Assignment 3 Solutions :

This document contains a complete solution for some questions and an added report for the questions which have been solved as codes in JuPyter Notebooks. It also has snapshots of the JuPyter notebooks, along with this document, the submission also contains attached JuPyter Notebooks and its .html files for easier viewing or checking codes as required.
Questions 1,2,5 have complete mathematical proof written, Questions 3,4 are written codes and attached report is written in this document, along with this, the codes are also attached in the submission.

Question 1:

Show that the running time of the merge-sort algorithm on n -element sequence is O(n log n), even when n is not a power of 2.

I will use the method of Mathematical induction to show the same.

Claim : $T(n) \le c1 \times nlogn + c2 \times n$, where c1,c2 are arbitrary real constants.

We know from the recursion that : $T(n) = T([n/2]) + T(n - [n/2]) + k2 \times n$,where k2 is some real constant (k2*n term comes from the merge call)

Base Case : $n = 1 \Rightarrow T(1) = c\ (since\ nothing\ to\ sort) \le c1 \times 1log1 + c2 \times 1$, such constants can be easily found, hence shown for base case.

Assumption : Let the claim be true for all integers from 1 to n-1 for some n>1.

To show that : The claim is true for n
Using the recursion; we have $T(n) = T([n/2]) + T(n - [n/2]) + k2 \times n$

Case 1: n is even $\Rightarrow$ $T(n) = 2 \times T(n/2) + k2 \times n$

Since for all integers less than n, the claim is true, it is true for n/2 and we can use that equation and substitute the value in the above equation.

$$\Rightarrow T(n) \le 2 \times \{c1 \times (n/2) \times log(n/2) + c2 \times (n/2)\} + k2 \times n$$
$$\Rightarrow T(n) \le c1 \times n \times (logn - log2) + c2 \times n + k2 \times n$$
$$\Rightarrow T(n) \le c1 \times nlogn + (c2 + k2 - c1log2) \times n$$

Since constants can be arbitrary real numbers in the inequality claimed for T(N), choose c1’ = c1 and c2’ = c2 + k2 - c1log2

$$\Rightarrow T(n) \le c1' \times nlogn + c2' \times n$$
Hence, the claim has been proven for even n.

Case 2 : n is odd $\Rightarrow$ $T(n) = T((n-1)/2) + T((n+1)/2) + k2 \times n$

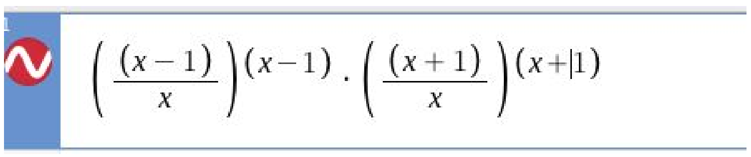
Since for all integers less than n, the claim is true, it is true for (n-1)/2 ,(n+1)/2 and we can use that equation and substitute the value in the above equation.

$$\Rightarrow T(n) \le c1 \times ((n-1)/2) \times log((n-1)/2) + c2 \times ((n-1)/2) + c1 \times ((n+1)/2) \times log((n+1)/2) + c2 \times ((n+1)/2) + k2 \times n$$

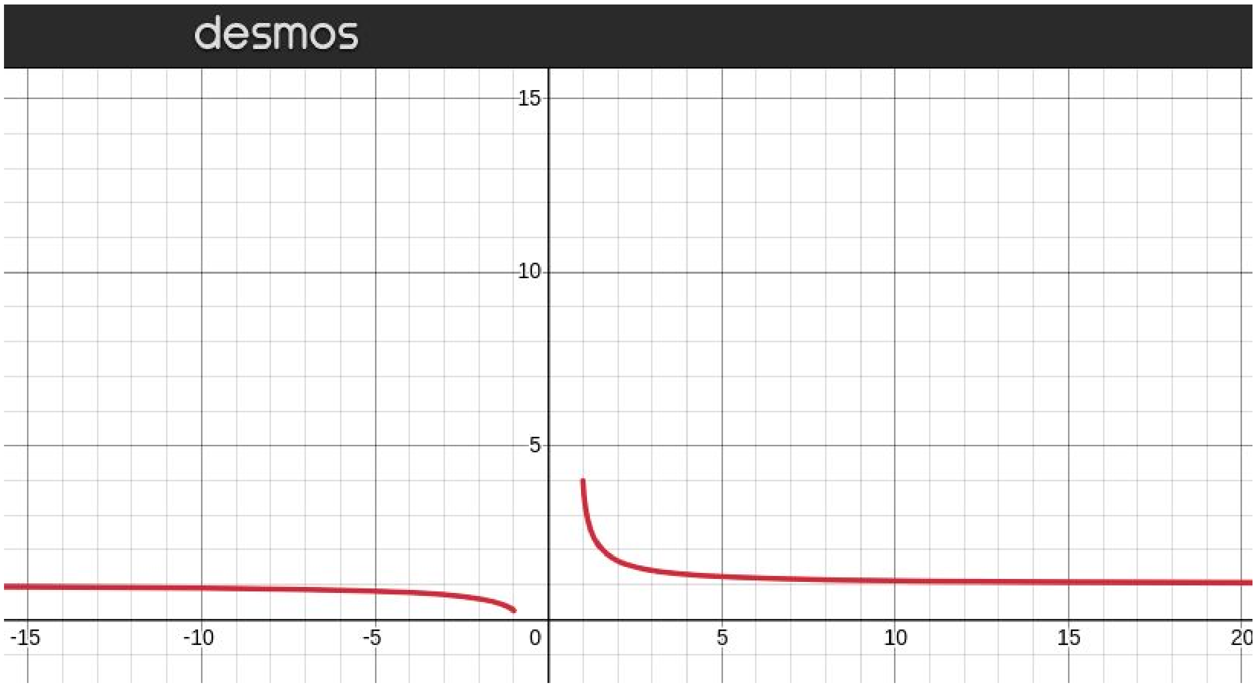
$$\Rightarrow T(n) \le c1 \times \{(n-1)/2 \times log(n-1) + (n+1)/2 \times log(n+1)\} - c1 \times nlog2 + c2 \times n + k2 \times n$$

