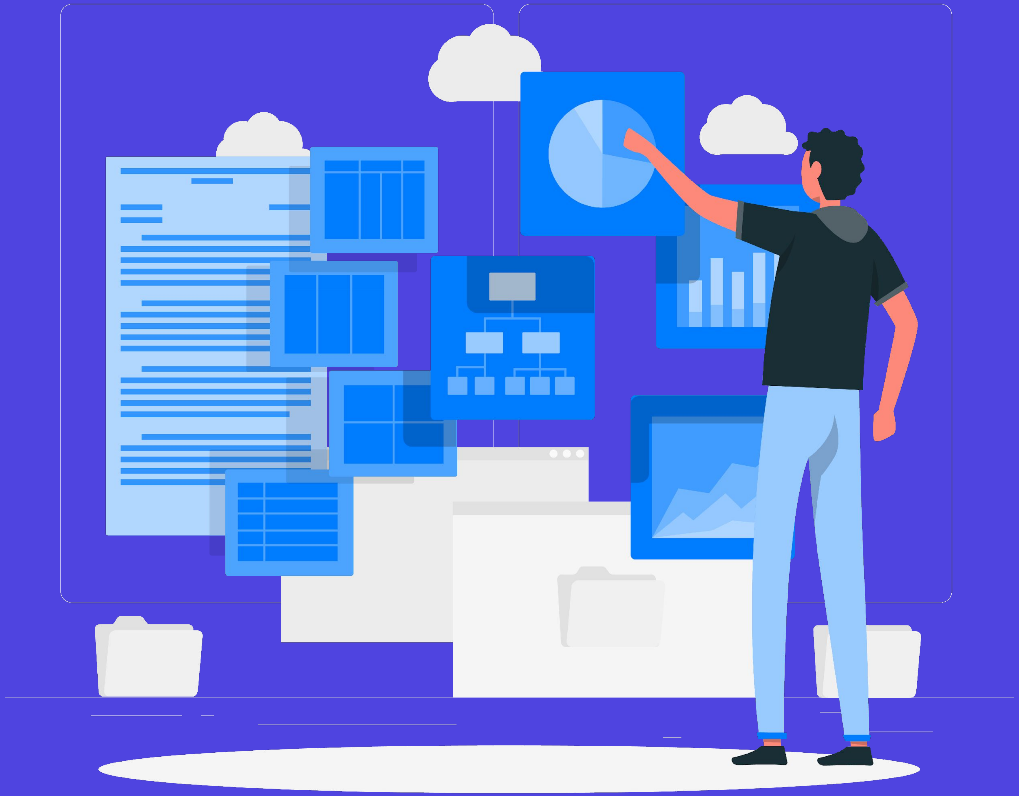
Algorithms: Quick Sort

Relevel
by Unacademy



Topics Covered

- Quick Sort
- Randomised Quick Sort
- Analysis
- Applications
- Quick select algorithm
- Solving order statistics problem
- median of medians vs quick select
- Merge sort vs quick sort

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(n2), respectively.

Choosing pivot in QuickSort

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

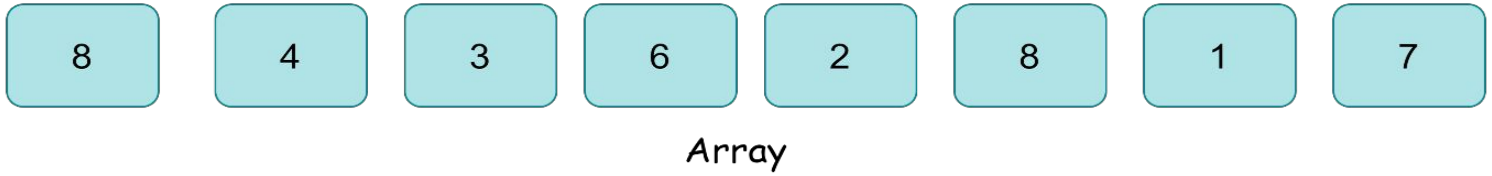
Selecting starting element as pivot

Selecting Middle element as pivot

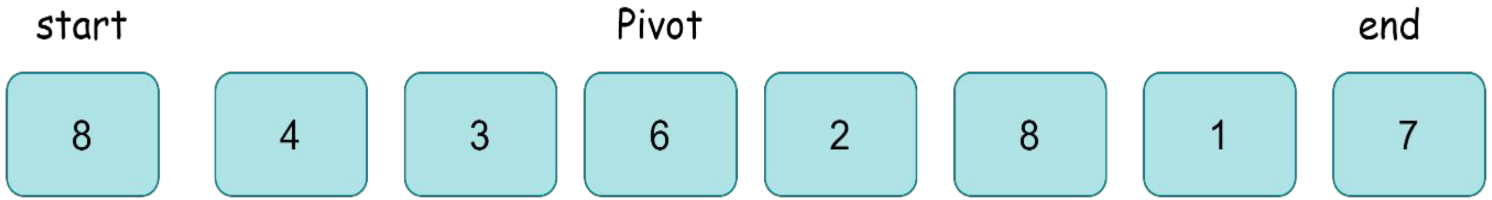
Selecting Last element as pivot

Picking a random number from the given list of elements as the pivot (Also called as randomised quicksort)

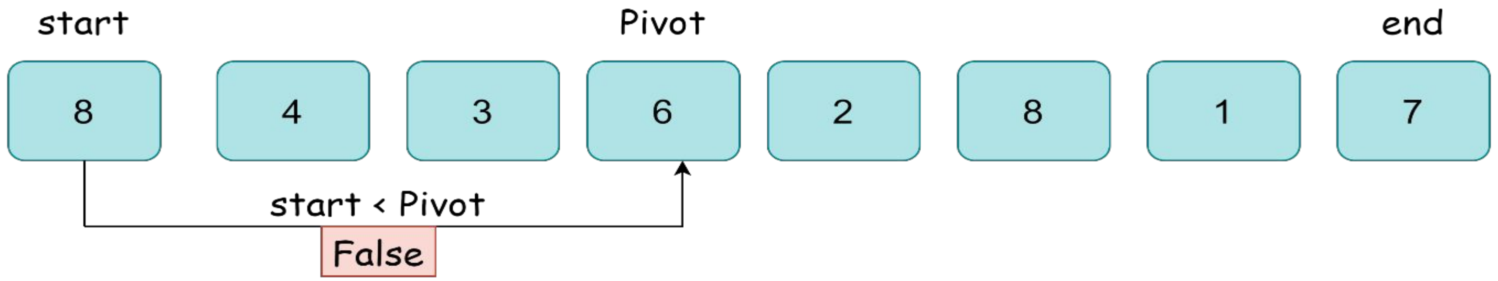
Let us consider the array of element 8, 4, 3, 6, 2, 8, 1, 7 and Middle element as Pivot



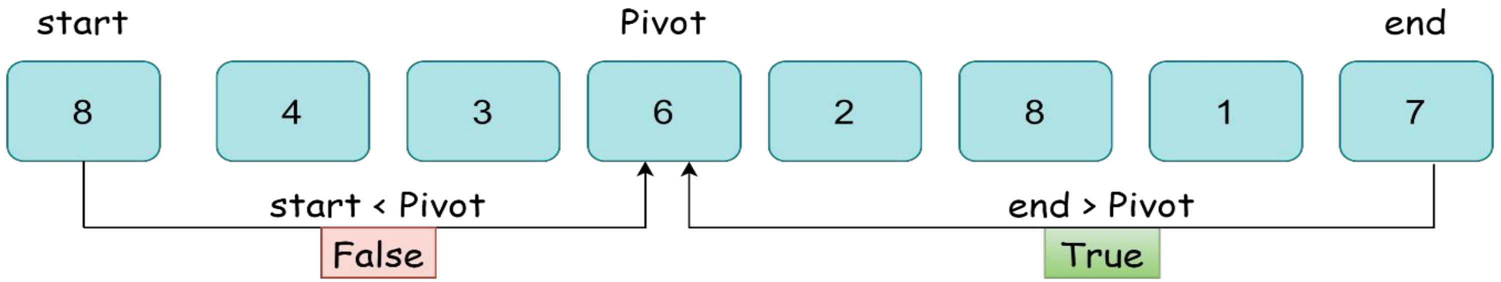
First element in the array as start and last element in the array as end



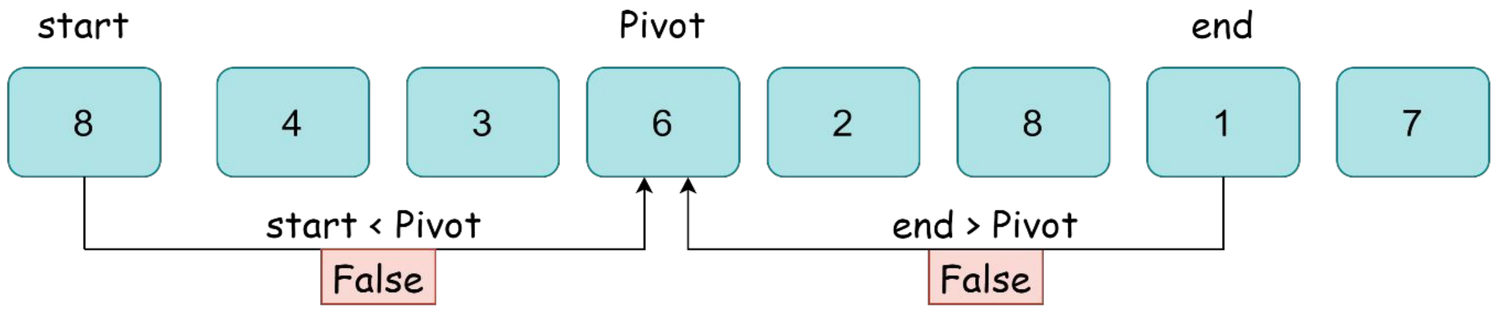
First it will check if the start is less than pivot and if the condition is false then move to right side of the pivot

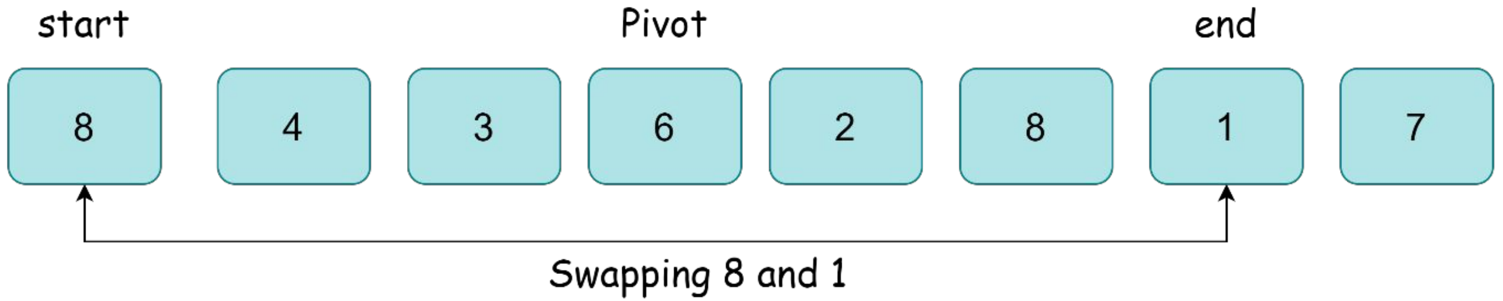


Check if the end is greater than pivot and if the condition is true decrease the end pointer

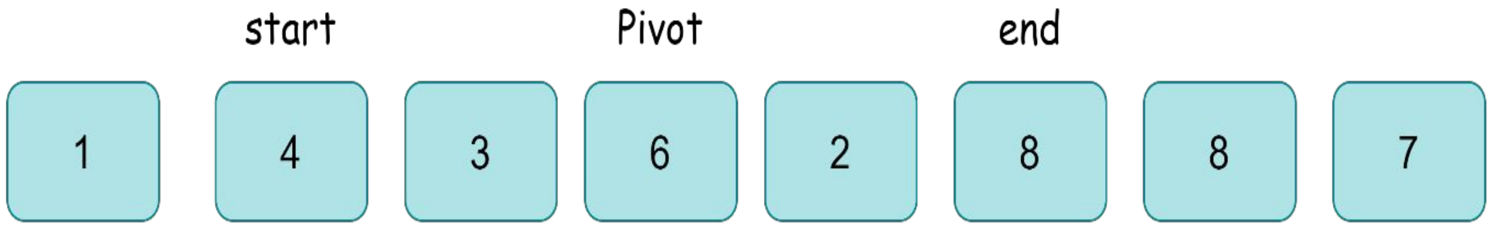


Again, check if the end is greater than pivot and if the condition is false then swap start and end index elements

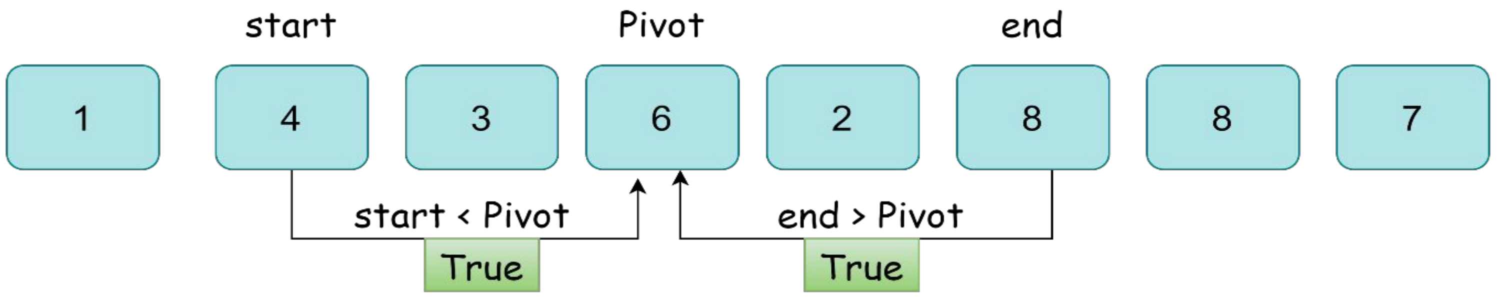




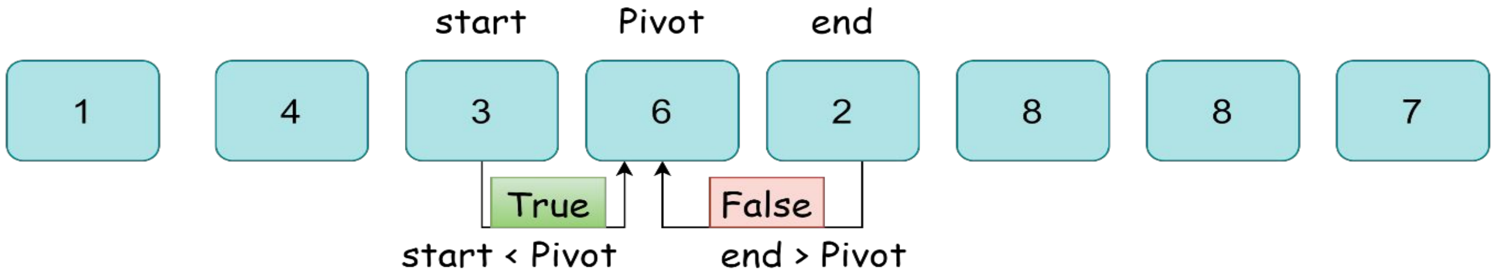
We will get a new array in the below and increase the start and decrease the end



