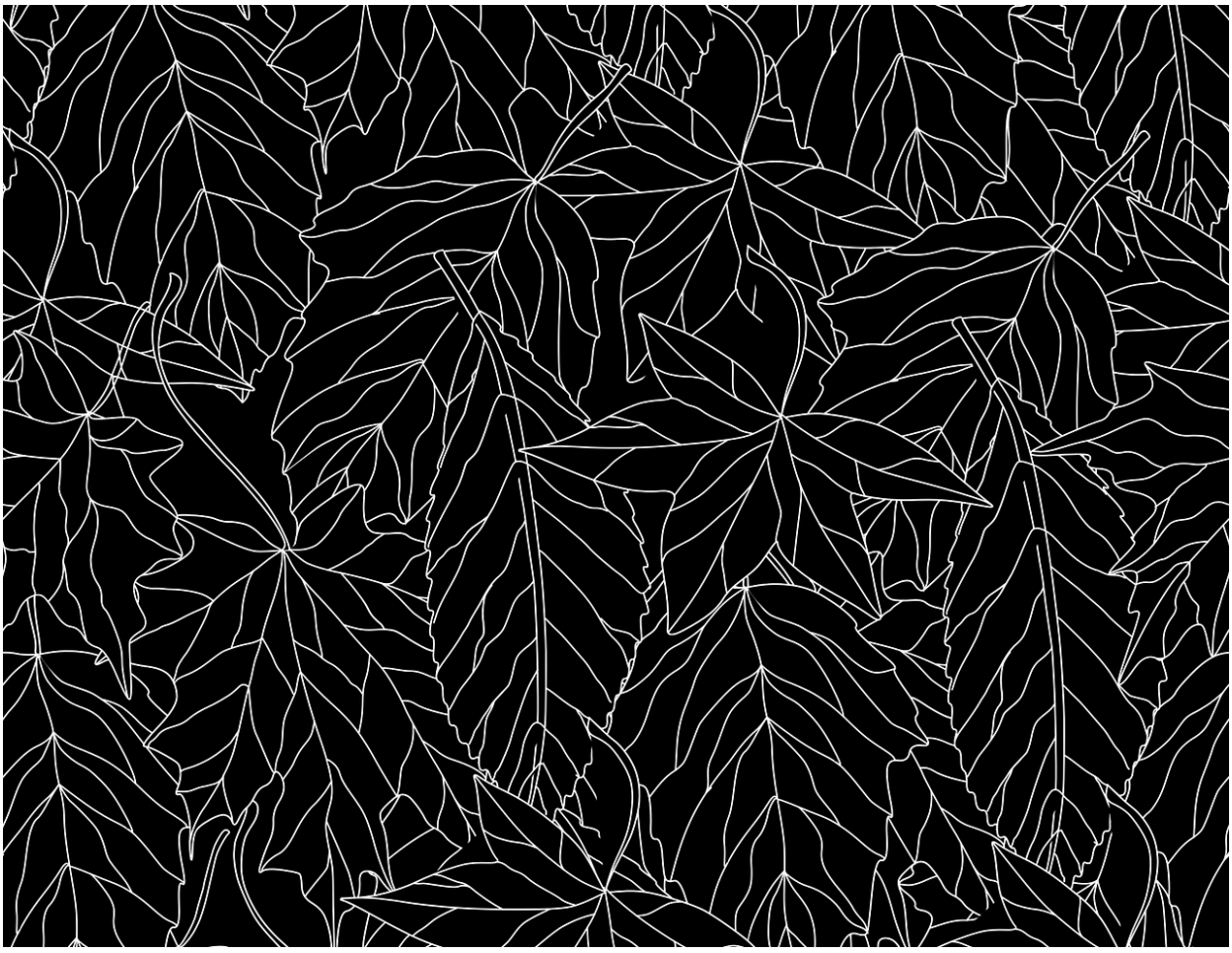
CS-2009, Spring-2024

Design and Analysis of Algorithms Semester Project



Hamza Ahmed 22i-1339 cs-g

PART A

Problem#1

Problem#2

Tauha Imran 22i-1239 cs-g

PART B

Problem#3

Problem#4

CONTENTS

Introduction **2**
Part A - problem#1 ..... **3**
Asymptotic Time complexity : O(n2) 3
Programming language used : C++ 3
Problem Statement 3
Pseudocode 3
Explanation 4
Part A - problem#2 ..... **5**
Asymptotic Time complexity : O(m*n3) 5
Programming language used : C++ 5
Problem Statement 5
Pseudocode 5
Part B - problem#3 ..... **10**
Asymptotic Time complexity : O(n) 10
Programming language used : Python 10
Problem Statement 10
Approach 10
Pseudocode 12
Runtime Analysis 18
Mathematical formula 19
Further Notes 19
Part B - problem#4 ..... **20**
Asymptotic Time complexity : O(m*n) 20
Programming language used : Python 20
Problem Statement 20
Approach 20
Pseudocode 21
Runtime Analysis 24
Recursive formula 24
Further Notes 25

Introduction

The following integrated Project Report Includes the algorithms in pseudocode form designed for the project and asymptotic time complexity analysis of our algorithms. Each group member has contributed to the report related to their allocation of problems.

> Hamza Ahmed (22i1339-g) , has tackled the Part - A

Encompassing problem#1 and problem#2

> Tauha Imran (22i1239-g) , has tackled the Part - B

Encompassing problem#3 and problem#4

You can find detailed explanations and pseudocode for the solutions of the project's problems 1-4 throughout the report.

Part A - problem#1

Max Valid Staircase Problem with N total blocks.

Asymptotic Time complexity : $O(n^2)$

Programming language used : C++

Problem Statement

Find the number maximum valid staircase permutations possible with N blocks available at our disposal where $3 \leq N \leq 200$. A valid permutation would be one where all blocks are used and each stage is smaller than the one preceding it.

Pseudocode

```
function countStructures(n, prevHeight):
      if n is 0:
            return 1
      if memo[n][prevHeight] is not 0:
            return memo[n][prevHeight]
      total_structures = 0
      for h from 1 to prevHeight - 1:
            if n >= h:
                  total_structures += countStructures(n - h, h)
      memo[n][prevHeight] = total_structures
      return total_structures
```

Explanation

· **`Memo`** is an **n*n** 2-dimensional array that stores the solutions to subproblems that have already been solved.

· **`Memo[i][j]`** represents the total number of valid structures possible with `'i'` blocks left and `'j'` set as the height of the previous stage.

· A valid structure is formed when the base case (i.e., **`if n is 0`**) is achieved. This means that no blocks are remaining.

· The main loop performs two functions:

  ○ Checks whether the current stage is smaller than the last stage

  ○ Check whether current stage is made with fewer than or equal to number of blocks remaining.

Part A - problem#2

Minimum Achievable Strategic Value in a railroad with N Depots and M attacks available.

Asymptotic Time complexity : $O(m * n^3)$

Programming language used : C++

Problem Statement

Calculate the permutation of M attacks on N depots that gives us the minimum possible strategic value. Where $M \leq N - 1$ and $1 \leq N \leq 5$.

Pseudocode

Declare a 3D array to implement dynamic programming

```
declare sv as a 3D array of int pointers
```

Define the function to calculate the strategic value between Depot at index 'x' and index 'y'

```
function calculateStrategicValue(list, x, y):
      val = 0
      for i from x to y:
            for j from i + 1 to y:
                  val += list[i] * list[j]
      return val
```

The function to calculate the minimum strategic value for ‘n’ depots and ‘m’ attacks.

```
function trains(m, n, val):
     allocate memory for sv[m+1][n][n]


     for i from 0 to m:
          for j from 0 to n-1:
               for k from 0 to n-1:
                    sv[i][j][k] = 0


     min_val = maximum possible value of unsigned long long


     for x from 0 to m:
          for y from 0 to n-2:
               for z from y+1 to n-1:
                    min_val = INT_MAX
                    if x is 0:
                      min_val = calculateStrategicValue(val,y,z)
                    else:
                    for k from y to z-1:
                         temp_val = min_val
                         min_val = min(min_val, sv[x-1][y][k] +
     sv[0][k+1][z])
```

```
                            if min_val is 0:
                                min_val = temp_val
                        if min_val = INT_MAX:
                            min_val = 0
                        sv[x][y][z] = min_val


    return sv[m][0][n-1]
```

1. Space of Sub-Problems and Optimality Substructure:

- **Parameterization of Sub-Problems:**
  - The sub-problems are parameterized using three indices: **x**, **y**, and **z**, which represent the current attack count, starting depot index, and ending depot index, respectively.
  - The goal is to find the minimum Strategic Value for a given range of depots (**y** to **z**) with a specific number of attacks (**x**).
  - The overall goal is reflected in these sub-problems as we recursively break down the problem into smaller subranges and attack counts.
- **Optimality Substructure:**
  - The optimality substructure condition is met because the optimal solution for a larger problem (more attacks, more depots) depends on optimal solutions to smaller sub-problems (fewer attacks, smaller ranges of depots).
  - The recursive approach considers all possible ways to split the range of depots and attacks, ensuring that the optimal solution is derived from optimal solutions to subproblems.

