Topics to cover:

1. Sorting algorithms

1. Sorting algorithms

1.1 What is Sorting?

A Sorting Algorithm is used to rearrange a given array or list of elements according to a comparison operator on the elements. The comparison operator is used to decide the new order of elements in the respective data structure. For example, consider an array A = {A1, A2, A3, A4, ?? An }, the array is called to be in ascending order if elements of A are arranged like A1 > A2 > A3 > A4 > A5 > … > An .

The Array sorted in ascending order will be given as;

A[] = { 2, 4, 5, 9, 10, 14, 18, 30, 34, 45 }

There are many techniques by using which, sorting can be performed. In this section of the tutorial, we will discuss each method in detail.

1.2 Different sorting algorithms.

There are numerous sorting algorithms. We will discuss some of the most common sorting algorithms in this tutorial. We can list them briefly in the following table.

SN	Sorting Algorithms	Description
1	Bubble sort	It is the simplest sort method which performs sorting by repeatedly moving the largest element to the highest index of the array. It comprises of comparing each element to its adjacent element and replace them accordingly.
2	Selection Sort	Selection sort finds the smallest element in the array and place it on the first place on the list, then it finds the second smallest element in the array and places it on the second place. This process continues until all the elements are moved to their correct ordering. It carries running time O(n2) which is worse than insertion sort.
3	Insertion sort	As the name suggests, insertion sort inserts each element of the array to its proper place. It is a very simple sort method which is used to arrange the deck of cards while playing bridge.
4	Merge sort	Merge sort follows divide and conquer approach in which, the list is first divided into the sets of equal elements and then each half of the list is sorted by using merge sort. The sorted list is combined again to form an elementary sorted array.
5	Quick sort	Quick sort is the most optimized sort algorithms which performs sorting in O(n log n) comparisons. Like Merge sort, quick sort also work by using divide and conquer approach.

1.2.1 Bubble sort

The working procedure of bubble sort is simplest. Bubble sort works on the repeatedly swapping of adjacent elements until they are not in the intended order. It is called bubble sort because the movement of array elements is just like the movement of air bubbles in the water. Bubbles in water rise up to the surface; similarly, the array elements in bubble sort move to the end in each iteration.

Although it is simple to use, it is primarily used as an educational tool because the performance of bubble sort is poor in the real world. It is not suitable for large data sets. The average and worst-case complexity of Bubble sort is O(n2), where n is a number of items.

Bubble short is majorly used where -

- complexity does not matter
- simple and shortcode is preferred

1.2.1.1 Bubble sort algorithm

In the algorithm given below, suppose **arr** is an array of **n** elements. The assumed **swap** function in the algorithm will swap the values of given array elements.

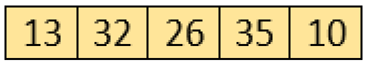
```
1. begin BubbleSort(arr)
2.    for all array elements
3.       if arr[i] > arr[i+1]
4.          swap(arr[i], arr[i+1])
5.       end if
6.    end for
7.    return arr
8. end BubbleSort
```

1.2.1.2 Working of Bubble sort algorithm:

Now, let's see the working of Bubble sort Algorithm.

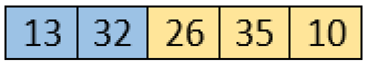
To understand the working of the bubble sort algorithm, let's take an unsorted array. We are taking a short and accurate array, as we know the complexity of bubble sort is **$O(n^2)$.**

Let the elements of array are -

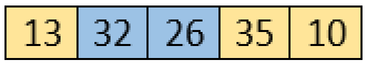


First Pass

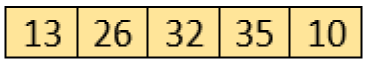
In this first pass, sorting will start from the initial two elements. Let's compare them to see which is greater.



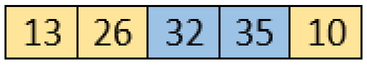
Here, 32 is greater than 13 (32 > 13), so it is already sorted. Now, compare 32 with 26.



Here, 26 is smaller than 36. So, swapping is required. After swapping new array will look like -

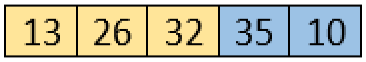


Now, compare 32 and 35.

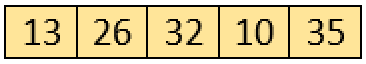


Here, 35 is greater than 32. So, there is no swapping required as they are already sorted.

Now, the comparison will be in between 35 and 10.



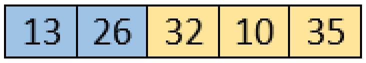
Here, 10 is smaller than 35 that are not sorted. So, swapping is required. Now, we reach at the end of the array. After first pass, the array will be -

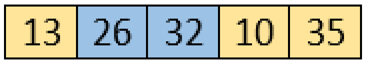


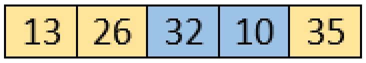
Now, move to the second iteration.

Second Pass

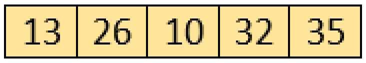
The same process will be followed for the second iteration.

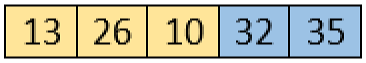






Here, 10 is smaller than 32. So, swapping is required. After swapping, the array will be -

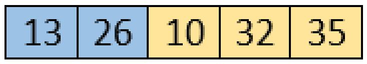


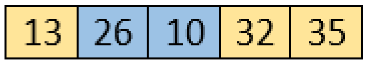


Now, move to the third iteration.

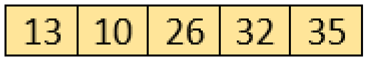
Third Pass

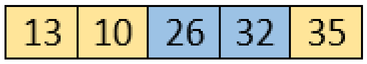
The same process will be followed for third iteration.

