CS5800 PS2 - Aditya Shanmugham - NUID : 002738073

Problem 1:

1) **T (n) = 3T (n/2) + n (This is the recursion for Karatsuba's Algorithm)**

   Here, we can apply master theorem to solve this problem

   - $a = 3$
   - $b = 2$
   - $n/b = n/2$
   - $f(n) = n$

   $log_2 3 = 1.58 > 1$

   $f(n) =< n^{log_2 3 - \epsilon}$

   Case 1 implies here:

   The time complexity is $T(n) = \theta(n^{log_2 3}) = \theta(n^{1.58})$

2) **T(n) = 3T(n/2) + n2**

   Here, we can apply master theorem to solve this problem

   - $a = 3$
   - $b = 2$
   - $n/b = n/2$
   - $f(n) = n^2$

   $log_2 3 = 1.58 < 2$

   $f(n) >= n^{log_2 3 + \epsilon}$

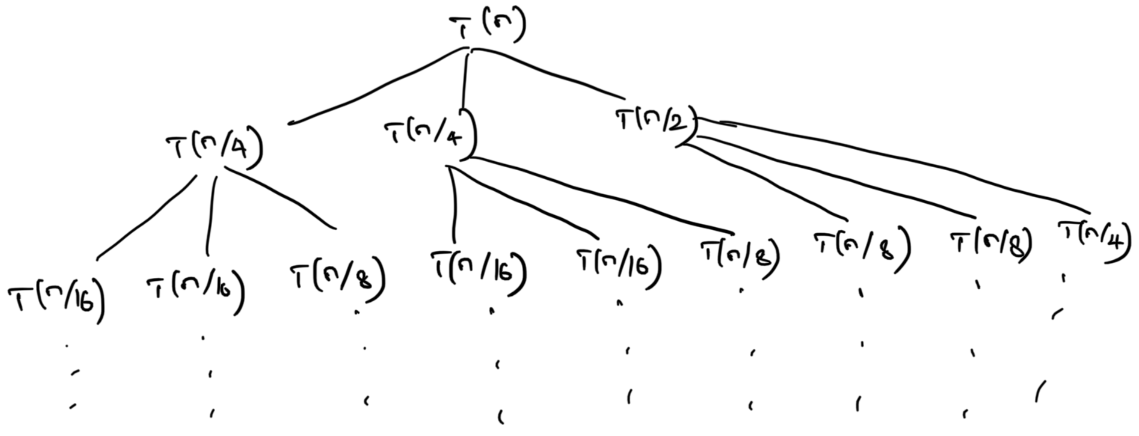
   Case 3 implies here

   The time complexity is $T(n) = \theta(n^2)$

3) T (n) = 2T (n/4) + T (n/2) + n (Hint: Think of the number of nodes at each level. You can determine it by solving a recurrence).

We cannot use master theorem in this problem as it doesnt have uniform divisions

Applying Recursive Tree Analysis:



We have two separate divisions of Recursive calls *h1* and *h2*

Where,

$$h1 \ = \ height\ of\ the\ tree\ associated\ with\ n/4\ division$$

$$h2 \ = \ height\ of\ the\ tree\ associated\ with\ n/2\ division$$

Time taken for *h1:*

$$h1 \ = \ O(nlog_4 n)$$

$$h2 \ = \ O(nlog_2 n)$$

Since we are considering the worst case scenario where $O(nlog_2 n)$ > $O(nlog_4 n)$, we can safely assume that

