**Definition:** An Algorithm is a set of instructions to solve specific problem.

* **Sorting Algorithms of two types**
* Iterative algorithms
* Recursive algorithms
* **Iterative Algorithms**
* Selection Sort
* Bubble Sort
* Insertion Sort
* **Recursive Algorithms**
* Quick Sort
* Merge Sort
* Bucket Sort
* Count Sort
* Heap Sort
* **Asymptotic Notations** are languages that allow us to analyze an algorithm's running time by identifying its behavior as the input size for the algorithm increases. This is also known as an algorithm's growth rate.
* Worst case can be declared using **Big** **Oh** notation.
* Best case can be declared using **Big** **Theta** notation.
* Average case can be declared using **Big** **Omega** notation.
* Generally we focus mainly on worst-case scenarios while designing any system.
* **Selection Sort:** This sorting algorithm finds minimum value index in the every iteration and pushes that element to the left side.
* **Pseudo Code**
* **For i=0 TO <array.length-1>**
* **MVI=i;**
* **For j=i+1 TO array.length**
* **If (array [MVI]> array [j])**
* **MVI=j;**
* **End If**
* **End For**
* **Swap (array [mvi], array [i])**
* **End For**
* **Time Complexity:**  This Algorithm time complexity can be calculated as below
* For i=0🡺 it iterates (n-1) times. (n - (0+1))
* For i=1🡺 it iterates (n-2) times. (n - (1+1))
* For i=k 🡺 it iterates (n- (k+1)) times.
* For i=n🡺 (n- (n+1)) times. i.e., 1 time.
* (n-1)+(n-2)+(n-3)…….. +3+2+1= n (n-1)/2= n^2/2-n/2; n^2= O (n^2)
* **Worst case: O (n^2).**

**public** **static** **int**[] doSelectionSort(**int**[] array) {

**for** (**int** i = 0; i < array.length - 1; i++) {

**int** minValueIndex = i;

**for** (**int** j = i + 1; j < array.length; j++) {

**if** (array[j] < array[minValueIndex]) {

minValueIndex = j;

}

}

**final** **int** temp = array[minValueIndex];

array[minValueIndex] = array[i];

array[i] = temp;

}

**return** array

}

* **BubbleSort**: This Alogirthm swaps its subsequent element to the right side. It moves higest element in the array to the right most position on every iteration
* For i=<array.length-1> TO i=0
* For j=0 TO j=i
* If <array [j]> array [j+1]>
* Swap (array [j], array [j+1]
* End If
* End For
* End For
* Time Complexity: Time Complexity can be calculated as follows
* For i=n 🡺 iterates through (n-1) times
* For i=n-1 🡺 ierates throgh (n-1-1) times (n-2)
* For i=n-2 🡺 iterates through (n-3)
* For i= k 🡺 iterates through (k)
* For i=1 🡺 ierates through 1 time
* (n-1) +(n-2) + (n-3) +………..+ 3+2+1 =n(n-1)/2 = n^2/2- n/2= n^2=**O(n^2)**

**public** **static** **int**[] bubbleSort(**int**[] array) {

**final** **int** length = array.length;

**for** (**int** m = length; m >= 0; m--) {

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

**if** (array[i] > array[i + 1]) {

**final** **int** temp = array[i + 1];

array[i + 1] = array[i];

array[i] = temp;

}

}

}

**return** array;

}

* **Insertion Sort:** Insetion Sort will sort the elements in its smallest sub array and grows continuosly.
* For i=1 TO <array.length>
* For j=i TO j=0
* If <array [j-1]> array [j]>
* Swap (array [j-1], array [j])
* End If
* End For
* End For
* **Time Complexity:**
* For i=1 🡺 1 times
* For i=2 🡺 2 times
* For i=k 🡺 k times
* For i=n 🡺 n-1 times
* 1+ 2+ 3 + …… + n = n(n+1)/2 = n2= O(n2)
* **public** **static** **int**[] insertSort(**int**[] array) {
* // NOTE: it starts with second element
* **for** (**int** i = 1; i < array.length; i++) {
* **for** (**int** j = i; j > 0; j--) {
* **if** (array[j - 1] > array[j]) {
* **final** **int** temp = array[j];
* array[j] = array[j - 1];
* array[j - 1] = temp;
* }
* }
* }
* **return** array;
* }
* **QuickSort:**  This works on divide and conque priciple. And it is recursive algorithm. This is faster than linear algorithms above.
* **Pseudo Code:**
* **QuickSort (data [], startIndex, endIndex)**
* **Pivot= (start+end)/2;**
* **<< Move the elements to left, which are less than pivot>>**
* **<< Move the elements to right, which are greater than pivot>>**
* **QuickSort (data, start, pivot-1)**
* **QuickSort (data, pivot, end)**
* **Time Complexity:**
* **Worst case analysis**

