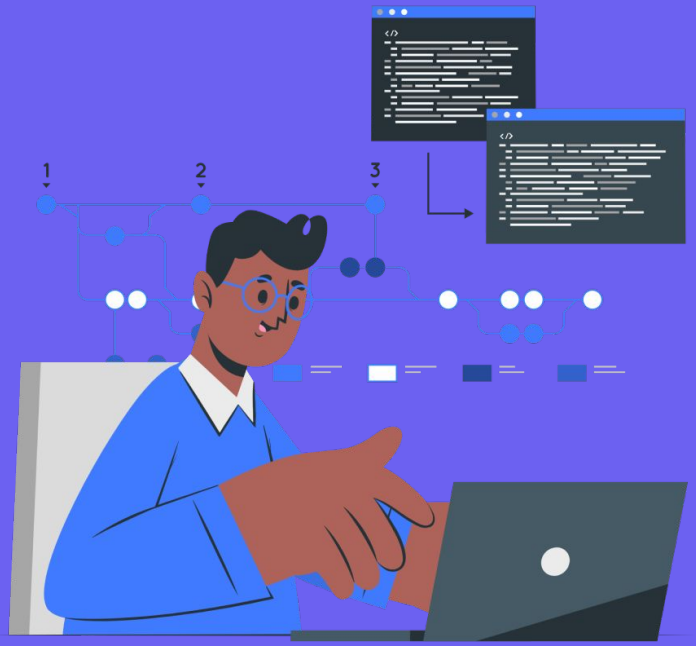
Algorithms: Quick sort -Part-1

Relevel
by Unacademy



Topics

- Quick Sort
- Randomised Quick Sort
- Analysis
- Applications

Quick Sort

Partition Algorithm

Partition algorithm works by choosing a 'pivot' element from the array and splitting (partitioning) the array into two sub-arrays, according to that we can move the element into left and right of the sub array whether they are less than or greater than the pivot thats why it is called partition-exchange sort.

Brute Force Approach: We could implement a nested loop finding all possible pairs of elements and adding them.

Two Pointer Technique: One pointer starts from beginning and other from the end and they proceed towards each other

Due to this, quicksort is also called "Partition Exchange" sort. Similar to merge sort, This algorithm also falls under the category of divide and conquer problem solving approach, because it first divide the array into two sub array recursively solve the array (conquer)

Let us look at the below problem, **Nuts & Bolts Problem (Lock & Key problem)**

Let's consider a simple example to begin with. If you have a “n”nut and bolt collection of distinct size and each nut has exactly one matching bolt of the same width,how should we match each nut to its corresponding bolts by comparing nuts to bolts? Constraint is you should not match nuts with nuts or bolts to bolts.

This is typically a sorting problem. consider the two arrays of n numbers such that one array list is a permutation of the other.

Here we can you use the Divide and conquer approach

Divide and conquer is nothing but dividing large problems into small problems and then solve each subproblem independently, finally combining the solutions of small subproblems for a large one.

Code - https://jsfiddle.net/saravananslb/fzdL14xh/

Explanation:

This algorithm first performs a partition by picking last element of bolts array as pivot, rearrange the array of nuts and returns the partition index ‘i’ such that all nuts smaller than nuts[i] are on the left side and all nuts greater than nuts[i] are on the right side. Next using the nuts[i] we can partition the array of bolts. Partitioning operations can easily be implemented in O(n). This operation also makes nuts and bolts array nicely partitioned. Now we apply this partitioning recursively on the left and right sub-array of nuts and bolts.

Here for the sake of simplicity, we have always chosen the last element as pivot. We can do randomized quicksort too.

Quick sort is a most efficient sorting algorithm and it works on partitioning (splitting) an array of data into smaller arrays using a technique called Divide and conquer. An array is partitioned into two arrays by partitioning mid value as pivot, one of which holds values smaller than the pivot value and another array holds values greater than the pivot value and it will form two subarrays.

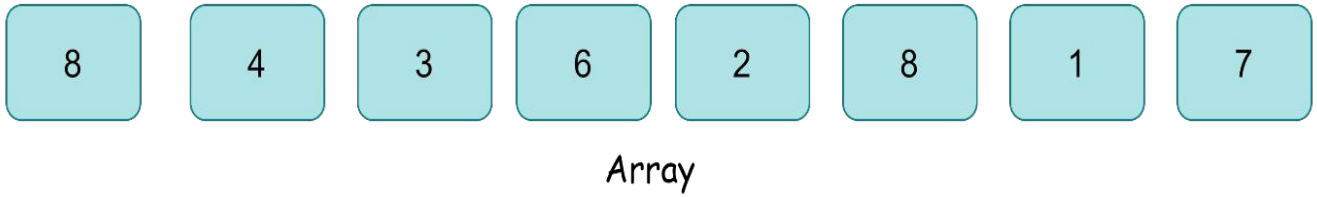
Again it will partition the subarray and then calls itself recursively twice(lower pivot subarray and higher pivot subarray) to sorting the two resulting sub arrays. This algorithm is efficient for large-sized data sets as its average and worst-case complexity are $O(n^2)$, respectively.

Choosing pivot in QuickSort

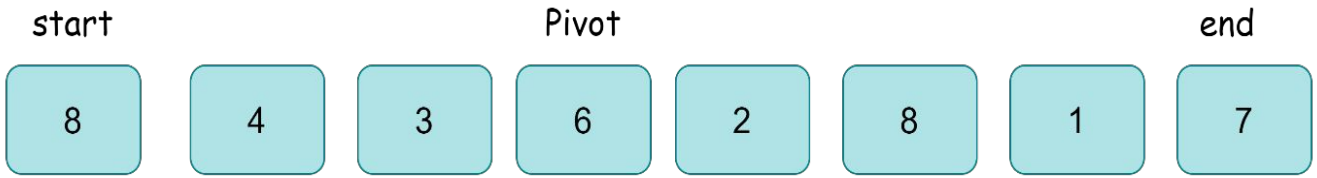
- There are multiple ways to choose a pivot in quicksort they are,

- Selecting starting element as pivot
- Selecting Middle element as pivot
- Selecting Last element as pivot
- Picking a random number from the given list of elements as the pivot (Also called as randomised quicksort)

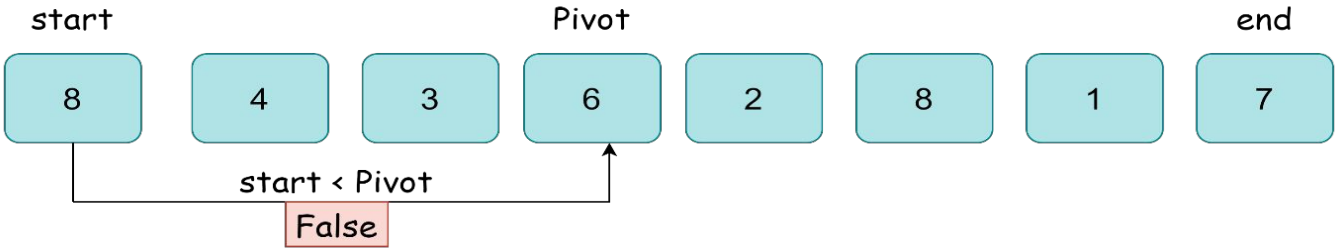
Let us consider the array of element 8, 4, 3, 6, 2, 8, 1, 7 and Middle element as Pivot



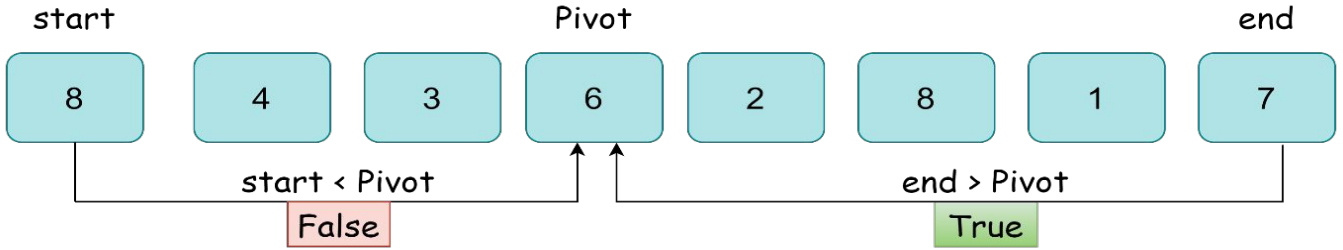
First element in the array as start and last element in the array as end



