Algorithms basics

What is an Algorithm?

An **algorithm** is a finite sequence of well-defined instructions that can be used to solve a computational problem. It provides a step-by-step procedure that convert an input into a desired output.

Algorithms typically follow a logical structure:

- **Input:** The algorithm receives input data.
- **Processing:** The algorithm performs a series of operations on the input data.
- **Output:** The algorithm produces the desired output.

What is the Need for Algorithms?

Algorithms are essential for solving complex computational problems efficiently and effectively. They provide a systematic approach to:

- **Solving problems:** Algorithms break down problems into smaller, manageable steps.
- **Optimizing solutions:** Algorithms find the best or near-optimal solutions to problems.
- **Automating tasks:** Algorithms can automate repetitive or complex tasks, saving time and effort.

Algorithms characteristics

- Inputs and output
  - What does algorithm accept, and what are the result?
- Algorithm complexity
  - Space complexity - How much memory does it require?
  - Time complexity – How much time does it require to complete?

Big-O notation

Big O notation is a way to characterize the time or resources needed to solve a computing problem. It's particularly useful in comparing various computing algorithms and approaches under consideration, such as those used in Machine Learning.

Below is a table summarizing Big O functions. The four most commonly referenced and important to remember are:

- **O(1)** - *Constant* access time such as the use of a hash table.
- **O(log n)** - *Logarithmic* access time such as a binary search of a sorted table.
- **O(n)** - *Linear* access time such as the search of an unsorted list.
- **O(n^2)** - Nested iterations.
- **O(n log(n))** - *Multiple* of log(n) access time such as using Quicksort or Merge sort.

Big-O notation table

Notation	Type	Examples	Description
O(1)	Constant	Hash table access	Remains constant regardless of the size of the data set
O(log n)	Logarithmic	Binary search of a sorted table	Increases by a constant. If n doubles, the time to perform increases by a constant, smaller than n amount
O(< n)	Sublinear	Search using parallel processing	Performs at less than linear and more than logarithmic levels
O(n)	Linear	Finding an item in an unsorted list	Increases in proportion to n. If n doubles, the time to perform doubles
O(n log(n))	n log(n)	Quicksort, Merge Sort	Increases at a multiple of a constant
$O(n^2)$	Quadratic	Bubble sort	Increases in proportion to the product of n*n
$O(c^n)$	Exponential	Travelling salesman problem solved using dynamic programming	Increases based on the exponent n of a constant c
O(n!)	Factorial	Travelling salesman problem solved using brute force	Increases in proportion to the product of all numbers included (e.g., 1*2*3*4...)

Search Algorithms – Linear

Linear or Sequential Search

This algorithm works by sequentially iterating through the whole array or list from one end until the target element is found. If the element is found, it returns its index, else -1.

Now let's look at an example and try to understand how it works:

```
arr = [2, 12, 15, 11, 7, 19, 45]
```

Suppose the target element we want to search is `7`.

Approach for Linear or Sequential Search

- Start with index 0 and compare each element with the target
- If the target is found to be equal to the element, return its index
- If the target is not found, return -1

Search Algorithms – Binary Search

This type of searching algorithm is used to find the position of a specific value contained **in a sorted array**. The binary search algorithm works on the principle of divide and conquer and it is considered the best searching algorithm because it's faster to run.

Now let's take a sorted array as an example and try to understand how it works:

```
arr = [2, 12, 15, 17, 27, 29, 45]
```

Suppose the target element to be searched is `17`.

Approach for Binary Search

- Compare the target element with the middle element of the array.
- If the target element is greater than the middle element, then the search continues in the right half.
- Else if the target element is less than the middle value, the search continues in the left half.
- This process is repeated until the middle element is equal to the target element, or the target element is not in the array
- If the target element is found, its index is returned, else -1 is returned.

Sort Algorithms – Bubble

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 are quite high.

