University American College Skopje

SINCE 2005

DATA STRUCTURES AND ALGORITHMS

- LECTURE 5-

(2ND YEAR OF STUDY)

Skopje, 2023/24

Contents

5. Sorting Algorithms

5.1. Introduction

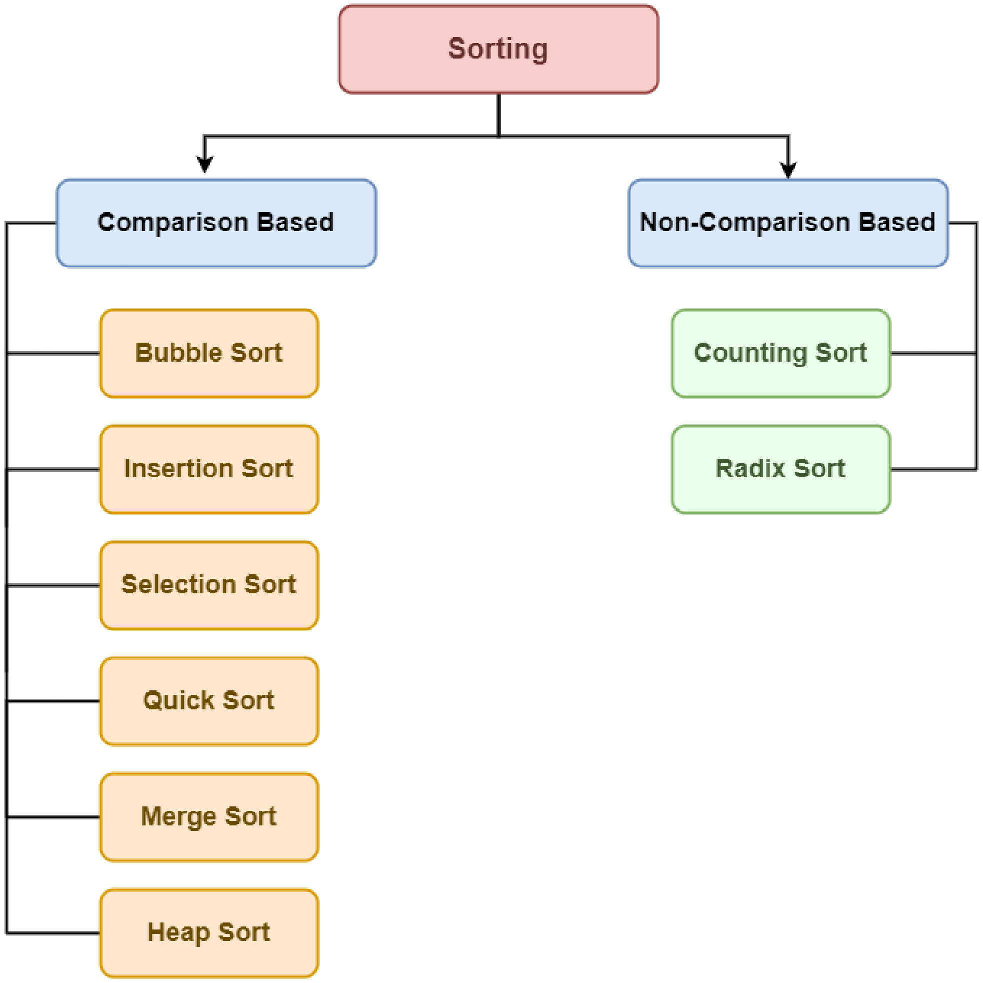
- **Sorting Algorithms** refer to rearrangement of a given array, or list of elements, according to a comparison operator on the elements. The comparison operator is used to decide the new order of elements in the respective data structure.
- When we have a large amount of data, it can be difficult to deal with it, especially when it is arranged randomly. When this happens, sorting that data becomes crucial.

5.1. Introduction

- **Types of Sorting Techniques:** There are various sorting algorithms used in data structures. The following two types of sorting algorithms can be broadly classified:
  - **Comparison-based:** We compare the elements in a comparison-based sorting algorithm.
  - **Non-comparison-based:** We do not compare the elements in a non-comparison-based sorting algorithm.
- The sorting algorithm is important in Computer Science because it reduces the complexity of a problem. There is a wide range of applications for these algorithms.

5.1. Introduction

- Sorting algorithms



5.1. Introduction

- Where sorting algorithms are used:
  - When you have hundreds of datasets you want to print, you might want to arrange them in some way.
  - Sorting algorithm is used to arrange the elements of a list in a certain order (either ascending or descending).
  - Searching any element in a huge data set becomes easy. We can use Binary search method for search if we have sorted data. So, Sorting become important here.
  - They can be used in software and in conceptual problems to solve more advanced problems.

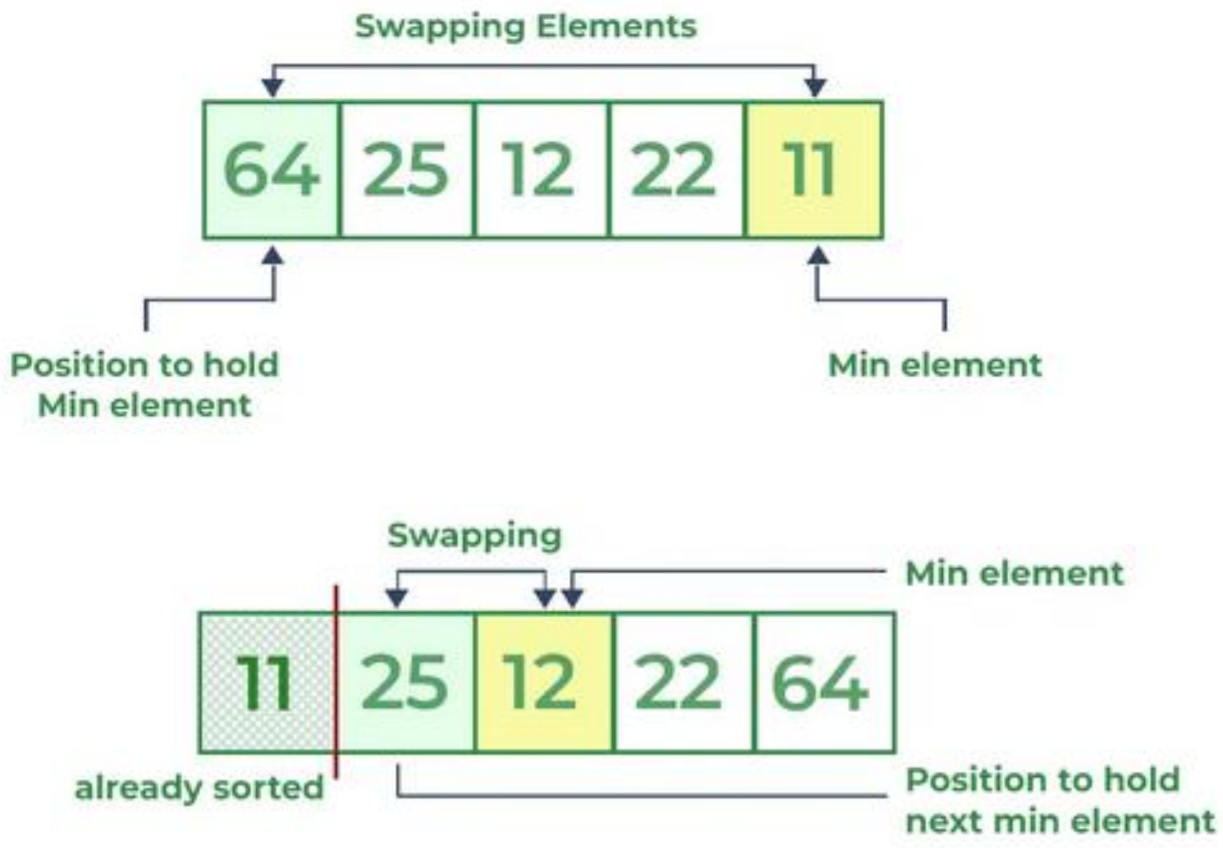
5.2. Selection Sort

- **Selection sort** is a simple and efficient sorting algorithm that repeatedly selects the smallest (or largest) element from the unsorted portion of the list and swaps it with the first element of the unsorted part. This process is repeated for the remaining unsorted portion until the entire list is sorted.
- In this technique we find the minimum element in every iteration and place it in the array beginning from the first index. Thus, a selection sort also gets divided into a sorted and unsorted subarray.

