Searching And Sorting

Searching

Searching means to find out whether a particular element is present in a given sequence or not. There are commonly two types of searching techniques:

- Linear search (We have studied about this in **Arrays and Lists**)
- Binary search

In this module, we will be discussing binary search.

Binary Search

Binary Search is a searching algorithm for finding an element's position in a sorted array. In a nutshell, this search algorithm takes advantage of a collection of elements of being already sorted by ignoring half of the elements after just one comparison.

Prerequisite: Binary search has one pre-requisite; unlike the linear search where elements could be any order, the array in **binary search** must be sorted,

The algorithm works as follows:

1. Let the element we are searching for, in the given array/list is X.
2. Compare X with the middle element in the array.
3. If X matches with the middle element, we return the middle index.
4. If X is greater than the middle element, then X can only lie in the right (greater) half subarray after the middle element, then we apply the algorithm again for the right half.
5. If X is smaller than the middle element, then X must lie in the left (lower) half, this is because the array is sorted. So we apply the algorithm for the left half.

Example Run

- Let us consider the array to be:

- Let x = 4 be the element to be searched.
- Set two pointers `low` and `high` at the first and the last element respectively.
- Find the middle element `mid` of the array ie. `arr[(low+high)/2] = 6`.

- If `x == mid`, then return `mid`. Else, compare the element to be searched with **m**.
- If `x > mid`, compare x with the middle element of the elements on the right side of mid. This is done by setting low to `low = mid + 1`.
- Else, compare x with the middle element of the elements on the left side of mid. This is done by setting high to `high = mid - 1`.

- Repeat these steps until low meets high. We found 4:

Python Code

```
#Function to implement Binary Search Algorithm
def binarySearch(array, x, low, high):

    #Repeat until the pointers low and high meet each other
    while low <= high:
        mid = low + (high - low) #Middle Index
        if array[mid] == x: #Element Found
            return mid
        elif array[mid] < x: #x is on the right side
            low = mid + 1
        else: #x is on the left side
            high = mid - 1
    return -1  #Element is not found

array = [3, 4, 5, 6, 7, 8, 9]
x = 4

result = binarySearch(array, x, 0, len(array)-1)

if result != -1: #If element is found
    print("Element is present at index " + str(result))
else: #If element is not found
    print("Not found")
```

We will get the **output** of the above code as:

```
Element is present at index 1
```

Advantages of Binary search:

- This searching technique is faster and easier to implement.
- Requires no extra space.
- Reduces the time complexity of the program to a greater extent. (The term **time complexity** might be new to you, you will get to understand this when you will be studying algorithmic analysis. For now, just consider it as the time taken by a particular algorithm in its execution, and time complexity is determined by the

number of operations that are performed by that algorithm i.e. time complexity is directly proportional to the number of operations in the program).

Sorting

Sorting is a permutation of a list of elements of such that the elements are either in increasing ***(ascending)*** order or decreasing ***(descending)*** order.

There are many different sorting techniques. The major difference is the amount of **space** and **time** they consume while being performed in the program.

For now, we will be discussing the following sorting techniques:

- Selection sort
- Bubble sort
- Insertion sort

Let us now discuss these sorting techniques in detail.

Selection Sort

Selection sort is an algorithm that selects the smallest element from an unsorted list in each iteration and places that element at the beginning of the unsorted list. The detailed algorithm is given below.

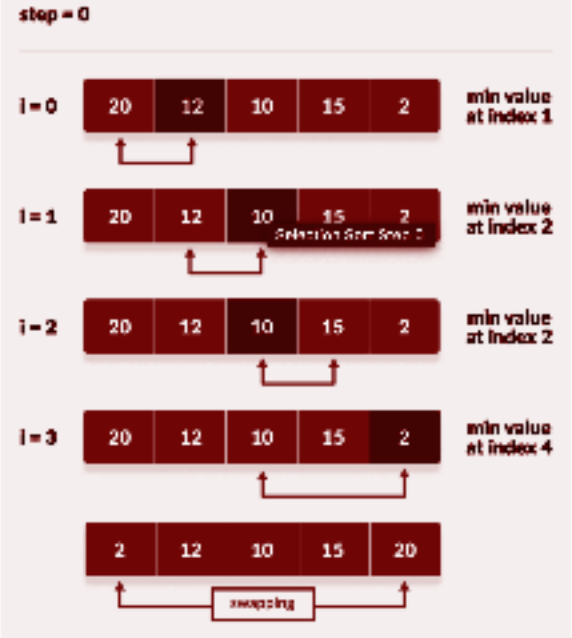
- Consider the given unsorted array to be:

- Set the first element as `minimum`.
- Compare `minimum` with the second element. If the second element is smaller than `minimum`, assign the second element as `minimum`.
- Compare `minimum` with the third element. Again, if the third element is smaller, then assign `minimum` to the third element otherwise do nothing. The process goes on until the last element.

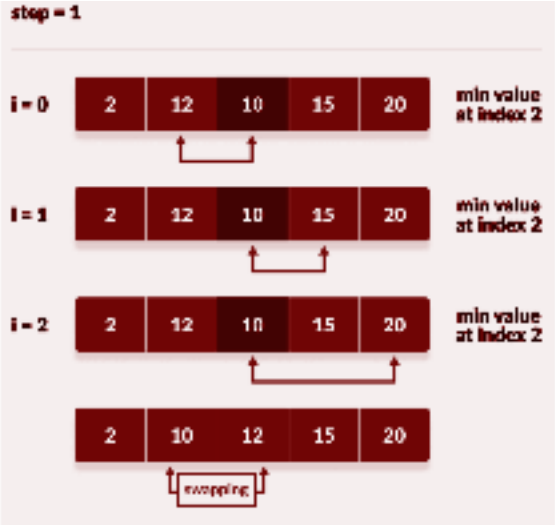
- After each iteration, `minimum` is placed in the front of the unsorted list.

- For each iteration, indexing starts from the first unsorted element. These steps are repeated until all the elements are placed at their correct positions.

First Iteration



Second Iteration:



Third Iteration

Fourth Iteration

Python Code

```
# Selection sort in Python
def selectionSort(arr, size):

    for step in range(size):
        minimum = step

        for i in range(step + 1, size):

            # to sort in descending order, change > to < in this line
            # select the minimum element in each loop
            if arr[i] < arr[minimum]:
                minimum = i

        # Swap
        (arr[step], arr[minimum]) = (arr[minimum], arr[step])


input = [20, 12, 10, 15, 2]
size = len(input)
selectionSort(input, size)
print('Sorted Array in Ascending Order: ')
print(input)
```

We will get the **output** of the above code as:

```
Sorted Array in Ascending Order: [2, 10, 12, 15, 20]
```

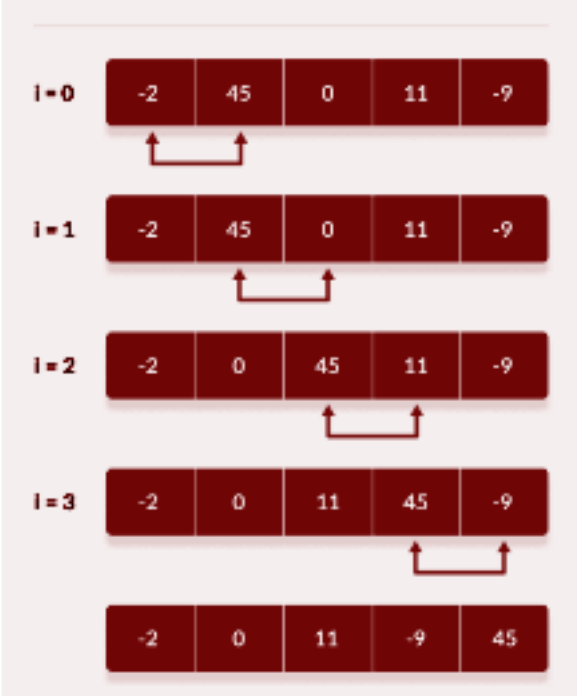
Bubble Sort

Bubble sort is an algorithm that compares the adjacent elements and swaps their positions if they are not in the intended order. The order can be ascending or descending.