2. Recursive Formula:

- **Recursive Formula:**
    - The recursive formula is implicitly defined in the code:

```
sv[x][y][z] = min(sv[x-1][y][k] + sv[0][k+1][z]) for all k in [y, z-1]
```

- Base Case: When **x** (attacks) is 0, the Strategic Value is calculated directly for the range **[y, z]** using **calculateStrategicValue** function.

3. Bottom-Up Implementation Order:

- **Order of Solving Sub-Problems:**
    - The code uses a bottom-up approach, solving subproblems in increasing order of the number of attacks **x** and increasing ranges **[y, z]**.
    - The nested loops ensure that smaller subproblems are solved before using their solutions to compute larger sub-problems.

4. Asymptotic Running Time and Space Usage:

- **Running Time:**
    - The time complexity of the overall algorithm is O($m * n$^3), where **m** is the number of attacks and **n** is the number of depots.
    - The triple nested loops iterate through all possible sub-problems.
- **Space Usage:**
    - The space complexity is also O($m * n$^2) due to the 3D array **sv** storing intermediate results for all sub-problems.
    - Additional space may be required for other variables and function calls, but they are typically constant or logarithmic in size relative to **n** and **m**.

Part B - problem#3

Finding two or more divisions of an array , containing the same missing non-negative integer or Minimum Not Presented Number (MNPN).

Asymptotic Time complexity : O(n)

Programming language used : Python

Problem Statement

The task at hand is to manage an array A of length n by dividing it into k segments where k>1. The goal is to ensure that each segment shares the same property: they all have the Minimum Not Presented Number (MNPN). However, if no such division exists, we should report -1.

Approach

The approach for this involves two functions. One for finding the Minimum Not Presented Number (MNPN) and the other for finding the sub arrays. Hence in the pseudocode you find two functions `min_missing_non_negative_integer(..)` and `mnpn-segments(..)` . The Idea of the Question was to figure out that in a given array, of length 'n' ., will it be possible to divide it into 'k' Number of segments , such that k>1 and all segments share the same property of MNPN , that is the all are missing the same non negative integer.

To solve this problem we have utilized some properties of this specific array. First some terms are

MNPN - the minimum number not represented / minimum missing non-negative integer

Valid segment - a segment/division/sub-array that has the same MNPN as the array

They are...

- The MNPN of the segments must be the same as that of the whole array
- If the MNPN = 0 , then the number zero does not exist in the array
    - Hence we can divide the array anyway we like and still have valid segments
    - Just make any two segments
    - display/return their ranges , and then terminate the algorithm
- If the MNPN > 0 the we have to keep in mind that that we need at least 2 zeros (arr[i]=0) in our array to have the possibility of the valid divisions
- Iterate the array and count the number of zeros
- If the number of zeros < 2 , display/return -1 , and terminate the algorithm
- If the number of zeros = 2 , check for if the point partition between the two zeros is not defined by a zero before continuing
    - Check if the two valid segments are one either side of...
        - 2 consecutive values in the array > MNPN
        - 2 consecutive values in the array = (MNPN-1)
        - An independent value in the array is 1 or 2 (minimum +ve case)
    - If found display/return the ranges of these two segments and terminate the algorithm
- If the number of zeros > 2 , continue ahead
- Track the location of the zeros and divide the array into segments according to the location of zeros , and count the number of segments
- Append the formed segments into a list
- iterate through the list and check if they are valid
    - If valid segment - append them in new list of valid segments
    - If Invalid segment - merge it with a valid segment , append the merged segment to the new list of valid segments , and decrease the count of the number of segments by one.
- Now check the count/number of segments

    - If the count of segments <=1 , display/return -1 (no segments can be formed)
    - If the count of segments > 1 , display the ranges of segments from the final list of valid segments made earlier
- Finally terminate the Algorithm