5.2. Selection Sort

- Let's consider the following array as an example: arr[ ] = {64, 25, 12, 22, 11}
- **First pass:**
  - For the first position in the sorted array, the whole array is traversed from index 0 to 4 sequentially. The first position where **64** is stored presently, after traversing whole array it is clear that **11** is the lowest value.
  - Thus, replace 64 with 11. After one iteration **11**, which happens to be the least value in the array, tends to appear in the first position of the sorted list.

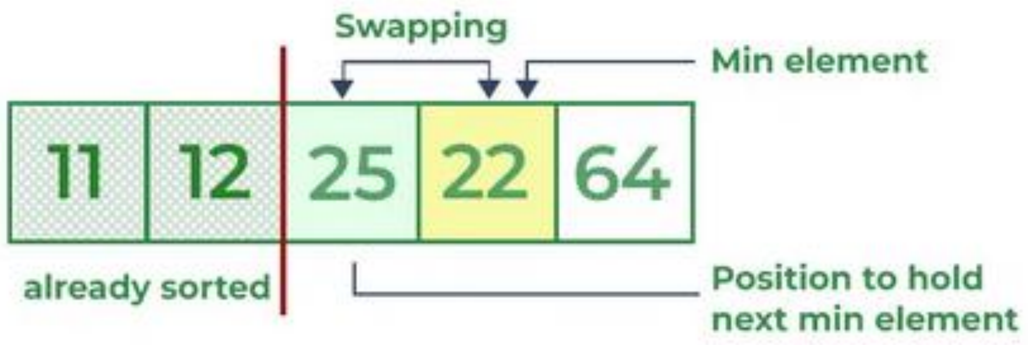
5.2. Selection Sort

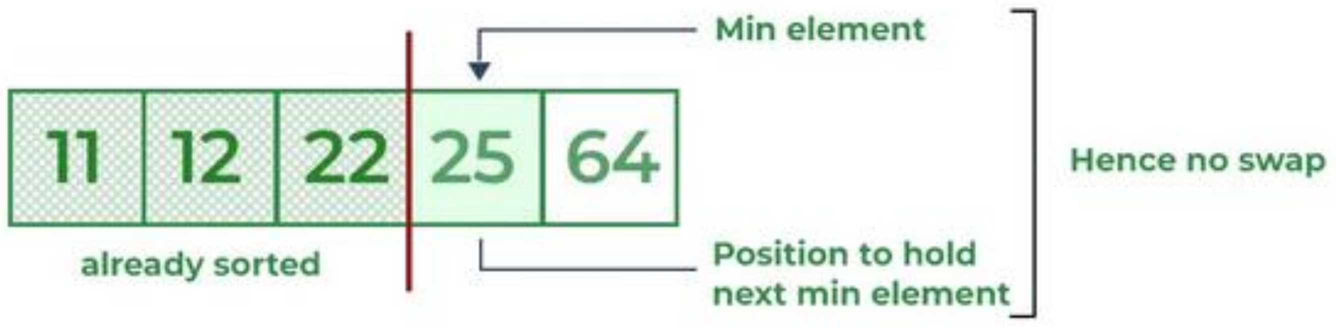


5.2. Selection Sort

- **Second Pass:**
  - For the second position, where 25 is present, again traverse the rest of the array in a sequential manner.
  - After traversing, we found that **12** is the second lowest value in the array and it should appear at the second place in the array, thus swap these values.
- **Third Pass:**
  - Now, for third place, where **25** is present again traverse the rest of the array and find the third least value present in the array.
  - While traversing, **22** came out to be the third least value and it should appear at the third place in the array, thus swap **22** with element present at third position.

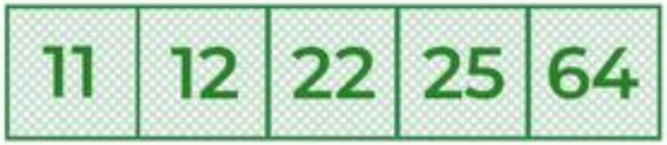
5.2. Selection Sort





5.2. Selection Sort

- **Fourth Pass:**
  - Similarly, for fourth position traverse the rest of the array and find the fourth least element in the array
  - As **25** is the 4th lowest value hence, it will place at the fourth position.
- **Fifth Pass:**
  - At last the largest value present in the array automatically get placed at the last position in the array.
  - The resulted array is the sorted array.



Sorted array

5.2. Selection Sort

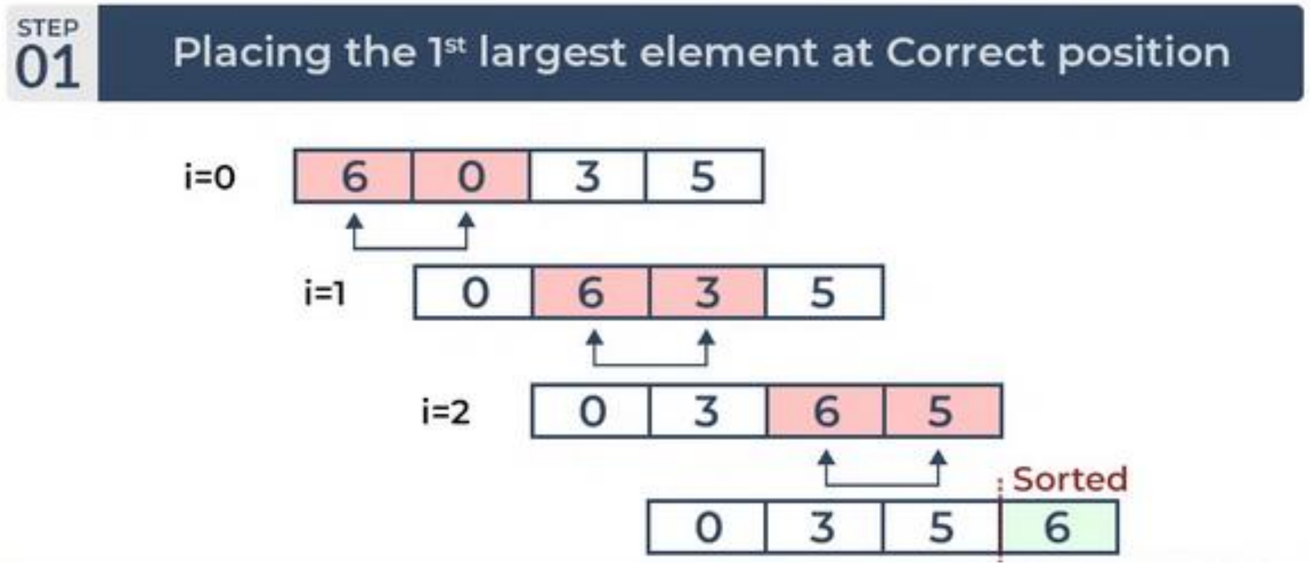
- **Time Complexity:** The time complexity of Selection Sort is **$O(N^2)$** as there are two nested loops:
  - One loop to select an element of Array one by one = O(N)
  - Another loop to compare that element with every other Array element = O(N)
  - Therefore overall complexity = O(N) * O(N) = O(N * N) = $O(N^2)$
- **Auxiliary Space:** O(1) as the only extra memory used is for temporary variables while swapping two values in Array. The selection sort never makes more than O(N) swaps and can be useful when memory writing is costly.

5.3. Bubble Sort

- **Bubble Sort** is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order. This algorithm is not suitable for large data sets, as its average and worst-case time complexity is quite high.
- In this algorithm, traverse from left and compare adjacent elements and the higher one is placed at right side.
- In this way, the largest element is moved to the rightmost end at first.
- This process is then continued to find the second largest and place it and so on until the data is sorted.