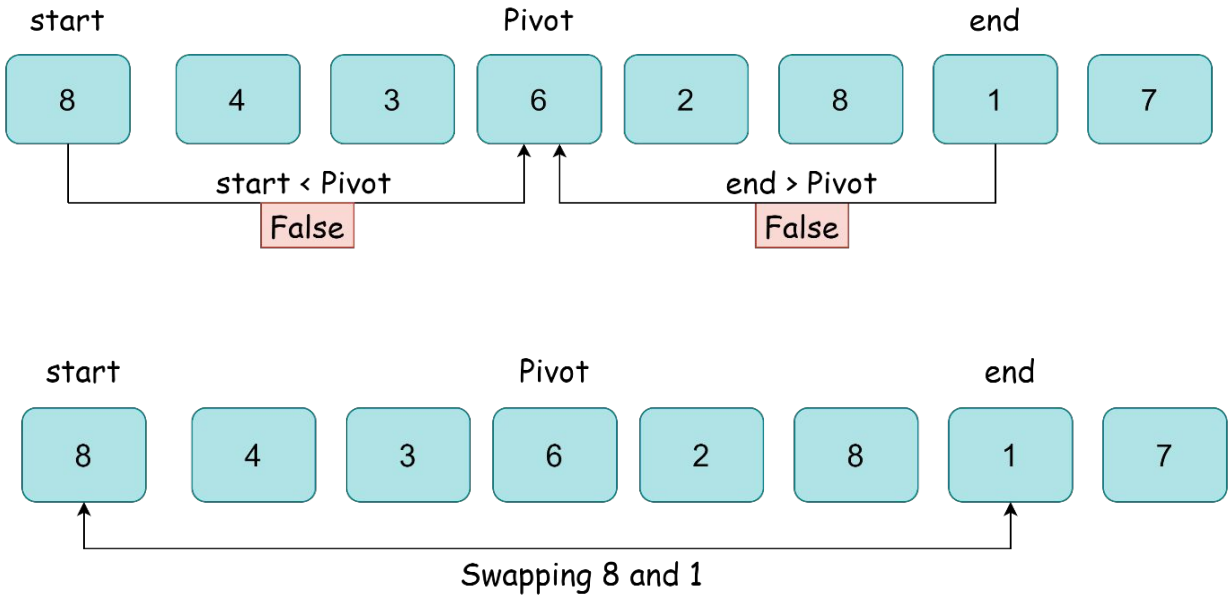
First it will check if the start is less than pivot and if the condition is false then move to right side of the pivot



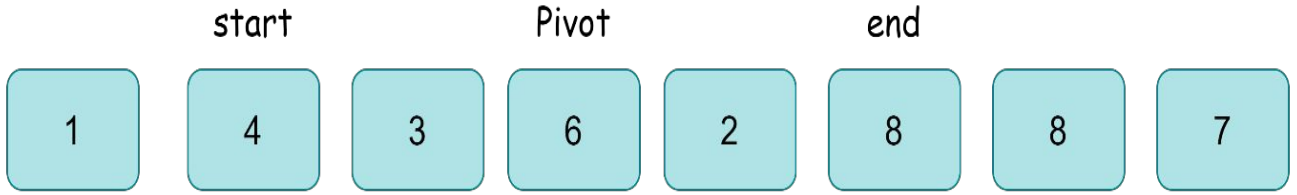
Check if the end is greater than pivot and if the condition is true decrease the end pointer



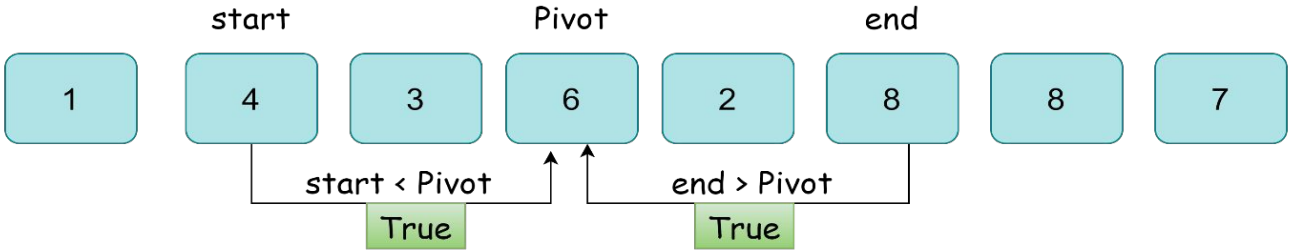
Again, check if the end is greater than pivot and if the condition is false then swap start and end index elements



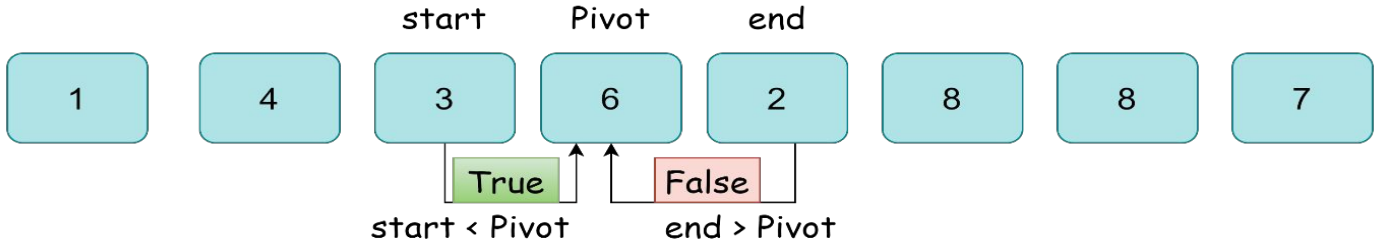
We will get a new array in the below and increase the start and decrease the end



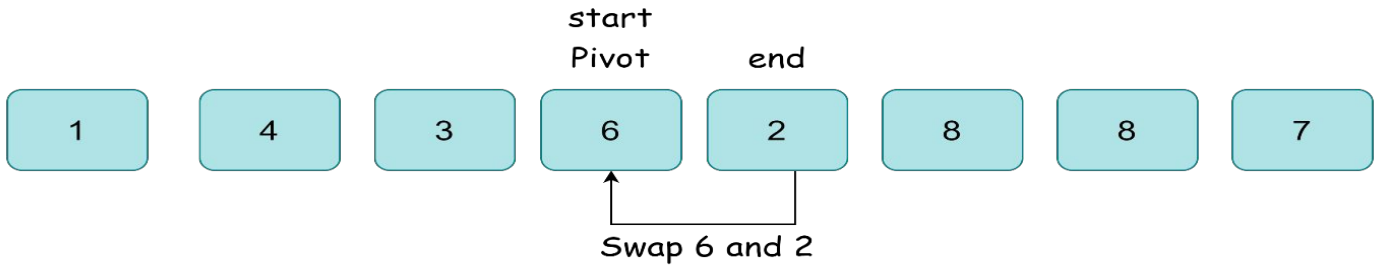
Again, check the condition if the start is less than pivot and end is greater than pivot both the conditions are met then increase the start and decrease the end



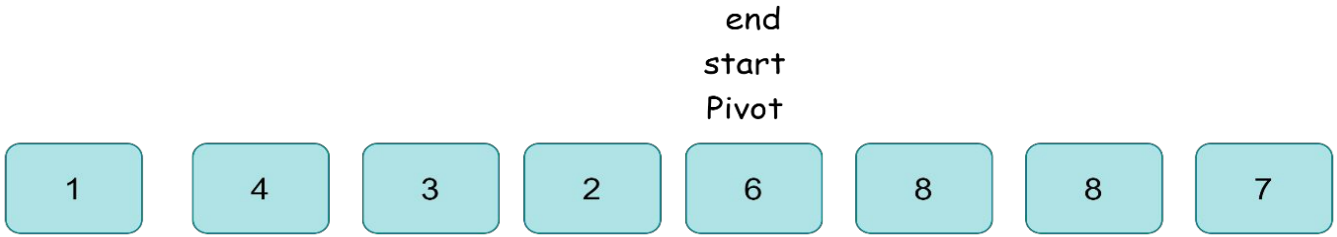
Again, check the condition if the start is less than pivot and end is greater than pivot one of the condition is met then increase the start



