TUTORIAL - 3

① Pseudocode for Linear Search

```
for(i=0 to n)
{   if ( arr[i] == value)
        //element found
}
```

②

```
void insertion (int arr[], int n)    // recursive
{
    if (n<=1)
      return
    insertion (arr, n-1);
    int nth = arr[n-1];
     int j = n-2;
    while ( j >=0 && arr[j] > nth)
      {   arr [j+1] = arr[j];
            j--;
      }
      arr [j+1] = nth;
}

for ( i=1 to n)
  {   key ← A[i];
        j ← i-1
    while ( j>=0 and A[j] > key)
        {  A[j+1] ← A[j]
             j ← j-1
        }
        A[j+1] ← key
}
```

Insertion sort is online sorting because it doesn't know the whole input, more input can be inserted with the insertion sorting is running

(3) Complexity

Name	Best	Worst	Average
Selection	$O(n^2)$	$O(n^2)$	$O(n^2)$
Bubble	$O(n)$	$O(n^2)$	$O(n^2)$
Insertion	$O(n)$	$O(n^2)$	$O(n^2)$
Heap	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Quick	$O(n \log n)$	$O(n^2)$	$O(n \log n)$
Merge	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$

Q4 =

Inplace sorting	Stable sorting	Online Sorting
Bubble	Merge	Insertion.
Selection	Bubble	
Insertion	Insertion	
Quick	count	
Heap		

Q5 =

```
int binary(int arr[], int l, int s, int x)   // recursive
{   if (r>=l)
    {  int mid = l+(s-l)/2;
       if (arr[mid]==x)
          return mid;
       else if (arr[mid]>x)
          return binary(arr, l, m-1, x);
       else
          return binary(arr, mid+1, r, x);
    }
    return -1;
}
```

```
int binary(int arr[], int l, int r, int x)
{ while (l <= r)
  { int m = l + (r-l)/2;
    if (arr[m] == x)
      return m;
    else if (arr[m] > x)
      r = m-1;
    else
      l = m+1
  }
  return -1;
}
```

Time complexity

Binary search => O(log n)

Linear search => O(n)

Q6 Recurrence relation for binary recursive search

$T(n) = T(n/2) + 1$

where $T(n)$ is the time required for binary search is an array of size n.

Q7

```
int find (A[], n, k)
{ sort (A, n)
  for (i=0 to n-1)
  { x = binarysearch (A, 0, n-1, K-A[i])
    if (n)
      return 1
  } return -1;
}
```

Time complexity $= O(n \log n) + n \cdot O(\log n)$

$= O(n \log n)$

⑧. Quick sort is the fastest general purpose sort. In most practical situations, quick sort is the method of choice. If stability is important and space is available, merge sort might be best.

Q9 A pair (a[i], a[j]) is said to be inversion if a[i] > a[j]
In arr[] = {7, 21, 31, 8, 10, 1, 20, 6, 4, 5}
total no of inversion are 31 using merge sort.

Q10 The worst case time complexity of quick sort is $O(n^2)$.
This case occurs when the picked pivot is always an extreme (smallest or largest) element. This happens when input array is sorted or reverse sorted.
The best case of quick sort is when we will select pivot as a mean element.

Q(i) Recurrence relation of
Merge sort → $T(n) = 2T(n/2) + n$
Quick sort → $T(n) = 2T(n/2) + n$.

(•) Merge sort is more efficient and works faster than quick sort in case of larger array size or datasets.

(•) Worst case complexity for quick sort is $O(n^2)$ whereas $O(n \log n)$ for merge sort.

Q12 Stable selection sort

```
void Stable selection (int arr[], int n)
{ for( int i=0; i< n-1 ; i++)
    { int min = i;
    for (int j = i+1; j<n; j++)
{     if (arr[min] > arr[j])
        min = j;
}
  int key = arr[min];
  while ( min >i)
  { arr[min] = arr[ min -1];
    m--;
  } arr[i] = key; }
```

Q13 Modified Bubble sorting

```
void bubble (int a[], int n)
{   for (int i=0; i<n; j++)
    { int swaps = 0;
      for (int j=0; j < n-1-i ; j++)
      { if (a[j] > a[j+1])
        { int t= a[j]; a[j];
          a[j] = a[j+1];
          a[j+1] = t;
          swaps ++;
        }
      }
      if (swaps==0)
        break;
    }
}
```