Unit 4
Searching & Sorting

What is Sorting?

Sorting refers 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. Sorting means reordering of all the elements either in ascending or in descending order.

Types of Sorting:

1. Selection Sort
2. Bubble Sort
3. Insertion Sort
4. Quick Sort
5. Merge Sort

Selection Sort

Selection sort is a simple and efficient sorting algorithm that works by repeatedly selecting the smallest (or largest) element from the unsorted portion of the list and moving it to the sorted portion of the list.

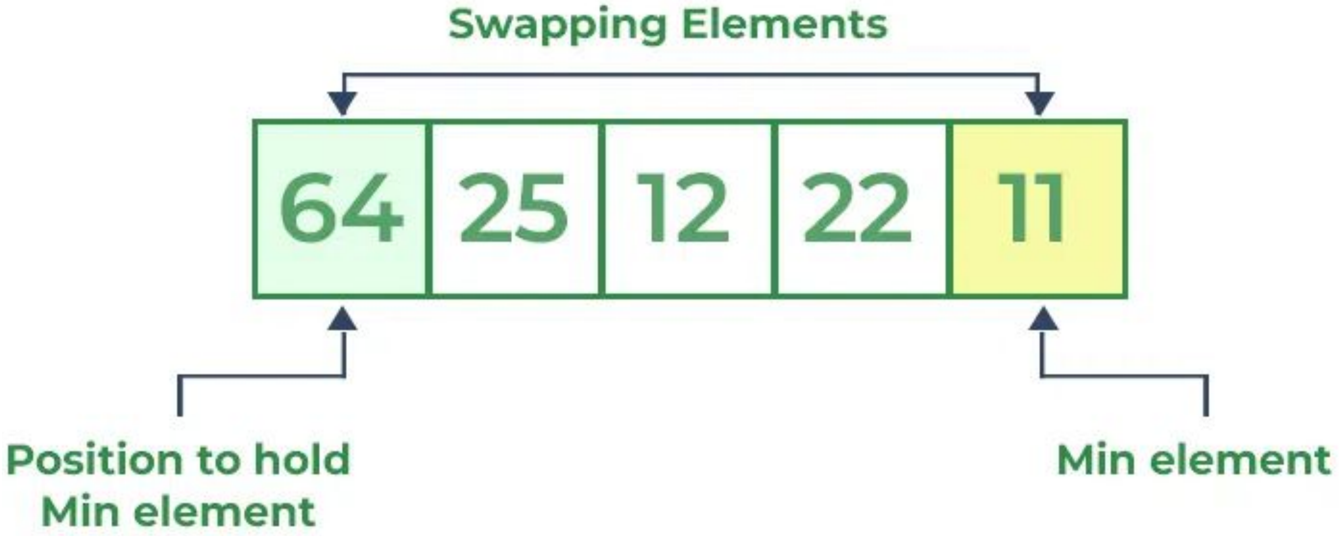
Let's consider the following array as an example: **arr[] = {64, 25, 12, 22, 11}**

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.

Selection Sort Algorithm | Swapping 1st element with the minimum in array

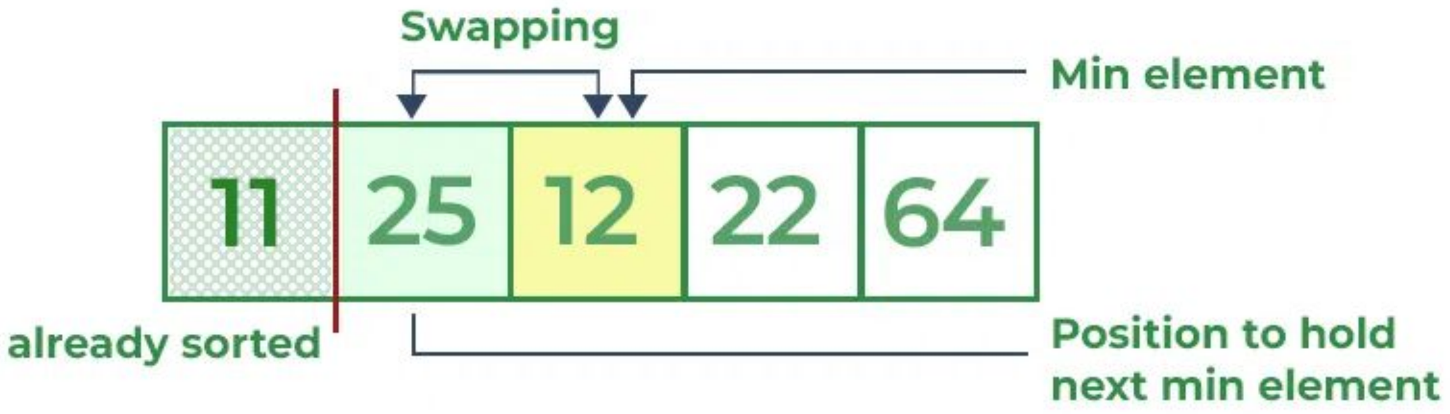
- 

Second Pass:

- For the second position, where 25 is present, again traverse the rest of the array in a sequential manner.

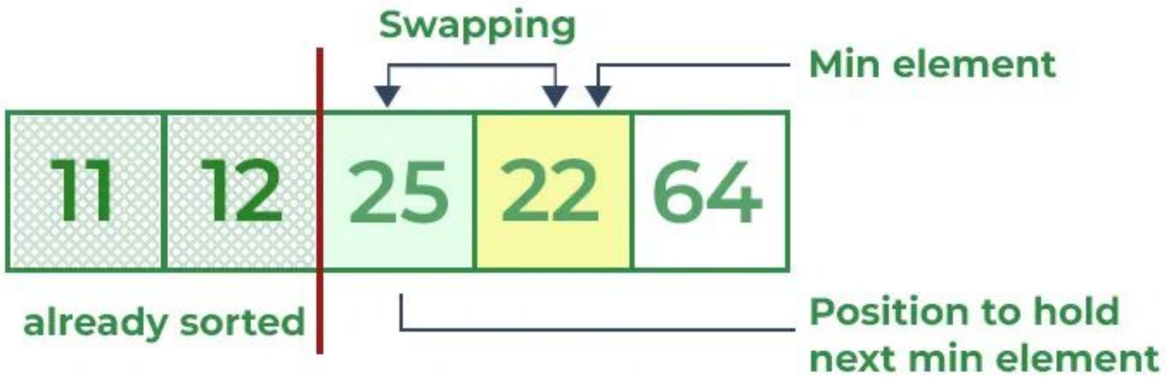
After traversing, we found that **12** is the second lowest value in the array and it should appear at the second place in the array, thus swap these values.

Selection Sort Algorithm | swapping i=1 with the next minimum element



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.

