Date : 23rd - 09 - 2020
Morning Session : 9am – 11.00 PM
By ~ Rohan Kumar

Topics: Sorting

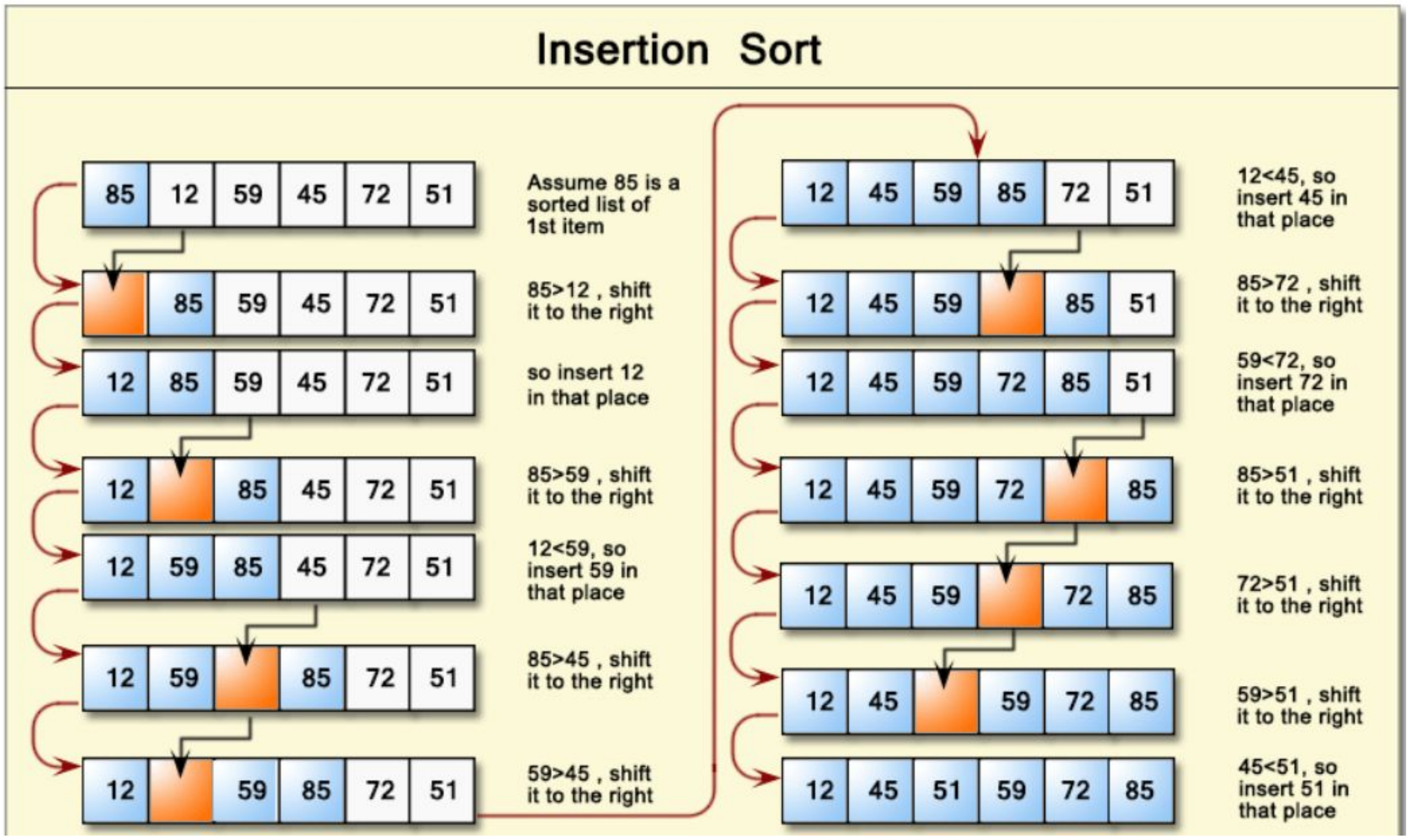
Insertion Sort : Insertion sort is a simple sorting algorithm that works similar to the way you sort playing cards in your hands. The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part.
Algorithm