$$T(n) \ = \ O(nlog_2 n)$$

4) T(n) = 5T(n/4) + n

Here, we can apply master theorem to solve this problem

- $a = 5$
- $b = 4$
- $n/b = n/4$
- $f(n) = n$

$log_4 5 = 1.16 > 1$

$f(n) <= n^{log_4 5 - \epsilon}$

Case 1 implies here

The time complexity is $T(n) = \theta(n^{log_4 5}) = \theta(n^{1.16})$

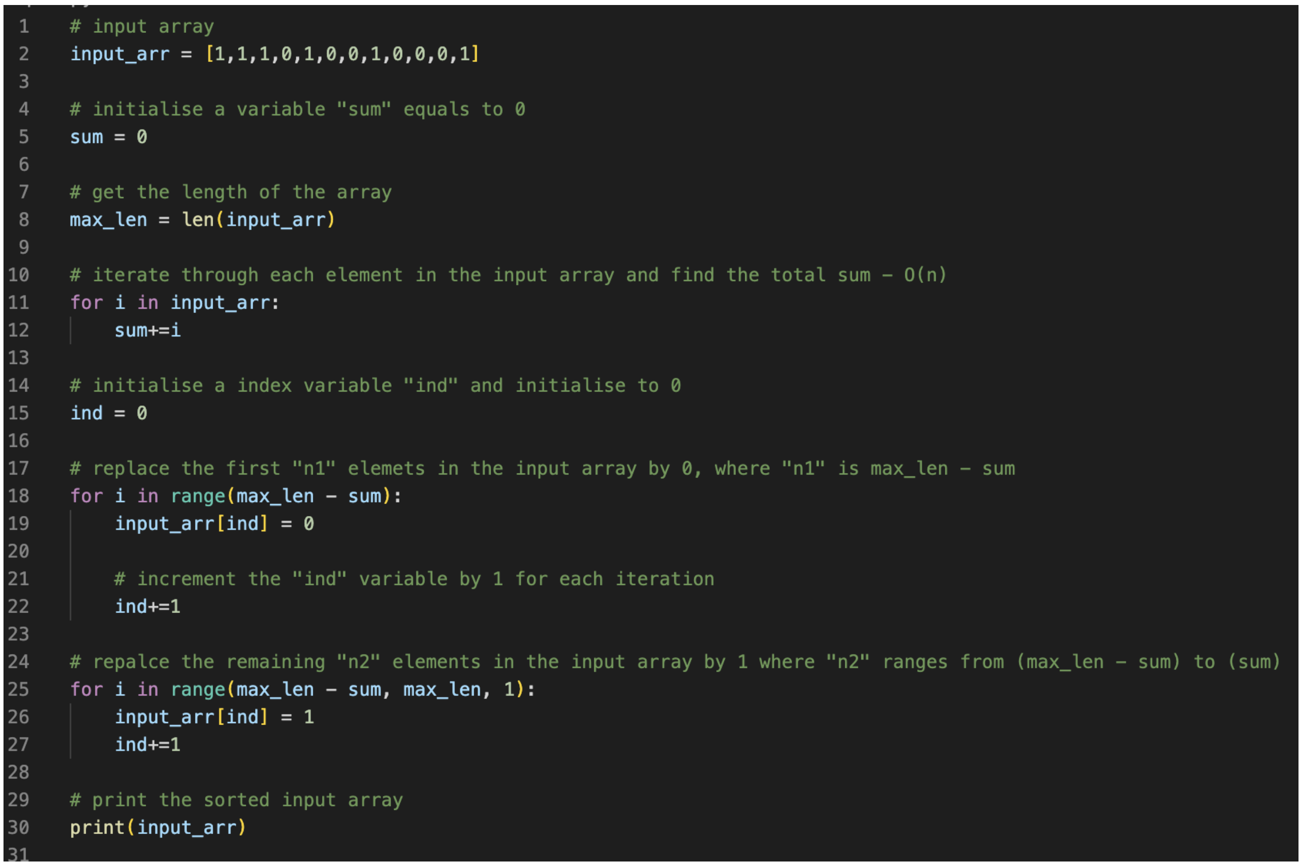
Problem 2:

1) **Come up with an efficient sorting algorithm for a boolean array B[1, ..., n].**

   We can use an algorithm which calculates the total sum and sorts the array

Pseudo Code:

- Find the length of the *input_arr*
- Add every element and store it in a variable *sum*
- Add *n1* zeros to the output array, where n1 = *max_len - sum*
- Add n2 ones to the output array, where n2 = *sum*



```python
# input array
input_arr = [1,1,1,0,1,0,0,1,0,0,0,1]

# initialise a variable "sum" equals to 0
sum = 0

# get the length of the array
max_len = len(input_arr)

# iterate through each element in the input array and find the total sum - O(n)
for i in input_arr:
    sum+=i

# initialise a index variable "ind" and initialise to 0
ind = 0

# replace the first "n1" elemets in the input array by 0, where "n1" is max_len - sum
for i in range(max_len - sum):
    input_arr[ind] = 0

    # increment the "ind" variable by 1 for each iteration
    ind+=1

# repalce the remaining "n2" elements in the input array by 1 where "n2" ranges from (max_len - sum) to (sum)
for i in range(max_len - sum, max_len, 1):
    input_arr[ind] = 1
    ind+=1

# print the sorted input array
print(input_arr)
```

