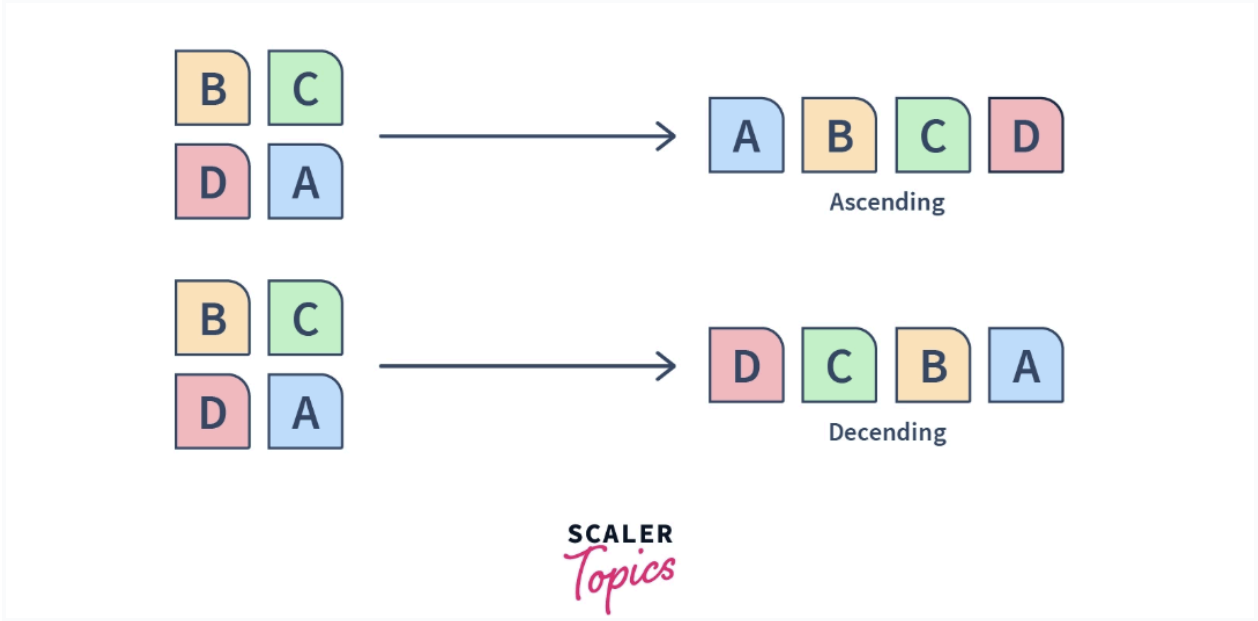
# Sorting

Sorting refers to the process of arranging a list or a sequence of given elements(that can be numbers or strings, etc.) or data into any particular order, which may be ascending or descending. Sorting is basically useful whenever we want our data to be in an ordered form.




Since sorting is often **capable of reducing the algorithmic complexity** of some particular types of problems, it has found its usage in a wide range of fields. Some of them are discussed below.