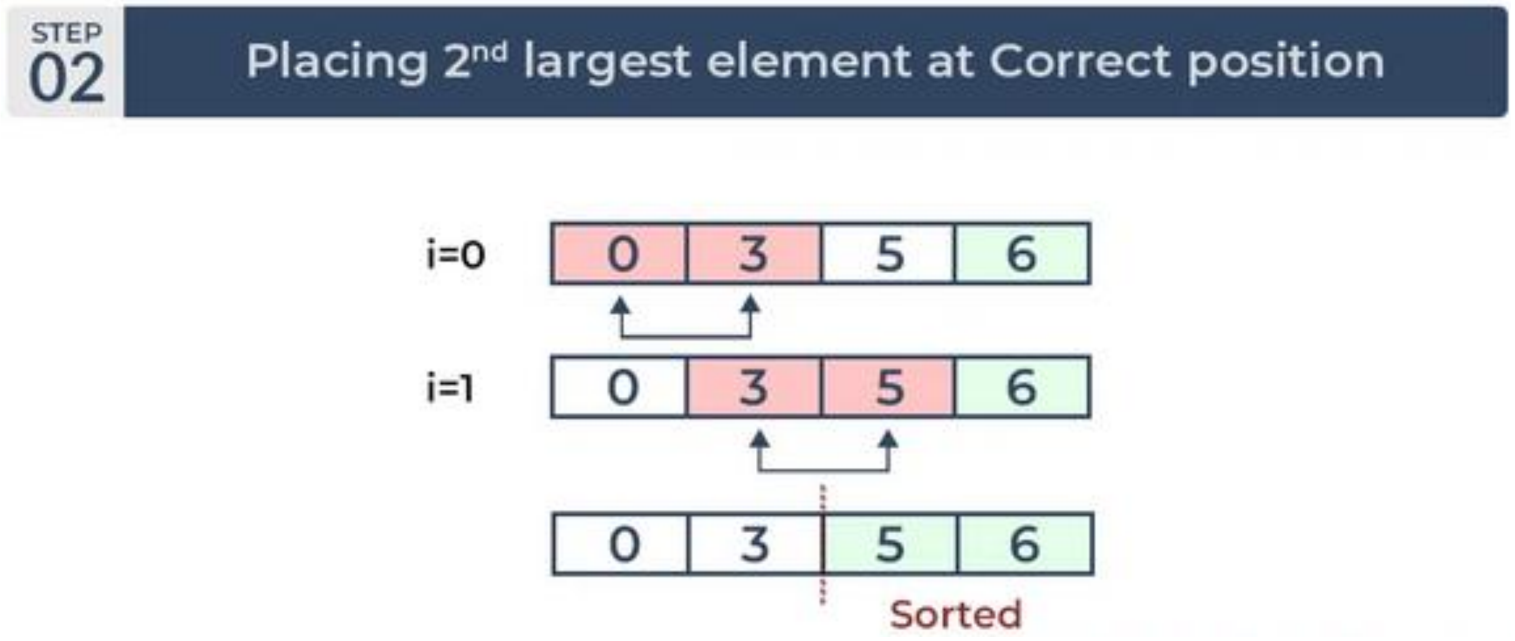
5.3. Bubble Sort

- Let us understand the working of bubble sort with: arr[ ] = {6, 3, 0, 5}
- **First Pass:**
  - The largest element is placed in its correct position, i.e., the end of the array.



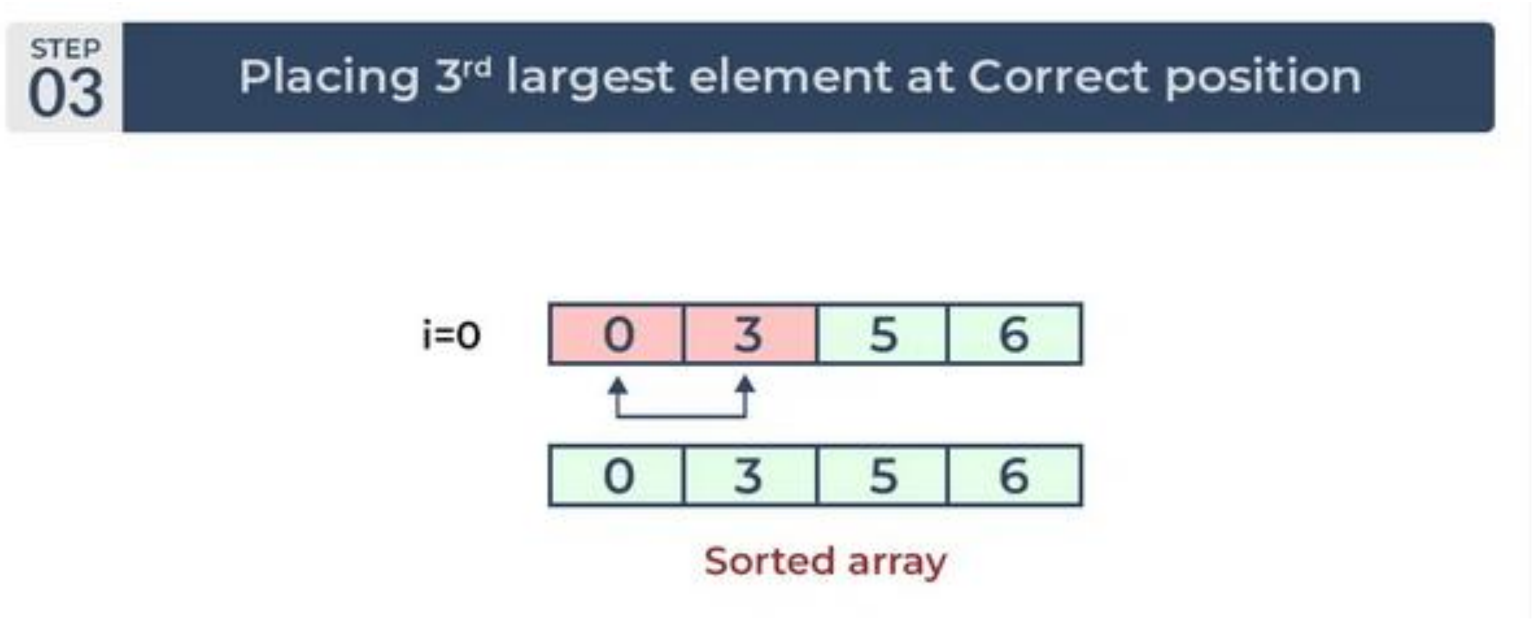
5.3. Bubble Sort

- **Second Pass:**
  - Place the second largest element at correct position.



5.3. Bubble Sort

- **Third Pass:**
  - Place the remaining two elements at their correct positions.



5.3. Bubble Sort

- Total number of passes: *n-1*
- Total number of comparisons: *n*(n-1)/2*
- Time Complexity: **O(N^2)**
- Auxiliary Space: O(1)
- Advantages of Bubble Sort:
  - Bubble sort is easy to understand and implement.
  - It does not require any additional memory space.
  - It is a stable sorting algorithm, meaning that elements with the same key value maintain their relative order in the sorted output.

5.3. Bubble Sort

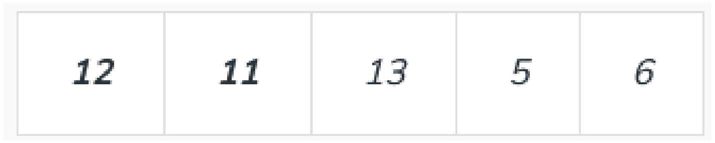
- Disadvantages of Bubble Sort:
  - Bubble sort has a time complexity of $O(N^2)$ which makes it very slow for large data sets.
  - Bubble sort is a comparison-based sorting algorithm, which means that it requires a comparison operator to determine the relative order of elements in the input data set. It can limit the efficiency of the algorithm in certain cases.
- In computer graphics, it is popular for its capability to detect a tiny error (like a swap of just two elements) in almost-sorted arrays and fix it with just linear complexity (2n).