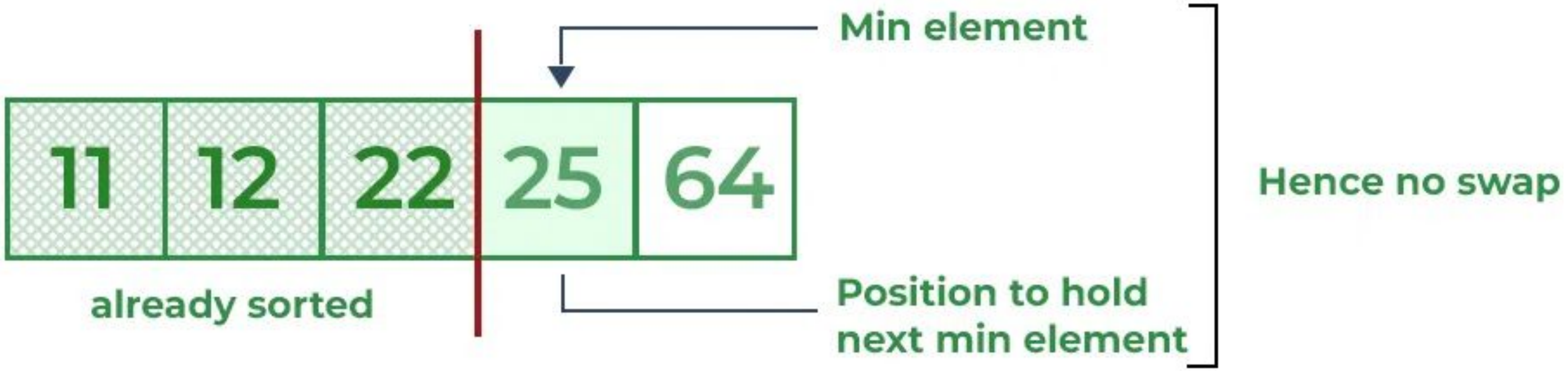


Selection Sort Algorithm | swapping i=2 with the next minimum element

Fourth pass:

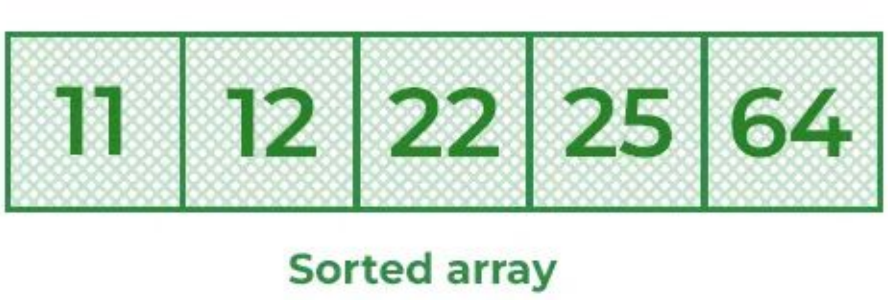
- Similarly, for fourth position traverse the rest of the array and find the fourth least element in the array
- As **25** is the 4th lowest value hence, it will place at the fourth position.

Selection Sort Algorithm | swapping i=3 with the next minimum element



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.



Advantages of Selection Sort Algorithm

- Simple and easy to understand.
- Works well with small datasets.

Disadvantages of the Selection Sort Algorithm

- Selection sort has a time complexity of O(n^2) in the worst and average case.
- Does not work well on large datasets.
- Does not preserve the relative order of items with equal keys which means it is not stable.

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.

Bubble Sort Algorithm

In Bubble Sort 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.*

How does Bubble Sort Work?

Let us understand the working of bubble sort with the help of the following illustration:

Input: *arr[] = {6, 0, 3, 5}*

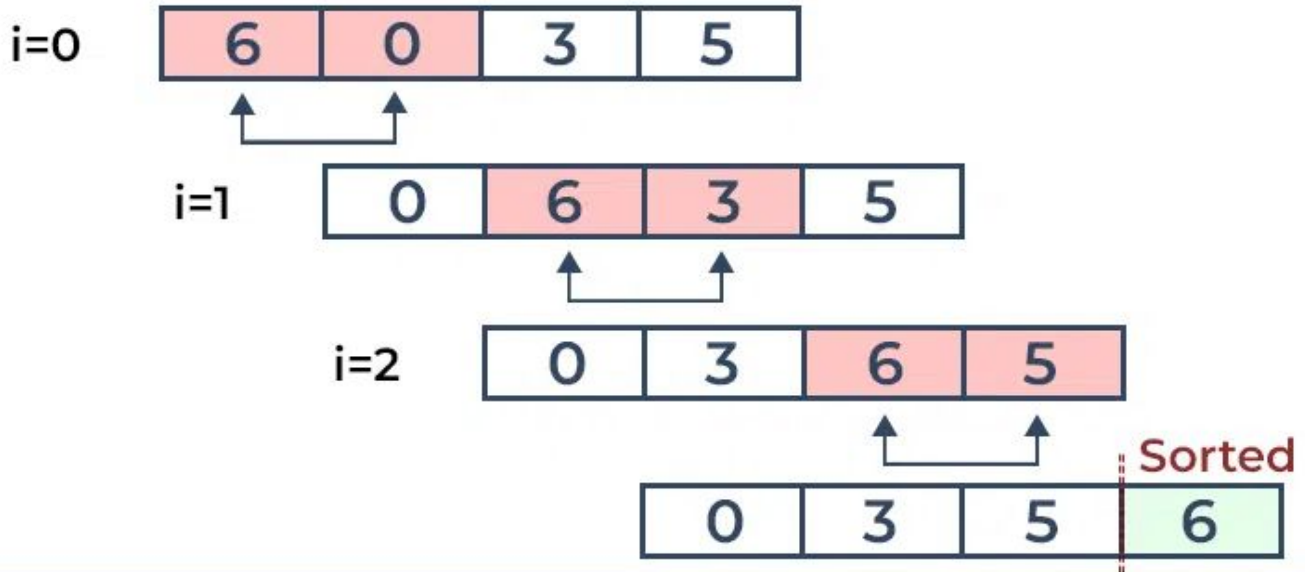
First Pass:

The largest element is placed in its correct position, i.e., the end of the array.

Bubble Sort Algorithm : Placing the largest element at correct position

STEP 01

Placing the 1st largest element at Correct position



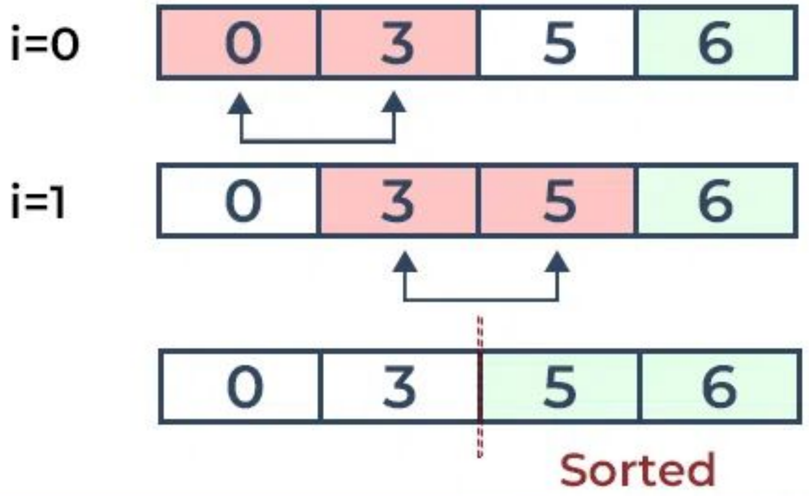
Second Pass:

Place the second largest element at correct position

Bubble Sort Algorithm : Placing the second largest element at correct position

STEP 02

Placing 2nd largest element at Correct position



Third Pass:

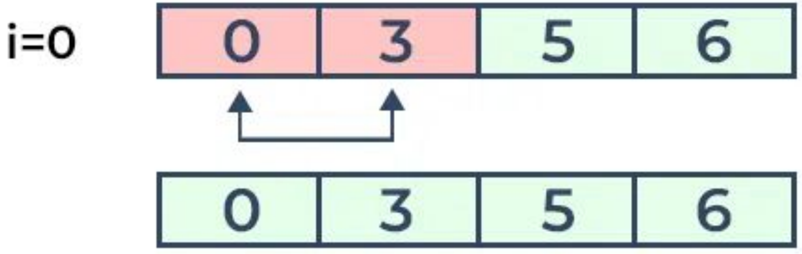
Place the remaining two elements at their correct positions.

Bubble Sort Algorithm : Placing the remaining elements at their correct positions

- **Total no. of passes:** n-1
- **Total no. of comparisons:** n*(n-1)/2

STEP 03

Placing 3rd largest element at Correct position



Sorted array

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.

Disadvantages of Bubble Sort:

- Bubble sort has a time complexity of O(N2) 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.