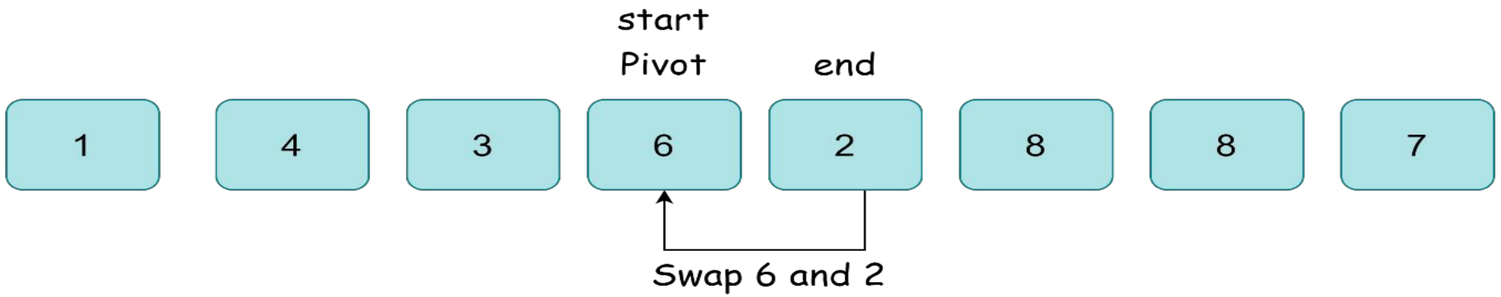
Again, check the condition if the start is less than pivot and end is greater than pivot both the conditions are met then increase the start and decrease the end



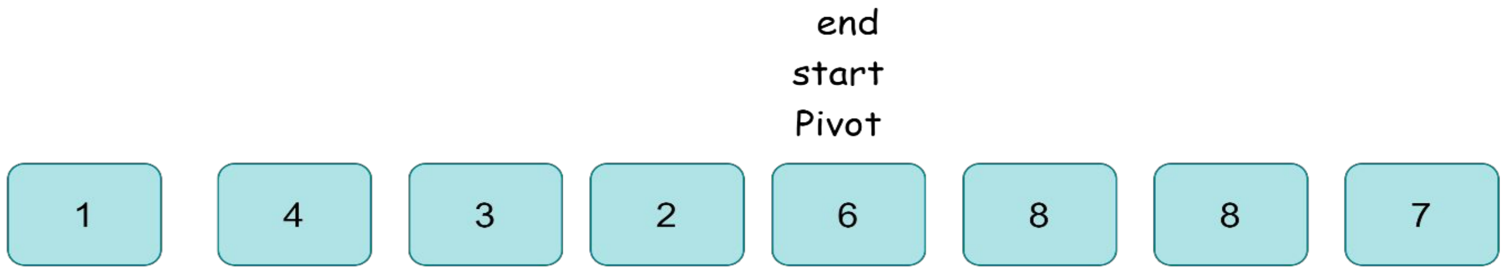
Again, check the condition if the start is less than pivot and end is greater than pivot one of the condition is met then increase the start



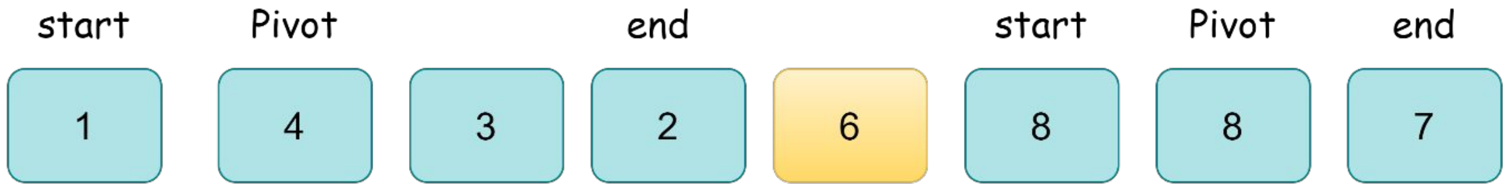
Swap the start and end values and increase the start and decrease the end



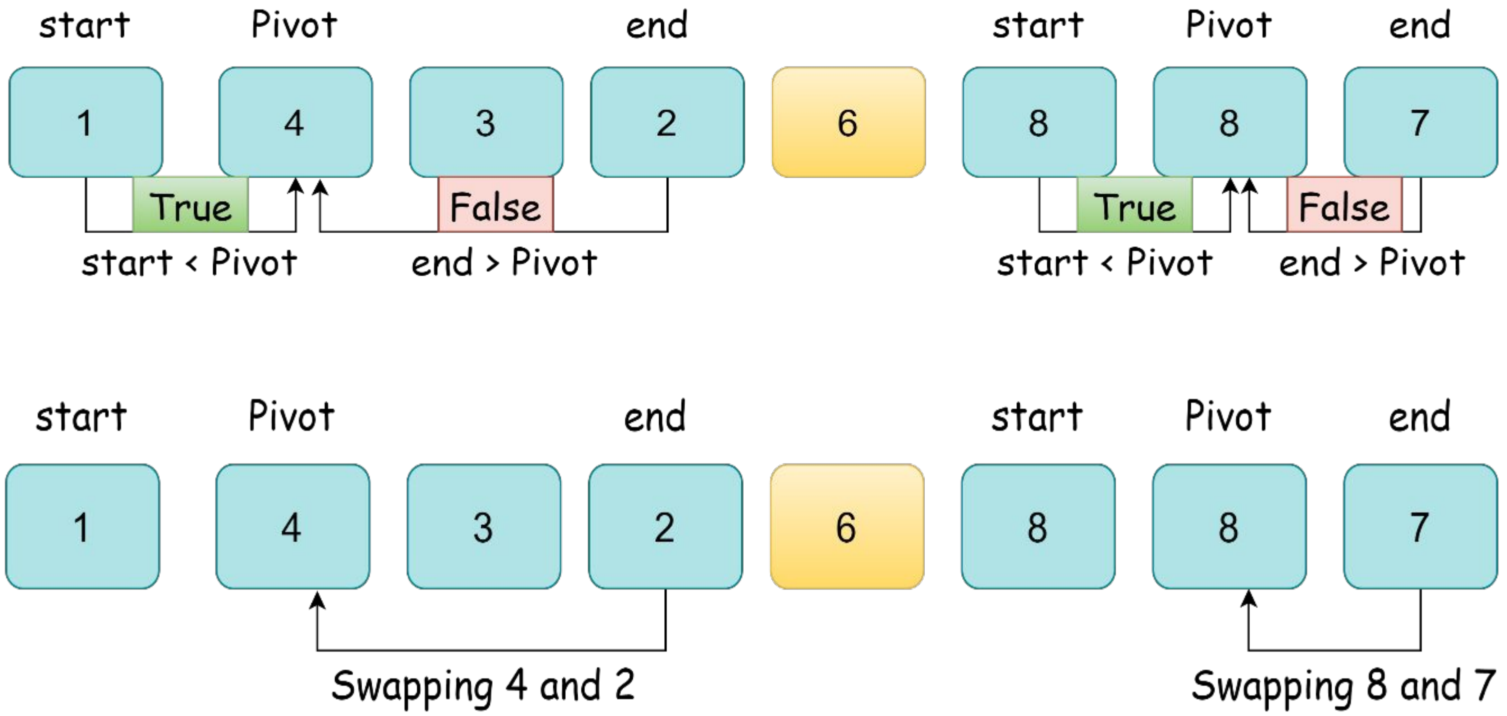
We will get an array after swapping

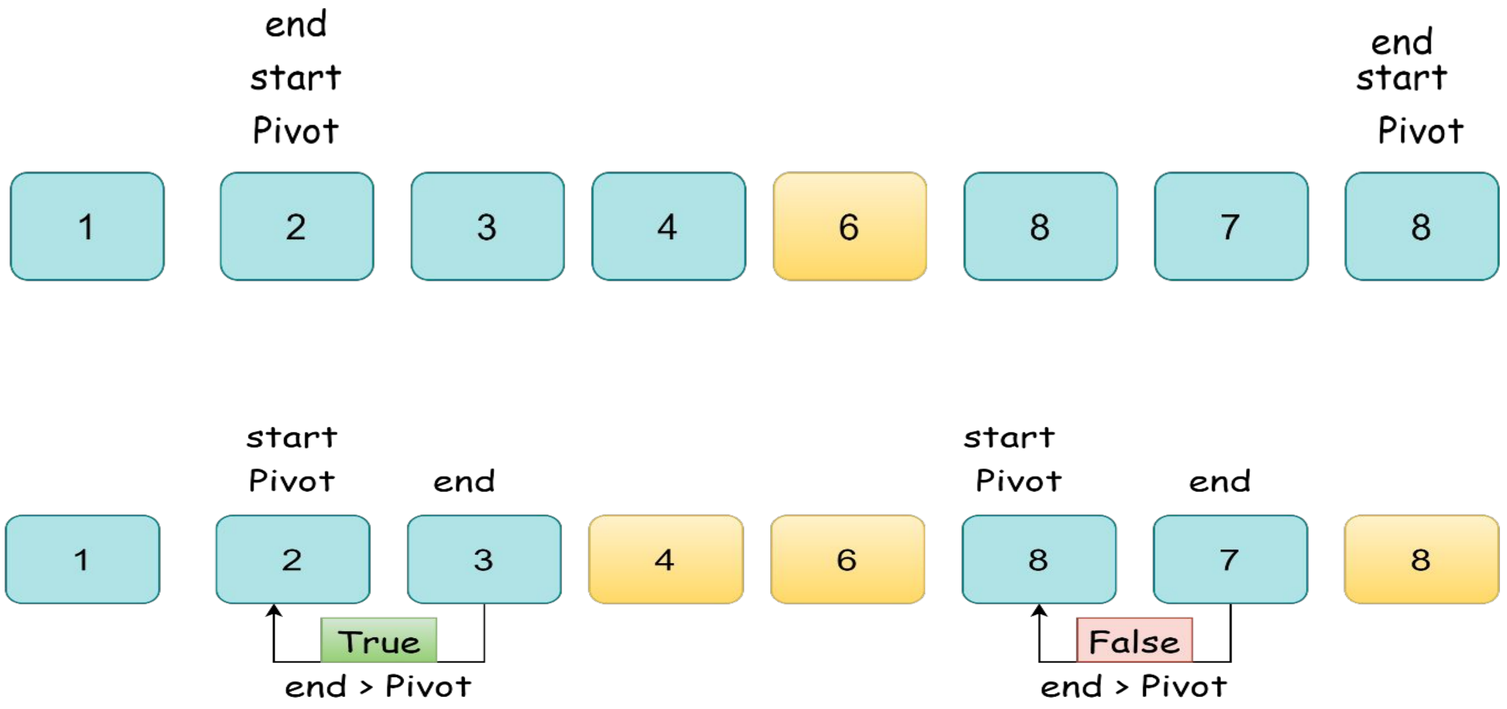


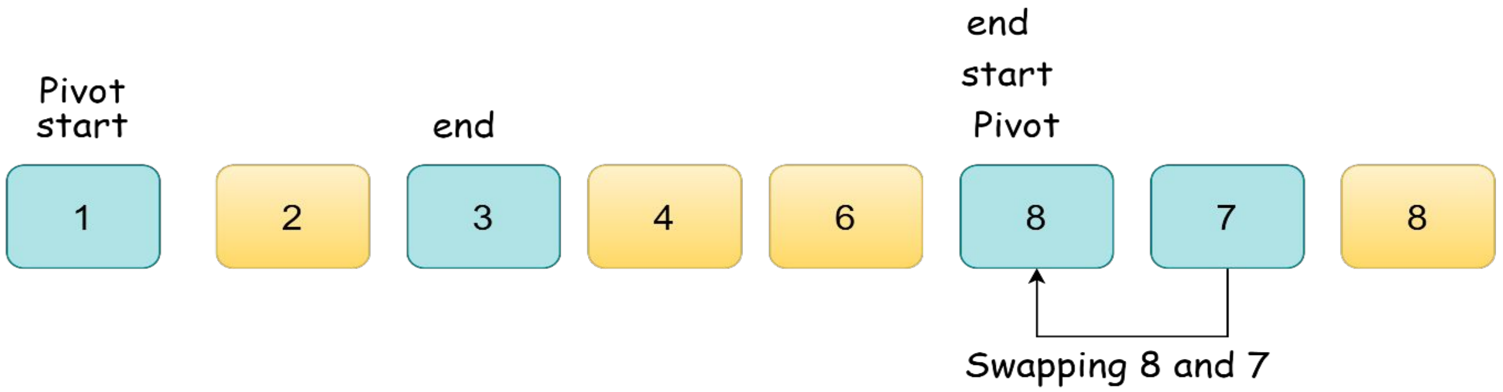
Now value 6 is sorted in the below array and formed into two sub array left having the values less than 6 and right having values greater than 6.

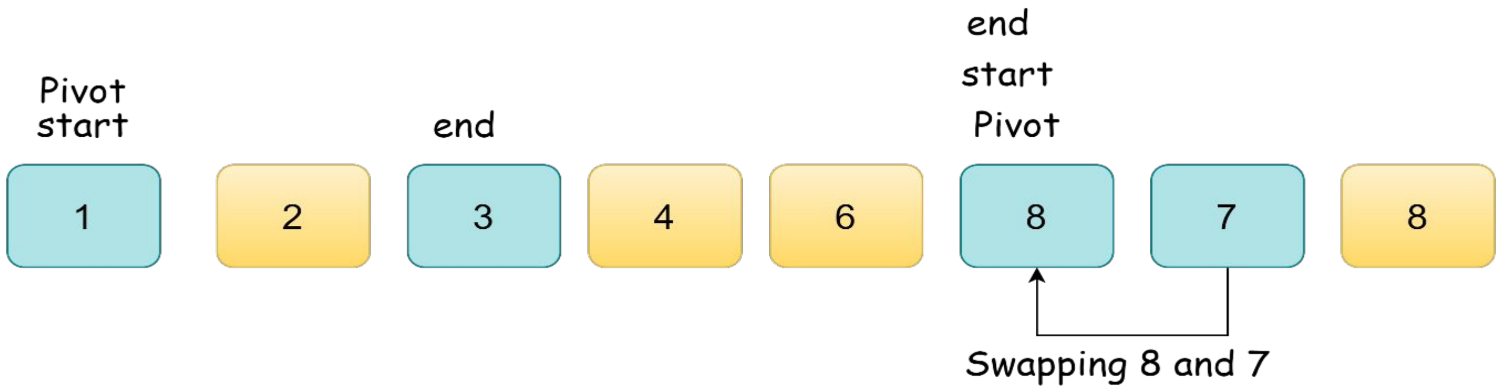


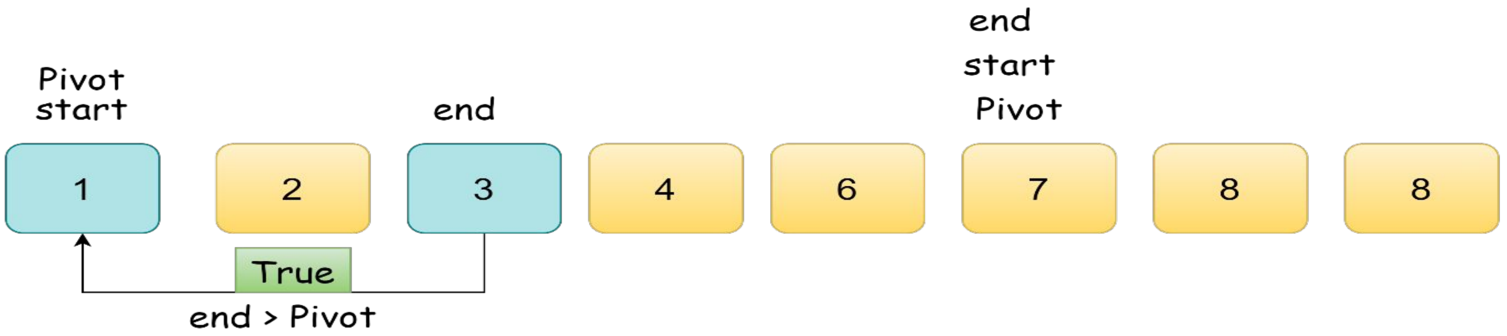
Again, Follow the same steps repeatedly till the condition start <= end