The pivot is the smallest element

T(N) = T(N-1) + cN, N > 1

Telescoping:

T(N-1) = T(N-2) + c(N-1)

T(N-2) = T(N-3) + c(N-2)

T(N-3) = T(N-4) + c(N-3)

T(2) = T(1) + c.2

Add all equations:

T(N) + T(N-1) + T(N-2) + … + T(2) =

= T(N-1) + T(N-2) + … + T(2) + T(1) + c(N) + c(N-1) + c(N-2) + … + c.2

T(N) = T(1) + c times (the sum of 2 thru N) = T(1) + c(N(N+1)/2 -1) = **O(N2)**

**Best-case analysis:**

The pivot is in the middle

T(N) = 2T(N/2) + cN

Divide by N:

T(N) / N = T(N/2) / (N/2) + c

Telescoping:

T(N/2) / (N/2) = T(N/4) / (N/4) + c

T(N/4) / (N/4) = T(N/8) / (N/8) + c

……

T(2) / 2 = T(1) / (1) + c

Add all equations:

T(N) / N + T(N/2) / (N/2) + T(N/4) / (N/4) + …. + T(2) / 2 =

= (N/2) / (N/2) + T(N/4) / (N/4) + … + T(1) / (1) + c.logN

After crossing the equal terms:

T(N)/N = T(1) + cLogN

T(N) = N + NcLogN = **O(NlogN)**

* **MergeSort: MegeSort will divide array into child arrays till it reaches one element in the array. And merge those two small arrays. It call is recursive.**
* **MergeSort (data []){**
* **If (data.length<=1)**
* **Return data;**
* **LeftArray [] = data [dataLength/2];**
* **RightArray [] =data [data.length-LeftArray.length];**
* **<< Copy Elements into LeftArray>>**
* **<< Copy Elements into RightArray>>**
* **LeftArray []= MergeSort (leftArray);**
* **RightArray []= MergeSort (rightArray);**
* **Return merge (LeftArray, RightArray);**
* **}**
* **MergeSort2 (data []){**
* **If (data.length>1){**
* **Copy firstHalf data [] into a []**
* **Copy SecondHalf data [] into b []**
* **A []= MergeSort2 [a []);**
* **B []=MergeSort2 (b []);**
* **Return Merge (A, B);**
* **}**

**}**

* **Time Complexity of merge sort:**
* Assumption: N is a power of two.
* For N = 1: time is a constant (denoted by 1)
* Otherwise: time to mergesort N elements = time to mergesort N/2 elements plus  
  time to merge two arrays each N/2 elements.
* Time to merge two arrays each N/2 elements is linear, i.e. N
* Thus we have:
* (1) T(1) = 1
* (2) T(N) = 2T(N/2) + N
* Next we will solve this recurrence relation. First we divide (2) by N:
* (3) T(N) / N = T(N/2) / (N/2) + 1
* N is a power of two, so we can write
* (4) T(N/2) / (N/2) = T(N/4) / (N/4) +1
* (5) T(N/4) / (N/4) = T(N/8) / (N/8) +1
* (6) T(N/8) / (N/8) = T(N/16) / (N/16) +1
* (7) ……  
  (8) T(2) / 2 = T(1) / 1 + 1
* Now we add equations (3) through (8) : the sum of their left-hand sides   
  will be equal to the sum of their right-hand sides:
* T(N) / N + T(N/2) / (N/2) + T(N/4) / (N/4) + … + T(2)/2 =
* T(N/2) / (N/2) + T(N/4) / (N/4) + ….+ T(2) / 2 + T(1) / 1 + LogN
* (LogN is the sum of 1s in the right-hand sides)
* After crossing the equal term, we get
* (9) T(N)/N = T(1)/1 + LogN
* T(1) is 1, hence we obtain
* (10) T(N) = N + NlogN = O(NlogN)
* Hence the complexity of the MergeSort algorithm is **O (NlogN).**