

# 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:

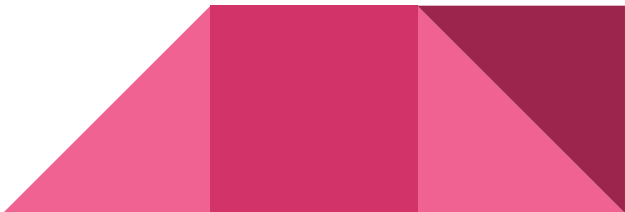
- Better running times
  - Merge sort runs in  $O(n \log n)$  time in all cases
  - Quicksort runs in  $O(n \log n)$  time in the best and average cases, but its worst-case running time is  $O(n^2)$



# 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)

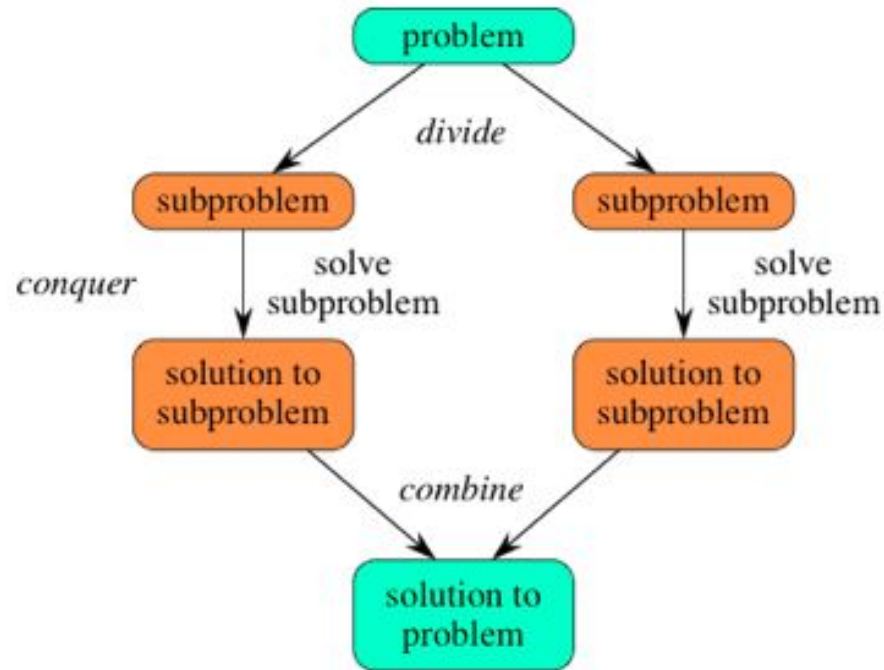