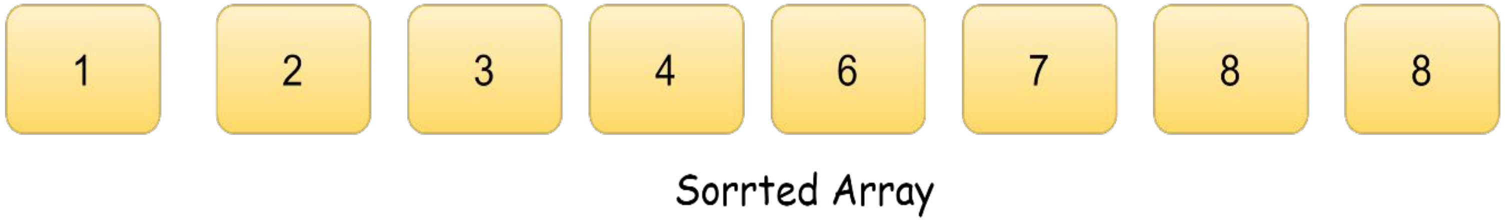

end
start
Pivot
end
start
Pivot
1
2
3
4
6
8
7
8
start
Pivot
end
start
Pivot
end
1
2
3
4
6
8
7
8
True
end > Pivot
False
end > Pivot


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


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

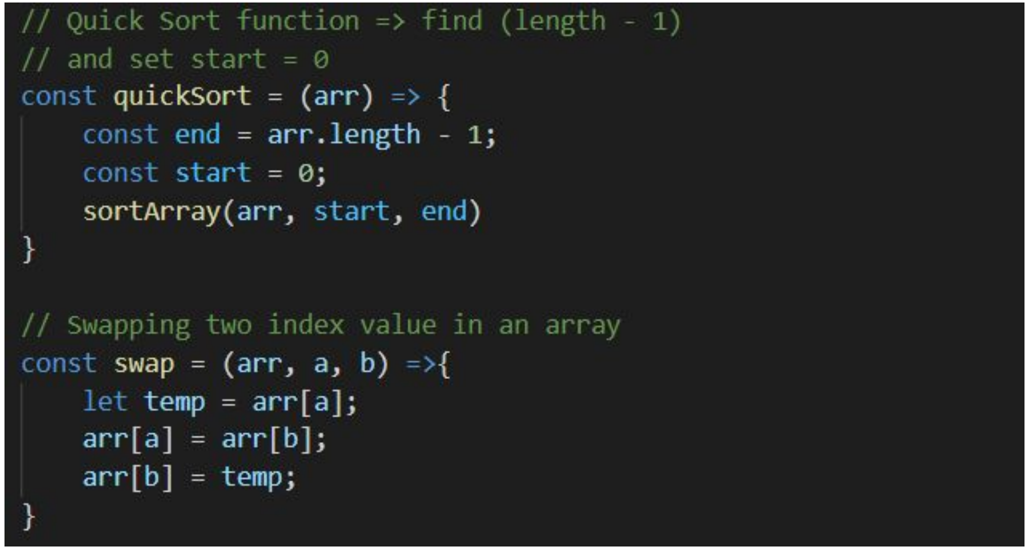


Finally we will get the sorted array

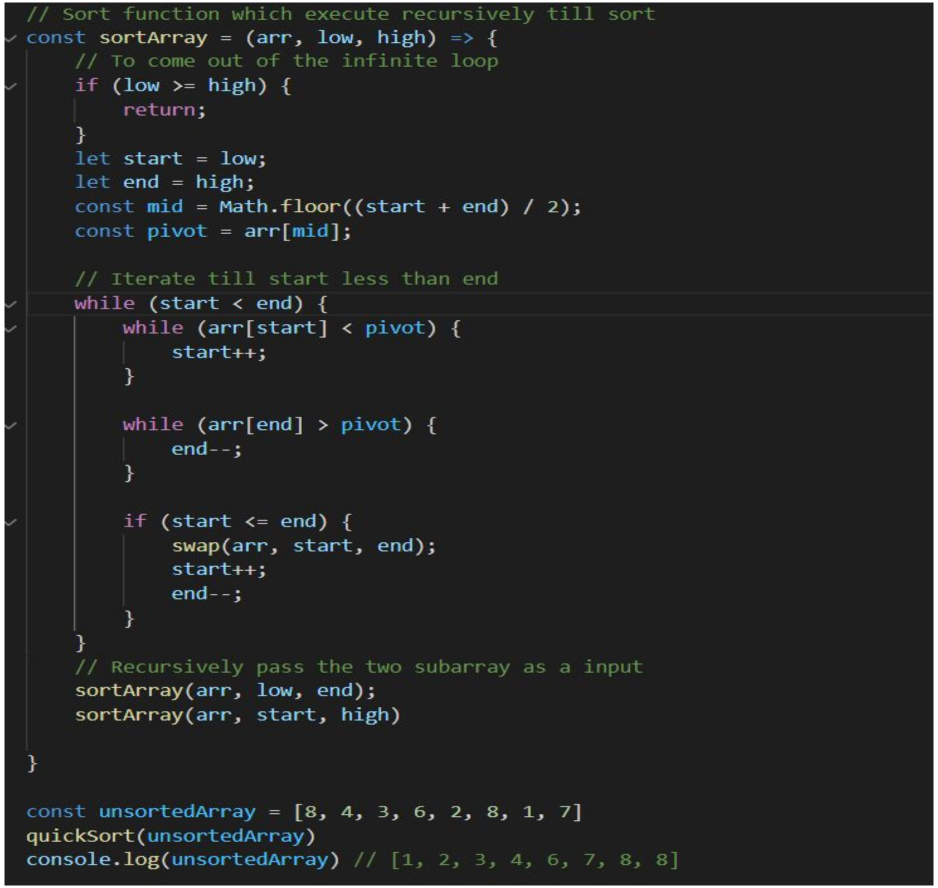


Program

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;
}
```



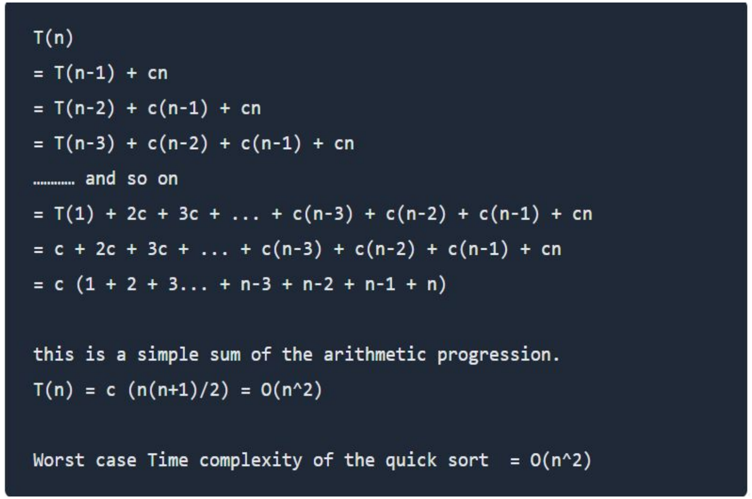
```
// Sort function which execute recursively till sort
const sortArray = (arr, low, high) => {
    // To come out of the infinite loop
    if (low >= high) {
        return;
    }
    let start = low;
    let end = high;
    const mid = Math.floor((start + end) / 2);
    const pivot = arr[mid];

    // 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

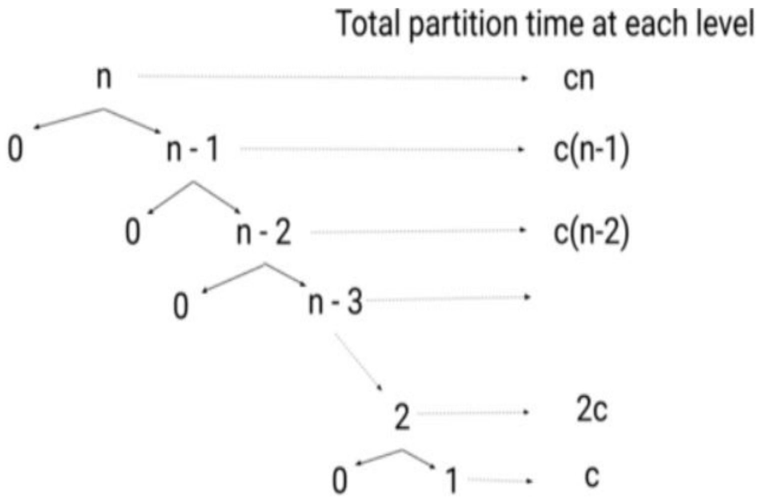
The worst-case condition of quicksort takes place while the partition method always selects the largest or smallest element as pivot. In this scenario, the partition method could be unbalanced, i.e., one subproblem with n-1 and the other sub problem with 0 elements. This situation occurs when the array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

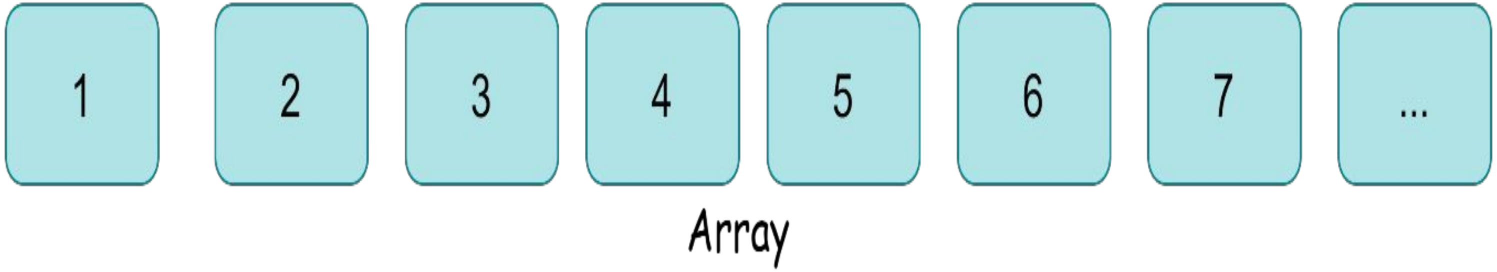
When we sum the entire partitioning instances for every level of recursion tree, we get =>

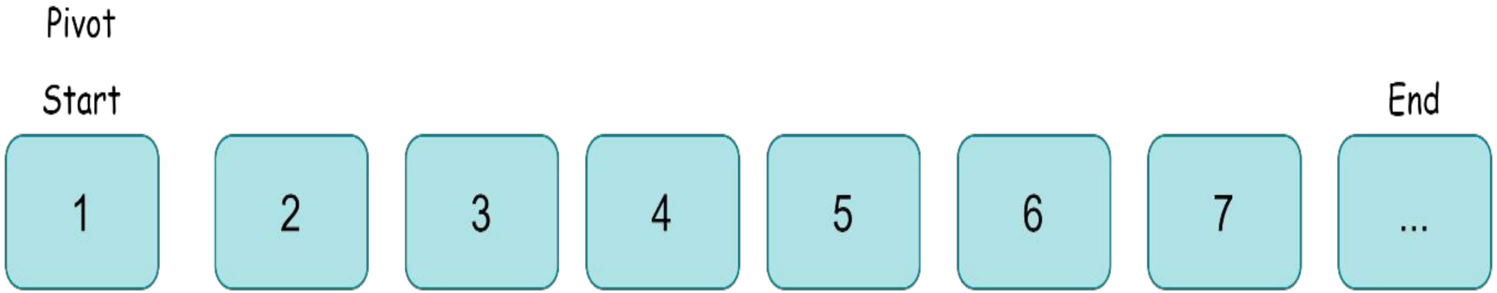
cn + c(n-1) + c(n-2) + …..+ 2c + c

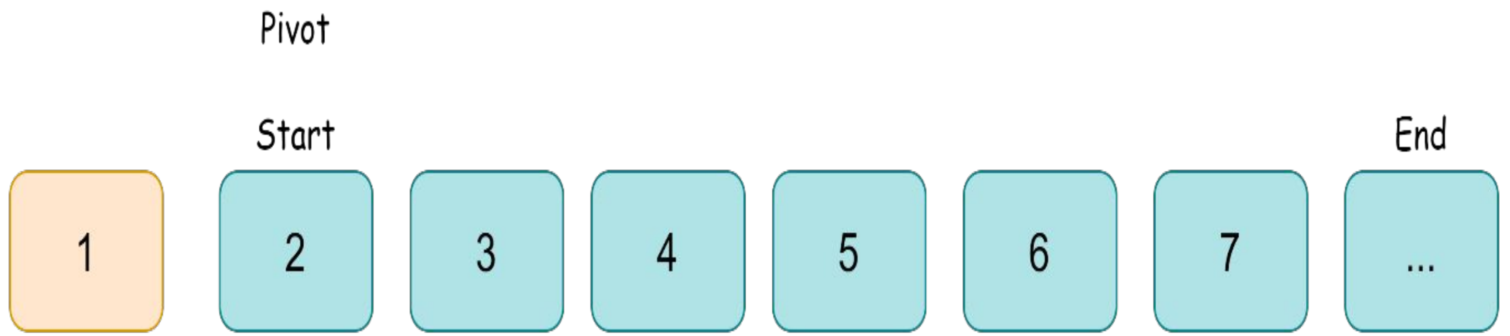
= c(n + n-1 + n-2 + …..+ 2 + 1)

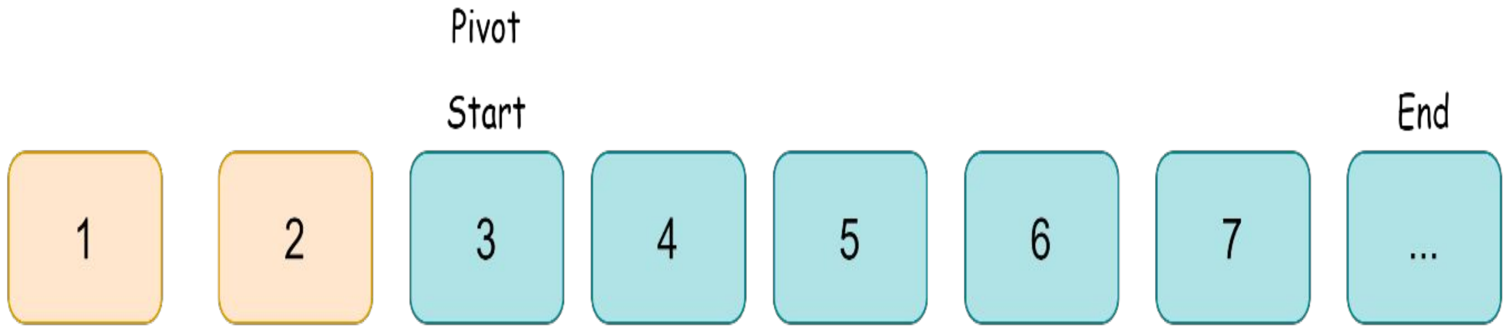
= c(n(n+1)/2)