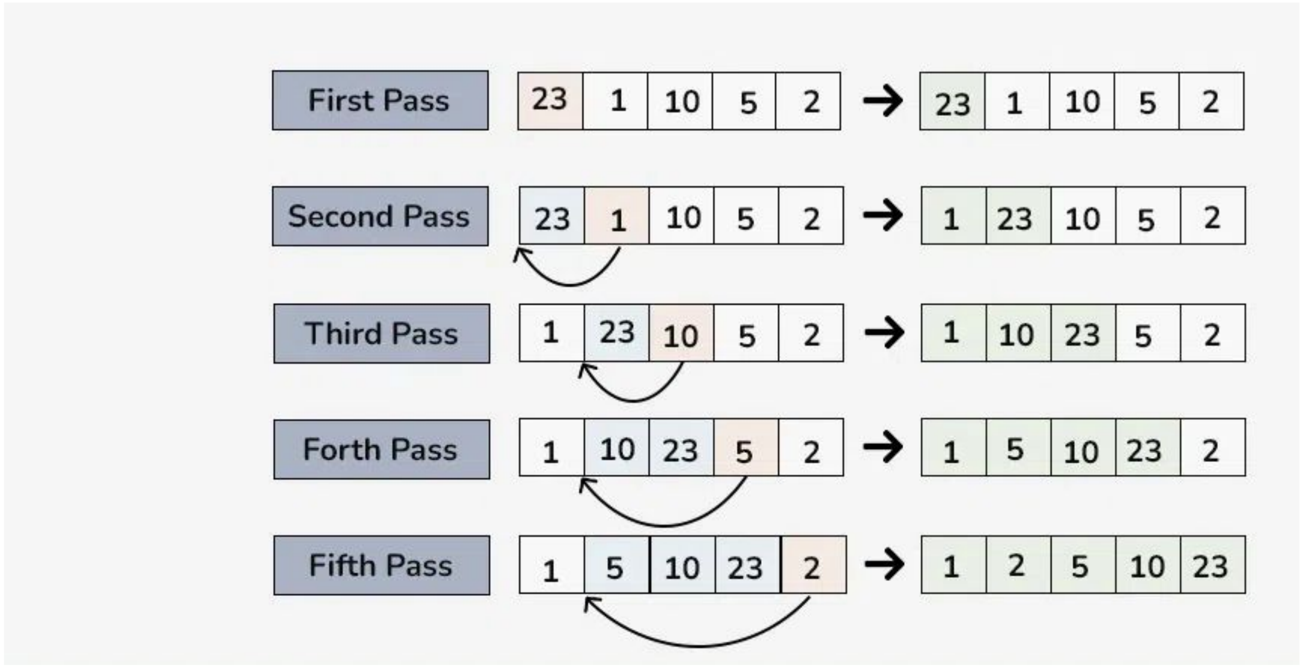
Insertion Sort

Insertion sort is a simple sorting algorithm that works by iteratively inserting each element of an unsorted list into its correct position in a sorted portion of the list. It is a **stable sorting** algorithm, meaning that elements with equal values maintain their relative order in the sorted output.

Insertion sort is like sorting playing cards in your hands. You split the cards into two groups: the sorted cards and the unsorted cards. Then, you pick a card from the unsorted group and put it in the right place in the sorted group.

Working of Insertion Sort Algorithm:

Consider an array having elements: ***{23, 1, 10, 5, 2}***



First Pass:

- *Current element is **23***
- *The first element in the array is assumed to be sorted.*
- *The sorted part until **0th** index is : **[23]***

Second Pass:

- *Compare **1** with **23** (current element with the sorted part).*
- *Since **1** is smaller, insert **1** before **23**.*
- *The sorted part until **1st** index is: **[1, 23]***

Third Pass:

- *Compare **10** with **1** and **23** (current element with the sorted part).*
- *Since **10** is greater than **1** and smaller than **23**, insert **10** between **1** and **23**.*
- *The sorted part until **2nd** index is: **[1, 10, 23]***

Fourth Pass:

- *Compare **5** with **1**, **10**, and **23** (current element with the sorted part).*
- *Since **5** is greater than **1** and smaller than **10**, insert **5** between **1** and **10**.*
- *The sorted part until **3rd** index is**: [1, 5, 10, 23]***

Fifth Pass:

- *Compare 2 with **1, 5, 10**, and **23** (current element with the sorted part).*
- *Since 2 is smaller than all elements in the sorted part, insert **2** at the beginning.*
- *The sorted part until **4th** index is: **[2, 1, 5, 10, 23]***

Final Array:

- *The sorted array is: **[2, 1, 5, 10, 23]***

Advantages of Insertion Sort:

- Simple and easy to implement.
- Stable sorting algorithm.
- Efficient for small lists and nearly sorted lists.
- Space-efficient.

Disadvantages of Insertion Sort:

- Inefficient for large lists.
- Not as efficient as other sorting algorithms (e.g., merge sort, quick sort) for most cases.

Applications of Insertion Sort:

Insertion sort is commonly used in situations where:

- The list is small or nearly sorted.
- Simplicity and stability are important.

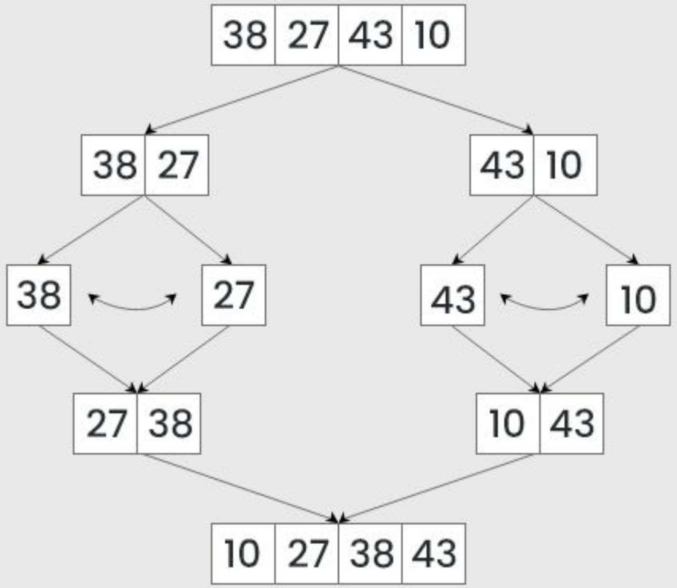
Merge Sort

Merge sort is a sorting algorithm that follows the **divide-and-conquer** approach. It works by recursively dividing the input array into smaller subarrays and sorting those subarrays then merging them back together to obtain the 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.

Merge Sort

Algorithm



How does Merge Sort work?

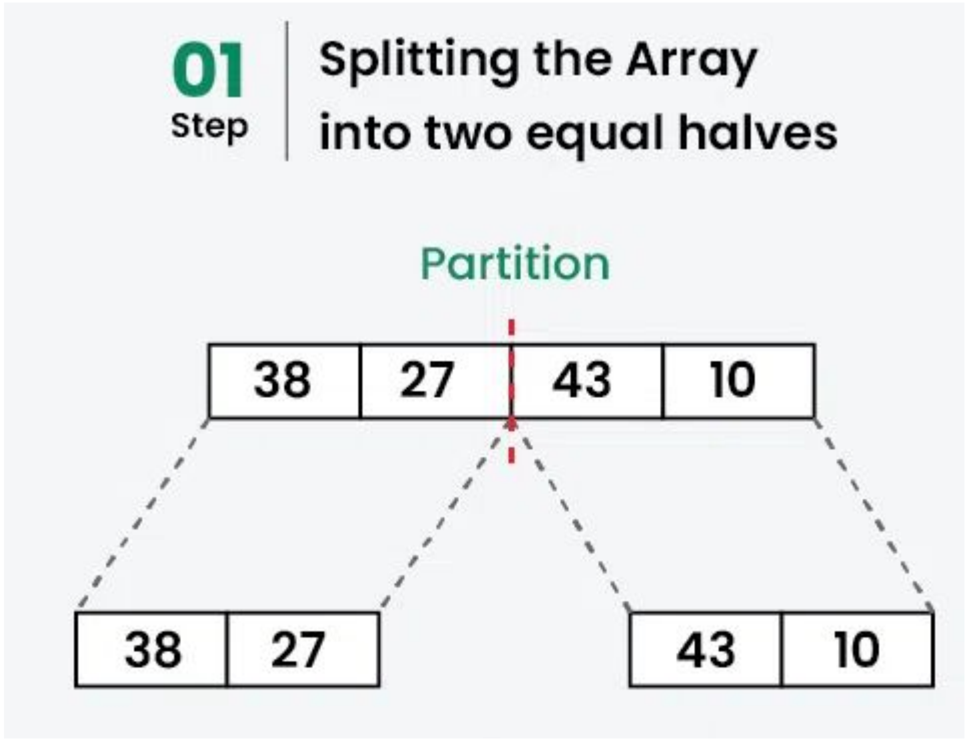
Merge sort is a popular sorting algorithm known for its efficiency and stability. It follows the **divide-and-conquer** approach to sort a given array of elements.