Pseudocode

We have made use of a the data structure - *list.* Here are a few functions for it

list() - makes & returns an empty list - O(1)

pop() - removes the element at the end of a list - O(1)

append() - adds an element to the to the end of a list - O(1)

```
class segment:

//variables...
    public start //start of range
    public end //end of range
    public mnpn //mnpn found in range

    //constructor...
    public segment ( start_x : end_x : mnpn_x )
          start <- start_x
          end <- end_x
          mnpn <- mnpn_x
      return

    //print function...
    public procedure print_range()

      print "(" + start + " -> " + end + ")"

    return

endclass
```

A class/object/structure for a single Segment

First function to find the MNPN

arr - is our array

start - our starting index

```
function min_missing_non_negative_integer( arr , start , end)

   n = arr.length
   frequency = array of size (n+1) initialized with 0


   for i <- (start-1) to end //traversing the array

      do

         if  arr[i] >= 0 and aar[i] <= n

           then frequency[arr[i]] = frequency[arr[i]] + 1

   missing <-0 //variable to track the MNPN

   for i<- 0 to end

      do

         if frequency[i] == 0
           then missing <- i
                return missing //return mnpn

   return missing //return mnpn where it is the value of n
```

end - our ending index

Second function to find segments

arr - our array

n - the length of the array

```
function mnpn_segments(arr,n):

  //input handling
  if( n > arr.length )
     then print "INVALID INPUT | -1"
     return

  full_mnpn <- min_missing_non_negative_integer(arr,1,n)

  Print "-> full_mnpn = " + full_mnpn //for user

  //initialize a an array with -1  values
  set_arr <- declare an array of length (n+1) & all values = -1

  //case 1 - if mnmpn is zero = 0
  if((full_mnpn==0)):
    then zero_dummy <- segment(1,1,0)
         zero_dummy.print_range()
         zero_dummy = segment(2,n,0)
         zero_dummy.print_range()
        return

  //..initializing variables
  segment_start <- 1 //start of segment
  segment_end <- 1 //end of segment
  zero_count <- 0 //to tally the number of zeros


  //loop#1: to compute the subset table in O(n)
  for i <- 0 to n
     do if(arr[i] == 0)or(full_mnpn==0)
          then set_arr[i] <- full_mnpn
               Zero_count <- zero_count + 1 //increment zero count

        else
             set_arr[i]=-1

  //re-initializing variables + new ones

  segment_end <- 1
  segment_start <- 1
```

```
last_segment <- segment(-1,-1,-1)
last_index <- -1 //to tally the final sets...

SEGMENTS <- list() //to store segments in a list

end_set <- True
seg_count <- 0 //to count num of segments



//loop#2: in case of only two zeros… O(n)
 if(zero_count==2)
   then

     for i <- 0 to n
       do
        if ( arr[i]>full_mnpn ) or ( arr[i]==(full_mnpn-1) ) or
           ( arr[i]==1 ) or ( arr[i]==2 )
         then
            lft_mnpn <- min_missing_non_negative_integer(arr,1,i+1)
            rt_mnpn  <- min_missing_non_negative_integer(arr,i+2,n)

        if ( lft_mnpn==rt_mnpn )
          then print "Output"
               Print "segments# 2"
               print  "( " + 1 + " -> " +  (i+1) + " )"
               print  "( " + (i+2) + " -> " + n + " )"
              return
        // there is no else...



//loop#3: to find segments and store them in a list.. O(n)
 for segment_end <- 0 to n+1
   do if(set_arr[segment_end]==full_mnpn)
        then // if a segment found add into set
             seg_count+=1 //count num seg.s

             //make and store an object of the found segment
             last_segment <- segment(segment_start,
```

```
                                        segment_end+1,
                                        set_arr[segment_end])

            //add segment obj to the list
            SEGMENTS.append(last_segment)
            segment_start <- segment_end + 2 //new start value
            segment_end <- segment_start
            last_index  <- segment_end

    //there is no else statement..


//for left over end of array
if( last_index < (n+1) ) and ( seg_count>1 )
  then // form a new segment going till end of array
      new_segment <- segment(last_segment.start,n,full_mnpn)
      SEGMENTS.pop() // remove last added segment
      SEGMENTS.append(new_segment) //add new segment in list


//checking validity of the segments
FINAL_SEGMENTS <- list()
validity <- declare an array of size `seg_count;
            & initializes all values = True
K_lim <- seg_count
True_count <- 0


//loop#4: runs k time with runtime O(n+k)
  for k <- 0 to K_lim
    do sub_segment <- SEGMENTS[k]
      sub_mnpn <- min_missing_non_negative_integer(arr,
                              sub_segment.start , sub_segment.end)

   //if valid true , else false

     if( full_mnpn == sub_mnpn )
       then validity[k]<-True
            true_count+=1
```

```
      else
           validity[k]=False


//check if merging even possible and valid

  if( true_count > 1 )
    then k<-0

     //loop#5: runs k times to make merged solution O(k)..
       while( k < K_lim )
          do  sub_segment <- SEGMENTS[k]

            while (validity[k]!=True) and (k<(K_lim-1))
                 do //getting next segment
                    next_segment <- SEGMENTS[k+1]
                    //updating the end
                    sub_segment.end = next_segment.end

                     k <- k+1

          //add merged segment in final answers list…
         if(validity[k]==True):
            FINAL_SEGMENTS.append(sub_segment)
            K <- 1

  else // for if ( true_count <= 1 )
      seg_count <- 0


//output - renderer…

  print "Output"


  //loop#6: to print values of a set
  if(seg_count>1)
     then print "segments#"  + seg_count

          for range_value <-0 FINAL_SEGMENTS.lenght
             do FINAL_SEGMENTS[range_value].print_range()
```

```
  else
     print "segments# 0"
     print -1


//**end of algorithm**//
```

