SEARCHING AND SORTING ALGORITHMS

Chapter4

Prepared by: Enas Abu Samra

KEY POINTS OF CHAPTER4

- Searching Algorithms:
  - ✓ Linear Search
  - ✓ Binary Search
- Sorting Algorithms:
  - ✓ Bubble Sort (Sinking Sort)
  - ✓ Insertion Sort
  - ✓ Selection Sort
  - ✓ Quick Sort
  - ✓ Merge Sort

SEARCHING ALGORITHMS

- The searching algorithms are used to search or find one or more than one element from a dataset. These type of algorithms are used to find elements from a specific data structures.
- There are three popular algorithms available:
  - ✓ Linear Search
  - ✓ Binary Search
  - ✓ Jump Search

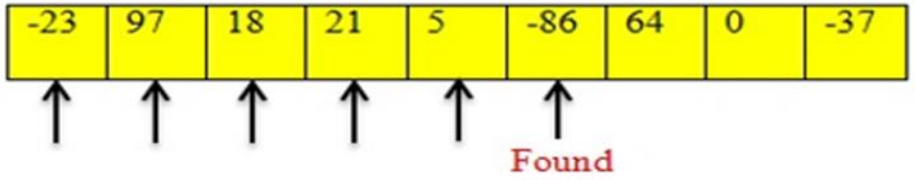
Linear Search

LINEAR SEARCH

- Linear search is a very simple search algorithm. In this type of search, a sequential search is made over all items one by one. Every item is checked and if a match is found then that item is returned, otherwise the search continues till the end of the data collection.
- **Inputs:**
  - ✓ Data Structure ➔ ex. array
  - ✓ Search Value

LINEAR SEARCH EXAMPLE

Linear Search Example: search for element -86



- Go from beginning of a list until element -86 is found

Linear Search Best case

The target value is in the first element of the list. So the search takes some constant amount of time. Computer scientists denote this as $\Omega(1)$ Note that in real life, we don't care about the best case, because it rarely happens.

Linear search worst case

The target value is in the last element of the list. So the search takes an amount of time proportional to the length of the list. Computer scientists denote this as $O(n)$.

Linear search Average case

Target value is somewhere in the list. So, on an average, the target value will be in the middle of the list. So the search takes half the length of the list, which can be denoted by $O(\frac{n}{2})$ ⇨ **Θ(n)**