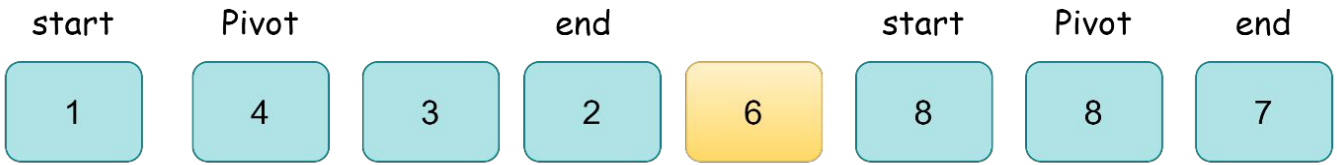
Swap the start and end values and increase the start and decrease the end



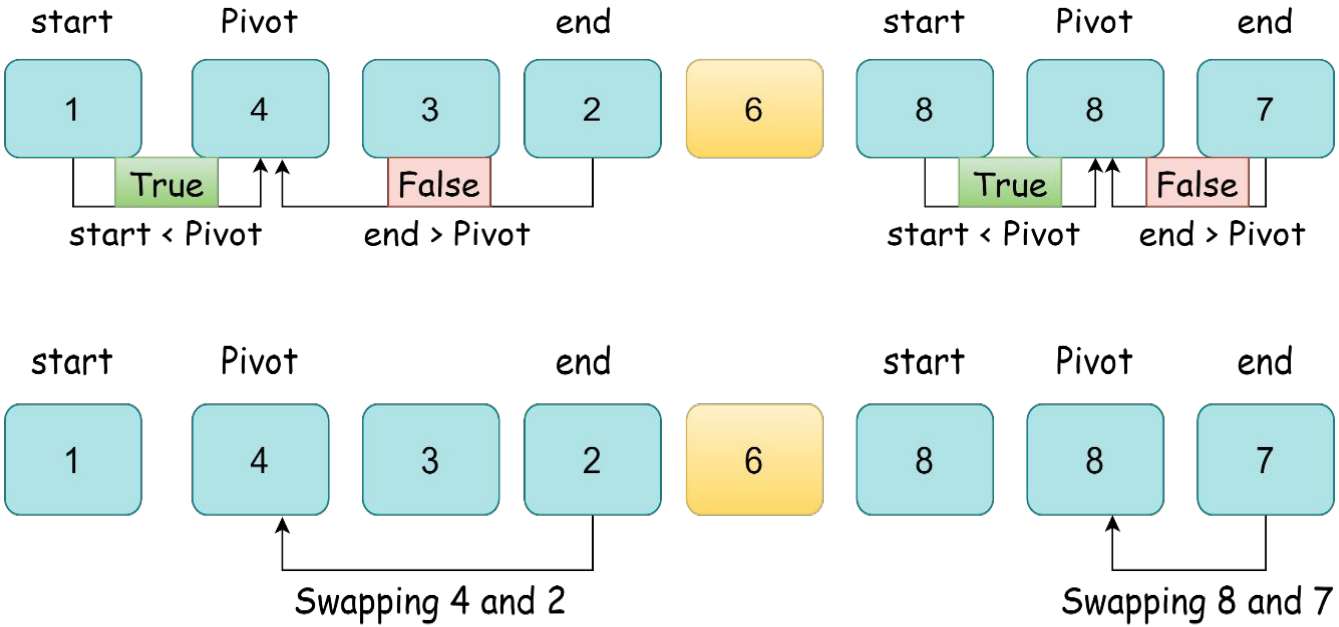
We will get an array after swapping

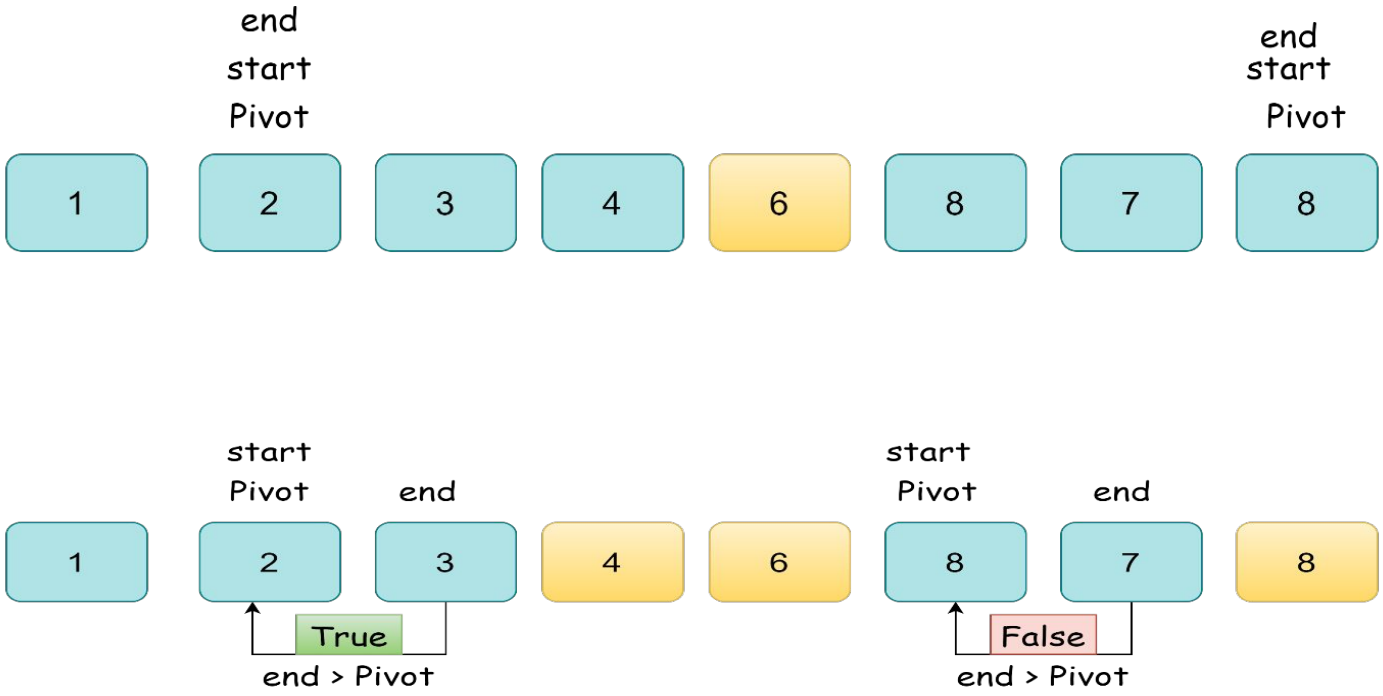


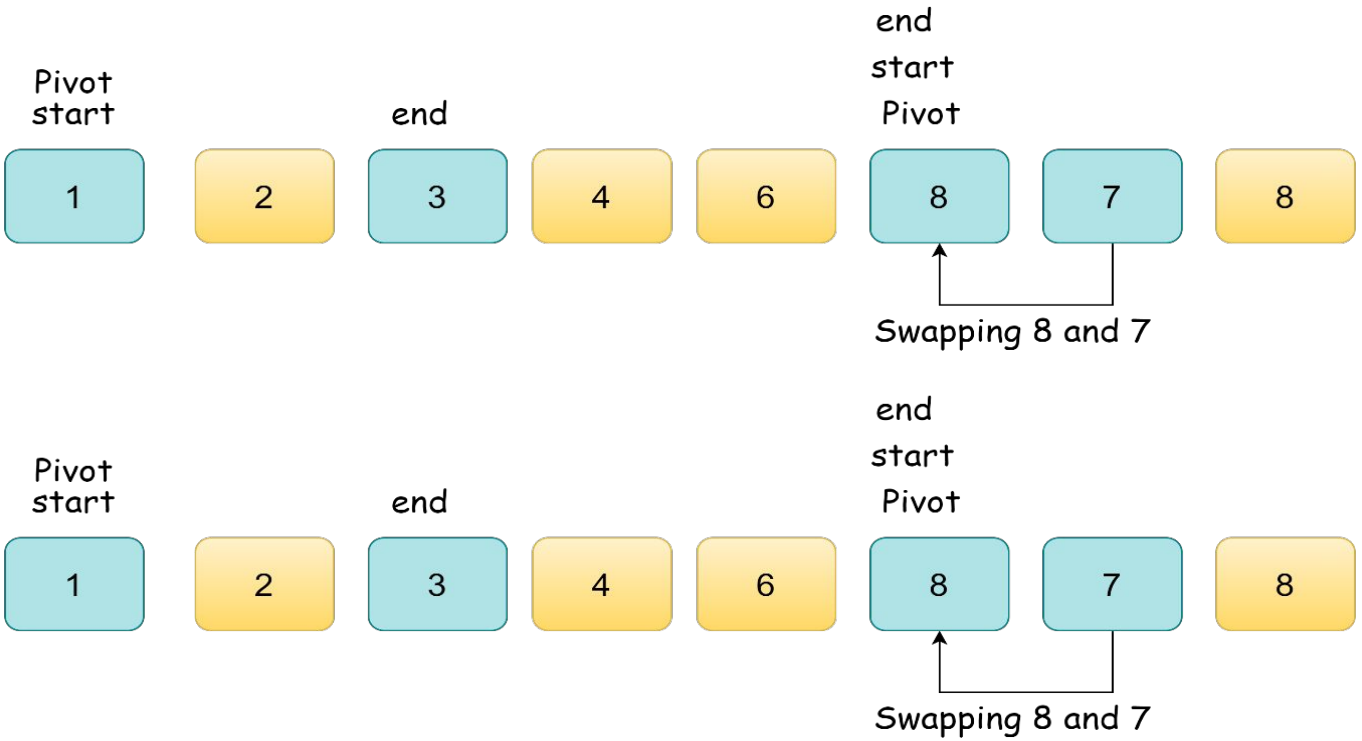
Now value 6 is sorted in the below array and formed into two sub array left having the values less than 6 and right having values greater than 6.

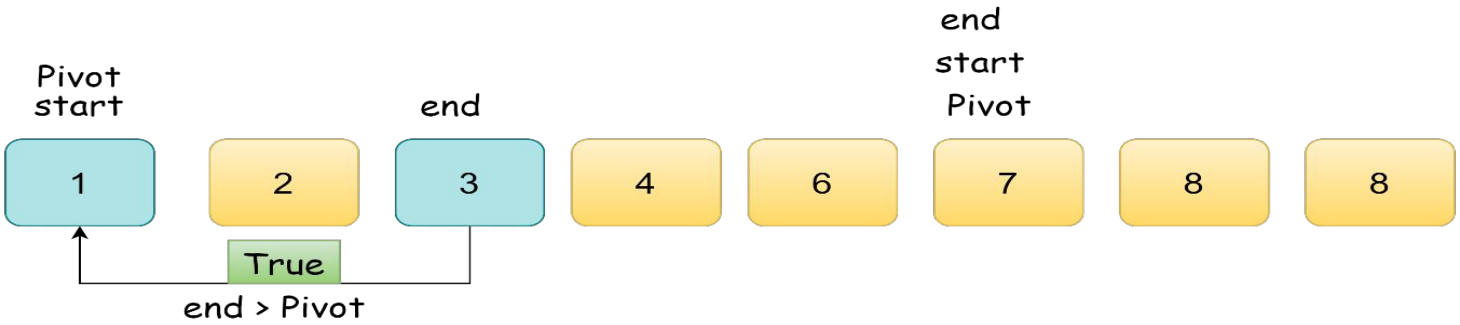


Again, Follow the same steps repeatedly till the condition start <= end




end
start
Pivot
end
start
Pivot
1
2
3
4
6
8
7
8
start
Pivot
end
start
Pivot
end
1
2
3
4
6
8
7
8
True
end > Pivot
False
end > Pivot


Pivot
start
end
end
start
Pivot
1
2
3
4
6
8
7
8
Swapping 8 and 7
Pivot
start
end
end
start
Pivot
1
2
3
4
6
8
7
8
Swapping 8 and 7



Finally we will get the sorted array

Program

jsfiddle - https://jsfiddle.net/saravananslb/xs349kzb/1/

```
// Quick Sort function => find (length - 1)
// and set start = 0
const quickSort = (arr) => {
    const end = arr.length - 1;
    const start = 0;
    sortArray(arr, start, end)
}

// Swapping two index value in an array
const swap = (arr, a, b) =>{
    let temp = arr[a];
    arr[a] = arr[b];
    arr[b] = temp;
}
```

```
// Sort function which execute recursively till sort
const sortArray = (arr, low, high) => {
    // To come out of the infinite loop
    if (low >= high) {
        return;
    }
    let start = low;
    let end = high;
    const mid = Math.floor((start + end) / 2);
    const pivot = arr[mid];

    // Iterate till start less than end
    while (start < end) {
        while (arr[start] < pivot) {
            start++;
        }

        while (arr[end] > pivot) {
