Lecture Notes

Divide and Conquer

So, in this session we dealt with various algorithms for a very important technique called searching. Before getting into the algorithms, we stated the importance of having a time efficient algorithm for searching where we discussed the Facebook example. You learnt that Facebook despite having a database of billions of users, takes little time to search for an email id the user enters. An efficient algorithm makes it possible. Then we dived into the following search algorithms

Linear Search

In Linear Search, you learnt that the algorithm compares the key element with each and every element in the array, starting from the first index, till the comparison results in a match. So, if the key element lies at the end of the array having N elements, the algorithm takes N steps to search for the key. The following code was used as Java implementation of Linear Search.

```
public class LinearSearch {

    public static int linearSearch(int[] arr, int key) {

        int size = arr.length;
        for (int i = 0; i < size; i++) {
            if (arr[i] == key) {
                return i;
            }
        }
        //This is the default value if the key is not found in the array.
        return -1;
    }

    public static void main(String a[]) {

        int[] arr1 = {23, 45, 21, 55, 234, 1, 34, 90};
        int searchKey = 34;
        System.out.println("Key " + searchKey + " found at index: " + linearSearch(arr1, searchKey));
        int[] arr2 = {123, 445, 421, 595, 2134, 41, 304, 190};
        searchKey = 421;
        System.out.println("Key " + searchKey + " found at index: " + linearSearch(arr2, searchKey));
    }
}
```

As per the code, if the element is found at the ith index, it returned the index i. In case the element is not found in the array, it returned -1.

Analysis of Linear Search

You learnt that Linear Search takes N steps in worst case to search for an element in the given array of N elements. This is because if the element lies at the end of the array or it does not exist in the array, the algorithm has to go through all the elements of the array to find the match or to ensure that the element does not exist in the array. Hence the Big O for Linear Search came out to be O(N).

Divide and Conquer

In this segment, you learnt that dividing a problem into smaller sub problems and then solving these subproblems to come up with a solution greatly improves the efficiency of the solution. This is something similar to what you learnt in Computational thinking in decomposition segment. In the Divide and Conquer segment, we discussed a phonebook example. There we were supposed to search for a name Cheng in the phonebook. We started at the middle (dividing the sample space into 2 halves) where the names were starting with L. Since C comes before L, so we ignored all the names that were to the right of middle and started searching to the left of L. In case we were searching for a name that starts with T, we would have ignored all the names to the left of L and would have started searching to the right of L. This straight away reduces the sample search space to one half of the original and we do not need to waste time searching there. On the ideas of Divide and Conquer, we moved on to another search algorithm called Binary Search.

Binary Search

As discussed in the videos, the first and foremost requirement for Binary Search is a sorted array. Binary search can only be applied to a sorted array.

In Binary Search, you learnt that the algorithm first compares the key to the middle element. If the key< middle element, the algorithm moves to the left of the middle and if key>middle element the algorithm moves to the right of the middle discarding the other half. In case key = middle element the algorithm returns the index of the middle element itself. Once the algorithm moves to the left or the right of the middle, it again computes the middle index of the corresponding half of the array. It again compares the key with the new middle element and moves to the left or right of the middle depending upon the comparison of the key and the middle element. The process continues till the match is found or the algorithm reaches the end of the array. Eg. If we were to search for the element 9 in the following array using Binary Search,

1	3	5	7	9	11

the algorithm first compares 9 with the middle element 5. Since 9>5, so it moves to the right and we are left with the following array.

7	9	11

It now compares 9 with the middle element of this array. This time the middle element is 9 itself, so the match is found and the algorithm returns the index of this element.

Code for Binary Search

The following code was used for the Java implementation of Binary Search.

```
public class BinarySearch {


    public int binarySearch(int[] inputArr, int key) {
        int start = 0;

        int end = inputArr.length - 1;

        while (start <= end) {

            int mid = (start + end) / 2;

            if (key == inputArr[mid]) {

                return mid;

            }

            if (key < inputArr[mid]) {

                end = mid - 1;

            } else {

                start = mid + 1;

            }

        }

        return -1;

    }


    public static void main(String[] args) {


        BinarySearch mbs = new BinarySearch();

        int[] arr = {2, 4, 6, 8, 10, 12, 14, 16};

        System.out.println("Key 14's position: " + mbs.binarySearch(arr, 14));

        int[] arr1 = {6, 34, 78, 123, 432, 900};

        System.out.println("Key 432's position: " + mbs.binarySearch(arr1, 432));

    }

}
```

The first part of the code defines the binary search function. It takes in 2 arguments, the integer array and the key the user is looking to search for.