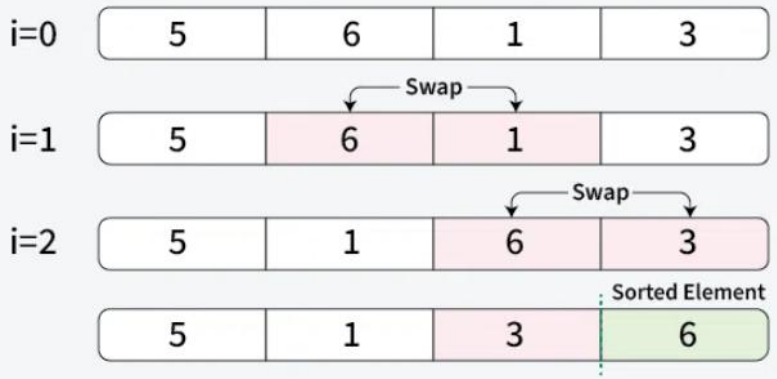
•We sort the array using multiple passes. After the first pass, the maximum element goes to end (its correct position). Same way, after second pass, the second largest element goes to second last position and so on.

•In every pass, we process only those elements that have already not moved to correct position. After k passes, the largest k elements must have been moved to the last k positions.

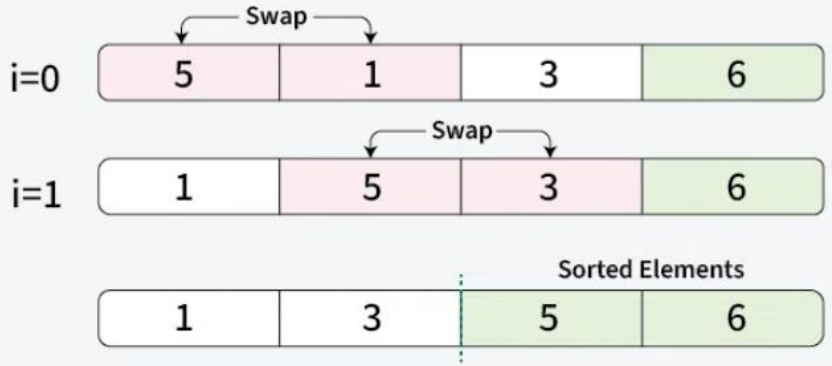
•In a pass, we consider remaining elements and compare all adjacent and swap if larger element is before a smaller element. If we keep doing this, we get the largest (among the remaining elements) at its correct position.

Sort Algorithms – Bubble Explanation

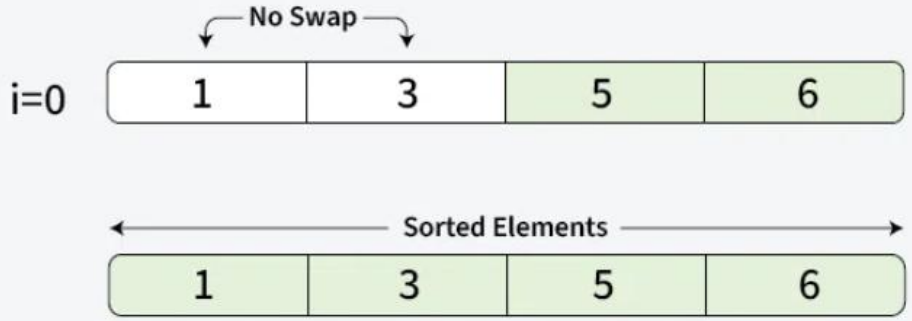
01 Step | Placing the 1st largest element at its correct position



02 Step | Placing 2nd largest element at its correct position

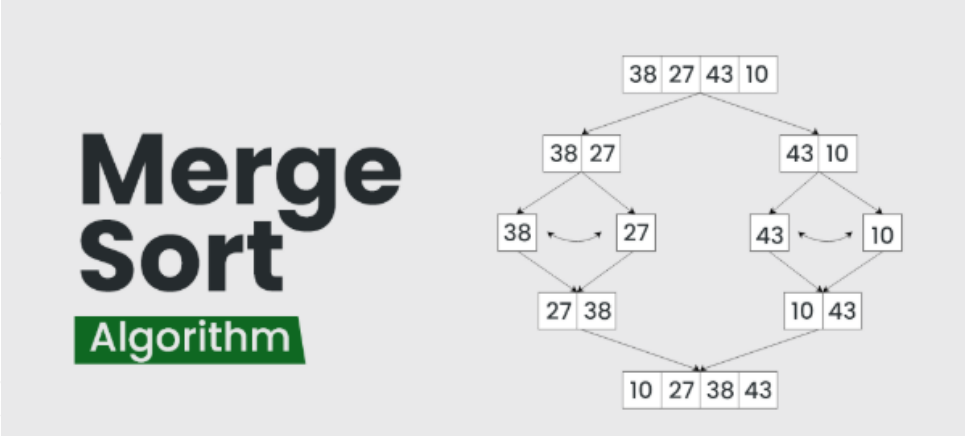


03 Step | Placing 3rd largest element at its correct position



Sort Algorithms – Merge

Merge sort is a popular sorting algorithm known for its efficiency and stability. It follows the divide-and-conquer approach. It works by recursively dividing the input array into two halves, recursively sorting the two halves and finally merging them back together to obtain the sorted array.



