

# QuickSelect - QuickSort - Partition

# Content

- QuickSelect
- QuickSort
- Partition




# Introduction

## Bubble Sort, Selection Sort and Insertion sort:

- Worst runtime  $O(n^2)$
- Inefficient when an array is large


## Quicksort and Merge Sort:

- Better running times
    - Merge sort runs in  $O(n \log n)$  time in all cases
    - Quicksort runs in  $O(n \log n)$  time in the best and average cases, but its worst-case running time is  $O(n^2)$
- 

# Divide and Conquer

Merge sort, quickSelect and quickSort employ a common algorithmic paradigm based on recursion.

A divide-and-conquer algorithm recursively breaks down a problem into two or more sub-problems of the same or related type, until these become simple enough to be solved directly. (Wikipedia)

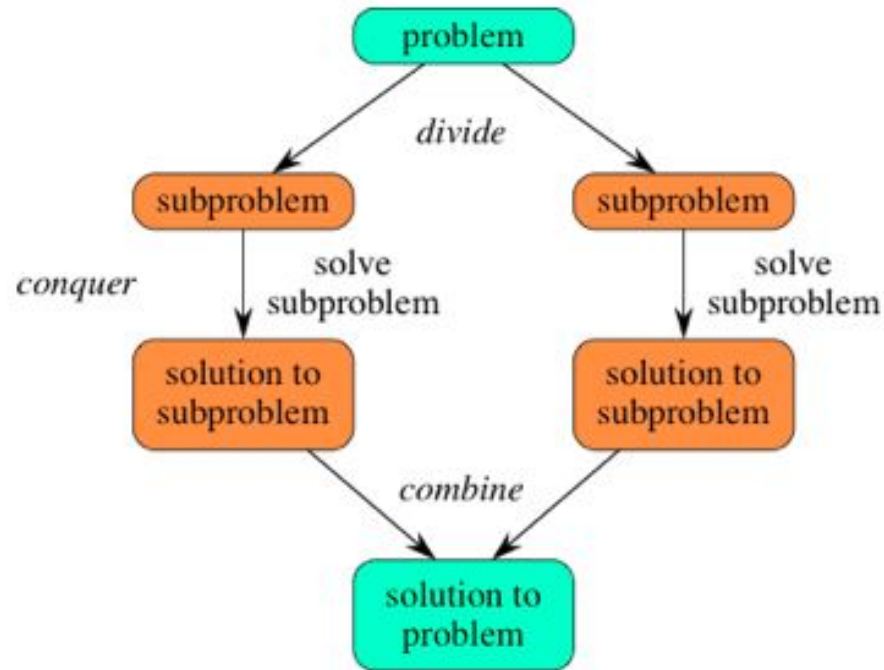


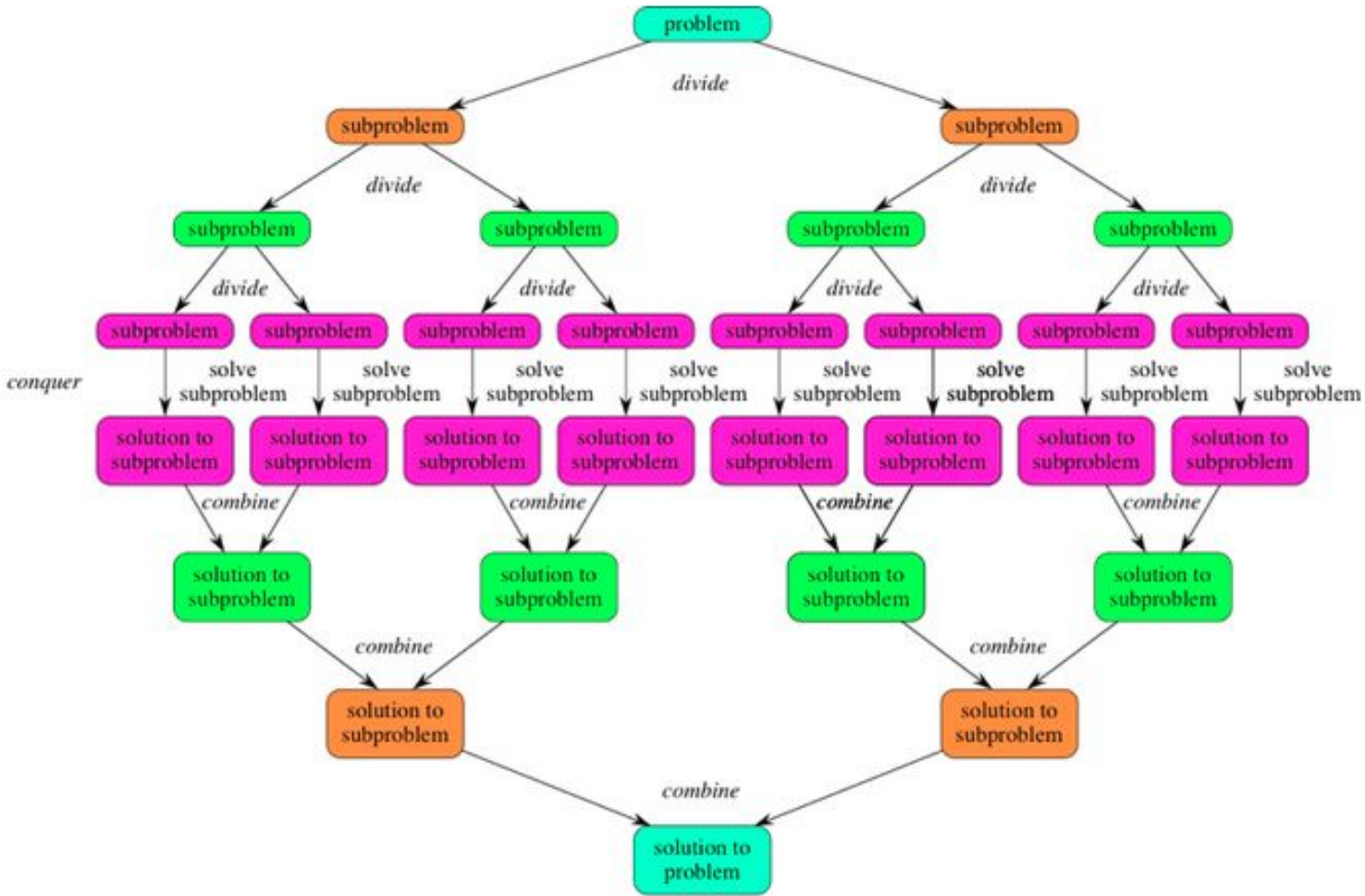
## Divide and conquer algorithm parts

1. **Divide** the problem into a number of subproblems that are smaller instances of the same problem.
2. **Conquer** the subproblems by solving them recursively.
3. **Combine** the solutions to the subproblems into the solution for the original problem.



# Divide and Conquer





# QuickSelect - Algorithm

QuickSelect is a selection algorithm to find the k-th smallest element in an unordered list.

Input: `arr[] = {7, 10, 4, 3, 20, 15}`

`k = 3`

Output: 7

How would you solve this algorithm?





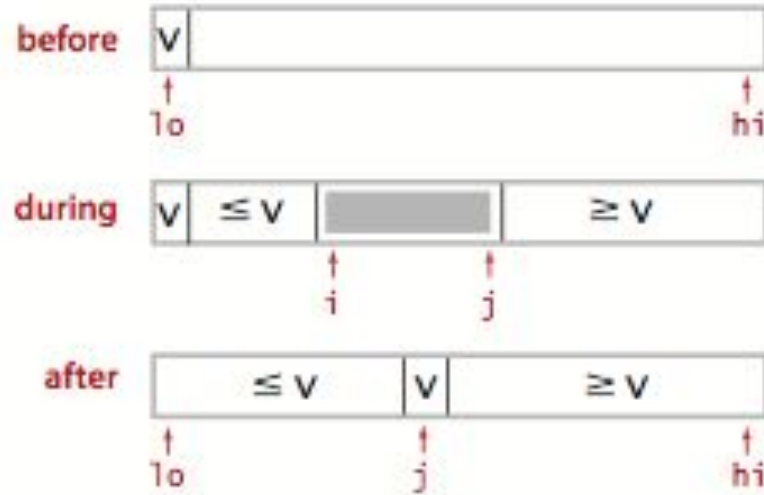
# A possible solution

Making Partitions???