As evident from the code itself, it first computes the middle element by dividing the sum of first and last index by 2.

int mid = (start + end) / 2;

The code then compares the key with the element at middle index. And if this results in a match, the index of the middle element is returned.

```
if (key == inputArr[mid]) {
            return mid;
        }
```

If the element at the middle index is greater than the key, the algorithms moves to the left of the middle. This is equivalent to setting the end index to mid-1.

```
if (key < inputArr[mid]) {
            end = mid - 1;
        }
```

If the key is greater than the element at the middle, the algorithm moves to the right of the middle. This is equivalent to setting start index as mid+1.

```
else {
            start = mid + 1;
        }
```

In case the element is not found, it returns -1.

```
    return -1;
```

At last, in the main function two integer arrays are initialized. Also, the object of the class Binary Search is created to access the binarySearch function to which the integer array and the key are passed as arguments.

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

    BinarySearch mbs = new BinarySearch();
    int[] arr = {2, 4, 6, 8, 10, 12, 14, 16};
    System.out.println("Key 14's position: " + mbs.binarySearch(arr, 14));
    int[] arr1 = {6, 34, 78, 123, 432, 900};
    System.out.println("Key 432's position: " + mbs.binarySearch(arr1, 432));
  }
}
```

Analysis of Binary Search

They you learnt about the efficiency of Binary Search. The recursion equation came out to be T(n) = T(n/2) + 2 where T(n) represents the number of comparisons done in an array of n elements using binary search. The rationale behind T(n/2) in R.HS. is that in every next iteration, half of the elements are discarded. So, if the array has n elements initially, in the next iteration binary search will be applied on n/2 elements and so on. 2 is added to it because of 2 additional comparisons done in every iteration.

1) The comparison to check if the middle element is equal to key or not.
2) If not equal, we need a comparison to check if the middle element is less than or greater than the key.

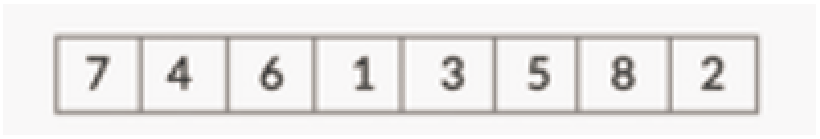
On solving this recursion equation, the Big O came out to be O(logn).

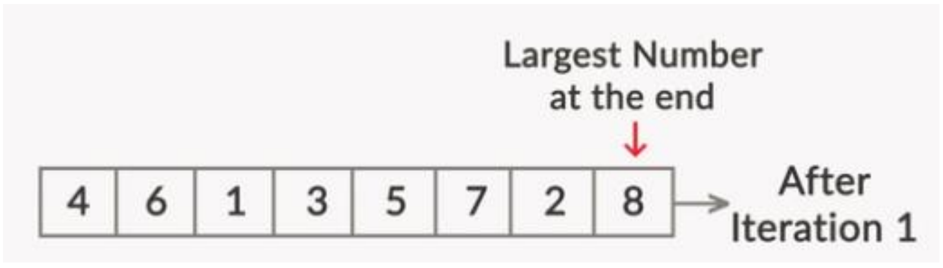
Session 2 – Sorting Algorithms -1

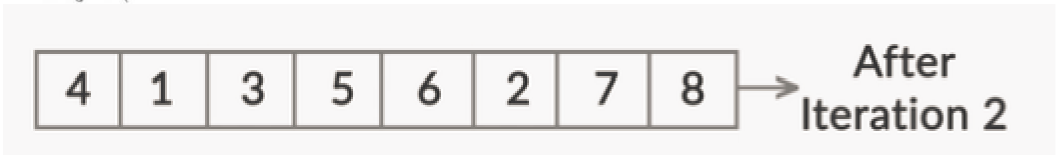
In this session you learnt about the sorting algorithms. You learnt that there are many real-life scenarios where sorting the data in certain order makes our lives a lot easier. Specifically, we discussed the Facebook example where you can sort your news feed based on the popularity. Also, on a shopping website the items can be sorted based on price, brands etc. In session 2 we discussed the sorting algorithms with a Big O efficiency of O(N^2). The first algorithm discussed in this session was bubble sort.

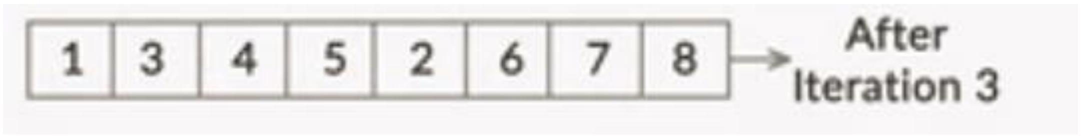
Bubble Sort

You started with the first two numbers in each iteration and did a comparison between them. When the numbers were not in order, you swapped them. The same steps were repeated for the second and third number, and so on. At the end of the first iteration, the largest number was pushed to the end of the array; at the end of the second iteration, the second largest number was pushed to the second last position, and so on. For the array used in the videos, it looked something like this.

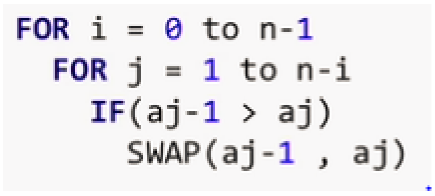








The following pseudo code was discussed for its java implementation.