5.4. Insertion Sort

- **Insertion sort** is a simple sorting algorithm that works similar to the way you sort playing cards in your hands. The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part.
- To sort an array of size N in ascending order iterate over the array and compare the current element (key) to its predecessor, if the key element is smaller than its predecessor, compare it to the elements before. Move the greater elements one position up to make space for the swapped element.

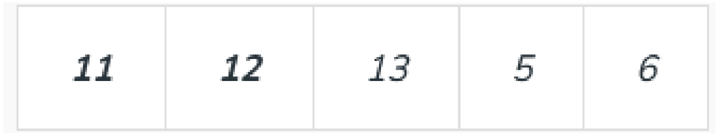
5.4. Insertion Sort

- Consider an example: arr[ ]: {12, 11, 13, 5, 6}
- **First Pass:**
  - Initially, the first two elements of the array are compared in insertion sort.



12	11	13	5	6

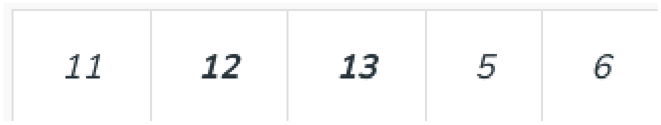
  - Here, 12 is greater than 11 hence they are not in the ascending order and 12 is not at its correct position. Thus, swap 11 and 12.
  - So, for now 11 is stored in a sorted sub-array.



11	12	13	5	6

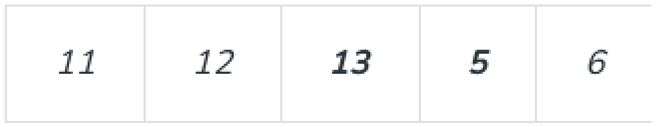
5.4. Insertion Sort

- **Second Pass:**
  - Now, move to the next two elements and compare them



11	**12**	**13**	5	6

  - Here, 13 is greater than 12, thus both elements seems to be in ascending order, hence, no swapping will occur. 12 also stored in a sorted sub-array along with 11.
- **Third Pass:**
  - Now, two elements are present in the sorted sub-array which are **11** and **12**
  - Moving forward to the next two elements which are 13 and 5



11	12	**13**	**5**	6

5.4. Insertion Sort

- Both 5 and 13 are not present at their correct place so swap them

11	12	**5**	**13**	6

- After swapping, elements 12 and 5 are not sorted, thus swap again

11	**5**	**12**	13	6

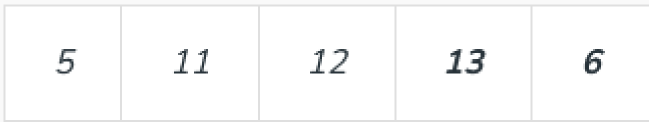
- Here, again 11 and 5 are not sorted, hence swap again

5	**11**	12	13	6

- Here, 5 is at its correct position.

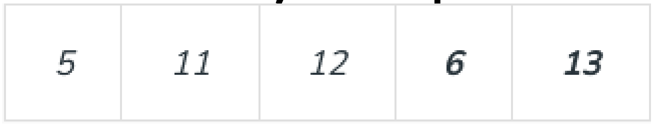
5.4. Insertion Sort

- **Fourth Pass:**
  - Now, the elements which are present in the sorted sub-array are **5, 11** and **12**
  - Moving to the next two elements 13 and 6



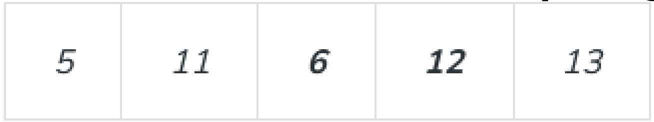
5	11	12	**13**	**6**

  - Clearly, they are not sorted, thus perform swap between both



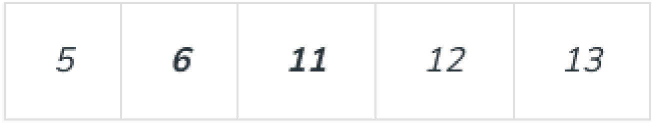
5	11	12	**6**	**13**

  - Now, 6 is smaller than 12, hence, swap again



5	11	**6**	**12**	13

  - Here, also swapping makes 11 and 6 unsorted hence, swap again

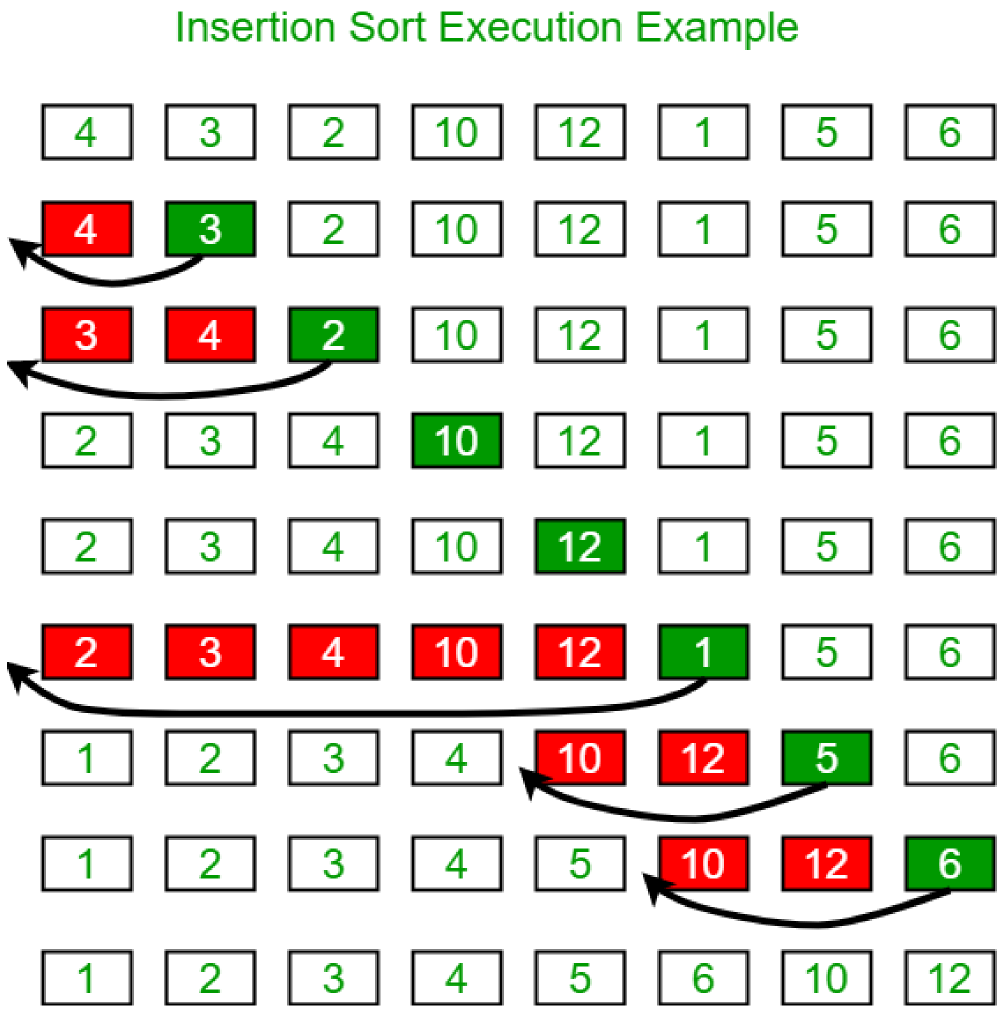


5	**6**	**11**	12	13

  - Finally, the array is completely sorted.

5.4. Insertion Sort

- Illustrations:



5.4. Insertion Sort

- Time Complexity of Insertion Sort
  - The **worst-case** time complexity of the Insertion sort is **O(N^2)**
  - The **average case** time complexity of the Insertion sort is **O(N^2)**
  - The time complexity of the **best case** is **O(N)**.
- Space Complexity of Insertion Sort
  - The auxiliary space complexity of Insertion Sort is **O(1)**

5.4. Insertion Sort

- Characteristics of Insertion Sort:
  - This algorithm is one of the simplest algorithms with a simple implementation.
  - Basically, Insertion sort is efficient for small data values.
  - Insertion sort is adaptive in nature, i.e. it is appropriate for data sets that are already partially sorted.
- The Boundary Cases of the Insertion Sort algorithm:
  - Insertion sort takes the maximum time to sort if elements are sorted in reverse order. And it takes minimum time (order of n) when elements are already sorted.

