Algorithms Assignment- 3:

**2 a]** We can use dynamic programming to compute the solution to this problem. But we need to break the solution in to smaller subparts i.e. to consider the weekly weigh-ins of each participant and apply dynamic programming to compute the longest losing streak for each participant. Further we can get the final result by finding the maximum value out of the results computed for each participant.

Let w[i][j] be a 2Dimensional array where each row represents the weights of each participant and column represents the week.

This problem exhibits optimal substructure property. W= [$w_1$, $w_2$, $w_3$,...$w_n$] be the weights of a participant for weeks 1 through n. Let's say $w_i$ is the weight which is at the end of the longest losing streak sequence.

Let LWLS(i) represent the sequence with $w_i$ as the last element.

LWLS(i) consists of either:
The element $w_i$ with just one element is a decreasing sequences since it has just one element.
The element $w_i$ appended to the end of the an decreasing sequence that ends with $w_j$, where weight $w_j > w_i$.

**2 b]** In this problem we need to find the longest losing streak for each participant and keep track of the weeks which resulted in the longest losing streak.

So we need to keep track of the context of the previous iteration to get the outcome of present Iteration. In-order to achieve this we use an one-dimensional array "prevIndx "which keeps track of the last week "i" which is part of the longest losing streak consisting for week j where 1<=i<=j

Apart from this we need to keep track of the last week index of the longest losing streak to know the start index for back traversal to get all the weeks that are part of longest losing streak. So we use an one dimensional array endIndx[0..m], to keep track of endInx of longest losing streak for each participant

As part of result, since we need to return the length of the longest losing streak. so we use a one dimensional array len[0..m] which hold the length of the longest losing streak for each participant. Which is further compared to get the participant with longest losing streak.

Finally an one dimensional array weekSeq[0..n] to hold the weeks which resulted in longest losing streak. This is done by backtraversing starting from the last week index of the longest losing streak till we reach -1(initially the array weekSeq is initialised to -1)

**2C]**
**Given:** m is the number of participants and n is the no of weeks, w is a 2Dimensional array where each row represents the weekly weigh-ins of each participant and each column represents week, m which represents the number of participants and n which represents the number of weeks and an array weekSeq to keep track of the
Weeks in which losing streak occurred.

**Returns :** the index of the participiant who has the longest weight losing streak and an array weekSeq which
consists
<u>FetchLongSeq</u>(w, m, n, weekSeq)

```
Let len[0..m] be a new array
Let res[0..n]  be a new array  initialized to 0
Let prevIndx[0..m][0..n] be a new array initialized to -1
Let endIndx[0...m] be a new array
Let weekSeq[0...n] be a new array
For k = 0 to m
        len[k] =1
        res[0] = 1
        for j = 1 to n
             for i = 0 to j
                 If    w[k][j] < w[k][i] && res[i]+1 > res[j]
                          res[j] = res[i] +1
                          prevIndx[k][j] = i
             If    res[j] > len[k]
                   endIndx[k] = j
                    len[k] = res[j]



maxRes = -∞
resIndx = -1
for  l = 0 to m
        if  maxRes < len[l]
                maxRes = len[l]
                resIndx = l


set all indices of weekSed[] array to -1

start= endIndx[resIndx]
k = 0
while   start > 0
        weekSeq[k] = start
        start = prevIndx[resIndx][start]
        k++
return resIndx, weekSeq
```

Time Complexity = $O(mn^2)$

**Question 1:**

**1 a>** Let's consider $P=\{P_1, P_2, P_3, P_4 \ldots\ldots P_n\}$ to be the set of points.
Where each $P_i$ is of the from $(X_i, Y_i)$ for all $1<= i<=n$

Let $I = \{ i_1, i_2, i_3, i_4, ..i_m\}$ where $1 <= m < n$, be the set of points which resulted in optimal partition

We need to partition Points P into segments which consists of contiguous points $P_i \ldots P_j$. For each of these segments we need to compute the error e(i,j) and then find the partition which results in minimum penalty

The penalty is the sum of the following:
1. The penalty associated with each segment in the partition, a constant C.
2. For each segment, the error value of line fit on the points in that segment

**Optimal sub-structure**

Let Opt(i) be the optimum solution for the points $P_1, P_2, \ldots P_i$ and e(i,j) denote the minimum error for the line segment of the form $P_i \ldots P_j$.