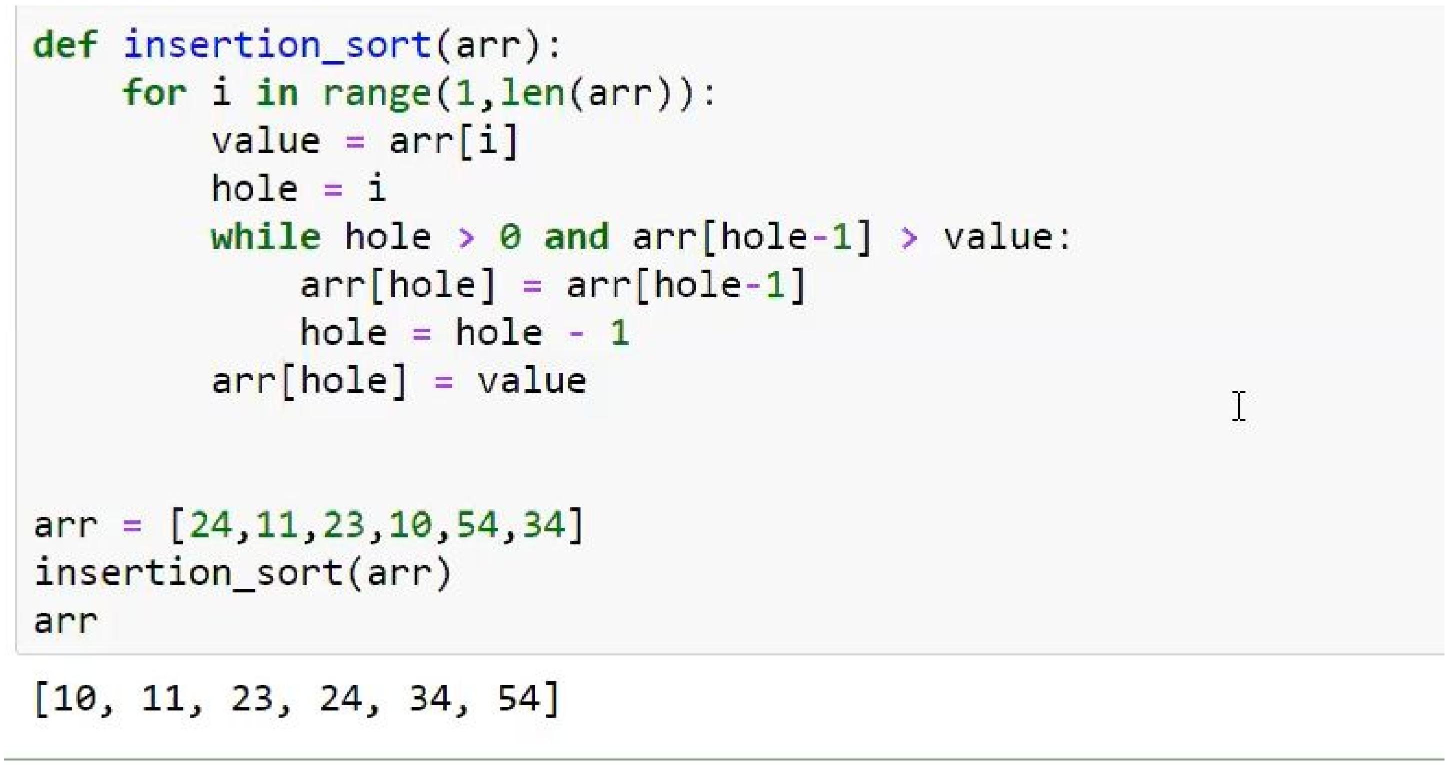
To sort an array of size n in ascending order:

1: Iterate from arr[1] to arr[n] over the array.

2: Compare the current element (key) to its predecessor.

3: If the key element is smaller than its predecessor, compare it to the elements before. Move the greater elements one position up to make space for the swapped element.


Insertion Sort
Assume 85 is a sorted list of 1st item
85>12 , shift it to the right
so insert 12 in that place
85>59 , shift it to the right
12<59, so insert 59 in that place
85>45 , shift it to the right
59>45 , shift it to the right
12<45, so insert 45 in that place
85>72 , shift it to the right
59<72, so insert 72 in that place
85>51 , shift it to the right
72>51 , shift it to the right
59>51 , shift it to the right
45<51, so insert 51 in that place



```python
def insertion_sort(arr):
    for i in range(1,len(arr)):
        value = arr[i]
        hole = i
        while hole > 0 and arr[hole-1] > value:
            arr[hole] = arr[hole-1]
            hole = hole - 1
        arr[hole] = value


arr = [24,11,23,10,54,34]
insertion_sort(arr)
arr
```