5.5. Quick Sort

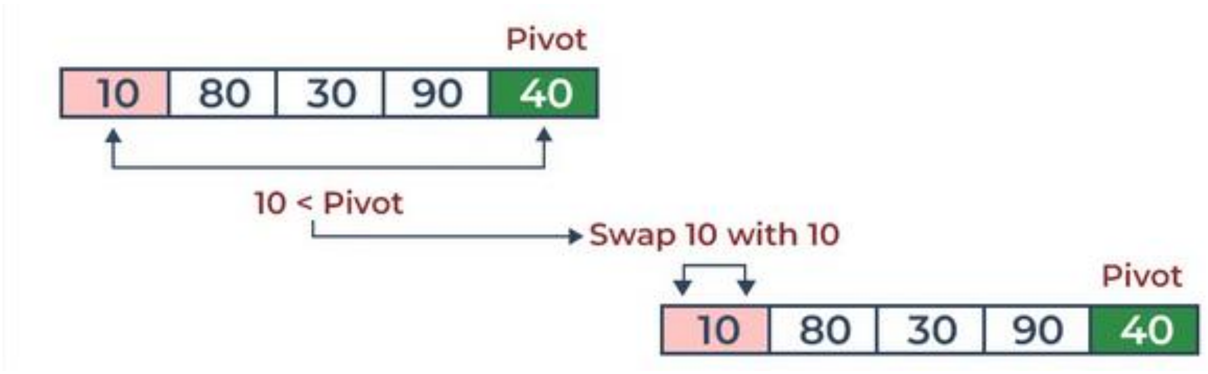
- **Quick Sort** is a sorting algorithm based on the ***Divide and Conquer algorithm*** that picks an element as a pivot, and partitions the given array around the picked pivot by placing the pivot in its correct position in the sorted array.
- There are many different choices for picking pivots:
  - Always pick the first element as a pivot.
  - Always pick the last element as a pivot.
  - Pick a random element as a pivot.
  - Pick the middle as the pivot.

5.5. Quick Sort

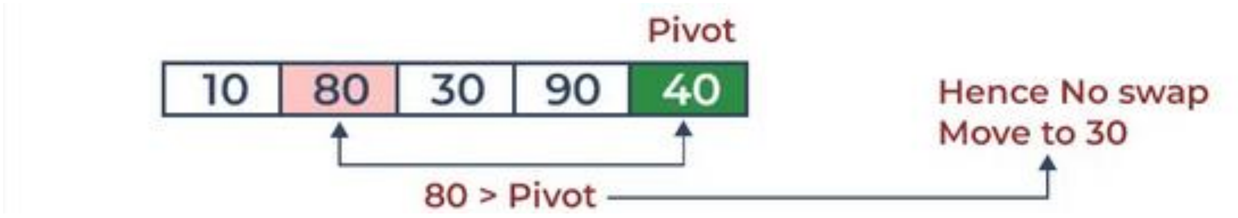
- The key process in **Quick Sort** is a **partition.**
- **Partition Algorithm:** The logic is simple, we start from the leftmost element and keep track of the index of smaller (or equal) elements as **i.** While traversing, if we find a smaller element, we swap the current element with arr[i]. Otherwise, we ignore the current element.
- Let us understand the working of partition and the Quick Sort algorithm with the help of the following example:

5.5. Quick Sort

- Consider: arr[ ] = {10, 80, 30, 90, 40}.
- Compare 10 with the pivot and as it is less than pivot arrange it accordingly.

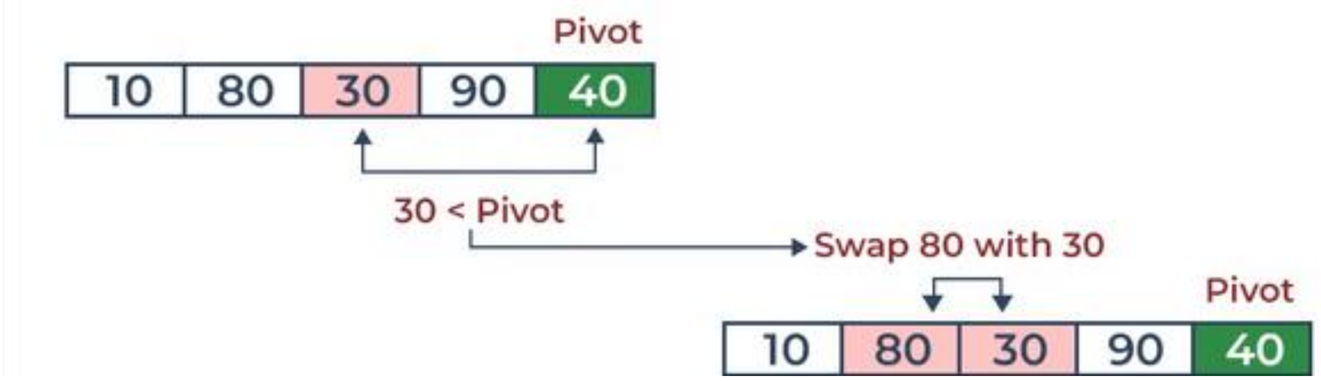


- Compare 80 with the pivot. It is greater than pivot.

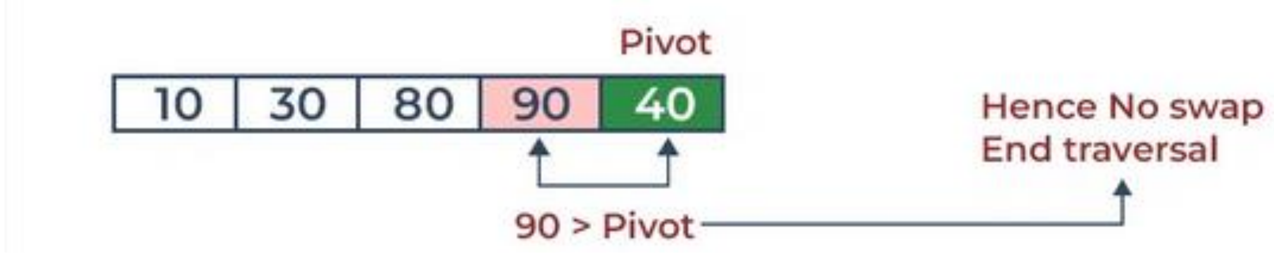


5.5. Quick Sort

- Compare 30 with pivot. It is less than pivot so arrange it accordingly.

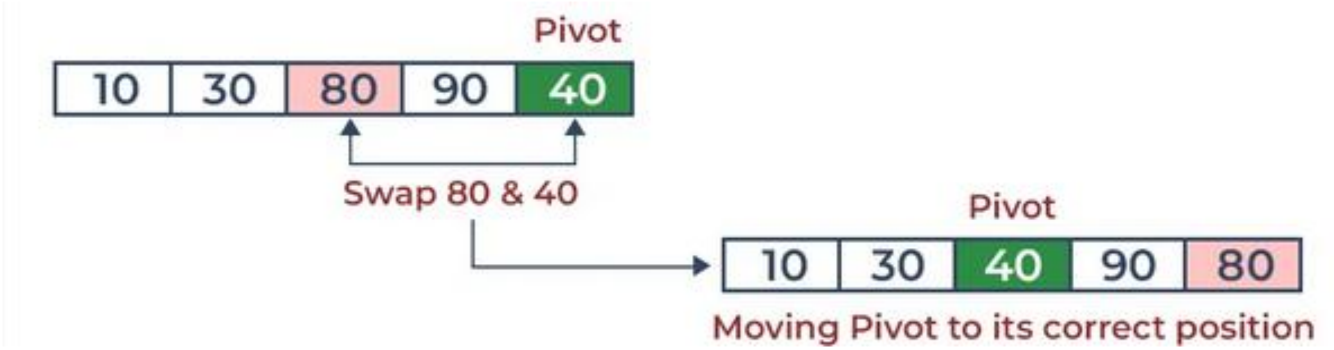


- Compare 90 with the pivot. It is greater than the pivot.