In an optimal solution if the last point $P_n$ is part of the segment which starts at $P_i$. We assume that segment is part of the solution and try to solve this recursively for all the points from $P_1$ to $P_{i-1}$. Segment $P_i \ldots P_j$ is considered only if it is minimum
Therefore the equation for optimal solution can be represented as :

| | |
|---|---|
| when n ==0 | Opt(n) = 0 |
| else | $Opt(n) = \min_{1<=i<=j}\{ error(i, j) + C+ Opt(i-1)\}$ |

**1 b>**
Given : 2Dimension array points which holds x-co-ordinate of points in first column and

y-co-ordinate of points in second column , a constant which is added to the penalty and n which represents the total number of points, a 2D array error which keeps track of error to fit the line between the given two points and segIndx is a pointer to the array which keeps track of indices which result in optimal solution

Returns : optimum partition penalty value opt[n] and an array consisting of the index which resulted in optimal partition for the segment Point $P_1$ from $P_i$ for 1<=i<=n

```
RECURSIVE_FETCH_SEGMENT(point, n, c, error, segIndx, opt)
        If n = = 0
             Set opt[n] to 0
        else if opt[n] ! = -1
              set min to ∞
              set x to -1
              for i = 1 to n
                 val= error[i][n] + c + RECURSIVE_FETCH_SE (a, i-1, c, error, segInx, opt)
                 if val < min
                         min = val
                         x = i
                 opt[n] = min
                 segIndx[n] = x
        return opt[n]
```

Given : 2Dimension array “point” which holds x-co-ordinate value in first column and y-co-ordinate value in second column and the constant “c” which is added to the penalty and “n” which represents the total number of points.

Returns : the optimum partition array consisting of indices of points which form the optimum partition i.e. optIndx and the penalty of the optimum partitioni.e. opt[n]

FETCH_SEGMENT(point, c, n)

sigmaX[1..n] represents the summation of x-coordinates
sigmaY[1..n] represents the summation of y-coordinates
sigmaXY[1..n] represents the summation of product of x-coordinates and y-coordinates
sigmaXsqr[1..n] represents the summation of (x-coordinate)$^2$
slope[1..n][1..n] where slope[i][j] represents the slope of the line drawn from point $P_i$ to $P_j$
yIntercept[1..n][1..n] where yIntercept[i][j] represents the y-intercept for the line drawn from point $P_i$ to $P_j$

error[1..n][1..n] where error[i][j] represents the error to fit the line between the points $P_i$ and $P_j$

Set sigmaX[0] , sigmaY[0] , sigmaXY[0] , sigmaXsqr[0] to 0
**for** j = 1 to n

```
        sigmaX[j] = sigmaX[j-1] + point[j][0]
        sigmaY[j] = sigmaY[j-1] + point[j][1]
        sigmaXsqr[j] = sigmaXsqr[j-1] + (point[j][0] * point[j][0])
        sigmaXY[j] = sigmaXY[j-1] + (point[j][0] * point[j][1])
        for   i = 1 to j

            sigX = sigmaX[j] - sigmaX[i-1]
            sigY = sigmaY[j] - sigmaY[i-1]
            sigXsqr = sigmaXsqr[j] - sigmaXsqr[i-1]
            sigXY = sigmaXY[j] - sigmaXY[i-1]

            Let n  be the interval  between i and j


            // computing the slope for the segment drawn between
            // points P_i  and P_j
            if   ((n * sigXY) - (sigX * sigY)) = = 0
                set slope[i][j]  to 0
            else if    ((n * sigXsqr) – (sigX * sigX)) == 0
                set slope[i][j]  to  ∞
            else
                slope[i][j] = ((n * sigXY) - (sigX * sigY))/ ((n * sigXsqr) –
                                    (sigX * sigX))
            // computing the y-intercept for the segment drawn between
            // points P_i  and P_j
            if  n = = 0
                set yIntercept[i][j]  to ∞
            else
                yIntercept[i][j] = (sigY - ( slope[i][j] * sigX))/ n

            // computing  cumulative error for all points ranging from i to j
            Set e[i][j] to 0
            for  r = i to j
                error[i][j] = error[i][j] +
                      ((slope[i][j]  * point[r][0]) +   yIntercept[i][j] -
                            point[r][1])
                      * (( slope[i][j]  * point[r][0]) +  yIntercept[i][j]
                            - point[r][1])

        opt [1..n]   an array to store the penality of optimum partition
                     where opt(i)  represents penality of partitions of P_1 to P_i

        set opt[1..n] to -1

        segInx[1..n] an array to keep partition for point P_i  which led to the optimal
                     partition of points P_1  to P_i


        opt[n] =RECURSIVE_FETCH_SEGMENT(point, n, c, error, segIndx, opt)
```

```
        optIndx[1..n ] initialized to -1  contains only to indices which led to the
                    optimal partition of points P_1 to P_n
        set idx  to 0

        for q = n ; q >= 0 ; q =segIndx[q]
                optIndx[idx] = q
                idx++


        return (opt[n], optIndx)
```

