SORTING ALGORITHMS

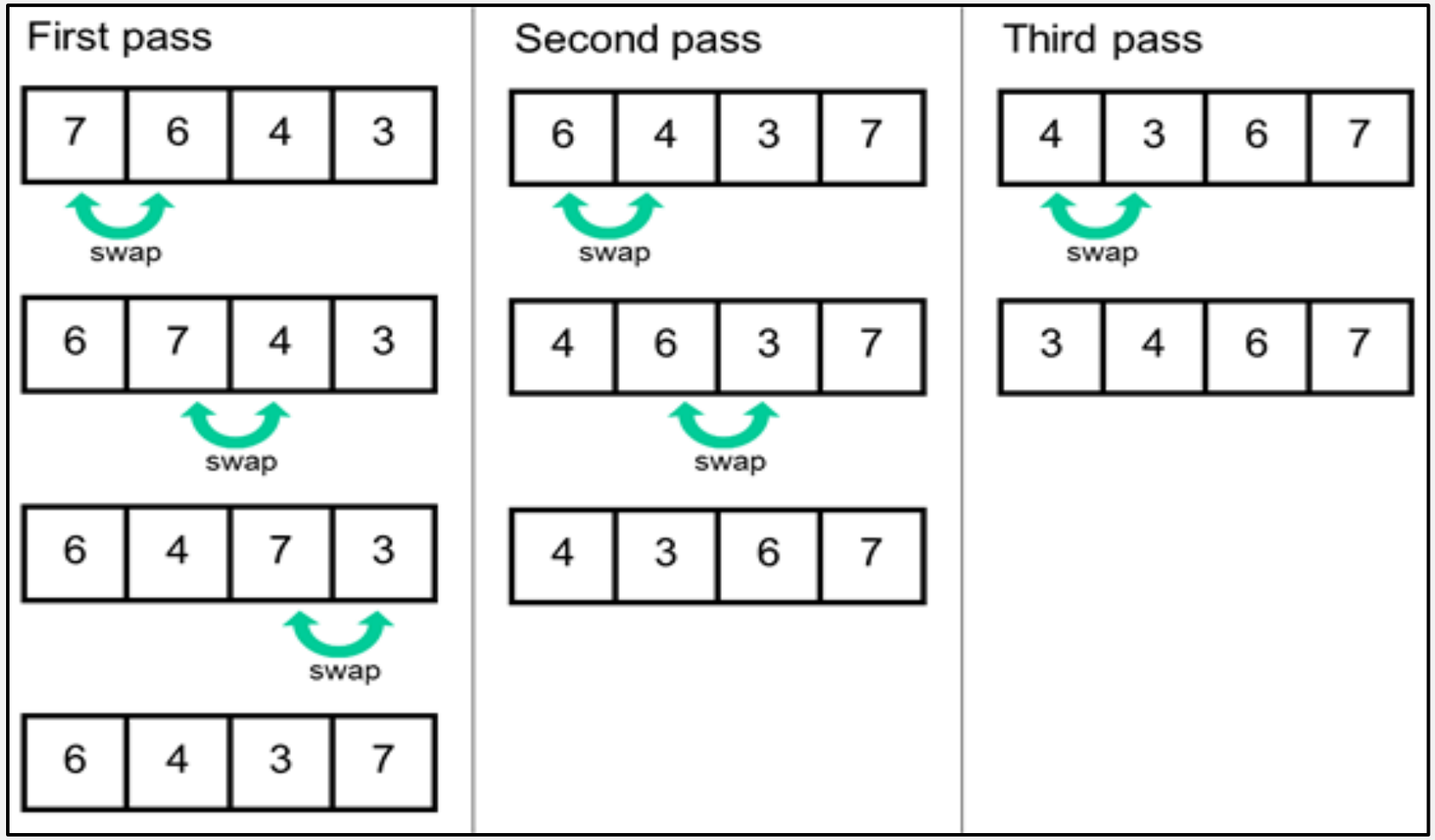
- There are many sorting algorithms:
  - ✓ Bubble Sort (Sinking Sort)
  - ✓ Insertion Sort
  - ✓ Selection Sort
  - ✓ Quick Sort
  - ✓ Merge Sort

and others

BUBBLE SORT ALGORITHM

- ✓ Bubble sort is a simple sorting algorithm which compares the adjacent elements in an array and swaps them if they are in the wrong order.
- ✓ Best case time complexity ➔ O(n)
- ✓ Worst and average case time complexity ➔ O(n^2)

BUBBLE SORT EXAMPLE



BUBBLE SORT CODE

```
function bubbleSort(array){
  for(var i = array.length; i > 0; i--){
    for(var j = 0; j < i - 1; j++){
      if(array[j] > array[j+1]){
        var temp = array[j]
        array[j] = array[j+1]
        array[j+1] = temp
      }
    }
  }
  return array;
}

bubbleSort([4,2,7,1,9])
```

SELECTION SORT ALGORITHM

- ✓ Selection sort is an algorithm that selects the **smallest** element from an unsorted array in each iteration and places that element at the beginning of the sorted array.
- ✓ Selection sort is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty, and the unsorted part is the entire list.
- ✓ The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array. This process continues moving unsorted array boundary by one element to the right.

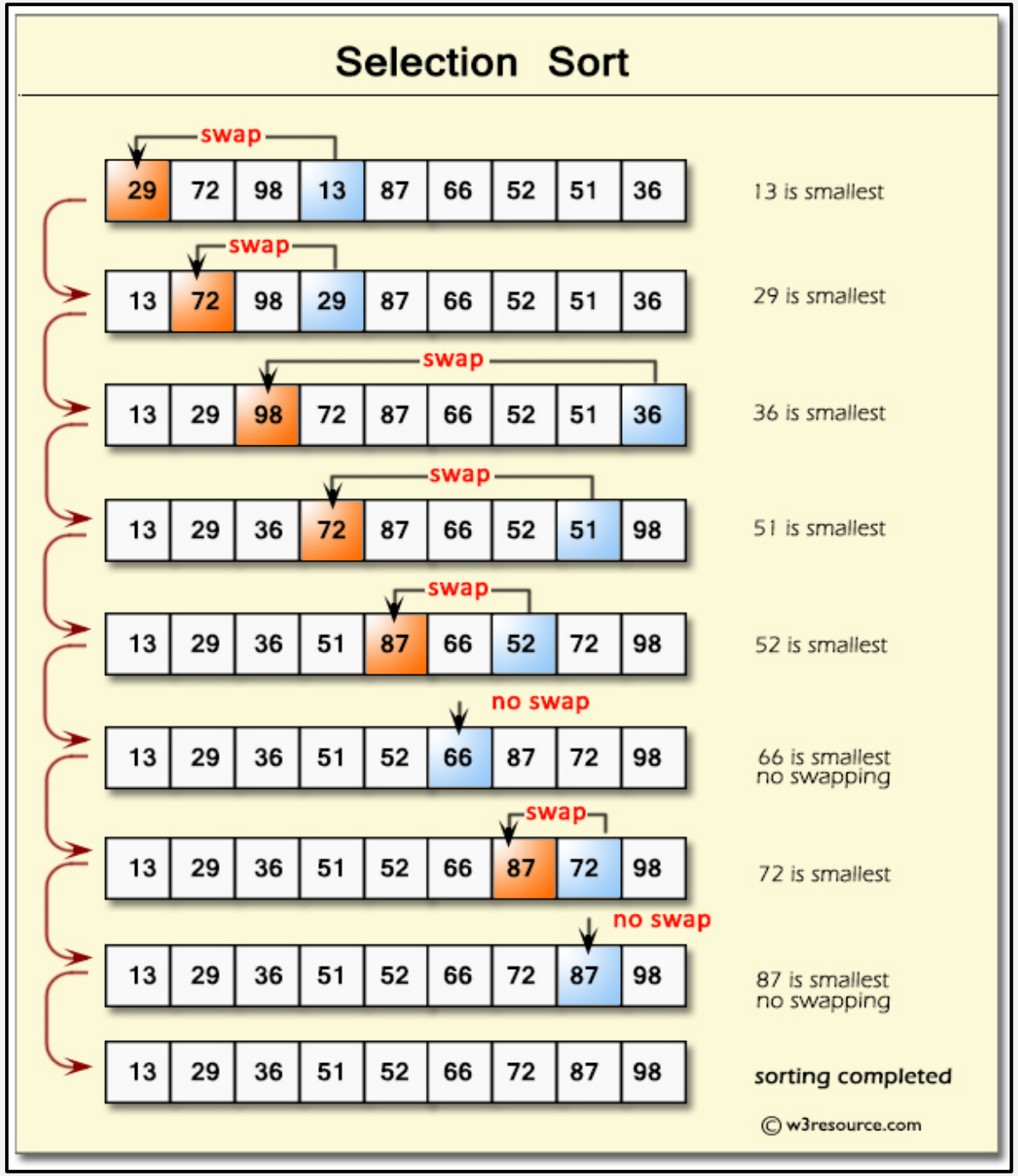
- ✓ **Note:** selection algorithm can select the largest element and place it at the end of the array.

