TUTORIAL-3

(1)
```
int linearsearch (int arr[], int n, int key)
{   for (int i=0; i<n; i++)
    {   if (arr[i] == key)
            return i;
    }
    return -1;
}
```

(2) Iterative Insertion Sort:—

```
void InsertionSort (int arr[], int n){
    int i, j, t=0;
    for (i=1; i<n; i++){
        t = arr[i];
        j = i-1;
        while (i>=0 && t<arr[j])
            arr[j+1] = arr[j];
            j--;
        }
        arr[j+1] = t;
    }
}
```

Recursive Insertion Sort:—

```
void insertionSort (int arr[], int n){
    if (n<=1)
        return;
    insertionSort (arr, n-1);
    last = arr[n-1];
    j = n-2;
```

```
while (j>=0 && arr[j] > last) {
    arr[j+1] = arr[j];
    j--;
}
arr[j+1] = last;
```

Insertion sort is called online sorting because it does not need to know anything about what values it will sort and which information is requested while the algorithm is running.

(3) (I) Bubble Sort:-

Time complexity - Best case = $O(n^2)$
Worst case = $O(n^2)$

Space complexity = $O(1)$

(II) Selection Sort:-

Time complexity - Best case - $O(n^2)$
Worst case - $O(n^2)$

Space complexity - $O(1)$

(III) Merge sort:-

Time complexity - Best case - $O(n \log n)$
Worst case - $O(n \log n)$

Space complexity - $O(n)$

(IV) Insertion Sort:-

Time complexity - Best case - $O(n)$
Worst case - $O(n^2)$

Space complexity - $O(1)$

(v) Quick Sort:-

Time Complexity - Best case - O(n log n)
Worst Case - O(n²)

Space complexity - O(n)

(vi) Heap Sort:-

Time complexity - Best case - O(n log n)
Worst Case - O(n log n)

Space complexity - O(1)

Sorting	Inplace	Stable	Online
Selection	✓		
Insertion	✓	✓	
Merge		✓	
Quick	✓		
Heap	✓		
Bubble	✓	✓	

Iterative Binary Search

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

Time complexity - Best case = O(1)
Avg case = O(log n)
Worst case = O(log n)

Recursive Binary Search

```
int binarySearch (int arr[], int l, int r, int key) {
    if (r >= l)
        int m = (l+r)/2;
    if (arr[m] == key)
        return m;
    else if (arr[m] > key)
        return binarySearch(arr, l, mid-1, ke
    else
        return binarySearch(arr, mid+1, r, key
}
```

Time complexity - Best Case - O(1)
Avg Case - O(log n)
Worst Case - O(log n)

Linear Search

Time complexity - Best case - O(1)
Avg. case - O(n)
Worst case - O(n)

(6) Recurrence relation for binary recursive Search -
$T(n) = T(n/2) + 1$

(7) $A[i] + A[j] = K$

(8) Quick Sort is the fastest general purpose sort. In most practical situations, quick sort is the method of ch of stability is important & space is available, merge sort might be best.

```
int merge(int arr[], int temp[], int left, int mid, int right)
{
    int i, j, k, inv_count = 0;
    i = left;
    j = mid;
    k = left;
    while((i <= mid-1) && (j <= right))
    {
        if(arr[i] <= arr[j])
            temp[k++] = arr[i++];
        else {
            temp[k++] = arr[j++];
            inv_count = inv_count + (mid - i);
        }
    }
    while(i <= mid-1)
        temp[k++] = arr[i++];
    while(j <= right)
        temp[k++] = arr[j++];
    for(i = left; i <= right; i++)
        arr[i] = temp[i];
    return inv_count;
}

int main() {
    int arr[] = {7, 21, 51, 8, 10, 1, 20, 6, 4, 5};
    int n = sizeof(arr)/sizeof(arr[0]);
    int ans = mergeSort(arr, n);
    cout << "No of inversions are " << ans;
    return 0;
}
```

(9) Inversion count for an array indicates - how far (close) the array is from being sorted. If the array is a sorted, then the inversion count is 0, but if the array is sorted in the reverse order, the inversion count is the maximum.

arr[] = {7, 21, 31, 8, 10, 1, 20, 6, 4, 5}

```
#include <bits/stdc++.h>
using namespace std;
int merge_sort (int arr[], int temp[], int left, int righ
int merge (int arr[], int temp[], int left, int mid, int ri

~~int merge_sort(int arr[], int~~
int mergesort (int arr[], int array_size)
{
    int temp [array_size];
    return merge_sort (arr, temp, 0, array_size -1
}
int merge_sort (int arr[], int temp[], int left, int rig
{
    int mid, inv_count = 0;
    if (right > left)
    {
        mid = left + (right - left)/2;
        inv_count += merge_sort (arr, temp, left, mid
        inv_count += merge_sort (arr, temp, mid+1, rig
        inv_count += merge (arr, temp, left, mid+1, rig
    }
    return inv_count;
}
```

6) The worst case time complexity of Quick sort is $O(n^2)$. The worst case occurs when the picked pivot element is always an extreme (smallest or largest) element. This happen when input array is sorted or reverse sorted & either first or last element is picked as pivot.

The least least case of quick sort is when we will select pivot as a mean element.

7) Recurrence Relation of:
(a) merge sort → $T(n) = 2T(n/2) + n$
(b) Quick sort → $T(n) = 2T(n/2) + n$

→ Merge sort is more efficient & works faster than quick sort in case of large array size or datasets.
→ Worst case complexity for quick sort is $O(n^2)$ whereas $O(n \log n)$ for merge sort.

8) Stable Selection Sort

```
void StableSelectionSort(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];
            min--;
        }
        arr[i]= key;
    }
```

```
int main()
{
    int arr = {4,5,3,2,4,1};
    int n = sizeof(arr)/sizeof(arr[0]);
    StableSelectionSort(arr, n);
    for(int i=0; i<n; i++)
        cout<<arr[i]<<" ";
    cout<<endl;
    return 0;
}
```

(13) The easiest way to do this is to use external so we divide our source file into temporary files of size to the size of the RAM & first sort these files.

- External Sorting:- If the input data is such that it cannot adjusted in the memory entirely at once it to be ~~adjusted~~ sorted on a hard disk, floppy disk o other storage device. This is called external sorting
- Internal Sorting:- If the input data is such that it adjusted in the main memory at once It is called int sorting