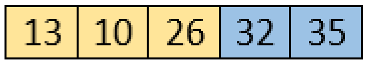




Here, 10 is smaller than 26. So, swapping is required. After swapping, the array will be -



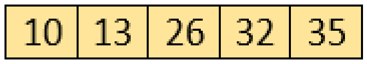




Now, move to the fourth iteration.

Fourth pass

Similarly, after the fourth iteration, the array will be -



Hence, there is no swapping required, so the array is completely sorted.

1.2.1.3 Program: Bubble sort:

```
class BubbleSort {
   public static void main(String[] args) {
```

```
            int len = args.length;
            int myArray[]=new int[len];
        int[] sortedArray = new int[len];

        System.out.print("Array before sorting: ");
        for(int i=0;i<len;i++){
                myArray[i]=Integer.parseInt(args[i]);
            System.out.print(myArray[i]+" ");
        }

        Sort s = new Sort();
        sortedArray=s.BubbleSort(myArray);
        System.out.print("\nArray after sorting: ");
        for(int i=0;i<len;i++){
            System.out.print(sortedArray[i]+" ");
        }
            System.out.println();
    }
}

class Sort{
      int[] BubbleSort(int[] myArray){
              int len=myArray.length;
          int temp;
          int numSteps=0;
          System.out.println("\n\nBubble sort is starting: ");
          for(int i=0;i<len;i++){
              System.out.print("\nAfter pass "+(i+1)+" : ");
              for(int j=0;j<len-1-i;j++){
                  if(myArray[j]>myArray[j+1]){
                      temp=myArray[j];
                      myArray[j]=myArray[j+1];
                      myArray[j+1]=temp;
                      numSteps++;
                  }
              }
              for(int k=0;k<len;k++){
                  System.out.print(myArray[k]+" ");
              }
          }
          System.out.print("\n\nTotal number of steps: "+numSteps+"\n");
          return myArray;
      }
    }
```

Output:

```
java BubbleSort 10 9 8 7 6 5 4 3 2 1
Array before sorting: 10 9 8 7 6 5 4 3 2 1

Bubble sort is starting:

After pass 1 : 9 8 7 6 5 4 3 2 1 10
After pass 2 : 8 7 6 5 4 3 2 1 9 10
After pass 3 : 7 6 5 4 3 2 1 8 9 10
After pass 4 : 6 5 4 3 2 1 7 8 9 10
After pass 5 : 5 4 3 2 1 6 7 8 9 10
After pass 6 : 4 3 2 1 5 6 7 8 9 10
After pass 7 : 3 2 1 4 5 6 7 8 9 10
After pass 8 : 2 1 3 4 5 6 7 8 9 10
After pass 9 : 1 2 3 4 5 6 7 8 9 10
After pass 10 : 1 2 3 4 5 6 7 8 9 10

Total number of steps: 45

Array after sorting: 1 2 3 4 5 6 7 8 9 10
```

1.2.1.4 Bubble sort complexity

Now, let's see the time complexity of bubble sort in the best case, average case, and worst case. We will also see the space complexity of bubble sort.

1.2.1.1.1 Time complexity

Case	**Time Complexity**
Best Case	$O(n)$
Average Case	$O(n^2)$
Worst Case	$O(n^2)$

- **Best Case Complexity -** It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of bubble sort is **O(n).**
- **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of bubble sort is **$O(n^2)$.**
- **Worst Case Complexity -** It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of bubble sort is **$O(n^2)$.**

1.2.1.1.2 Space complexity

Space Complexity	O(1)
Stable	YES

The space complexity of bubble sort is O(1). It is because, in bubble sort, an extra variable is required for swapping.

1.2.2 Selection sort

The working procedure of selection sort is also simple.

In selection sort, the smallest value among the unsorted elements of the array is selected in every pass and inserted to its appropriate position into the array. It is also a simple algorithm. It is an in-place comparison sorting algorithm. In this algorithm, the array is divided into two parts, first is the sorted part, and another one is the unsorted part. Initially, the sorted part of the array is empty, and the unsorted part is the given array. Sorted part is placed at the left, while the unsorted part is placed at the right.

In selection sort, the first smallest element is selected from the unsorted array and placed at the first position. After that second smallest element is selected and placed in the second position. The process continues until the array is entirely sorted.

The average and worst-case complexity of selection sort is **O(n^2)**, where **n** is the number of items. Due to this, it is not suitable for large data sets.

Selection sort is generally used when -

- A small array is to be sorted
- Swapping cost doesn't matter
- It is compulsory to check all elements

1.2.2.1 The Selection sort algorithm

```
SELECTION SORT(arr, n)

Step 1: Repeat Steps 2 and 3 for i = 0 to n-1
Step 2: CALL SMALLEST(arr, i, n, pos)
Step 3: SWAP arr[i] with arr[pos]
[END OF LOOP]
Step 4: EXIT

SMALLEST (arr, i, n, pos)
```

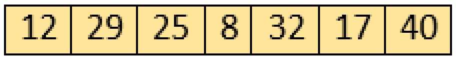
```
Step 1: [INITIALIZE] SET SMALL = arr[i]
Step 2: [INITIALIZE] SET pos = i
Step 3: Repeat for j = i+1 to n
if (SMALL > arr[j])
    SET SMALL = arr[j]
SET pos = j
[END OF if]
[END OF LOOP]
Step 4: RETURN pos
```

1.2.2.2 The working of Selection sort algorithm

Now, let's see the working of the Selection sort Algorithm.

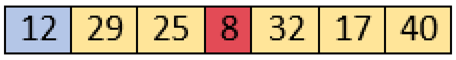
To understand the working of the Selection sort algorithm, let's take an unsorted array. It will be easier to understand the Selection sort via an example.

Let the elements of array are -

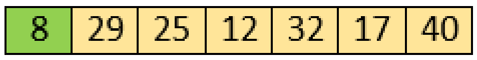


Now, for the first position in the sorted array, the entire array is to be scanned sequentially.

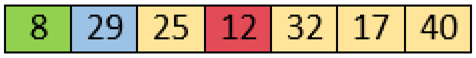
At present, **12** is stored at the first position, after searching the entire array, it is found that **8** is the smallest value.



