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**)
- 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 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:

Java Code

```
// Function to implement Binary Search Algorithm
public static int binarySearch(int arr[], int n, int x) {
      int start = 0, end = n - 1;
      // Repeat until the pointers start and end meet each other
      while(start <= end) {
            int mid = (start + end) / 2; // Middle Index
            if(arr[mid] == x) {  // element found
                  return mid;
            }
            else if(x < arr[mid]) {   // x is on the left side
                  end = mid - 1;
            }
            else {                    // x is on the right side
                  start = mid + 1;
            }
      }

      return -1;    // Element is not found
}

public static void main(String[] args) {

      int[] input = {3, 4, 5, 6, 7, 8, 9};

      int x = 4;

      System.out.print(binarySearch(input, n, x)); // print index
}
```

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

```
1                                   // Element found 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 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

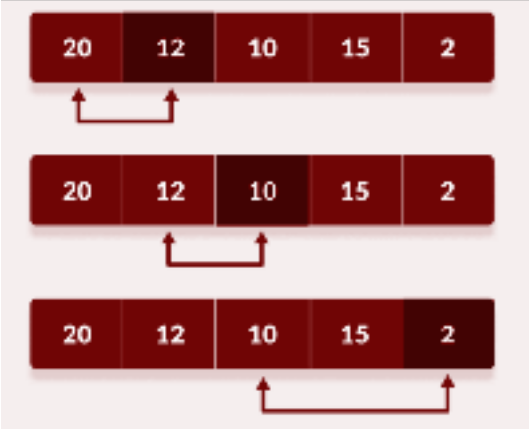
Let us now discuss these sorting techniques in detail.

Selection Sort

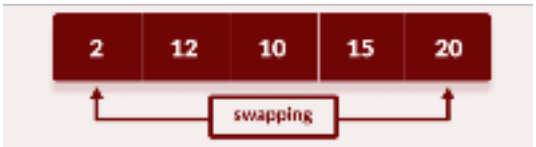
Selection sort is an algorithm that selects the smallest element from an unsorted list in each iteration and places that element at the beginning of the unsorted list. The detailed algorithm is given below.

- 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.

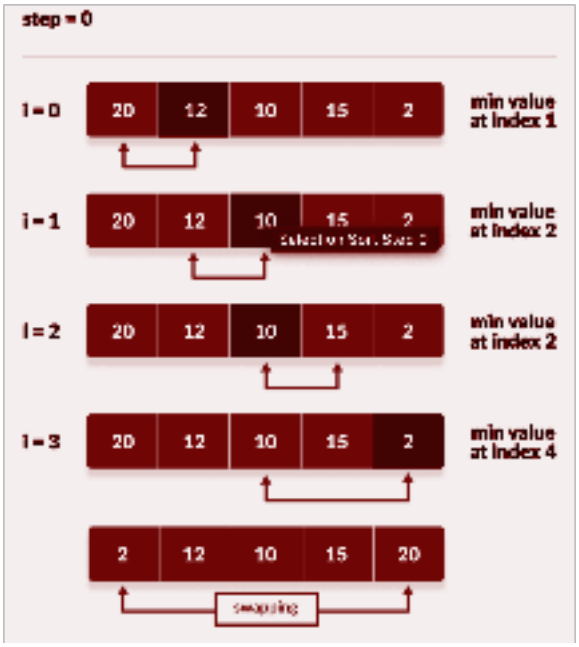


- After each iteration, `minimum` is placed in the front of the unsorted list.

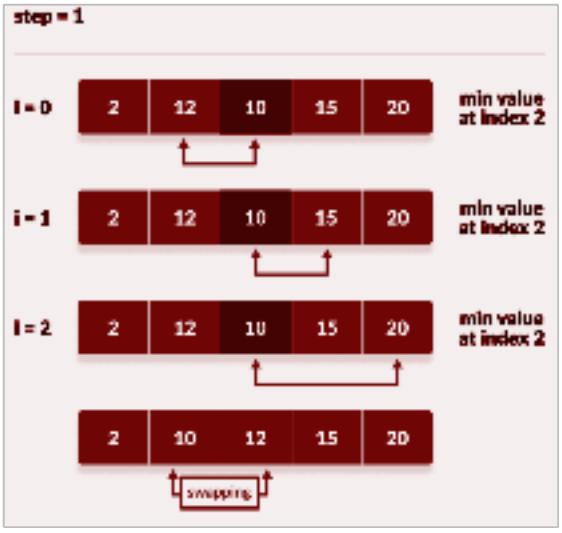


- For each iteration, indexing starts from the first unsorted element. These steps are repeated until all the elements are placed at their correct positions.

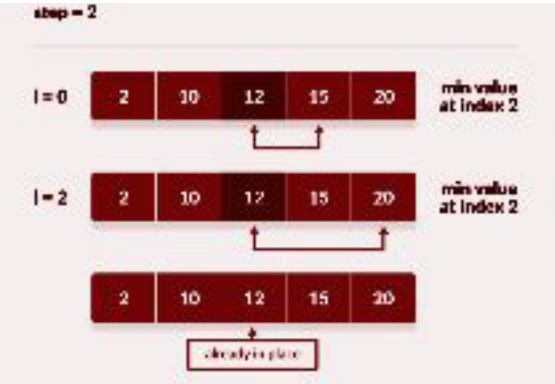
First Iteration



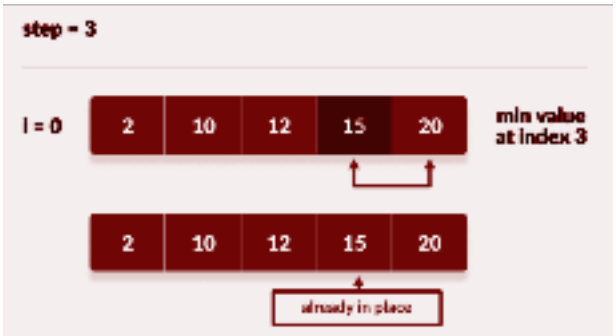
Second Iteration:



Third Iteration



Fourth Iteration



Java Code