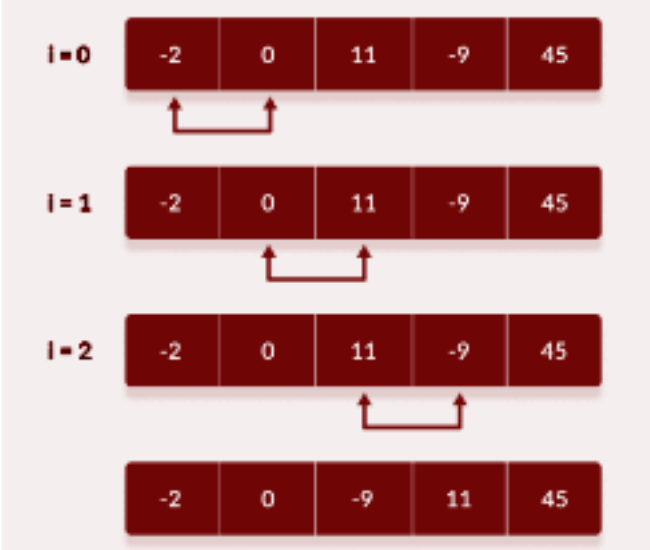
How does Bubble Sort work?

- Starting from the first index, compare the first and the second elements. If the first element is greater than the second element, they are swapped.
- Now, compare the second and third elements. Swap them if they are not in order.
- The above process goes on until the last element.
- The same process goes on for the remaining iterations. After each iteration, the largest element among the unsorted elements is placed at the end.
- In each iteration, the comparison takes place up to the last unsorted element.
- The array is sorted when all the unsorted elements are placed at their correct positions.

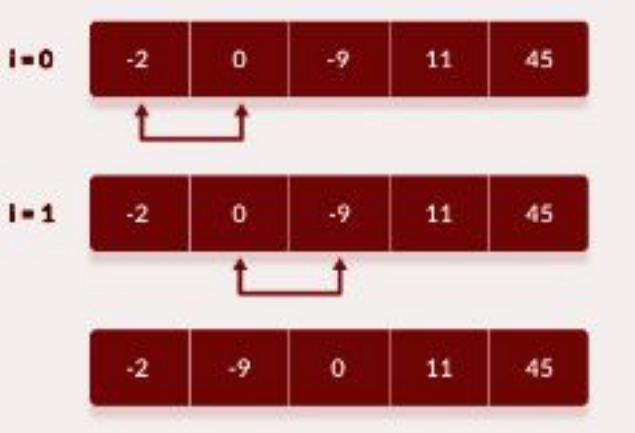
Let the array be `[-2, 45, 0, 11, -9]`.



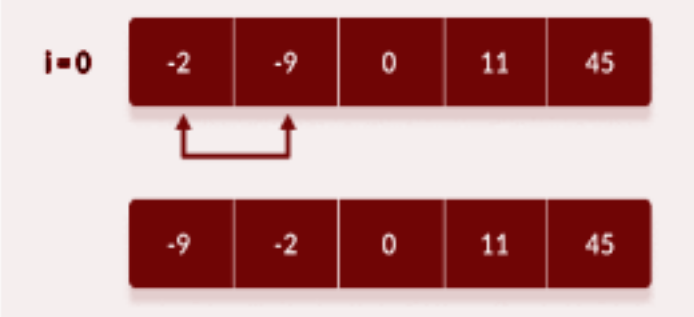
First Iteration



Second Iteration



Third Iteration



Fourth Iteration

Python Code

```
# Bubble sort in Python
def bubbleSort(arr):
    # run loops two times: one for walking through the array
    # and the other for comparison
```

```
    for i in range(len(arr)):
        for j in range(0, len(arr) - i - 1):
            # To sort in descending order, change > to < in this line.
            if arr[j] > arr[j + 1]:
                # swap if greater is at the rear position
                (arr[j], arr[j + 1]) = (arr[j + 1], arr[j])

input = [-2, 45, 0, 11, -9]
bubbleSort(input)
print('Sorted Array in Ascending Order:')
print(input)
```