LINEAR SEARCH CODE

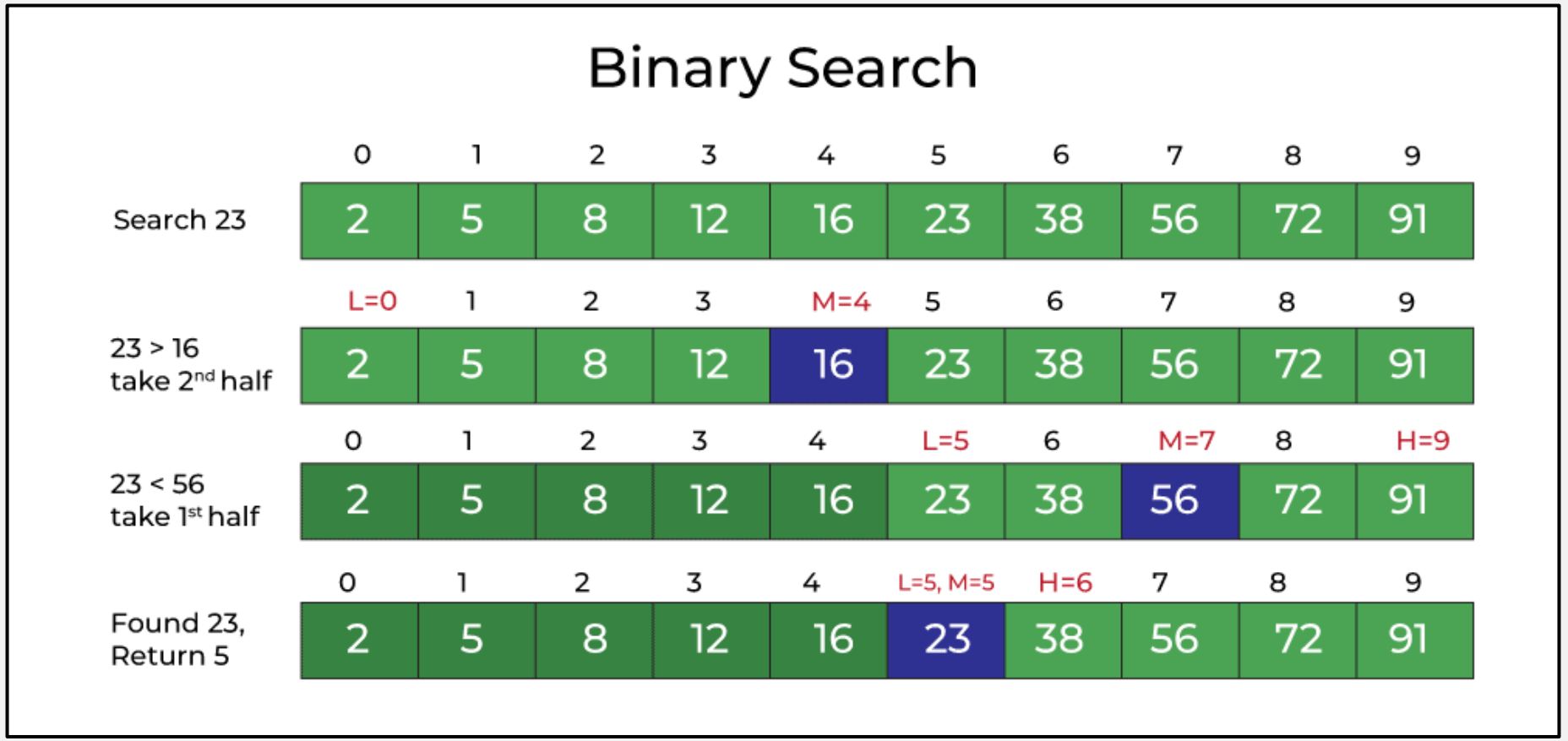
```
lst = []
items = int(input("Enter the number of items: "))
for n in range(items):
    numbers = int(input("Enter the %d number: " %n))
    lst.append(numbers)
keyValue = int(input("Enter number to search for: "))
found = False
for i in range(len(lst)):
    if lst[i] == keyValue:
        found = True
        print("%d found at location %d" % (keyValue, i))
        break
if not found:
    print("%d is not in list" % keyValue)
input()
```

Binary Search

BINARY SEARCH

- Binary search is a search algorithm that finds the position of a target value within **a sorted array.**
- A binary search begins by comparing the middle element of the array with the target value. If the target value matches the middle element, its position in the array is returned. If the target value is less or more than the middle element, the search continues the lower or upper half of the array respectively with a new middle element, eliminating the other half from consideration.
- **Inputs:**
  - ✓ Data Structure ➔ ex. array
  - ✓ Search Value
- Best case time complexity ➔ O(1), worst and average case time complexity ➔ O(logn)

BINARY SEARCH EXAMPLE



BINARY SEARCH CODE

```
def BinarySearch(list,n):

    l = 0
    h = len(list)-1
    found = 0

    while i <= h:
        mid = (l + h) // 2

        if list[mid] == n:
            print("Element {} found at position {}".format(n,mid+1))
            found = 1
            return True
        if list[mid] > n:
            h = mid - 1
        if list[mid] < n:
             l = mid + 1
    if found !=1:
        print("Searching element {} not found in the array list".format(n))

    return

list = []
size = int(input("Enter the size of the array: "))

for i in range(size):
     x =  int(input("Enter the element at {} position in the array: ".format(i+1)))
     list.append(x)

list.sort()
print("Entered array elements are: ")
for lists in list:
    print(lists,end="\t")

se = int(input("\nEnter the array element to be searched: "))
BinarySearch(list, se)
```

BINARY SEARCH CODE USING RECURSION

```
59 // binary search
60 bool BinarySearch(int key, int array[], int min, int max)
61 {
62    if (min <= max)
63    {
64         int middle = (min + max)/2;
65
66         if (key == array[middle])
67             return true;
68         else if (key < array[middle])
69             BinarySearch(key, array, min, middle - 1);
70         else if (key > array[middle])
71             BinarySearch(key, array, middle + 1, max);
72    }
73
74    return false;
75
76 }
```