Here’s a step-by-step explanation of how merge sort works:

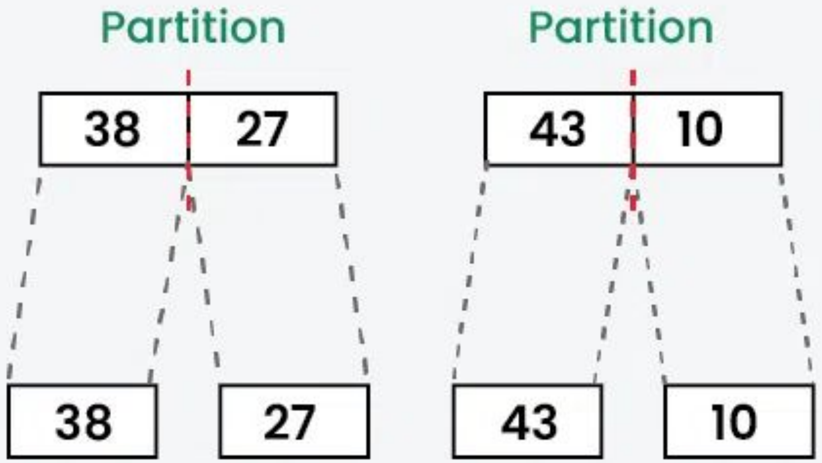
1. **Divide:** Divide the list or array recursively into two halves until it can no more be divided.
2. **Conquer:** Each subarray is sorted individually using the merge sort algorithm.
3. **Merge:** The sorted subarrays are merged back together in sorted order. The process continues until all elements from both subarrays have been merged.

Illustration of Merge Sort:

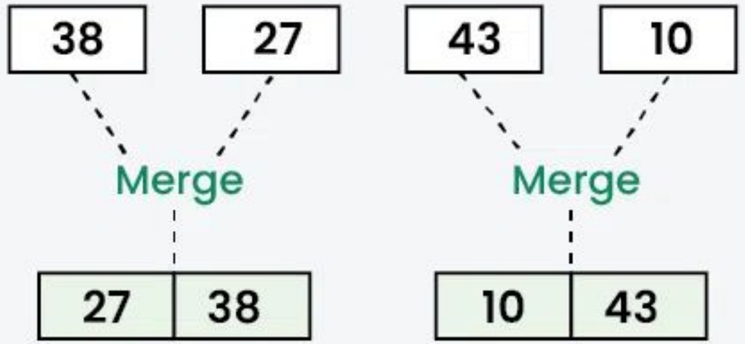
Let’s sort the array or list **[38, 27, 43, 10]** using Merge Sort



02 Step | Splitting the subarrays into two halves

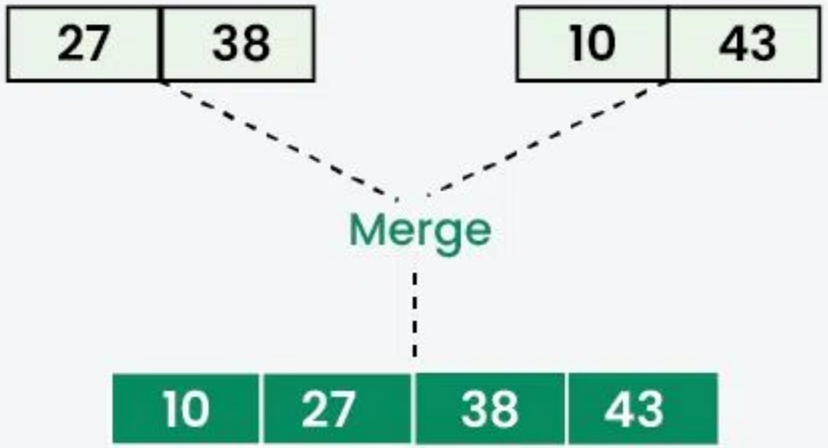


03 Step | Merging unit length cells into sorted subarrays



04
Step

Merging sorted subarrays into the sorted array



Let’s look at the working of above example:

Divide:

- ***[38, 27, 43, 10]*** *is divided into* ***[38, 27***_] and_ ***[43, 10]***_._
- ***[38, 27]*** *is divided into* ***[38]*** *and* ***[27]***_._
- ***[43, 10]*** *is divided into* ***[43]*** *and* ***[10]***_._

Conquer:

- ***[38]*** *is already sorted.*
- ***[27]*** *is already sorted.*
- ***[43]*** *is already sorted.*
- ***[10]*** *is already sorted.*

Merge:

- *Merge **[38]** and **[27]** to get **[27, 38]**.*
- *Merge **[43]** and **[10]** to get **[10,43]**.*
- *Merge **[27, 38]** and **[10,43]** to get the final sorted list **[10, 27, 38, 43]***

*Therefore, the sorted list is **[10, 27, 38, 43]**.*

Applications of Merge Sort:

- Sorting large datasets
- External sorting (when the dataset is too large to fit in memory)
- Inversion counting (counting the number of inversions in an array)
- Finding the median of an array

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 logN)**, which means it performs well even on large datasets.
- **Simple to implement:** The divide-and-conquer approach is straightforward.

Disadvantage 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.