**1 c>**

Given : 2Dimension array “point” which holds x-co-ordinate value in first column and y-co-ordinate value in second column and the constant “c” which is added to the penalty and “n” which represents the total number of points.

Returns :  the optimum partition array consisting of indices of points which form the optimum partition i.e. optIndx and the penalty of the optimum partitioni.e. opt[n]

FETCH_SEGMENT(point, c, n)

```
sigmaX[1..n]        represents the summation of x-coordinates
sigmaY[1..n]        represents the summation of y-coordinates
sigmaXY[1..n]       represents the summation of  product of x-coordinates and
                    y-coordinates
sigmaXsqr[1..n]   represents the summation of (x-coordinate)^2
slope[1..n][1..n]   where slope[i][j] represents the slope of the line drawn from point P_i
                        to P_j
yIntercept[1..n][1..n]  where yIntercept[i][j] represents the y-intercept for the line
                    drawn from point P_i  to P_j

error[1..n][1..n] where error[i][j] represents the error to fit the line between the points
            P_i  and P_j


Set sigmaX[0] , sigmaY[0] , sigmaXY[0] , sigmaXsqr[0] to 0
for j = 1 to n

        sigmaX[j] = sigmaX[j-1] + point[j][0]
        sigmaY[j] = sigmaY[j-1] + point[j][1]
        sigmaXsqr[j] = sigmaXsqr[j-1] + (point[j][0] * point[j][0])
        sigmaXY[j] = sigmaXY[j-1] + (point[j][0] * point[j][1])
        for   i = 1 to j

                sigX = sigmaX[j] - sigmaX[i-1]
                sigY = sigmaY[j] - sigmaY[i-1]
                sigXsqr = sigmaXsqr[j] - sigmaXsqr[i-1]
                sigXY = sigmaXY[j] - sigmaXY[i-1]
```

```
        Let n  be the interval  (j-i+1)

        // computing the slope for the line segment drawn between
        // points P_i  and P_j
        if   ((n * sigXY) - (sigX * sigY)) = = 0
                set slope[i][j]  to 0
        else if    ((n * sigXsqr) – (sigX * sigX)) == 0
                set slope[i][j]  to  ∞
        else
                slope[i][j] = ((n * sigXY) - (sigX * sigY))/ ((n * sigXsqr) –
                                            (sigX * sigX))

        // computing the y-intercept for the line segment drawn between
        // points P_i  and P_j
        if  n = = 0
                set yIntercept[i][j]  to ∞
        else
                yIntercept[i][j] = (sigY - ( slope[i][j] * sigX))/ n


        // computing  cumulative error for all points ranging from i to j
        Set e[i][j] to 0
        for  r  = i to j
              error[i][j] = error[i][j] +
                         ((slope[i][j]  * point[r][0]) +   yIntercept[i][j] -
                                        point[r][1])
                          * (( slope[i][j]  * point[r][0]) +  yIntercept[i][j]
                                        - point[r][1])

opt [1..n]   an array to store the penality of optimum partition
             where opt(i)  represents penality of partitions of P_1 to P_i


segInx[1..n] an array to keep partition for point P_i  which led to the optimal
              partition of points P_1  to P_i


Set opt[0] =0
for  j =   1 to n
        opt [j] = ∞
        indx = -1
        for i = 1 to j
                res = error[i][j]  + opt(i-1) + c
                if   res <  opt[j]
                        opt[j] = res
                        indx  = i
        opt[j] = res
        segIndx[j] = indx
```

```
48          optIndx[1..n ] initialized to -1  contains only to indices which led to the
                      optimal partition
49          idx =0

50          for q = n ; q >= 0 ; q =segIndx[q]
51               optIndx[idx] = q
52               idx++

53          return (opt[n], optIndx)
```