```
FOR i = 0 to n-1
  FOR j = 1 to n-i
    IF(aj-1 > aj)
      SWAP(aj-1 , aj)
```

As per the pseudo code, for an array which is supposed to be sorted in an increasing order, if the number at (j-1) th index is greater than the number at the jth index, we swap them. The inner loop with j counter takes care of all the comparisons in each iteration and the outer loop corresponds to n-1 iteration required to sort the array.

One the pseudo code was discussed we moved on to the Java implementation of Bubble Sort. Following code was used by the Professor for this.

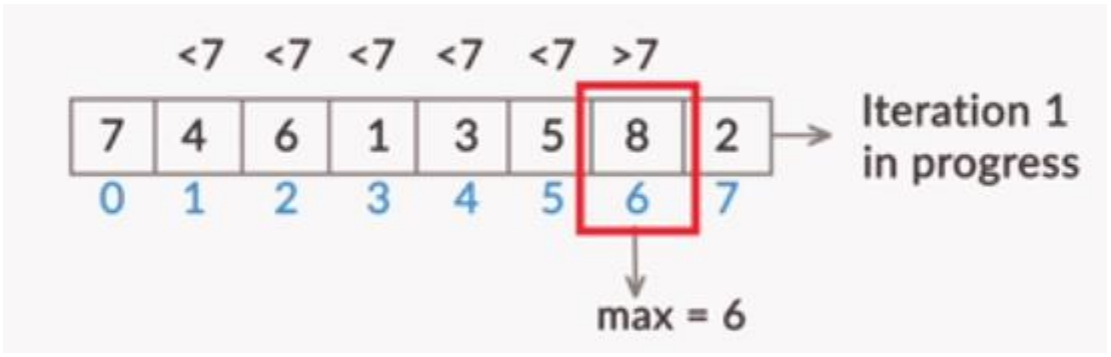
```
import java.util.Arrays;

public class BubbleSort {

   public static int[] sort(int[] numbers) {
       for (int i = 0; i < numbers.length; i++) {
           for (int j = 1; j < (numbers.length - i); j++) {
               if (numbers[j - 1] > numbers[j]) {
                   //swap elements
                   swap(j - 1, j, numbers);
               }
           }
       }
       return numbers; // returning the final sorted array
   }

   public static void swap(int i, int j, int[] array) {
       int temp = array[i];
       array[i] = array[j];
```