We will get the **output** of the above code as:

```
Sorted Array in Ascending Order: [-9, -2, 0, 11, 45]
```

Insertion Sort

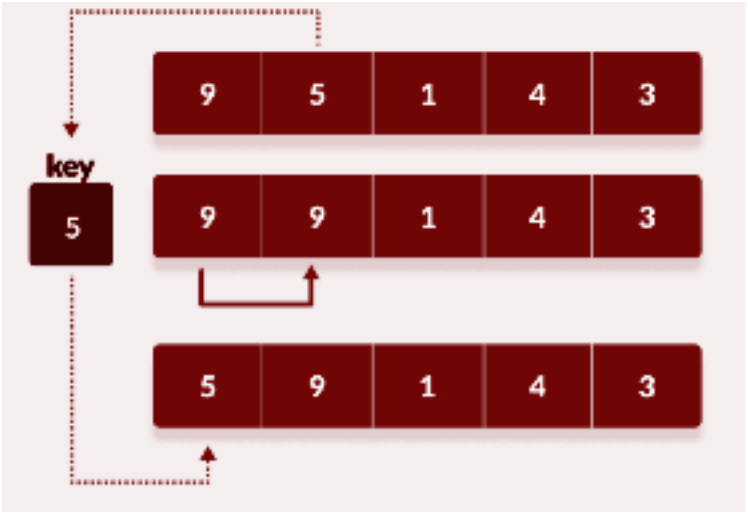
- Insertion sort works similarly as we sort cards in our hand in a card game.
- We assume that the first card is already sorted
- Then, we select an unsorted card.
- If the unsorted card is greater than the card in hand, it is placed on the right otherwise, to the left.
- In the same way, other unsorted cards are taken and put in the right place. A similar approach is used by insertion sort.
- Insertion sort is a sorting algorithm that places an unsorted element at its suitable place in each iteration

Algorithm

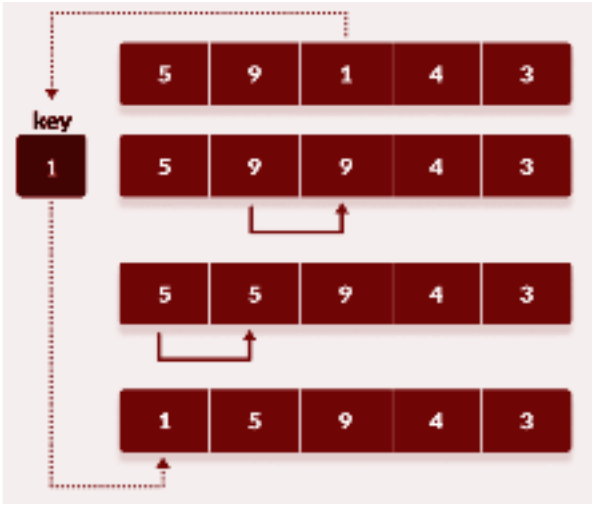
- Suppose we need to sort the following array.

- The first element in the array is assumed to be sorted. Take the second element and store it separately in **key**.
- Compare **key** with the first element. If the first element is greater than **key**, then **key** is placed in front of the first element.
- If the first element is greater than **key**, then **key** is placed in front of the first element.
- Now, the first two elements are sorted.
- Take the third element and compare it with the elements on the left of it. Placed it just behind the element smaller than it. If there is no element smaller than it, then place it at the beginning of the array.
- Similarly, place every unsorted element at its correct position.