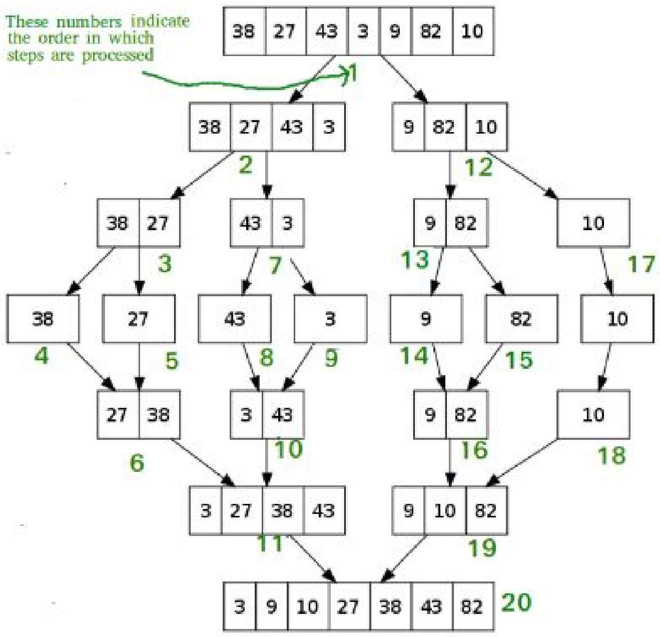
```
[10, 11, 23, 24, 34, 54]
```

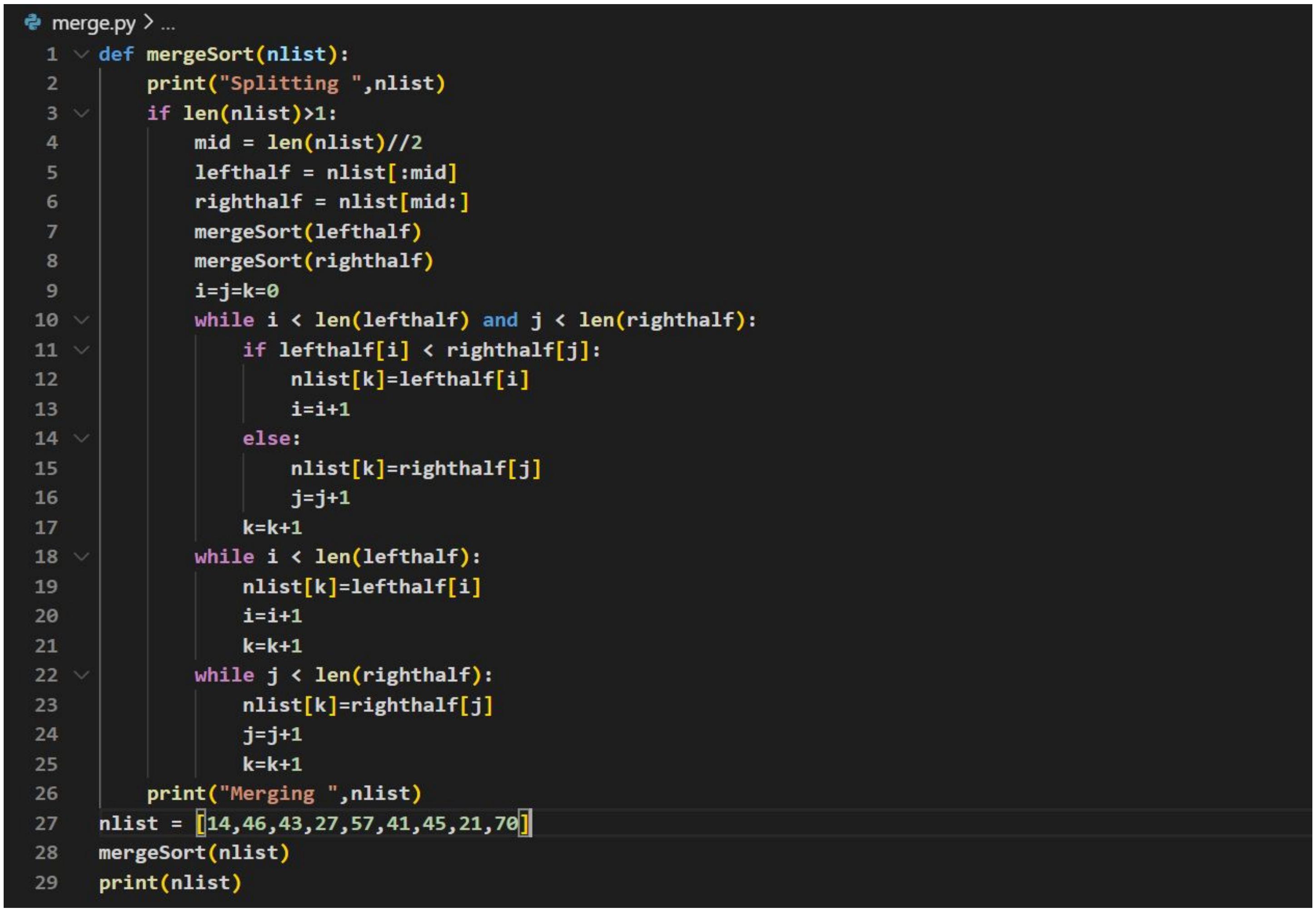
Time Complexity: O(n*2)