Runtime Analysis

Here

'mnpn' is the the minimum missing non-negative integer in the array

'n' is the size of the array

'k' is the number of segments made

```
//function min_missing_non_negative_integer( arr , start , end)
```

- First loop iterates between start-1 and end , which run can run the length of the whole array , meaning the total runtime for this is **O(n)**

- The second loop iterates o till (end+1) , leading to a runtime of **O(n+1)**
- Total runtime for the first function is O(n)+O(n+1) **>> O(n) ***

```
//function mnpn_segments(arr,n):
```

- **loop#1:** a simple loop computing the number of zeros found and storing the point where a zero is found in the array in another array. Hence the runtime for this loop is simply **>> O(n)**
- **loop#2:** The loop computes the extraneous cases for when there are only two zeros in the array. With some analysis , we see that we compute the mnpn for both sides of the array a few times , but not always. Hence we run an operation for a constant number of operations, say ***C*** ,for the case , which are very few to

almost no cases. Hence the time complexity for this loop is **O(n + Cn) ,** *where **C** is always a constant* so can tie complexity will be **>> O(n)**

- **loop#3:** This loop simply finds the segments throughout the array with the help of previously computed information and then adds them into a list of segments, hence it has a runtime of **>> O(n)**
- **loop#4:** A loop to run through the segments made and finds their mnpn through iterating the array via function. This nature of the loop leads all operations to sum up to the length of the array plus the number of segments we have . Making the time complexity **>> O(n+k)**
- **loop#5:** Two dependant while loop , that merge segments into valid segments , with a maximum runtime of **>> O(k)**
- **loop#6:** This loop simply prints the final result nad has a runtime of **>> O(k)**

*Summing up all runtimes = 5*O(n)+3O(k) >> O(n)*

Total runtime - O(n)

Mathematical formula

Let the input array be a set **A** of size **n** which can have **K** segments or subsets denoted by a_i where the missing non negative integer is x . Hence we can show the function for the output of the program based on the number of segments **K** as

$$f(k) = \begin{bmatrix} k & 1 < k \leqslant n \\ -1 & k \leqslant 1 \end{bmatrix}$$

Where **K** is the determined by the number of segments formed

Further Notes

The algorithm is Inplace and stable and meets all the required constraints

Part B - problem#4

Most valuable team of maximum ‘m’ members decided according to a few constraints from a 2d-array

Asymptotic Time complexity : O(m*n)

Programming language used : Python

Problem Statement

A total of **‘m*n’** hunters arrive at Laiba's session, forming **‘m’** rows of equal size, each containing exactly **‘n’** individuals. These adventurers are numbered from 1 to n in each row, following the order from left to right. Assemble an unbeatable team, selecting members strategically . The selection process involves choosing adventurers from left to right, ensuring that the column index of each chosen member (excluding the first column) is strictly greater than the column index of the previously chosen hunter. Avoid consecutive selections from the same row. The first hunter can be chosen from any of the **‘m*n’** participants without any additional constraints. The team can accommodate no more than **‘n’** members.

