What is sorting?

Sorting is an algorithm that arranges the elements of a list in certain order [Either ascending or descending]. The output is a permutation or reordering of the input.

Why sorting is necessary?

Sorting is one of the important categories of algorithm in computer science. Sometimes sorting significantly reduces the complexity of a problem.

We can use sorting as a technique to reduce the search complexity. Great research went into this category of algorithms because of its importance. These algorithms are used in many computer algorithms [for example, searching elements], databases algorithm and many more.

Bubble Sort:

Bubble sort is the simplest sorting algorithm. It works by iterating the input array from the first element to last, comparing each pair of elements and swapping them if needed. Bubble sort continues its iterations until no more swaps are needed.

The algorithm gets its name from the way smaller elements “bubble to the one end of the list”.

Generally, insertion sort has better performance than bubble sort.

- **The only significant advantages that bubble sort has over other implementation is that it can detect whether the input list is already sorted or not.**

Email: aatish.azad7@gmail.com

```
public static int[] bubbleSort(int[] arr) {
    int temp = 0;
    for(int i=arr.length-1;i>=0;i--) {
        for(int j=0;j<=i-1;j++) {
            if(arr[j] > arr[j+1]) {
                temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
    return arr;
}
```

Bubble Sort Algorithms:

1. Run an Outer for loop with int i value equals to the length of the array -1. Condition is (i >= 0), If i is more that or equals to zero, decrement i by 1.
2. Inside the outer loop, there will be a inner loop with int j=0 and the condition is **j<=i-1**, j will be incremented by 1.
3. Inside the Inner loop:

   If (arr[i] > arr[j+1]){

   //swap

   }

The above algorithm takes $O(n^2)$ even in the best case. We can improve it by using one extra flag. No more swaps indicate the completion of sorting. If the list is already sorted, we can use this flag to skip the remaining passes.

```java
/**
 * Optimized bubble sort which can detect if the given
 * array is already sorted
 * @param arr
 * @return
 */
public static int[] bubbleSortOptimized(int[] arr) {
    int temp = 0;
    boolean noSwap = true;
    for(int i=arr.length-1;i>=0;i--) {
        noSwap = true;
        for(int j=0;j<=i-1;j++) {
            if(arr[j] > arr[j+1]) {
                temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
                noSwap = false;;
            }
        }
        if(noSwap) {
            break;
        }
    }
    return arr;
}
```


Selection Sort

Selection Sort is similar to bubble sort, but instead of first placing large values into sorted position, it places small values into sorted position.

```java
public static int[] selectionSort(int arr[]) { //{6,2,7,2,1,7}

    int index = 0;
    for(int i=0;i<arr.length-1;i++) {
        int min = arr[i];
        for(int j=i;j<arr.length-1;j++) {
            if(arr[j+1] < min) {
                min = arr[j+1];
                index = j+1;
            }
        }
        if(arr[i] != min) {
            int temp = arr[i];
            arr[i] = arr[index];
            arr[index] = temp;
        }
        System.out.println(Arrays.toString(arr));
    }
    return arr;
}
```

Selection sort Sudo Code:

1. Store the first element as the smallest value you've seen as far.
2. Compare this item to the next item in the array until you find a smaller number.
3. If a smaller number is found. Designate that smaller number to be the new "**minimum**" and continue until the end of the array.
4. If the minimum is not the value (index) you initially began with, swap the two values.
5. Repeat this with the next element until array is sorted.

Advantages:

- Easy to implement.
- In-Place sort (requires no additional storage space)

Disadvantages:

- Doesn't scale well. O(n2)

Insertion Sort

Insertion sort is a simple and efficient comparison sort. In this algorithm each iteration removes an element from the input data and inserts it into the correct position in the list being sorted. The choice of the element being removed from the input is random and this process is repeated until all input elements have gone through.

```
public static int[] insertionSsort(int[] arr){
    if(arr.length == 2) {
        if(arr[0] > arr[1]) {
            int temp = arr[0];
            arr[0] = arr[1];
            arr[1] = temp;
        }
    }
    int i,j,value;
    for(i=2;i<=arr.length-1;i++) {
        value = arr[i];
        j = i;
        while(arr[j-1] > value && j>1) {
            arr[j] = arr[j-1];
            j--;
        }
        arr[j] = value;
    }
    return arr;
}
```

Advantages:

- Simple implementation
- Efficient for small data
- Adaptive: If the input list is presorted [May not be completely] then insertion sort takes O(n + d), where d is the number of inversion.
- Practically more efficient than selection and bubble sort even though all of them have $O(n^2)$ worst case complexity.
- Stable: Maintains relative order of input data if the keys are same.

- In-place: It removes only a constant amount O(1) of additional memory space.
- Online: Insertion sort can sort the list as it receives it.