```
public static void selectionSort(int input[], int n) {
      for(int i = 0; i < n-1; i++ ) {
          // Find min element in the array
          int min = input[i], minIndex = i;
          for(int j = i+1; j < n; j++) {
               // to sort in descending order, change < to > in this
                // line select the minimum element in each loop
              if(input[j] < min) {
                 min = input[j];
                 minIndex = j;
              }
          }
          // Swap
          int temp = input[i];
          input[i] = input[minIndex];
```

```
            input[minIndex] = temp;
        }
}

public static void main(String[] args) {
      int input[] = {20, 12, 10, 15, 2};
      selectionSort(input, 6);
      for(int i = 0; i < 6; i++) {
            System.out.print(input[i] + " ");

      }
}
```

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

```
2 10 12 15 20                                    // sorted array
```

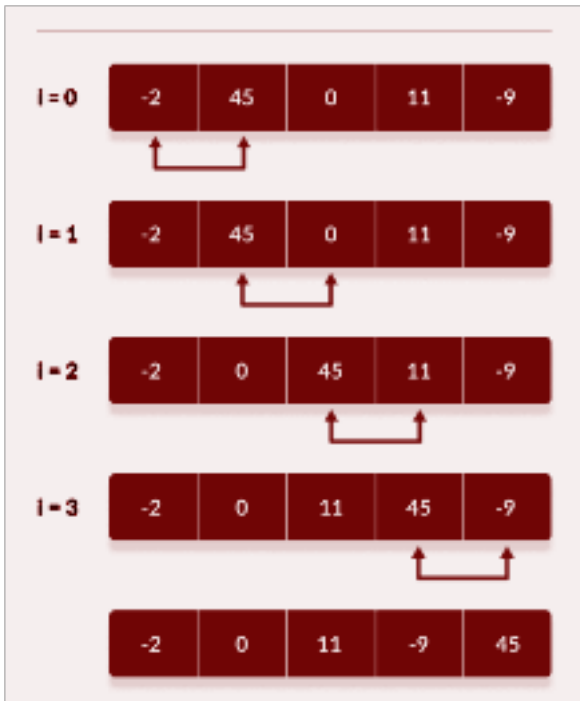
Bubble Sort

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.

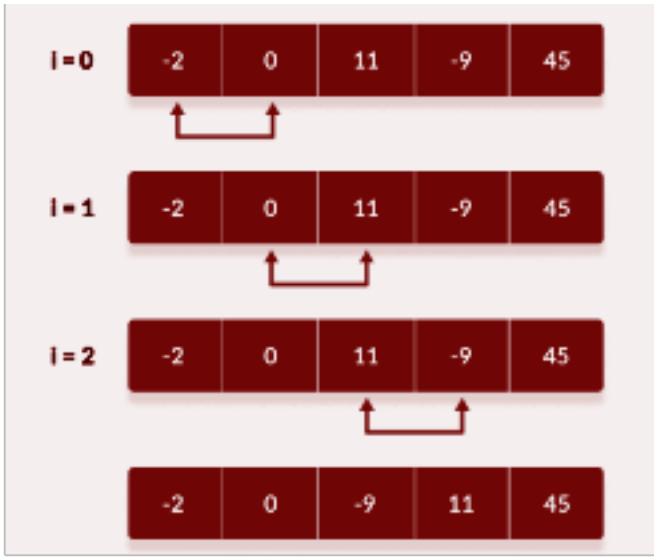
How does Bubble Sort work?

- 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.

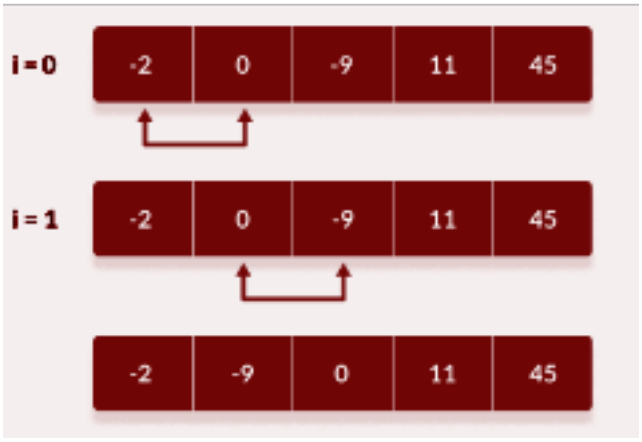
Let the array be `[-2, 45, 0, 11, -9]`.



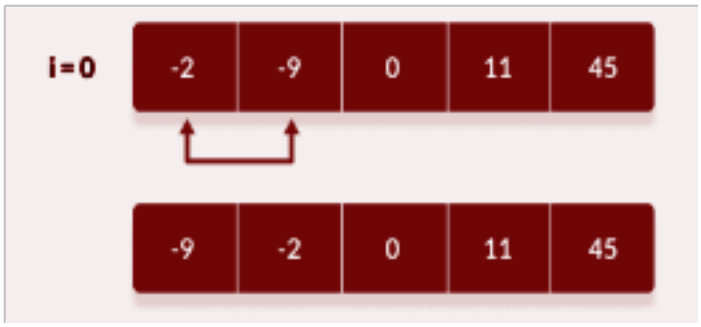
First Iteration



Second Iteration



Third Iteration



Fourth Iteration

Java Code

```
public static void bubbleSort(int array[], int size) {

  // run loops two times: one for walking through the array
  // and the other for comparison
  for (int step = 0; step < size - 1; ++step) {
    for (int i = 0; i < size - step - 1; ++i) {