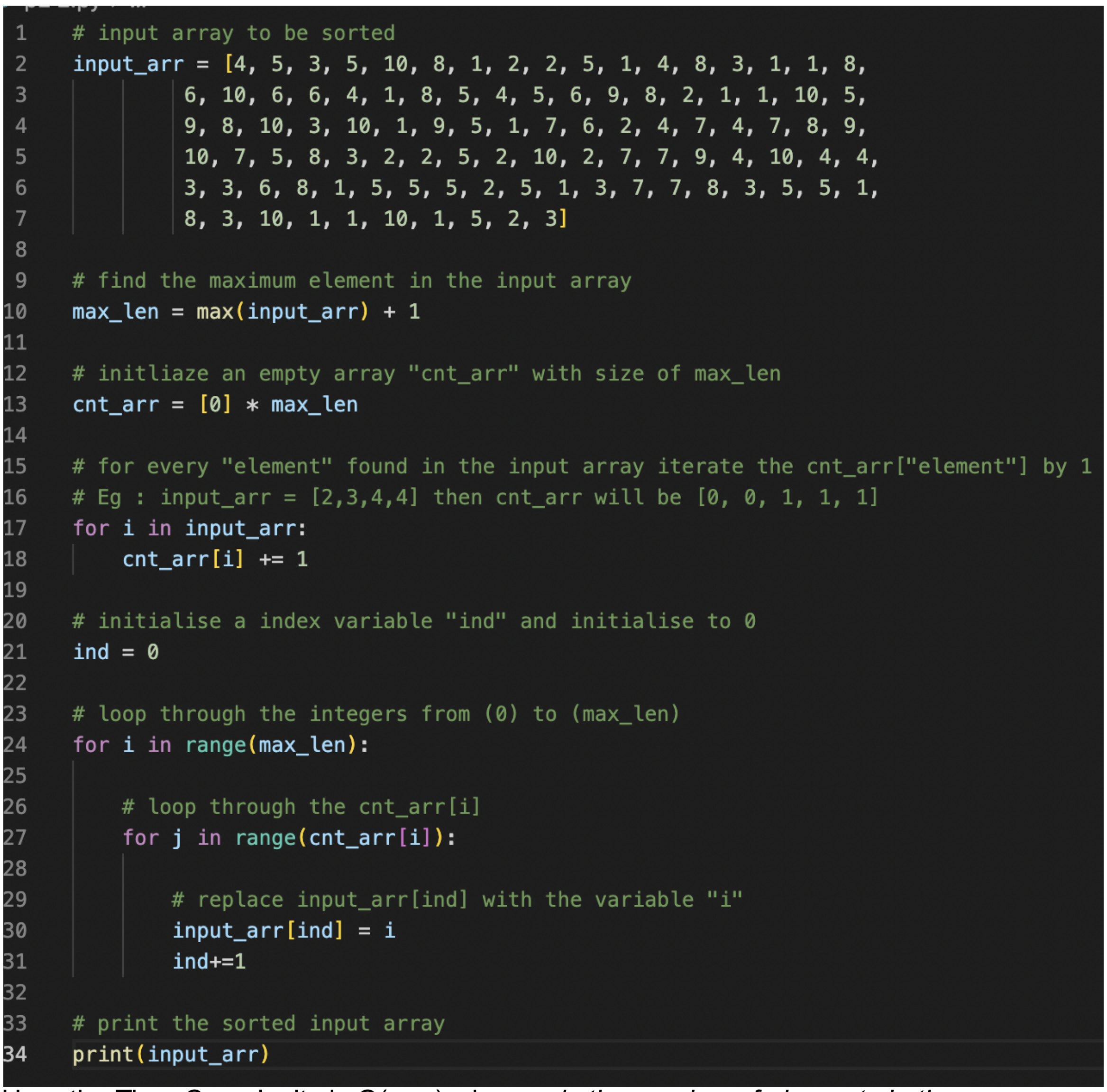
Here the Time Complexity is O(n) where n is the *number of elements in the array.*

2) Come up with an efficient sorting algorithm for an array C[1,...,n] whose elements are taken from the set {1,2,3,4,5,6,7,8,9,10}.