```
        array[j] = temp;
    }

    public static void main(String args[]) {
        int[] randomNumbers = {13, 3242, 23, 2351, 352, 3915, 123, 32, 1, 5, 0};
        int[] sortedNumbers;

        sortedNumbers = sort(randomNumbers); // bubble sort

        // print out the sorted numbers
        System.out.println(Arrays.toString(sortedNumbers));
    }
}
```

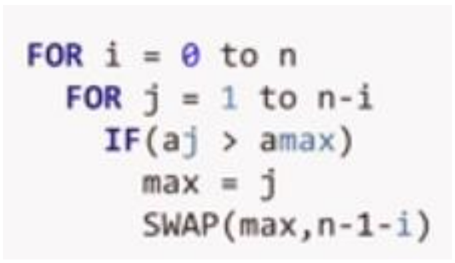
Once you learnt the Java implementation of Bubble Sort, we moved on to the analysis of this algorithm. As discussed by the Professor, bubble sort has a time complexity of O(N^2), where N is the number of elements in the array. In first iteration, it does (N-1) comparisons; in the second iteration it does (N-2) comparisons; in third, it does (N-3) comparisons. This continues for a total of N iterations. So, the total number of comparisons effectively become the sum of first (N-1) natural numbers.

Selection Sort

The next algorithm that you learnt was Selection Sort. You learnt that selection sort also, at the end of each iteration, pushes the next highest number to the end. However, this time it was done with fewer number of swaps. You just picked the highest number in each iteration and swapped it with the last, unsorted number. So, each iteration in the worst case needs only one swap. This is unlike bubble sort where you must compare and swap every two numbers every time they are out of order. It locates the highest number and puts it at the end of the sequence. In the next iteration, it locates the next highest number and puts it at the second last position of the array and so on.



The following pseudo code was discussed for its java implementation.



```
FOR i = 0 to n
  FOR j = 1 to n-i
    IF(aj > amax)
      max = j
      SWAP(max,n-1-i)
```

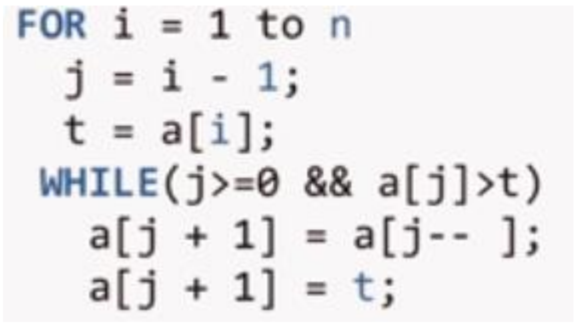
As per the pseudo code if the number at jth index is greater than the number at the current max index, we update the value of max with j. At the end of the iteration, we swap the number at the max index with the number at the (n-1-i)th position. The following code was used for its Java implementation.

```java
import java.util.Arrays;

public class SelectionSort {

   public static int[] sort(int[] numbers) {
       for (int i = 0; i < numbers.length - 1; i++) {

           int highestIndex = 0;

           // looping through the rest of the array to find a bigger element
           for (int j = 1; j < numbers.length - i; j++) {
               if (numbers[j] > numbers[highestIndex]) {
                   highestIndex = j; //bigger element found
               }
           }

           // swap the smallest element found with the ith element
           swap(highestIndex, numbers.length -1 - i, numbers);
       }
       return numbers;
   }

   public static void swap(int i, int j, int[] array) {
       int temp = array[i];
       array[i] = array[j];
       array[j] = temp;
   }

   public static void main(String args[]) {
       int[] randomNumbers = {7,4,6,1,3,5,8,2};
       int [] sortedNumbers = sort(randomNumbers); // selection sort

       // print out the sorted numbers
       System.out.println(Arrays.toString(sortedNumbers));
   }
}
```

Post that, you learnt about the efficiency of selection sort. So, as far as Big O goes, selection sort also turned out to have a time complexity of O(N^2). Remember that you learnt that it performs fewer swaps than bubble sort. So ideally, it should run a bit more efficiently than bubble sort. We then used the cards demonstration to solve this. There you learnt that the actual number of steps a selection sort takes is **N^2/2**. But **N^2/2** also falls into the time complexity of O(N^2). Therefore, despite selection sort being more efficient than bubble sort, it falls into the same Big O category as does bubble sort.

Insertion Sort

The next algorithm that you learnt was Insertion sort. As discussed by our professor, in insertion sort, you compare an element with the element to its left. If the element to its left is greater, you should shift the greater element to the right by one position and the smaller one to the left. In the next iteration, you need to compare this smaller element with the one to its left, and shift it if the element to the left is greater. You stop when you find that the element to the left is smaller than the element you are comparing with. Following pseudo code was discussed for its java implementation.