            end--;
        }

        if (start <= end) {
            swap(arr, start, end);
            start++;
            end--;
        }
    }
    // Recursively pass the two subarray as a input
    sortArray(arr, low, end);
    sortArray(arr, start, high)

}

const unsortedArray = [8, 4, 3, 6, 2, 8, 1, 7]
quickSort(unsortedArray)
console.log(unsortedArray) // [1, 2, 3, 4, 6, 7, 8, 8]
```

Analysis - Time Complexity Analysis
Worst Case

The worst-case condition of quicksort takes place while the partition method always selects the largest or smallest element as pivot. In this scenario, the partition method could be unbalanced, i.e., one subproblem with n-1 and the other sub problem with 0 elements. This situation occurs when the array will be sorted in ascending or descending order.

Let us anticipate that unbalanced partitioning arises at every recursive call. So for calculating the time complexity withinside the worst case, we placed i=n-1 in the above components of T(n).

```
T(n) = T(n - 1) + T(0) + cn
T(n) = T(n - 1) + cn
```

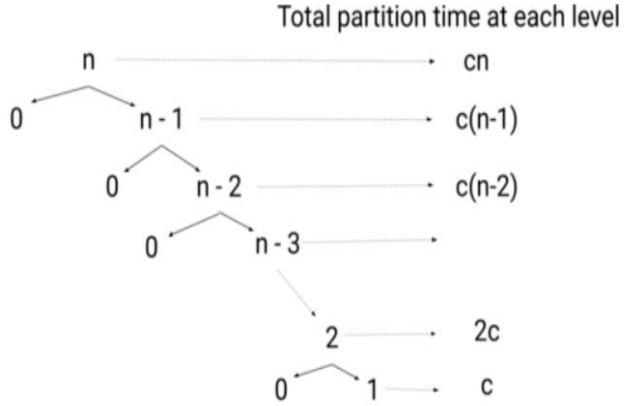
Analysis of worst-case using the substitution method

We certainly extend the recurrence relation through substituting the all intermediate value of T(i) (from I = n-1 to 1). By the end of this process, we ended up in a sum of arithmetic series.

```
T(n)
= T(n-1) + cn
= T(n-2) + c(n-1) + cn
= T(n-3) + c(n-2) + c(n-1) + cn
..... and so on
= T(1) + 2c + 3c + ... + c(n-3) + c(n-2) + c(n-1) + cn
= c + 2c + 3c + ... + c(n-3) + c(n-2) + c(n-1) + cn
= c (1 + 2 + 3... + n-3 + n-2 + n-1 + n)

this is a simple sum of the arithmetic progression.
T(n) = c (n(n+1)/2) = O(n^2)

Worst case Time complexity of the quick sort  = O(n^2)
```



Analysis of worst-case using recursion tree method

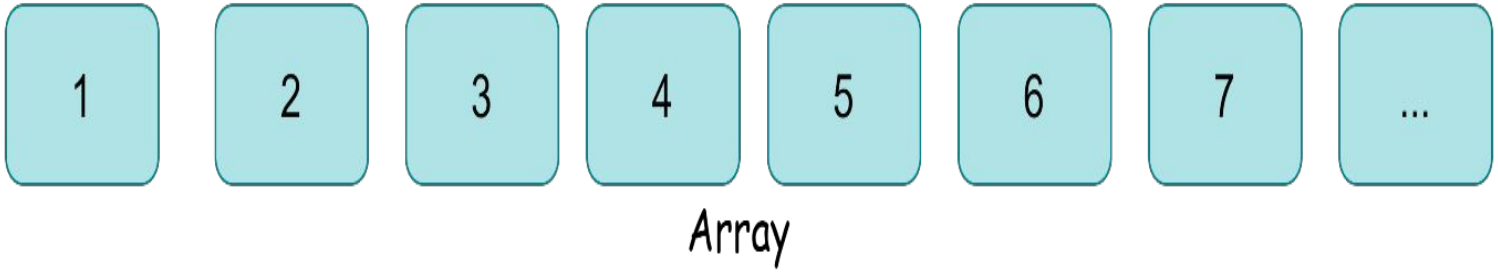
When we sum the entire partitioning instances for every level of recursion tree, we get =>

cn + c(n-1) + c(n-2) + ...+ 2c + c

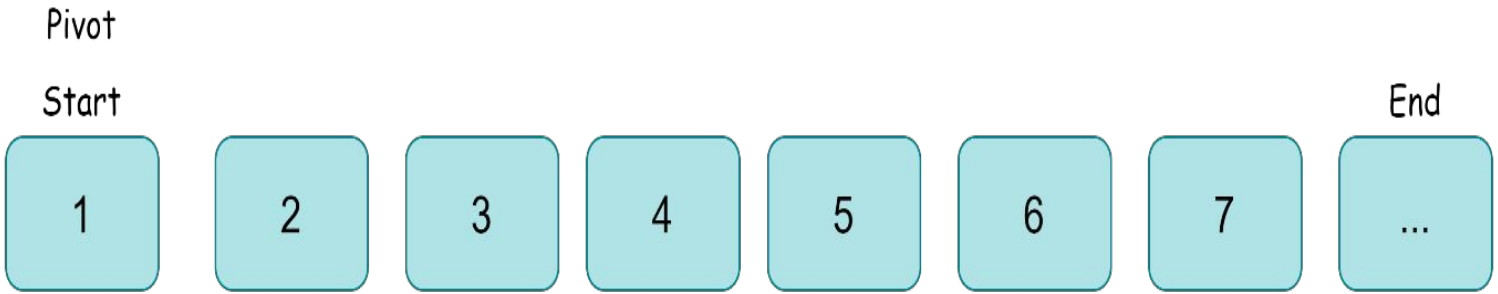
= c(n + n-1 + n-2 + ...+ 2 + 1)

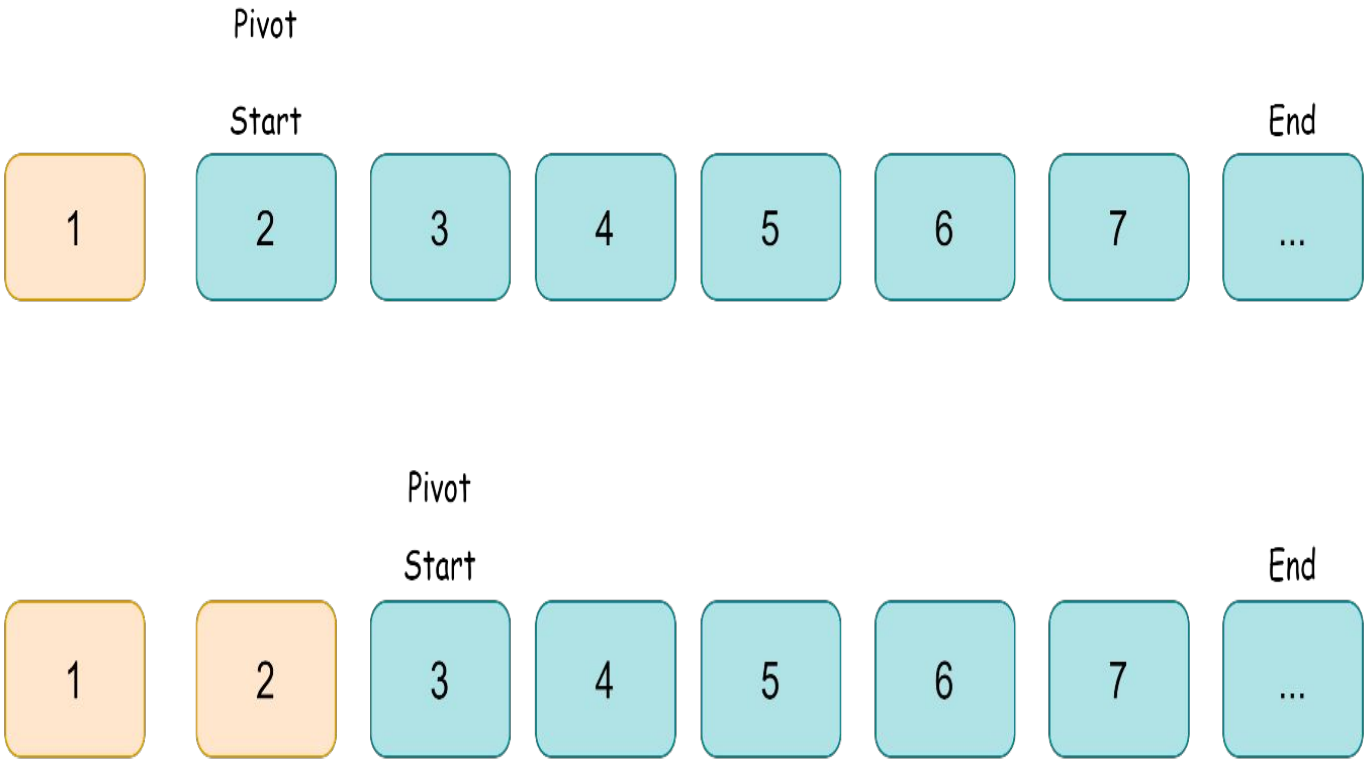
= c(n(n+1)/2)

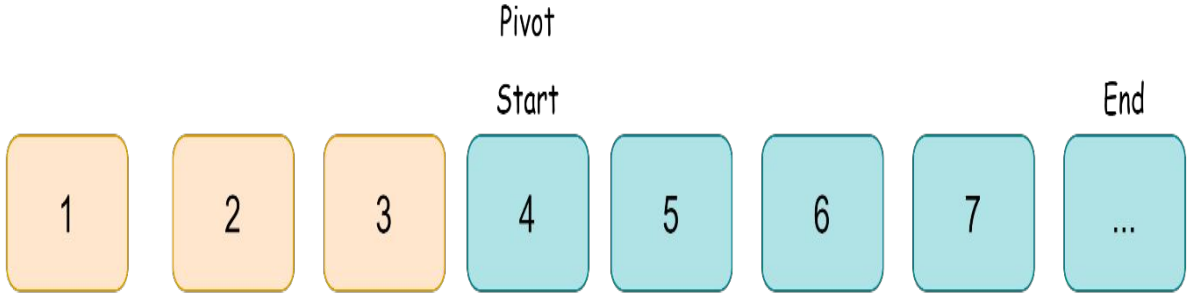
= O(n^2)

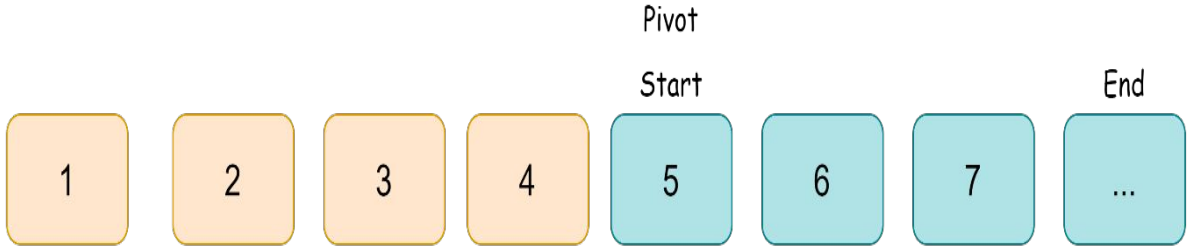


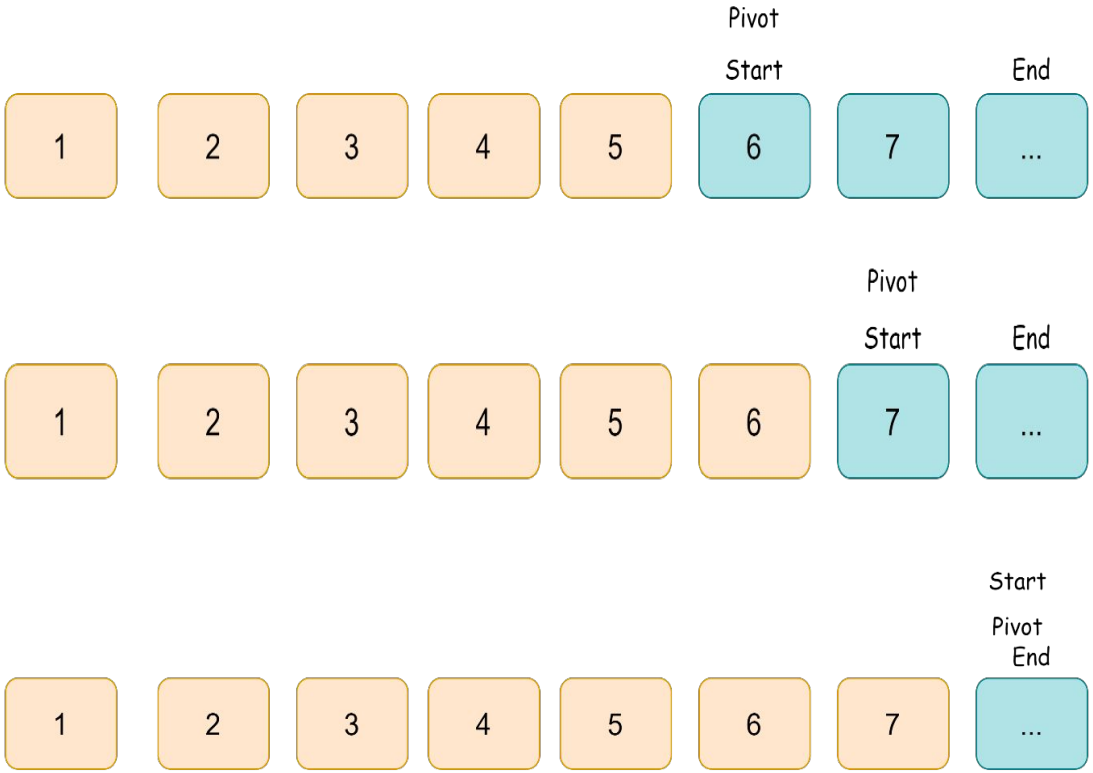
Array




Pivot
Start
End
1
2
3
4
5
6
7
...
Pivot
Start
End
1
2
3
4
5
6
7
...



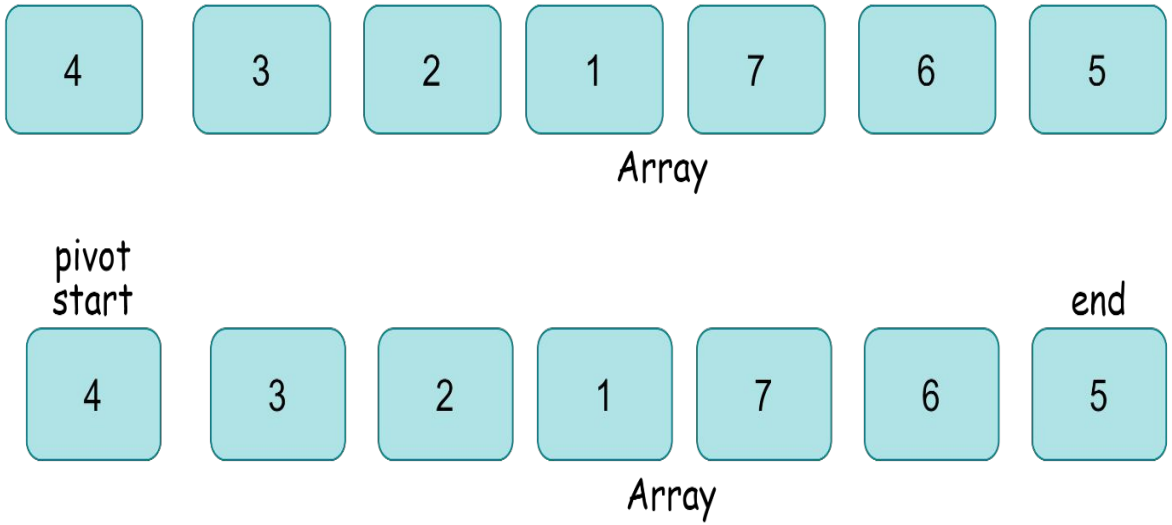


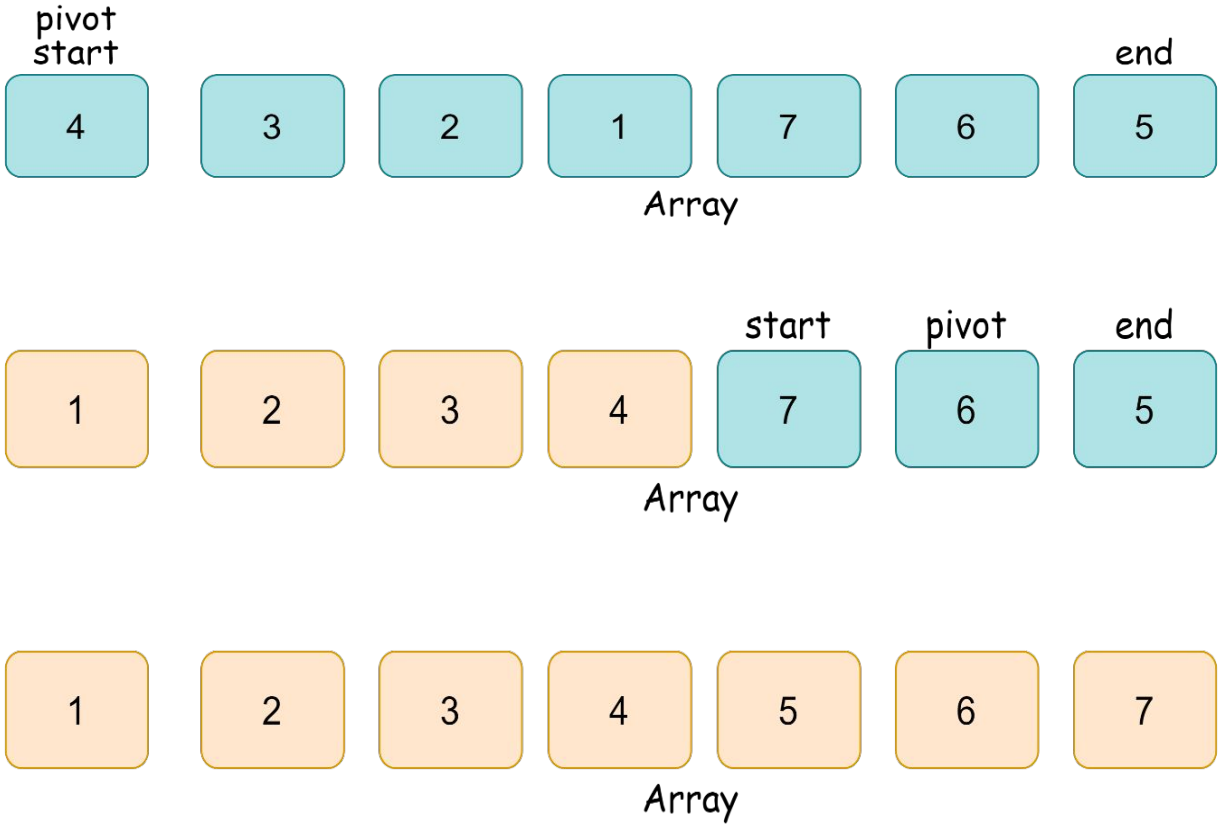

Pivot
Start
End
1
2
3
4
5
6
7
...
Pivot
Start
End
1
2
3
4
5
6
7
...
Start
Pivot
End
1
2
3
4
5
6
7
...

Best Case

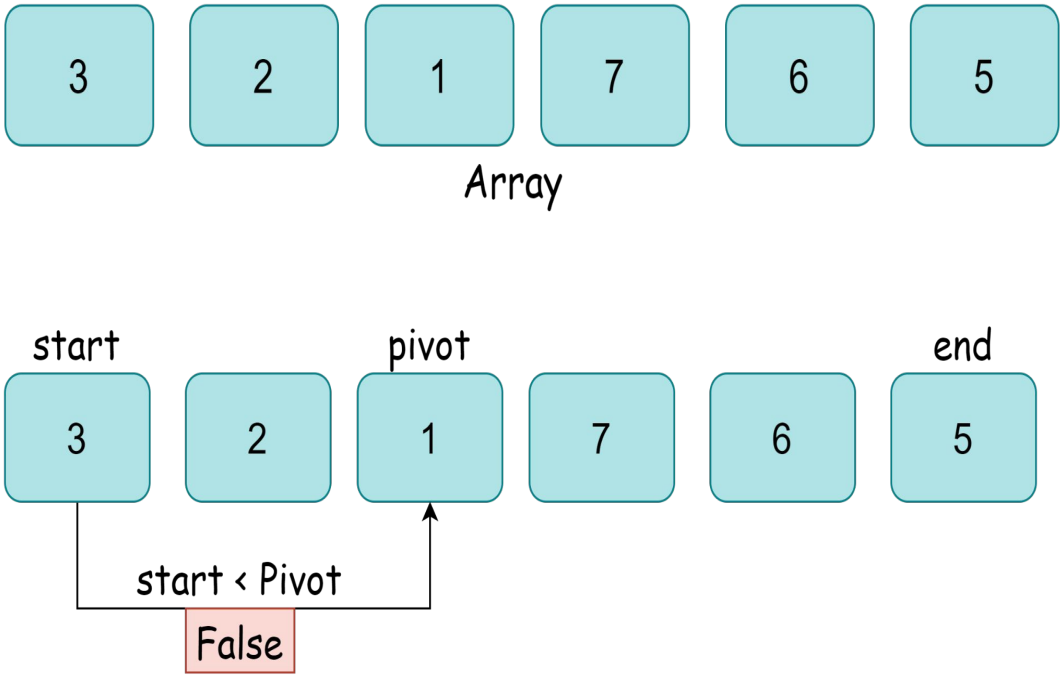
The Best case occur only when the array is partitioned into two sub arrays having the same sub array length.

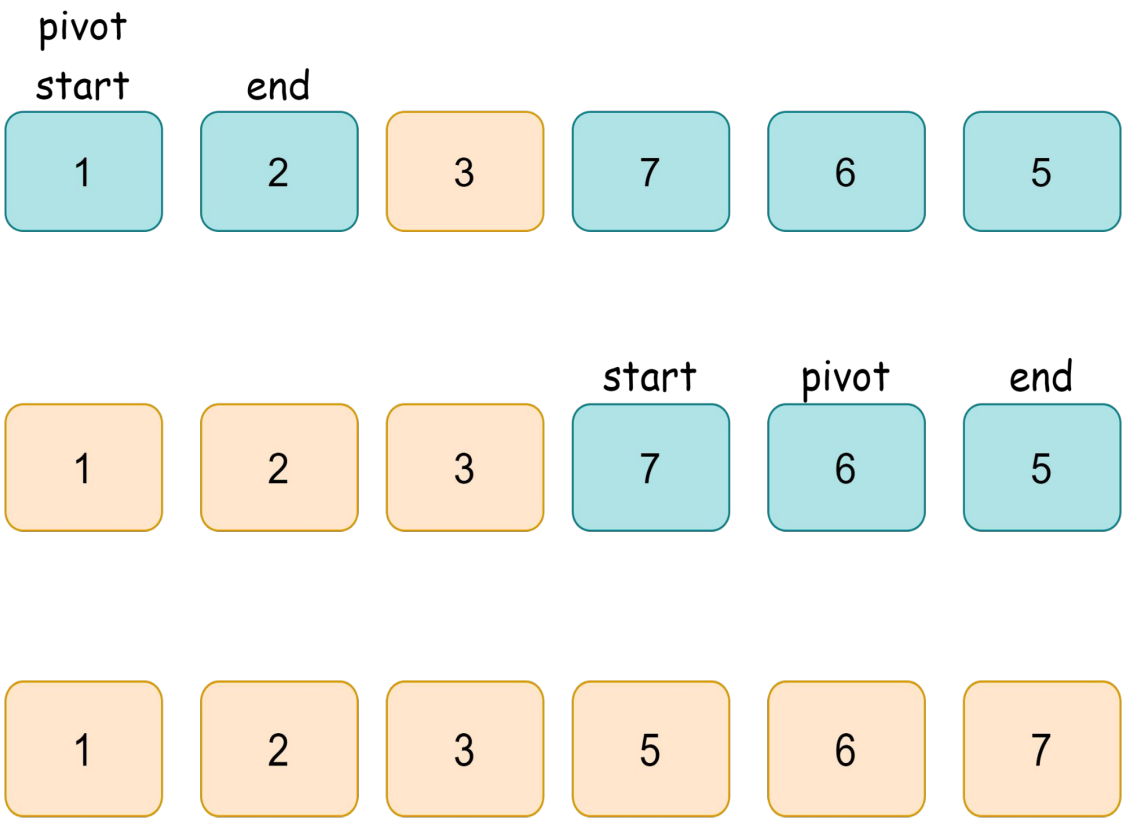
Example 1 (both left and right sub array of same length)




pivot
start
end
4
3
2
1
7
6
5
Array
start
pivot
end
1
2
3
4
7
6
5
Array
1
2
3
4
5
6
7
Array

Example 2 (left and right sub array of different length)




pivot
start
end
1
2
3
7
6
5
start
pivot
end
1
2
3
7
6
5
1
2
3
5
6
7

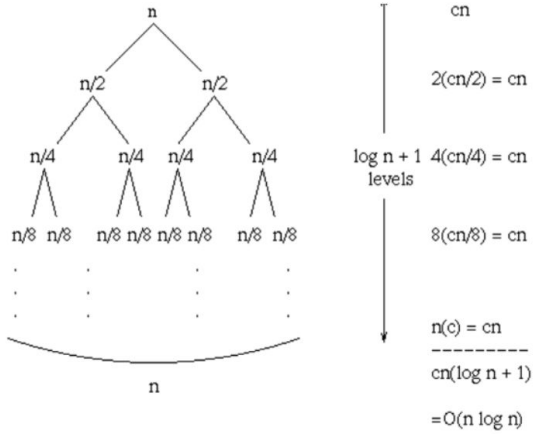
In best-case middle element is considered as a pivot element. In other words, this is a case of the balanced partition wherein each sub-problems are n/2 size each.

Let us expect that balanced partitioning arises in every recursive call. So for calculating the time complexity for best case, we place l = n/2 in the above formula of T(n)

```
T(n) = T(n/2) + T(n - 1 - n/2) + cn
= T(n/2) + T(n/2 - 1) + cn
~ 2 T(n/2)+ cn

T(n) = 2 T(n/2)+ cn
```

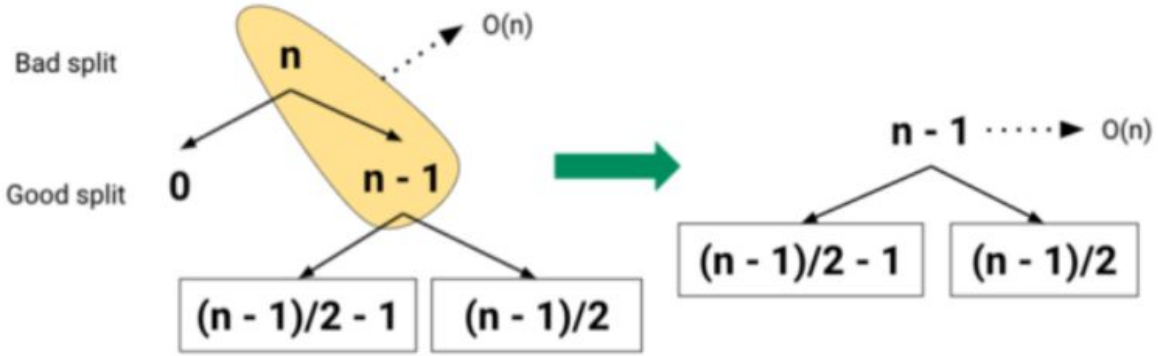
The quick sort's best case time complexity is O. (nlogn)



Average Case

The behaviour of quicksort relies upon the relative order of the values of input.