We can use pigeon sort algorithm to sort the input array

Pseudo Code:

- Find the max element in the array and store it in the variable *max_len*
- Create a new array - *cnt_arr* consisting of zeros with size (*max_len* + *1)*
- Iterate through each element in the *input_arr* and increment the *cnt_arr[element]* by 1
- Now Iterate through each element in the *cnt_arr* and add the index to the output array.



```python
# input array to be sorted
input_arr = [4, 5, 3, 5, 10, 8, 1, 2, 2, 5, 1, 4, 8, 3, 1, 1, 8,
             6, 10, 6, 6, 4, 1, 8, 5, 4, 5, 6, 9, 8, 2, 1, 1, 10, 5,
             9, 8, 10, 3, 10, 1, 9, 5, 1, 7, 6, 2, 4, 7, 4, 7, 8, 9,
             10, 7, 5, 8, 3, 2, 2, 5, 2, 10, 2, 7, 7, 9, 4, 10, 4, 4,
             3, 3, 6, 8, 1, 5, 5, 5, 2, 5, 1, 3, 7, 7, 8, 3, 5, 5, 1,
             8, 3, 10, 1, 1, 10, 1, 5, 2, 3]

# find the maximum element in the input array
max_len = max(input_arr) + 1

# initliaze an empty array "cnt_arr" with size of max_len
cnt_arr = [0] * max_len

# for every "element" found in the input array iterate the cnt_arr["element"] by 1
# Eg : input_arr = [2,3,4,4] then cnt_arr will be [0, 0, 1, 1, 1]
for i in input_arr:
    cnt_arr[i] += 1

# initialise a index variable "ind" and initialise to 0
ind = 0

# loop through the integers from (0) to (max_len)
for i in range(max_len):

    # loop through the cnt_arr[i]
    for j in range(cnt_arr[i]):

        # replace input_arr[ind] with the variable "i"
        input_arr[ind] = i
        ind+=1

# print the sorted input array
print(input_arr)
```

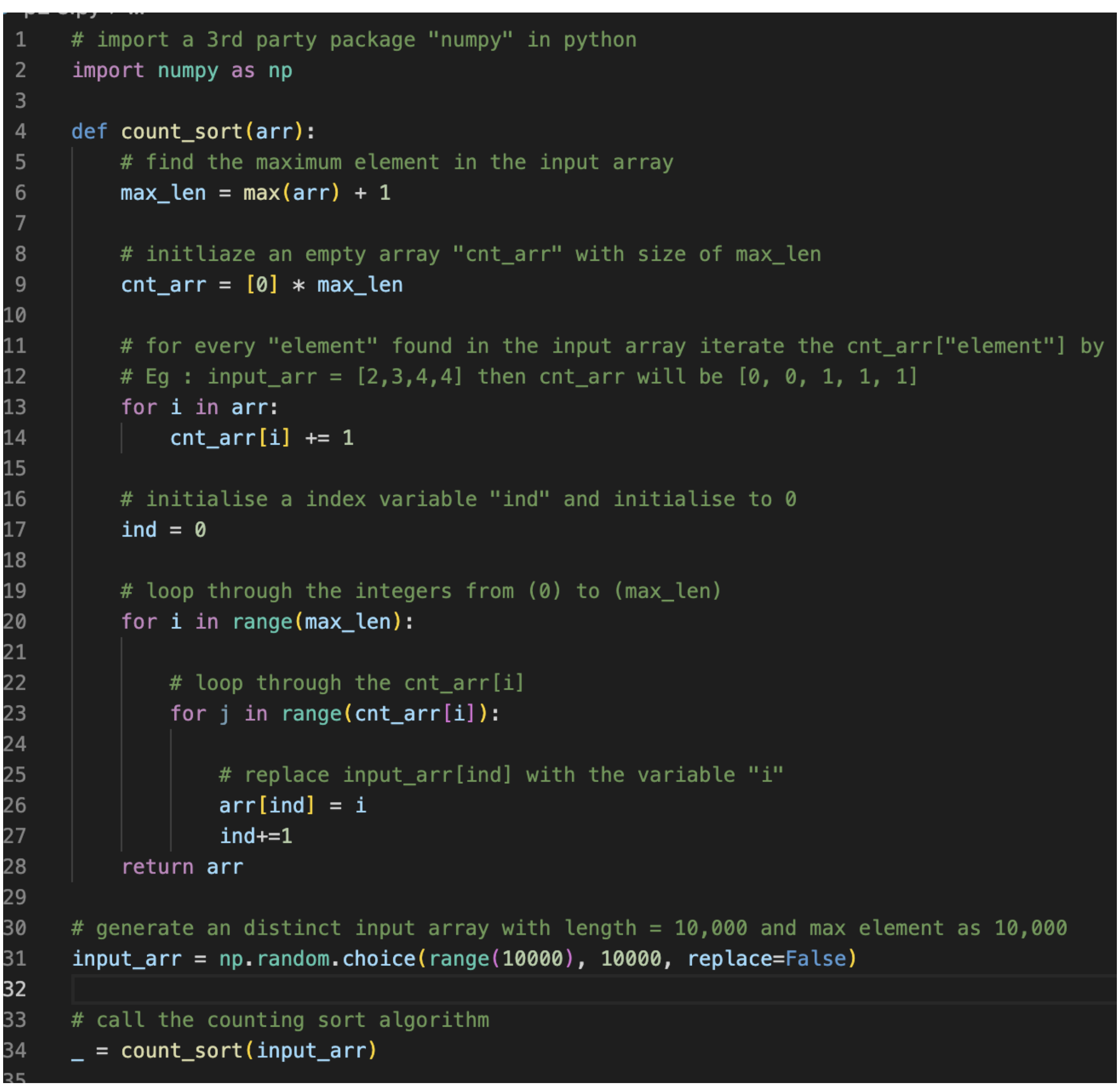
Here the Time Complexity is O(n+e) where *n is the number of elements in the array* and *e is the range of elements in the array.*