Sorting Algorithms

SORTING ALGORITHMS

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

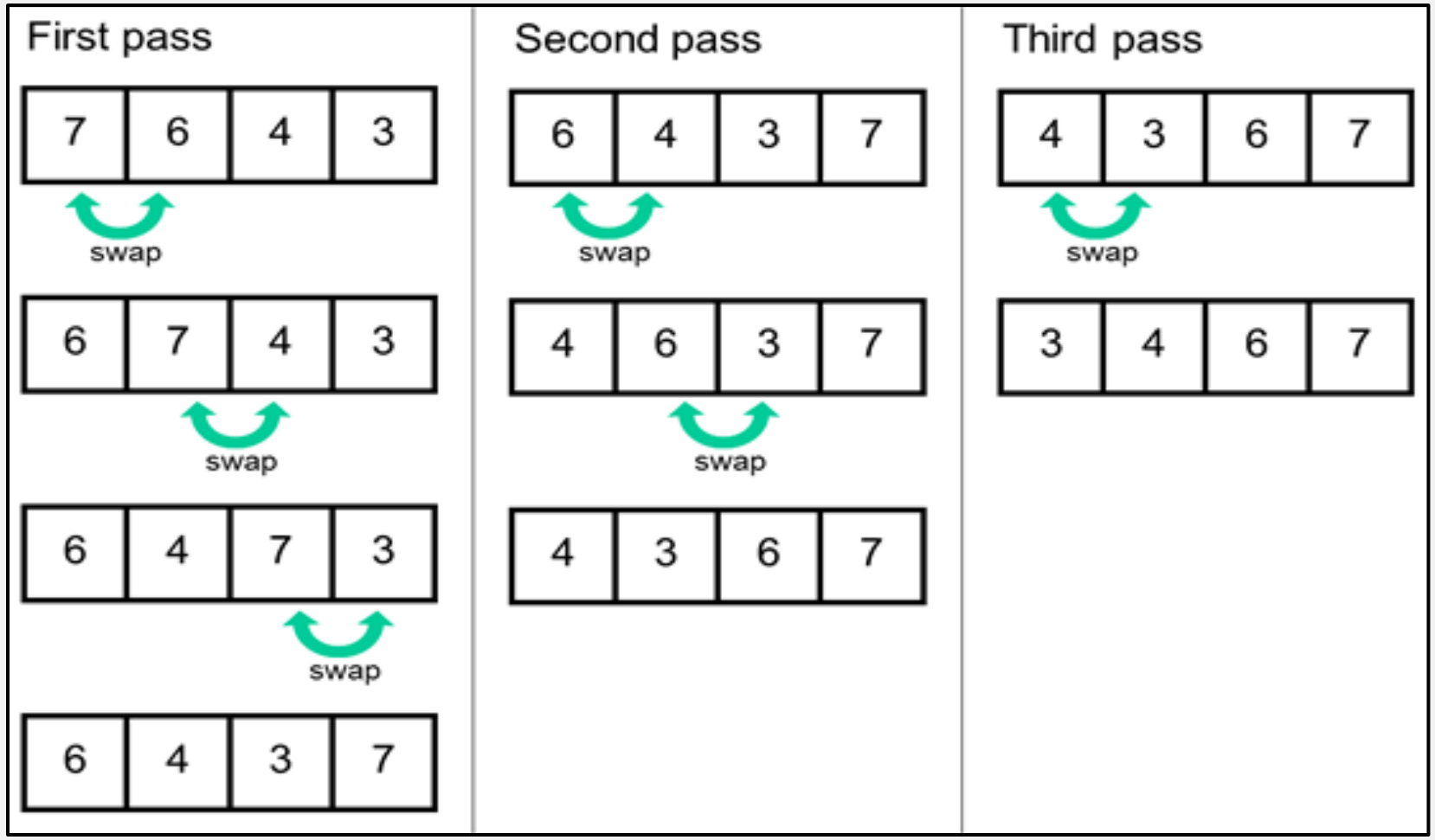
and others

Bubble Sort Algorithm

BUBBLE SORT 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])
```

Insertion Sort Algorithm

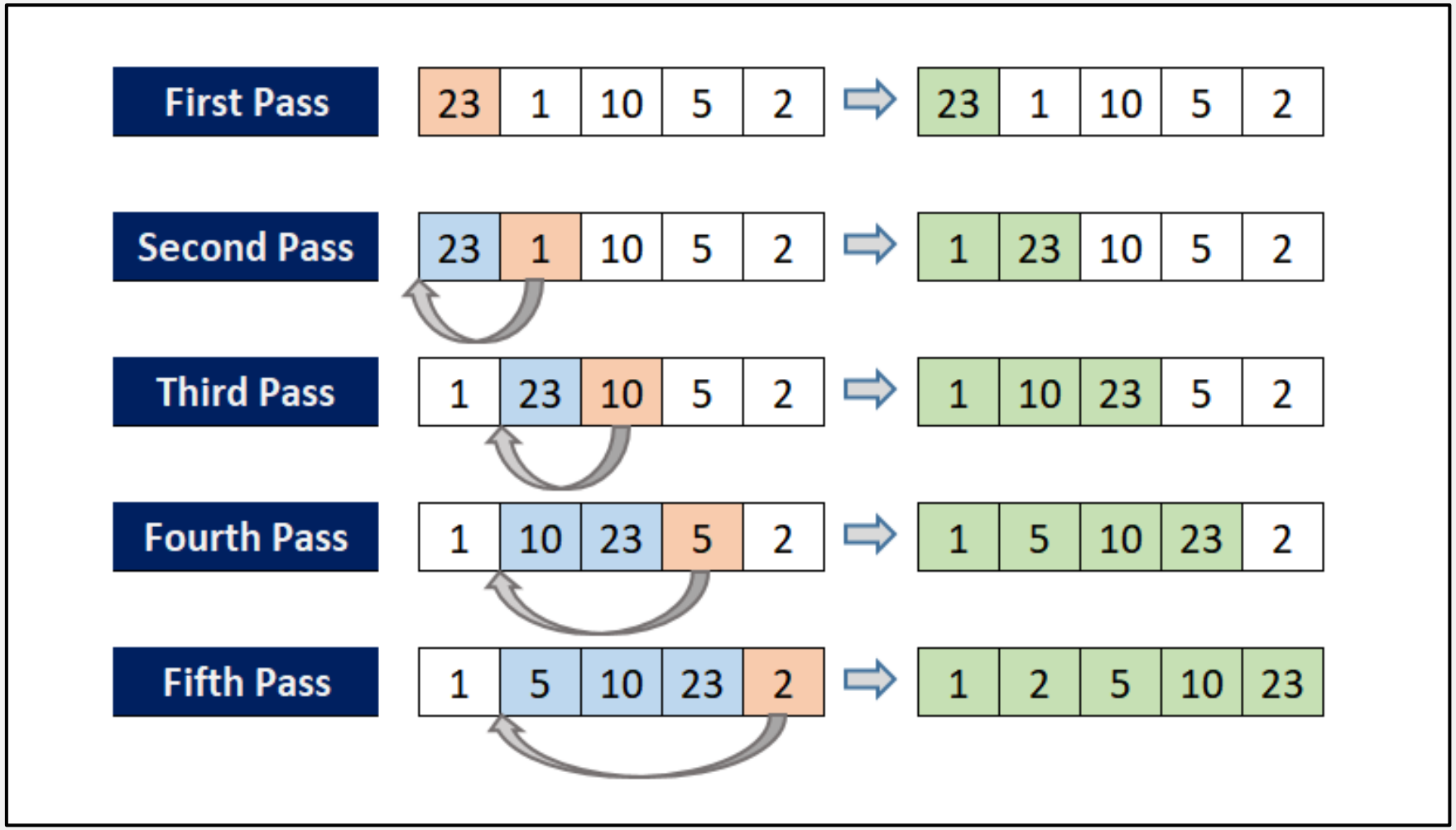
INSERTION SORT ALGORITHM

- ✓ The basic idea of insertion sort is that one element from the input elements is consumed in each iteration to find its correct position i.e., the position to which it belongs in a sorted array.
- ✓ Best case time complexity ➔ O(n)
- ✓ Average and worst case time complexity ➔ O(n^2)

INSERTION SORT STEPS

- ✓ The first step involves the comparison of the current element (in the beginning it will be the first element in the data set) with its adjacent element.
- ✓ If the current element can be inserted at a particular position, then space is created for it by shifting the other elements one position to the right and inserting the element at the suitable position.
- ✓ The above procedure is repeated until all the element in the array are sorted.

INSERTION SORT EXAMPLE



INSERTION SORT CODE