**Time Complexity and space complexity**

**1 c> time complexity of bottom up approach**

Let T(n) be the time complexity for the recursive approach. The computation of the error(i, j) involves three for loops at line number 11, 16 and 32 in FETCH_SEGMENT. All the steps from 11 to 32 get executed maximum of "n" times i.e. when j= n , i loops runs for n times and when j = n and i= 0, r loop runs n times , where n is the total number of points given.

Therefore in the worst case run time for calculating error(i, j)

$$= \sum_{j=1}^{n} \sum_{i=0}^{j} \sum_{r=i}^{r} c = n * n * n* c = n^3 * c1$$

Next, the calculation of minimum penalty partition involves two for loops which runs for a maximum of n time i.e. when i = n, j loop runs for a maximum of n times . So the worst case total run time for all the steps from 38 to 47 is

$$= \sum_{j=1}^{n} \sum_{i=0}^{j} c = n*n * c = n^2 * c2$$

Max run time for steps 50 to 52= n *c3 which involves looping through the segIndx array of max n times to fetch the indices which led in optimal partition

All the remaining steps take constant time . Let's consider that to be C

Therefore total time complexity = $(n^3 * c1) + (n^2 * c2) + (n * c3) + C$

- ⇨ Time complexity, $T(n) = O(n^3)$

**1 c> Time complexity for recursive approach**

Let T(n) be the time complexity for the recursive approach. The computation of the error(i, j) involves three for loops at line number 11, 16 and 32 in FETCH_SEGMENT. All the steps from 11 to 32 get executed maximum of "n" times i.e. when j= n , i loops runs

for n times and when j = n and i= 0, r loop runs n times , where n is the total number of points given.
Therefore in the worst case run time for calculating error(i, j)

$$= \sum_{j=1}^{n} \sum_{i=0}^{j} \sum_{r=i}^{r} c = n * n * n * c = n^3 * c$$

RECURSIVE_FETCH_SEGMENT involves a for loop i = 1,2,3 …. n and a recursive call j = i-1 ,i-2,i-3,…..,0 where 1 <= i <= n.

So, maximum number of times j can run is (n-1) when i = n. We can represent the recursion execution as follows:

$$= \sum_{i=1}^{n} \sum_{j=0}^{i-1} c = n^2 * c$$

max run time for steps 40 to 42 = n *c3 , which involves looping through the segIndx array of max size n to fetch the indices which result in optimal partition

All the remaining steps take constant time . Let's consider that to be C

$T(n) = c * n^2 + c1 * n^3 + n * c3 + C$

- ⇨ $T(n) = O(n^3)$

**1 c> Space Complexity of recursive solution and bottom up approach**

n -> is the number of points

In both **recursive** and **bottom up approach,** in-order to keep track of sigmaX, sigmaXY, sigmaY and sigmaXsqr, opt, optIndx and segIndx we use one dimensional array of size n to keep track of cumulative sum for all point Pi .

This takes a maximum space "n" to keep track of summation of x, y, xy ,$x^2$ optimum partition penalty value, indices which led to optimal path for each segment and an array to keep track of minimum penalty partition respectively.

To hold slope and y-intercept values for each combination of segment $P_i$ .. $P_j$ for all
1 <= i <= n and 1<= j <= m for each segment, we use 2Dimensional array of size n*n

Total space complexity = $7*n + 2*n^2$ => space complexity = $O(n^2)$