Note:

- Bubble Sort, Selection Sort and Insertion Sort are called as elementary sorting algorithm, also they are often called as quadratic sorting algorithm, because of their Big O of **$O(n^2)$.**
- Bubble sort and Insertion sort works well in case of almost sorted Data.
- Space complexity of bubble sort, Selection sort and Insertion sort is **O(1).** Hence, called as in-place sorting.
- When data sets are small these algorithm works well.

Merge Sort

Merge sort is an example of divide and conquer.

Important Notes:

- Merging is the process of combining tow sorted files to make one bigger sorted file.
- Selection is the process of dividing a file two parts: k smallest elements and n-k largest elements.
- Selection and merging are opposite operations
    - Selection splits a list into two lists
    - Merging joins two files to make one file
- Merge sort is fast and stable
- Merge sort requires additional space proportional to the size of the input array.

- Merge sort is relatively simple to code and offers performance slightly below of quicksort.

In merge sort the input list is divided into two parts and these are solved recursively. After solving the sub problems they are merged by scanning the resultant sub problems.

Worst case time complexity: O(n log n)

Worst case space complexity: O(n)


```
public class MergeSort {
    private int[] array;
    private int[] tempMergArr;
    private int length;


    public void sort(int inputArr[]) {
        this.array = inputArr;
        this.length = inputArr.length;
        this.tempMergArr = new int[length];
        doMergeSort(0, length - 1);
    }

    private void doMergeSort(int lowerIndex, int higherIndex) {

        if (lowerIndex < higherIndex) {
            int middle = lowerIndex + (higherIndex - lowerIndex) / 2;
            // Below step sorts the left side of the array
            doMergeSort(lowerIndex, middle);
            // Below step sorts the right side of the array
            doMergeSort(middle + 1, higherIndex);
            // Now merge both sides
            mergeParts(lowerIndex, middle, higherIndex);
        }
    }
```

```
    private void mergeParts(int lowerIndex, int middle, int higherIndex) {

        int i = lowerIndex;
        int j = middle + 1;
        int k = lowerIndex;

        //copy of the original array into the temp array
        for (int z = lowerIndex; z <= higherIndex; z++) {
            tempMergArr[z] = array[z];
        }


        while (i <= middle && j <= higherIndex) {
            if (tempMergArr[i] <= tempMergArr[j]) {
                array[k] = tempMergArr[i];
                i++;
            } else {
                array[k] = tempMergArr[j];
                j++;
            }
            k++;
        }

        while (i <= middle) {
            array[k] = tempMergArr[i];
            k++;
            i++;
        }
    }

}
```

Quick Sort:

Quick sort is an example of divide and conquer algorithmic technique. It is also called as partition exchange sort. It uses recursive calls for sorting the elements. It is one of the famous algorithms comparison-based sorting algorithms.

Divide: The array arr[low...high] is partitioned into two non-empty sub arrays arr[low...q] and arr[q+1...high], such that each element of

arr[low...high] is less than or equal to each element of arr[q+1...high]. The index q is computed as part of this portioning procedure.

Conquer: The two sub arrays arr[low...q] and arr[q+1...high] are sorted by recursive calls to Quick sort.

Algorithm:

1. If there are one or no elements in the array to be sorted, return.
2. Pick an element in the array to serve as **"pivot"** point. (Usually the left-most element in the array is used.)
3. Split the array two parts – one with elements larger than the pivot and the other with elements smaller that the pivot.
4. Recursively repeat the algorithm for both halves of the original array.

```java
public class QuickSort {

    private int input[];
    private int length;

    public void sort(int[] numbers) {

        if (numbers == null || numbers.length == 0) {
            return;
        }
        this.input = numbers;
        length = numbers.length;
        quickSort(0, length - 1);
    }
    /**
     * This method implements in-place quicksort algorithm recursively.
     */
    private void quickSort(int low, int high) {
        int i = low;
        int j = high;

        // pivot is middle index
        int pivot = input[low + (high - low) / 2];

        // Divide into two arrays
        while (i <= j){
            /*
             * In each iteration, we will identify a number from left side which is
             * greater then the pivot value, and a number from right side which
             * is less then the pivot value. Once search is complete,
             * we can swap both numbers.
             */
            while (input[i] < pivot) {
                i++;
            }
            while (input[j] > pivot) {
                j--;
            }
            if (i <= j) {
                swap(i, j);
                // move index to next position on both sides
                i++;
                j--;
            }
        }

        // calls quickSort() method recursively
        if (low < j) {
            quickSort(low, j);
        }

        if (i < high) {
            quickSort(i, high);
        }
    }

    private void swap(int i, int j) {
        int temp = input[i];
        input[i] = input[j];
        input[j] = temp;
    }
}
```

Worst Case time complexity: $O(n^2)$

Email: aatish.azad7@gmail.com