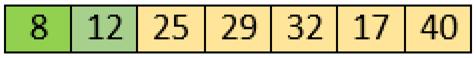
So, swap 12 with 8. After the first iteration, 8 will appear at the first position in the sorted array.



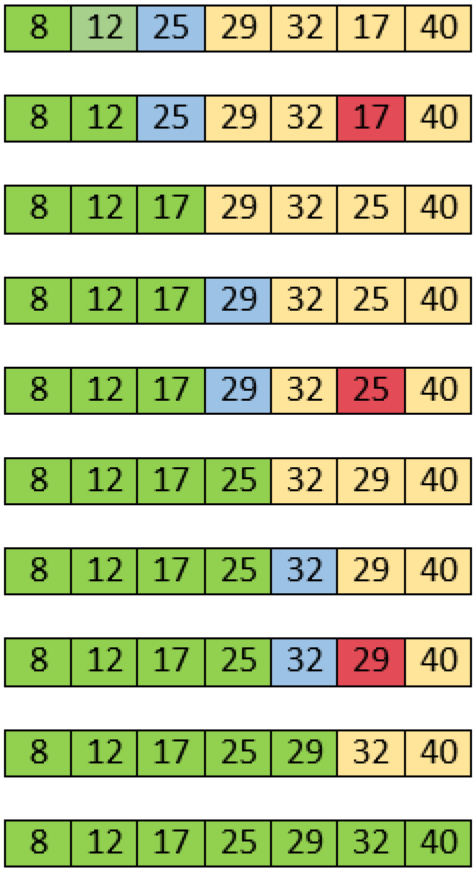
For the second position, where 29 is stored presently, we again sequentially scan the rest of the items of unsorted array. After scanning, we find that 12 is the second lowest element in the array that should be appeared at second position.



Now, swap 29 with 12. After the second iteration, 12 will appear at the second position in the sorted array. So, after two iterations, the two smallest values are placed at the beginning in a sorted way.



The same process is applied to the rest of the array elements. Now, we are showing a pictorial representation of the entire sorting process.



Now, the array is completely sorted.

1.2.2.2 Complexity of selection sort

1.2.2.2.1 Time Complexity

Case	Time Complexity
Best Case	$O(n^2)$
Average Case	$O(n^2)$
Worst Case	$O(n^2)$

- **Best Case Complexity -** It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of selection sort is **$O(n^2)$**.
- **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of selection sort is **$O(n^2)$**.
- **Worst Case Complexity -** It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of selection sort is **$O(n^2)$**.

1.2.2.2.2 Space Complexity

Space Complexity	O(1)

Stable	YES

- The space complexity of the selection sort is O(1). It is because, in selection sort, an extra variable is required for swapping.

1.2.2.3 Program implementation of selection sort

```
class SelectionMain {
        public static void main(String[] args) {
   int len = args.length;
   int myArray[]=new int[len];
        int[] sortedArray = new int[len];

        System.out.print("Array before sorting: ");
        for(int i=0;i<len;i++){
        myArray[i]=Integer.parseInt(args[i]);
        System.out.print(myArray[i]+" ");
        }

        Sort s = new Sort();
        sortedArray=s.SelectionSort(myArray);
        System.out.print("\nArray after sorting: ");
        for(int i=0;i<len;i++){
        System.out.print(sortedArray[i]+" ");
        }
```

```
    System.out.println();
            }
}

class Sort{
    int[] SelectionSort(int[] a){
            int i, j, small;
            int n = a.length;
             int steps=0;
            System.out.println("\nSelection sort is starting....");
                        for (i = 0; i < n-1; i++){
                                small = i; //minimum element in unsorted array

                                for (j = i+1; j < n; j++){
                                if (a[j] < a[small]){
                                          steps++;
                                         small = j;
                                }

                                // Swap the minimum element with the first element
                                          int temp = a[small];
                                          a[small] = a[i];
                                          a[i] = temp;
                                }
                        System.out.print("\nAfter pass "+(i+1)+" : ");
```

```
                for(int k=0;k<n;k++){
                        System.out.print(a[k]+" ");
                    }

            }
            System.out.print("Total number of steps:  "+steps);
             return a;
    }
}
```

Output:

1.2.3 Insertion sort

Insertion sort works similar to the sorting of playing cards in hands. It is assumed that the first card is already sorted in the card game, and then we select an unsorted card. If the selected unsorted card is greater than the first card, it will be placed at the right side; otherwise, it will be placed at the left side. Similarly, all unsorted cards are taken and put in their exact place.

The same approach is applied in insertion sort. The idea behind the insertion sort is that first take one element, iterate it through the sorted array. Although it is simple to use, it is not appropriate for large data sets as the time complexity of insertion sort in the average case and worst case is **O(n^2)**, where n is the number of items. Insertion sort is less efficient than the other sorting algorithms like heap sort, quick sort, merge sort, etc.

Insertion sort has various advantages such as -

- Simple implementation
- Efficient for small data sets
- Adaptive, i.e., it is appropriate for data sets that are already substantially sorted.

Now, let's see the algorithm of insertion sort.

1.2.3.1 Insertion sort algorithm:

The simple steps of achieving the insertion sort are listed as follows -

Step 1 - If the element is the first element, assume that it is already sorted. Return 1.

Step2 - Pick the next element, and store it separately in a **key.**

Step3 - Now, compare the **key** with all elements in the sorted array.

Step 4 - If the element in the sorted array is smaller than the current element, then move to the next element. Else, shift greater elements in the array towards the right.

Step 5 - Insert the value.

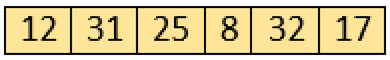
Step 6 - Repeat until the array is sorted.

1.2.3.1 Working of Insertion sort:

Now, let's see the working of the insertion sort Algorithm.

