Sorting Algorithms

Objectives:

- Understand the implement Bubble Sort, Insertion Sort and Quick Sort algorithms
- Able to use examples to explain these algorithms

1. Introduction

Sorting refers to arranging a *fixed* set of data in a particular *order*. Sorting orders could be *numerical* (1, 2, 3, ...), *lexicographical* ('AA', 'AB', 'AC', ...) or custom ('Mon', 'Tue', 'Wed', ...).

Sorting algorithms specify ways to arrange data in particular ways to put the data in order.

Comparing of Algorithms

Sorting algorithms could be compared based on

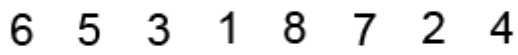
- **Time taken** to complete the work
- **Memory requirement** to complete the work
- **Stability** in keeping the input orders of items having the same sorting order

2. Bubble Sort

This algorithm iterates over a list multiple time.

- In each iteration, it takes 2 consecutive elements and compare them.
- It swaps them to move smaller value to the left and larger value to the right.
- It repeats until the larger elements "bubble up" to the end of the list, and the smaller elements moves to the "bottom".

The right-hand side of the list are sorted.



Implementation

- For 1st iteration, we need to make n-1 comparison. It will bring the largest value to the exteme right.
- For 2nd iteration, we need to make n-2 comparison. It will bring 2nd largest value to the 2nd extreme right.
- And so on...

We need a nested loops to make multiple iterations.

In [1]:

```
def bubble_sort(nums):
    for ceil in range(len(nums)-1,0,-1):
        for idx in range(ceil):
            if nums[idx]>nums[idx+1]:
                nums[idx],nums[idx+1] = nums[idx+1],nums[idx]

```

In [2]:

```
import random
arr = [random.randint(1,20) for i in range(10)]
print(arr)
bubble_sort(arr)
print(arr)
```

```
[3, 11, 6, 12, 18, 17, 18, 18, 20, 14]
[3, 6, 11, 12, 14, 17, 18, 18, 18, 20]
```

Time Complexity

The amount of comparisons are (n - 1) + (n - 2) + ... + 1, which gives Selection Sort a time complexity of O(n^2).

- Best case performance is O(n)
- Worst case performance is O(n^2)

3. Insertion Sort

In Insertion sort, you compare the key element with the previous elements.

- If the previous elements are greater than the key element, then you move the previous element to the next position.
- Usually, start from index 1 of the input array.

The left-hand side of the list are sorted.

Implementation

In [3]:

```
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key_element=arr[i]
        j=i-1
        while j>=0:
            if arr[j]>key_element:
                arr[j+1] = arr[j]
            else:
                break
            j=j-1

        arr[j+1]=key_element

    return arr
```

In [4]:

```
import random
arr = [random.randint(1,20) for i in range(10)]
print(arr)
insertion_sort(arr)
print(arr)
```

```
[14, 2, 17, 19, 19, 20, 19, 15, 6, 6]
[2, 6, 6, 14, 15, 17, 19, 19, 19, 20]
```

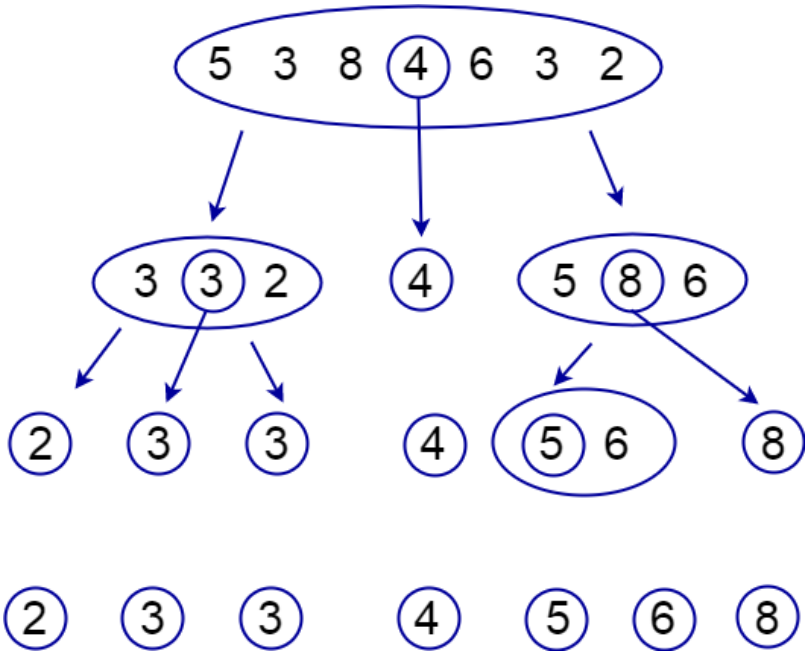
Time Complexity

- The outer for-loop in Insertion Sort function always iterates `n-1` times.
- The inner for-loop will make 1 + 2 + 3 ... n-1 comparison in worst case.
- Thus overall time complexity is O(n^2)

4. Quick Sort

Quicksort first selects a pivot element and partitions the list around the pivot, putting every smaller element into a low array and every larger element into a high array.

- The `pivot` element is selected randomly.
- The first pass partitions data into 3 sub-lists, `lesser` (less than pivot), `equal` (equal to pivot) and `greater` (greater than pivot).
- The process repeats for `lesser` list and `greater` list.



Implementation

In [5]:

```
def quick_sort(arr):
    if not arr:
        return []

    pivot = arr[len(arr)//2]
    lesser = [i for i in arr if i<pivot]
    equal = [i for i in arr if i==pivot]
    greater  = [i for i in arr if i>pivot]

    res = quick_sort(lesser) + equal + quick_sort(greater)
    return res

```

In [6]:

```
import random
arr = [random.randint(1,20) for i in range(10)]
print(arr)
arr = quick_sort(arr)
print(arr)
```

```
[3, 6, 8, 1, 7, 6, 10, 14, 20, 17]
[1, 3, 6, 6, 7, 8, 10, 14, 17, 20]
```

Complexity

The worst case scenario is when the smallest or largest element is always selected as the pivot.

- This would create partitions of size n-1, causing recursive calls n-1 times.
- This leads us to a worst case time complexity of O(n^2).

With a good pivot, the input list is partitioned in linear time, O(n), and this process repeats recursively an average of $log_2 n$ times.

average of $log_2 n$ times.

- This leads to a final complexity of O(n $log_2 n$).

5. Bubble Sort vs. Insertion Sort vs. Quick Sort

In [7]:

```
from IPython.display import YouTubeVideo
YouTubeVideo(id='WaNLJf8xzC4', width=621, height=349)
```

Out[7]: