Quicksort

Sorting Algorithm - Divide and Conquer

- Not a stable sorting algorithm
- Inplace Sorting algorithm
- Quicksort is widely used in practical/real life applications

Stable Sorting Algorithm :

20	3	1	20’	10	34	9	#Values
0	1	2	3	4	5	6	#Index

After sorting of above array :

1	3	9	10	20’	20	34	#Values
0	1	2	3	4	5	6	#Index

Above output represents that it is not a stable sorting algorithm

QuickSort Implementation :

QuickSort(arr,0,6)

x

35	20	30	10	50	69	70	#Values
0	1	2	3	4	5	6	#Index

return -> 4th index

QuickSort(arr,0,3) -> Left hand side

10	20	30	35	#Values
0	1	2	3	#Index

QuickSort(arr,5,6) -> Right hand side

69	70	#Values
0	1	#Index

Final overall result

10	20	30	35	50	69	70	#Values

And we see that above result is a sorted array

Observations :

1. Exact location of that pivot element will be returned after execution of a code.

2. Left hand side elements are lesser than the pivot element and right hand side elements are greater than the pivot elements.

3. The position that is returned for the pivot element indirectly indicates that the pivot element is the nth smallest element in an array.

Implementation of Partition Algorithm :

```
def Partition(arr,p,q):     #arr in the list form, p is a pivot element
        x=arr[p]            #Pivot Element
        i=p
```

```
    for j in range(i+1,q+1):                    #O(n)
        if (arr[j]<=arr[p]):   #70<=50 False
            i=i+1
            swap(a[i],a[j])
    swap(arr[p],arr[i])
    return i
```

Implementation of Recursive QuickSort :

```
def QuickSort(arr,p,q):
    # small problem
    if(p == q):    # Array contains single element
        return arr[p]        # O(1) - constant time complexity
    # big problem
    else:
        m = Partition(arr,p,q)          # O(n)
        QuickSort(arr,p,m-1)            # Left hand side T(m-p)
        QuickSort(arr,m+1,q)            # Right hand side T(q-m)
```

Randomized QuickSort : When we choose the pivot element randomly from an array then it is known as Randomized Quicksort.

Implementation of Randomized QuickSort :

```
Partition(arr,p,q):
```

```
r = RG(arr,p,q)

swap(arr[r],arr[p])

x = arr[p]                    # Pivot Element

i = p

for(int j = i+1; j <=q ;j++):

        if(arr[j] <= arr[p]):              # 70 <= 50 False

                i = i + 1

                swap(a[i],a[j])

swap(arr[p],arr[i])

return i
```

Note : Randomized Quicksort gives better results as compared to Quicksort because here our pivot element is not fixed.

Recurrence Relation Of QuickSort Algorithm :

T(n) = O(1) n = 1 ; small problem

O(n) + T(m-p) + T(q-m) n > 1 ; big problem

Best Case Scenario :

Best case means that the position of the pivot element divides an array in such a way that there should be almost equal number of elements present in left as well as right hand side. Then, we consider that type of case as the best case scenario for a quicksort algorithm.

T(n) = O(n) + T(n/2) + T(n/2)

= O(n) + 2T(n/2)

Using master's theorem we can easily solve the above recurrence

relation, $n^{\log_b a} = n$ f(n) = n

O(f(n)logn) = O(n logn)

So, O(nlogn) is the best case time complexity of the Quicksort

algorithm. Worst Case Scenario :

T(n) = O(n) + T(1) + T(n-1)

= T(n-1) + n

Using the substitution method, and finally showed you that overall time complexity of Quicksort in worst case scenario is :

$T(n) = O(n^2)$

Note : Whenever an array is almost sorted or completely sorted, it's never recommendable to use quicksort in these types of scenarios because if we use quicksort in these situations this will give us worst case time complexity of $O(n^2)$.

If an array is almost sorted, it's recommendable to use insertion sort to sort an array completely because this will give us an overall time complexity of O(n) in that particular case.

Some of the questions asked in interviews related to QuickSort :
Problem 1 :

In quicksort sorting of n numbers the n/7th smallest element is selected as the pivot element using $O(n^2)$ time complexity algorithm. Then what will be the worst case time complexity of the quicksort.

Solution :

T(n) = n^2 + n + T(n/7 - 1) + T(n - n/7)

= n ^ 2 + T(n/7) + T(6n/7)

Now, solve this recurrence relation with the help of an Recursive tree method and after solving the above recurrence relation with the help of recursive tree method, we get an overall time complexity of

O(n^2)

Problem 2 :

In quicksort sorting of n numbers the 25th largest element is selected as the pivot element using $O(n^2)$ as the time complexity. Then what will be the worst case time complexity of the above algorithm.

Solution :

$T(n) = n^2 + n + T(n-25) + T(24)$

n^2 - choosing the pivot element

n - Partition of algorithm

T(n-25) - Left Part

T(24) - Right Part

$T(n) = n^2 + T(n-25)$ (To be done by you)
After solving of this particular recurrence relation, we get an overall time complexity of $O(n^3)$

NOTE :

- **If in question it is mentioned about the nth smallest element then take the position of pivot element from the left hand side of an array just we**

consider in Problem number 1.

- **If in question it is mentioned about the nth largest element then take the position of pivot element from the right hand side of an array like we consider in Problem number 2.**