SELECTION SORT STEPS

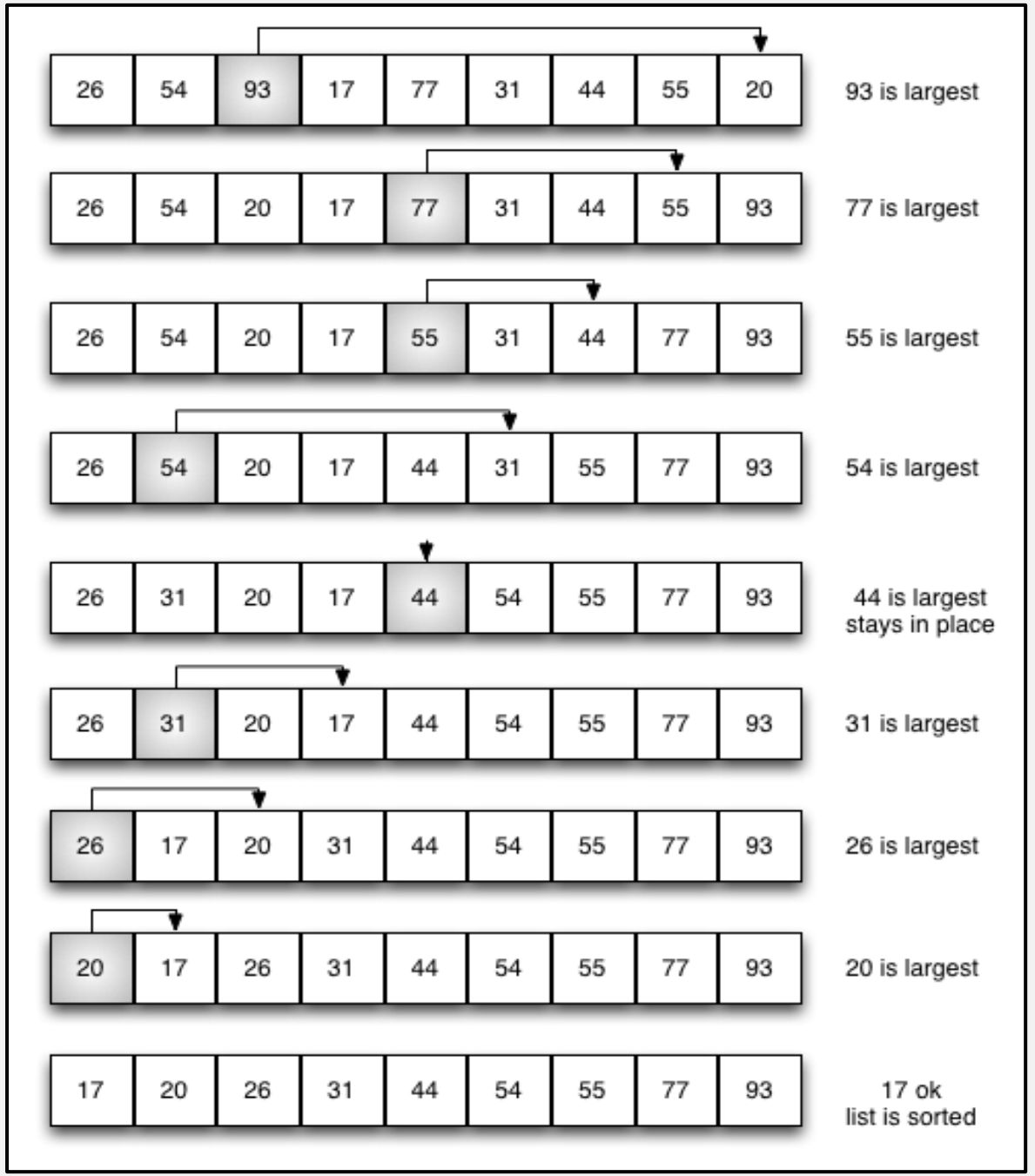
- ✓ **Step1** → Set MIN to location 0
- ✓ **Step2** → Search the minimum element in the array
- ✓ **Step3** → Swap with value at location MIN
- ✓ **Step4** → Increment MIN to point to next element
- ✓ **Step5** → Repeat until the array is sorted