```
FOR i = 1 to n
  j = i - 1;
  t = a[i];
 WHILE(j>=0 && a[j]>t)
   a[j + 1] = a[j-- ];
   a[j + 1] = t;
```

As per the pseudo code, if a[i-1] > a[i], we swap them. The two lines written inside the while loop takes care of this swap. Please refer to the In-Video question asked in the video for the explanation of this pseudo code. The following code was used for the Java implementation of Insertion Sort.

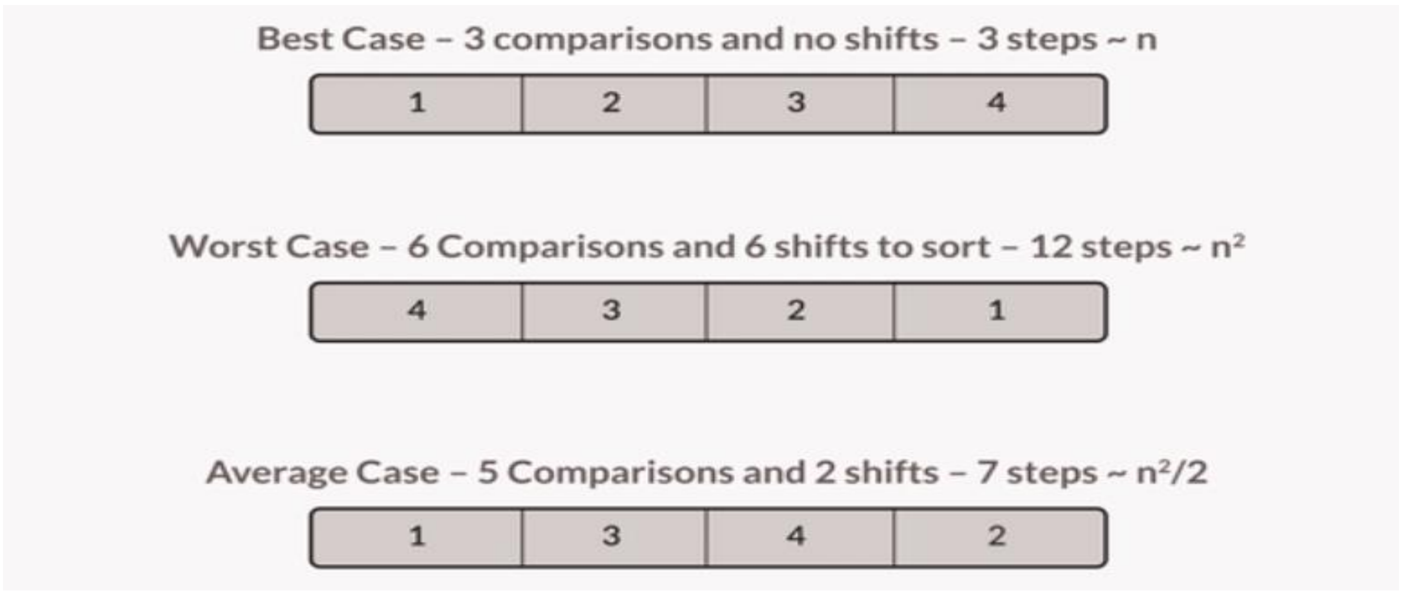
```
import java.util.Arrays;

public class InsertionSort {

   public static int[] sort(int[] numbers) {
       // going through each number in int[] numbers
       for (int i = 1; i < numbers.length; i++) {
           int j = i - 1;
           int t = numbers[i];
           while (j >= 0 && numbers[j] > t) {
               numbers[j + 1] = numbers[j--];
           }
```

```
            numbers[j + 1] = t;

