Quick Sort

Friday, September 8, 2023   10:46 AM

The main idea of quick sort is to sort the array by placing the "pivot" (pivot is just a value of the array) at its actual positio
been placed at its actual position we need to sort the data on left and right side of the pivot. So the sub problems need to
ranging from **startIndex to pivotIndex-1** and from **pivotIndex+1 to LastIndex**. The main question is how we will be able to
its actual position in an efficient way.

Let's discuss the idea:
Problem: Given an array "**Arr[]**" of "**N**" elements and we want to store the **pivot** (any value of the array e.g., value of last in
position in a newly created temporary array "**temp[]**".
Abstract Idea:
1: Select last index of Arr[] as pivot.
2: Create three iterators namely (**i, j, and k**).
3: Initialize the iterators "i and j" with the first and last indices of the **temp[]** respectively and the iterator "k" will be initializ
index of **Arr[].**
4: Now iterate from k=1 to N, and in every iteration you need to compare the value at index "k" with pivot.
    There are three possibility i.e., Arr[k] is less than pivot, or Arr[k] is greater than pivot, or Arr[k] is equals to the pivot.
    => if Arr[k] is less than pivot, then store the value of index "k" at index "i" in temp[]. Increment both the iterators i.e., "i"
    => if Arr[k] is greater than the pivot, then store the value of index "k" and index "j" in temp[]. Increment "k" and decrem
    => if Arr[k] is equals to the pivot, then simply increment the value of iterator "k"
5: when the loop terminates, place the pivot at index "i" in temp[] i.e., temp[i] = Arr[LastIndex]

In this way the pivot will be placed at its actual position i.e., (data values will be lesser than the pivot on the left side of piv
the pivot on the right side of the pivot). Overall time required for this task is linear i.e., O(N). Since we are utilizing extra a
we know very well that it is **not an in-place** approach.

Problem: Place the pivot at its actual position in linear time by using **in-place** approach.
Main Idea: The main idea will remains the same with minor changes.

```
    1: Select last index value as pivot i.e., pivot = Arr[right]
    2: Create two iterators "i and j" and initialize them with the indices first and (right-1) of Arr[] respectively.
      ==> why we are creating these iterators? Basically we will compare ith and jth index values with the pivot and swap
    required, to make sure that the data on the left and
         right side of the pivot will remain smaller and greater than the pivot respectively.
    3: iterate until "i" is less than "j":  while (i < j )
        3.1: iterate until the values at index "i" are smaller than pivot AND "i" is less than "j": while (Arr[i] <= pivot AND i <
            3.1.1: Increment "i" by 1 to move on to the next index because value at index "i" is already smaller than pivot
            move this value to any other position:  i = i+1
        3.2: Now iterate until values at index "j" are greater than pivot AND "i" is less than "j" i.e., while(Arr[j] > pivot AND
            3.2.1: Decrement "j" by 1 to move on to the previous index because value at "j" is already graeter than pivot s
            move this value to any other position:  j = j-1
        3.3: if "i" is less than "j" then swap the data of index "i" with "j" and update both indices: if(i <j )
            3.3.1:     swap(Arr[i], Arr[j]) ,   i = i+1,   j= j-1
    4: when the loop terminates you need to swap LastIndex value with ith index value to place the pivot at actual positio
    Arr[right])
    5: return ith index to the calling function since you need to return position of the pivot.
```

It resembles **pre order traversal** i.e., place the pivot at its actual position and then make a recursive call to left and right s
ranging from **start to pivotIndex-1 and from pivotIndex+1 to end**.

