**Question 1 (8 points each)**

**The array PARTITION contains the following values:**

**PARTITION:**

**Index: 0 1 2 3 4**

**Value: A B A B A**

**a. What does the array rpA look like after applying the PARTITION algorithm?**

Consider that Partition algorithm is using the Lomuto partition scheme. That places those elements that are equal or less than from the pivot on the left side of it and greater than pivot on the right side of it.

If we consider the last element in the Pivot is A.

Now we see the array

Original: [A, B, A, B, A] (pivot = A)

Here we make comparison of every element with the Pivot Element that is A.

We compare each element with pivot A:

* At Index 0 A is equal swap it with itself
* At index 1 B is greater than do nothing
* At index 2 A is equal so swap A with B at the index 1
* At index 3 B is greater so do nothing

Now the array after PARTITION is

[A, A, A, B, B]

rpA=[A,A,A,B,B] after resulting

**b. What is the purpose of the QUICKSORT algorithm in general?**

The purpose of the QUICKSORT algorithm is for sorting an array list by the technique of recursively partitioning the array into two subarrays based on the pivot element. Every partitioning will places pivot elements to it correct position and make sure that all elements less the pivot are on the left side of it and greater that pivot are right side of it

**c. What is the role of the PARTITION procedure within the QUICKSORT algorithm?**

The PARTITION procedure has an important role in subroutine of the QUICKSORT that is

* First choose an element that is pivot
* After choosing it rearrange the whole array in such a way that elements less than or equal to pivot are placed left side of it
* Elements that will be greater than pivot will be swapped to right side of it.
* Now we will get the pivot element at its actual position. Now sort out the left and right subroutine recursively

**d. What is the time complexity of the QUICKSORT algorithm in the best-case, worst-case, and average-case scenarios?**

Time Complexity is QUICKSORT algorithm is given below

* For **Best case** is O(nlogn) because pivot is dividing the list to two equal halves.
* The **Average case** is O(n log n) because random pivot will give average balancing
* The **Worst case** is O(n2) because pivot will divide the array not into equal halves

**Question 2 (36 points – A:16p , B:8p, C:12p) You are given three sorted arrays in ascending order, each of size n, which contain only natural numbers (positive integers) — no duplicates within each individual array.**

* **Suggest an efficient algorithm for finding a number that appears in all three arrays (i.e., a common element).**

We will use three-pointer approach that will be best for it.

Suppose we have three arrays A,B and C. They are in sorted order

First we initialize three pointers like i=0 , j=0 and k=0 at the beginning of the array A,B and C.

Now compare elements at the given positions that are pointed out by the pointer

* Let A[i]=B[j]=C[k] in this way we will found the common element in between them
* Let A[i]<B[j] in such a way increment i.
* Let A[i]>B[j] and B[j]<C[k] in such a way increment j.
* Let A[i]>C[k] and B[j]>C[k] in such a way increment k.  
  Repeat step 2 until one pointer will reach the end of its array. In such process if no element is found that will match before ending then there will be no elemnet common between them

**• What is the time complexity of your suggested algorithm?**

The time complexity of the algorithm is O(n) is here. In situation if there is a worst-case then we have to perform traversing for all the elements of these three arrays at least once. As every comparison and pointer is taking constant time O(1). The maximum number of steps are 3n in worst case where possibly we have to check all elements. But overall time complexity is O(n).

**• Is it possible to achieve better complexity?**

No, The reason is that we have to check every element at least one time to make sure that no element we have miss matched. But the array is in sorted order and size of array is n. So if asymptotically faster algorithms will skip elements that are unseen will potentially missing a match. So O(n) is considered to be the optimal time complexity.

**Question 3 (10 points)**

**Regarding the array:**

**A = (3, 7, 8, 1, 8, 3, 9, 1, 8, 7, 8)**

Step-by-step breakdown:

1. **Find the range of values**:
   * Minimum value = 1
   * Maximum value = 9
   * So C array must go from index 0 to 9 (or more generally 0 to k where k = max(A)).

We break down it into step by step process

**1.For finding the range of the values:**

* The minimum value is =1
* The maximum value is =9
* So now the array C will go from index 0 to 9

**2.Intitial array C**

C = [0] \* 10 # 0 to 9

# Counting occurrences

for j in range(len(A)):

C[A[j]] += 1

The array A is consists of

* 1 two time
* 3 two time
* 7 two time
* 8 four time
* 9 one time

At the start of line 6 Array c will become

C = [0, 2, 0, 2, 0, 0, 0, 2, 4, 1]

**3. C array Cumulative**

for i in range(1, len(C)):

C[i] += C[i - 1]

**After the Cumulative sum**

C = [0, 2, 2, 4, 4, 4, 4, 6, 10, 11]

**4. Sorted array B at the end of subroutine**

We have to move in reverse order for A for maintain stability

B = [0] \* len(A)

for j in range(len(A)-1, -1, -1):

B[C[A[j]] - 1] = A[j]

C[A[j]] -= 1

**Performing iteration in Backward for A**

A = [3, 7, 8, 1, 8, 3, 9, 1, 8, 7, 8]

Indexes: 0 1 2 3 4 5 6 7 8 9 10

**B will be filled for decreasing Value of C**

After the processing of 8: the B[9] = 8 and C[8] = 9

After the processing of 7: the B[5] = 7 and C[7] = 5

After the processing of 8: the B[8] = 8 and C[8] = 8

After the processing of 1: the B[1] = 1 and C[1] = 1

After the processing of 9: the B[10] = 9 and C[9] = 10

After the processing of 3: the B[3] = 3 and C[3] = 3

After the processing of 8: the B[7] = 8 and C[8] = 7

After the processing of 1: the B[0] = 1 and C[1] = 0

After the processing of 8: the B[6] = 8 and C[8] = 6

After the processing of 7: the B[4] = 7 and C[7] = 4

After the processing of 3: the B[2] = 3 and C[3] = 2

**Now the Final sorted array B**

B = [1, 1, 3, 3, 7, 7, 8, 8, 8, 8, 9]

**Question 3b: Is the algorithm still stable?**

Yes algorithm will remain still in stable condition even after number of times of the looping process

for j ← 1 to length[A]

for j in range(0,len(A)) it will not maintain the stability.

**Original COUNTING-SORT uses backward traversal**

for j = length[A] down to 1

It is ensuring stability as early elements that have same value stay ahead of the sorted array.

**So,** If A process is in forward order than algorithm will not be in stable

Because if we will preserve the reverse order then it will remain stable even if the loop is rewritten technically.