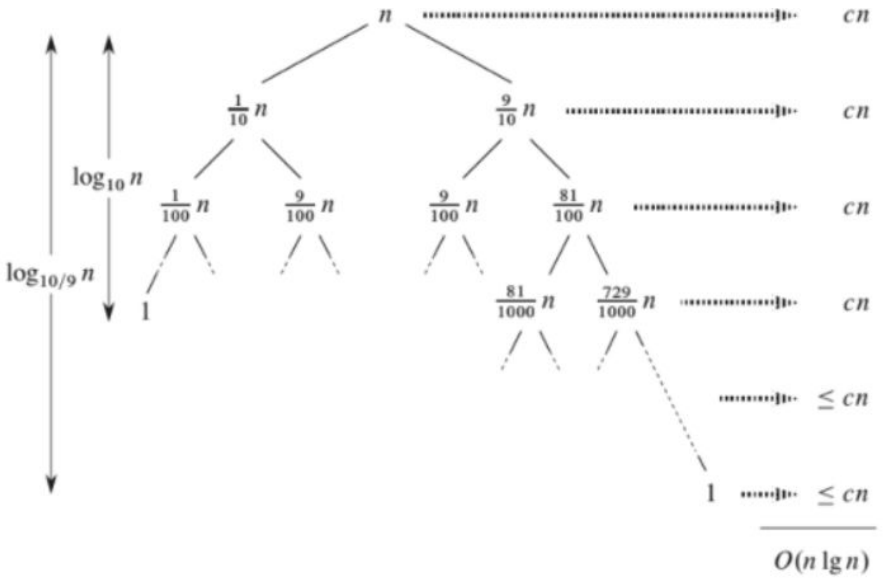
In the average case, the partition procedure generates a mix of good(balance partition) and bad (unbalanced partition) splits.



Average Case

Assume recursion tree, partition technique generates a good split, and at the subsequent level, generates a bad split. The value of the partitioning technique might be O(n) at each level. So the total partitioning cost is O(n).

From the above example, the average case time complexity is O(N log N), For better understanding, let's anticipate the partitioning algorithm produces a partially unbalanced split withinside the ratio of 9 to 1. The recurrence relation for this will be T(n) = T(9n/10) + T(n/10) + cn.



Time Complexity

Time Complexity	
Best	**Ω**(n*log n)
Worst	$O(n^2)$
Average	**Θ**(n*log n)
Space Complexity	O(log n)
Stability	No

Application

- When a stable sort isn't required, the quick sort method is utilised. It may be used in information searching, operational research, and event-driven simulation without requiring any additional store capacity.
- Quick sort is also tail recursive, which is optimized by the compiler.
- It is used in place where memory is the main concerns
- It is used in sorting the element in excel application
- some programming language in built sort functions are built using this algorithm

Advantages of QuickSort

Some of the advantages of quick sort are

1. Takes less space compared to merge sort in average complexity .
2. Used in Internal sorting algorithms.
3. The probability of hitting the worst case is very low in quicksort.
4. Useful in solving order statistics problems.

What is a Randomised Algorithm?

Randomized algorithms that make random choices (coin-tossing) during their executions. As a result, their output does not depend only on their (external) inputs it will also depend on the random choice.

- The technique uses a random element in the array as a pivot in addition to the input.
- During execution, it takes random choices from the array because of those random numbers, the output can vary if the algorithm will runs multiple times with the same input.

Advantages:

- Selecting a random element in a array as a pivot is fast.
- Randomized algorithms are mostly simpler and faster because of their deterministic algorithms since selecting random pivot will help us to sort quickly.

Random Number Generation

There are a lot of methods for generating a random number. Some of the classic examples are flipping coin, shuffling of playing cards and dice rolling.

Randomised QuickSort Algorithm(20 min)

The two ways of adding randomization in Quick Sort is

1. Randomly placing the input data in the array
2. Randomly choosing the element in the input data for pivot

Second is the most preferred one

In Quick Sort we always choose the leftmost element in the array list to be sorted. Instead of always choosing A[low] as pivot element, we will use a randomly selected element from A[low...high-1].It is done by exchanging A[low] with an element randomly chosen from A[low...high-1]. so that the random elements are chosen equally

Example

Let's consider an array num[] = [1,2,3,4,5] which is already sorted

if we choose 3 as a pivot element randomly here ,it will check on the left side of the array . all the elements are less than pivot elements so there is no swap. similarly for the right side of the array.

Here the time complexity is T(n)<= O(n^2)

Random QuickSort Analysis(20 min)

code link - https://jsfiddle.net/saravananslb/6jq2Lu1x/

Worst Case: O(N^2)

Worst case will happen when the pivot is the smallest or the largest element. When one of the partitions is empty, we repeat the procedure for N-1 elements.

Worst-case Running Time

The worst case for quick-sort occurs when the pivot is the unique left array or right array element. The running time is proportional to the sum (n + (n - 1) + ... + 2 + 1) Thus, that worst-case running time of quick-sort is O (N^2)

Depth time

0 N

1 n - 1

... ...

n – 1 1

$T(N) = T(N-1) + cN, N > 1$

Telescoping:

$T(N-1) = T(N-2) + c(N-1)$

$T(N-2) = T(N-3) + c(N-2)$

$T(N-3) = T(N-4) + c(N-3)$

.....

$T(2) = T(1) + c.2$

$T(N) + T(N-1) + T(N-2) + \ldots + T(2) =$

$= T(N-1) + T(N-2) + \ldots + T(2) + T(1) +$

$C(N) + c(N-1) + c(N-2) + \ldots + c.2$

$T(N) = T(1) +$ c times (the sum of 2 thru N)

$= T(1) + c(N(N+1)/2 - 1) = O(N2)$

Average-case: O (N logN)

Best-case: O (N logN)

The Best case will occur when the pivot is the median of the array, the left and the right parts of the sub array have the same number of elements. That are logN partitions, and to obtain each partition we do N comparisons . Hence the complexity is O (NlogN).

Best case Analysis:

The time taken to sort the file is equal to the time taken to sort the left partition of the array with i elements plus the time taken to sort the right partition of the array with N-i-1 elements, plus the time taken to build the partitions.

Then the pivot is in the middle of the array

$T(N) = 2\,T(N/2) + cN$

Divide by N: $T(N) / N = T(N/2) / (N/2) + c$

Telescoping:

$T(N) / N = T(N/2) / (N/2) + c$

$T(N/2) / (N/2) = T(N/4) / (N/4) + c$

$T(N/4) / (N/4) = T(N/8) / (N/8) + c$

.....

$T(2) / 2 = T(1) / (1) + c$

Add all equations:

$T(N) / N + T(N/2) / (N/2) + T(N/4) / (N/4) + \ldots + T(2) / 2 =$

$= (N/2) / (N/2) + T(N/4) / (N/4) + \ldots + T(1) / (1) + c.\log N$

After crossing the equal terms:

$T(N)/N = T(1) + c * \log N$

$T(N) = N + N * c * \log N = O(N \log N)$

Practice Problem

1) Given an array. You need to find a triplet that will be having sum as a target. A Triplet is a set of 3 numbers.
   Input - [3,6,5,2,10]
   Target - 11
   Output - [3,6,2]
1) Write a program to sort the elements in the array by descending order using a random pivot.

Thank you