Algorithm of Quick Sort

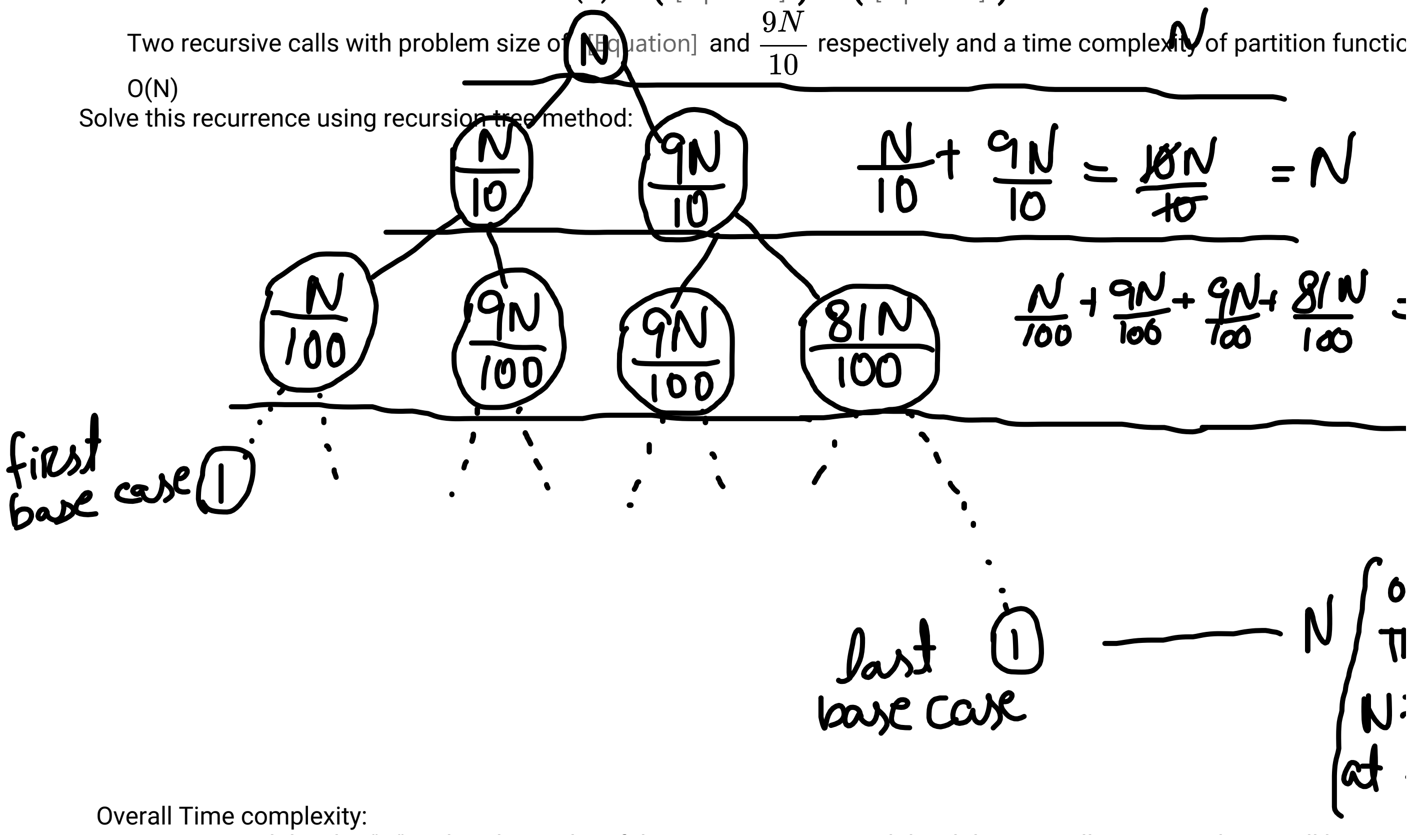
```
QuickSort(Arr[], left, right)
{
    if(left < right)
    {
        pivot_index = partition(Arr, left, right)
        QuickSort(Arr, left, pivot_index-1)
        QuickSort(Arr, pivot_index+1, right)
    }
}
Partition (Arr[], left, right)
{
//We can select any index value as pivot e.g., generate a random index in the range (left to right) and then swap that rand
last index.
    pivot = Arr[right]
    i = left
    j = right -1
    while(i < j)
    {
        while(Arr[i] <= pivot AND i < j)
            i++
        while(Arr[j] > pivot AND i < j)
            j--
        If(i < j)
            swap(Arr[i], Arr[j])
            i++
            j--
    }
    if(Arr[i] >= Arr[right])
         swap(Arr[i], Arr[right])
         return I
    Else
         swap(Arr[i+1], Arr[right])
        return i+1
}
```