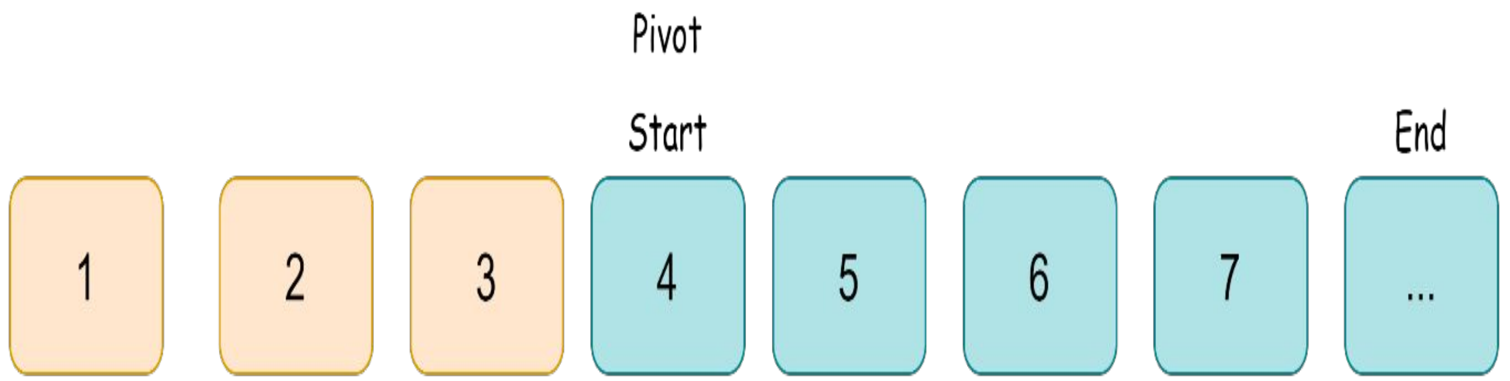
= O(n^2)

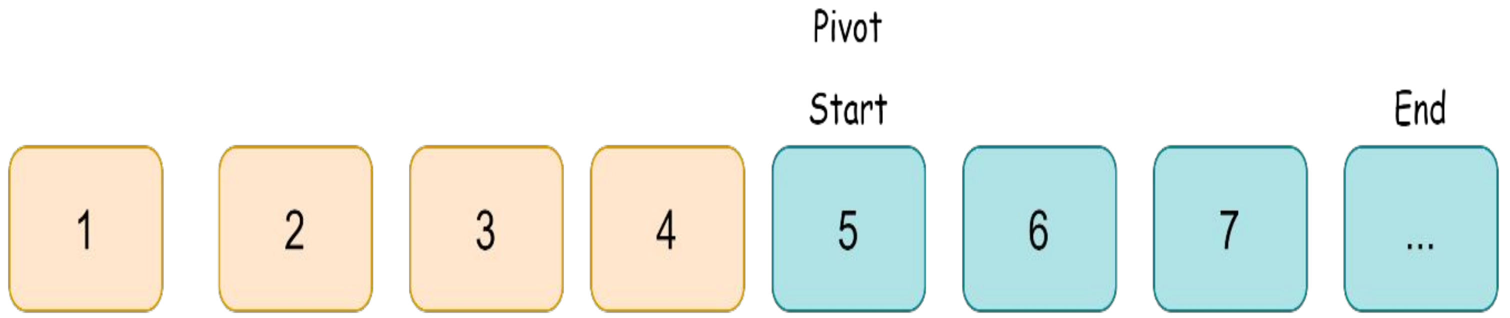

1
2
3
4
5
6
7
...
Array

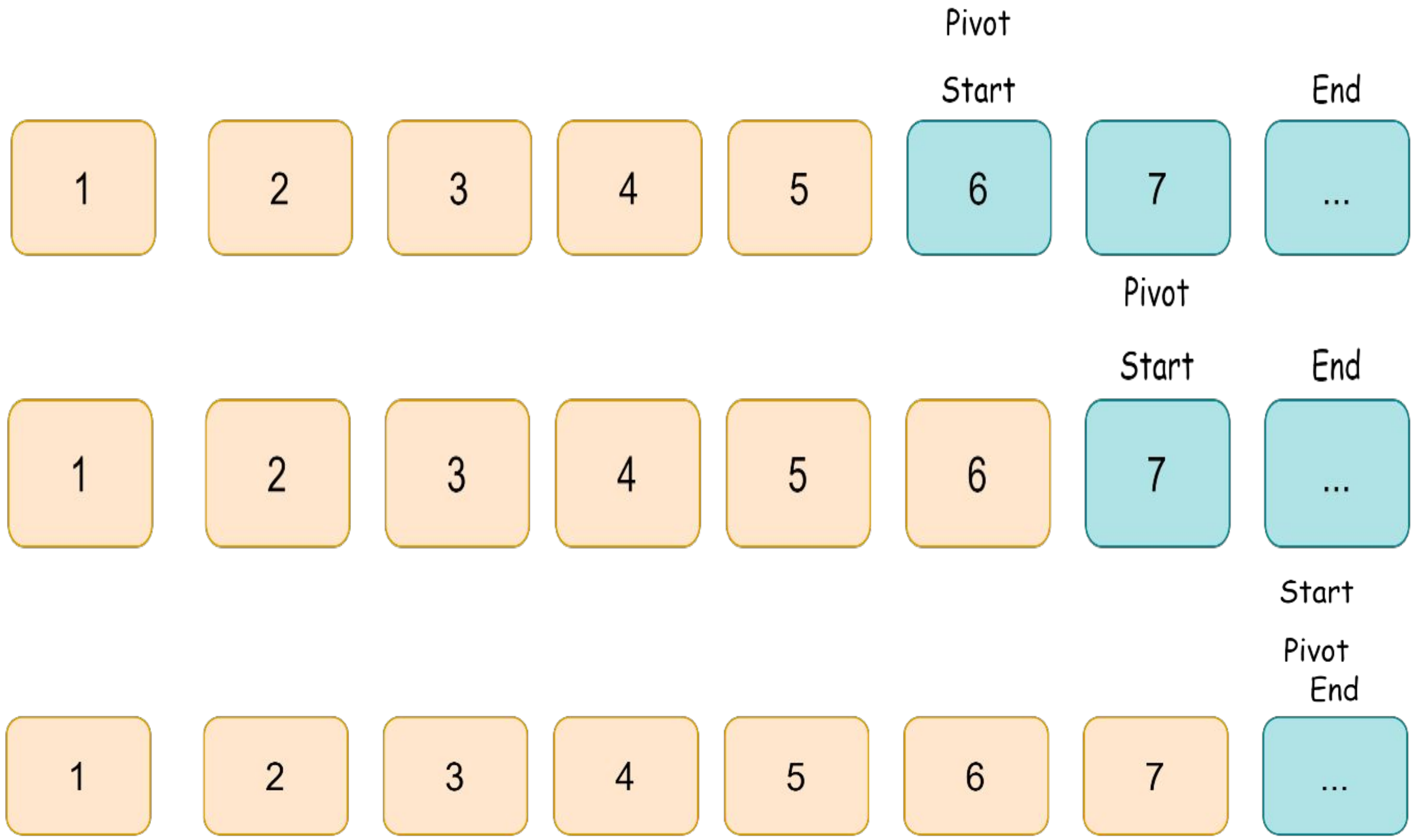

Pivot
Start
End
1
2
3
4
5
6
7
...


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


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

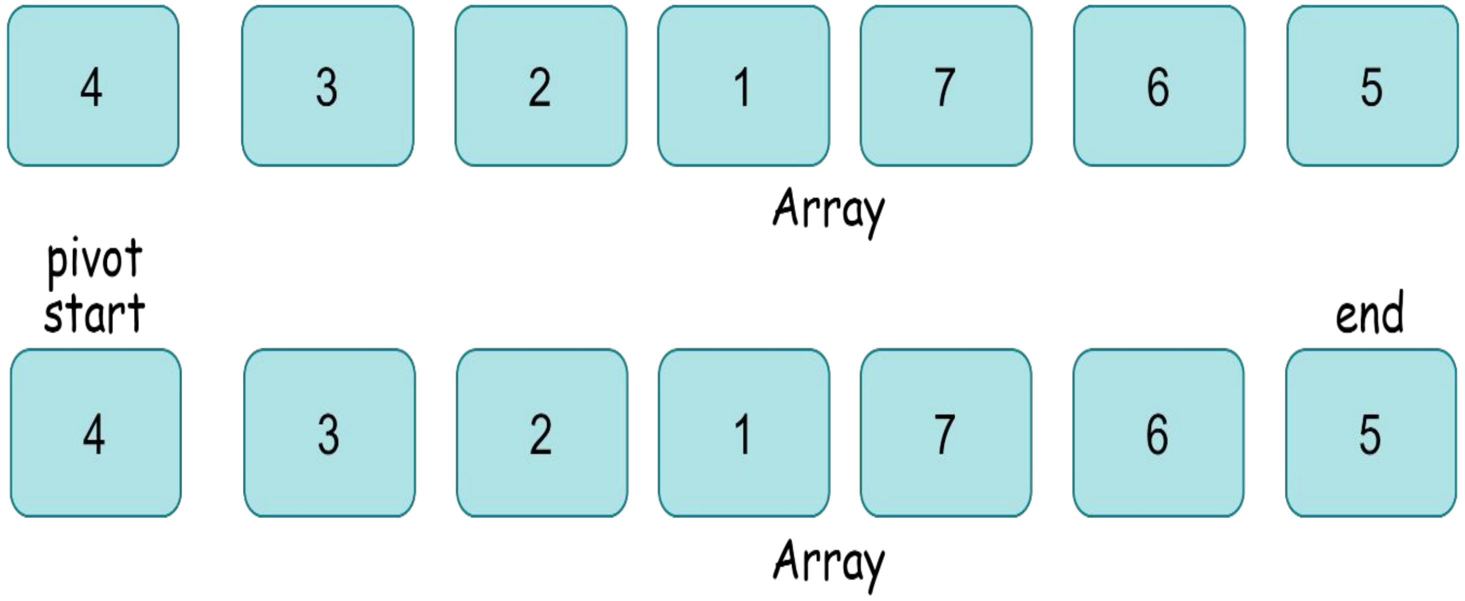

Pivot
Start
End
1
2
3
4
5
6
7
...

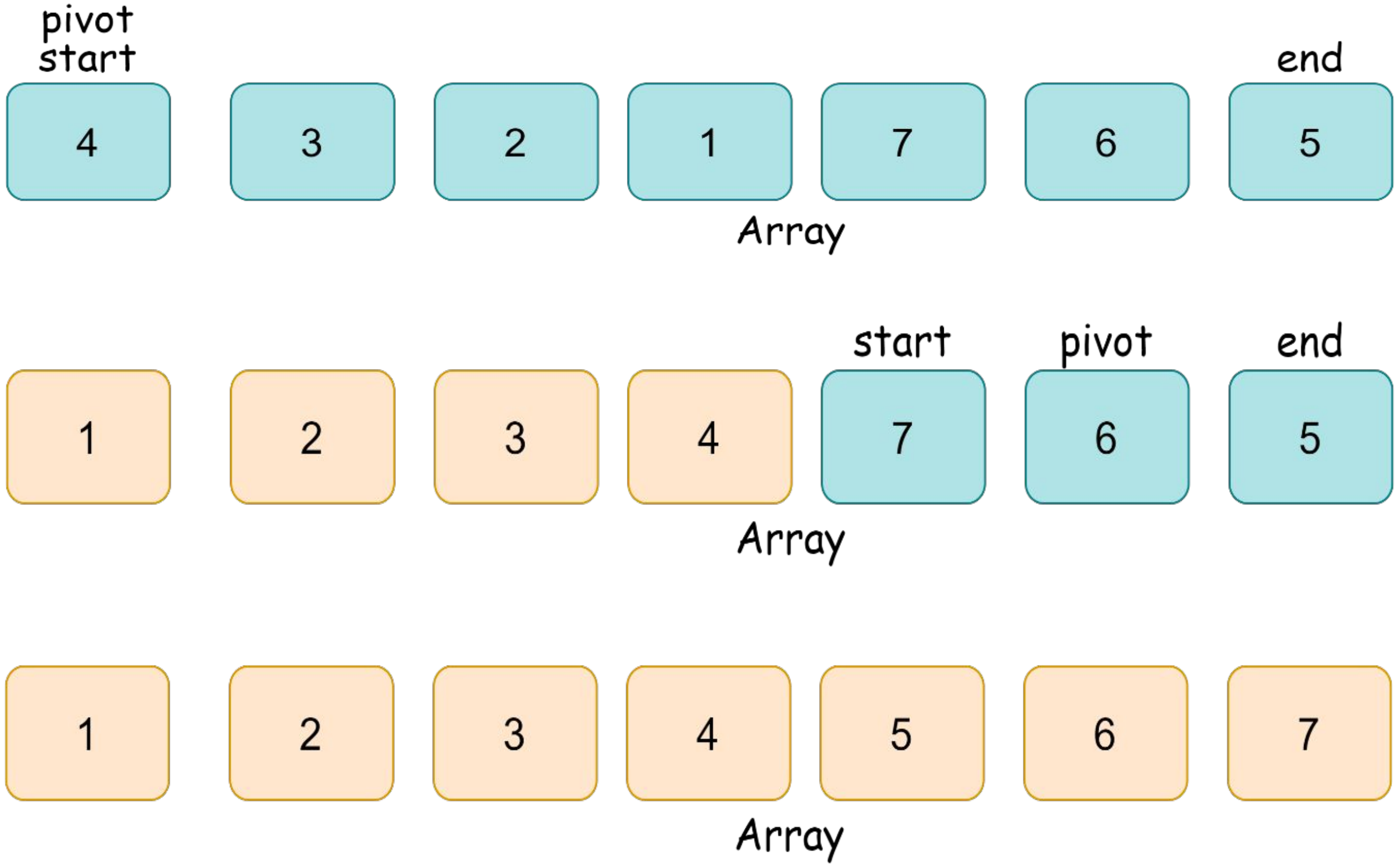

Pivot
Start
End
1
2
3
4
5
6
7
...


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
...