      // To sort in descending order, change > to < in this line.
      if (array[i] > array[i + 1]) {

        // swap if greater is at the rear position
        int temp = array[i];
        array[i] = array[i + 1];
        array[i + 1] = temp;
      }
    }
  }
}

// driver code
public static void main(String[] args) {
  int input[] = {-2, 45, 0, 11, -9};
      bubbleSort(input, 6);
      for(int i = 0; i < 6; i++) {
            System.out.print(input[i] + " ");
      }
}
```

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

```
-9 -2 0 11 45                          // sorted array
```

Insertion Sort

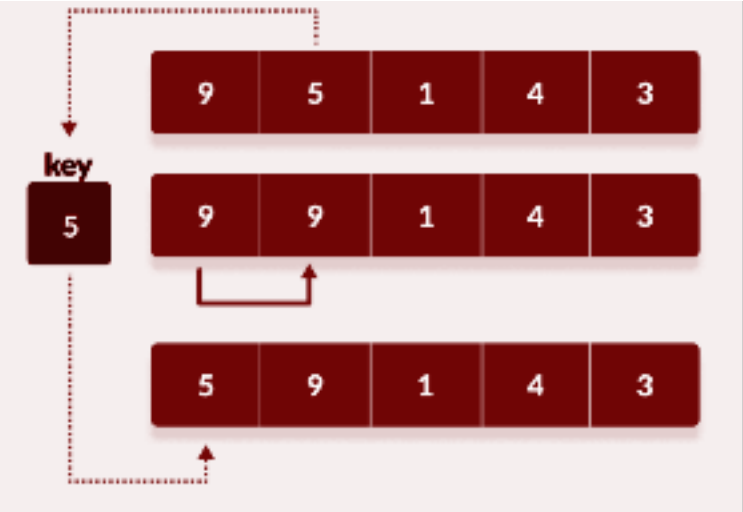
- Insertion sort works similarly as we sort cards in our hand in a card game.
- We assume that the first card is already sorted
- Then, we select an unsorted card.
- If the unsorted card is greater than the card in hand, it is placed on the right otherwise, to the left.
- In the same way, other unsorted 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

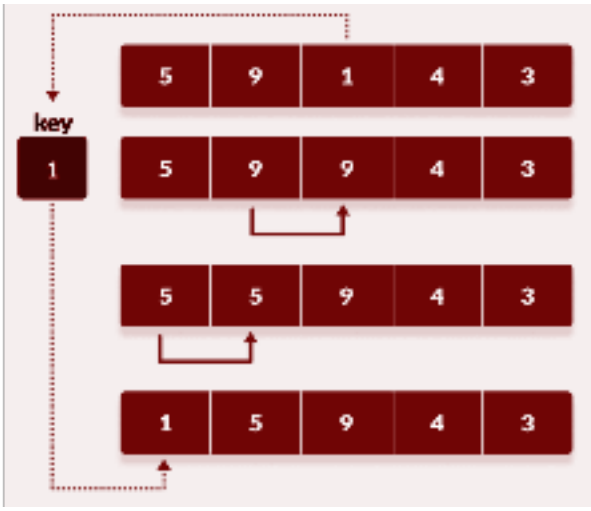
- Suppose we need to sort the following array.

- The first element in the array is assumed to be sorted. Take the second element and store it separately in **key**.
- Compare the 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.

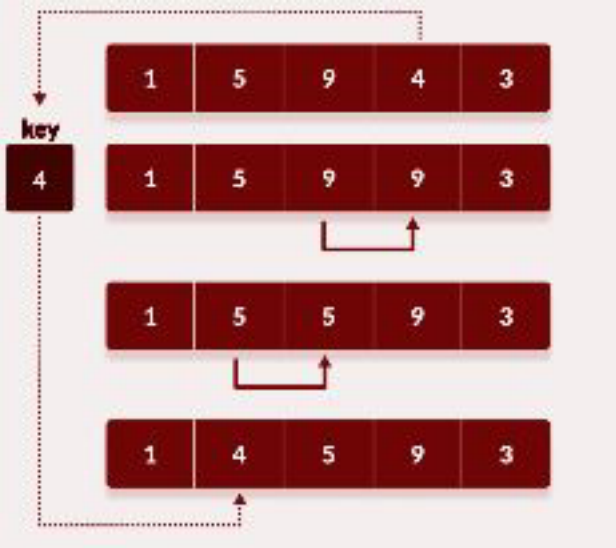
The various iterations are depicted below:



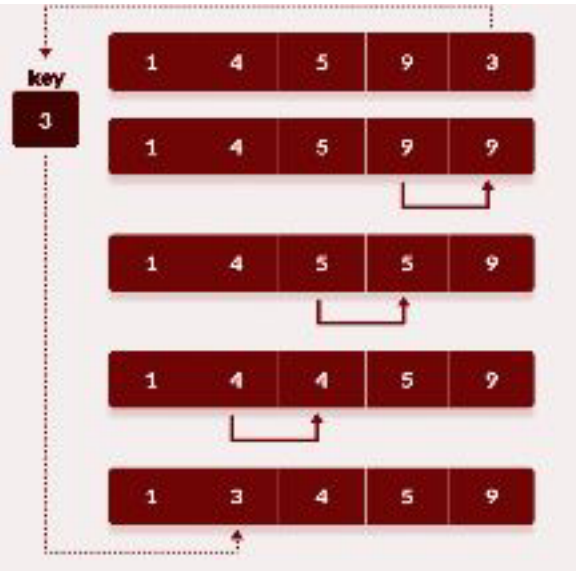
First Iteration



Second Iteration



Third Iteration



Fourth Iteration

Java Code