## Divide and conquer algorithm parts

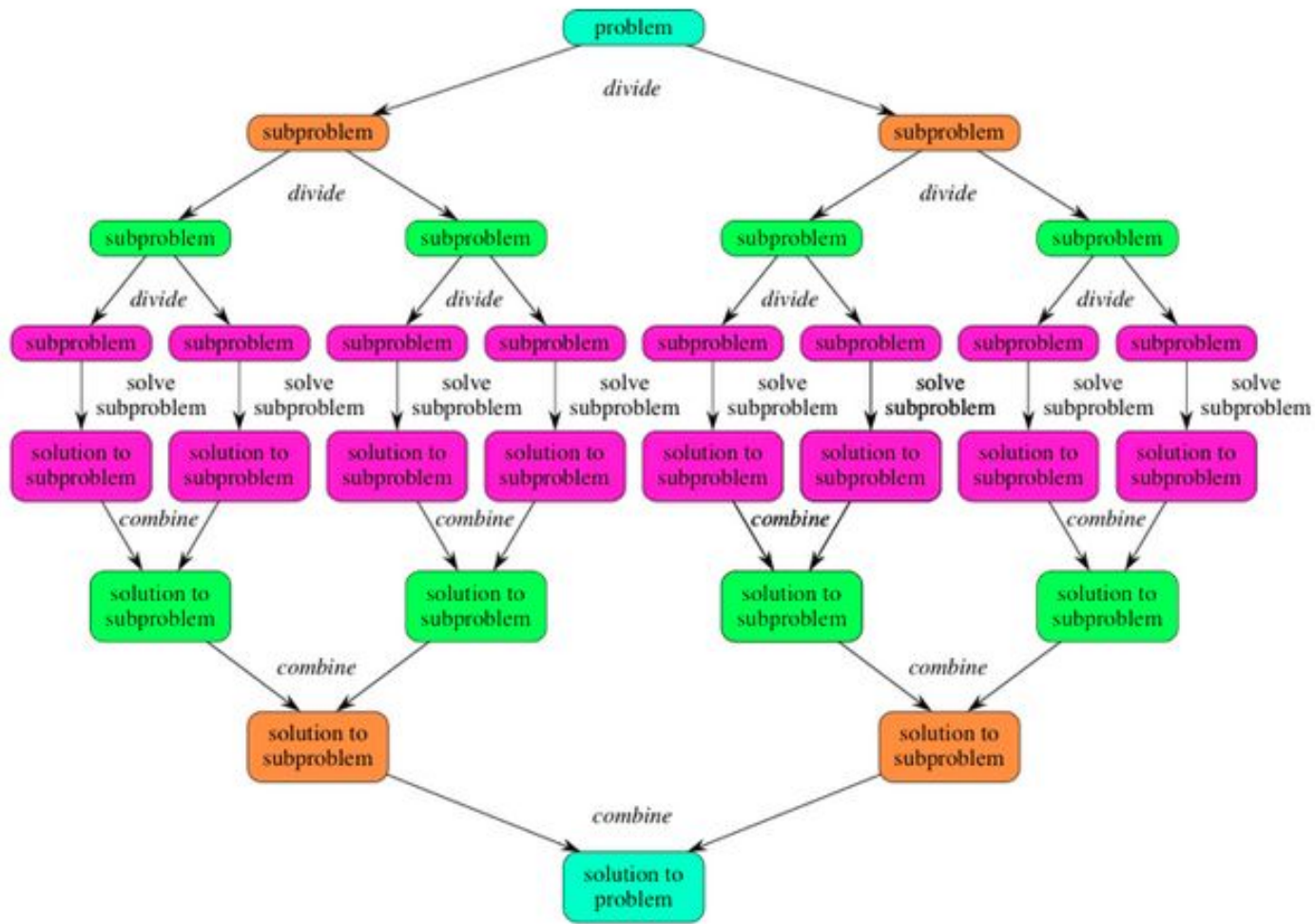
1. **Divide** the problem into a number of subproblems that are smaller instances of the same problem.
2. **Conquer** the subproblems by solving them recursively.
3. **Combine** the solutions to the subproblems into the solution for the original problem.