```python
def insertionSort(List):
    for i in range(1, len(List)):
        currentNumber = List[i]
        for j in range(i - 1, -1, -1):
            if List[j] > currentNumber :
                List[j], List[j + 1] = List[j + 1], List[j]
            else:
                List[j + 1] = currentNumber
                break

    return List

if __name__ == '__main__':
    List = [3,7,2,8,4,1,9,5]
    print('Sorted List:',insertionSort(List))

#clcoding.com
```

```
Sorted List: [1, 2, 3, 4, 5, 7, 8, 9]
```

Selection Sort Algorithm

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.

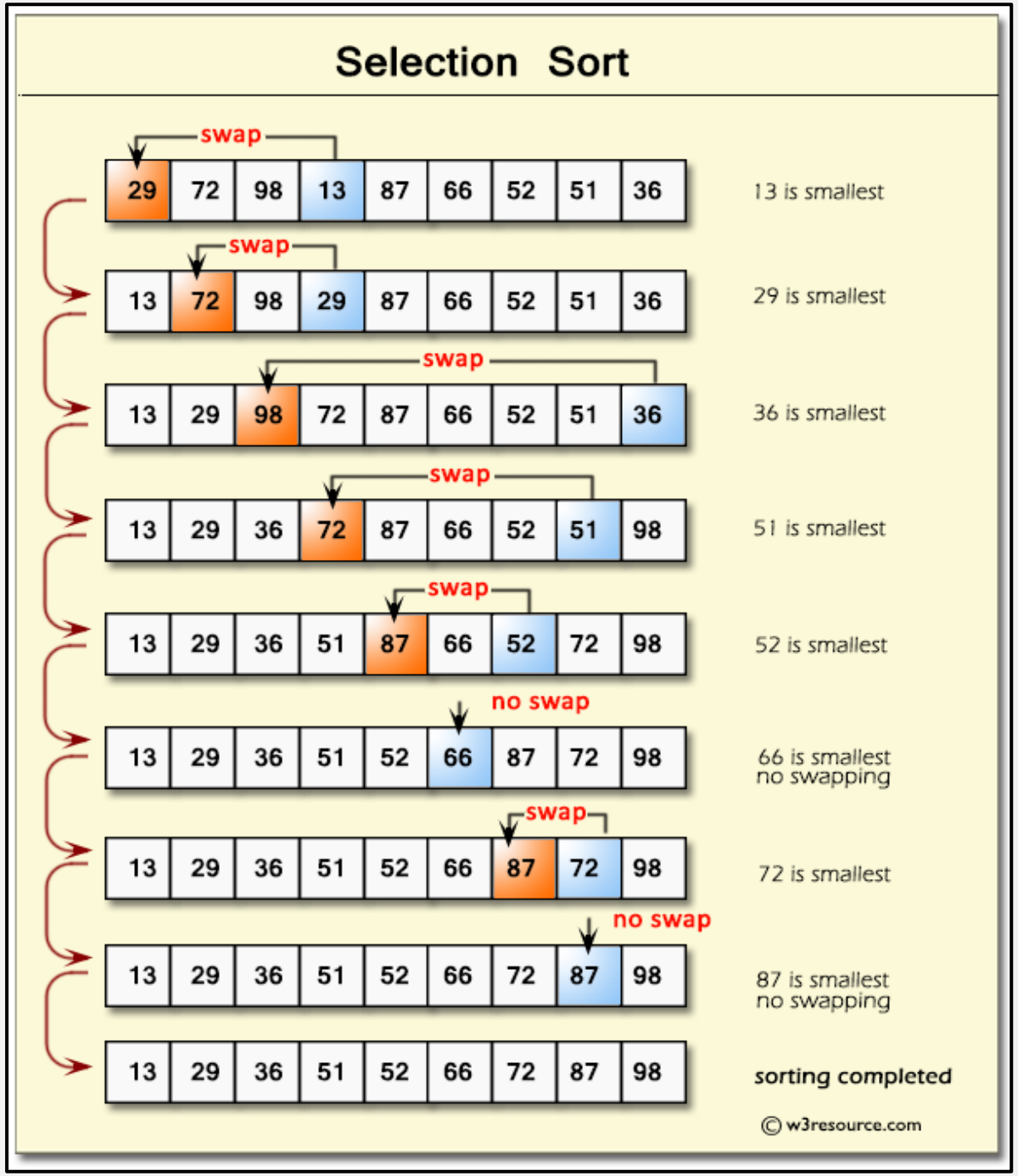
- **Note:** selection algorithm can select the largest element and place it at the end of the array.