        }
        return numbers; // returning the final sorted array
    }


    public static void main(String args[]) {
        int[] randomNumbers = {10, 5, 2, 3, 4, 98, 67};
        int[] sortedNumbers;



        sortedNumbers = sort(randomNumbers); // insertion sort

        // print out the sorted numbers
        System.out.println(Arrays.toString(sortedNumbers));
    }
 }
```

As far as analysis of Insertion sort is concerned, you learnt that it also has an efficiency of O(N^2). However, it follows O(N) in the best case. So, if the array is almost sorted, i.e. it is nearer to the best case than the worst case, the insertion sort will have a time complexity somewhere between O(N^2) and O(N). It will be nearer to O(N) if the array is almost sorted, and nearer to O(N^2) if the array is arranged in the opposite order of our preference. For example, consider an array where all elements are in descending order, and you want to sort it in ascending order.

Insertion Sort vs Selection Sort

Next, we compared insertion sort and selection sort. In this segment you learnt that when two algorithms fall into the same Big O category, we need to think beyond Big O to choose more efficient of the two. The kind of data set we are dealing with must be considered. The following example was discussed to communicate the idea.


Best Case – 3 comparisons and no shifts – 3 steps ~ n
1 2 3 4
Worst Case – 6 Comparisons and 6 shifts to sort – 12 steps ~ n²
4 3 2 1
Average Case – 5 Comparisons and 2 shifts – 7 steps ~ n²/2
1 3 4 2

Merge Sort

Then you moved on to the next session where you learnt about sorting algorithms following the efficiency of O(N logN). The first one that was discussed was Merge Sort. There you learnt that you keep on divind the array into halves until you reach one single element. After that these arrays, having one element each, are merged back in the required order. Following was the code that was discussed for the algorithm.

```
import java.util.Arrays;


public class MergeSort {

    public static int[] sort(int[] randomNumbers) {

        return mergeSort(randomNumbers, 0, randomNumbers.length - 1);

    }

    public static int[] mergeSort(int[] numbers, int first, int last) { if (first <
        last) {

            int mid = (first + last) / 2;

            mergeSort(numbers, first, mid);

            mergeSort(numbers, mid + 1, last);

            merge(numbers, first, mid, last);

        }

        return numbers;

    }

    private static int[] merge(int[] numbers, int i, int m, int j) { int l = i;
        //inital index of first subarray

        int r = m + 1; // initial index of second subarray int k = 0;
        // initial index of merged array int[] t = new int
        [numbers.length];

        while (l <= m && r <= j) {

            if (numbers[l] <= numbers[r]) {

                t[k] = numbers[l];

                k++;

                l++;

            } else {

                t[k] = numbers[r];

                k++;

                r++;
            }
        }


        // Copy the remaining elements on left half , if there are any while (l
        <= m) {
            t[k] = numbers[l];
            k++;
            l++;

        }


        // Copy the remaining elements on right half , if there are any while (r
        <= j) {
            t[k] = numbers[r];
            k++;
            r++;
        }
        // Copy the remaining elements from array t back the numbers array l = i;
        k = 0;
        while (l <= j) {
            numbers[l] = t[k];
            l++;
            k++;
        }
        return numbers;
    }
    public static void main(String args[]) {
        int[] randomNumbers = {13, 3242, 23, 2351, 352, 3915, 123, 32, 67,5, 9};
        int[] sortedNumbers;
        sortedNumbers = sort(randomNumbers); // Merge Sort

        // print out the sorted numbers
        System.out.println(Arrays.toString(sortedNumbers));
    }