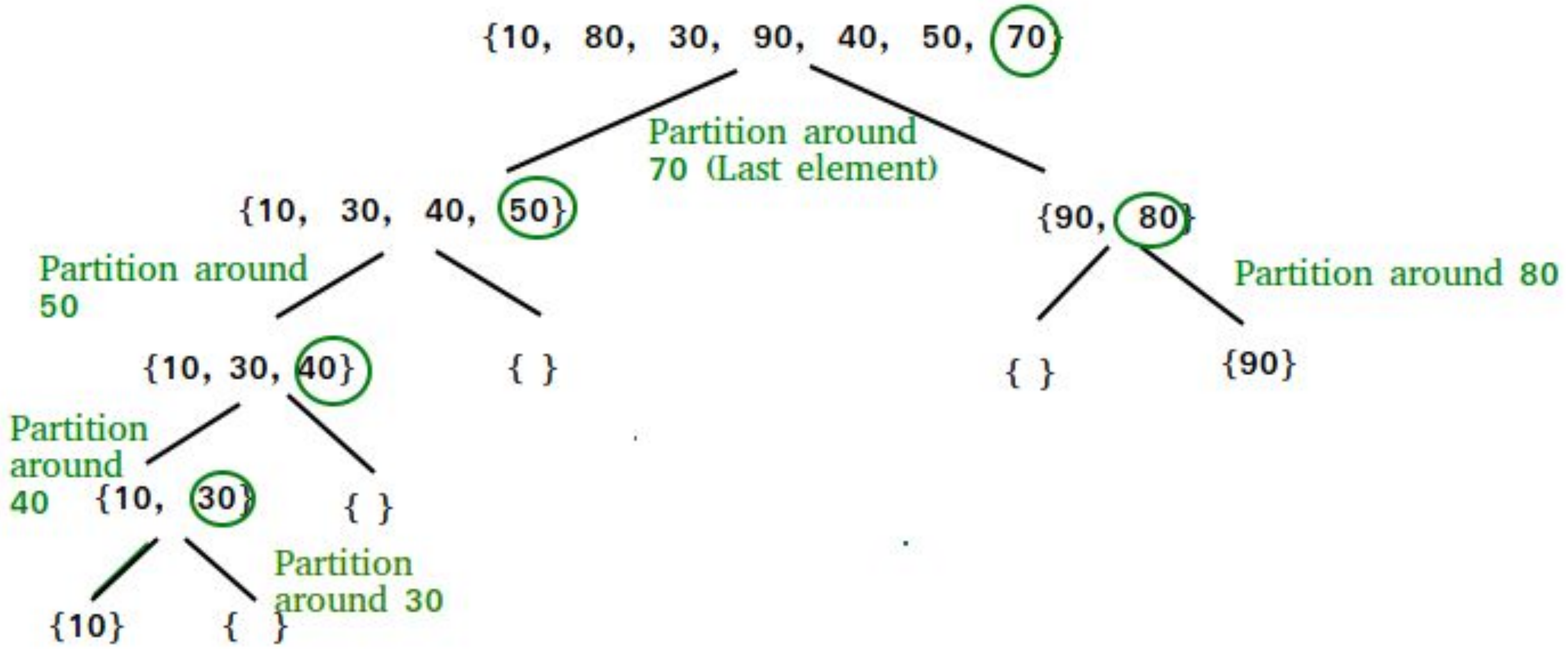
QuickSort

QuickSort 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.

How does QuickSort work?

The key process in **quickSort** is a **partition()**. The target of partitions is to place the pivot (any element can be chosen to be a pivot) at its correct position in the sorted array and put all smaller elements to the left of the pivot, and all greater elements to the right of the pivot.

Partition is done recursively on each side of the pivot after the pivot is placed in its correct position and this finally sorts the array.


{10, 80, 30, 90, 40, 50, 70}
Partition around 70 (Last element)
{10, 30, 40, 50}
{90, 80}
Partition around 50
Partition around 80
{10, 30, 40}
{ }
{ }
{90}
Partition around 40
{10, 30}
{ }
Partition around 30
{10}
{ }

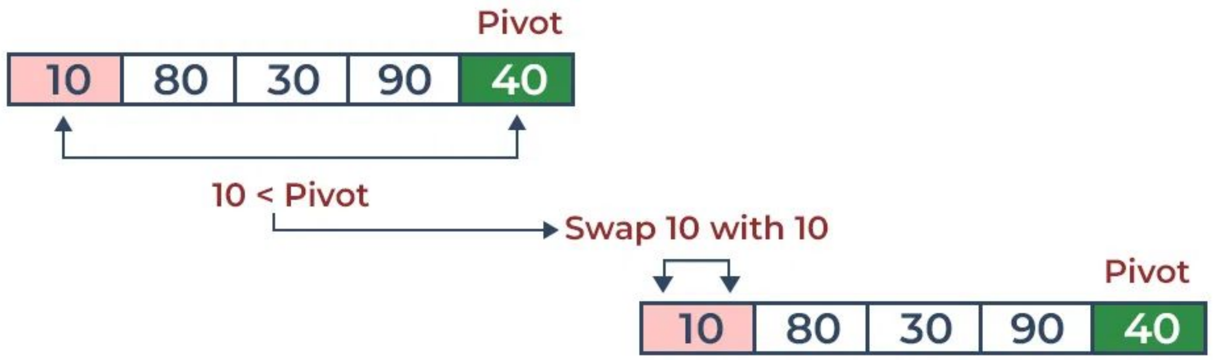
Choice of Pivot:

There are many different choices for picking pivots.

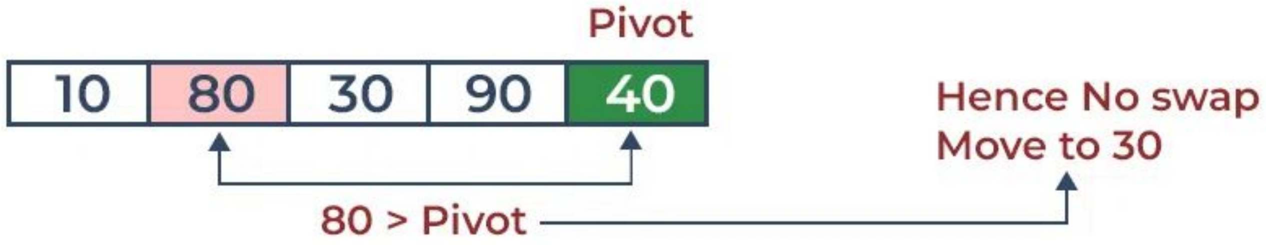
- Always pick the first element as a pivot.
- Always pick the last element as a pivot (implemented below)
- Pick the random element as a pivot.
- Pick the middle as the pivot.

Consider: arr[] = {10, 80, 30, 90, 40}.

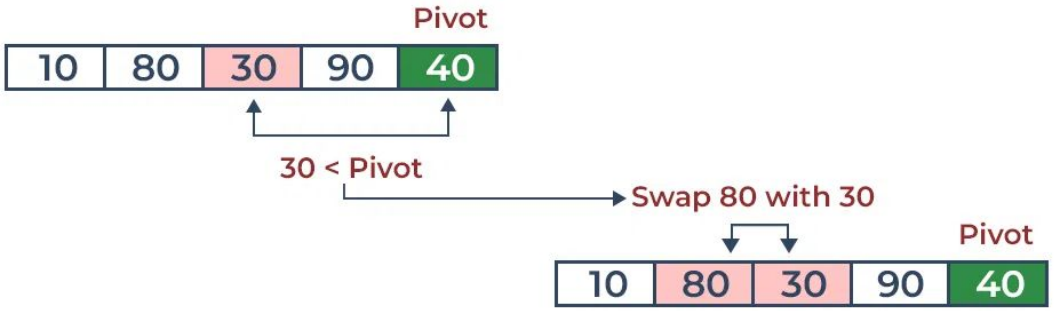
- *Compare 10 with the pivot and as it is less than pivot arrange it accrodingly.*



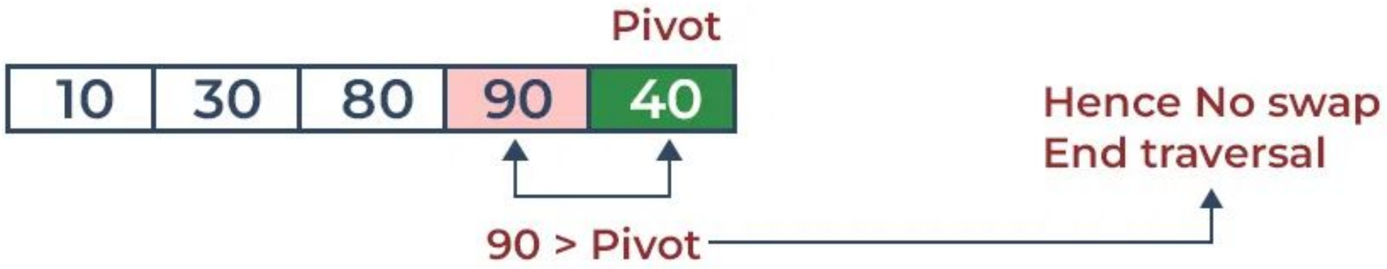
- Compare 80 with the pivot. It is greater than pivot.