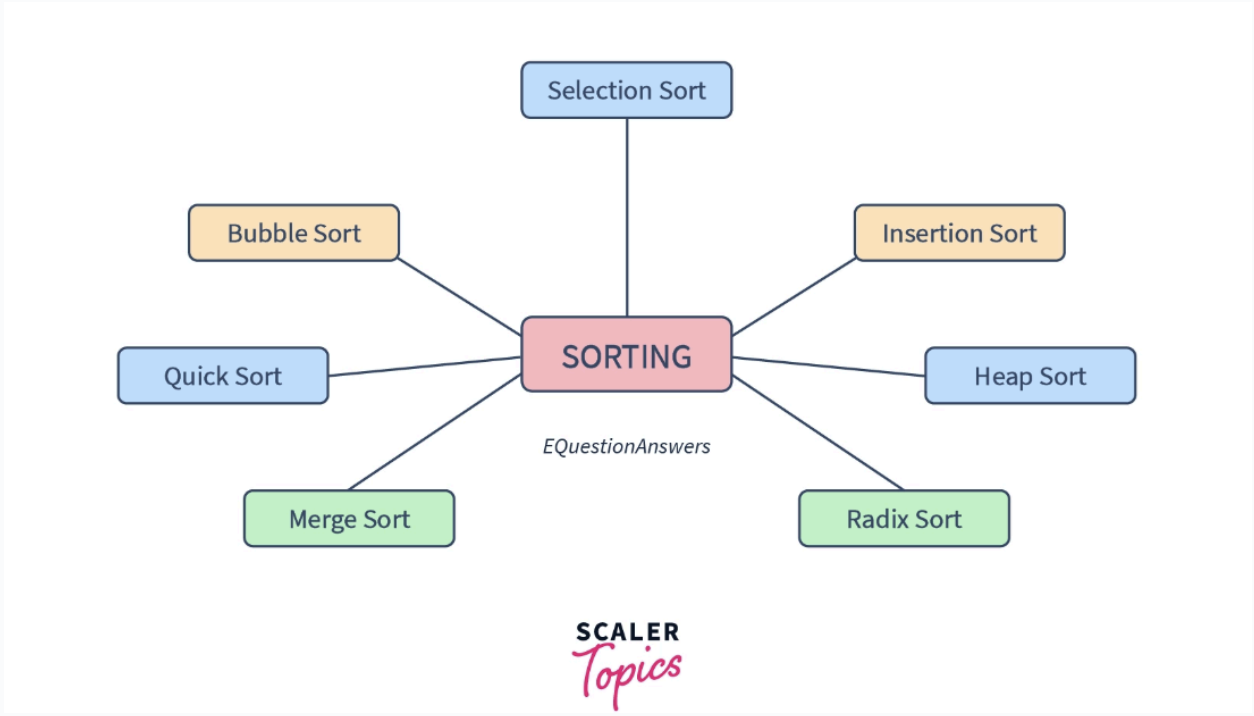
- **Dictionary**: The dictionary stores words in alphabetical order so that searching for any word becomes easy.
- **Telephone Book**: The telephone book stores the telephone numbers of people sorted by their names in alphabetical order, so that the names can be searched easily.
- **E-commerce application**: While shopping through any e-commerce application like amazon or flipkart, etc. , we preferably tend to sort our items based on their price range, popularity, size, etc.

Also, the sorting algorithm's real-life usage is just miraculous! For example --

- **Quick sort** is used for updating the real-time scores of various sports.
- **Bubble sort** is used in TV to sort the channels based on audience viewing timings.

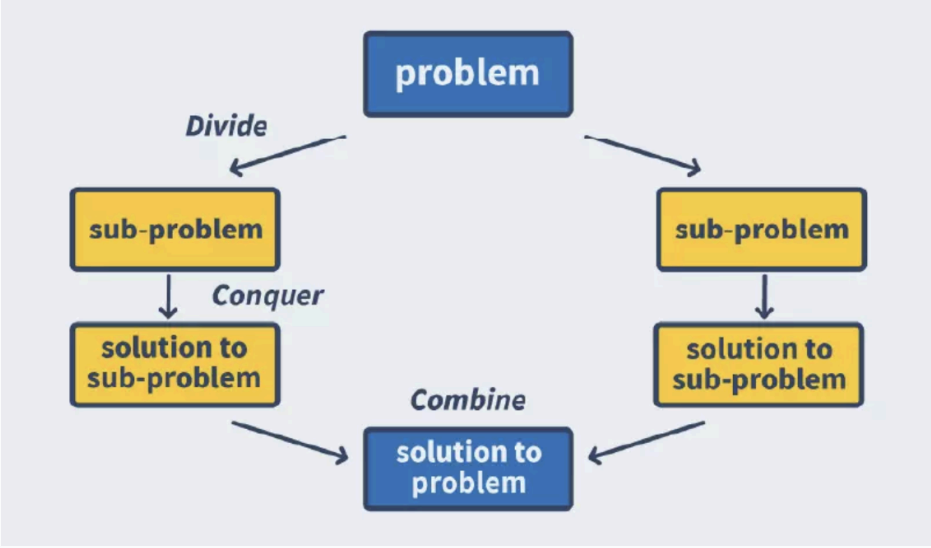
- **Merge Sort** is very popularly used by databases to load a huge amount of data.