3) ***Come up with an efficient sorting algorithm for an array D[1, ..., n] whose elements are distinct (D[i] ≠ D[j], for every i ≠ j ∈ {1, ..., n}) and are taken from the set {1, 2, ..., 100n}.***

We can use counting algorithm to sort the input array

Pseudo Code for counting algorithm:

- Find the max element in the array and store it in the variable *max_len*
- Create a new array - *cnt_arrw* consisting of zeros with size (*max_len* + *1)*
- Iterate through each element in the *input_arr* and increment the *cnt_arr[element]* by 1
- Now Iterate through each element in the *cnt_arr* and add the index to the output array.



```python
# import a 3rd party package "numpy" in python
import numpy as np

def count_sort(arr):
    # find the maximum element in the input array
    max_len = max(arr) + 1

    # initliaze an empty array "cnt_arr" with size of max_len
    cnt_arr = [0] * max_len

    # for every "element" found in the input array iterate the cnt_arr["element"] by
    # Eg : input_arr = [2,3,4,4] then cnt_arr will be [0, 0, 1, 1, 1]
    for i in arr:
        cnt_arr[i] += 1

    # initialise a index variable "ind" and initialise to 0
    ind = 0

    # loop through the integers from (0) to (max_len)
    for i in range(max_len):

        # loop through the cnt_arr[i]
        for j in range(cnt_arr[i]):

            # replace input_arr[ind] with the variable "i"
            arr[ind] = i
            ind+=1
    return arr

# generate an distinct input array with length = 10,000 and max element as 10,000
input_arr = np.random.choice(range(10000), 10000, replace=False)

# call the counting sort algorithm
_ = count_sort(input_arr)
```

Here the Time Complexity is O(n+e) where *n is the number of elements in the array* and *e is the range of elements in the array.* Since we are dealing with very large array with large numbers the time complexity of counting algorithm will be actually greater than the lower bound of the comparison based algorithms (Merge Sort)

Even Though, the Asymptotic notation for Counting sort is Linear - O(n+e) when we are dealing with large numbers, merge sort performs better - O(nlogn).

Pseudo Code for Merge Sort:

- Declare left and right var which will mark the extreme indices of the array
- Left will be assigned to 0 and right will be assigned to n-1
- Find mid = (left+right)/2
- Call mergeSort on (left,mid) and (mid+1,rear)
- Above will continue till left<right
- Then we will call merge on the 2 subproblem

The reason why counting sort is slower than merge sort is because of the variable “e” in the time complexity of the counting sort algorithm. When the range of the number becomes very large, O(nlogn) will be faster than O(n+e).

4) **In case you designed linear-time sorting algorithms for the previous subparts, does it mean that the lower bound for sorting of Ω(n log n) is wrong? Explain.**

The above sorting algorithms are not based on comparisons, so in general comparison algorithms have lower bounds of O(nlogn).

In the above examples the counting based algorithms have a linear time complexity as we are exploiting some relaxation in the given question.

Since we are using counting based sorting algorithms, the time complexity will be greater than O(nlogn) when we deal with large numbers and very long lists.