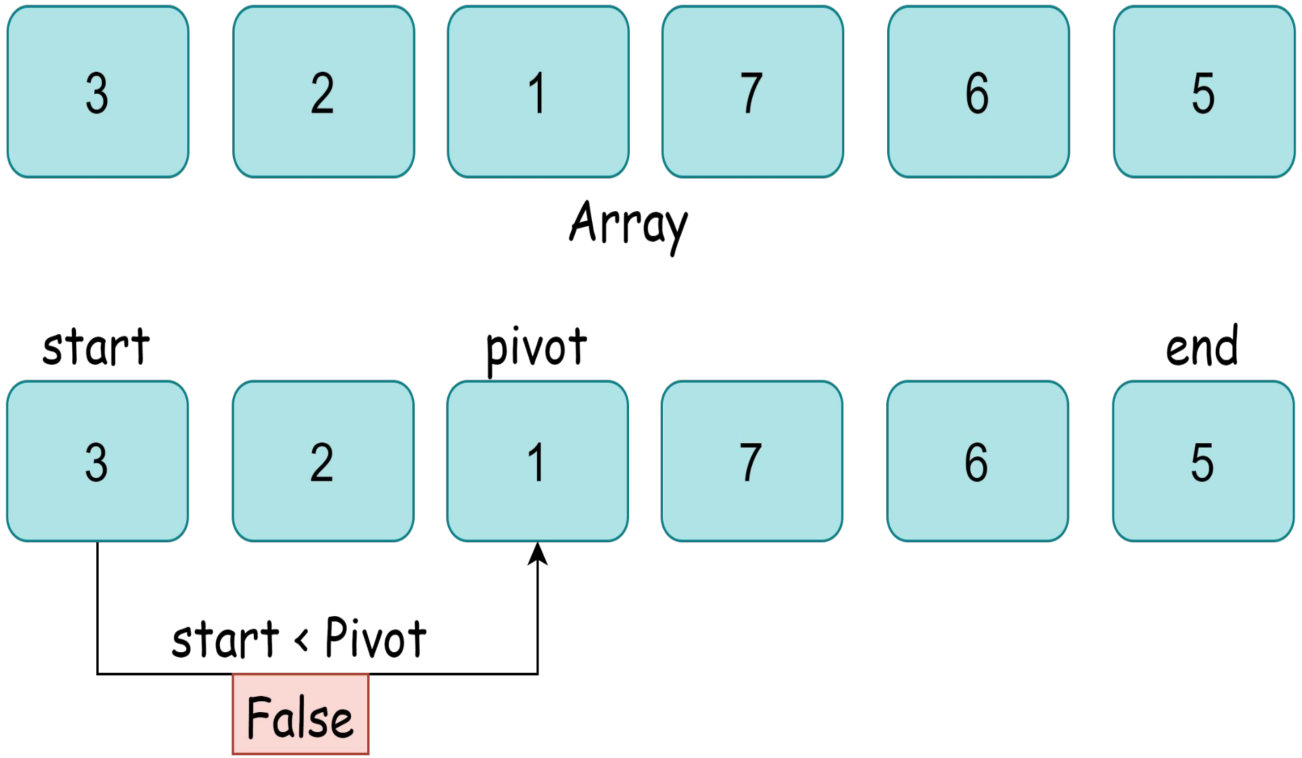
Best case

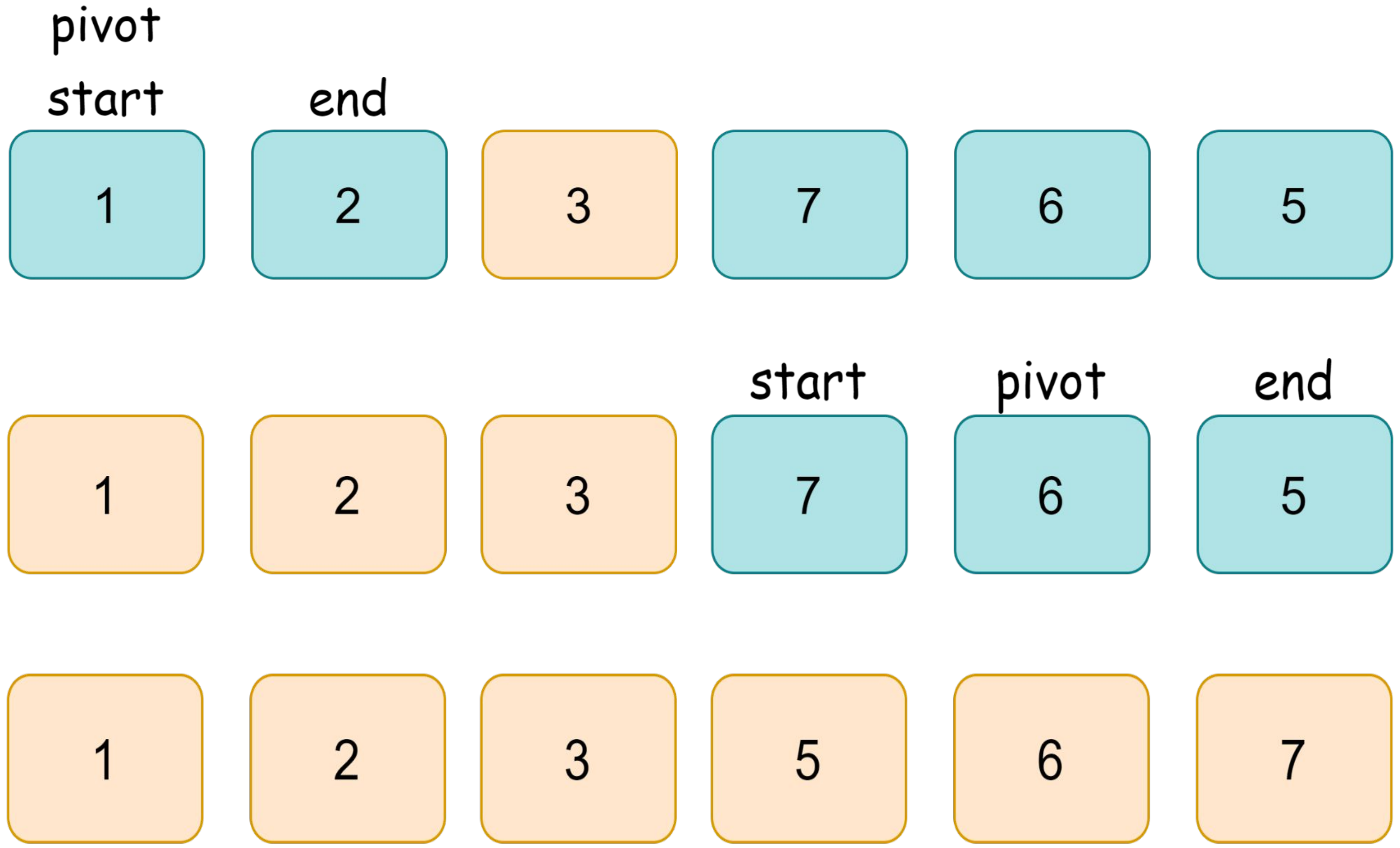
The Best case occur only when the array is partitioned into two sub arrays having the same sub array length.
Example 1 (both left and right sub array of same length)




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)