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

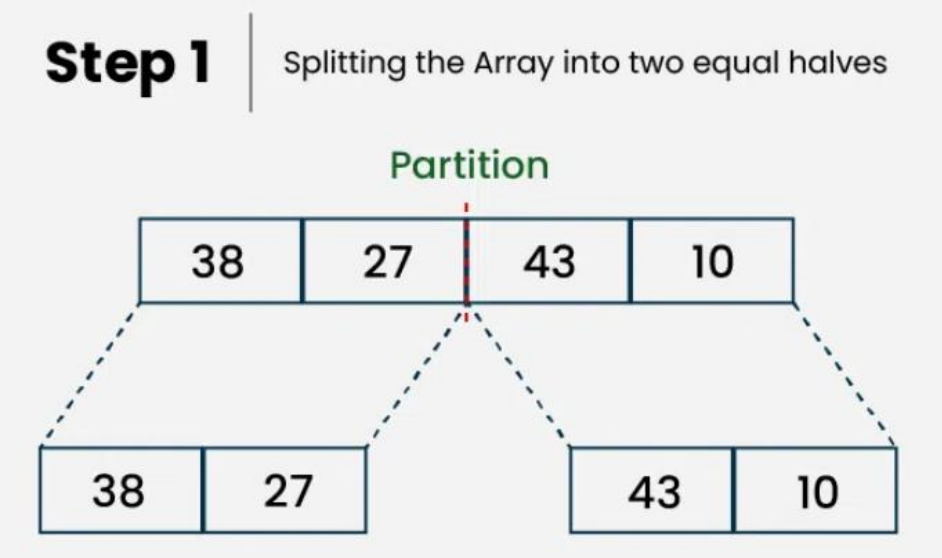
1.Divide: *Divide the list or array recursively into two halves until it can no more be divided.*

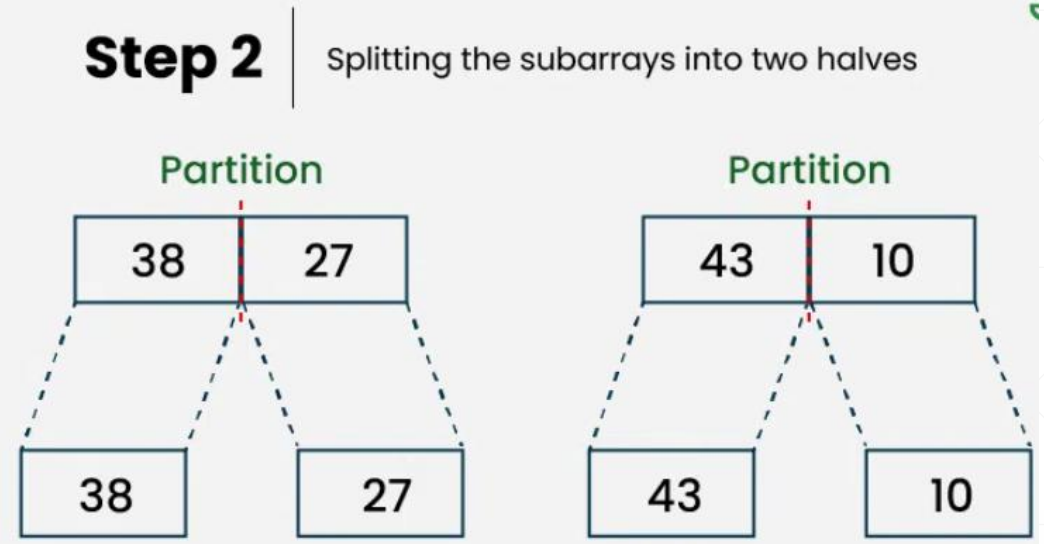
2.Conquer: *Each subarray is sorted individually using the merge sort algorithm.*