We might use the partition on the input array recursively and work on only one side of the partition. We will choose either the left or right side according to the position of a pivot element.

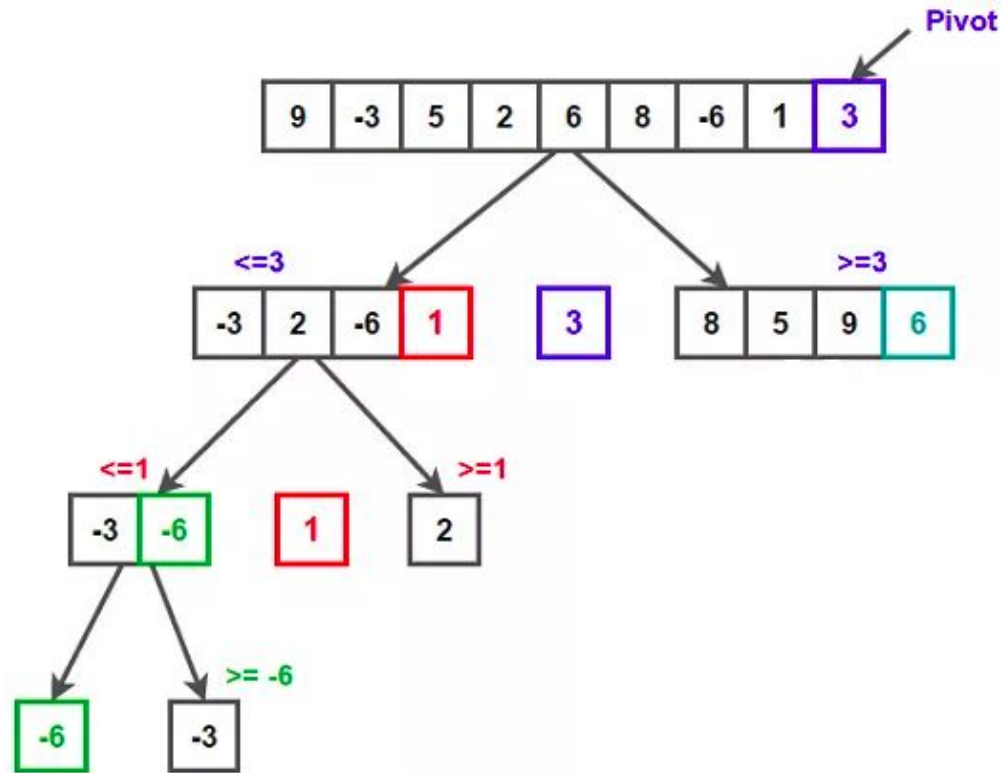


# Partition method



Having the position of the pivot could help find the k-th smallest element?

# Quick Select



# Quick Select Summary

It is a selection algorithm which means it looks for the  $k$ -th smallest element in an unsorted array.

It uses a partition strategy to find if index of the partitioned element is more than  $k$ , then we recur for the left part. If index is the same as  $k$ , we have found the  $k$ -th smallest element and we return. If index is less than  $k$ , then we recur for the right part.

Using a random pivot when defining partitions we can avoid the worst case (largest/smallest element picked as pivot).



# Let's create a partition method in Java

1. Create a folder QuickSelect in your classwork folder (assignments repo)
2. Inside QuickSelect create a file QuickSelect.java
3. Implement a method partition:
  - a. The method must receive these parameters: an array of integer, and start and end positions for the partition (data, 0, data.length - 1)
  - b. The method will randomly choose an index from the array (start and end inclusive). We are going to call this index pivot. (int) (Math.random() \* (upper - lower)) + lower
  - c. If pivot is not zero, swap the element at pivot with the element at index zero
  - d. Elements smaller than the element at pivot should be moved to the left (explained in class)
  - e. Elements larger than the element at pivot should be moved to the right (explained in class)
  - f. Swap the element at pivot to its final position
  - g. Return the index of the final position of the pivot element

```
public static int partition( int [] data, int start, int end){  
  
}
```



