Bubble Sort

```
BubbleSort ( Arr[], n)
{
    int  i, j, temp, x
    for( i= 0 to n-1)
    {
        x=0
        for ( j=0 to n-i -1)
        {
            if ( Arr[j] > Arr[j+1])
            {
                x=1
                temp = Arr[j]
                Arr [j ]= Arr[j+1]
                Arr[j+1]= temp
            }
        }
        if(x=0)
            break;
    }
}
```

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

Selection Sort

```
SelectionSort( Arr[], n)
{
    int  i, j, min, temp
    for( i = 0 to n)
    {
        min = i
        for ( j = i+1  to n)
        {
            if( Arr[j] < Arr[min])
                min = j
        }
        swap ( Arr[i]  and  Arr [min])
    }
}
```

Time Complexity

Worst Case: $O(n^2)$

Best Case: $O(n^2)$

Insertion Sort

```
InsertionSort( Arr[], n)
{
    int  i, j, x
    for  ( i=0  to n)
    {
        x = Arr[i]
        j = i
        while  ( j ≥ 0  and  Arr[j] > x)
        {
            Arr[j+1] = Arr[j]
            j--
        }
        Arr[j+1]  = x
    }
}
```

Time Complexity

Worst Case: $O(n^2)$

Best Case: $O(n)$

MERGE SORT

Time Complexity: $O(n \log n)$
(all cases)

Recurrence Relation: $T(n) = 2T(\frac{n}{2}) + n$

```
MergeSort( Arr[], l, r)
{
        if( l < r)                                  //base condition
        {
            mid = (l+r)/2
            MergeSort ( Arr[], l, mid)        //left sub-array
            MergeSort ( Arr[], mid+1, r)      //right sub-arr
            Merge( Arr[], l, mid, r)          //Merging both
        }
}

// Merging Logic
Merge ( Arr[], l, mid, r)
{
    n1 = mid - l + 1 , n2 = r - mid , i, j, K
    L[n1], R[n2]                  // temp. Left & Right sub-arrays
    for (i = 0 to n1)
            L[i] = Arr[l+i]
    for (j = 0 to n2)
            R[j] = Arr[mid+1 +j]
    i=0, j=0, K=l
    for (K= l to r)
    {   if (L[i] <= R[j])
        {   A[K] = L[i] , i++ }
        else { A[K] = R[j], j++ }
    } K++
}
```

Quick Sort

```
QuickSort( Arr[], l, r)
{       if ( l<r)
        {
            p= Partition(Arr[], l, r)   // pivot Index
            QuickSort (Arr[], l, p-1)
            QuickSort (Arr[], p+1, r)
        }
}

// Partitioning Logic

Partition( Arr[], l, r)
{
        pivot = Arr[r]
        i= l-1
        for( j= l to r-1)
        {       if ( Arr[j] < pivot)
                {   i++
                    swap Arr[i] and Arr[j]
                }
        } swap Arr[i+1] and Arr[r]
        return (i+1)            // pivot index
}
```

- Time Complexity: Best: $O(n \log n)$
  Average: $O(n \log n)$
  Worst: $O(n^2)$

Recurrence Relation:

Worst : $T(n) = T(n-1) + \Theta(n)$

Best: $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$

Searching

Linear search

```
LinearS( Arr[], n, Key)
{
    for ( i=0 to n)
    {     if (Arr[i] = Key)
                  return i  //found
    }   return (-1)      // not found
}
```

Time Complexity:

Best Case: $O(1)$

Worst Case: $O(n)$

Binary Search (Divide and Conquer)

```
BinaryS( Arr[], n, Key)
{
    low = 0, high = n-1
    while ( low ≤ high )
    {
        mid = (low + high) / 2
        if ( Arr[mid] = Key )
                return mid            // found
        else if ( Arr[mid] > Key )
                high = mid-1
        else
                low = mid+1
    }
    return (-1)          //not found
}
```

Time Complexity	Recurrence Relation
Worst Case: $O(\log n)$	$T(n) = T\left(\frac{n}{2}\right) + 1$
Best Case: $O(1)$	