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



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



- *Arrange the pivot in its correct position.*

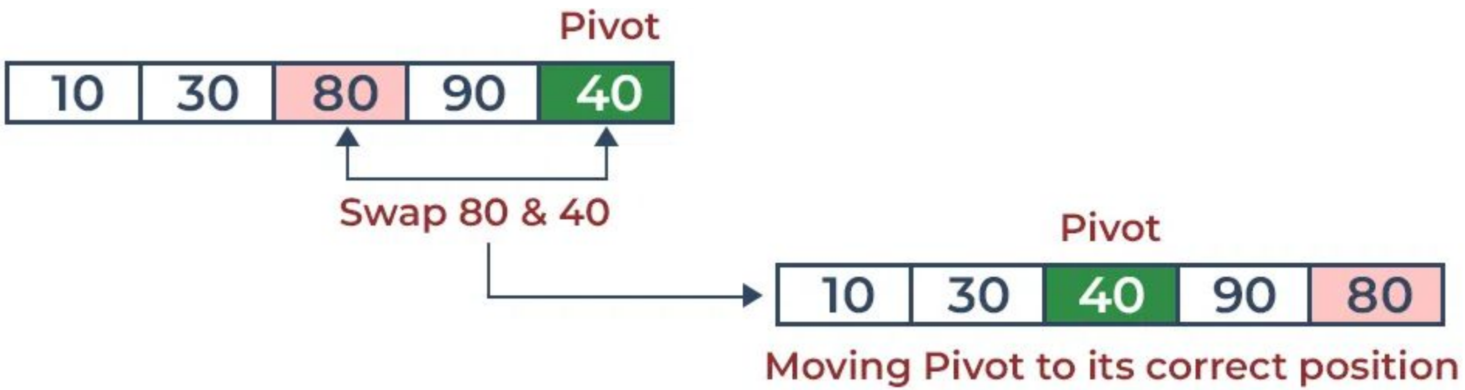
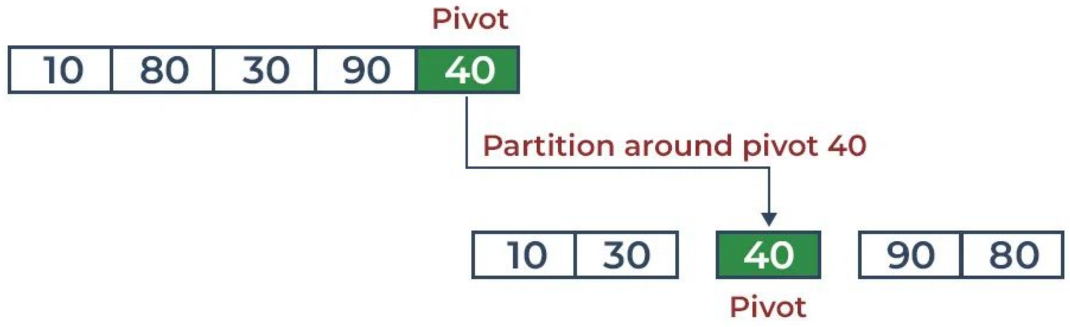

Pivot
10
30
80
90
40
Swap 80 & 40
Pivot
10
30
40
90
80
Moving Pivot to its correct position

Illustration of Quicksort:

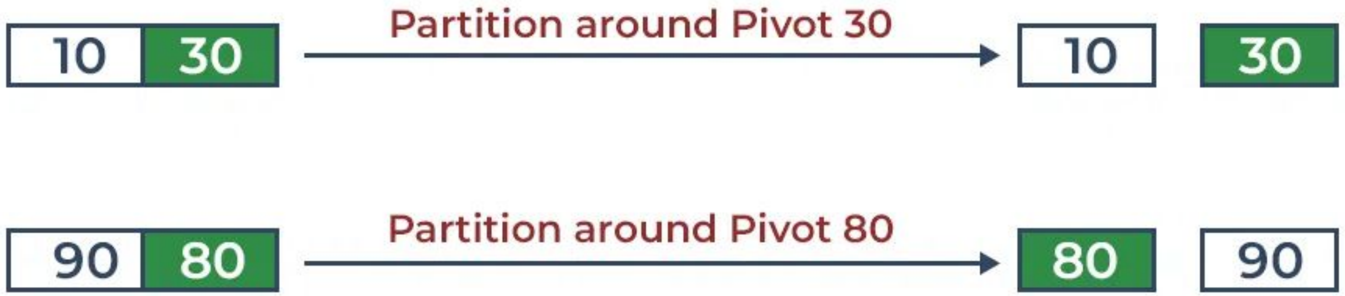
As the partition process is done recursively, it keeps on putting the pivot in its actual position in the sorted array. Repeatedly putting pivots in their actual position makes the array sorted.

Follow the below images to understand how the recursive implementation of the partition algorithm helps to sort the array.

- Initial partition on the main array:



- *Partitioning of the subarrays:*



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.

Disadvantages of Quick Sort:

- It has a worst-case time complexity of O(N2), 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)

Searching algorithms are essential tools in computer science used to locate specific items within a collection of data. These algorithms are designed to efficiently navigate through data structures to find the desired information, making them fundamental in various applications such as **databases, web search engines**, and more.

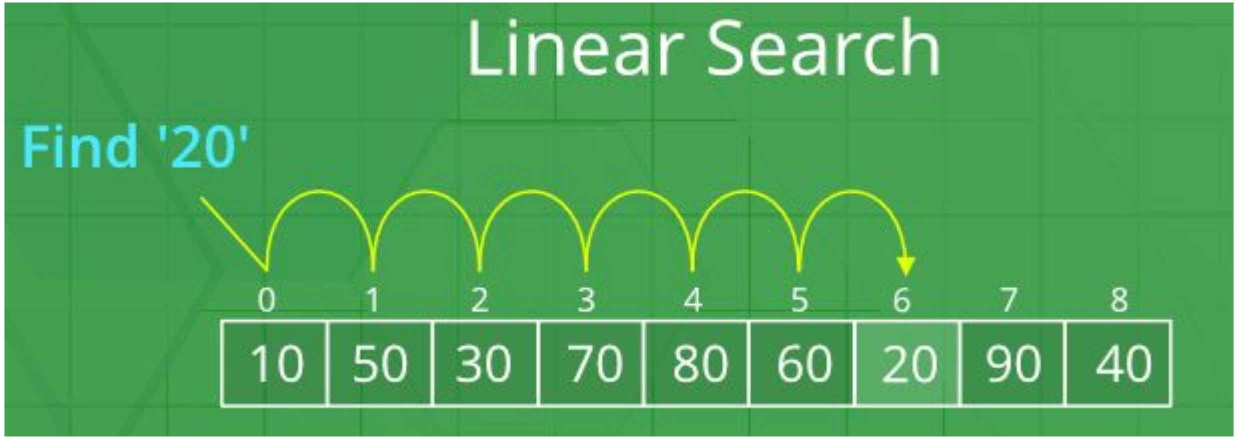
Searching terminologies:

Target Element:

In searching, there is always a specific target element or item that you want to find within the data collection. This target could be a value, a record, a key, or any other data entity of interest.

Linear Search

Linear Search is defined as a sequential search algorithm that starts at one end and goes through each element of a list until the desired element is found, otherwise the search continues till the end of the data set.



How Does Linear Search Algorithm Work?