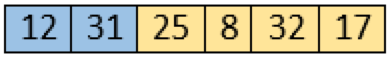
To understand the working of the insertion sort algorithm, let's take an unsorted array. It will be easier to understand the insertion sort via an example.

Let the elements of array are -

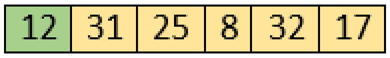


12	31	25	8	32	17

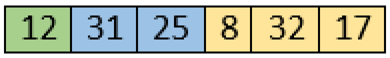
Initially, the first two elements are compared in insertion sort.

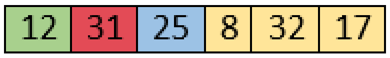


Here, 31 is greater than 12. That means both elements are already in ascending order. So, for now, 12 is stored in a sorted sub-array.



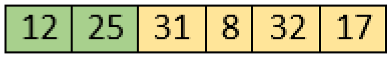
Now, move to the next two elements and compare them.



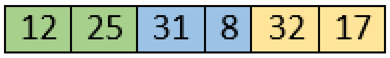


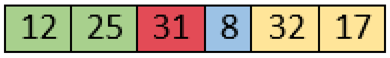
Here, 25 is smaller than 31. So, 31 is not at correct position. Now, swap 31 with 25. Along with swapping, insertion sort will also check it with all elements in the sorted array.

For now, the sorted array has only one element, i.e. 12. So, 25 is greater than 12. Hence, the sorted array remains sorted after swapping.

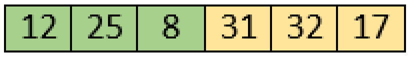


Now, two elements in the sorted array are 12 and 25. Move forward to the next elements that are 31 and 8.

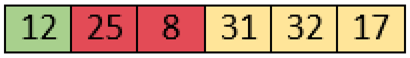




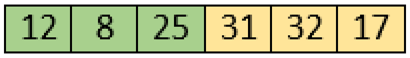
Both 31 and 8 are not sorted. So, swap them.



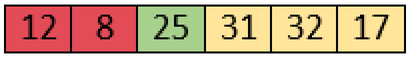
After swapping, elements 25 and 8 are unsorted.



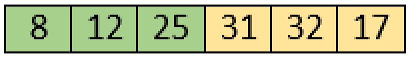
So, swap them.



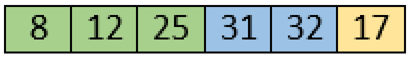
Now, elements 12 and 8 are unsorted.



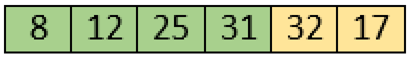
So, swap them too.



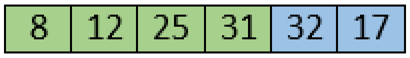
Now, the sorted array has three items that are 8, 12 and 25. Move to the next items that are 31 and 32.



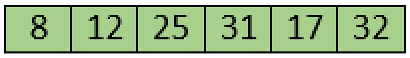
Hence, they are already sorted. Now, the sorted array includes 8, 12, 25 and 31.

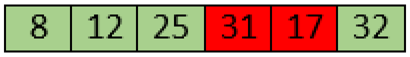


Move to the next elements that are 32 and 17.

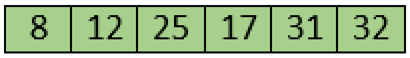


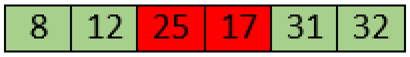
17 is smaller than 32. So, swap them.



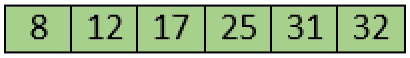


Swapping makes 31 and 17 unsorted. So, swap them too.





Now, swapping makes 25 and 17 unsorted. So, perform swapping again.



Now, the array is completely sorted.

1.2.3.2 Complexity of Insertion sort:

Now, let's see the time complexity of insertion sort in best case, average case, and in worst case. We will also see the space complexity of insertion sort.

1.2.3.2.1 Time complexity:

Case	**Time Complexity**
Best Case	O(n)
Average Case	O(n^2)
Worst Case	O(n^2)

- **Best Case Complexity -** It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of insertion sort is **O(n)**.
- **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of insertion sort is **O(n^2)**.
- **Worst Case Complexity -** It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of insertion sort is **O(n^2)**.

1.2.3.2.2 Space complexity:

Space Complexity	O(1)
Stable	YES

- The space complexity of insertion sort is O(1). It is because, in insertion sort, an extra variable is required for swapping.

1.2.3.3 Programming implementation of Insertion sort:

```
public class InsertionSort

{

   void insert(int a[])
```

```
{
    int i, j, temp;
    int n = a.length;
    for (i = 1; i < n; i++) {
        temp = a[i];
        j = i - 1;

        while(j>=0 && temp <= a[j])  /* Move the elements greater than temp to one position
ahead from their current position*/
        {
            a[j+1] = a[j];
            j = j-1;
        }
        a[j+1] = temp;
    }
}

void printArr(int a[])
{
    int i;

    int n = a.length;

    for (i = 0; i < n; i++)

    System.out.print(a[i] + " ");

}



    public static void main(String[] args) {

    int a[] = { 92, 50, 5, 20, 11, 22 };

    InsertionSort i1 = new InsertionSort();

    System.out.println("\nBefore sorting array elements are - ");

    i1.printArr(a);

    i1.insert(a);

    System.out.println("\n\nAfter sorting array elements are - ");

    i1.printArr(a);

    System.out.println();

    }

}
```