To find a constant k1 such that :
$$(n-1)/2 \times log(n-1) + (n+1)/2 \times log(n+1) \le k1 \times nlogn$$
$$\Leftrightarrow log(n-1)^{(n-1)} + log(n+1)^{(n+1)} \le logn^{(2\times k1 \times n)}$$
$$\Leftrightarrow (n-1)^{(n-1)} \times (n+1)^{(n+1)} \le n^{(2\times k1\times n)}$$

$$\Leftrightarrow ((n-1)/n)^{(n-1)} \times ((n+1)/n)^{(n+1)} \le n^{2n(k1-1)}$$


$$\left(\frac{(x-1)}{x}\right)^{(x-1)} \cdot \left(\frac{(x+1)}{x}\right)^{(x+1)}$$

The graph for the above equation is as follows :



Which implies, that LHS ≤ 4 and equality occurs as n tends to 1.
Since we clearly have n>1; we have to find a k1 such that :

$4 \leq n^{2n(k1-1)}$ and we have n $\geq 2 \Rightarrow k1 = 2$ *will also suffice the equality for every n.*

Choose k1 = 2 for this case :

$\Rightarrow T(n) \leq 2c1 \times nlogn + (c2 + k2 - c1log2) \times n$

Since constants can be arbitrarily chosen, take c1' = 2c1 , c2' = c2 + k2 -c1log2

$\Rightarrow T(n) \leq c1' \times nlogn + c2' \times n$

Hence, the claim has been proven for odd n.

Since it has been shown that the claim is true for all the cases, we can now say by the method of Mathematical Induction that the claim is true for all integers n.
$\Rightarrow T(n) = O(nlogn) \Leftrightarrow$ *for some* c, $T(n) \leq c \times nlogn$ *for all* $n \geq$ *some* m
Choose c = c1' + c2' $\Rightarrow c1' \times nlogn + c2' \times n \leq (c1' + c2') \times (nlogn)$
Hence proven that T(n) = O(nlogn) for all integers n

Question 2:

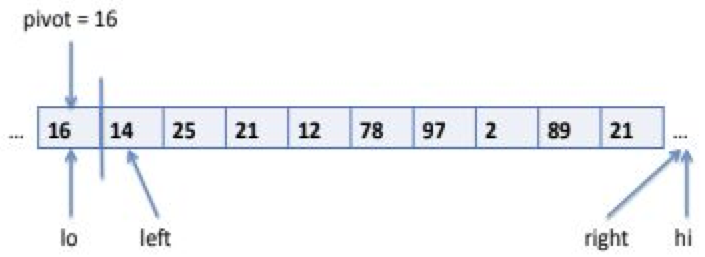
Consider a modification of the deterministic version of the quick-sort algorithm where we choose the element at index [n/2] as our pivot. Describe the kind of sequence that would cause this version of quick-sort to run in Ω (n^2) time.

Solution : I will first explain how a quick sort algorithm is implemented.

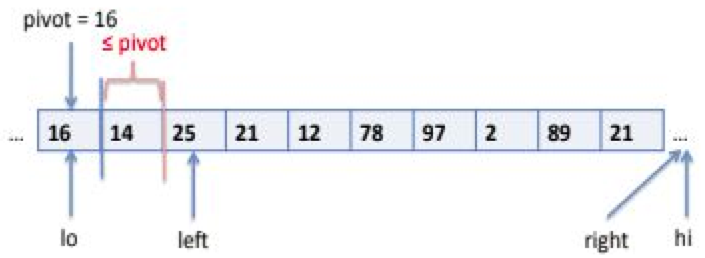
All the elements are compared with the pivot, a left_counter is set at 0 and a right_counter is set at length of array, now when each element is compared, if it is smaller than pivot then it is swapped with the element at position of left_counter and the left_counter is incremented by 1, whereas if it is larger than the pivot, then it is swapped with the element at position of right_counter and right_counter is decremented by 1.

Following is attached the step by step process of partitioning for an example array :

Step 1:

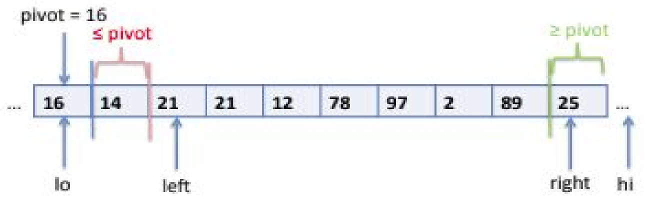


Since 14 $<$ *pivot*, we can advance the *left* index: this element is in the proper place.