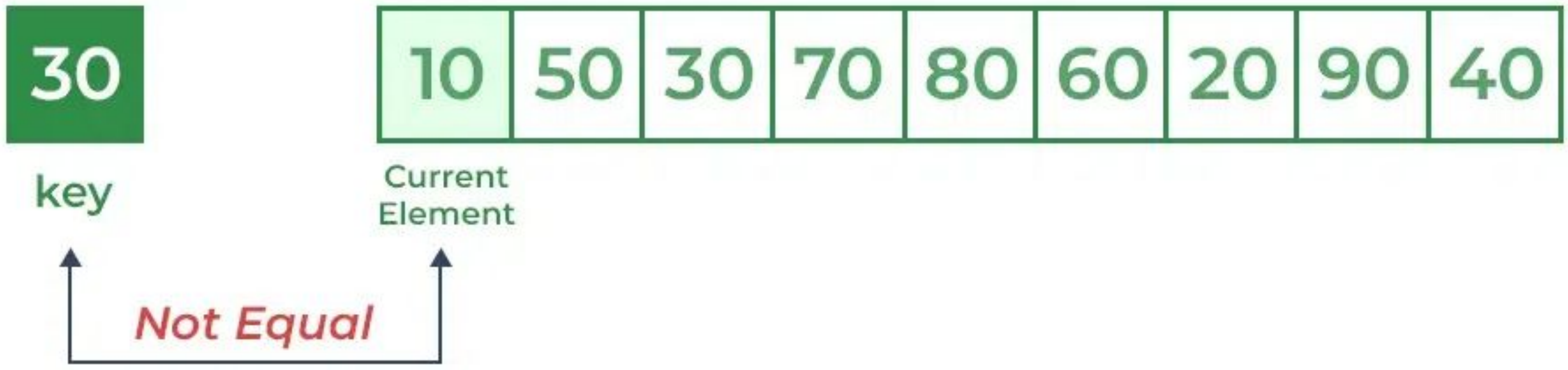
In Linear Search Algorithm,

- Every element is considered as a potential match for the key and checked for the same.
- If any element is found equal to the key, the search is successful and the index of that element is returned.
- If no element is found equal to the key, the search yields “No match found”.

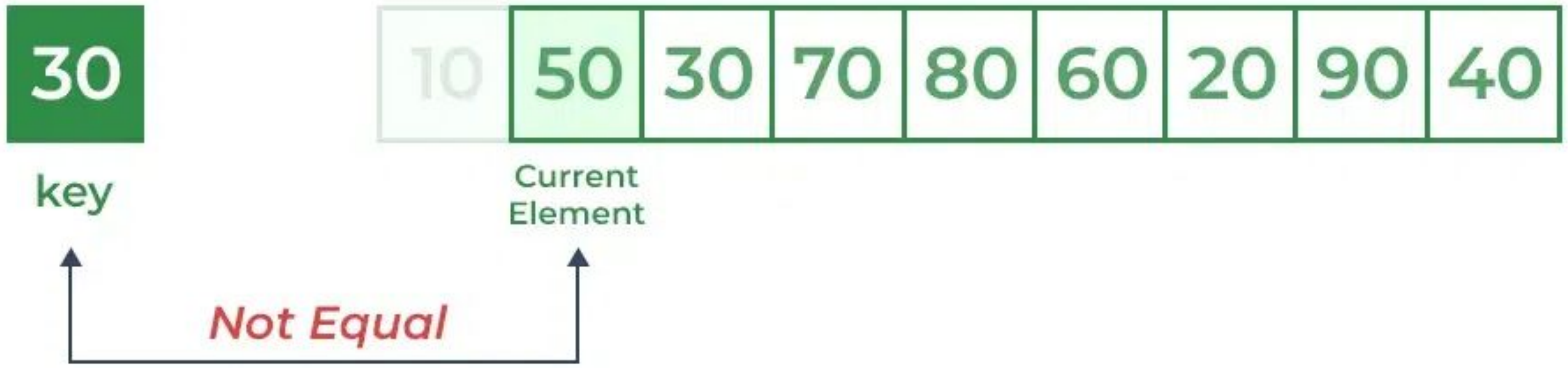
For example: Consider the array **arr[] = {10, 50, 30, 70, 80, 20, 90, 40}** and **key** = 30

Step 1: *Start from the first element (index 0) and compare* ***key*** *with each element (arr[i]).*

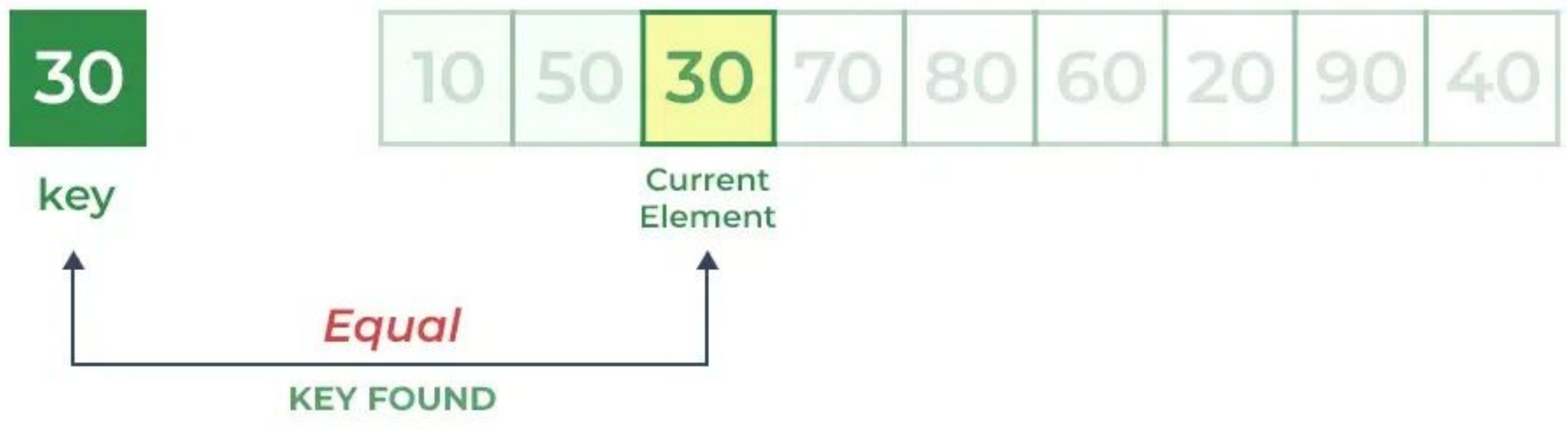
- *Comparing key with first element arr[0]. SInce not equal, the iterator moves to the next element as a potential match.*



- *Comparing key with next element arr[1]. SInce not equal, the iterator moves to the next element as a potential match.*



Step 2: *Now when comparing arr[2] with key, the value matches. So the Linear Search Algorithm will yield a successful message and return the index of the element when key is found (here 2).*



Advantages of Linear Search:

- Linear search can be used irrespective of whether the array is sorted or not. It can be used on arrays of any data type.
- Does not require any additional memory.
- It is a well-suited algorithm for small datasets.

Drawbacks of Linear Search:

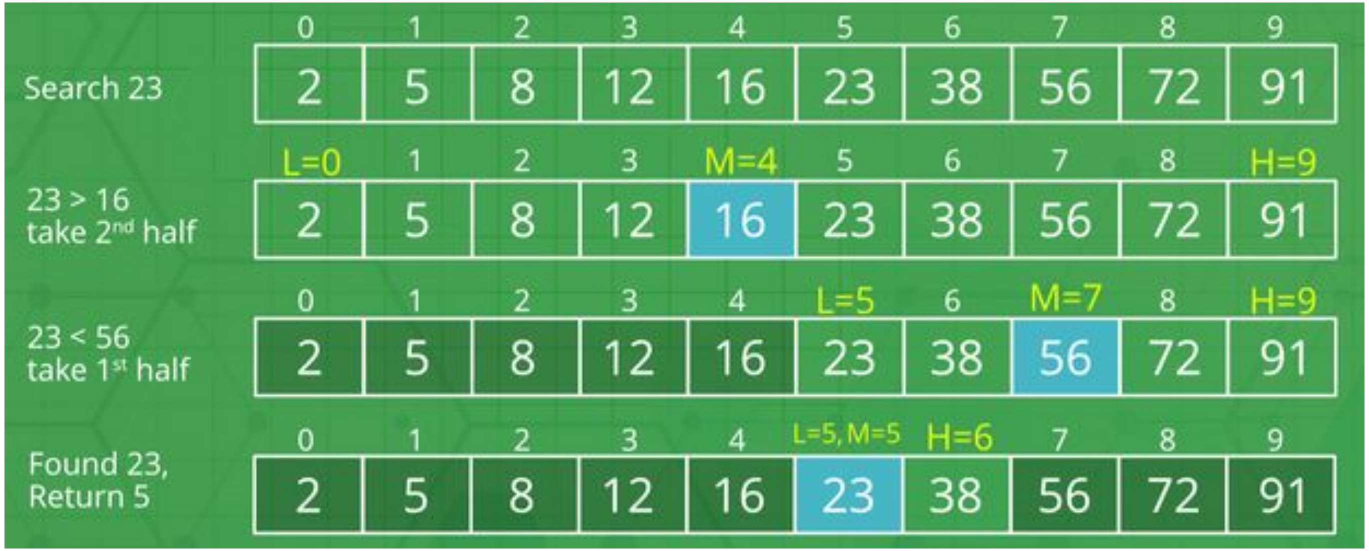
- Linear search has a time complexity of O(N), which in turn makes it slow for large datasets.
- Not suitable for large arrays.

