Part B

Q1.) Chapter 5, Exercise 6: Significant Inversion Problem

Ans.) Let there be a recursive divide-and-conquer algorithm, A1 which takes a randomized set of numbers, $a_1 \ldots a_n$, within a defined range and returns the number of significant inversions, N to get the sequence sorted. This is a modified version of Mergesort.

A1 will in principle be similar to an algorithm that simply counts the number of inversions. However, we'll be merging the sub-arrays twice. First, we'll merge $a_1,\ldots,a_k$ with $a_{k+1},\ldots,a_n$ for the sorting and then we'll merge, as per the definition of significant inversions, $2a_{k+1},\ldots, 2a_n$. for counting the significant inversions.

For, n = 1 -> A1 will simply return N = 0 significant inversions.

Pseudocode ->

```
def mergesort(arr, temp, left, right):
    inv_count = 0
    # mid = 0
    if right > left:
        mid = int((right + left) / 2)  # Wrap the expression to avoid a "Recursion Error"
        # inv_count = mergesort(arr, temp, left, right)
        # Start by checking the inversion count of the left part
        inv_count = mergesort(arr, temp, left, mid)
        # inv_count = inv_count + mergesort(arr, temp, mid + 1, right)
        # Count the inversions on the right part
        inv_count += mergesort(arr, temp, mid + 1, right)
        # Merge the above two inversion counts
        inv_count += merge(arr, temp, left, mid + 1, right)
    return inv_count
```

The above function divides the input array into sub-arrays and then recursively runs to check the left sub-array to obtain N_1 and the right sub-array to obtain N_2. It then runs the "merge" function to combine the two sub-arrays and if there are any inversions still remaining then N_3 is added to get the total number of inversions.

In short,

Inv_count (from the above function) = $N_1 + N_2 + N_3$

The "merge" can happen in O(n) time while the division of the array occurs in O(n log n) time. So, the total time complexity of the algorithm is O(n log n).

The definition of "merge" is given below:

```
def merge(arr, temp, left, mid, right): # Temp array is used to store the elements post comparison and prior to merging
    inv_count = 0

    i = left
    j = mid
    # k = left

    # First pass to count the number of significant inversions
    while (i <= mid - 1) and (j <= right):
        if arr[i] > 2 * arr[j]: # This is required as per the question stem to be the definition
            #  of significant inversions
            inv_count += (mid - i)  # If the above condition holds check the left side of the ith element
            j = j + 1
        else:
            i = i + 1

    # Re-activate the indices to include the index of the resultant sub-array
    # Also, needs to be done to prevent infinite looping which will result in a StackOverflow
    # resulting in a "RecursionError"
    i = left
    j = mid
```

```
k = left

# Second pass happens like a simple inversion count
while (i <= mid - 1) and (j <= right):
    if arr[i] <= arr[j]:
        temp[k] = arr[i]
        i = i + 1
        k = k + 1
    else:
        temp[k] = arr[j]
        k = k + 1
        j = j + 1
# The pointers will not be moving simultaneously unlike a simple inversion count
# Also, once we have determined that the condition holds true both the pointers need to
# move forward separately unlike normal inversion count

# Check for remaining elements in both the right and left sub-arrays to be copied to temp
# For Left Sub-Array
while i <= mid - 1:
    temp[k] = arr[i]
    k = k + 1
    i = i + 1
# For Right Sub-Array
while j <= right:
    temp[k] = arr[j]
    k = k + 1
    j = j + 1

# Copy the merged elements back to the original array
```

```
    for i in range(left, right + 1):
        arr[i] = temp[i]


    return inv_count
```

Q2.) Chapter 5, Exercise 6: Local Minimum Problem

Ans.) **Given**: A complete binary tree -> T, $n = 2^d - 1$, where n is the number of nodes. Each node, v of the tree is labeled with a real number.