## Sorting Algorithms




### Merge Sort

Merge Sort is a **Divide and Conquer algorithm**. It divides the input array into two halves, calls itself for the two halves, and then merges the two sorted halves.



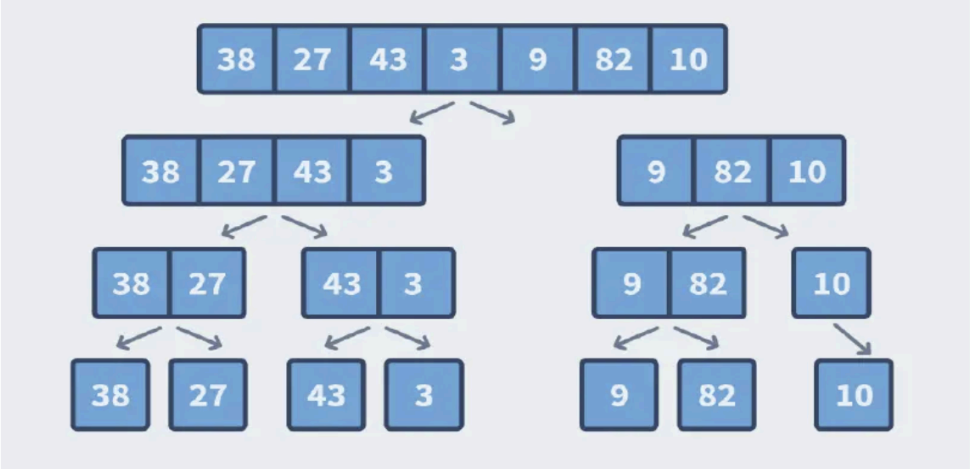
## How does the Merge Sort Algorithm work?

### Divide and Conquer Strategy

Let us consider an array = {38,27,43,3,9,82,10}.
1. Divide - In this step, we find the midpoint of the given array by using the formula mid=start+(end-start)/2.

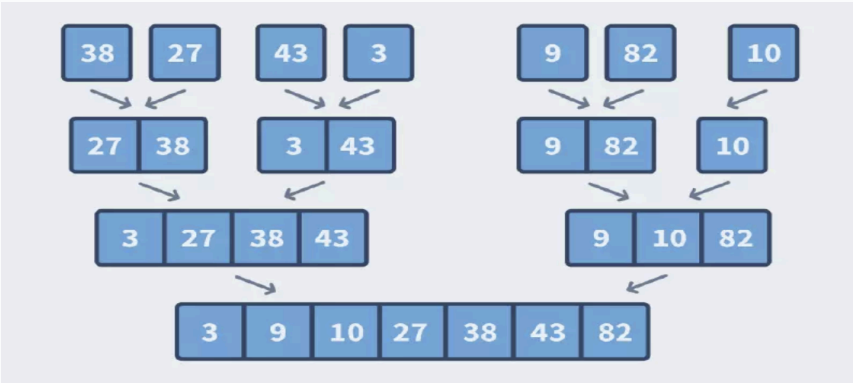
2. Conquer - In this step, we divide the array into subarrays using the midpoint calculated. We recursively keep dividing the array and keep calculating the midpoint for doing the same. It is important to note that a single array element is always sorted.

So, our aim is to continuously divide until all elements in the array are single array elements. Once that is done, the array elements are combined to form a sorted array.




Our goal is to keep dividing till all elements in the array are single array elements hence, this is the base case i.e. when there are n subarrays of the original array that consisted of n integers. It is now the turn to sort them and combine them.

3. **Combine**
Now that all our subarrays are formed, it is now time to combine them in sorted order.