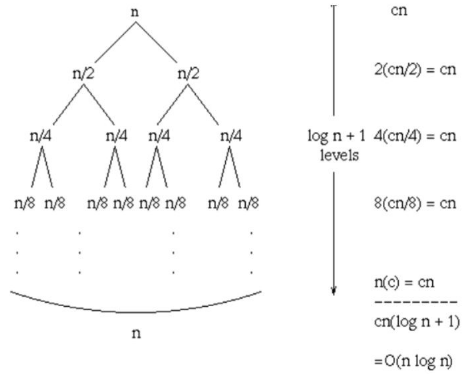



pivot
start
end
1
2
3
7
6
5
start
pivot
end
1
2
3
7
6
5
1
2
3
5
6
7

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
```

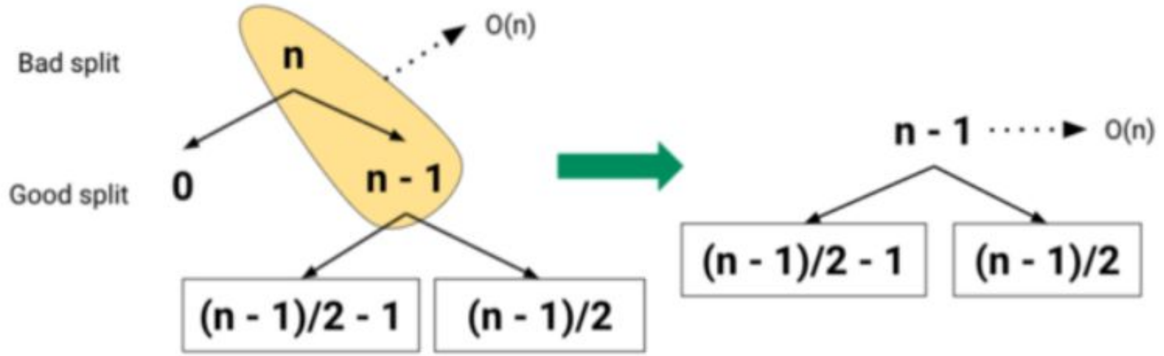
The quick sort's best case time complexity is O. (nlogn)



Average Case

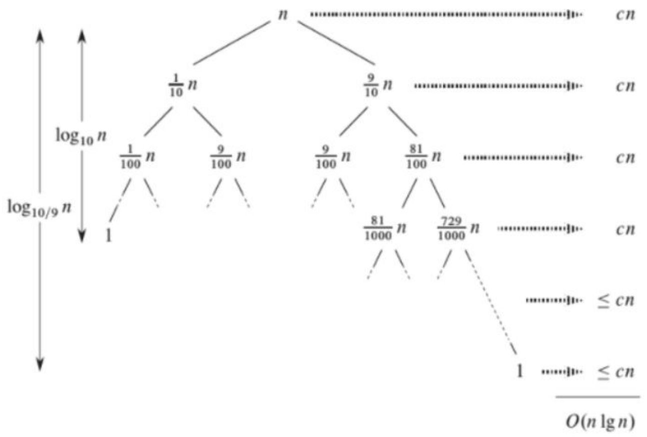
The behaviour of quicksort relies upon the relative order of the values of input.

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

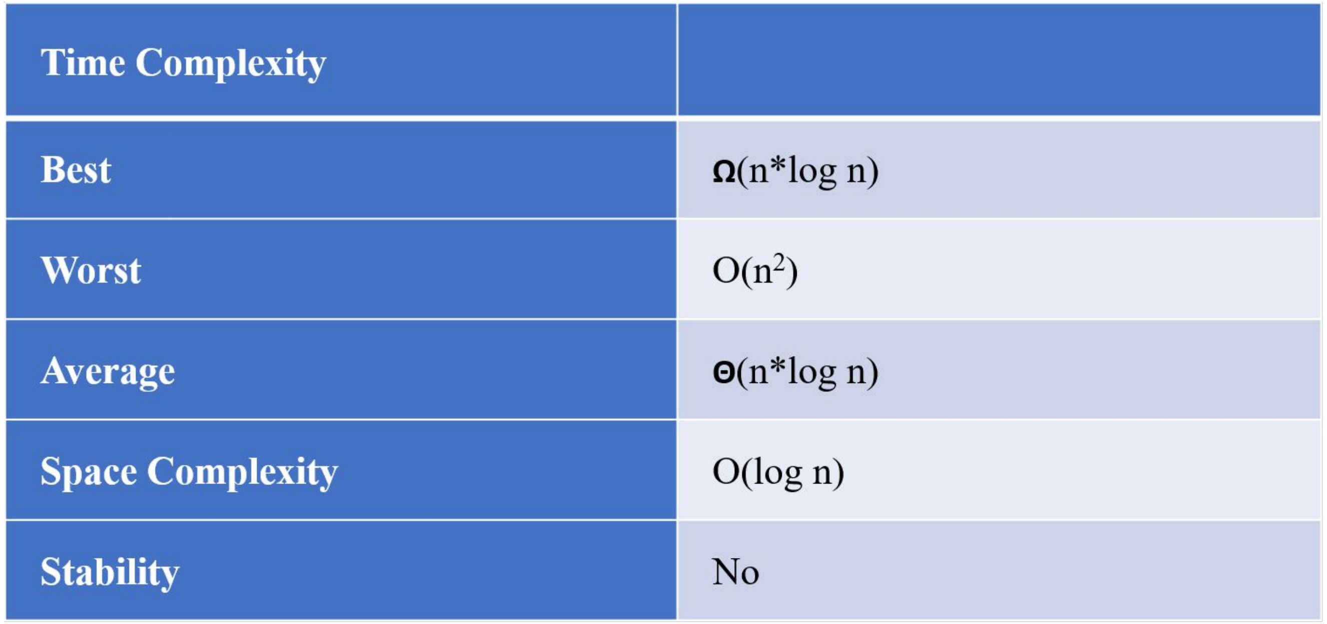


Average Case

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



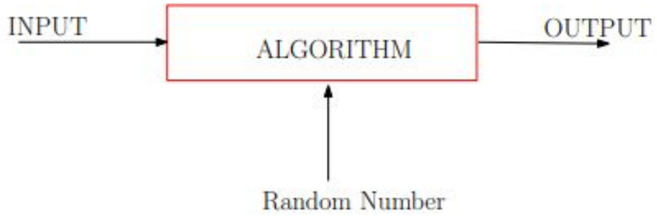
Time Complexity	
Best	Ω(n*log n)
Worst	$O(n^2)$
Average	Θ(n*log n)
Space Complexity	O(log n)
Stability	No

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)

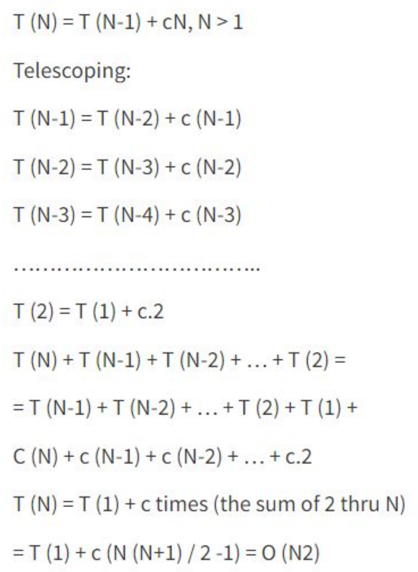
code link - https://jsfiddle.net/saravananslb/6jq2Lu1x/

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)



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

Telescoping:

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

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

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

……………………………………

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

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

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

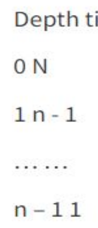
$C(N) + c(N-1) + c(N-2) + \dots + c.2$

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

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

Worst-Case Analysis

The pivot is the smallest (first element) or the largest (last element) in the array



Depth time

0 N

1 n - 1

… …

n – 1 1

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).

$$T(N) = T(i) + T(N - i - 1) + cN$$

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

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

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:

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

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

After crossing the equal terms:

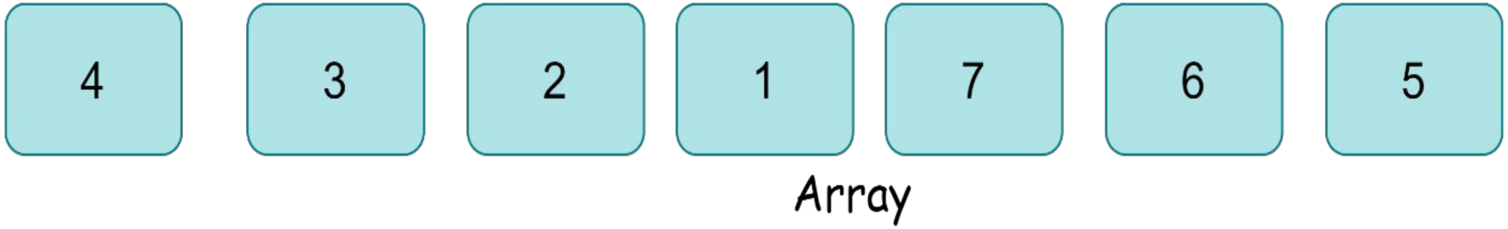
$T(N)/N = T(1) + c * \log N$

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

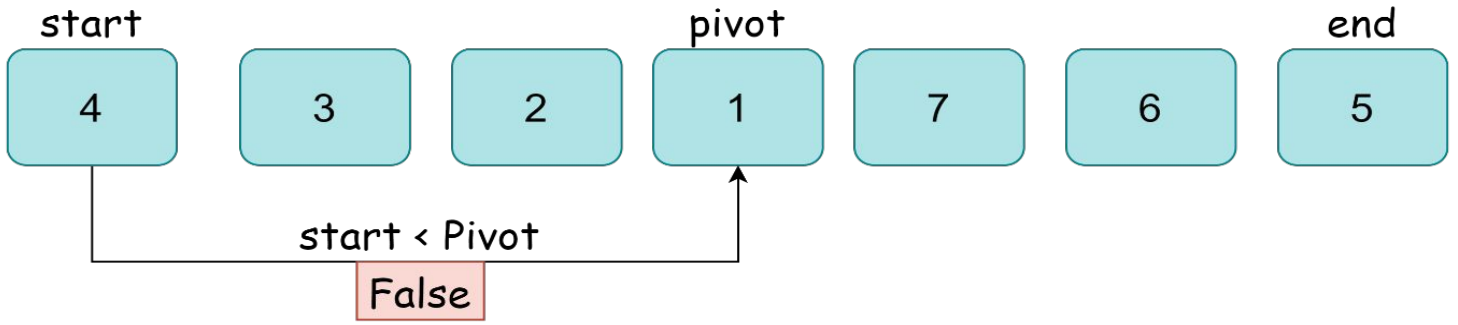
Quick Select Algorithm

Quick Select is similar to the quicksort algorithm. but the only difference we no need to sort the entire array. It is an optimized way to find the kth smallest/largest element in an unsorted array.

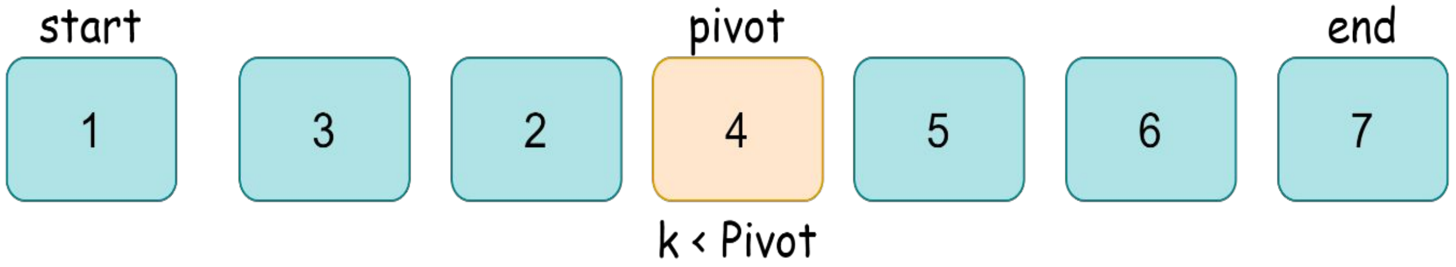
- The partition (splitting) part of the algorithm is the same as that of quick sort algorithm.
- After the partition function split the array into two sub array then check the pivot element with Kth index , so instead of recursing both sides of the pivot index, we recurse only for the part that contains our kth element
- Consider an array of element 4, 3, 2, 1, 7, 6, 5 and K is 5

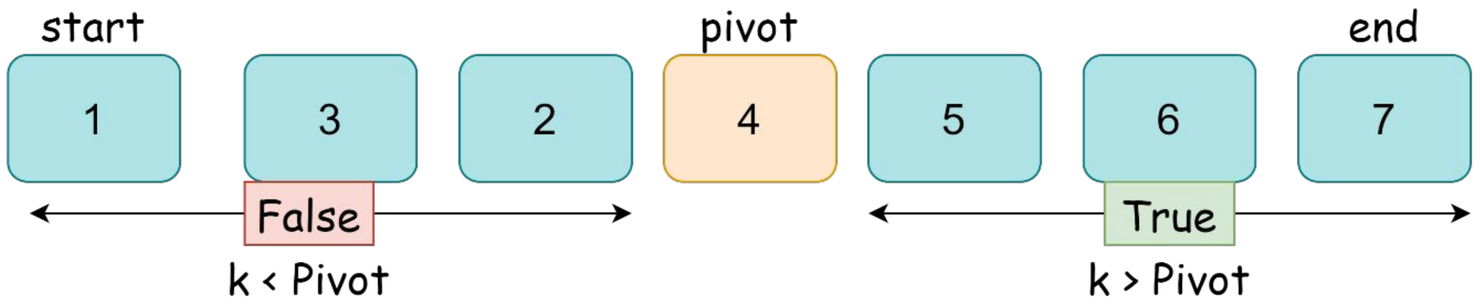


Consider a pivot element as 1 and partition the sub array ie. one having element less than pivot and one having element greater than pivot

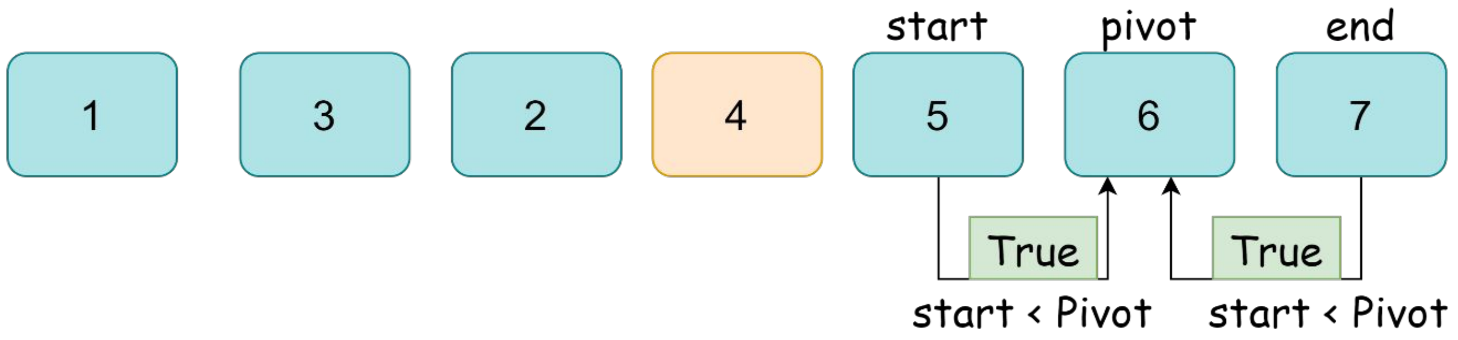


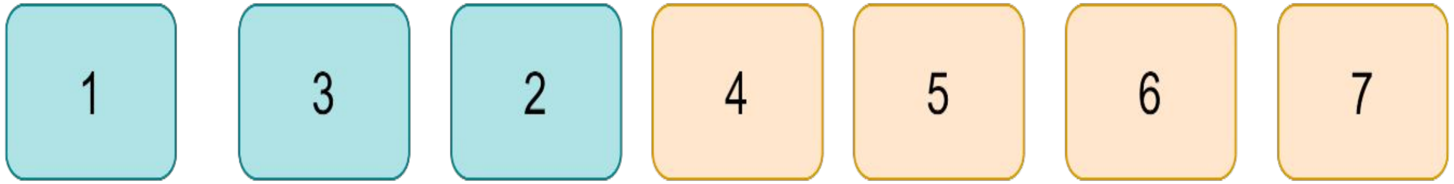
If K is less than pivot then consider the left sub array for next step sort
If K is greater than pivot then consider the right sub array for next step sort

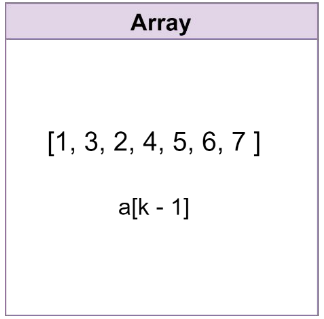




Sort only the right sub array not the left array because sorted pivot element is in position 3rd index which is greater than K







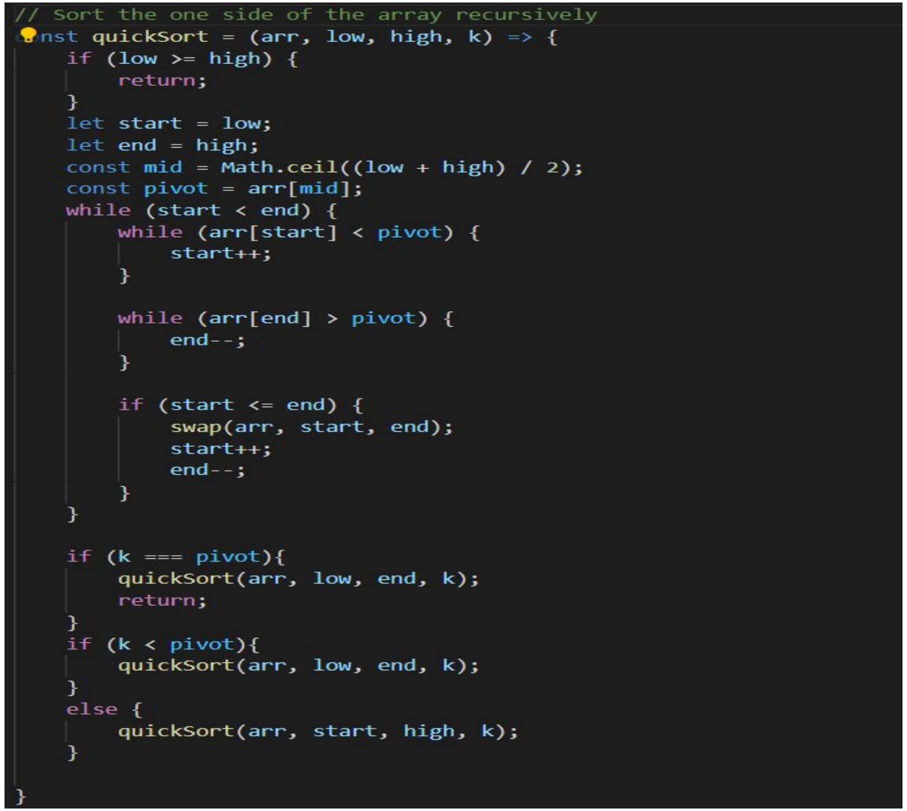
[1, 3, 2, 4, 5, 6, 7]

a[k - 1]

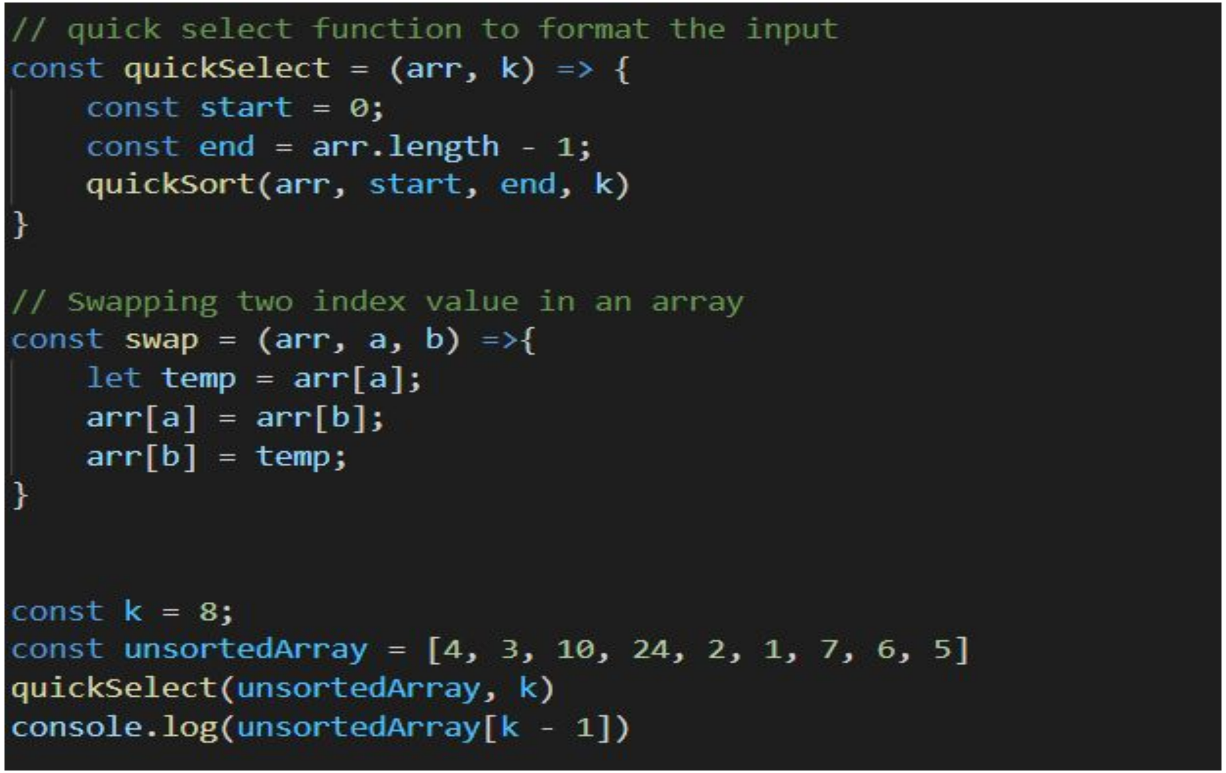
Now you will get the Kth index element in the partially sorted array and by using index will find the Kth value

Code

Code link - https://jsfiddle.net/saravanansIb/q2uo41hc/



```
// Sort the one side of the array recursively
const quickSort = (arr, low, high, k) => {
    if (low >= high) {
        return;
    }
    let start = low;
    let end = high;
    const mid = Math.ceil((low + high) / 2);
    const pivot = arr[mid];
    while (start < end) {
        while (arr[start] < pivot) {
            start++;
        }

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

        if (start <= end) {
            swap(arr, start, end);
            start++;
            end--;
        }
    }

    if (k === pivot){
        quickSort(arr, low, end, k);
        return;
    }
    if (k < pivot){
        quickSort(arr, low, end, k);
    }
    else {
        quickSort(arr, start, high, k);
    }

}
```



```
// quick select function to format the input
const quickSelect = (arr, k) => {
    const start = 0;
    const end = arr.length - 1;
    quickSort(arr, start, end, k)
}