Space complexity : O(1)

Merge Sort : Merge Sort is a Divide and Conquer algorithm. It divides the input array in two halves, calls itself for the two halves and then merges the two sorted halves. The merge() function is used for merging two halves. The merge(arr, l, m, r) is a key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted subarrays into one.

```
MergeSort(arr[], l,  r)
If r > l
     1. Find the middle point to divide the array into two halves:
             middle m = (l+r)/2
     2. Call mergeSort for first half:
             Call mergeSort(arr, l, m)
     3. Call mergeSort for second half:
             Call mergeSort(arr, m+1, r)
     4. Merge the two halves sorted in step 2 and 3:
             Call merge(arr, l, m, r)
```




merge.py > ...

```python
def mergeSort(nlist):
    print("Splitting ",nlist)
    if len(nlist)>1:
        mid = len(nlist)//2
        lefthalf = nlist[:mid]
        righthalf = nlist[mid:]
        mergeSort(lefthalf)
        mergeSort(righthalf)
        i=j=k=0
        while i < len(lefthalf) and j < len(righthalf):
            if lefthalf[i] < righthalf[j]:
                nlist[k]=lefthalf[i]
                i=i+1
            else:
                nlist[k]=righthalf[j]
                j=j+1
            k=k+1
        while i < len(lefthalf):
            nlist[k]=lefthalf[i]
            i=i+1
            k=k+1
        while j < len(righthalf):
            nlist[k]=righthalf[j]
            j=j+1
            k=k+1
    print("Merging ",nlist)
nlist = [14,46,43,27,57,41,45,21,70]
mergeSort(nlist)
print(nlist)
```

Merge-sort with Transylvanian-saxon (German) folk dance

Overall time complexity of Merge sort is O(nLogn). It is more efficient as it is in worst case also the runtime is O(nlogn)

The space complexity of Merge sort is O(n). This means that this algorithm takes a lot of space and may slower down operations for the last data sets.

Merge Sort : Quick Sort is also based on the concept of Divide and Conquer, just like merge sort. But in quick sort all the heavy lifting(major work) is done while dividing the array into subarrays, while in case of merge sort, all the real work happens during merging the subarrays. In case of quick sort, the combine step does absolutely nothing.

It is also called partition-exchange sort. This algorithm divides the list into three main parts:

1. Elements less than the Pivot element
2. Pivot element(Central element)
3. Elements greater than the pivot element

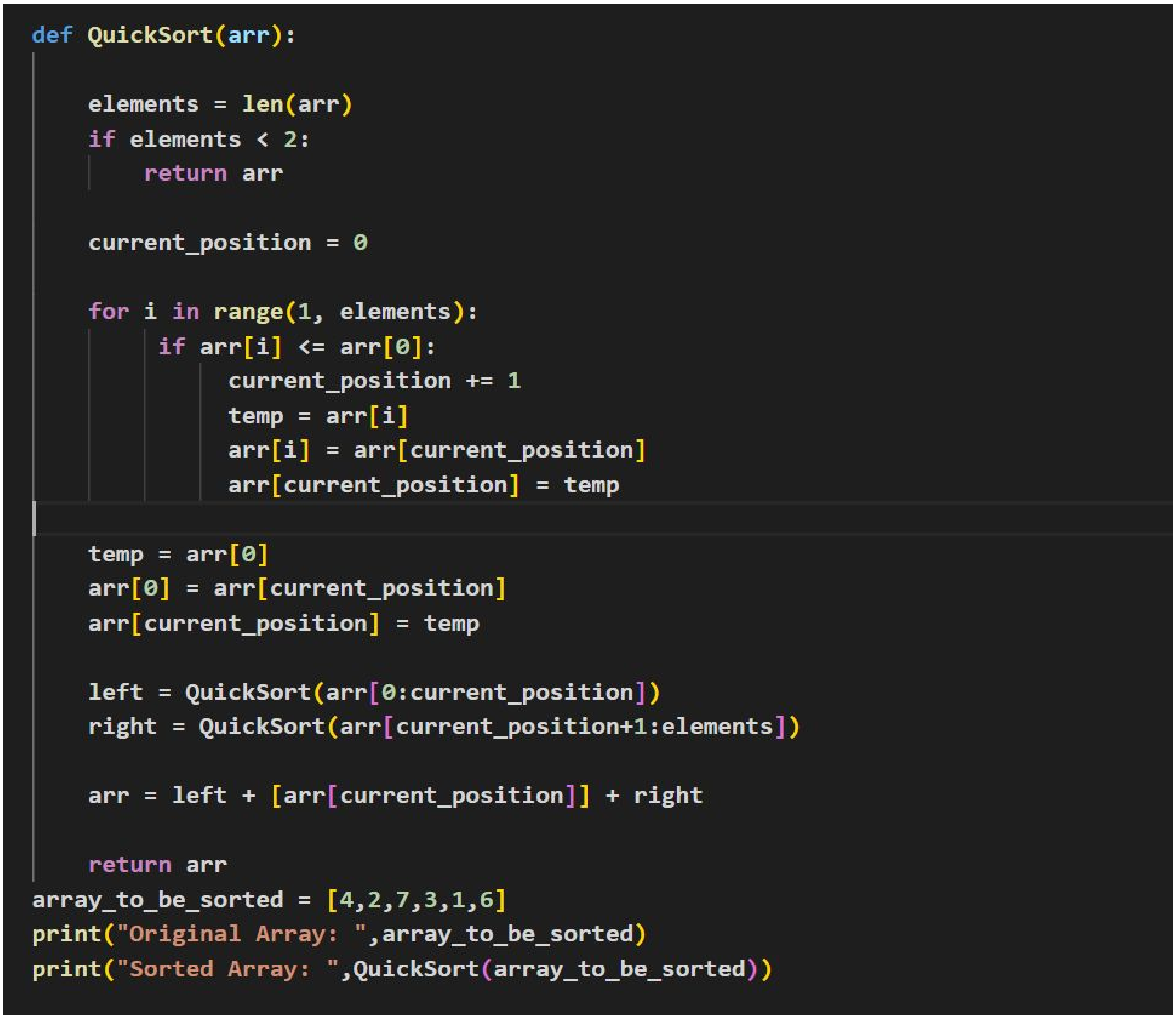
Pivot element can be any element from the array, it can be the first element, the last element or any random element. In this tutorial, we will take the rightmost element or the last element as pivot.

How Quick Sorting Works?

Following are the steps involved in quick sort algorithm:

1. After selecting an element as **pivot**, which is the last index of the array in our case, we divide the array for the first time.
2. In quick sort, we call this **partitioning**. It is not simple breaking down of array into 2 subarrays, but in case of partitioning, the array elements are so positioned that all the elements smaller than the **pivot** will be on the left side of the pivot and all the elements greater than the pivot will be on the right side of it.
3. And the **pivot** element will be at its final **sorted** position.
4. The elements to the left and right, may not be sorted.
5. Then we pick subarrays, elements on the left of **pivot** and elements on the right of **pivot**, and we perform **partitioning** on them by choosing a **pivot** in the subarrays.

https://youtu.be/ywWBy6J5gz8



```python
def QuickSort(arr):

    elements = len(arr)
    if elements < 2:
        return arr

    current_position = 0

    for i in range(1, elements):
         if arr[i] <= arr[0]:
              current_position += 1
              temp = arr[i]
              arr[i] = arr[current_position]
              arr[current_position] = temp

    temp = arr[0]
    arr[0] = arr[current_position]
    arr[current_position] = temp

    left = QuickSort(arr[0:current_position])
    right = QuickSort(arr[current_position+1:elements])

    arr = left + [arr[current_position]] + right

    return arr
array_to_be_sorted = [4,2,7,3,1,6]
print("Original Array: ",array_to_be_sorted)
print("Sorted Array: ",QuickSort(array_to_be_sorted))
```

MCQ 1:

1. Time complexity of merge sort is

a. O(n)