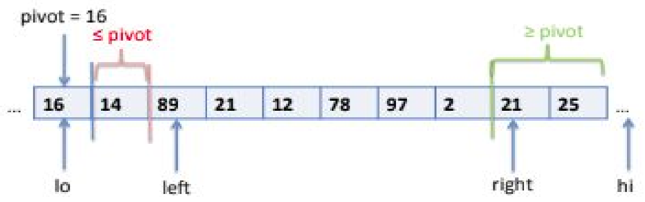


Step 2:

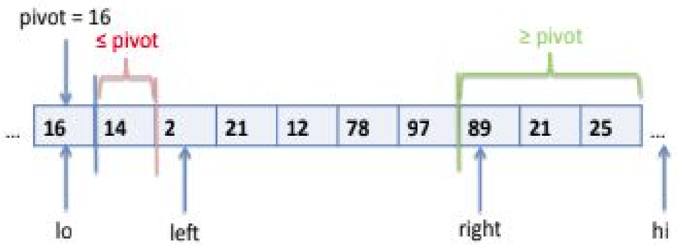
At this point, $25 > pivot$, it needs to go on the right side of the array. If we put it on the *extreme* right end of the array, we can then say that it is in it's proper place. We swap it into $A[right - 1]$ and decrement the $right$ index.



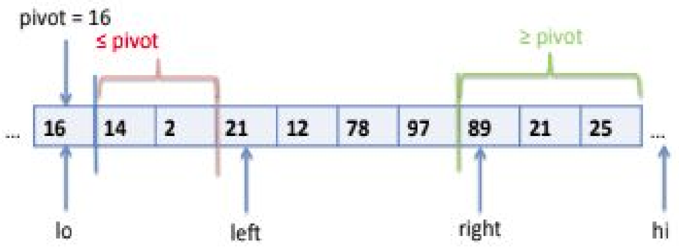
In the next two steps, we proceed by making swaps. First, we decide that the 21 that is currently at $left$ can be properly placed to the left of the 25, so we swap it with the element to the left of 25. Then, we have 89 at $A[left]$, and so we can decide this is well-placed to the left of that 21.



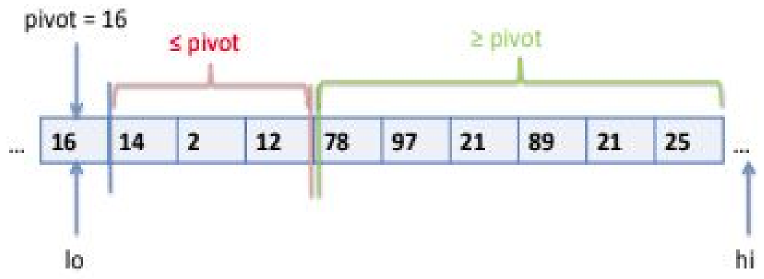
Step 3:



Let's take one more step: $2 < pivot$, so we again just decide that the 2 is fine where it is and increment $left$.



Partitioned array :



The idea which I will use for the purpose of building a worst case scenario is that the chosen pivot should always turn out to be the maximum of the partition, so it will have to be compared with all the elements in the array and then placed at the last position.

So, if the length of the partition is r, then for comparisons with the chosen pivot from the partition, we will have to iterate through a for loop of length (r-1) once. Now, the initial length of partition is n and it reduces by 1 every time since we are assuming that the chosen pivot is always maximum so it will always be shifted to the last position in that partition by an algorithm as stated above.

$$\Rightarrow T(n) = \sum_{1}^{n}(r-1)$$
$$\Rightarrow T(n) = n(n-1)/2$$

To show that : $T(n) = \Omega(n^2)$

We have, $T(n) \geq cn^2$ for c=1/4, for every
$n \geq 2,\ since\ (\ (n^2-n)/2\ \geq n^2/4\ \Leftrightarrow n^2 \geq 2n$
$\Rightarrow T(n) = \Omega(n^2)$

Now, we had made an assumption, that the chosen pivot should always be the maximum element of the partition for the above proof to hold true, now it can happen in an array where :

Largest element is at the position [n/2], this will be chosen as pivot, compared with all and shifted to last position without disturbing the order of the remaining array, now in the partition, the largest element of that partition should be at the position [(n-1)/2] since length of partition now is n-1 and we are choosing pivot as the element at floor function of (length of partition)/2 , and so on till the end.

In general, an n element array can be constructed such that the worst case happens by :

Case 1 : n is even

For r = 1 to (n/2) -1, value of element at rth position is (2r+1)
For r = n/2, value of element at rth position is n
For r = (n/2) +1 to n-1, value of element at rth position is 2(n-r)
For r = n, value of element at rth position is 1

An example of such an array for n=10 is :
3,5,7,9,10,8,6,4,2,1

Case 2 : n is odd

For r = 1 to (n-1)/2, value of element at rth position is (2r+1)
For r = (n+1)/2 to n-1, value of element at rth position is 2(n-r)
For r = n, value of element at rth position is 1

An example of such an array for n=11 is :
3,5,7,9,11,10,8,6,4,2,1

Note that an array with all elements equal will also cause quick sort to take $\Omega(n^2)$ time since it will also do all comparisons every time.

Question 3:

Describe and analyze an efficient method for removing all duplicates from a collection A of n elements.