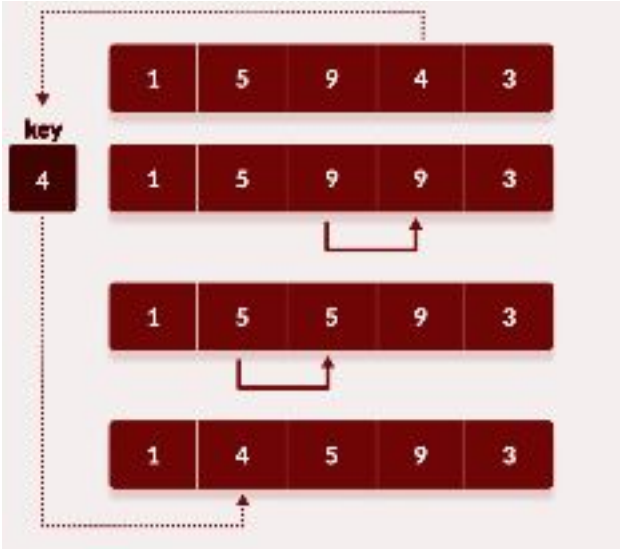
The various iterations are depicted below:



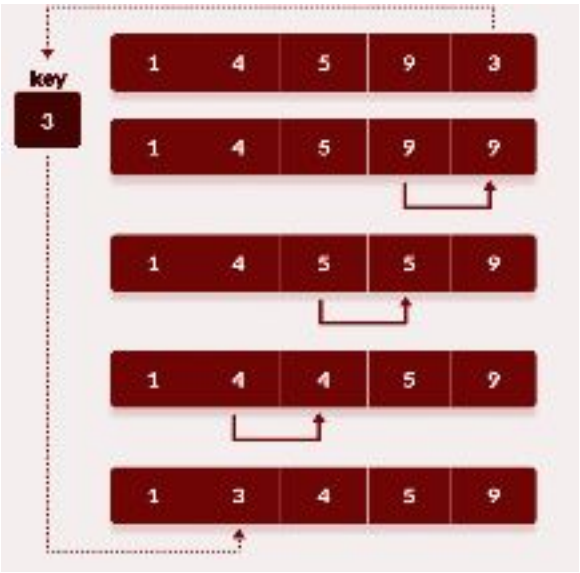
First Iteration



Second Iteration



Third Iteration



Fourth Iteration

Python Code

```
# Insertion sort in Python
def insertionSort(arr):

    for step in range(1, len(arr)):
        key = arr[step]
        j = step - 1

# Compare key with each element on the left of it until an element smaller
than it is found
        # For descending order, change key<array[j] to key>array[j].
        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]
```

```
            j = j - 1

        # Place the key at after the element just smaller than it.
        arr[j + 1] = key


input = [9, 5, 1, 4, 3]
insertionSort(input)
print('Sorted Array in Ascending Order:')
print(input)
```

We will get the **output** as:

```
Sorted Array in Ascending Order: [1, 3, 4, 5, 9]
```

Binary Search (Edge Case)

Remember the exploratory task we gave you?

Suppose your sorted array has duplicate numbers. For example, if the array is as follows: [1, 2, 2, 3, 3, 3, 4, 8, 9, 19, 19, 19].

Let's see how we can modify the code to find the first occurrence of the element **x** to be found.

In the current code, we exit when any of the following two conditions are satisfied

- When low < high, or
- When arr[mid] == x.

The modification we need is quite simple. Instead of exiting the code when arr[mid] == x, we shall assign **high** and **index** as **mid** as follows:

```
   index = mid
   high = mid
```

Let us see the entire code.

Python Code

```
#Function to implement Binary Search Algorithm
def binarySearch(array, x, low, high):

    low = 0;
    high = n
    mid = low
    index = -1

    while low < high :

        mid = low + (high - low) // 2

        if array[mid] > x :
            high = mid
        elif array[mid] < x :
            low = mid + 1
        else :
            index = mid
            high = mid

    return index

array = [1, 3, 3, 3, 3, 6, 7, 9, 9]
x = 3

result = binarySearch(array, x, 0, len(array)-1)

if result != -1: #If element is found
    print("Element is present at index " + str(result))
else: #If element is not found
    print("Not found")
```