# 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





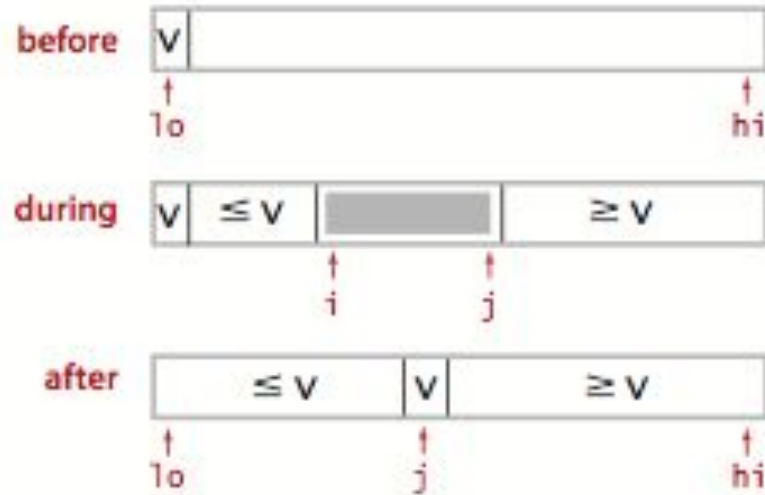
# A possible solution

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){  
  
    }
```



# Quick Select

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).



# Quick Select - Algorithm

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

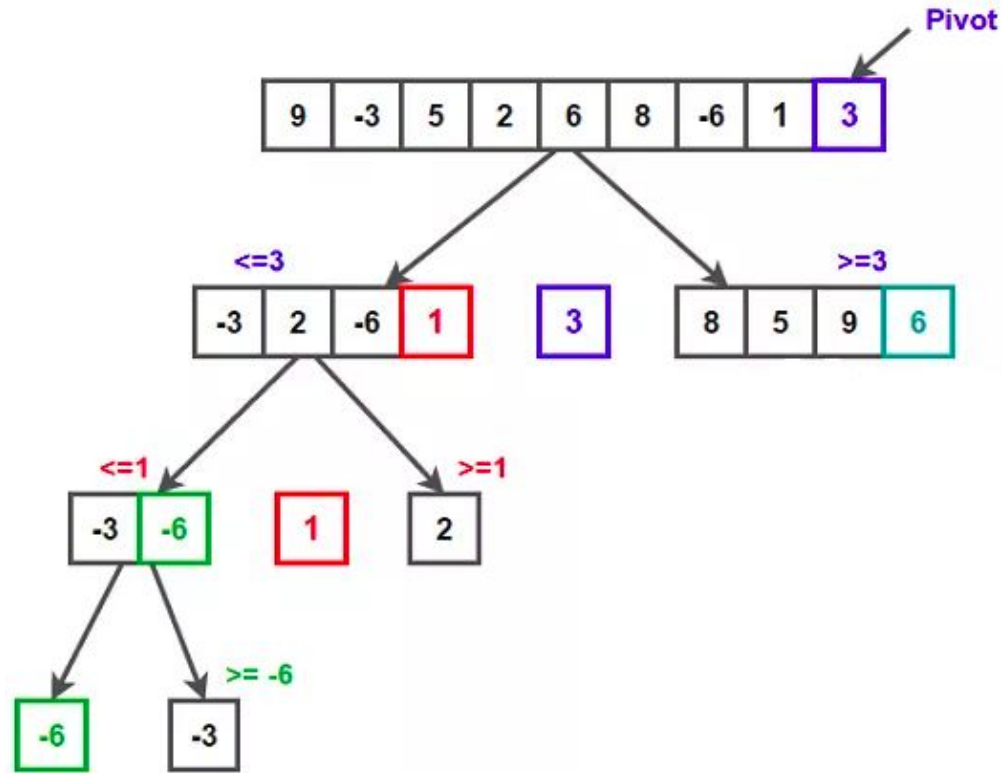