- ✓ Best, average and worst case time **complexity** ➔ O(n^2)

SELECTION SORT EXAMPLE (SMALLEST ELEMENT)



SELECTION SORT EXAMPLE (LARGEST ELEMENT)



SELECTION SORT CODE (SMALLEST ELEMENT)

```python
def selectionSort(List):
    for i in range(len(List) - 1): #For iterating n - 1 times
        minimum = i
        for j in range( i + 1, len(List)): # Compare i and i + 1 element
            if(List[j] < List[minimum]):
                minimum = j
        if(minimum != i):
            List[i], List[minimum] = List[minimum], List[i]
    return List

if __name__ == '__main__':
    List = [4,6,9,8,1,7,3]
    print('Sorted List:',selectionSort(List))

#clcoding.com
```

```
Sorted List: [1, 3, 4, 6, 7, 8, 9]
```

MERGE SORT ALGORITHM

- Merge Sort follows the rule of **Divide and Conquer** to sort a given set of numbers/elements, recursively, hence consuming less time.
- **Divide and Conquer**

  The concept of Divide and Conquer involves three steps:

  1. **Divide** the problem into multiple small problems.

  2. **Conquer** the subproblems by solving them. The idea is to break down the problem into atomic subproblems, where they are solved.

  3. **Combine** the solutions of the subproblems to find the solution of the actual problem.
- Best, average and worst case time **complexity** ➔ O(nlogn)