SELECTION SORT STEPS

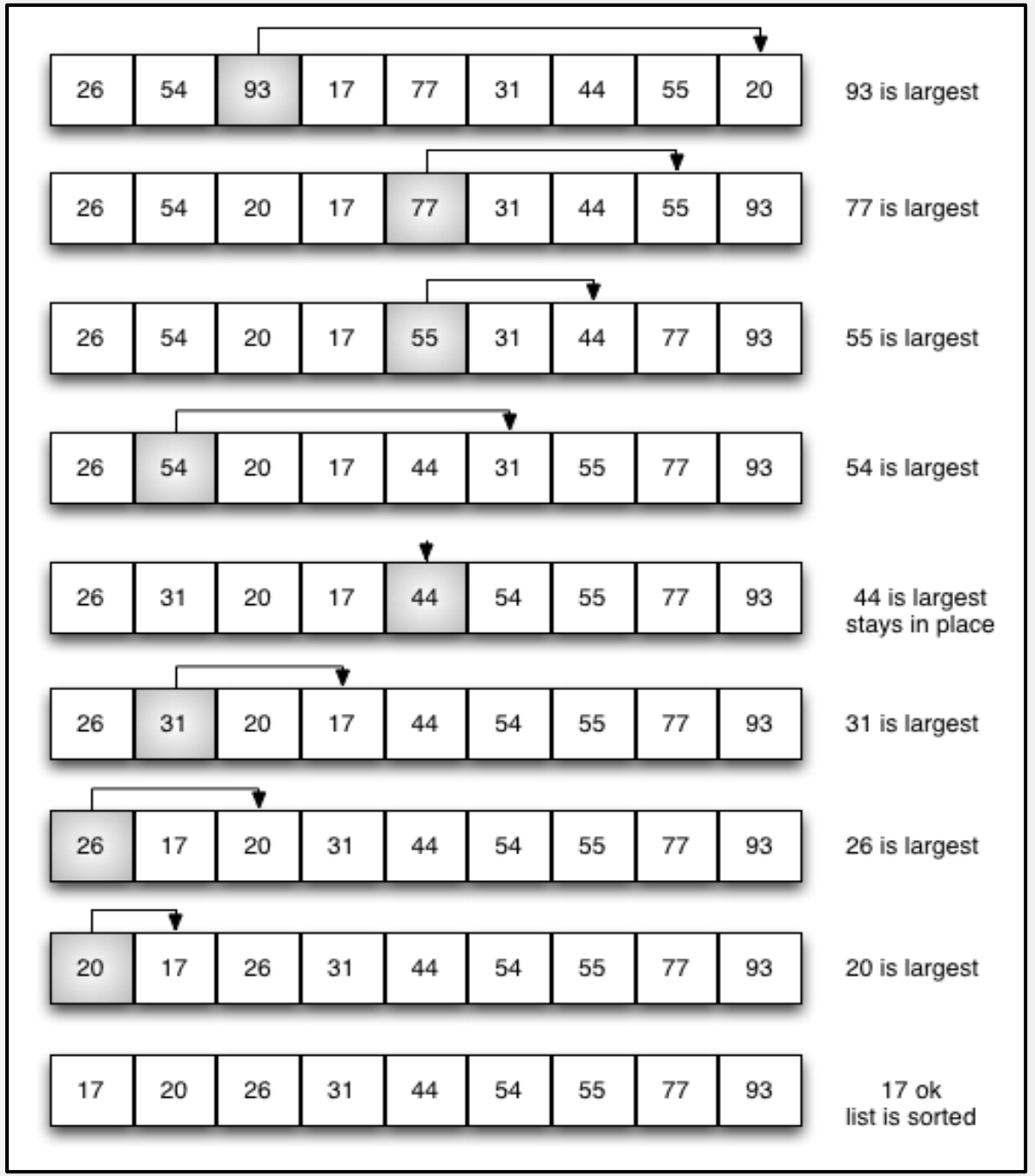
- ✓ **Step1** → Set MIN to location 0
- ✓ **Step2** → Search the minimum element in the array
- ✓ **Step3** → Swap with value at location MIN
- ✓ **Step4** → Increment MIN to point to next element
- ✓ **Step5** → Repeat until the array is sorted