In counting based algorithms we are trading off the time complexity for the space complexity of the given problem, and also when the range of the array increases the counting based algorithms will prove to be non-effective compared to the comparison based algorithms (merge sort, quick sort, etc.). We can safely assume that sorting an array in general scenarios will be much faster when we use comparison based algorithms O(nlogn).

Problem 3:

1) **Show an Ω(log n) lower bound for finding elements in a sorted array (in a comparison based model).**

Assume we have a sorted array of [1,2…n], using naive approach the time taken to search for an element will be O(n)..

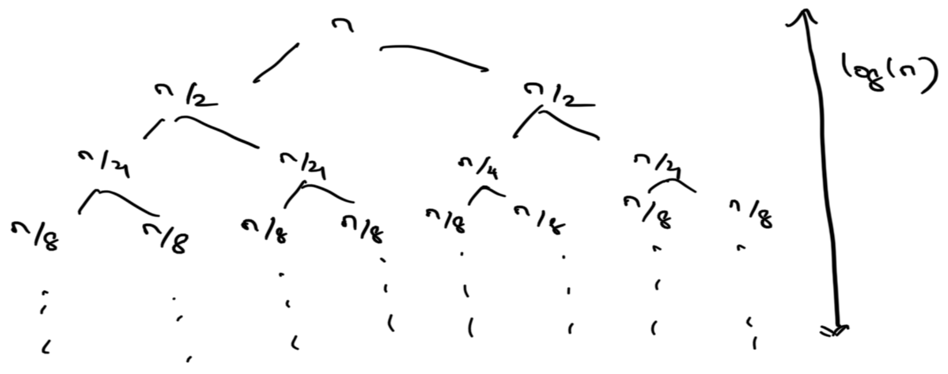
Since the input array is sorted, we can use comparison based decision tree to search for the element in the array.

The Pseudo Code for the decision tree algorithms:

```
Function find_elem(arr):
    mid = len(arr)/2
    If arr[mid] == target:
        return arr[mid]
    else if arr[mid] > target:
        return find_elem(arr[0:mid])
    else:
        return find_elem(arr[mid:])
```

In binary tree (decision tree), the depth of the tree will be log(n) where n is the number of nodes present in the tree. In the worst case scenario, the element can be at the end of the array (largest element in the array), now we need to traverse log(n) nodes to find the element, so the worst case time complexity will be a lower bound of O(logn) for searching an element in a sorted array.

Even there exist a algorithm that claims to be faster than O(logn), it needs to at least compare 2 elements [O(1)] and then traverse through either of its children [branching factor = b] and then traverse through the whole length of the tree [depth = d]. When we take the time complexity of the algorithm, it will be O($log_b d$). So the lower bound for searching an element in a sorted array will be O(logn).



We can also formulate the lower bound as,

$$T(n/2) + 1, T(n/2^{2}) + 1, T(n/2^{4}) + 1, ... T(n/2^{k}) + 1, T(n/2^{logn}) + 1$$

Which can be reduced as

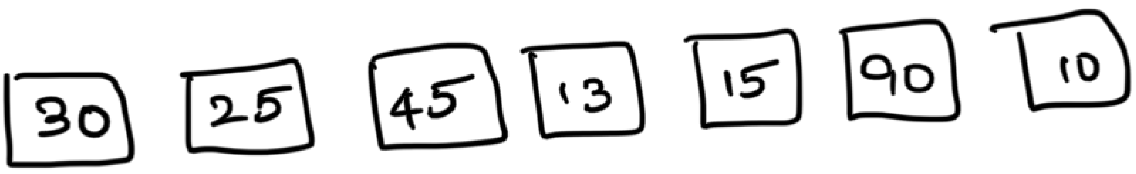
$$= O(logn)$$

Problem 4:

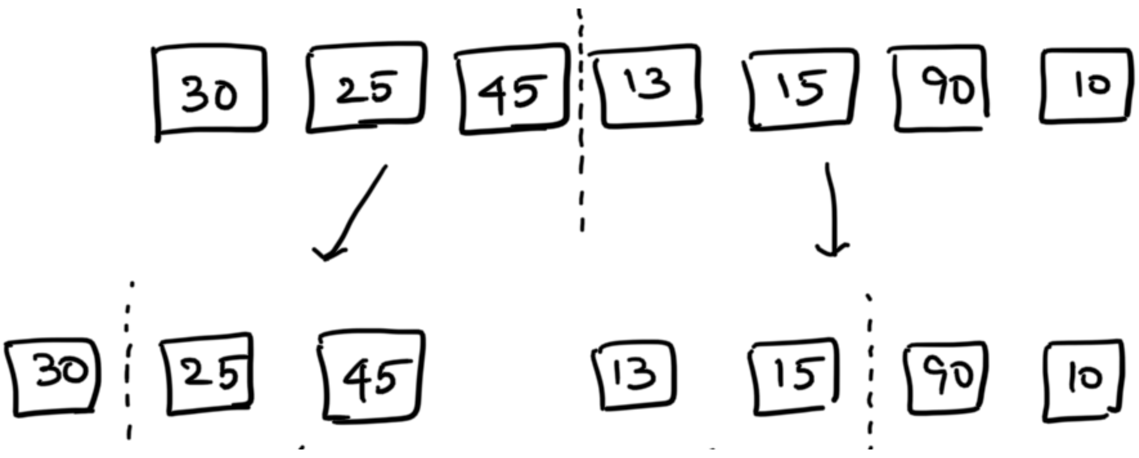
1) Description of the algorithm:

My algorithm uses Merge sort to find the number of times a swap has taken place. Since merge sort works on the principle of divide and conquer, effectively the solution is also based on divide and conquer approach.

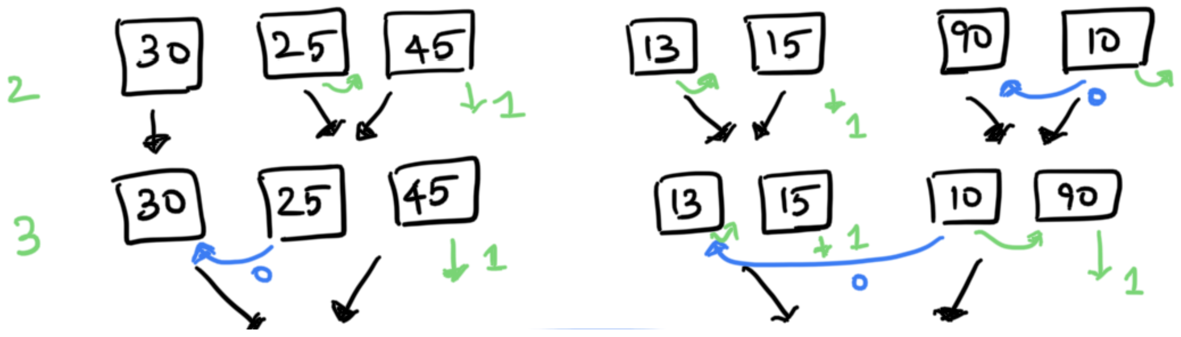
a) The main principle is to divide the array into small subarrays and conquer the value by recursively solving it. Initially we will divide the array into ***left subarray*** and ***right subarray*** and recursively call the merge sort function on these subarrays. Check if *L(0)* is less than *R(len(arr) = 7),* if so then split the array into left subarray right subarray from the middle element.



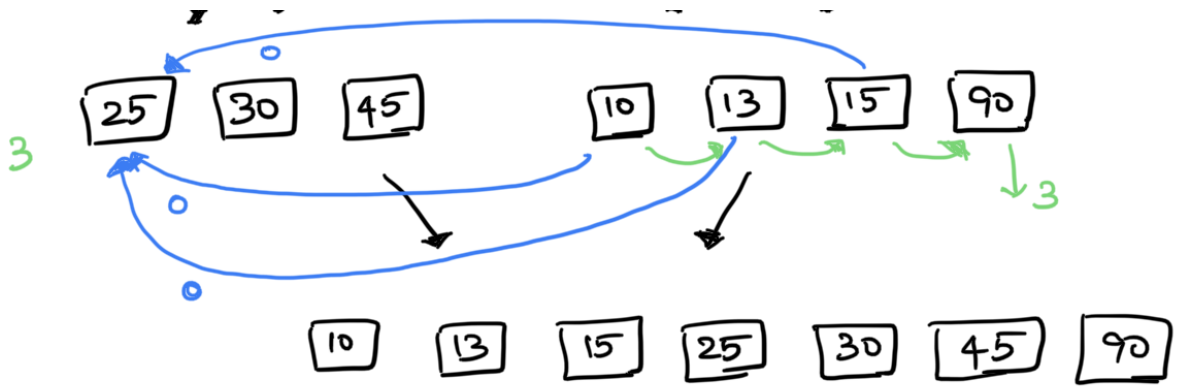
b) Now again call the merge_sort function and check if L is less than R, if so split again.



c) Keep on dividing the array using recursive call until singleton array is reached where the further division is not possible. After dividing the array into smaller units, start merging the array forming a sorted subarray based on comparing each element in the left and right sub arrays.



d) Increment "*ans"* by "*position i"* if the L[i] > R[j], after all the comparisons are done in that recursion call, increment *"ans"* by *"position i"* for every remaining elements in the right sub array.

