sadā śiva samāramabhāṃ śaṅkarācārya madhyamām..
asmadācārya paryantāṃ vande guru paramparām..

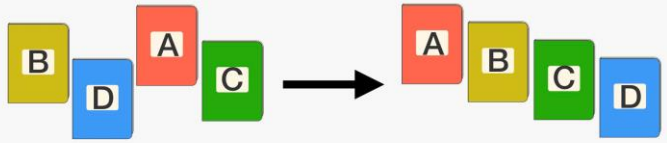
*Salutation to the lineage starting with lord **Sadasiva**, with **Adi Sankara** in the middle and continuing up to my immediate teacher.*

*సదాశివుడు మొదలుకొని మధ్యలో ఆదిశంకరునితో మొదలై నా తక్షణ గురువు వరకు కొనసాగే వంశానికి వందనం.*

# DATA STRUCTURES ALGORITHMS

## SORTING

# Sorting Algorithms




- Insertion Sort
- Bubble Sort
- Merge Sort

# Insertion Sort

Insertion Sort is a simple, yet powerful algorithm for sorting data. The algorithm works by taking one element at a time from an unsorted list and inserting it into a sorted list in the correct position. In this article, we will learn how the Insertion Sort algorithm works and its implementation in Python. Refer the below-animated visualization of insertion sort:

## How Insertion Sort Works:

The Insertion Sort algorithm works as follows:

1. Take an unsorted list of n elements.
2. Pick the first element and insert it into a sorted list.
3. Take the next element and insert it into the sorted list in the correct position.
4. Repeat step 3 until all elements have been inserted into the sorted list.

## Step-by-step Explanation of Insertion sort:

Let’s take a closer look at how the algorithm works with an example:

- Suppose we have an unsorted list [5, 2, 4, 6, 1, 3].
- We start by picking the first element, 5, and inserting it into a sorted list [5].
- We then take the second element, 2, and compare it with 5. Since 2 is smaller than 5, we swap them, and the sorted list becomes [2, 5].
- We then take the third element, 4, and insert it into the sorted list.
- We compare 4 with 5 and swap them, giving us [2, 4, 5].
- We continue this process until we have a sorted list of all elements.

```python
def insertionSort(arrayToSort):
    """
    Complexity:
        best    :   O(n)
        average :   O(n^2)
        worst   :   O(n^2)
    parameters:
        arrayToSort: Array to be sorted

    returns:
        sorted array
    """
    for i in range(1, len(arrayToSort)):
        j = i
        while j > 0 and arrayToSort[j] < arrayToSort[j - 1]:
            arrayToSort[j], arrayToSort[j - 1] = arrayToSort[j - 1], arrayToSort[j]
            j -= 1
    return arrayToSort


if __name__ == "__main__":
    array = [2, 5, 1, 5, 8, 9, 0, 10]
    print(insertionSort(array))
```

## Conclusion:

Insertion Sort is a simple and efficient algorithm for sorting data. It works by inserting each element into a sorted list in the correct position. While Insertion Sort is less efficient than Merge Sort for large datasets, it can be more efficient than Bubble Sort for small datasets. It is important to choose the appropriate sorting algorithm depending on the size of the dataset and the specific use case

# What is Merge Sort?

Merge sort is based on a divide-and-conquer principle. This works by breaking down the array into smaller subarrays, sorting those subarrays, and then merging them back together. Once you will follow the below example with python code, it will be more precise.

6 5 3 1 8 7 2 4

Merge sort is commonly used for sorting large datasets, as it is efficient and fast, even for very large arrays. It’s also useful for parallel sorting, where multiple processors can be used to sort the data.

Python code to implement merge sort in Python. The time complexity of merge sort is O(n*log n) where n is the number of elements in the array.

```
def mergeSort(array, asc=True):
    '''
    Time complexity : O(nlog(n)))
    parameters:
        array   : Array of numbers to be sorted
        asc     : boolean flag to sort ascending
                  descending. Default = True (ascending)
    '''
    midIdx = len(array) // 2

    if len(array) == 1:
        return array

    leftArr = array[:midIdx]
    rightArr = array[midIdx:]
    # Double recursion
    # 1. To keep on dividing the entire
    #    in half until only one item is left
    #    in both left and right side arrays
    # 2. Each time those sub-arrays will be passed
    #    in the other recursive function, whihc will
    #    return the sorted sub array.
    return sortArray(mergeSort(leftArr, asc), mergeSort(rightArr, asc), asc)
```

How merge sort can be used to sort an array of integers in Python. The algorithm divides the array in half, sorts each half recursively, and then merges the sorted halves back together

# Is Merge Sort Faster Than Quicksort?

The time complexity of merge sort is O(n*log n), where n is the number of elements in the array. This means that the time required to sort the array increases logarithmically with the number of elements. Both merge sort and quicksort are efficient sorting algorithms, but they have different strengths and weaknesses. Merge sort is generally considered to be more efficient for sorting large datasets, while quicksort is better for small to medium-sized datasets.