Approach

The solution to this problem follows a bottom up approach algorithm. We start from the last column and go till the first column , iterating through each of the columns top to bottom. During this we know that the method of choosing the values (adventurers) has led to a variety of possibilities from which we are only concerned with the maximum output. So we go right to left for a bottom up approach rather than the left-right iterative approach which would be way too time consuming,

Now, since we are only concerned with the max values, we will traverse a column and find the max value in that column along with its index. We will also store the value that comes second to the max. Next we will traverse to the next column on the left and and traverse it whilst adding out max value separately to each value to find the maximum sum. However if the value is on the same row as the previous max as from (we can tell via a stored index value) , we will add it with the value second to the max. After the column is traversed we will see if our max and value second to max needs to be updated accordingly.

Continue like this until the last column has also been traversed. Then simply return the latest stored maximum as your answer.

Pseudocode

function to print a 2d-array

```
function print_2d_array(arr):

    for i<-0 to arr.length
       do row <- arr[i]
          print row + '\n' //whole row of 2d array + enter
```

arr - a 2-dimensional array

Function to find maximum power possible for a team

n - number of rows

m - number of columns & the maximum size of the team

```
function maximize_power( arr,n,m):

  //to show the user their input
  print "n*m = " + n + "*" + m
  print_2d_array(arr)

  max1 <- //major max for most cases --- true max
  max2 <- //max for one conflicting case -- second max
  index <- -1 // a tally for the index=row num of max1
  sum <- 0 //our answer
  temp <- 0 //a temporary variable



  //input validation
  if (arr.length_of_a_column < n ) or( arr.lenght_of_a_row <m )
      print "INVALID INPUT | -1"
      return



  //case1 -if only one column
  if( n == 1 )
    for col <- 0 to m//top to bottom rows (y-axis-go-down)
        do //for calculating last column maxes
           temp <- arr[0][col]
           //updating & verifying max1,max2 & index
           if ( temp > max1 )
             then max1 <- temp //update second max
    print "Output"
    print max1
    print "------------------"
    return

 //case2 - not a single row

  //loop for the last column…
  for row <- 0 to n //top to bottom rows (y-axis-go-down)
     do //for calculating last column maxes
        temp= arr[row][m-1]
```

```
      //updating & verifying max1,max2 & index
      if (temp > max1 )
         max2  <-  max1 //update second max
         max1  <-  temp //update first max
         index <-  row  //update max1's rown no.

     elseif (temp > max2) and (temp != max1)
         max2  <-  temp



for col <- m-2 downto 0
   do //right to left columns (x-axis-go-left)
      m1=0
      m2=0
      idx=-1

     for row <- 0 to n //top to bottom rows (y-axis-go-down)
         do //for calculating last column maxes
            if ( index==row )
               then temp <- max2 + arr[row][col]

     else
         do temp <- max1 + arr[row][col]

    //updating & verifying max1,max2 & index

    if ( temp > m1 )
         then m2  <-  m1 //update second max
              m1  <-  temp //update first max
              idx <-  row  //update max1's rown no.

    elseif (temp > m2) and (temp != m1)
         then m2  <-  temp

    if ( m1 > max1 ) //new max is valid
       then max2 <- max1
            max1 <- m1
            index <- idx

    if( m2>max2 )//new second max is also valid
      then max2 <- m2
```

```
    if(m2>max2) //only new second max is valid
      then max2 <- m2

sum <- max1 //updating the sum
print "Output"
print sum
print "------------------"
```

Runtime Analysis

There are two major nested loops here

- The first two nested loops in the 2d-array printing function have a simple runtime of **>> O(n*m)**
- The Other function (maximize_power) also has only two nested loops traversing the whole 2d array also giving it a runtime of **>> O(n*m**)

Both nested-loops have the same runtime hence..

Total runtime - O(n*m)

Recursive formula

The recursive formula follows a 5 value function where

x - new input value

i - index of value of x

I - index of value of m1

m1 - our pre-existing value of max

m2 - our pre-existing value of second to that of max

$$f\left(x,i,I,m1,m2\right)=\begin{bmatrix}x+m1 & i\neq I\\ x+m2 & i=I\end{bmatrix}$$

The values of our maxes can be determined like

$$m1_i=\begin{bmatrix}m1i & m1i\geqslant m1_{i-1}\\ m1_{i-1} & m1i<m1_{i-1}\end{bmatrix}$$

$$m2_i=\begin{bmatrix}m2i & m2i\geqslant m2_{i-1},m1_i>m2i\\ m2_{i-1} & m2i<m2_{i-1},m1_i>m2i-1\end{bmatrix}$$

Further Notes

The algorithm is Inplace and stable and meets all the required constraints