        Recursive call to your QuickSelect - parameters: pivot + 1, end

    Else

        Kth element found return the value [k-1]



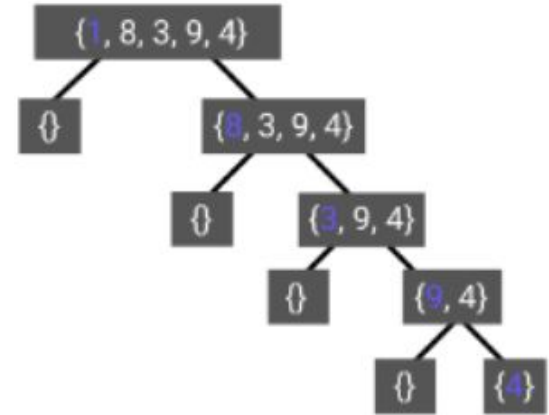
# Quick Select



# 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

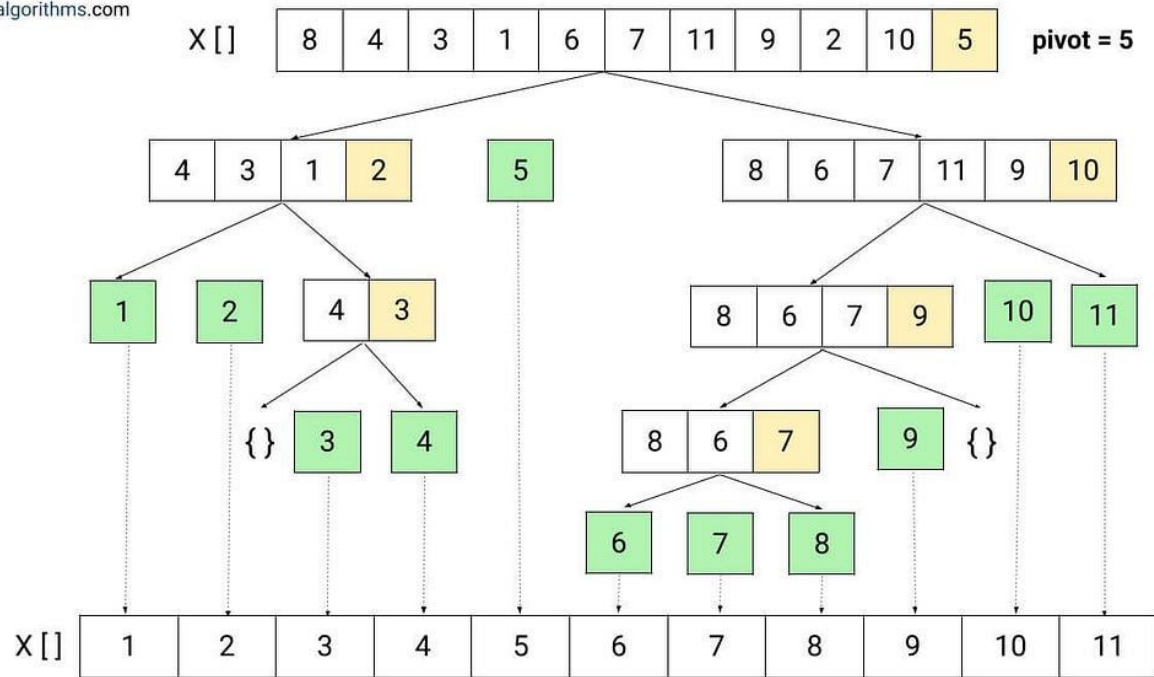
A pivot element for the QuickSort can be picked in different ways:

- 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)
```