Output:

```
Array before sorting: 10 9 8 7 6 5 4 3 2 1
Insertion sort starting...

After pass 1 : [9, 10, 8, 7, 6, 5, 4, 3, 2, 1]
After pass 2 : [8, 9, 10, 7, 6, 5, 4, 3, 2, 1]
After pass 3 : [7, 8, 9, 10, 6, 5, 4, 3, 2, 1]
After pass 4 : [6, 7, 8, 9, 10, 5, 4, 3, 2, 1]
After pass 5 : [5, 6, 7, 8, 9, 10, 4, 3, 2, 1]
After pass 6 : [4, 5, 6, 7, 8, 9, 10, 3, 2, 1]
After pass 7 : [3, 4, 5, 6, 7, 8, 9, 10, 2, 1]
After pass 8 : [2, 3, 4, 5, 6, 7, 8, 9, 10, 1]
After pass 9 : [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Total number of steps: 45
Array after sorting: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

1.2.4 Quick sort:

In this article, we will discuss the Quicksort Algorithm. The working procedure of Quicksort is also simple. This article will be very helpful and interesting to students as they might face quicksort as a question in their examinations. So, it is important to discuss the topic.

Sorting is a way of arranging items in a systematic manner. Quicksort is the widely used sorting algorithm that makes **n log n** comparisons in average case for sorting an array of n elements. It is a faster and highly efficient sorting algorithm. This algorithm follows the divide and conquer approach. Divide and conquer is a technique of breaking down the algorithms into subproblems, then solving the subproblems, and combining the results back together to solve the original problem.

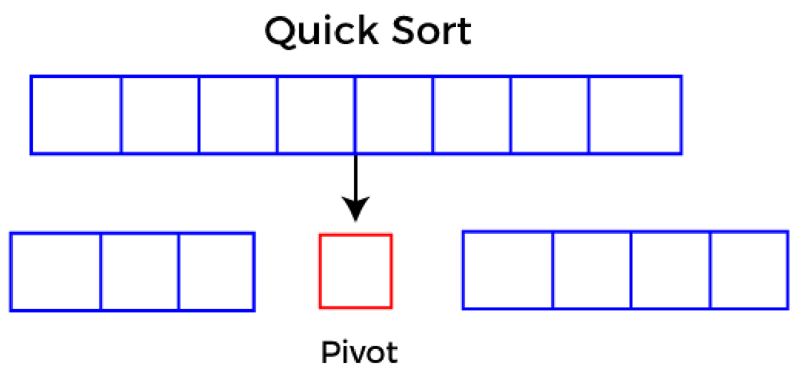
Divide: In Divide, first pick a pivot element. After that, partition or rearrange the array into two sub-arrays such that each element in the left sub-array is less than or equal to the pivot element and each element in the right sub-array is larger than the pivot element.

Conquer: Recursively, sort two subarrays with Quicksort.

Combine: Combine the already sorted array.

Quicksort picks an element as pivot, and then it partitions the given array around the picked pivot element. In quick sort, a large array is divided into two arrays in which one holds values that are smaller than the specified value (Pivot), and another array holds the values that are greater than the pivot.

After that, left and right sub-arrays are also partitioned using the same approach. It will continue until the single element remains in the sub-array.



Choosing the pivot

Picking a good pivot is necessary for the fast implementation of quicksort. However, it is typical to determine a good pivot. Some of the ways of choosing a pivot are as follows -

- Pivot can be random, i.e. select the random pivot from the given array.
- Pivot can either be the rightmost element of the leftmost element of the given array.
- Select median as the pivot element.

The pseudo code for quick sort agorithm:

```
// low  –> Starting index,  high  –> Ending index
```

```
quickSort(arr[], low, high) {
    if (low < high) {

        // pi is partitioning index, arr[pi] is now at right place
        pi = partition(arr, low, high);
        quickSort(arr, low, pi – 1);  // Before pi
        quickSort(arr, pi + 1, high); // After pi
    }
}
```

The pseudo code for partition agorithm:

The partition algorithm rearranges the sub-arrays in a place.

/* This function takes first element as pivot, places the pivot element at its correct position in sorted array, and places all smaller (smaller than or equal to pivot) to left of pivot and all greater elements to right of pivot */