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


const k = 8;
const unsortedArray = [4, 3, 10, 24, 2, 1, 7, 6, 5]
quickSelect(unsortedArray, k)
console.log(unsortedArray[k - 1])
```

Time Complexity Analysis

- Worst case: Worst case occurs when we pick the largest/smallest element as pivot.

  $T(n) = T(n-1) + cn$

  $= T(n-1) + T(n-2) + cn + c(n-1)$

  $= O(n^2)$

- Best case: The best-case occurs when QuickSelect chooses the K-th largest element as the pivot in the very first call. Then, the algorithm performs theta(n) steps during the first (and only) partitioning, after which it terminates. Therefore, QuickSelect is theta(n) in the best case.

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

  $= T(n/4) + c(n/2) + cn$

  $= n(1 + \frac{1}{2} + \frac{1}{4} + \ldots.) = 2n = \Omega(n)$

Time Complexity Analysis

- Average case:
- As a quick select we don't need to sort all the element, sort the half of the earch sub array since our kth element is either greater than pivot or smaller than pivot.
- $n + 1/2\ n + 1/4\ n + 1/8\ n + \ldots.. < 2\ n$
- Quick select allows us to solve the problem in $\Theta\ (n)$ on an average. Using randomized pivot

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

  $= n + n/2 + T(n/4)$

  $= n + n/2 + n/4 + \ldots\ n/n$

  $= n(1 + 1/2 + 1/4 + \ldots + 1/2\text{^}\log2(n)) = n\ (1/(1-(1/2))) = 2n$

The time complexity for the average case for quick select is O(n) (reduced from O(nlogn) — quick sort). The worst case time complexity is still $O(n^2)$ but by using a random pivot, the worst case can be avoided in most cases. So, on an average quick select provides a O(n) solution to find the kth largest/smallest element in an unsorted list.

Solving Order Static Problems

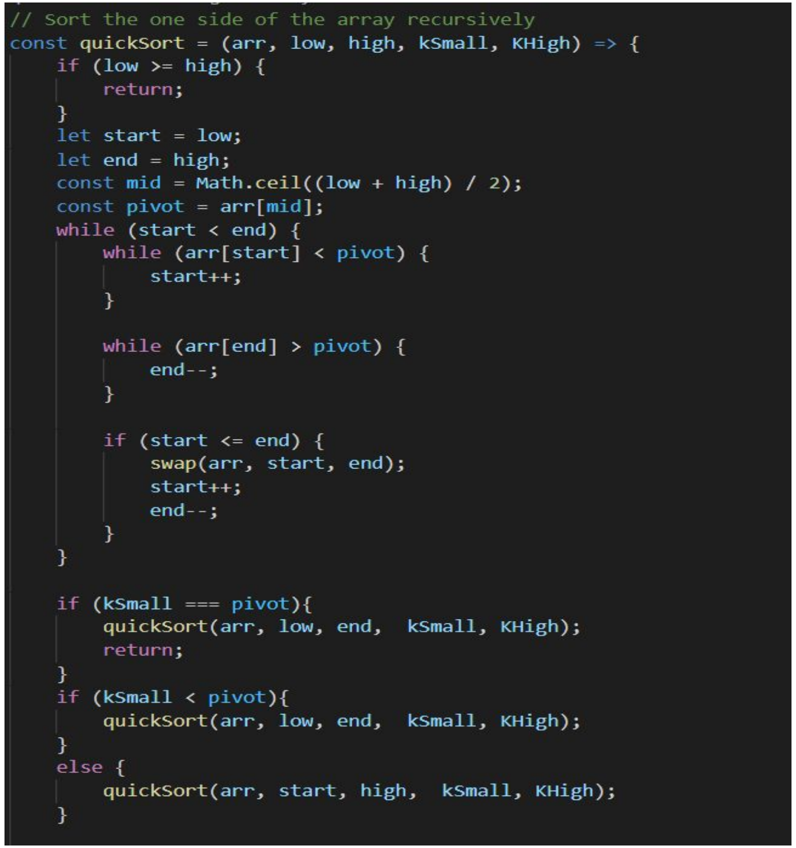
Quick Select is a variation of the quicksort algorithm. It is an optimized way to find the kth smallest/largest element in an unsorted array. This should be the starting part for explaining order statistics problem.

Done

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

1. Sum and product of k smallest and k largest numbers in the array

Given input of array [4, 3, 2, 10, 24, 2, 1, 7, 6, 5]



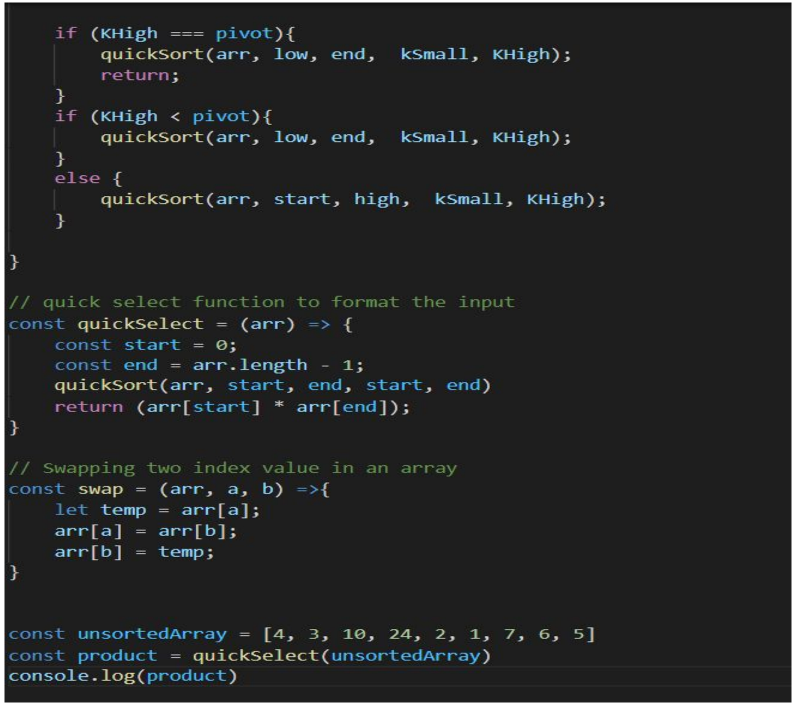
```
// Sort the one side of the array recursively
const quickSort = (arr, low, high, kSmall, KHigh) => {
    if (low >= high) {
        return;
    }
    let start = low;
    let end = high;
    const mid = Math.ceil((low + high) / 2);
    const pivot = arr[mid];
    while (start < end) {
        while (arr[start] < pivot) {
            start++;
        }

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

        if (start <= end) {
            swap(arr, start, end);
            start++;
            end--;
        }
    }

    if (kSmall === pivot){
        quickSort(arr, low, end,  kSmall, KHigh);
        return;
    }
    if (kSmall < pivot){
        quickSort(arr, low, end,  kSmall, KHigh);
    }
    else {
        quickSort(arr, start, high,  kSmall, KHigh);
    }
```

Explanation:
First split the array into two sub array for getting the smallest and largest element.

Next choose the pivot in the left array and again split into two sub array then check if the pivot is lesser or greater than K

If k less than pivot choose the left sub array if not choose the right sub array
Recursively do this process till find the Kth smallest element

Do the same for Kth largest element



```
    if (KHigh === pivot){
        quickSort(arr, low, end,  kSmall, KHigh);
        return;
    }
    if (KHigh < pivot){
        quickSort(arr, low, end,  kSmall, KHigh);
    }
    else {
        quickSort(arr, start, high,  kSmall, KHigh);
    }

}

// quick select function to format the input
const quickSelect = (arr) => {
    const start = 0;
    const end = arr.length - 1;
    quickSort(arr, start, end, start, end)
    return (arr[start] * arr[end]);
}

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


const unsortedArray = [4, 3, 10, 24, 2, 1, 7, 6, 5]
const product = quickSelect(unsortedArray)
console.log(product)
```

Median of medians

- Median of medians is an approximate (median) selection algorithm, frequently used to supply a good pivot for an exact selection algorithm, mainly the quickselect, that selects the kth smallest element of an initially unsorted array.
- It is based on divide and conquer algorithm in that it will returns a pivot that in the worst case it will divide a list of unsorted array elements into sub-problems of size 3n/10 and 7n/10 assuming we choose a sublist size of 5.
- Let us consider 10 element and break this into 5 and 5 elements and for choosing the pivot it will make a partition based on the element (left and right sub array) then find the middle element as pivot for the first 5 elements.
- It will provide a good pivot that in the worst case will give a pivot in the range between 30% and 70 % of the list of size n.

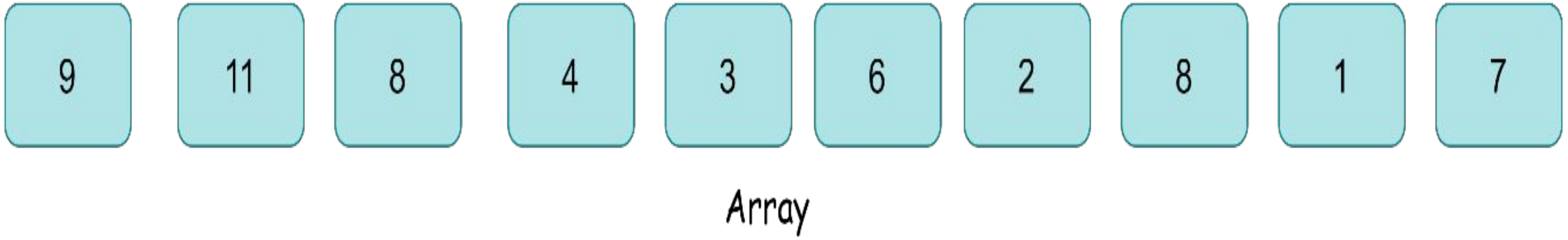
  If you want to choose a good pivot, it is essential for it to be around the middle, 30-70% most of the case the pivot will be around the middle 40% of the list.

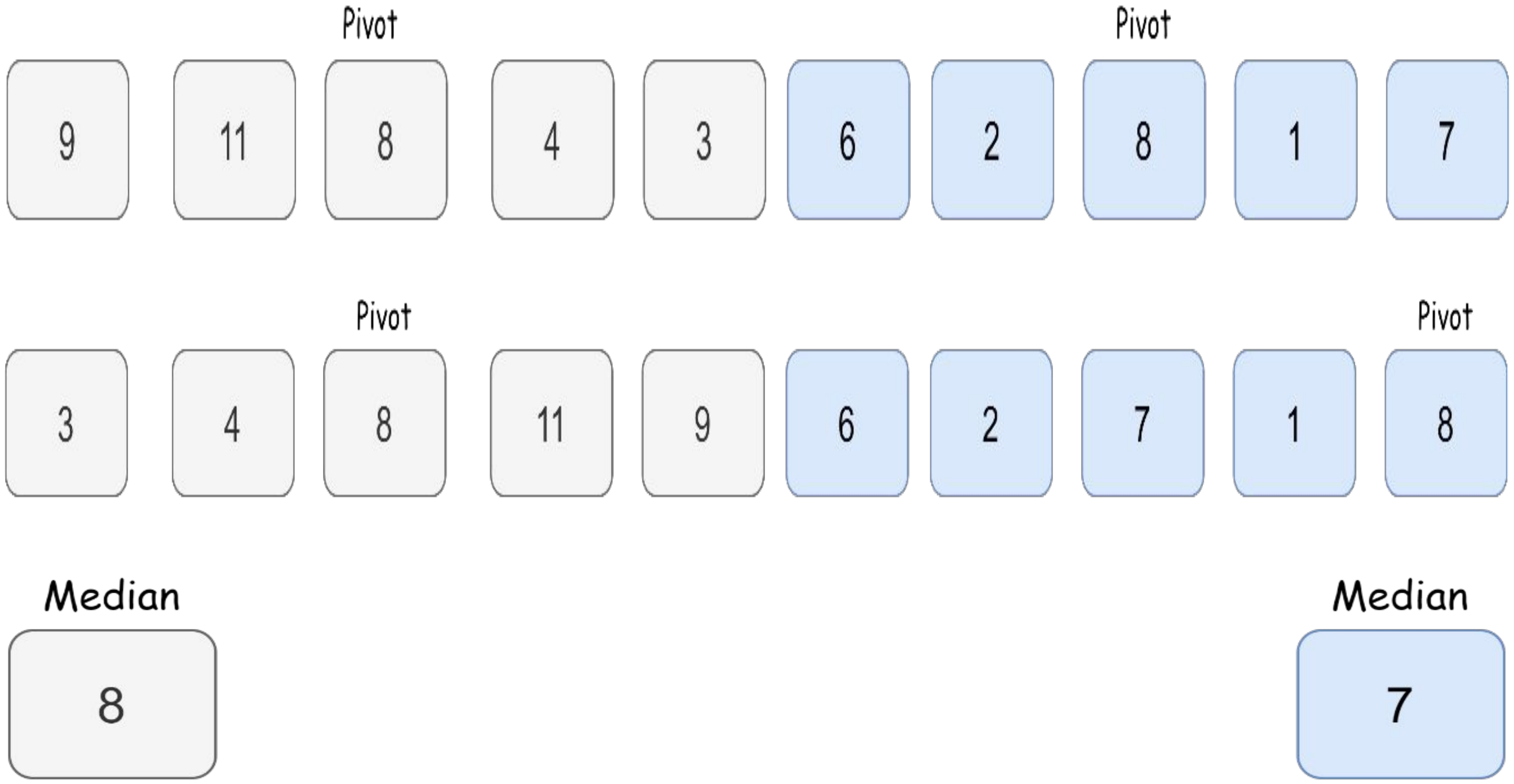
Median of medians

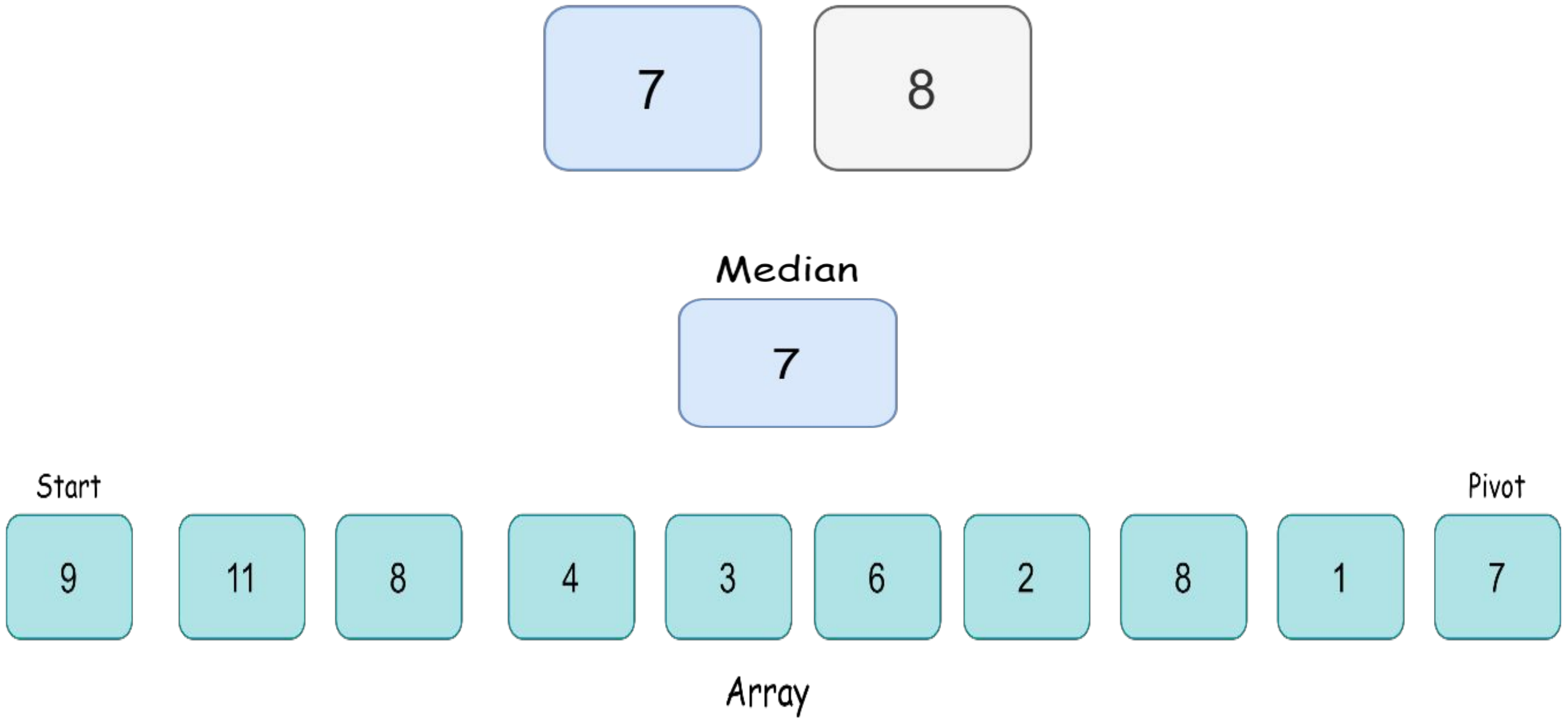
Time and Space Complexity of Median of Medians Algorithm

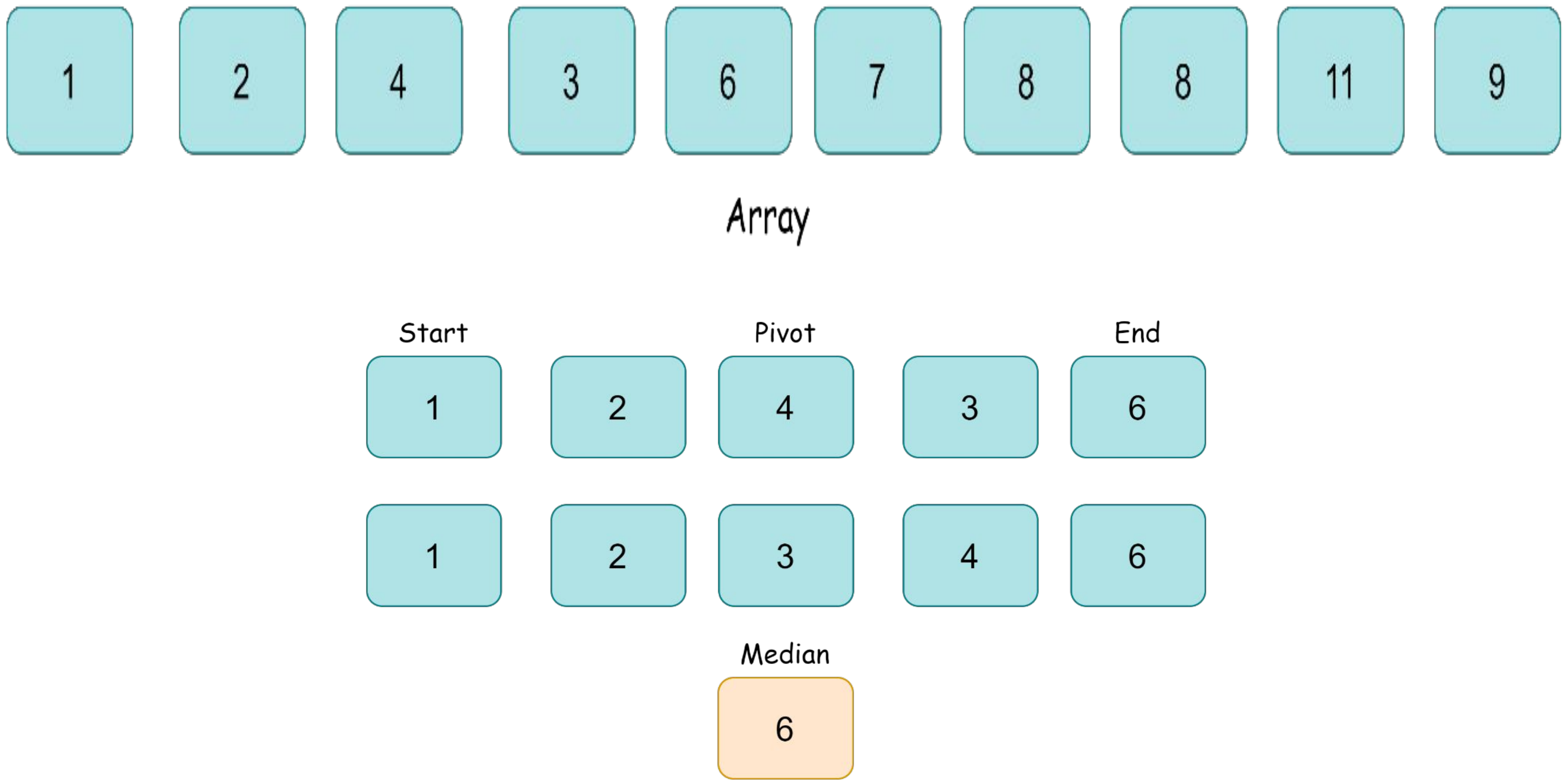
- This algorithm works in O(n) linear time complexity, we traverse the array once to find the medians in the sub array and another time to find the true median to be used as a pivot.
- The space complexity of mom is O(logn) and the memory used will be proportional to the size of the array.

Code link - https://jsfiddle.net/saravananslb/kn2rzjf8/

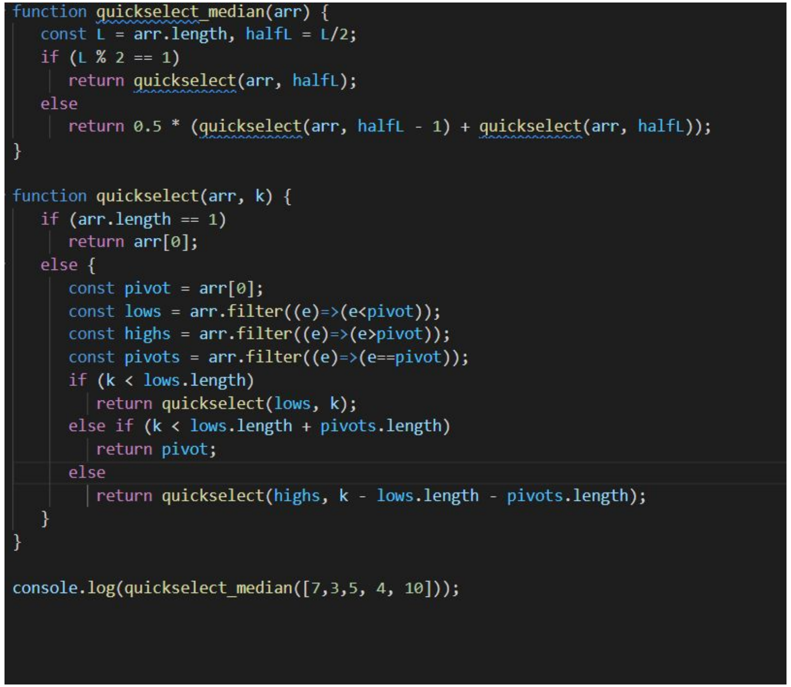



Pivot
Pivot
9
11
8
4
3
6
2
8
1
7
Pivot
Pivot
3
4
8
11
9
6
2
7
1
8
Median
8
Median
7


7
8
Median
7
Start
Pivot
9
11
8
4
3
6
2
8
1
7
Array


1
2
4
3
6
7
8
8
11
9
Array
Start
Pivot
End
1
2
4
3
6
1
2
3
4
6
Median
6

Code link - https://jsfiddle.net/saravananslb/kn2rzjf8/