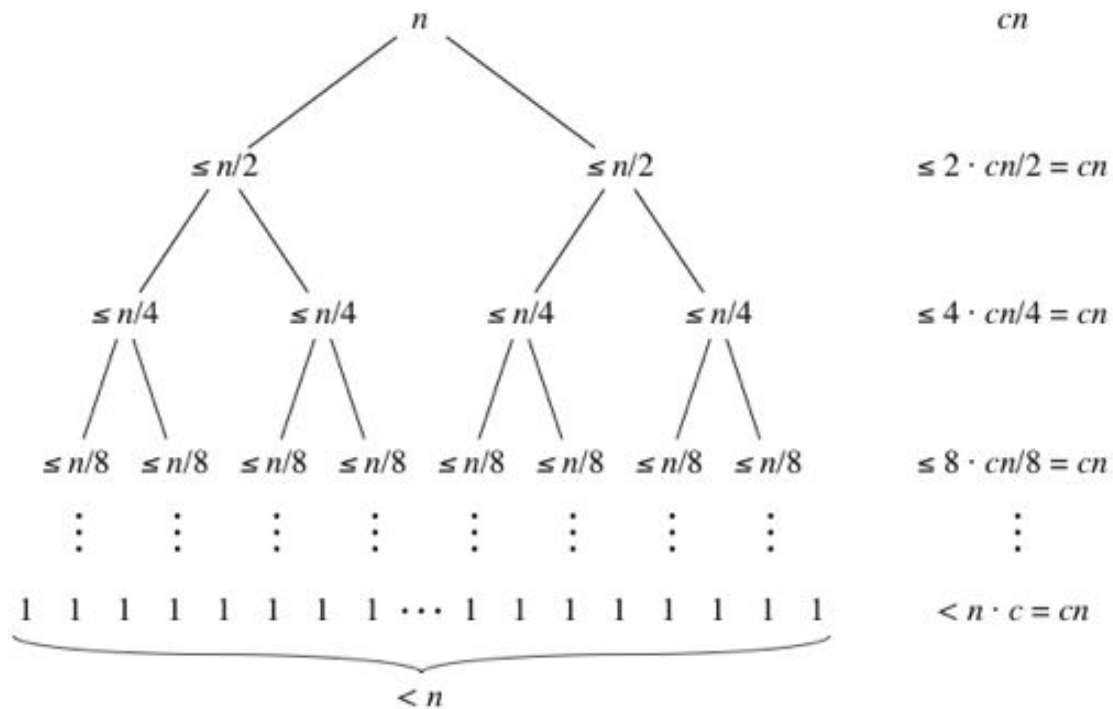
```
        QuickSort(data, start, pivot - 1)
```

```
        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.



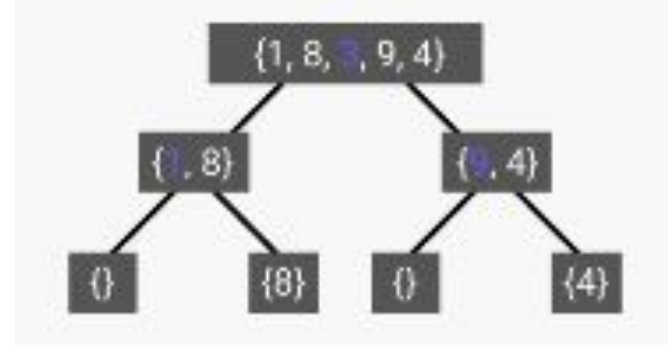
# QuickSort - Time Complexity

**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  $\log N * N$

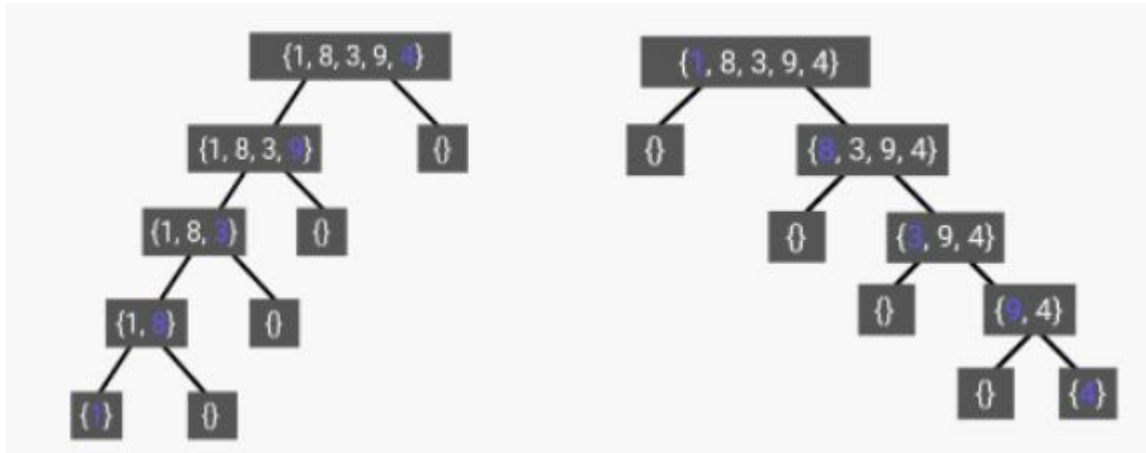
As we have selected mean element as pivot then the array will be divided in branches of equal size so that the height of the tree will be minimum



# 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.



# Quicksort using Dutch National Flag Algorithm


Solution: **Quicksort using Dutch National Flag**

It Implements QuickSort efficiently for inputs containing many repeated elements.

We can use an alternative linear-time partition routine to solve this problem that separates the values into three groups:

- The values less than the pivot,
- The values equal to the pivot, and
- The values greater than the pivot.

The values equal to the pivot are already sorted, so only the less-than and greater-than partitions need to be recursively sorted.



# Merge Sort

## Do now

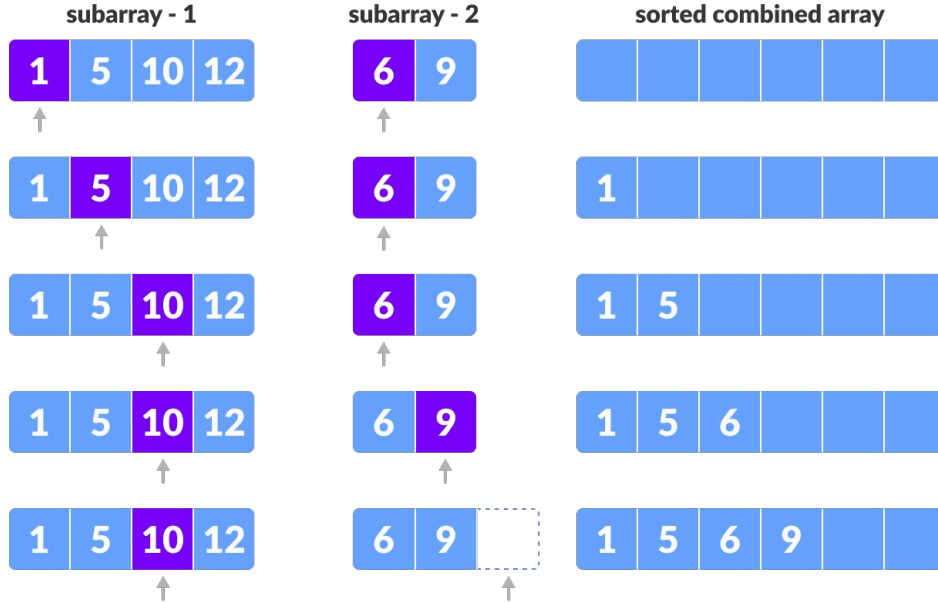
If we have 2 sorted arrays, how would you combine them to have only 1 sorted array?

Input: [4, 8, 9, 20, 25] [2, 5, 7, 12, 30]

Output: [2, 4, 5, 7, 8, 9, 12, 20, 25, 30]



# Merging arrays



Since there are no more elements remaining in the second array, and we know that both the arrays were sorted when we started, we can copy the remaining elements from the first array directly.



How can we write a merge method in Java?  
Parameters?  
Return?

# How can we write a merge method?