## Question Time!

**Q:** Given an integer array count the number of inversion pairs in the array.
Note: Inversion Pair (i,j) are elements such that `Arr[i]>Arr[j]` **and** `i<j`

**Solution**

**Brute force approach:**
To count the number of inversion pairs in an integer array using a brute force approach, you can follow these steps:

- Iterate through each element in the array.
- For each element, compare it with all the subsequent elements in the array.
- If the current element is greater than any of the subsequent elements, increment the count variable by 1.
- Continue the iteration until all pairs have been compared.
- The time complexity of this brute force solution is O(n^2), where n is the size of the array, as it involves nested iterations to compare each pair of elements.

**Optimized Approach:** To count the number of inversion pairs in an integer array, we can use the merge sort algorithm. The basic idea is to divide the array into two halves recursively, count the inversion pairs in each half, and then merge the two halves while counting additional inversion pairs that arise during the merging process.

Here's the code intuition for counting inversion pairs using merge sort:

- Divide the array into two halves.
- Recursively count the inversion pairs in the left and right halves.

- Merge the two sorted halves while counting inversion pairs.

During the merge step, whenever we take an element from the right half and place it before an element from the left half, we know that we have found inversion pairs. We can count the number of inversion pairs by adding the number of remaining elements in the left half.

**Pseudo Code Implementation:**

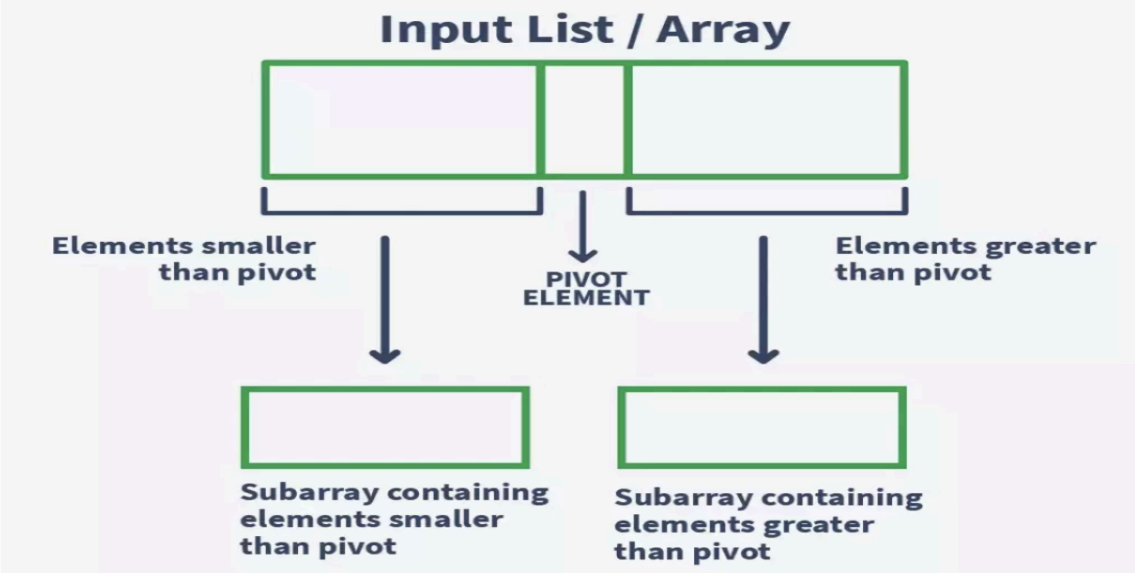
```
private int Mod = 1000 * 1000 * 1000 + 7;
public int solve(int[] A) {
    return (int) mergeSort(A);
}
public long mergeSort(int[] A) {
    int[] temp = new int[A.length];
    return _mergeSort(A, temp, 0, A.length - 1) % Mod;
}
public long _mergeSort(int arr[], int temp[], int left, int right) {
    int mid;
    long inv_count = 0;
    if (right > left) {

        mid = (right + left) / 2;
        inv_count += _mergeSort(arr, temp, left, mid);
        inv_count += _mergeSort(arr, temp, mid + 1, right);
        inv_count += merge(arr, temp, left, mid + 1, right);
    }
    return inv_count % Mod;
}
```

```
public long merge(int arr[], int temp[], int left, int mid, int right) {
    int i, j, k;
    long inv_count = 0;
    i = left;
    j = mid;
    k = left;
    while ((i <= mid - 1) && (j <= right)) {
        if (arr[i] <= arr[j])
            temp[k++] = arr[i++];
        else {
            temp[k++] = arr[j++];
            inv_count = inv_count + (mid - i);
        }
    }
    while (i <= mid - 1)
        temp[k++] = arr[i++];
    while (j <= right)
        temp[k++] = arr[j++];
    for (i = left; i <= right; i++)
        arr[i] = temp[i];
    return inv_count % Mod;
}
```