This can be done using 2 methods:

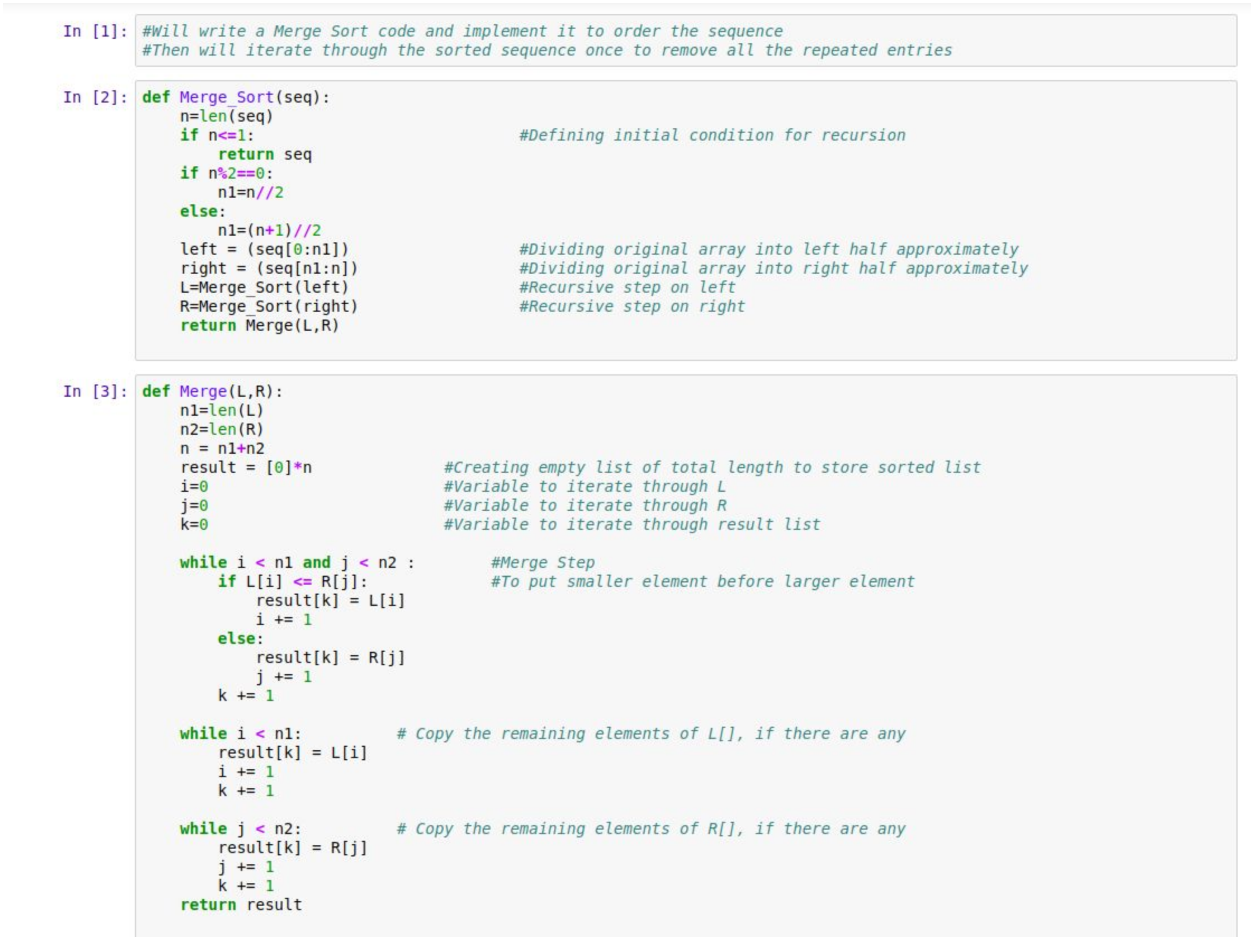
1st method (Using merge sort):

The assumption for this method is that each element of the array is a float or an integer (since merge sort can be implemented on numbers).

The idea is to sort the array using merge sort in time complexity O(nlogn) and then delete all the duplicates which can be done in time complexity O(n) since we have to iterate through the sorted array only once to delete all duplicates.

So total time complexity is O(nlogn).

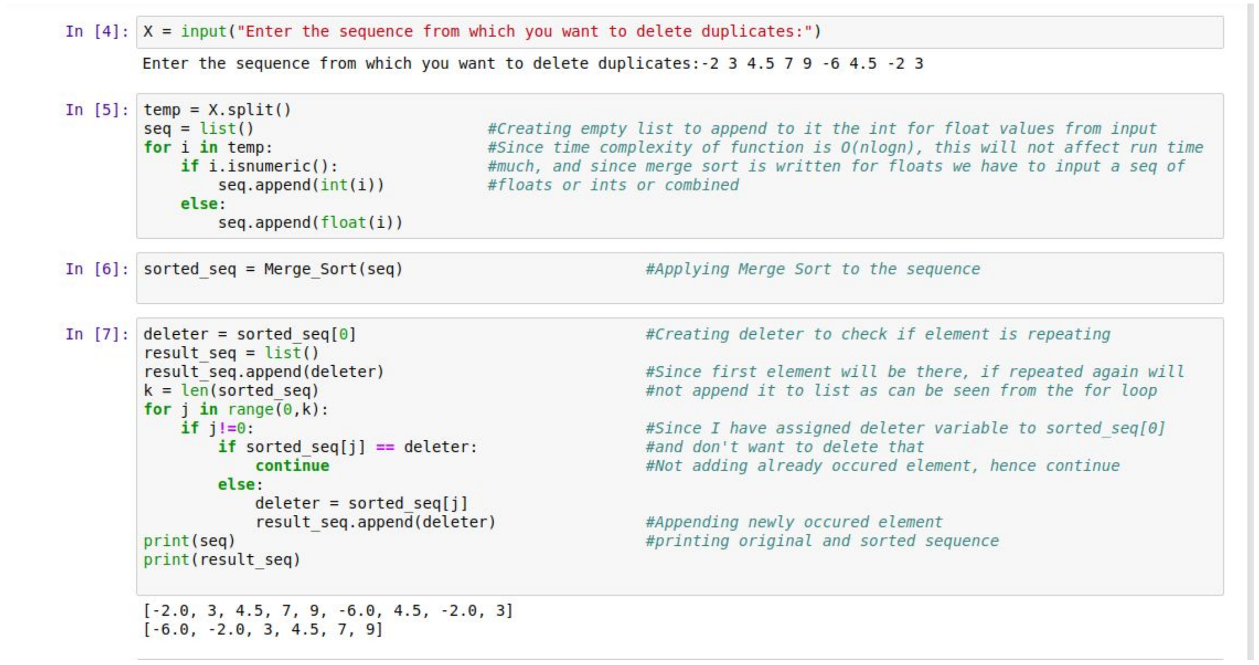
Following is the attached code (I have also included the .ipynb file and its downloaded .html form in the submission)



```
In [1]: #Will write a Merge Sort code and implement it to order the sequence
        #Then will iterate through the sorted sequence once to remove all the repeated entries
```

```
In [2]: def Merge_Sort(seq):
            n=len(seq)
            if n<=1:                                #Defining initial condition for recursion
                return seq
            if n%2==0:
                n1=n//2
            else:
                n1=(n+1)//2
            left = (seq[0:n1])                      #Dividing original array into left half approximately
            right = (seq[n1:n])                     #Dividing original array into right half approximately
            L=Merge_Sort(left)                      #Recursive step on left
            R=Merge_Sort(right)                     #Recursive step on right
            return Merge(L,R)
```

```
In [3]: def Merge(L,R):
            n1=len(L)
            n2=len(R)
            n = n1+n2
            result = [0]*n              #Creating empty list of total length to store sorted list
            i=0                         #Variable to iterate through L
            j=0                         #Variable to iterate through R
            k=0                         #Variable to iterate through result list

            while i < n1 and j < n2 :        #Merge Step
                if L[i] <= R[j]:             #To put smaller element before larger element
                    result[k] = L[i]
                    i += 1
                else:
                    result[k] = R[j]
                    j += 1
                k += 1

            while i < n1:         # Copy the remaining elements of L[], if there are any
                result[k] = L[i]
                i += 1
                k += 1

            while j < n2:         # Copy the remaining elements of R[], if there are any
                result[k] = R[j]
                j += 1
                k += 1
            return result
```



```
In [4]: X = input("Enter the sequence from which you want to delete duplicates:")

        Enter the sequence from which you want to delete duplicates:-2 3 4.5 7 9 -6 4.5 -2 3

In [5]: temp = X.split()
        seq = list()                  #Creating empty list to append to it the int for float values from input
        for i in temp:                #Since time complexity of function is O(nlogn), this will not affect run time
            if i.isnumeric():         #much, and since merge sort is written for floats we have to input a seq of
                seq.append(int(i))    #floats or ints or combined
            else:
                seq.append(float(i))

In [6]: sorted_seq = Merge_Sort(seq)                    #Applying Merge Sort to the sequence

In [7]: deleter = sorted_seq[0]                         #Creating deleter to check if element is repeating
        result_seq = list()
        result_seq.append(deleter)                      #Since first element will be there, if repeated again will
        k = len(sorted_seq)                             #not append it to list as can be seen from the for loop
        for j in range(0,k):
            if j!=0:                                    #Since I have assigned deleter variable to sorted_seq[0]
                if sorted_seq[j] == deleter:            #and don't want to delete that
                    continue                            #Not adding already occured element, hence continue
                else:
                    deleter = sorted_seq[j]
                    result_seq.append(deleter)          #Appending newly occured element
        print(seq)                                      #printing original and sorted sequence
        print(result_seq)

        [-2.0, 3, 4.5, 7, 9, -6.0, 4.5, -2.0, 3]
        [-6.0, -2.0, 3, 4.5, 7, 9]
```

2nd method: Using dictionaries which uses the hash table structures.

Using a Hash table is much more efficient since the time complexity is O(n) for this implementation. Implementing hash table algorithm from scratch for inputs is difficult, but it can be done using the in-built structure of dictionaries which uses the technique of hashing (i.e creating key/value pairs to store elements in it)
Each element in a dictionary is given a key which can be used to access the value given to the key in O(1) from the dictionary.