# Quick Select - Pseudocode

QuickSelect method to find the Kth smallest element

    pivot = partition(data, start, end)

    if pivot > k - 1

        Recursive call to your QuickSelect - parameters: start, pivot - 1

    Else if pivot < k - 1

        Recursive call to your QuickSelect - parameters: pivot + 1, end

    Else

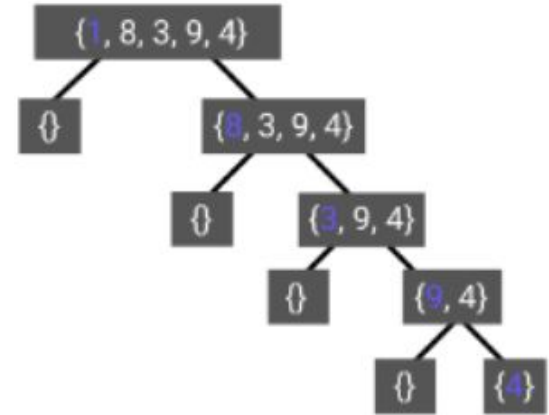
        Kth element found return the value [k-1]



# Quick Select - Time Complexity

Worst case: larger/smallest element picked as pivot

$$\begin{aligned} T(n) &= T(n-1) + cn \\ &= T(n-1) + T(n-2) + cn + c(n-1) \\ &= O(n^2) \end{aligned}$$



The height of the tree will be  $n$  and in top node we will be doing  $N$  operations  
then  $n-1$  and so on until 1

# Quick Select - Time Complexity

Best Case: when we partition the list into two halves and continue with only the half we are interested in.

$$\begin{aligned}T(n) &= T(n/2) + cn \\&= T(n/4) + c(n/2) + cn \\&= n(1 + 1/2 + 1/4 + \dots) \\&= 2n \\&= O(n)\end{aligned}$$





# QuickSort

QuickSort works similar to the QuickSelect:

- It picks an element as a pivot
- It creates partitions based on the picked pivot
- Select sort returns the values of the Kth element while the goal of the QuickSort is to use the partitions logic to sort the array.



# QuickSort versions

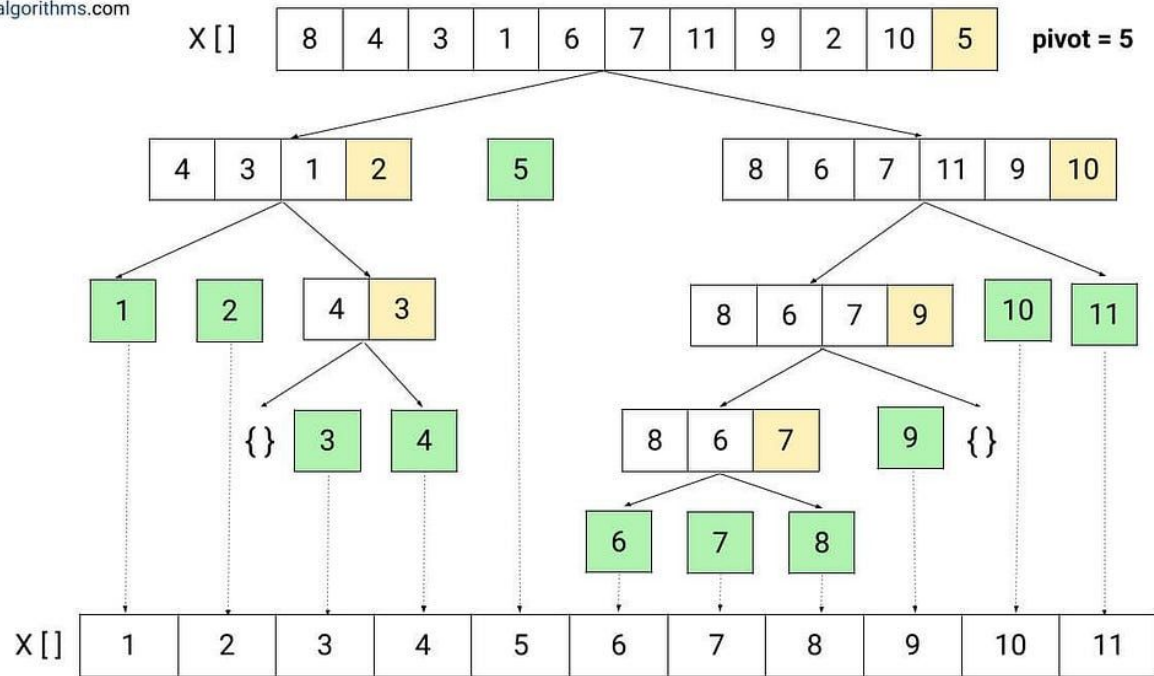
A pivot element for the QuickSort can be picked in different ways:

- Pick median as the pivot.
- Pick a random element as a pivot.
- Always pick the first element as a pivot.
- Always pick the last element as a pivot.



# QuickSort

enjoyalgorithms.com



# QuickSort Algorithm

```
QuickSort(data, start, end)
```

```
    if(start < end)
```

```
        pivot = partition(data, start, end)
```

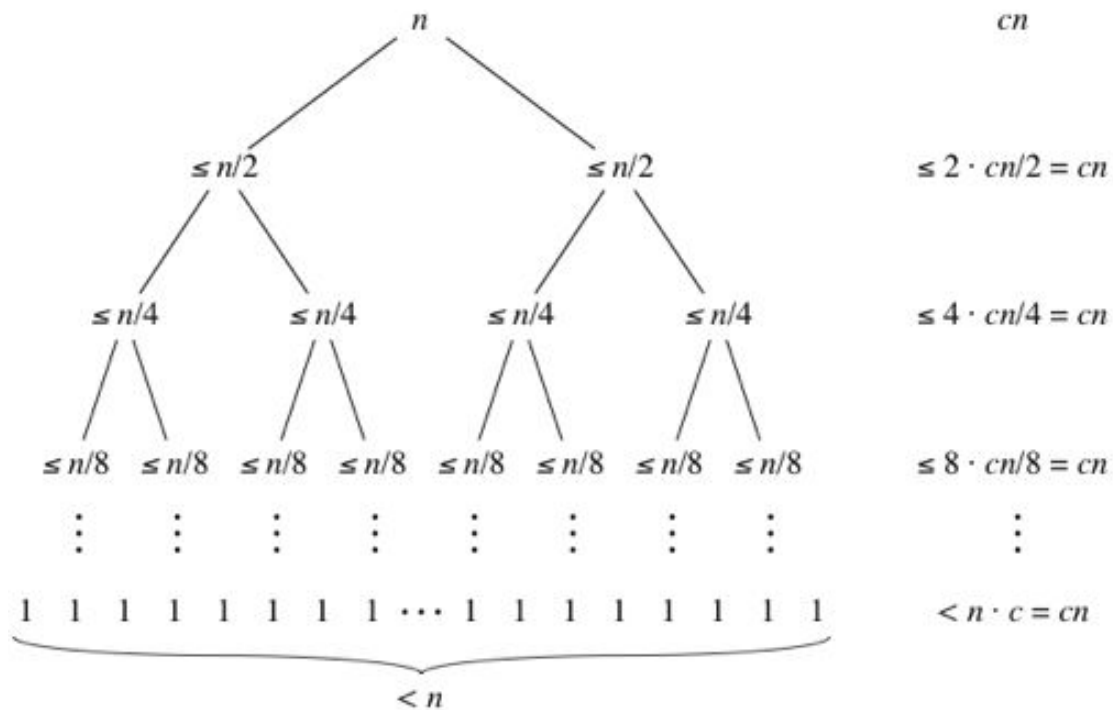
```
        QuickSort(data, start, pivot - 1)
```

```
        QuickSort(data, pivot + 1, end)
```



Subproblem  
size

Total partitioning time  
for all subproblems of  
this size



# QuickSort - Time Complexity

The **average** time complexity of quick sort is  **$O(N \log(N))$** .

