Q.1.

ASSIGNMENT-2

```
Algorithm SelectKthRanked(A[1...n], B[1...m], K)
{
    if A.length = 0
        return B[K]

    if B.length = 0
        return A[K]

    mid1 = ⌈n/2⌉
    mid2 = ⌈m/2⌉

    if (mid + mid2 <= K)
        if (A[mid1] > A[mid2])
            return SelectKthRanked(A[1...n], B[mid2+1...m],
                                   K-mid2)
        else
            return SelectKthRanked(A[mid1+1...n], B[1...m],
                                   K-mid1)

    else
        if (A[mid1] > A[mid2])
            return SelectKthRanked(A[1...mid1-1],
                                   B[1...m], K)
        else
            return SelectKthRanked(A[1...n],
                                   B[1...mid2-1],
                                   K)
}
```

Explanation: (Note: 'K' refers to the number of elements we need at any point to reach the Kth element.

1) We first define the base cases:
   i) if A empty, return B[K]
   ii) if B empty, return A[K]

Q27. At each call, we compute mid1 and mid2.

If mid1+mid2 <= K:

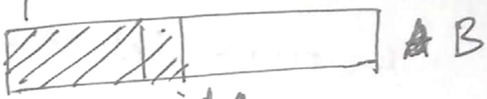
Here, there is a possibility that the elements A[1..mid1] and B[1...mid2] might all be in the first K elements ~~of~~ ~~A∪B~~.

Here, there are 2 cases:

Case-1: A[mid1] > B[mid2]

In this case there might be elements to the right of mid2 which are lesser than A[mid1]. Also, there might be elements to the left of mid1 which might be greater than B[mid2]

But the portion of elements - B[1...mid2]



must definitely be present in the first K elements ~~of~~ ~~the~~ A∪B.

So we recurse only on:

B[mid2+1...m] and A[1...n].

Also, K = K - mid2 (since B[1...mid2] are surely in the first K elements)

Case-2: B[mid2] > A[mid1]

Similar to the above explanation except than in this case, we recurse on:

A[mid1+1...m]

K = K - mid1 (since A[1...mid1] elements have been known to surely be in the first k elements)

If mid1 + mid2 > K:

Here, all the elements A[1...mid1] and B[1...mid2] cannot be in the first K elements we need.

Case-3: A[mid1] > B[mid2]

Since only K elements are required, A[mid1...n] cannot be in the first K elements we need.

So we recurse only on:

A[1... mid1-1] and B[1...m]

K remains the same since we still need K elements.

Case-4: A[mid1] < B[mid2]

Similarly to the above case, except that here we only recurse on:

A[1... n] and B[1.. mid2-1]

K remains the same since we still need K elements.

Running time:

1) The base cases take O(1) time.

2) Computing mid1 and mid2 takes O(1) time.

3) In the worst case we recurse until both A and B contain only one element each.

4) Our array sizes of A and B are reduced

by half in every alternate call (in the worst case)
until both have one element each).

$\therefore T(n) = O(\log(n) + \log(m))$

2. Cascading Waterfall.

input: Set of n points in a plane (P)

output: Set of non-dominated points in P i.e. S

Divide-and Conquer Approach:

```
Algorithm Cascade(P[1...n])

    if n=1 return P
    if n=2 return max_{x,y}(P[1], P[2])
    m = find-MedianX(P[1...n])
    Let L,R be 2 sets
    for i=1 to n
        if P[i].x < n
            put P[i] in set L
        else
            put P[i] in set R
    set S1 = Cascade(L)
    Set S2 = Cascade(R)
        Ymin = ∞
        for k=1 to S2.size()
            if S2[k].y < Ymin
                Ymin = S1[k].y

        for k=1 to S1.size()
            if S1[k].y < Ymin
                remove S1[k] from S1
        return S1 U S2
```

Explanation:

1) The base cases are trivial.

2) We find the median of the points in the current set w.r.t. x-coordinates

of the points which takes O(n) time.

3) We put the points with x-coordinate less than Xmedian (or m) in set L and the remaining in set R. This is done by traversing through the array in O(n) time.