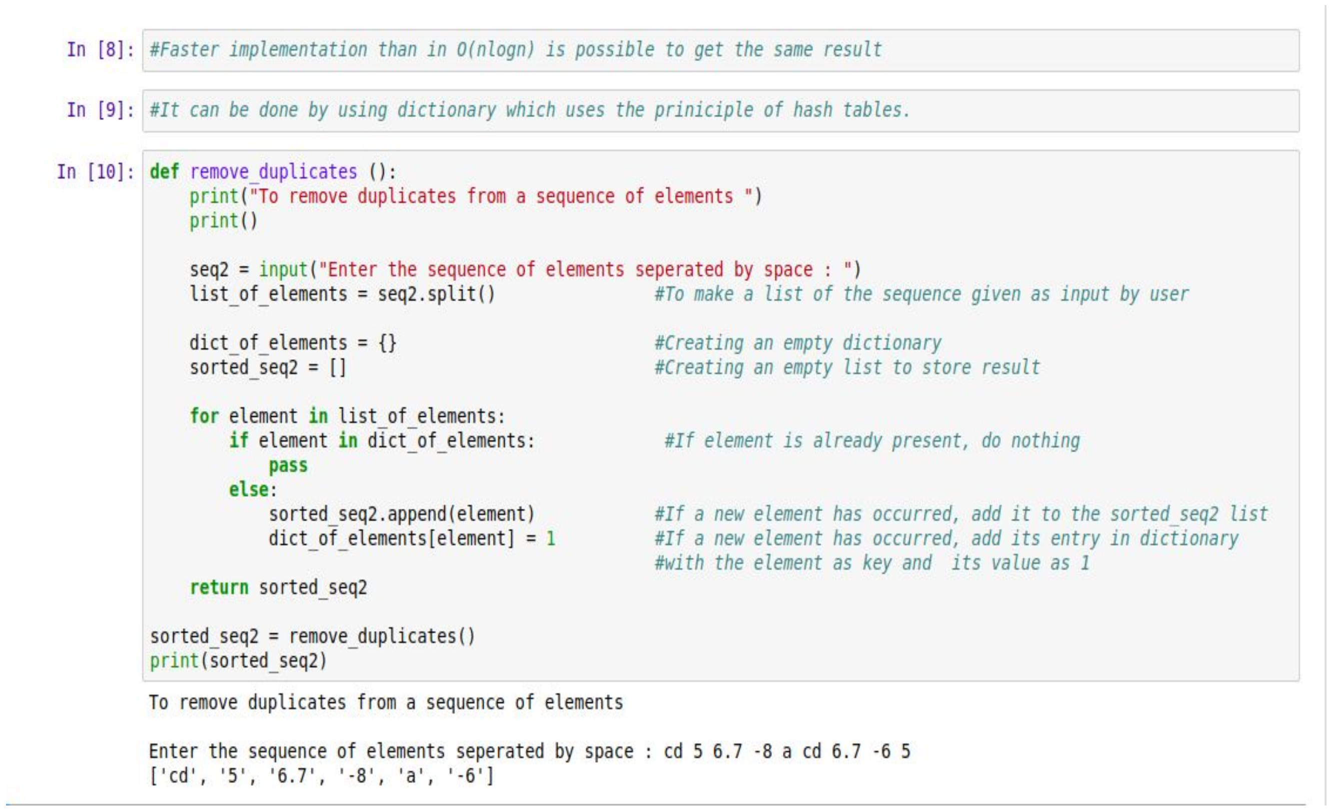
This method does not need to make any assumptions on the input, since we are not sorting, we are just inserting the entries in a dictionary or doing look-up on the dictionary every time a new entry form the list comes.

So we will initiate an empty dictionary and iterate through the list, we will add an element as key = element, value = 1 to the dictionary if it is already not present in it (which can be checked in O(1)) and therefore run time of this implementation will be O(n).

How hashing works :

- When adding entries to the table, we start with some slot, that is based on the hash of the key.
- If that slot is empty, the entry is added to the slot.
- If the slot is occupied, Python compares the hash AND the key of the entry in the slot against the hash and key of the current entry to be inserted respectively.
- If both match, then it thinks the entry already exists, gives up and moves on to the next entry to be inserted. If either hash or the key don't match, it starts probing.
- Probing just means it searches the slots by slot to find an empty slot.
- Python uses random probing. In random probing, the next slot is picked in a pseudo random order. The entry is added to the first empty slot. What is important is that the slots are probed until the first empty slot is found. The same thing happens for lookups, just starts with the initial slot i (where i depends on the hash of the key).
- If the hash and the key both don't match the entry in the slot, it starts probing, until it finds a slot with a match. If all slots are exhausted, it reports a fail.
- When a new dict is initialized it starts with 8 slots and the dict will be resized if it is two-thirds full. This avoids slowing down lookups.
- This implementation causes adding and lookup in a dictionary take O(1) in general

Following is attached the code for the same :



```
In [8]: #Faster implementation than in O(nlogn) is possible to get the same result

In [9]: #It can be done by using dictionary which uses the priniciple of hash tables.

In [10]: def remove_duplicates ():
             print("To remove duplicates from a sequence of elements ")
             print()

             seq2 = input("Enter the sequence of elements seperated by space : ")
             list_of_elements = seq2.split()             #To make a list of the sequence given as input by user

             dict_of_elements = {}                       #Creating an empty dictionary
             sorted_seq2 = []                            #Creating an empty list to store result

             for element in list_of_elements:
                 if element in dict_of_elements:          #If element is already present, do nothing
                     pass
                 else:
                     sorted_seq2.append(element)         #If a new element has occurred, add it to the sorted_seq2 list
                     dict_of_elements[element] = 1       #If a new element has occurred, add its entry in dictionary
                                                         #with the element as key and  its value as 1
             return sorted_seq2

         sorted_seq2 = remove_duplicates()
         print(sorted_seq2)

         To remove duplicates from a sequence of elements

         Enter the sequence of elements seperated by space : cd 5 6.7 -8 a cd 6.7 -6 5
         ['cd', '5', '6.7', '-8', 'a', '-6']
```

Question 4:

Given an array A of n integers in the range [0, n^2 - 1], describe a simple method for sorting A in O (n) time.

To solve this, we will use the Radix sort method. The idea of Radix Sort is to do digit by digit sort starting from least significant digit to most significant digit. Radix sort uses counting sort as a subroutine to sort. Counting sort is a linear time sorting algorithm that sorts in O(n+k) time when elements are in the range from 1 to k.