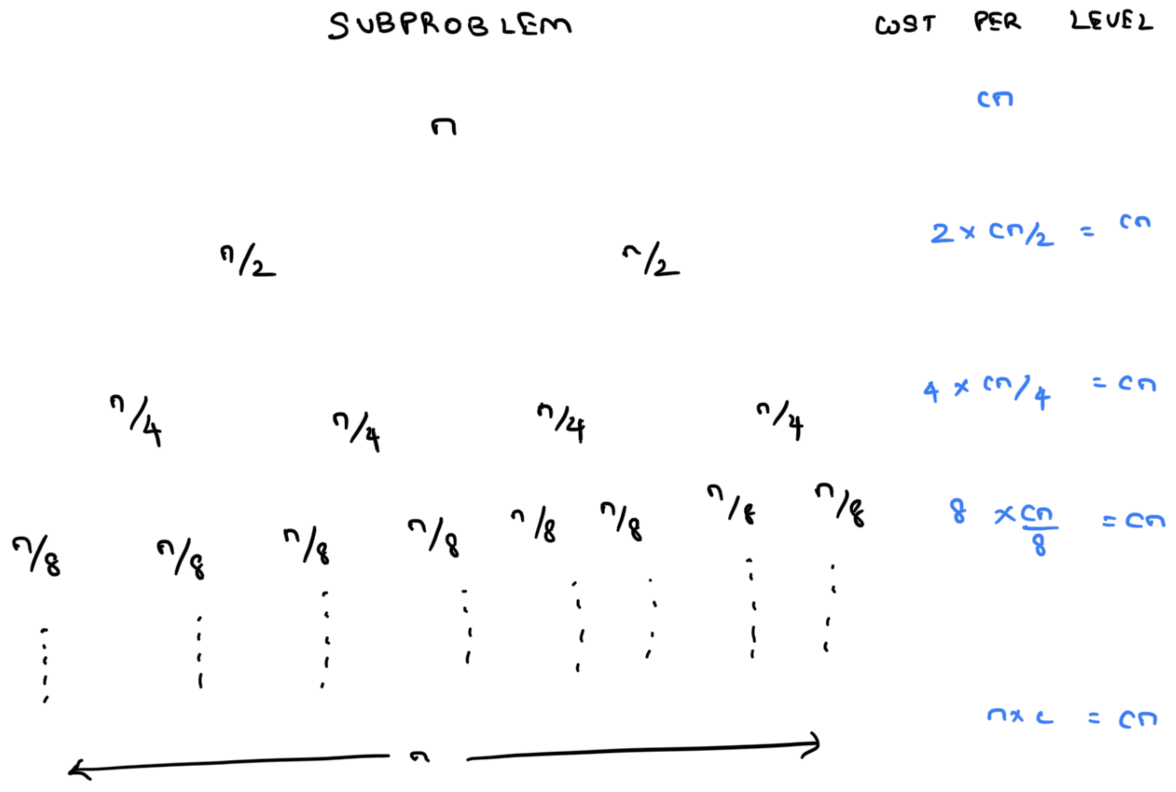


In the step ‘d’, we are counting the number of swaps undertaken by the merge sort algorithm, this can be calculated easily without duplicates by adding “i” every time L[i] > R[j] and adding “i” for the remaining elements in the R[j] sub array. Finally when we return the “ans” for the above example, the number of peaks detected will be 8.

Since I am using Merge Sort and counting the number of peaks, this algorithm is based on the divide and conquer approach.

2) Runtime Analysis:

Since my algorithm is running on merge sort algorithm, it takes $O(nlogn)$ as we are dividing the array into multiple sub array and comparing those sub arrays.



Here c = cost per stage = O(1) + c’, where c’ is constant of the internal operations and O(1) corresponds to incrementing ans by “i”.

The total time complexity is

$$\boxed{cost\ per\ stage\ *\ number\ of\ stages\ =\ (cn)\ *\ log(n)\ =\ c\ *\ (nlogn)\ =\ nlogn}$$