When to use Linear Search?

- When we are dealing with a small dataset.
- When you are searching for a dataset stored in contiguous memory.

Binary Search

Binary Search *is defined as a searching algorithm used in a sorted array by* ***repeatedly dividing the search interval in half****. The idea of binary search is to use the information that the array is sorted and reduce the time complexity to O(log N).*



Conditions for when to apply Binary Search in a Data Structure:

To apply Binary Search algorithm:

- The data structure must be sorted.
- Access to any element of the data structure takes constant time.

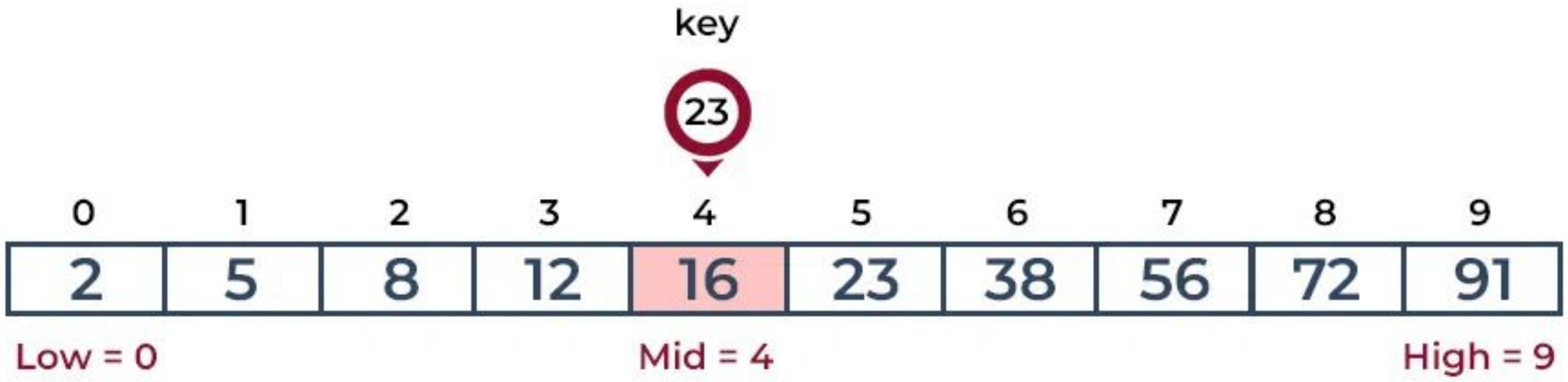
How does Binary Search work?

To understand the working of binary search, consider the following illustration:

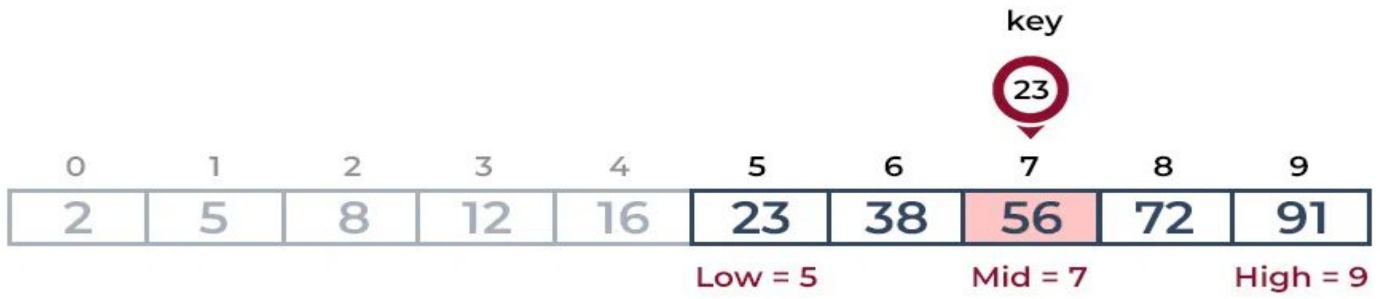
Consider an array **arr[] = {2, 5, 8, 12, 16, 23, 38, 56, 72, 91}**, and the **target = 23**.

First Step: Calculate the mid and compare the mid element with the key. If the key is less than mid element, move to left and if it is greater than the mid then move search space to the right.

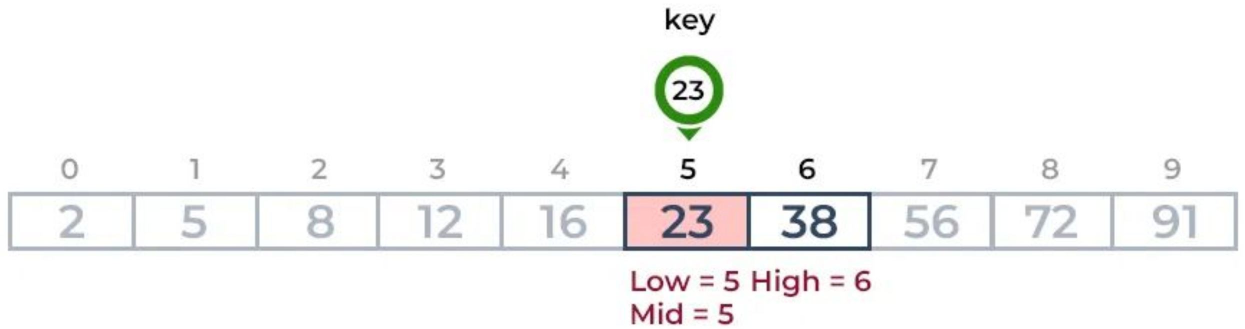
- Key (i.e., 23) is greater than current mid element (i.e., 16). The search space moves to the right.



- *Key is less than the current mid 56. The search space moves to the left.*



Second Step: If the key matches the value of the mid element, the element is found and stop search.



How to Implement Binary Search?

The **Binary Search Algorithm** can be implemented in the following two ways

- Iterative Binary Search Algorithm
- Recursive Binary Search Algorithm

Applications of Binary Search:

- Binary search can be used as a building block for more complex algorithms used in machine learning, such as algorithms for training neural networks or finding the optimal hyperparameters for a model.
- It can be used for searching in computer graphics such as algorithms for ray tracing or texture mapping.
- It can be used for searching a database.

Advantages of Binary Search:

- Binary search is faster than linear search, especially for large arrays.
- More efficient than other searching algorithms with a similar time complexity, such as interpolation search or exponential search.
- Binary search is well-suited for searching large datasets that are stored in external memory, such as on a hard drive or in the cloud.

Drawbacks of Binary Search:

- The array should be sorted.
- Binary search requires that the data structure being searched be stored in contiguous memory locations.
- Binary search requires that the elements of the array be comparable, meaning that they must be able to be ordered.