What radix sort does is basically it sorts the 1s place of all entries using counting sort, then it sorts the 10s place, 100s place and so on while preserving the order of lower sorts for equal values in any sorting iteration. (where the base of the radix sort is 10)

Counting Sort will work based on the base of the numbers (binary,octal,decimal etc) we are taking in. It will initialize an empty count array of length of base with all 0s.

Step 1: for j=0 to size
Find the total count of each unique digit in dth place of elements.
Store the count at jth index in the count array.

Step 2: for i=1 to max
Find the cumulative sum and store it in the count array itself.

Step 3: for j=1 to size
Restore the elements to array using the indices of the count array.
Decrease count of each element restored by 1.

Example of counting sort for base 10:

Original list: 1,3,5,1,6,8,9,8,6

Step 1: count array = [0,2,0,1,0,1,2,0,2,1]

Step 2: count array = [0,2,2,3,3,4,6,6,8,9]

Step 3: sorted array = [1,1,3,5,6,6,8,8,9]

Example of Radix algorithm with base 10 using counting sort:

Original, unsorted list:

170, 45, 75, 90, 802, 24, 2, 66

Step1: Arranging according to 1s place

170,90,802,2,24,45,75,66

Step2: Arranging according to 10s place

802,2,24,45,66,170,75,90

Step3: Arranging according to 100s place

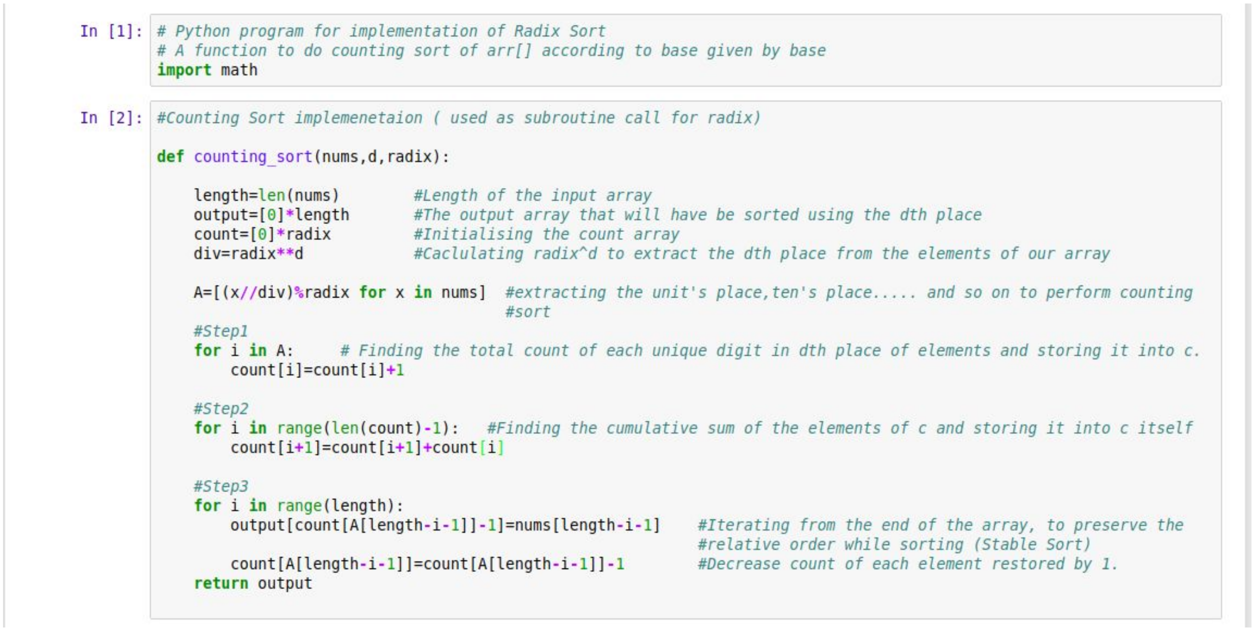
2,24,45,66,75,90,170,802

Time Complexity analysis : Let there be d digits in input integers. Radix Sort takes O(d*(n+b)) time where b is the base for representing numbers, for example, for the decimal system, b is 10.

In our question, since numbers are from 0 to $n^2 - 1$, we will choose the base to be n, which implies number of digits is 2 (q*n + r = number; q,r will be the digits).