- ✓ 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 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)

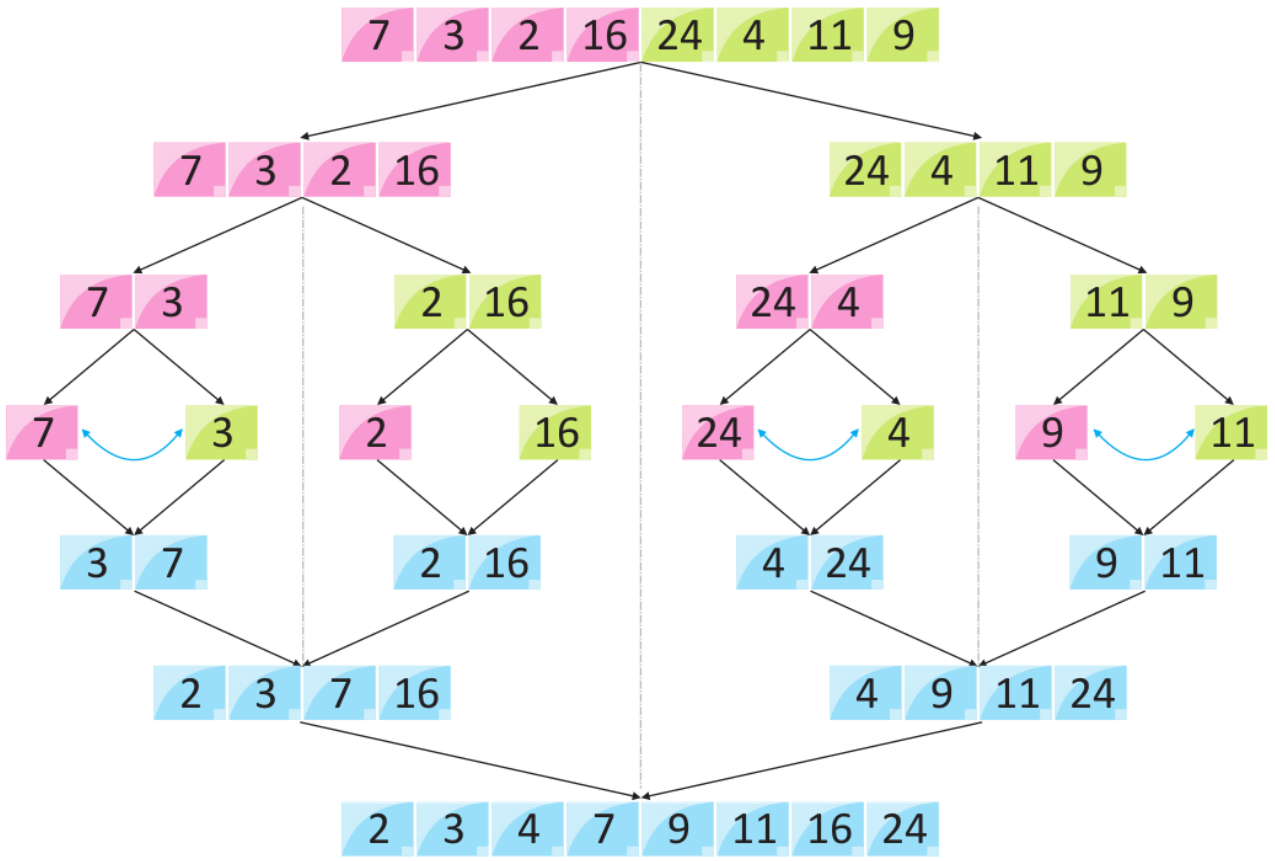
MERGE SORT STEPS

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

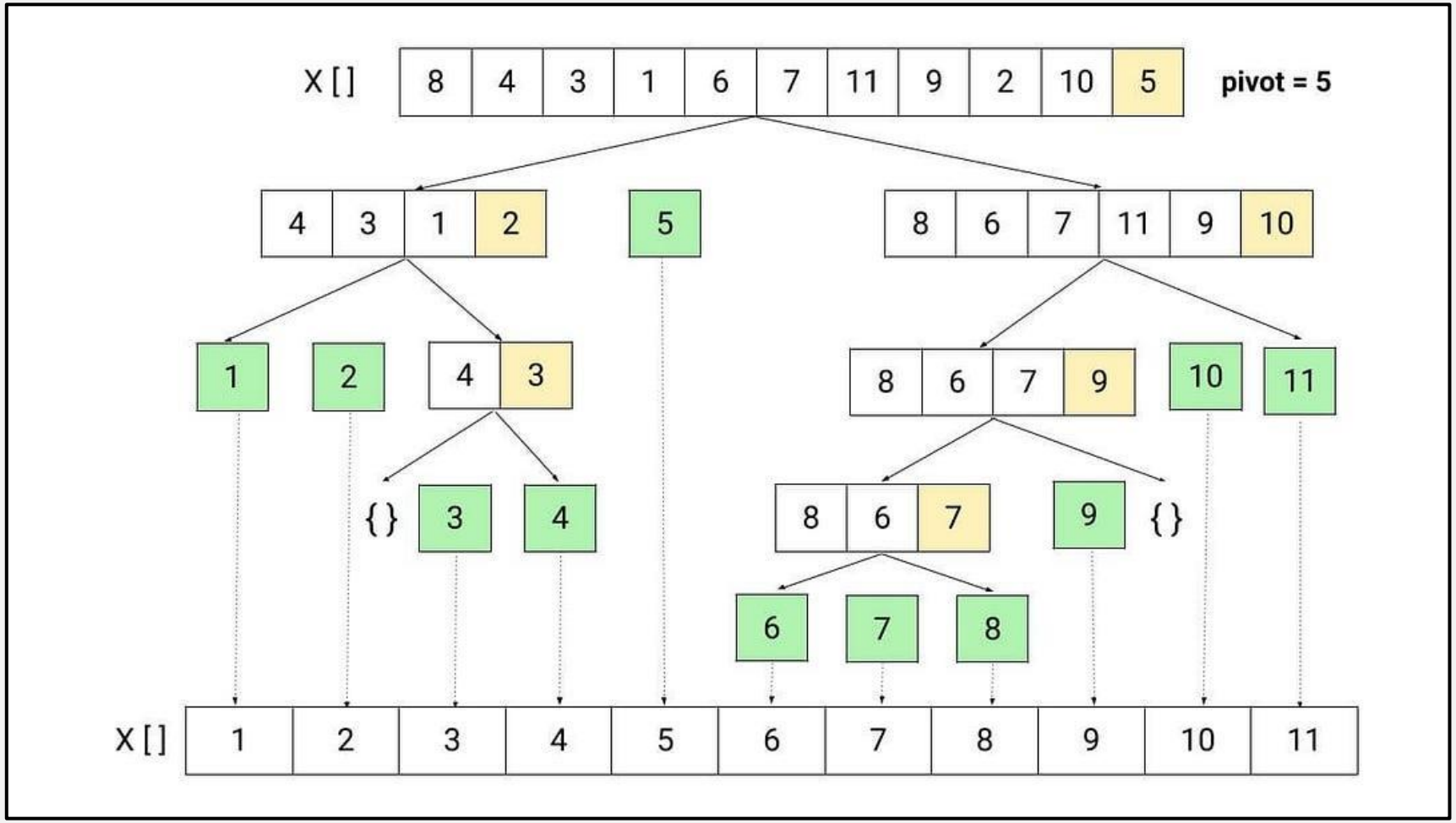
```
import sys

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 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)

END OF CHAPTER4