4). We recursively call the function on L and R, getting sets S_1 and S_2 respectively of the non-dominated points in L and R.

5). We observe that all the non-dominated points of S_2 must be in our resulting list.

6) However, any point in S_1 with y-coordinate less than the minimum y-coordinate in S_2 cannot be in the resulting list. We eliminate all such points.

7) Find y_{min} takes O(n) time and eliminating points in S_2 also takes O(n) time.

8). We return $S_1 \cup S_2$.

<u>Running Time</u>:

1) The base case takes O(1) time.

2). Median find takes O(n) time.

3). Putting points into L and R takes O(n) time

3) Finding Y_{min} takes $O(n)$ time.

4) Eliminating points in S_1 takes $O(n)$ time.

All operations other than recursive calls take $O(n)$ time.

$$T(n) = T(n/2) + O(n) + O(n) + O(n) + O(n)$$

$$= T(n/2) + O(n)$$

$$= O(n\log n)$$

Note: Each operation other than a recursive call takes $O(n)$ time for all recursive calls at a level. i.e. $O(n)$ time is needed for an operation to be executed in all recursive calls at any level.

Q.3. We need to find the maximum length of a convex subsequence $x_1 \ldots x_m$ s.t. for all $1<i<m$

$$x_{i-1} + x_{i+1} > 2x_i \quad \text{—— (1).}$$

Recursive Algorithm:

```
Algorithm Convex (A[1...n])
{
   if n<=2                          // Base case
      return n
   result = ConvexRec(1,2)
   if(result != 2)
         return result
   else
         return 0
}
```

Explanation:

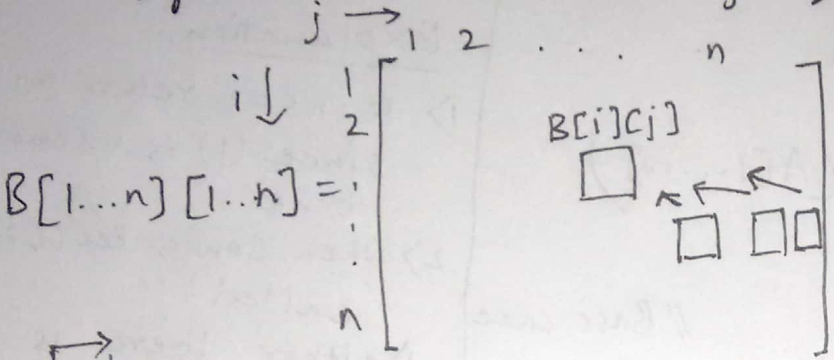
1) if $n<=2$ return n. Since (1) is vacuously true.
2) when ConvexRec(1,2) is called:
   1) either there is a longest convex subsequence beginning at A[1], or
   2) not
   We use a for loop to iterate over values of K where $j+1 \le K \le n$.
3) If the value 2 is returned there is no convex subsequence.

```
ConvexRec(i,j)
{
   best = 2

     for K = j+1 to n
         if ( A[i] + A[K] > 2*A[j] )
             best = max{ best, 1+ ConvexRec(j,K) }

     return best
}
```

Note: Here i and j are the first 2 indices of the array portion we are recursing on. i.e. A[i,j, ... n]

Dynamic Programming solution :

We can use a table of size $n \times n$ to store the values. The parameter i is represented by the rows and 'j' by the columns.




```
best = 2
if (condition satisfied)
    best = max{best, ConvexRec(j, k)}
return best
```

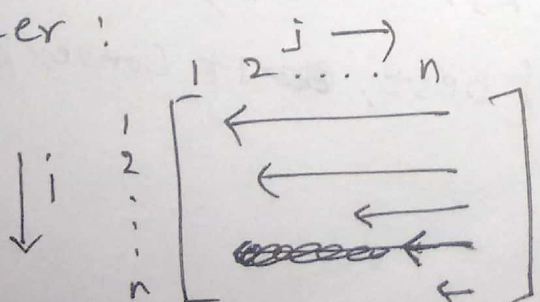
Here,

$i < j < k$

$j < k \leq n$.