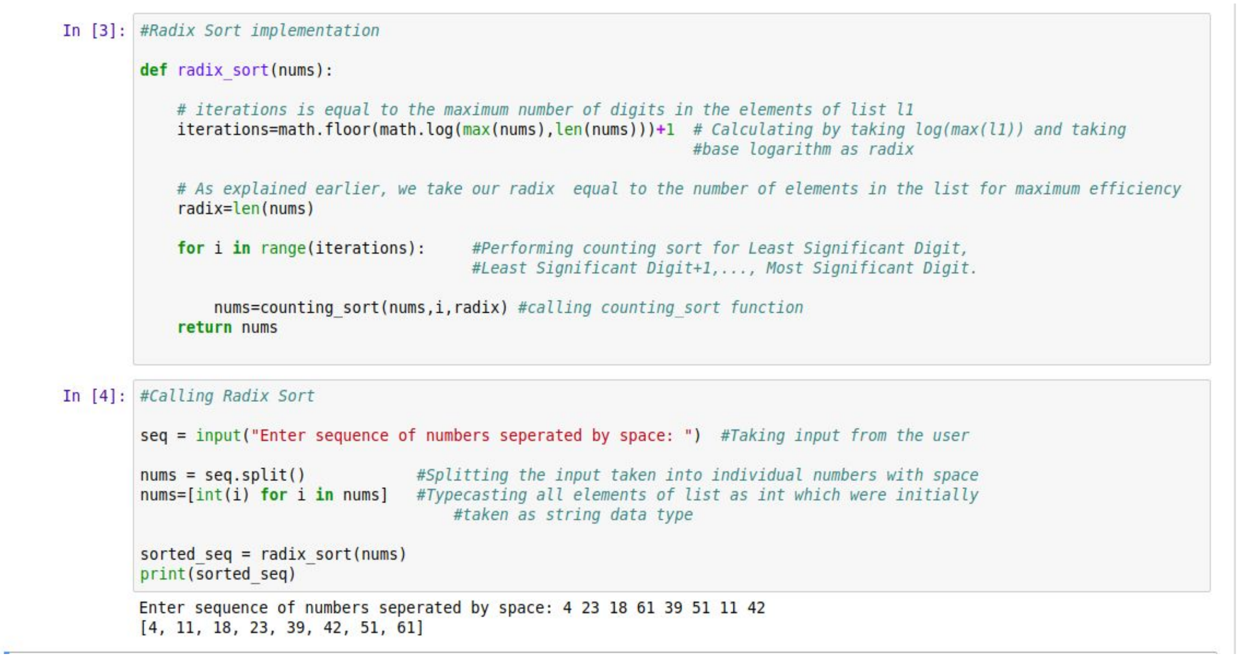
Therefore, Radix Sort will take O(2*(n+n)) = O(4n) = O(n) time to sort the list.

Following is the attached code for the same (have also attached a .ipynb and a downloaded .html in the submission)



```
In [1]: # Python program for implementation of Radix Sort
        # A function to do counting sort of arr[] according to base given by base
        import math
```

```
In [2]: #Counting Sort implemenetaion ( used as subroutine call for radix)

        def counting_sort(nums,d,radix):

            length=len(nums)        #Length of the input array
            output=[0]*length       #The output array that will have be sorted using the dth place
            count=[0]*radix         #Initialising the count array
            div=radix**d            #Caclulating radix^d to extract the dth place from the elements of our array

            A=[(x//div)%radix for x in nums]  #extracting the unit's place,ten's place..... and so on to perform counting
                                              #sort
            #Step1
            for i in A:     # Finding the total count of each unique digit in dth place of elements and storing it into c.
                count[i]=count[i]+1

            #Step2
            for i in range(len(count)-1):   #Finding the cumulative sum of the elements of c and storing it into c itself
                count[i+1]=count[i+1]+count[i]

            #Step3
            for i in range(length):
                output[count[A[length-i-1]]-1]=nums[length-i-1]    #Iterating from the end of the array, to preserve the
                                                                   #relative order while sorting (Stable Sort)
                count[A[length-i-1]]=count[A[length-i-1]]-1        #Decrease count of each element restored by 1.
            return output
```



```
In [3]: #Radix Sort implementation

        def radix_sort(nums):

            # iterations is equal to the maximum number of digits in the elements of list l1
            iterations=math.floor(math.log(max(nums),len(nums)))+1  # Calculating by taking log(max(l1)) and taking
                                                                    #base logarithm as radix

            # As explained earlier, we take our radix  equal to the number of elements in the list for maximum efficiency
            radix=len(nums)

            for i in range(iterations):     #Performing counting sort for Least Significant Digit,
                                            #Least Significant Digit+1,..., Most Significant Digit.

                nums=counting_sort(nums,i,radix) #calling counting_sort function
            return nums

In [4]: #Calling Radix Sort

        seq = input("Enter sequence of numbers seperated by space: ")  #Taking input from the user

        nums = seq.split()             #Splitting the input taken into individual numbers with space
        nums=[int(i) for i in nums]    #Typecasting all elements of list as int which were initially
                                           #taken as string data type

        sorted_seq = radix_sort(nums)
        print(sorted_seq)

        Enter sequence of numbers seperated by space: 4 23 18 61 39 51 11 42
        [4, 11, 18, 23, 39, 42, 51, 61]
```

Question 5:

Show that quicksort's best-case running time is Ω (n log n).

Quicksort involves 3 steps:

Choosing a pivot (O(1))

Comparing with all elements (O(length of array))

Partitioning into left and right and recursively using quick sort on both.

$\Rightarrow T(n) = kn + T(partition1) + T(partition2)$

To minimise the time complexity, we need to minimize the levels in the recursion, and at each level the partition with more elements will add more levels to recursion, so to minimise the recursive calls, both partitions should be equal which will minimise the number of recursive calls and hence give the fastest possible runtime. This happens when the chosen pivot is always the median of the specific partition.

So, in this case, our recursive equation becomes of the form :

$T(n) = kn + 2 * T(n/2)$, where n/2 is the floor function of n/2

Now, this is the same recursive equation as the one we got for merge sort, so by following the same procedure using the method of Mathematical Inductance, it can be shown again similarly that :

$T(n) = c1 \times n + c2 \times nlogn$, where c1,c2 are arbitrary constants

To show that : $T(n) = \Omega(nlogn)$

i.e to show that there exists a c such that :

$T(n) \geq c \times nlogn\, for\ all\ n \geq some\ m$

Choose c = c2;

$\Rightarrow c1 \times n + c2 \times nlogn \geq c2 \times nlogn\ which\ is\ true\, for\ all\ n \geq 0$

Therefore, by mathematical definition we have $T(n) = \Omega(nlogn)$