5.5. Quick Sort

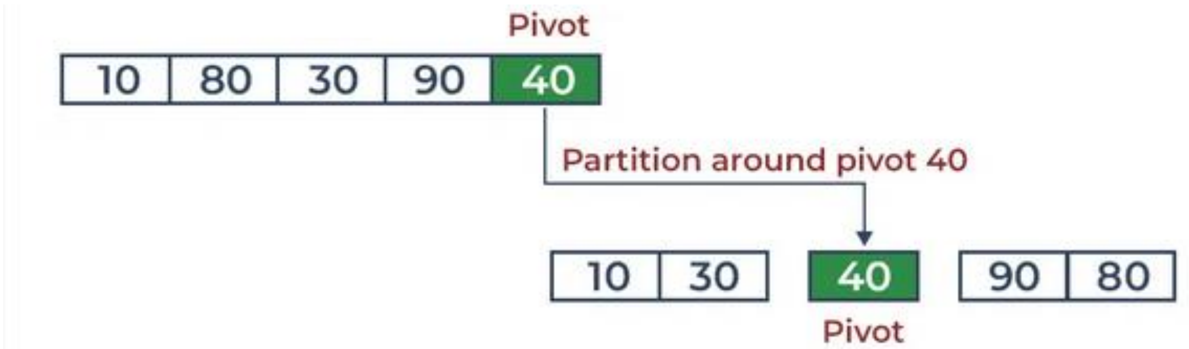
- Arrange the pivot in its correct position.



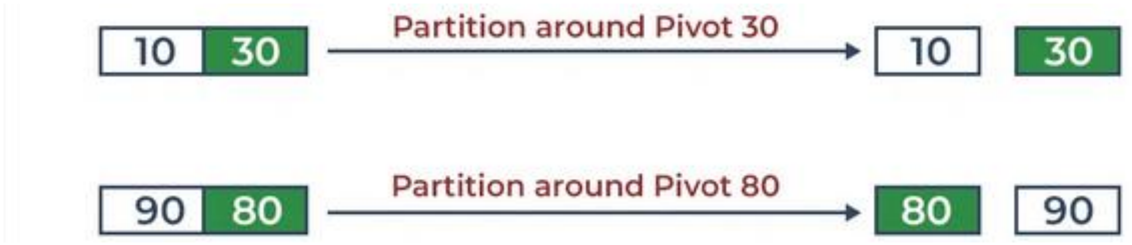
- As the partition process is done recursively, it keeps on putting the pivot in its actual position in the sorted array. Repeatedly putting pivots in their actual position makes the array sorted.
- Follow the below images to understand how the recursive implementation of the partition algorithm helps to sort the array.

5.5. Quick Sort

- Initial partition on the main array:



- Partitioning of the subarrays:



5.5. Quick Sort

- Time Complexity:
  - **Best Case: Ω(N*log (N))**
    The best-case scenario for quick sort occurs when the pivot chosen at the each step divides the array into roughly equal halves. In this case, the algorithm will make balanced partitions, leading to efficient Sorting.
  - **Average Case: θ(N*log (N))**
    Quick sort's average-case performance is usually very good in practice, making it one of the fastest sorting Algorithm.

5.5. Quick Sort

- **Worst Case: O(N^2)**
  The worst-case Scenario for quick sort occurs when the pivot at each step consistently results in highly unbalanced partitions. When the array is already sorted and the pivot is always chosen as the smallest or largest element. To mitigate the worst-case Scenario, various techniques are used such as choosing a good pivot (e.g., median of three) and using Randomized algorithm (***Randomized Quick sort***) to shuffle the element before sorting.

- **Auxiliary Space:** O(1), if we don't consider the recursive stack space. If we consider the recursive stack space then, in the worst case quick sort could make O(*N*).

5.5. Quick Sort

- Advantages of Quick Sort:
  - It is a divide-and-conquer algorithm that makes it easier to solve problems.
  - It is efficient on large data sets.
  - It has a low overhead, as it only requires a small amount of memory to function.

5.5. Quick Sort

- Disadvantages of Quick Sort:
  - It has a worst-case time complexity of $O(N^2)$, which occurs when the pivot is chosen poorly.
  - It is not a good choice for small data sets.
  - It is not a stable sort, meaning that if two elements have the same key, their relative order will not be preserved in the sorted output in case of quick sort, because here we are swapping elements according to the pivot's position (without considering their original positions).

5.6. Merge Sort

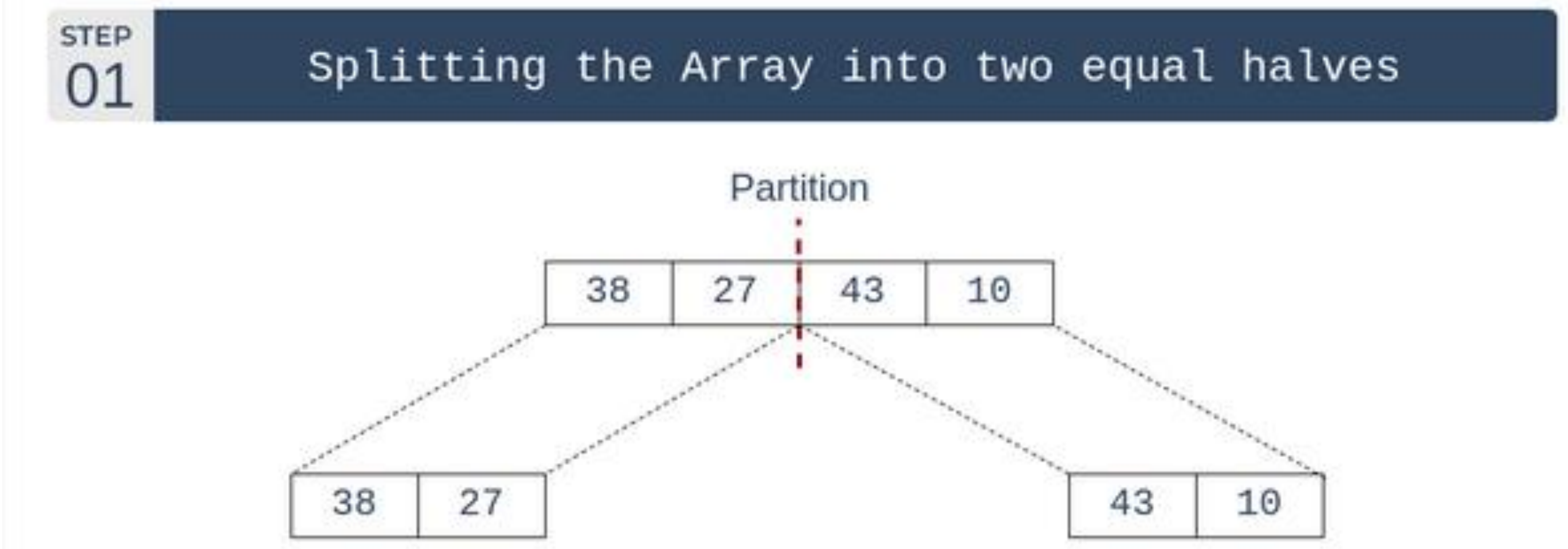
- **Merge sort** is defined as a sorting algorithm that works by dividing an array into smaller subarrays, sorting each subarray, and then merging the sorted subarrays back together to form the final sorted array.
- In simple terms, we can say that the process of merge sort is to divide the array into two halves, sort each half, and then merge the sorted halves back together. This process is repeated until the entire array is sorted.

5.6. Merge Sort

- Merge sort is a recursive algorithm that continuously splits the array in half until it cannot be further divided, i.e., the array has only one element left (an array with one element is always sorted). Then the sorted subarrays are merged into one sorted array.
- See the below illustration to understand the working of merge sort.