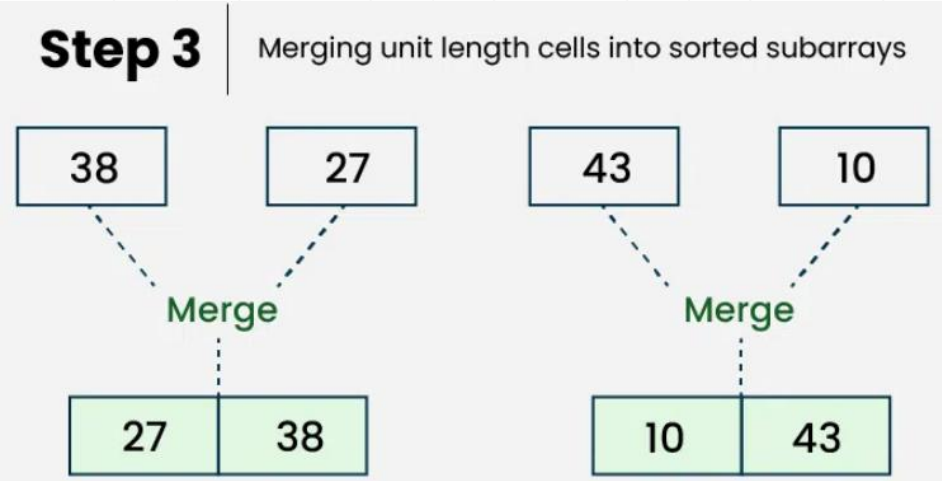
3.Merge: *The sorted subarrays are merged back together in sorted order. The process continues until all elements from both subarrays have been merged.*

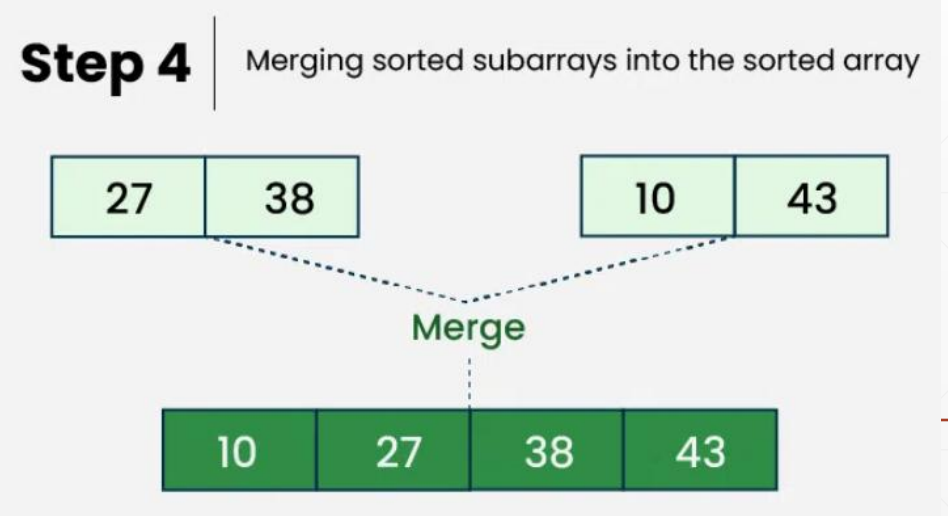
Sort Algorithms – Merge Explanation

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









Sort Algorithms – Quick

QuickSort is a sorting algorithm based on the Divide and Conquer 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.

It works on the principle of **divide and conquer**, breaking down the problem into smaller sub-problems.

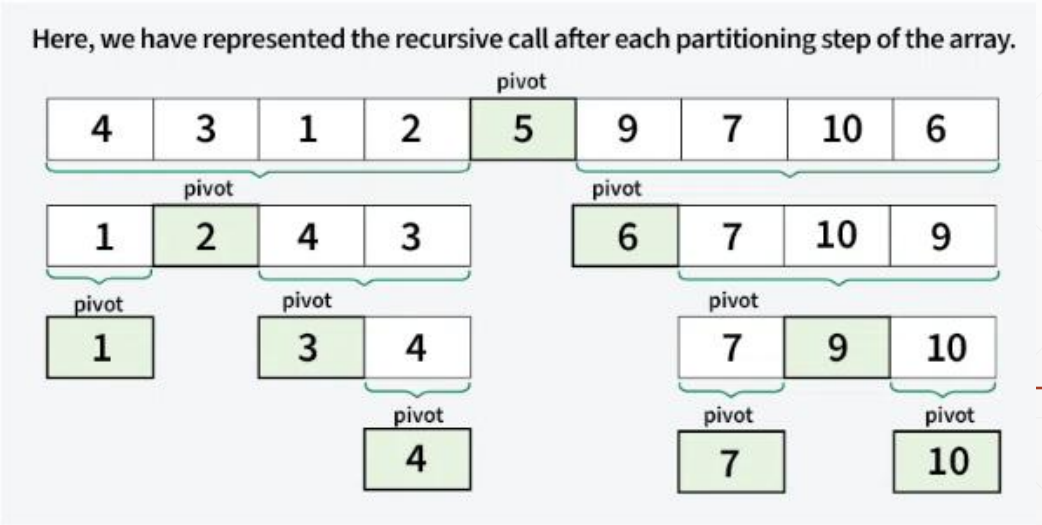
There are mainly three steps in the algorithm:

1.Choose a Pivot: Select an element from the array as the pivot. The choice of pivot can vary (e.g., first element, last element, random element, or median).

2.Partition the Array: Rearrange the array around the pivot. After partitioning, all elements smaller than the pivot will be on its left, and all elements greater than the pivot will be on its right. The pivot is then in its correct position, and we obtain the index of the pivot.

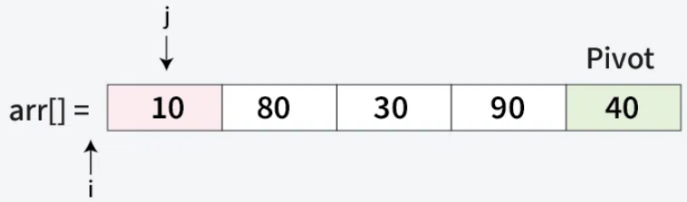
3.Recursively Call: Recursively apply the same process to the two partitioned sub-arrays (left and right of the pivot).

4.Base Case: The recursion stops when there is only one element left in the sub-array, as a single element is already sorted.



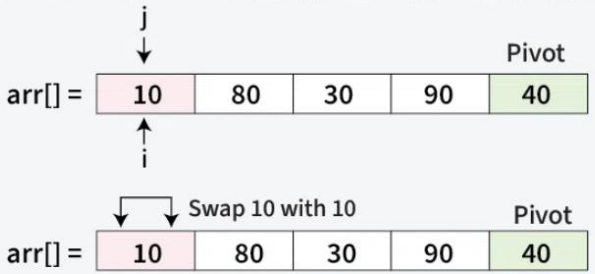
Sort Algorithms – Quick – Explanation

01 Step Pivot Selection: The last element arr[4] = 40 is chosen as the pivot.
Initial Pointers: i = -1 and j = 0.

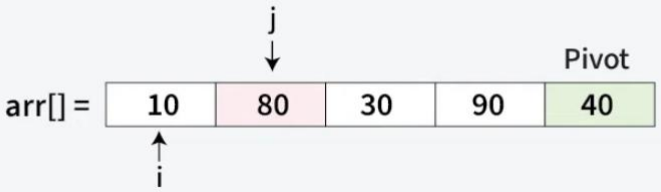


Quick sort

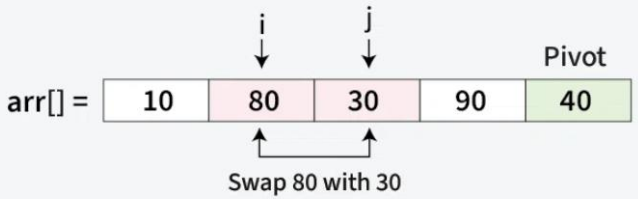
02 Step Since, arr[j] < pivot (10<40)
Increment i to 0 and swap arr[i] with arr[j]. Increment j by 1



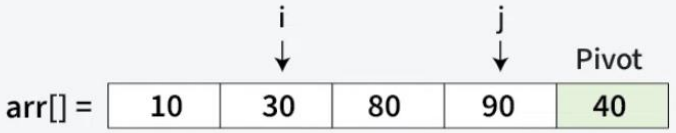
03 Step Since, arr[j] > pivot (80>40)
No swap needed. Increment j by 1



04 Step Since, arr[j] < pivot (30<40)
Increment i by 1 and swap arr[i] with arr[j]. Increment j by 1



05 Step Since, arr[j] > pivot (90>40)
No swap needed. Increment j by 1



06 Step Since traversal of j has ended. Now move pivot to its correct position, Swap arr[i + 1] = arr[2] with arr[4] = 40.