}
```

Then you learnt about the efficiency of the merge sort algorithm. On analysis it was found that merge sort follows O(NlogN).

Master’s Theorem

Next you learnt about Master’s theorem. You learn that The master theorem helps in calculating the time complexity of algorithms that use recursion.

So, we discussed the following cases in the master theorem:

Case 1: $T(n) = \quad (n_d)$ if $a<b_d$

Case 2: $T(n) = \quad (n_d \log n)$ if $a = b_d$

Case 3 $T(n) = \quad (n_{loga})$ if $a> b_d$

Here, a, b and d are defined as follows:

$T(n) = aT(n/b) + f(n)$

$T(1) = c$

Here, $a > 1$, $b > 2$, $c > 0$

$(n) = n_d$

Quick Sort

The last sorting algorithm that was discussed was Quick Sort. You learnt that quicksort uses a pivot element, which is chosen randomly from the array, to partition the array such that all the elements smaller than or equal to the pivot are shifted to its left-hand side, and all the elements greater than the pivot are shifted to its right-hand side. Now, to do this partition, you needed a partition function. Therefore, you are using ‘l’ (the left pointer) and ‘r’ (the right pointer) to scan through the elements on the left and the right sides of the pivot, identify elements that are out of place, and swap those elements with the ultimate goal of arranging all the elements smaller than the pivot to its left and all the elements that are larger than the pivot to its right. You used two pointers, namely, ‘l’ and ‘r’. The first pointer ‘l’ was used for the indices located on the left side of the pivot, and the second pointer ‘r’ was used for indices located on the right side of the pivot. We compared the element at ‘l’ with the pivot and kept incrementing the value of ‘l’ by 1 until an element greater than the pivot was found. If it was greater than the pivot, you stopped and started comparing the element at r. Similar to ‘l’, you kept decrementing the value of ‘r’ until an element smaller than the pivot was found. If such an element is found, you stopped, and at this point, the elements at ‘l’ and ‘r’ were swapped. Following code was discussed for Quick Sort.

```
import java.util.Arrays;
import java.util.Random;



public class QuickSort {
```

```
static Random random = new Random();


public static int[] sort(int[] numbers) { // let's sort
    numbers using quick sort quickSort(numbers, 0,
    numbers.length - 1); return numbers;

}



public static void quickSort(int[] numbers, int first, int last) { if (first <
    last) {
        // select a pivot point

        int pivotIndex = first + random.nextInt(last - first + 1); int pivot
        = numbers[pivotIndex];

        int k = partition(numbers, first, last, pivot);


        // recursively sort the elements to the left of the pivot
        quickSort(numbers, first, k);


        // recursively sort the elements to the right of the pivot
        quickSort(numbers, k + 1, last);

    }


}



public static int partition(int[] numbers, int first, int last, int pivot) {

    int l = first;

    int r = last;


    while (l <= r) {

        // In each iteration, we will identify a number


        // from left side which is greater than the pivot // value,
        and also we will identify a number from // right side which
        is less then the pivot value.

        // Once the search is done, then we exchange both numbers.


        while (l <= r && numbers[l] <= pivot) { l++;


        }



        while (l <= r && numbers[r] > pivot) {

            r--;


        }
```

```
            if (l <= r) {
                exchangeNumbers(numbers, l, r);
                //move index to next position on both sides
                l++;

                r--;

            }

        }


        return l - 1;
    }


    public static void exchangeNumbers(int[] numbers, int i, int j) { int temp =
        numbers[i];
        numbers[i] = numbers[j];

        numbers[j] = temp;


             // exchange numbers using XOR, which doesn't require a temp
variable

    }


    public static void main(String args[]) {
        int[] randomNumbers = {13, 3242, 23, 2351, 352, 3915, 123, 32, 67,
5, 9};
        int[] sortedNumbers;


        sortedNumbers = sort(randomNumbers); // Quicksort


        // print out the sorted numbers
        System.out.println(Arrays.toString(sortedNumbers));

    }

}
```

Then we discussed the analysis of this algorithm. On analysis it was found to follow the time complexity of O(NlogN) for an average case scenario. However, you learnt that in worst case Quicksort follows O(N^2).

Comparison of Sorting Algorithms

In this segment we compared all the sorting algorithms you learnt so far. You learnt that bubble sort, selection sort, and insertion sort have a time complexity of O(N^2) in average and worst cases. So, generally they are not used unless we are very sure that the dataset is very close to the best case (where insertion sort has an O(N) time complexity). Both merge sort and quicksort has an O(N log N) time complexity. But quicksort has O(N^2) in the worst case. Also, merge sort is a stable sort, which quicksort is not. So, in case you need a stable sorting algorithm, the only sorting algorithm you can opt for is merge sort.