# Why is Merge Sort More Effective?

Merge sort has several advantages over other sorting algorithms, such as:

- It is highly scalable and can handle large datasets.
- It is a stable sorting algorithm that maintains the relative order of equal elements.
- It is easy to implement and understand.

## Conclusion:

Merge sort is a powerful and efficient sorting algorithm widely used in computer science. Now you know everything about the merge sort algorithm such as `how merge sort works` and `how to implement it in Python`

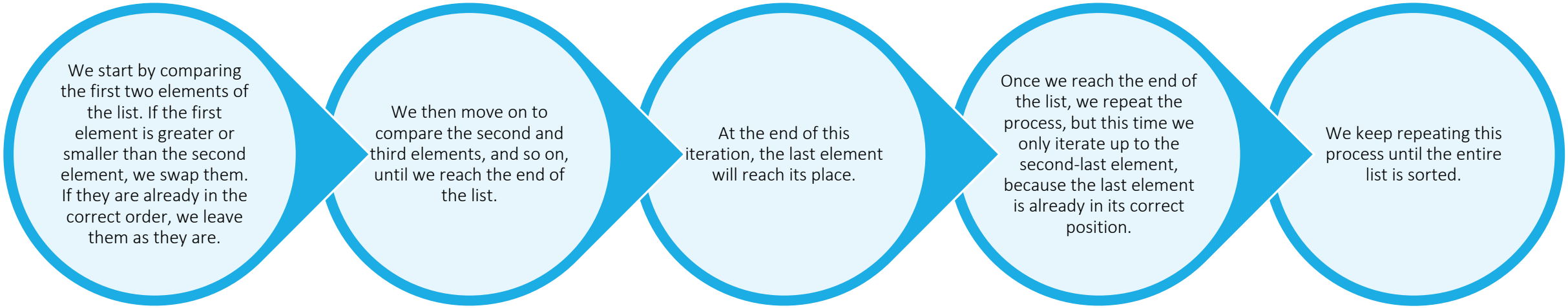
# What is Bubble Sort?

Bubble sort is a simple sorting algorithm that works by repeatedly swapping adjacent elements if they are in the wrong order. You swap them according to the order in which you want to sort the list. It is named bubble sort as in the process of sorting, every element will keep on popping and swapping to each other like bubbles

## Note:

Although **Bubble sort** is very easy to understand and implement, it is not the most efficient sorting algorithm for large data sets. The time complexity of Bubble sort is `O(n^2)` in both the worst and average cases. Merge sort is a suitable and efficient algorithm for large data sets. The time complexity of merge sort is `O(n * log n)` which is way better than Bubble Sort.

# How Bubble Sort works?



1. We start by comparing the first two elements of the list. If the first element is greater or smaller than the second element, we swap them. If they are already in the correct order, we leave them as they are.
2. We then move on to compare the second and third elements, and so on, until we reach the end of the list.
3. At the end of this iteration, the last element will reach its place.
4. Once we reach the end of the list, we repeat the process, but this time we only iterate up to the second-last element, because the last element is already in its correct position.
5. We keep repeating this process until the entire list is sorted.

6 5 3 1 8 7 2 4

## Implementation of Bubble sort in Python – Worst

Following is the simplest implementation of Bubble sort where time complexity in all the cases, best and worst, would be O(n^2). Even if the provided array is already sorted, it will still take the same time to return the sorted array.

Can we improve it?

```
def bubbleSort(arrayToSort):
    """
    Time complexity:
        best    : O(n^2)
        average : O(n^2)
        worst   : O(n^2)
    parameters:
        arrayToSort : Array to be sorted
    returns:
        Sorted array
    """
    size = len(arrayToSort)
    for i in range(size):
        for j in range(0, size - i - 1):
            if arrayToSort[j] > arrayToSort[j + 1]:
                arrayToSort[j], arrayToSort[j + 1] = arrayToSort[j + 1],
arrayToSort[j]
            print(i, j)
    return arrayToSort
```

## Implementation of Bubble sort in Python – Better

In the below implementation, the best time complexity can be O(n). Although the worst or average time complexity in the below implementation would also be the same O(n^2).

```
def bubbleSort(arrayToSort):
    """
    Time complexity:
        best    : O(n) - when the array is already sorted
        average : O(n^2)
        worst   : O(n^2)
    parameters:
        arrayToSort : Array to be sorted
    returns:
        Sorted array
    """
    iCount = 0
    isSorted = False
    size = len(arrayToSort)
    while not isSorted:
        isSorted = True
        for i in range(0, size - iCount - 1):
            if arrayToSort[i] > arrayToSort[i + 1]:
                arrayToSort[i], arrayToSort[i + 1] = arrayToSort[i + 1],
arrayToSort[i]
                isSorted = False
            print(i)
        iCount += 1
    return arrayToSort
```

## Leet code Sorting Problems

https://leetcode.com/tag/sorting/