

# Sorting Algorithms

## Part 2

# Content

- Quick Select
- Quick Sort
- Merge Sort




# Introduction

## Bubble Sort, Selection Sort and Insertion sort:

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

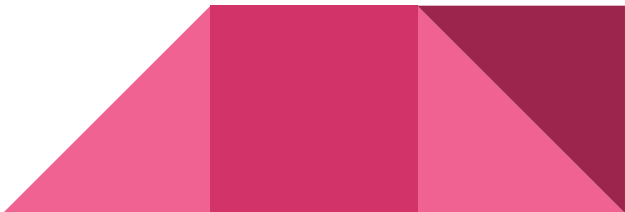
## Quicksort and Merge Sort:

- 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 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)

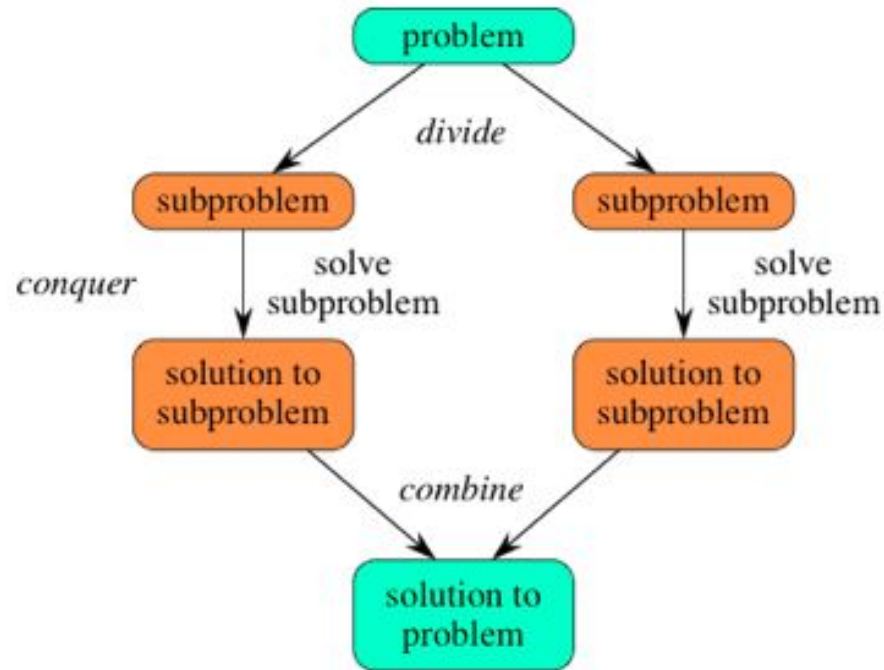


## Divide and conquer algorithm parts

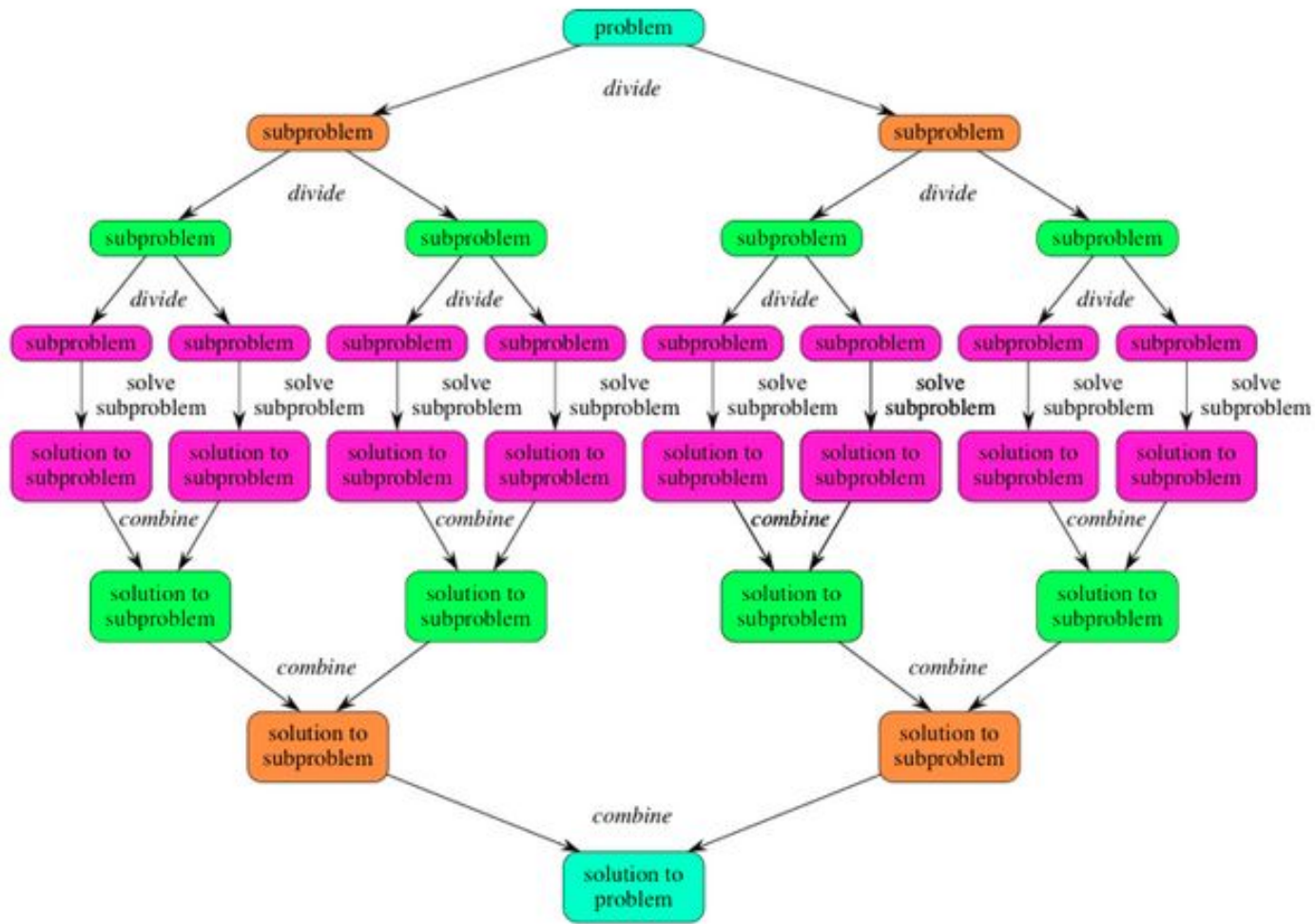
1. **Divide** the problem into a number of subproblems that are smaller instances of the same problem.
