**Sorting algorithm**

[**http://www.java2novice.com/java-sorting-algorithms/**](http://www.java2novice.com/java-sorting-algorithms/)

**Bubble sort :**

Bubble sort, also referred to as sinking sort, is a simple sorting algorithm that works by repeatedly stepping through the list to be sorted, comparing each pair of adjacent items and swapping them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted. The algorithm gets its name from the way smaller elements "bubble" to the top of the list. Because it only uses comparisons to operate on elements, it is a comparison sort. Although the algorithm is simple, most of the other sorting algorithms are more efficient for large lists.

public static void bubble\_srt(int array[]) {

        int n = array.length;

        int k;

        for (int m = n; m >= 0; m--) {

            for (int i = 0; i < n - 1; i++) {

                k = i + 1;

                if (array[i] > array[k]) {

                    swapNumbers(i, k, array);

                }

            }

            printNumbers(array);

        }

    }

**Selection sort:**

The selection sort is a combination of searching and sorting. During each pass, the unsorted element with the smallest (or largest) value is moved to its proper position in the array. The number of times the sort passes through the array is one less than the number of items in the array. In the selection sort, the inner loop finds the next smallest (or largest) value and the outer loop places that value into its proper location.

public static int[] doSelectionSort(int[] arr){

     for (int i = 0; i < arr.length - 1; i++)

        {

            int index = i;

            for (int j = i + 1; j < arr.length; j++)

                if (arr[j] < arr[index])

                    index = j;

            int smallerNumber = arr[index];

            arr[index] = arr[i];

            arr[i] = smallerNumber;

        }

        return arr;

    }

**Insertion sort:**

**Advantages of Insertion Sort:**

1) It is very simple.

2) It is very efficient for small data sets.

3) It is stable; i.e., it does not change the relative order of elements with equal keys.

4) In-place; i.e., only requires a constant amount O(1) of additional memory space.

Insertion sort iterates through the list by consuming one input element at each repetition, and growing a sorted output list. On a repetition, insertion sort removes one element from the input data, finds the location it belongs within the sorted list, and inserts it there. It repeats until no input elements remain.

public static int[] doInsertionSort(int[] input){

        int temp;

        for (int i = 1; i < input.length; i++) {

            for(int j = i ; j > 0 ; j--){

                if(input[j] < input[j-1]){

                    temp = input[j];

                    input[j] = input[j-1];

                    input[j-1] = temp;

                }

            }

        }

        return input;

    }

**Merge sort;**

**\*** Merge sort is a divide and conquer algorithm.

**Steps to implement Merge Sort:**

1) Divide the unsorted array into n partitions, each partition contains 1 element. Here the one element is considered as sorted.

2) Repeatedly merge partitioned units to produce new sublists until there is only 1 sublist remaining. This will be the sorted list at the end.

Merge sort is a fast, stable sorting routine with guaranteed O(n\*log(n)) efficiency. When sorting arrays, merge sort requires additional scratch space proportional to the size of the input array. Merge sort is relatively simple to code and offers performance typically only slightly below that of quicksort.

**Quick sort:**

**Steps to implement quick sort:**

1) Choose an element, called pivot, from the list or array. Generally pivot is the middle element of array.

2) Reorder the list so that all elements with values less than the pivot come before the pivot, and all elements with values greater than the pivot come after it (equal values can go either way). This is also known as partitioning. After partitioning the pivot is in its final position.

3) Recursively apply the above steps to the sub-list of elements with smaller values and separately the sub-list of elements with greater values. If the array contains only one element or zero elements then the array is sorted

**searching algorithm:**

1. A binary search or half-interval search algorithm finds the position of a specified value (the input "key") within a sorted array.
2. In each step, the algorithm compares the input key value with the key value of the middle element of the array. If the keys match, then a matching element has been found so its index, or position, is returned.
3. Otherwise, if the sought key is less than the middle element's key, then the algorithm repeats its action on the sub-array to the left of the middle element or, if the input key is greater, on the sub-array to the right.
4. If the remaining array to be searched is reduced to zero, then the key cannot be found in the array and a special "Not found" indication is returned.
5. Binary search requires a sorted collection. Also, binary searching can only be applied to a collection that allows random access (indexing).

public int binarySearch(int[] inputArr, int key) {

i nt 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;

    }