```
public static void insertionSort(int array[], int size) {
   int key, j;
   for(int i = 1; i<size; i++) {
      key = array[i];                //take value
      j = i;
      // Compare key with each element on the left of it until an
      // element smaller than it is found
      while(j > 0 && array[j-1]>key) {
         array[j] = array[j-1];
         j--;
      }
      array[j] = key;                 //insert in right place
   }
}

// driver code
public static void main(String[] args) {
  int input[] = {9, 5, 1, 4, 3};
      insertionSort(input, 6);
      for(int i = 0; i < 6; i++) {
           System.out.print(input[i] + " ");
      }
}
```

We will get the **output** as:

```
1 3 4 5 9                          // sorted array
```

Now, practice different questions to get more familiar with the concepts. In the advanced course, you will study more types of sorting techniques.

Problem Statement - Merge 2 Sorted Arrays

In this problem, you are given 2 sorted arrays, you need to return a new array including all the elements of both given arrays in a sorted manner.

Algorithm

1. Create an array arr3[] of size n1 + n2.
2. Simultaneously traverse arr1[] and arr2[].
    - Pick a smaller of current elements in arr1[] and arr2[], copy this smaller element to the next position in arr3[] and move ahead in arr3[] as well as in the array whose element is picked.
    - Keep reiterating the above step, till either one of arr1[] and arr2[] are completely traversed.
3. Copy the remaining elements of the array which is left untraversed if there exists into the arr3, as the above loop breaks if any of the pointers exceeds their respective size.

Java Code

```
public static void mergeArrays(int arr1[], int[] arr2) {
   int n1 = arr1.length;
   int n2 = arr2.length;

   int[] arr3 = new int[n1+n2];
   int i = 0, j = 0, k = 0;
   while (i<n1 && j <n2){
       if (arr1[i] < arr2[j])
           arr3[k++] = arr1[i++];
       else
            arr3[k++] = arr2[j++];
   }

   // Store remaining elements of first array
   while (i < n1)
       arr3[k++] = arr1[i++];

   // Store remaining elements of second array
   while (j < n2)
       arr3[k++] = arr2[j++];

   return arr3;
}
```