The derivation is based on the following notation:

At each step, the input of size  $N$  is broken into two parts say  $J$  and  $N-J$ .

$$T(N) = T(J) + T(N-J) + M(N)$$

where

- $T(N)$  = Time Complexity of Quick Sort for input of size  $N$ .
- $T(J)$  = Time Complexity of Quick Sort for input of size  $J$ .
- $T(N-J)$  = Time Complexity of Quick Sort for input of size  $N-J$ .
- $M(N)$  = Time Complexity of finding the pivot element for  $N$  elements.



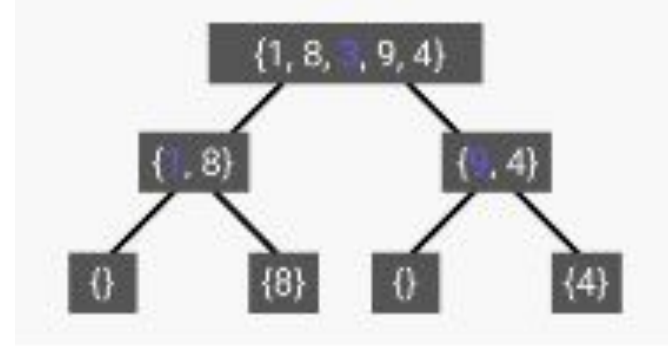
# QuickSort - Time Complexity

**Best case** time complexity  $O(N \log(N))$ .

The best-case occurs when the pivot element is the middle element or near to the middle element

In this case the recursion will look as shown in diagram, as we can see in diagram the height of tree is  $\log N$  and in each level we will be traversing to all the elements with total operations will be  $\log N * N$

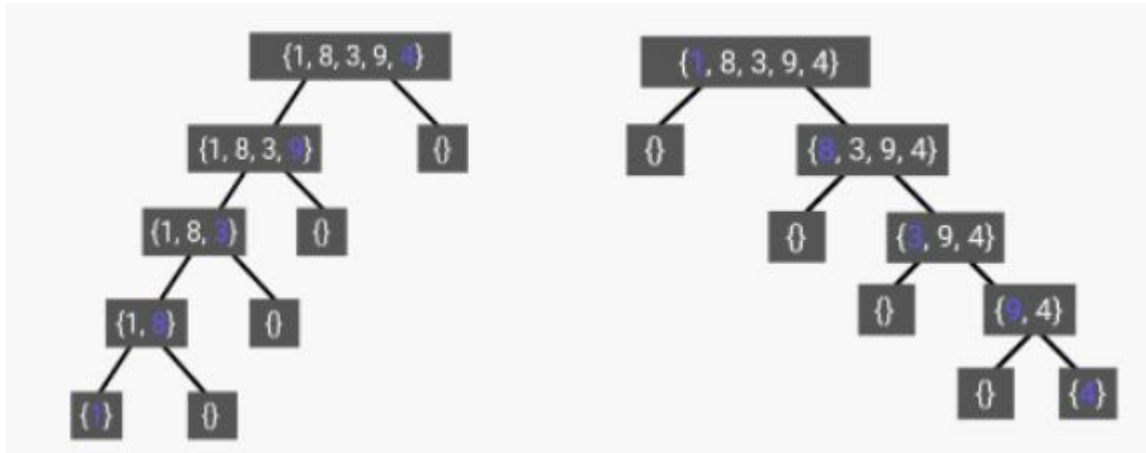
As we have selected mean element as pivot then the array will be divided in branches of equal size so that the height of the tree will be minimum



# QuickSort - Time Complexity

Worst case complexity: In quick sort, worst case occurs when the pivot element is either greatest or smallest element.

The **worst-case** time complexity of quicksort is  $O(n^2)$ .





## QuickSort: Performance issue

QuickSort exhibits poor performance for inputs that contain many repeated elements. The problem is visible when all the input elements are equal.

Then at each point in recursion, the left partition is empty (no input values are less than the pivot), and the right partition has only decreased by one element (the pivot is removed).

The algorithm takes quadratic time to sort an array of equal values.