For a node to be local minimum -> $x_v < x_w$

Let there be a node 'w' which is smaller than the node 'v' i.e. $x_v > x_w$. There exists a set S where 'w' has a smaller value than any of the nodes in S. Let 'r' be the root of the tree. The algorithm designed will terminate if one of the two conditions is met ->

1.) There exists a node, 'v' which is smaller than its child nodes.
2.) We reach 'w' which would be the local minimum

Either of the conditions will hold true once a successive iteration is made. This way, we also ensure that the algorithm terminates.

Pseudocode ->

```
Def tree(r, v, w):
        x = integer.min
    If (r < v & r < w):  //Check to see if root node is smaller than its children
        r = x
        print("Root is the local minimum)
        break
  ElseIf (r > v & r > w):
          v1, v2
          If(v1 > v & v2 > v):
                  v = x
                  print("v is the local minimum)
                  break
          Else:
```

Iterate for w

Print("w is the local minimum)

Break

Since, $n = 2^d - 1$, the time complexity will be O(d * log n) as it'll take O(log n) probes of the tree to identify the local minimum.

The algorithm will definitely terminate because –

1.) If case 1 is true, then 'v' is the local minimum as it is smaller than its parent and child nodes.
2.) If case 2 is true, then 'w' is the local minimum as it is smaller than its parent and child nodes.

The above two situations are true as all the nodes will be taken up in one of the probe iterations.

Part A

Q1.) Chapter 5, Solved Exercise 1

Ans.) **Pseudocode** ->

```
Def peakentry(arr,n, p):
    For i in range(n):
        i = i + 1
        If(p < arr[n/2] & p < arr[(n/2) + 1]):
            arr[i] = p
            print("Peak is present at: ", arr[i] = p)
            break
        ElseIf(p > arr[(n/2) – 1] & p > arr[n/2]):
            arr[i] = p
            print("Peak is present at: ", arr[i] = p)
            break
        Else:
            arr[n/2] = p
            print("The mid-point is the peak")
```

break;

Example: arr = [1,2,3,4,5,6,7,8,9,10,11]

n = 11

p = 9

From the above pseudocode, we see that we are doing at most three passes of the array and in each, we are dividing the array at most by half.

In the first instance, the array is divided by half and the 5^{th} element is checked against 'p'. As p > n/2 we'll move on to the second condition to probe the $((n/2) + 1)^{st}$ part of the array and do a second pass. We'll chuck elements [1,2,3,4,5] and the resultant new array is [6,7,8,9,10,11].

As we are taking ceiling(n/2), p > n/2. So, we'll probe the $((n/2) + 1)^{st}$ part of the array and do a second pass. We'll chuck elements [6,7] and the resultant new array is -> [8,9,10,11].

Now, p = n/2. So the loop stops and we can conclude that the peak is actually after the original mid-point.

The time complexity of the above algorithm is O(log n).

Q2.) Solved Exercise #2

Ans.) **Pseudocode** ->

```
Def comparestocks(left, right, compare):
    Result =[]
    i = 0
    j = 0
    while (i < len(left) and j < len(right)):
    if compare(left[i],right[j]):
        result.append(left[i])
        i += 1
    else:
        result.append(right[j])
        j += 1
```

```
    while (i < len(left)):
        result.append(left[i])
        i += 1
    while (j < len(right)):
        result.append(right[j])
        j += 1
    return result

    def optimaltrade(arr, compare = lambda  pi, pj: max(pi – pj)):
        if len(arr) < 2:
        return arr[:]
    else:
        middle = len(arr) / 2
        left = optimaltrade(arr[:middle], compare)
        right = optimaltrade(arr[middle:], compare)
        return results = comparestocks(left, right, compare)
    print(“Buy on : ”, results)
```

Example: n= 10, p(1) = 4, p(2) = 2, p(3) = 7, p(4) = 1, p(5) = 11, p(6) = 12, p(7) = 3, p(8) = 9, p(9) = 8, p(10) = 10 -> This will return -> “Buy on 5, 6, 10”

Time complexity of the above algorithm is -> O (n log n)