In quick sort algorithm, we cannot guarantee that the data split will be equal because the amount of data being split for tl
depending upon pivot_Index. In ideal scenario, the data split will be equal just like merge sort and we know that time com
is N*logN but Let's consider a scenario of very bad data split e.g., 10% and 90% i.e., if there are total "N" elements in the a
amount of data being passed to one of the sub-problem will be $\frac{N}{10}$ and the other one will be of $\frac{9N}{10}$ elements.
Now if we want to write the recurrence for this particular scenario it will be like

T(N) = T([Equation]) + T([Equation]) + N

Two recursive calls with problem size of [Equation] and $\frac{9N}{10}$ respectively and a time complexity of partition functic
O(N)

Solve this recurrence using recursion tree method:



Overall Time complexity:

Cost at each level is "N" and we know that if the cost is same at each level then overall time complexity will be:

Height of Tree * cost at Each level

We will consider the height of tree of extreme right branch of the tree because it is the longest depth.

Height of this tree is [Equation] **(How?)**

Height of tree is dependent upon the decay factor. If there is an exponential decay with dividing factor of 2 then he
[Equation]

In this particular example the dividing factor of the longest branch is (10/9) because [Equation] **is equivalent to**

Overall Time complexity : [Equation] but we know that the base of log doesn't matter because they eqauivalen
constant multiple. So we will simply write [Equation] instead of [Equation]

Time complexity of quick sort for such a bad split of data is [Equation]

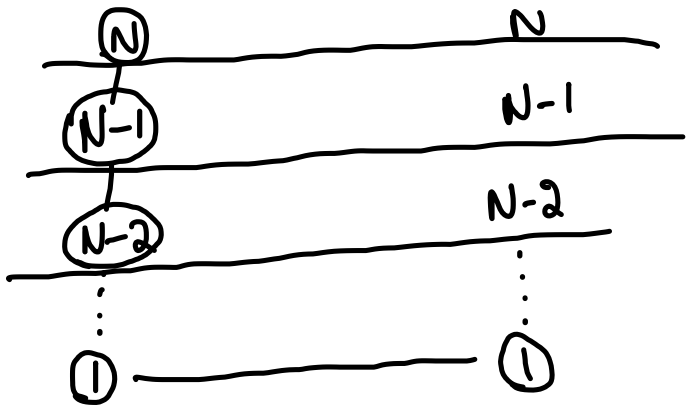
Let's consider another very interesting scenario of data split

Assume that the **input array is already sorted** and you are selecting the last index as pivot. So every time the pivot
actual position. As per the algorithm, the data of sub-problems will be ranging from (startingIndex to pivotIndex-1)
(pivotIndex+1 to lastIndex).

If the array is already sorted and we are selecting the last index as pivot then **data of one of subproblem will be N-1** there will no element in the other subproblem. It means that we can skip one recursive call because it will be return [Equation] by the base case.
Recurrence of this particular scenario:

[Equation]

Basically we are left with only one recursive call with [Equation] elements and partition function will take linear tir



Here we only need vertical accumulation which is:

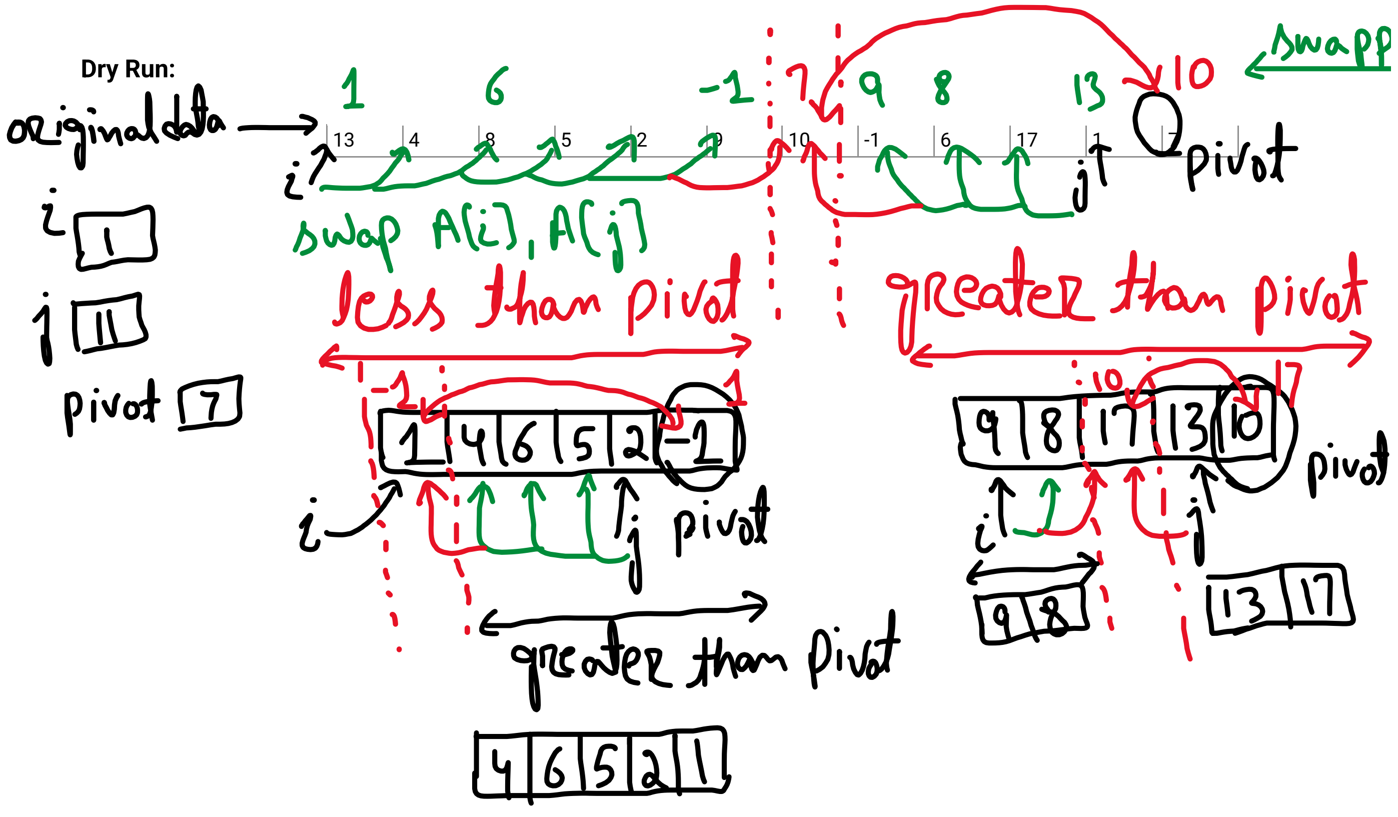
[Equation]

It is basically a summation of arithmetic series whose formula is : [Equation] = [Equation]

Overall time complexity for this particular scenario i.e., (for sorted array) = [Equation]

So, one interesting fact is, time complexity of quick sort for unsorted array is [Equation] but it is [Equation] **for sorte** [Equation] is because of unbalanced data split where we are left with [Equation] elements in each sub-problem. If sc manage a balanced data split then we can reduce the time complexity to [Equation] We can achieve this by selecting because median is the central value of data and it will produce a balanced split of data. **But the question is how we car the median as a pivot ?**

Dry Run:



Complete the dry run