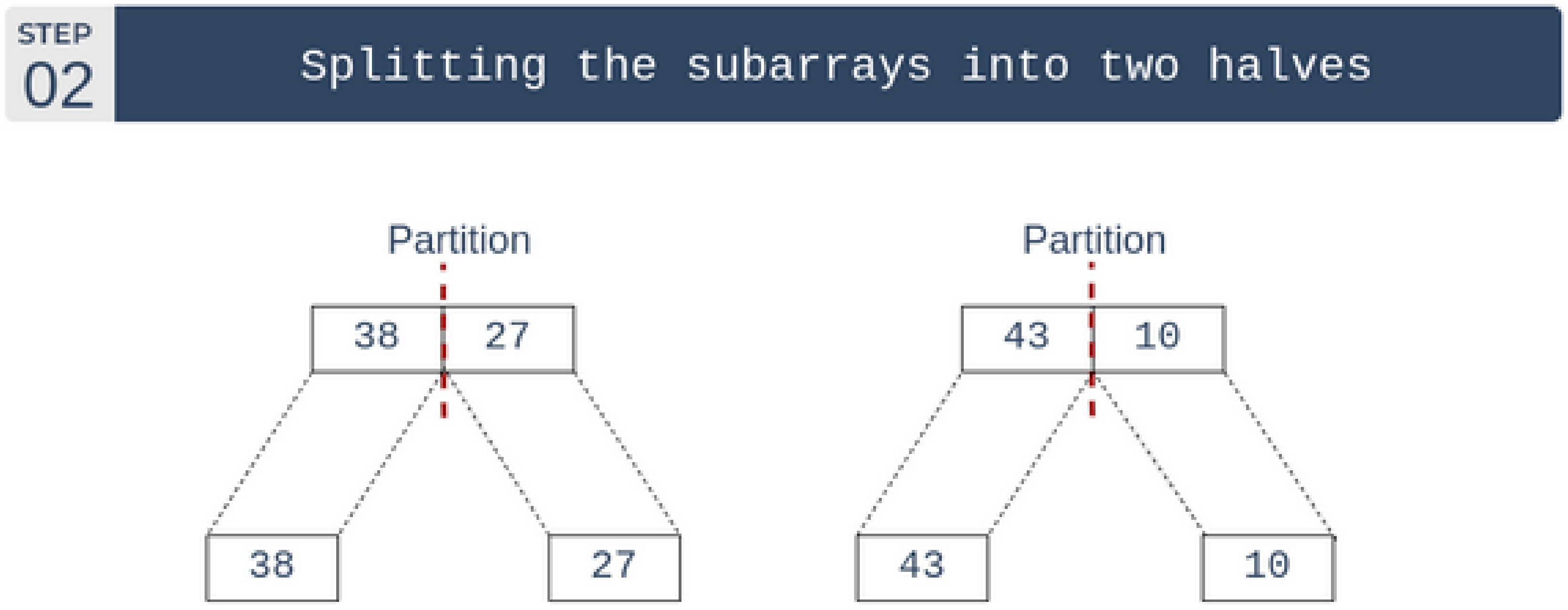
5.6. Merge Sort

- Lets consider an array arr[ ] = {38, 27, 43, 10}
- Initially divide the array into two equal halves:



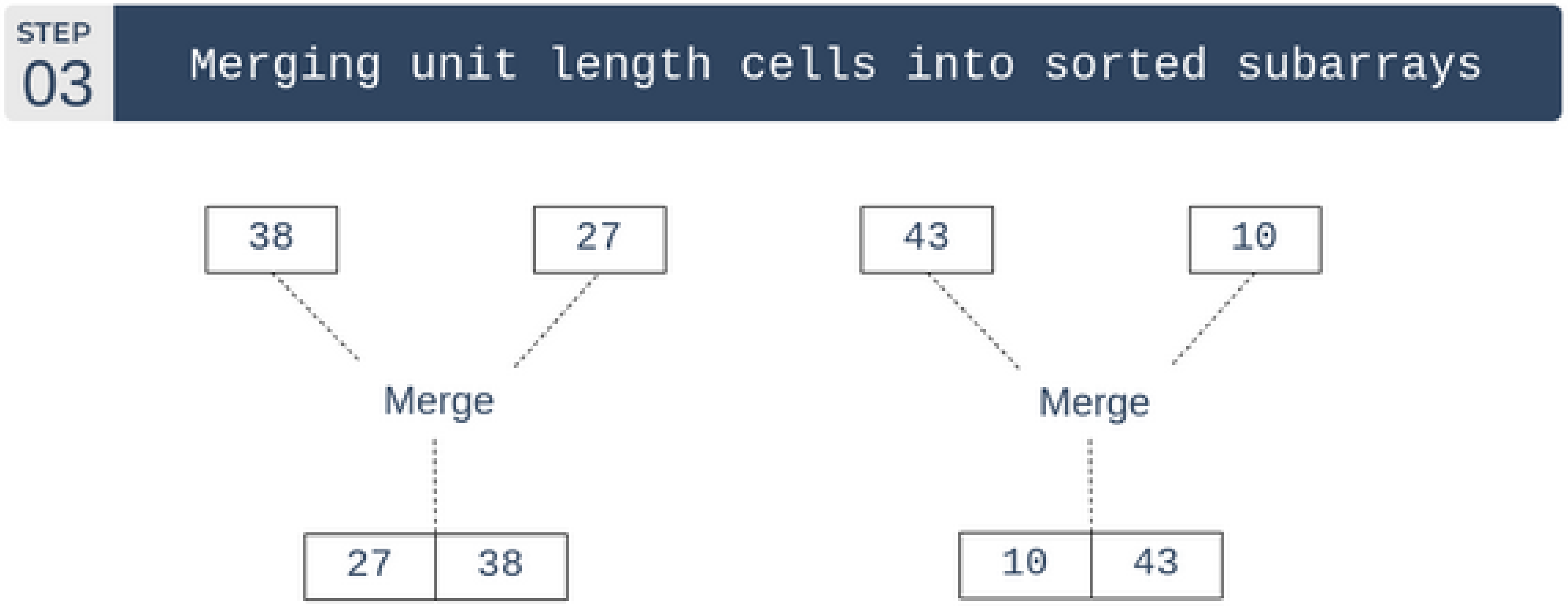
5.6. Merge Sort

- These subarrays are further divided into two halves. Now they become array of unit length that can no longer be divided and array of unit length are always sorted.