```
partition (arr[], low, high) {
    // first element as pivot
    pivot = arr[low]
    k = high
    for (i = high; i > low; i–) {
        if (arr[i] > pivot){
            swap arr[i] and arr[k];
            k–;
        }
    }
    swap arr[k] and arr[low]
    return k;
}
```

Illustration of partition algorithm

Now, let's see the working of the Quicksort Algorithm.

To understand the working of quick sort, let's take an unsorted array. It will make the concept more clear and understandable.

Illustration of partition() :

Consider: arr[] = { 7,   6,   10,   5,   9,   2,   1,   15,   7 }

First Partition: low = 0, high = 8, pivot = arr[low] = 7

Initialize index of right most element k = high = 8.

Traverse from i = high to low:

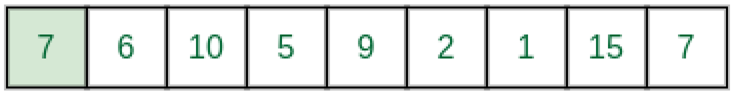
if arr[i] is greater than pivot:

Swap arr[i] and arr[k].

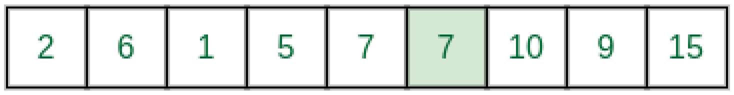
Decrement k;

At the end swap arr[low] and arr[k].

Now the correct position of pivot is index 5



Partition around the first element 7



Correct position for 7

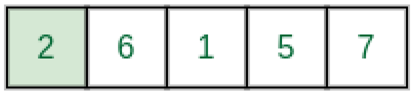
First partition

First partition

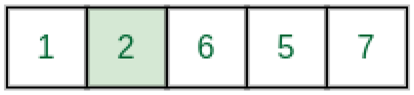
Second Partition: low = 0, high = 4, pivot = arr[low] = 2

Similarly initialize k = high = 4;

The correct position of 2 becomes index 1. And the left part is only one element and the right part has {6, 5, 7}.



Partition around the first element 2

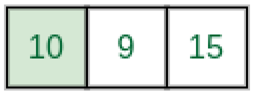


Correct position of 2

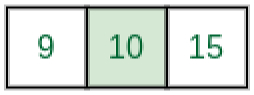
On the other hand partition happens on the segment [6, 8] i.e., the array {10, 9, 15}.

Here low = 6, high = 8, pivot = 10 and k = 8.

The correct position of 10 becomes index 7 and the right and left part both has only one element.



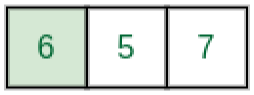
Partition around the first element 10



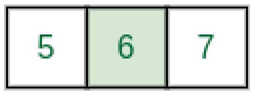
Correct position of 10

Third partition: Here partition the segment {6, 5, 7}. The low = 2, high = 4, pivot = 6 and k = 4.

If the same process is applied, we get correct position of 6 as index 3 and the left and the right part is having only one element.

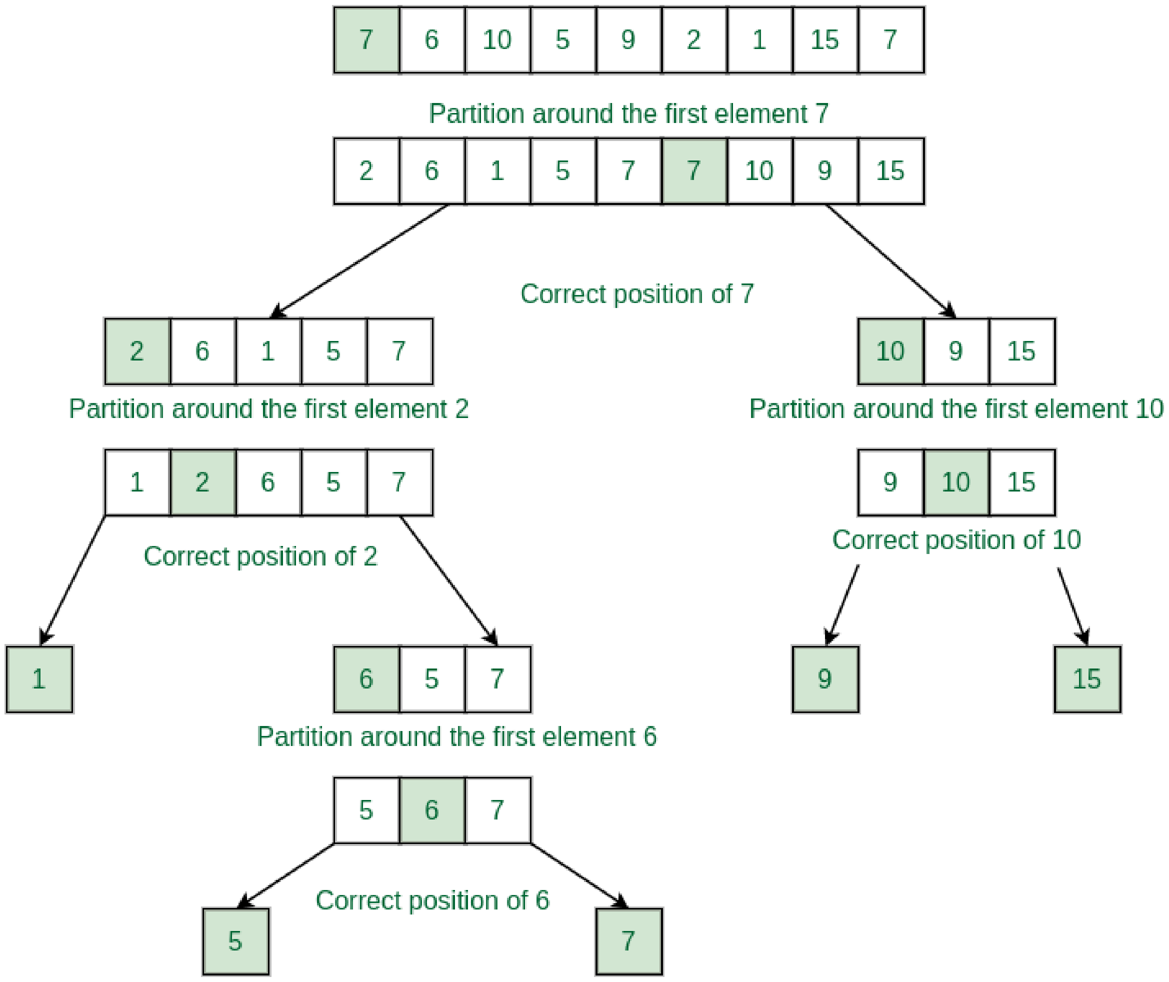


Partition around the first element 10



Correct position of 6

The total array becomes sorted in this way. Check the below image for the recursion tree



Follow the below steps to implement the approach.

Use a recursive function (say quickSort) to initialize the function.

Call the partition function to partition the array and inside the partition function do the following

Take the first element as pivot and initialize and iterator k = high.

Iterate in a for loop from i = high to low+1:

If arr[i] is greater than pivot then swap arr[i] and arr[k] and decrement k.

After the iteration is swap the pivot with arr[k].

Return k-1 as the point of partition.

Now recursively call quickSort for the left half and right half of the partition index.

Implementation of the above approach.

Quicksort complexity

Now, let's see the time complexity of quicksort in best case, average case, and in worst case. We will also see the space complexity of quicksort.

1. Time Complexity

Case	Time Complexity
Best Case	O(n*logn)
Average Case	O(n*logn)
Worst Case	$O(n^2)$

- Best Case Complexity - In Quicksort, the best-case occurs when the pivot element is the middle element or near to the middle element. The best-case time complexity of quicksort is O(n*logn).
- Average Case Complexity - It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of quicksort is O(n*logn).
- Worst Case Complexity - In quicksort, the worst case occurs when the pivot element is either the greatest or smallest element. Suppose, if the pivot element is always the last element of the array, the worst case would occur when the given array is sorted already in ascending or descending order. The worst-case time complexity of quicksort is O(n^2).

Though the worst-case complexity of quicksort is more than other sorting algorithms such as Merge sort and Heap sort, still it is faster in practice. Worst case in quick sort rarely occurs because by changing the choice of pivot, it can be implemented in different ways. Worst case in quicksort can be avoided by choosing the right pivot element.

2. Space Complexity

Space Complexity	O(n*logn)
Stable	NO

- The space complexity of quicksort is O(n*logn).

Implementation of quicksort