## Quick Sort

Quick sort, also known as partition-exchange sort, is an in-place sorting algorithm. It is a divide-and-conquer algorithm that works on the idea of selecting a pivot element and dividing the array into two subarrays around that pivot.
In quick sort, after selecting the pivot element, the array is split into two subarrays. One subarray contains elements smaller than the pivot element, and the other subarray contains elements greater than the pivot element.




## PseudoCode:-

```
function quickSort(arr, low, high):
      if low < high:
            pivotIndex = partition(arr, low, high)
            quickSort(arr, low, pivotIndex - 1)
            quickSort(arr, pivotIndex + 1, high)

function partition(arr, low, high):
      pivot = arr[high]
      i = low - 1
      for j = low to high - 1:
            if arr[j] <= pivot:
                  i = i + 1
                  swap(arr[i], arr[j])

      swap(arr[i + 1], arr[high])
      return i + 1
```

## Count Sort

Counting sort is a sorting algorithm that works on an integer array with a specific range of values. It is an efficient algorithm with a time complexity of O(N + K), where N is the number of elements in the input array and K is the range of values.

### Key Points about Counting Sort

- Counting sort assumes that the input elements are integers and have a known range.
- It creates a count array to store the frequency of each element in the input array.
- The count array is initialized with all zeros.
- The algorithm then traverses the input array and increments the count of each element in the count array.
- After counting the frequencies, the count array is modified to store the cumulative sum of counts.
- The cumulative sum array determines the correct position of each element in the sorted output array.
- Finally, the algorithm traverses the input array again and places each element in its correct position in the output array using the count array.
- The sorted array is obtained as the output.

Pseudocode -

```
private static void countingSort(int[] arr) {
        int n = arr.length;
        int max = maximum(arr);
        int[] count = new int[max + 1];
        for (int i = 0; i < n; ++i) {
            count[arr[i]]++;
        }
        int lastIndex = 0;
        for (int i = 0; i <= max; ++i) {
            while (count[i]-- > 0) {
                arr[lastIndex++] = i;
            }
        }
}
```

## Selection Sort

- Selection Sort is a simple comparison-based sorting algorithm.
- It repeatedly finds the minimum element from the unsorted portion and places it at the beginning, for sorting in ascending order.
- The process is repeated until the entire array is sorted.
- **Time complexity:** O(n^2), where n is the number of elements.
- **Space complexity:** O(1), as it requires a constant amount of additional space.

## Insertion Sort

- Insertion Sort is a simple comparison-based sorting algorithm.
- It iterates through the array, comparing each element with its previous elements and inserting it at the correct position.
- Time complexity: O(n^2) in the worst case and O(n) in the best case when the array is already sorted.
- Space complexity: O(1), as it operates on the input array itself without requiring additional space.

## Radix Sort

- Radix Sort is a non-comparison-based algorithm that sorts the ele by their digits.
- It works by sorting the elements based on each digit, starting from the least significant digit to the most significant digit.
- It uses a stable sorting algorithm (typically Count Sort or Bucket Sort) as a subroutine for each digit.
- Time complexity: O(d * (n + k)), where d is the number of digits, n is the number of elements, and k is the range of digits (typically 10).
- Space complexity: O(n + k), as it requires additional space for the count array

  **Revision Video** - https://drive.google.com/file/d/10Bfov3--bJ_xl7AO5s0vX5DMqcDBUXHh/view?usp=share_link

  https://drive.google.com/file/d/1oC4QG_Zie3OG_kFNpbU8_RW1-BV3RsNt/view?usp=share_link