To compute $B[i][j]$, we need all values of $B[j][k]$, where $j < k \leq n$

The table must be filled in the following order :



(left to right, bottom row to top row).

The last row is empty since $B[n][j]$ cannot exist $\forall j$.

Q.3) The logic here is that whenever B[j][K] is called then the condition:

$$A[i] + A[K] > 2A[j] \text{ valid} \quad \text{——— (1)}.$$

must be true, i.e. there are indices i, j, k such that the above condition is true.

So the length of the subsequence convex(i,j) or B[i][j] should be atleast 3 ~~if we stay~~ or more.

That is why the base cases (last column entries are set to 2).

B[i][j] is also set to 2 inside the second 'for' loop before iterating over K.

4) We need the value of B[1][2] to be returned by the first recursive call.

if B[1][2] = 2, it means there is no $j \le K \le n$ for which condition (1) is true.

Hence, we return 0 in that case.

Running time:

1) There are approximately $O(n^2)$ table entries.

2) Each table entry takes $O(n)$ time to compute since we must iterate over values of K.

3) Other operations and base cases are $O(1)$ time

$$T(n) = O(n^2 \cdot n) = O(n^3).$$

Algorithm ConvexDP(A[1..n])
{
if (n<=2) return n
Initialise B[1...n][1..n]

for i = 1 to n − 1
  B[i][n] = 2

for i = n−2 down to 1
  for j = n−1 down to i+1
    B[i][j] = 2
    for k = j+1 to n
      if (A[i] + A[k] > 2A[j])
        B[i][j] = max{ B[i][j], 1 + ~~ConvexDP(A[~~ B[j][k] }

if (B[1][2] != 2)
  return B[1][2]
else
  return 0

}

Explanation:

1) Last column entries are set to '2' (base cases, helps in length calculation).

2). Three 'for' loops

i) One for 'i' = n−2 to 1

ii) One for 'j' = n−1 to i

iii) We set B[i][j] = 2 and iterate over values of k.

Q.4. MAXIMUM SUBARRAY PRODUCT :

```
Algorithm Max_Sub_Product (A[1...n])
{
  if n=0
      return 0
  if n=1
      return A[1]

  ans = A[1]
  cur_max = A[1]
  cur_min = A[1]

  for i=2 to n
      prev= cur_max
      cur_max= max { cur_max * A[i] , A[i]
                     cur_min* A[i] }
      cur_min = min { prev * A[i] , A[i]
                      cur_min * A[i] }

      ans= max {ans, cur_max }

  return ans
}
```

Explanation :

1) The base cases handle the trivial scenarios.

2> The variables cur-max, cur-min and ans are set equal to A[1] initially.

3> for i = 2 to n.

cur_max is set to the maximum of 3 things

A[i], cur_max * A[i], cur_min * A[i].

i) cur-max stores the maximum product in the currently running subarray.

ii) cur-min stores minimum product is useful if A[i] is negative to multiply with (since (-) × (-) = +)

iii) if A[i] = 0, both cur-min and cur-max are set to 0, which means that the max subarray cannot end at 0. So, the max subarray product has already been calculated or we start again.

cur-min is set to the minimum of 3 things:

cur_max * A[i], cur_min * A[i], A[i]

4> At each iteration, the variable ans is updated.

ans = max(ans, cur_max).

5>. We return 'ans' when the control exits the for loop.

Running Time:

1> Base cases are O(1) time.

Q.2.> Alternate approach (Non divide-and-conquer)

Algorithm Cascade (P[1...n])

Merge-sort-Y (P)

xmax = -∞

for K = n down to 1

if P[K].x >= xmax

add P[K] to set S.

x max = P[K].x

return S.

Explanation:

1> We first sort the points based on their y-coordinates (in increasing order).

2> Starting from the last point in the sorted array (point with highest y-coordinate value), we add a point to our set S if P[i].x >= xmax. i.e. we are now finding points in a non-decreasing order of x coordinates.

Running Time:

1> Merge-sort_Y(P) takes O(n logn) time

2> The for loop takes O(n) time. Operations inside the for loop take O(1) time.

$$T(n) = O(n) + O(n\log n)$$

$$T(n) = O(n\log n)$$