```
public class QuickSort {


    int partition(int arr[], int low, int high)

    {

        // First element as pivot

        int pivot = arr[low];

        int st = low;

        int end

            = high;

        int k = high;

        for (int i = high; i > low; i--) {

            if (arr[i] > pivot)

                swap(arr, i, k--);

        }

        swap(arr, low, k);


        return k;
    }


    public static void swap(int[] arr, int i, int j)

    {

        int temp = arr[i];

        arr[i] = arr[j];

        arr[j] = temp;

    }


    void quickSort(int arr[], int low, int high)

    {

        if (low < high) {

            int idx = partition(arr, low, high);

            quickSort(arr, low, idx - 1);

            quickSort(arr, idx + 1, high);

        }

    }


    void printArray(int arr[], int size)
```

```
{
    int i;
    for (i = 0; i < size; i++)
        System.out.print(arr[i] + " ");
    System.out.println();
}

public static void main(String args[])
{
    int arr[] = { 7, 6, 10, 5, 9, 2, 1, 15, 7 };
    int n = arr.length;

    QuickSort ob = new QuickSort();

    System.out.println("Unsorted array");
    ob.printArray(arr, n);
    ob.quickSort(arr, 0, n - 1);

    System.out.println("Sorted array");
```

```
        ob.printArray(arr, n);

    }

}
```

Output

```
Unsorted array

7 6 10 5 9 2 1 15 7

Sorted array

1 2 5 6 7 7 9 10 15
```

Merge Sort Algorithm

conquer approach. This article will be very helpful and interesting to students as they might face merge sort as a question in their examinations. In coding or technical interviews for software engineers, sorting algorithms are widely asked. So, it is important to discuss the topic.

Merge sort is similar to the quick sort algorithm as it uses the divide and conquer approach to sort the elements. It is one of the most popular and efficient sorting algorithm. It divides the given list into two equal halves, calls itself for the two halves and then merges the two sorted halves. We have to define the **merge()** function to perform the merging.

The sub-lists are divided again and again into halves until the list cannot be divided further. Then we combine the pair of one element lists into two-element lists, sorting them in the process. The sorted two-element pairs is merged into the four-element lists, and so on until we get the sorted list.

Now, let's see the algorithm of merge sort.

Algorithm

In the following algorithm, **arr** is the given array, **beg** is the starting element, and **end** is the last element of the array.

```
MERGE_SORT(arr, beg, end)

if beg < end
set mid = (beg + end)/2
MERGE_SORT(arr, beg, mid)
MERGE_SORT(arr, mid + 1, end)
MERGE (arr, beg, mid, end)
end of if

END MERGE_SORT
```

The important part of the merge sort is the **MERGE** function. This function performs the merging of two sorted sub-arrays that are **A[beg…mid]** and **A[mid+1…end]**, to build one sorted array **A[beg…end]**. So, the inputs of the **MERGE** function are **A[], beg, mid,** and **end**.

The implementation of the **MERGE** function is given as follows -

```
/* Function to merge the subarrays of a[] */
void merge(int a[], int beg, int mid, int end)
{
```

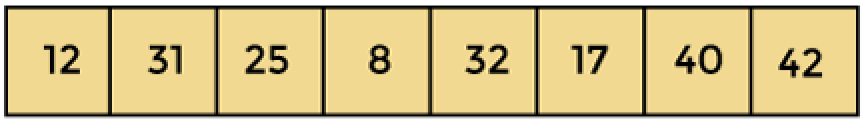
```
int i, j, k;
int n1 = mid - beg + 1;
int n2 = end - mid;

int LeftArray[n1], RightArray[n2]; //temporary arrays

/* copy data to temp arrays */
for (int i = 0; i < n1; i++)
LeftArray[i] = a[beg + i];
for (int j = 0; j < n2; j++)
RightArray[j] = a[mid + 1 + j];

i = 0, /* initial index of first sub-array */
j = 0; /* initial index of second sub-array */
k = beg;  /* initial index of merged sub-array */

while (i < n1 && j < n2)
{
    if(LeftArray[i] <= RightArray[j])
    {
        a[k] = LeftArray[i];
        i++;
    }
    else
    {
        a[k] = RightArray[j];
        j++;
    }
    k++;
}
while (i<n1)
{
    a[k] = LeftArray[i];
    i++;
    k++;
}