```
public static int[] merge(int [] left, int[] right)
```

```
public static void merge(int[] destination, int []  
left, int[] right)
```



# Merge sort process

**Recursive algorithm:** It splits the array in half until it cannot be further divided.

**Base case:** When the array becomes empty or has only one element.

If the array has multiple elements, split the array into halves and recursively invoke the merge sort on each of the halves.

When both halves are sorted, the merge operation is applied, which means we have to combine two smaller sorted array to eventually make a larger one.



# Merge Sort

Steps:

1. It divides an array into smaller subarrays.
2. It sorts each subarray.
3. It merges the sorted subarrays back together to form the final sorted array.



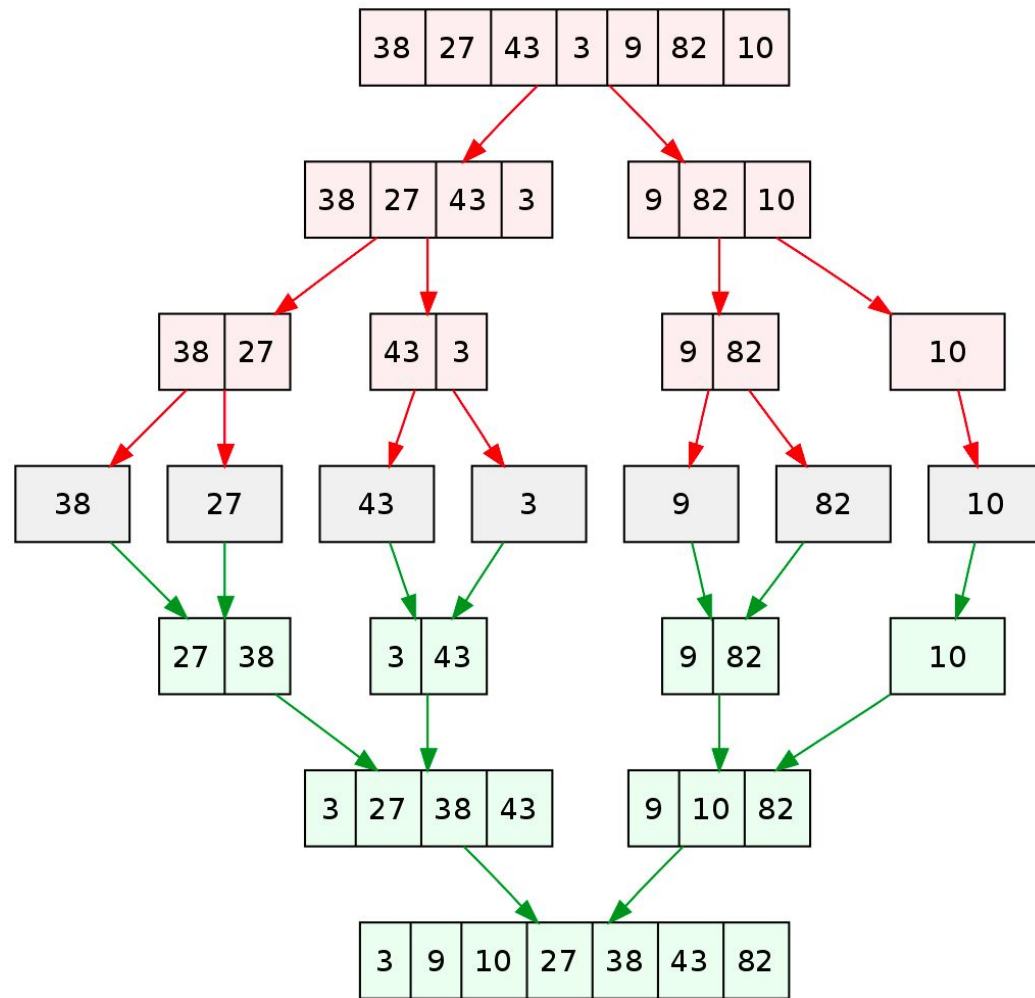


# Merge Sort

The list of size  $n$  is divided into a max of  $\log n$  parts, and the merging of all sublists into a single list takes  $O(n)$  time. The worst, average and best cases have a run time of  **$O(n \log n)$** .

This algorithm is efficient because it quickly sort large arrays.

The red section is where the recursive calls split into sub-arrays. The green section would be when the values return and merge



# Merge Sort Pseudocode

```
int[] mergesort(data){  
    if more than 1 element{  
        L = mergesort left side  
        R = mergesort right side  
        return merge(L,R)  
    }else{  
        return data  
    }  
}
```



# Let's create the method merge in Java

1. Create a folder MergeSort in your classwork folder (assignments repo)
2. Inside MergeSort create a file MergeSort.java
3. Implement the method merge:

```
public static int[] merge(int [] left, int[] right){  
  
}
```

You may write a pseudocode for the method before implementation.

You may test this method from a Driver.java file.