2. **Conquer** the subproblems by solving them recursively.
3. **Combine** the solutions to the subproblems into the solution for the original problem.



# Divide and Conquer



conquer



# How would you solve the following problem?

Given an array of  $n$  integer elements and a positive integer  $K$ , find the  $K$ th smallest element in the array. Assume, all array elements are distinct.

`data = [10, 3, 6, 9, 2, 4, 15, 23], K = 4`

Output: 6





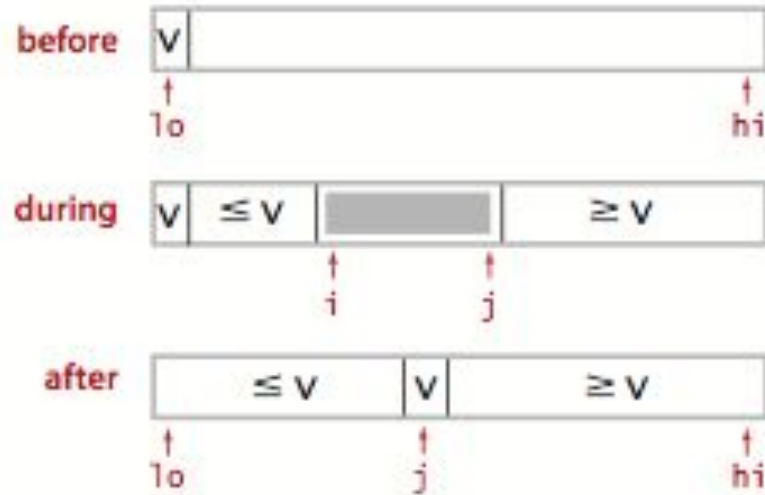
# A possible solution

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.



# Partition method



# 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){  
  
}
```