while (j<n2)
{
    a[k] = RightArray[j];
    j++;
    k++;
}
}
```

Working of Merge sort Algorithm

Now, let's see the working of merge sort Algorithm.

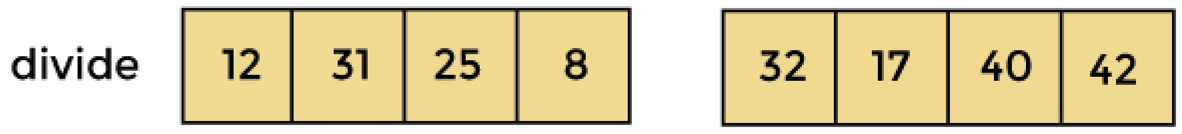
To understand the working of the merge sort algorithm, let's take an unsorted array. It will be easier to understand the merge sort via an example.

Let the elements of array are -

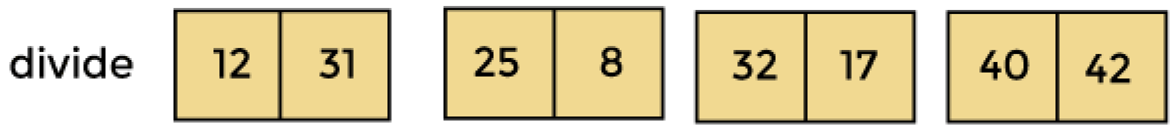


According to the merge sort, first divide the given array into two equal halves. Merge sort keeps dividing the list into equal parts until it cannot be further divided.

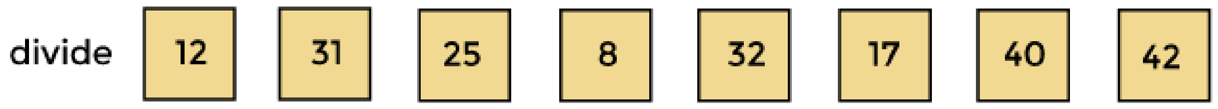
As there are eight elements in the given array, so it is divided into two arrays of size 4.



Now, again divide these two arrays into halves. As they are of size 4, so divide them into new arrays of size 2.



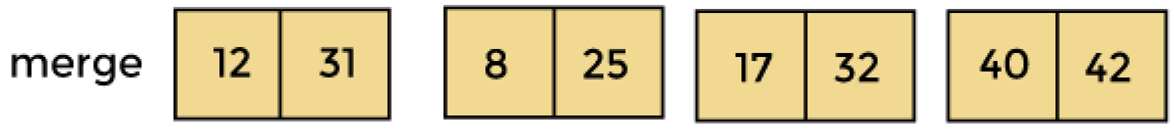
Now, again divide these arrays to get the atomic value that cannot be further divided.



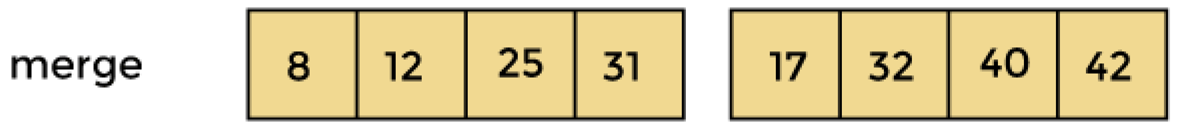
Now, combine them in the same manner they were broken.

In combining, first compare the element of each array and then combine them into another array in sorted order.

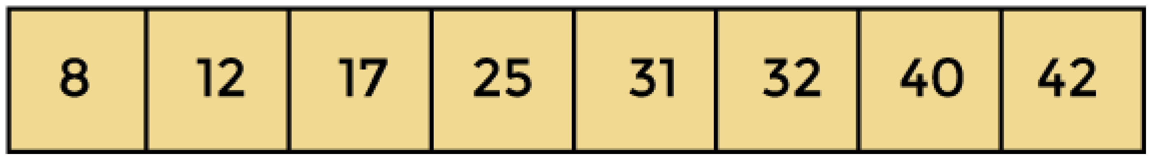
So, first compare 12 and 31, both are in sorted positions. Then compare 25 and 8, and in the list of two values, put 8 first followed by 25. Then compare 32 and 17, sort them and put 17 first followed by 32. After that, compare 40 and 42, and place them sequentially.



In the next iteration of combining, now compare the arrays with two data values and merge them into an array of found values in sorted order.



Now, there is a final merging of the arrays. After the final merging of above arrays, the array will look like -



Now, the array is completely sorted.

Merge sort complexity

Now, let's see the time complexity of merge sort in best case, average case, and in worst case. We will also see the space complexity of the merge sort.

1. Time Complexity

Case	Time Complexity
Best Case	O(n*logn)
Average Case	O(n*logn)
Worst Case	O(n*logn)

- **Best Case Complexity -** It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of merge sort is **O(n*logn)**.
- **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of merge sort is **O(n*logn)**.
- **Worst Case Complexity -** It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of merge sort is **O(n*logn)**.

2. Space Complexity

Space Complexity	O(n)
Stable	YES

- The space complexity of merge sort is O(n). It is because, in merge sort, an extra variable is required for swapping.

Implementation of merge sort

```
public class Main {

void merge(int a[], int beg, int mid, int end)

{

    int i, j, k;
    int n1 = mid - beg + 1;

    int n2 = end - mid;

   /* temporary Arrays */

        int LeftArray[] = new int[n1];

        int RightArray[] = new int[n2];

    /* copy data to temp arrays */

    for (i = 0; i < n1; i++)

    LeftArray[i] = a[beg + i];

    for (j = 0; j < n2; j++)

    RightArray[j] = a[mid + 1 + j];

    i = 0; /* initial index of first sub-array */

    j = 0; /* initial index of second sub-array */

    k = beg;  /* initial index of merged sub-array */

while (i < n1 && j < n2)

    {

        if(LeftArray[i] <= RightArray[j])

        {

            a[k] = LeftArray[i];

            i++;
        }
        else
        {
            a[k] = RightArray[j];
            j++;
        }
        k++;
    }
    while (i<n1)
    {
        a[k] = LeftArray[i];
        i++;
        k++;
    }
while (j<n2)
    {
        a[k] = RightArray[j];
        j++;
        k++;
    }

}


void mergeSort(int a[], int beg, int end)

{

    if (beg < end)

    {

        int mid = (beg + end) / 2;

        mergeSort(a, beg, mid);

        mergeSort(a, mid + 1, end);

        merge(a, beg, mid, end);

    }

}

/* Function to print the array */

void printArray(int a[], int n)

{

    int i;

    for (i = 0; i < n; i++)

        System.out.print(a[i] + " ");
    }

    public static void main(String args[])

    {

        int a[] = { 11, 30, 24, 7, 31, 16, 39, 41 };

        int n = a.length;

        Main m1 = new Main();

        System.out.println("\nBefore sorting array elements are - ");

        m1.printArray(a, n);

        m1.mergeSort(a, 0, n - 1);

        System.out.println("\nAfter sorting array elements are - ");

        m1.printArray(a, n);

        System.out.println("");

    } }
```

Output:

```
Before sorting array elements are -
11 30 24 7 31 16 39 41
After sorting array elements are -
7 11 16 24 30 31 39 41
```