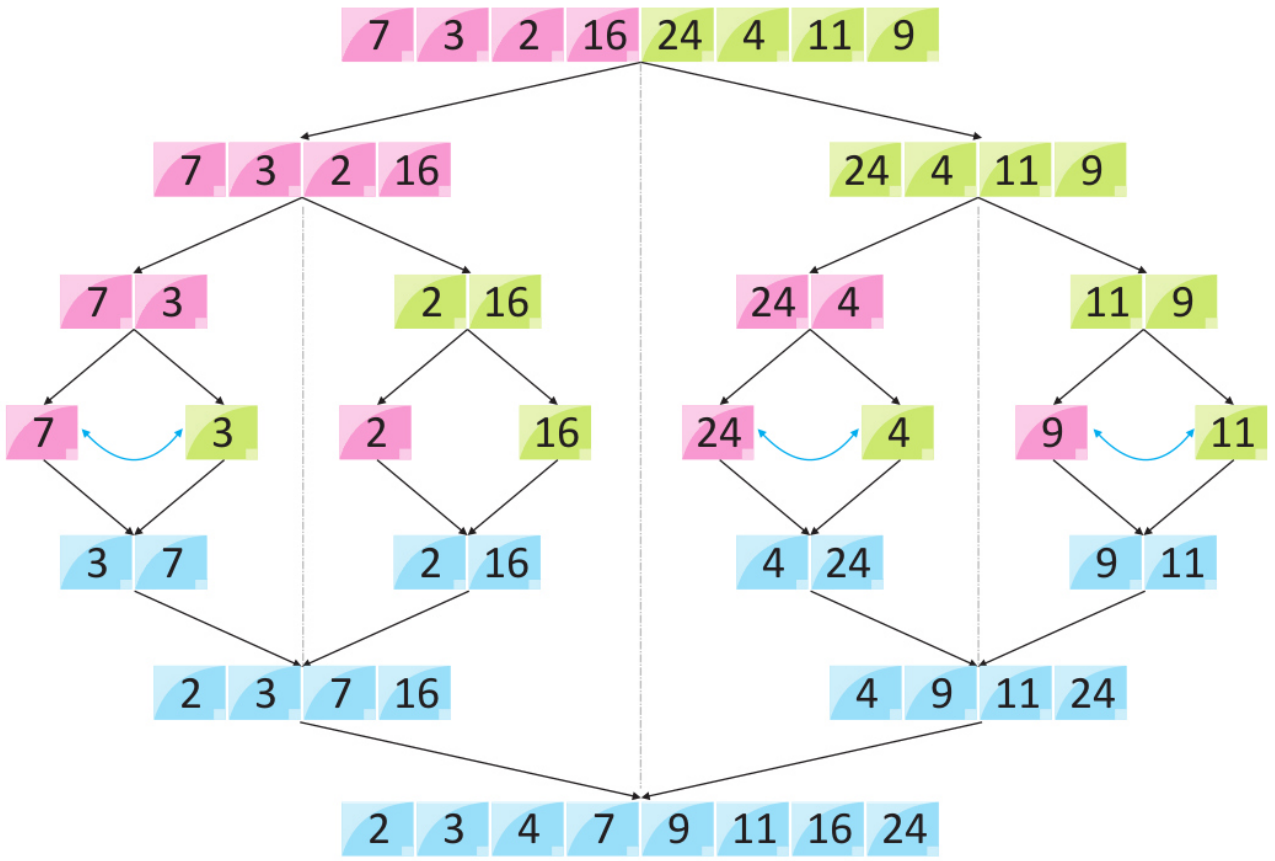
MERGE SORT STEPS

MergeSort(arr[], left, right)

1. Find the middle point to divide the array into two halves: **middle** m = (left +right)/2
2. Call mergeSort for **first half:**
   Call mergeSort(arr, left, m)
3. Call mergeSort for **second half:**
   Call mergeSort(arr, m+1, right)
4. **Merge** the two halves sorted in step 2 and 3:
   Call merge(arr, left, m, right)

Note: Continue the process of breaking into halves until reaching single elements.