```
function quickselect_median(arr) {
    const L = arr.length, halfL = L/2;
    if (L % 2 == 1)
        return quickselect(arr, halfL);
    else
        return 0.5 * (quickselect(arr, halfL - 1) + quickselect(arr, halfL));
}

function quickselect(arr, k) {
    if (arr.length == 1)
        return arr[0];
    else {
        const pivot = arr[0];
        const lows = arr.filter((e)=>(e<pivot));
        const highs = arr.filter((e)=>(e>pivot));
        const pivots = arr.filter((e)=>(e==pivot));
        if (k < lows.length)
            return quickselect(lows, k);
        else if (k < lows.length + pivots.length)
            return pivot;
        else
            return quickselect(highs, k - lows.length - pivots.length);
    }
}

console.log(quickselect_median([7,3,5, 4, 10]));
```

The algorithm is as follows,

1. Divide the array into sub arrays of size n, assume 5.
2. Loop through the whole array in sizes of 5, considering our array is divisible by 5.
3. For n/5 sub arrays, use select brute-force subroutine algorithm to select a median m, which is in the 3rd place out of 5 elements.
4. Add medians obtained from the sub arrays to the array M.
5. Use quick Select recursively to find the median from array M, The median obtained is the best pivot.
6. Stop the recursion loop once the base case is hit, ie., when the sub array becomes small enough. Use Select brute-force subroutine to find the median.

Note: We used each splitted sub arrays of size 5 because selecting a median from a list when the size of the array is an odd number is easier. Even numbers require additional computation. We can also select 7 or any other odd number as we shall see in the proofs below.

Median of medians vs quick select

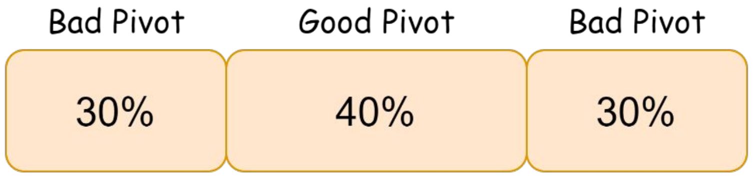
To avoid the O(n^2) worst case scenario for quick select, we will pick either of the below in the quick sort

1. Randomly choose a pivot index
2. Use median of medians (MoM) to select an approximate median and pivot around that

When using MoM with quick select, we can guarantee worst case O(n). When choosing pivot as random, we can't guarantee worst case O(n), but the probability of the algorithm going to O(n^2) should be extremely small.

The overhead cost of median of medians is much more than choosing random pivot, where the latter adds little to no additional complexity.

- Assuming quickselect is truly using a random choice of pivot at each step, you can prove that not only is the expected runtime O(n), but that the probability that its runtime exceeds Θ(n log n) is very, very small (at most 1 / nk for any choice of constant k). So in that sense, if you have the ability to select pivots at random, quickselect will likely be faster.



- However, not all implementations of quickselect use true randomness for the pivots, and some use deterministic pivot selection algorithms. This, unfortunately, can lead to pathological inputs that trigger the Θ(n2) worst-case runtime, which is a problem if you have adversarially-chosen inputs.

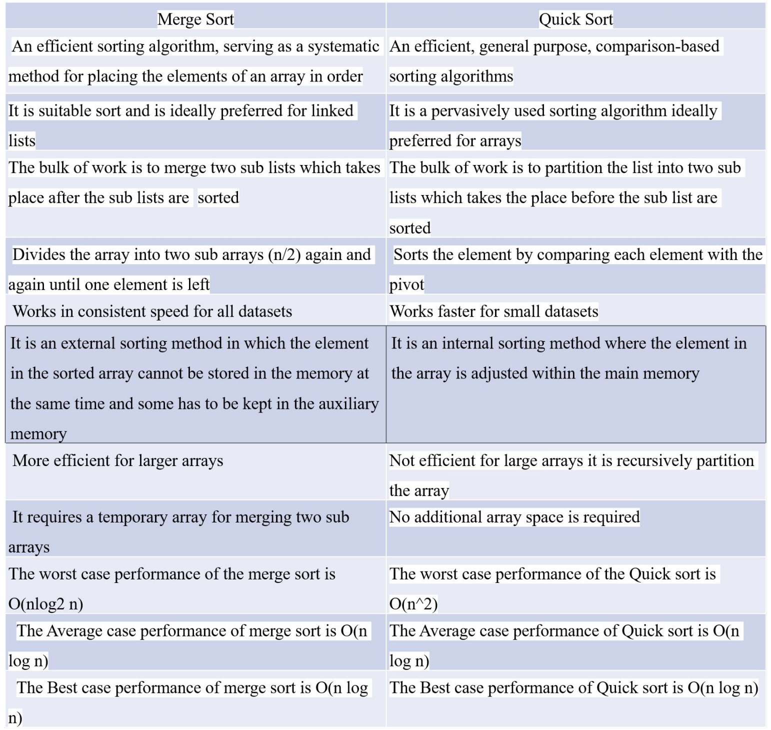
- Once nice compromise between the two is introselect. The basic idea behind introselect is to use quickselect with a deterministic pivot selection algorithm. As the algorithm is running, it keeps track of how many times it's picked a pivot without throwing away at least 30% the input array.

If that number exceeds some threshold, it stops using a random pivot choice and switches to the median-of-medians approach to select a good pivot, forcing a 30% size reduction.

This approach means that in the common case when quickselect rapidly reduces the input size, introselect is basically identical to quickselect with a tiny bookkeeping overhead. However, in cases where quickselect would degrade to quadratic, introselect stops and switches to the worst-case efficient median-of-medians approach, ensuring the worst-case runtime is O(n).

This gives you, essentially, the best of both worlds - it's fast on average, and its worst-case is never worse than O(n).

Merge Sort vs Quick Sort



Merge Sort	Quick Sort
An efficient sorting algorithm, serving as a systematic method for placing the elements of an array in order	An efficient, general purpose, comparison-based sorting algorithms
It is suitable sort and is ideally preferred for linked lists	It is a pervasively used sorting algorithm ideally preferred for arrays
The bulk of work is to merge two sub lists which takes place after the sub lists are sorted	The bulk of work is to partition the list into two sub lists which takes the place before the sub list are sorted
Divides the array into two sub arrays (n/2) again and again until one element is left	Sorts the element by comparing each element with the pivot
Works in consistent speed for all datasets	Works faster for small datasets
It is an external sorting method in which the element in the sorted array cannot be stored in the memory at the same time and some has to be kept in the auxiliary memory	It is an internal sorting method where the element in the array is adjusted within the main memory
More efficient for larger arrays	Not efficient for large arrays it is recursively partition the array
It requires a temporary array for merging two sub arrays	No additional array space is required
The worst case performance of the merge sort is O(nlog2 n)	The worst case performance of the Quick sort is O(n^2)
The Average case performance of merge sort is O(n log n)	The Average case performance of Quick sort is O(n log n)
The Best case performance of merge sort is O(n log n)	The Best case performance of Quick sort is O(n log n)

MCQ

1. Which of the below sorting algorithms is the fastest?

A) Merge sort

B) Quick sort

C) Insertion sort

D) Shell sort

Answer: B

Explanation: Quick sort is the fastest known sorting algorithm because of its highly optimized inner loop.

2. Which one of the below is best method to choose a pivot element?

A) Choosing a random element as pivot

B) Choosing the first element as pivot

C) Choosing the last element as pivot

D) Median-of-three partitioning method

Answer: A

Explanation: This is the safest method to choose the pivot element since it is very unlikely that a random pivot would consistently provide a poor partition

3. Which approach of selecting a pivot element is the worst?

A) mid element as pivot

B) last element as pivot

C) median-of-three partitioning

D) first element as pivot

Answer: D

Explanation: Choosing the first element as pivot is the worst method because if the input is pre-sorted or in reverse order, then the pivot provides a poor partition.

4. What is the average running time of a quick sort algorithm?

A) O(N2)

B) O(N)

C) O(N log N)

D) O(log N)

Answer: C

Explanation: The best case and average case analysis of a quick sort algorithm are mathematically found to be O(N log N).

5. How many sub-arrays can the quick sort algorithm split the entire array into?

A) one

B) two

C) three

D) four

Answer: B

Explanation: Quick sort split the array into two sub arrays, left and right sub arrays

Practice Problem

1. Write a program to find the Kth largest element in the array using quick select.

2. Write a program to sort the element in the array by descending order using random pivot.

Upcoming Teaser

- Problem Solving on Sorting

Thank You!