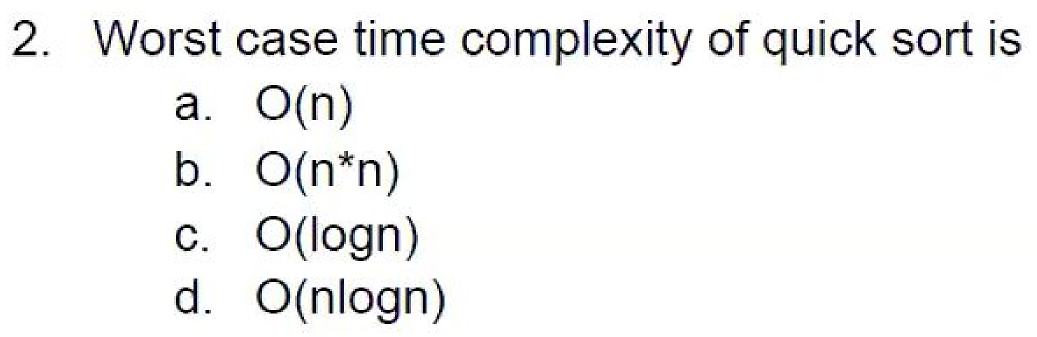
b. O(n*n)

c. O(logn)

d. O(nlogn)

Answer: D, O(nlogn)

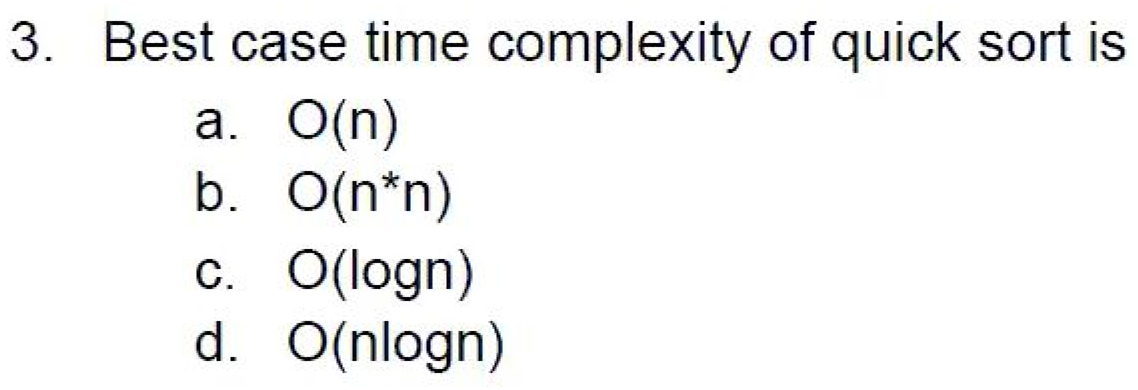
MCQ 2:



2. Worst case time complexity of quick sort is
    a. O(n)
    b. O(n*n)
    c. O(logn)
    d. O(nlogn)

Answer: B, O(n * n)

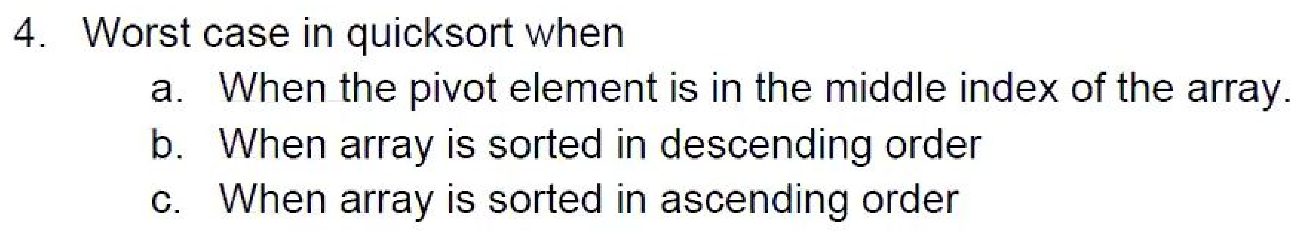
MCQ 3:



3. Best case time complexity of quick sort is
    a. O(n)
    b. O(n*n)
    c. O(logn)
    d. O(nlogn)

Answer: D , O(nlogn)

MCQ 4:



4. Worst case in quicksort when
    a. When the pivot element is in the middle index of the array.
    b. When array is sorted in descending order
    c. When array is sorted in ascending order

Answer: C, When array is Sorted in Ascending Order.