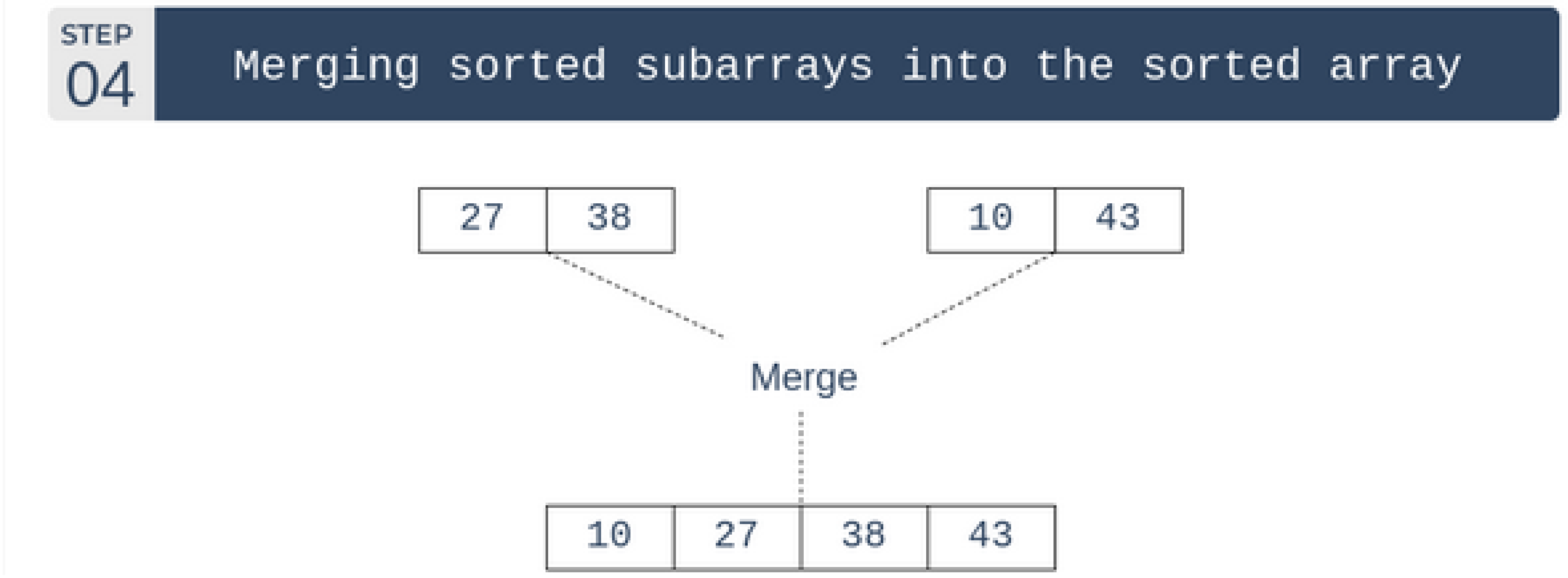
5.6. Merge Sort

- These sorted subarrays are merged together, and we get bigger sorted subarrays.



5.6. Merge Sort

- This merging process is continued until the sorted array is built from the smaller subarrays.



5.6. Merge Sort

- Time Complexity: **O(N*log(N))**, Merge Sort is a recursive algorithm and time complexity can be expressed as following recurrence relation.

$$T(n) = 2*T(n/2) + \theta(n)$$

  - The above recurrence can be solved either using the ***Recurrence Tree method*** or the ***Master method***. It falls in case II of the Master Method, and the solution of the recurrence is θ(N*log(N)).
  - The time complexity of Merge Sort is θ(N*log(N)) in all 3 cases (worst, average, and best) as merge sort always divides the array into two halves and takes linear time to merge two halves.

5.6. Merge Sort

- **Auxiliary Space:** O(N), In merge sort all elements are copied into an auxiliary array. So N auxiliary space is required for merge sort.
- **Applications of Merge Sort:**
  - **Sorting large datasets:** Merge sort is particularly well-suited for sorting large datasets due to its guaranteed worst-case time complexity of O(n*log(n)).
  - **External sorting:** Merge sort is commonly used in external sorting, where the data to be sorted is too large to fit into memory.
  - **Custom sorting:** Merge sort can be adapted to handle different input distributions, such as partially sorted, nearly sorted, or completely unsorted data.

5.6. Merge Sort

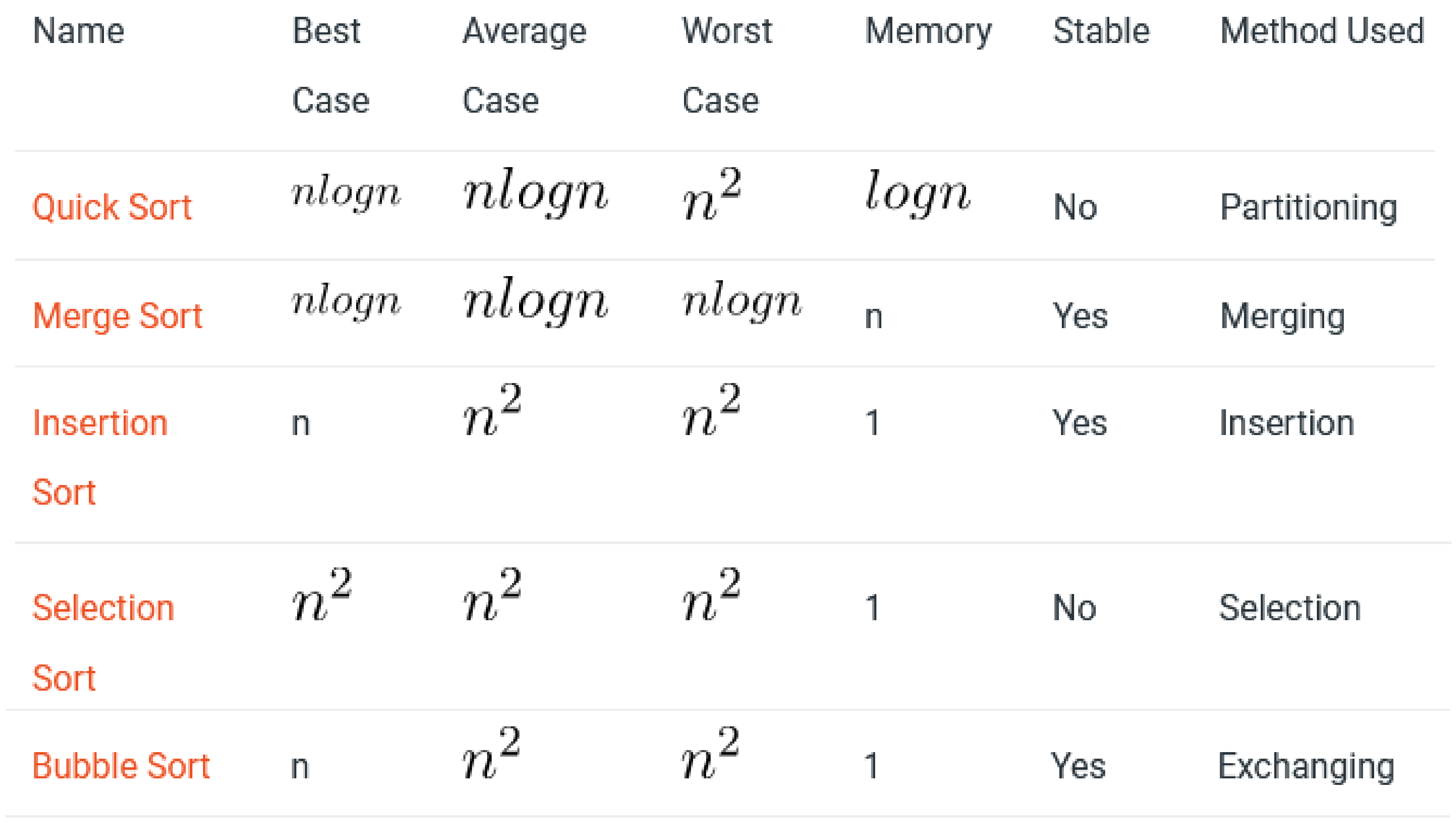
- Advantages of Merge Sort:
  - **Stability**: Merge sort is a stable sorting algorithm, which means it maintains the relative order of equal elements in the input array.
  - **Guaranteed worst-case performance:** Merge sort has a worst-case time complexity of O(N*log(N)), which means it performs well even on large datasets.
  - **Parallelizable**: Merge sort is a naturally parallelizable algorithm, which means it can be easily parallelized to take advantage of multiple processors or threads.

5.6. Merge Sort

- Drawbacks of Merge Sort:
  - **Space complexity:** Merge sort requires additional memory to store the merged sub-arrays during the sorting process.
  - **Not in-place:** Merge sort is not an in-place sorting algorithm, which means it requires additional memory to store the sorted data. This can be a disadvantage in applications where memory usage is a concern.
  - **Not always optimal for small datasets:** For small datasets, Merge sort has a higher time complexity than some other sorting algorithms, such as Insertion sort. This can result in slower performance for very small datasets.

5.7. Complexity Comparison

- Table of Complexity Comparison:



Name	Best Case	Average Case	Worst Case	Memory	Stable	Method Used
Quick Sort	$nlogn$	$nlogn$	n^2	$logn$	No	Partitioning
Merge Sort	$nlogn$	$nlogn$	$nlogn$	n	Yes	Merging
Insertion Sort	n	n^2	n^2	1	Yes	Insertion
Selection Sort	n^2	n^2	n^2	1	No	Selection
Bubble Sort	n	n^2	n^2	1	Yes	Exchanging