MERGE SORT EXAMPLE



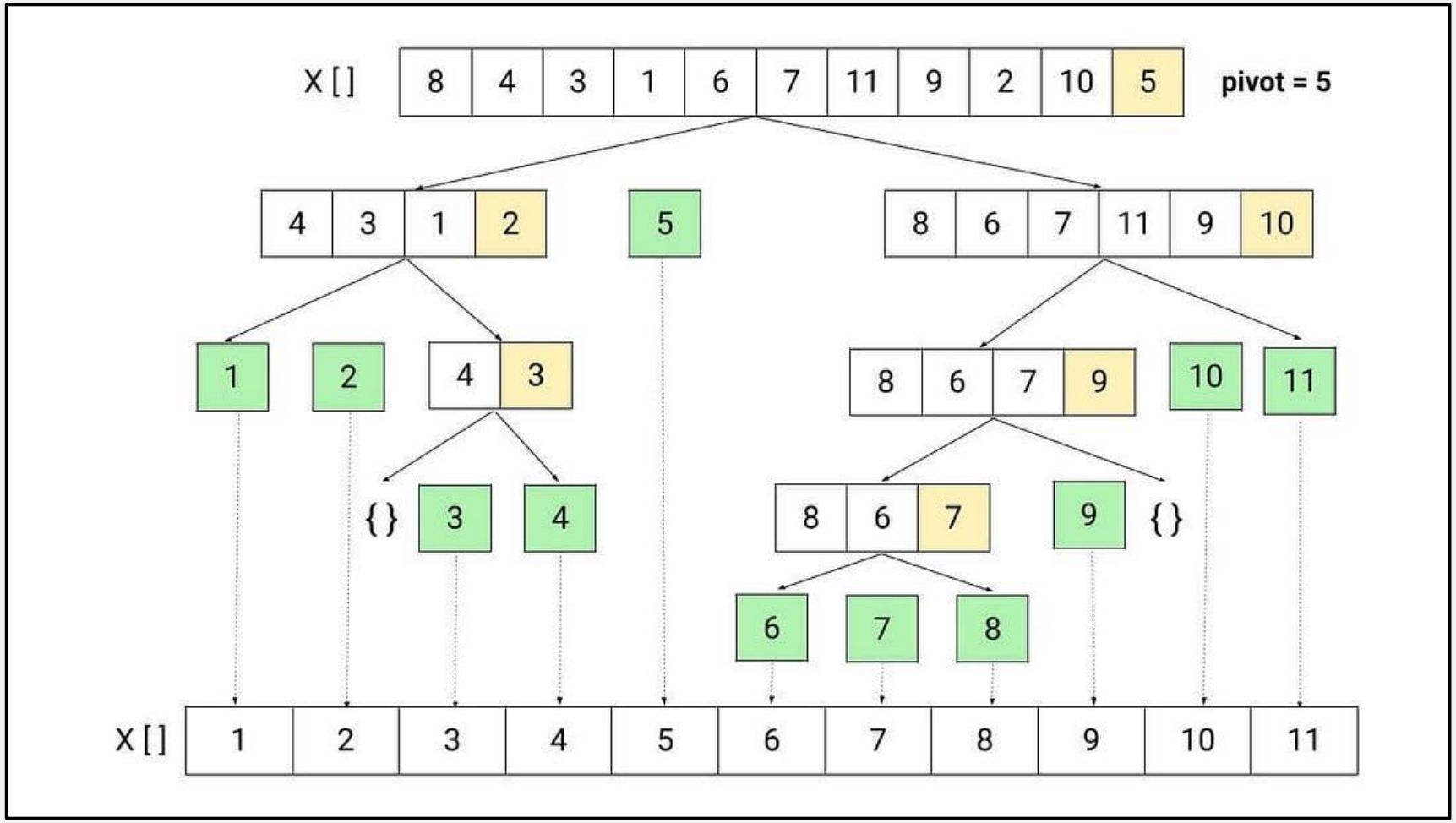
MERGE SORT CODE

```
import sys

def merge(left, right):
    #
    # FILL IN THE CODE HERE
    #
    pass

# YOU DO NOT NEED TO CHANGE THE CODE BELOW THIS LINE #

def merge_sort(lst):
    if len(lst) <= 1:
        return lst
    mid = len(lst) // 2
    left = merge_sort(lst[:mid])
    right = merge_sort(lst[mid:])
    return merge(left,right)
    pass
```

QUICK SORT ALGORITHM

- ✓ Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays. A large array is partitioned into **two** arrays one of which holds values **smaller** than the specified value, say pivot, based on which the partition is made, and another array holds values **greater** than the pivot value.
- ✓ Quick Sort follows the rule of **Divide and Conquer** to sort a given set of numbers/elements, recursively, hence consuming less time.
- ✓ Best and average time complexity ➔ O(nlogn)
- ✓ Worst case time complexity ➔ O(n^2)

QUICK SORT EXAMPLE



QUICK SORT CODE

```python
#Quicksort_Test_Methodl.py

def sort(array):

    less = []
    equal= []
    greater= []
    if len(array) > 1:
        pivot = array[0]
        for i in array:
            if i < pivot:
                less.append(i)
            if i == pivot:
                equal.append(i)
            if i > pivot:
                greater.append(i)
        return (sort(less) + equal + sort(greater))
    else:
        return array
array=[16,0,1,4,3,6,8,10,10,13,15,17,2,0,-1,-2,-3,100]
print("original array: %s" %array)
sorted_array=sort(array)
print("sorted array: %s" %sorted_array)
```

TIME COMPLEXITY OF SEARCHING AND SORTING ALGORITHMS

Algorithm	Best Time Complexity	Average Time Complexity	Worst Time Complexity	Worst Space Complexity
Linear Search	O(1)	O(n)	O(n)	O(1)
Binary Search	O(1)	O(log n)	O(log n)	O(1)
Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)
Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)
Merge Sort	O(nlogn)	O(nlogn)	O(nlogn)	O(n)
Quick Sort	O(nlogn)	O(nlogn)	O(n^2)	O(log n)