Date. ______
Page No. ______

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 ( j>=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 the 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 - $O(n)$

Date. ______
Page No. ______

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 - $O(n^2)$
Space complexity - $O(n)$

vi) Heap sort -
Time complexity - Best case - $O(n\log n)$
Worst case - $O(n\log n)$
Space compexity - $O(1)$

4.

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

5. iterative binary search

```
int binarysearch (int arr[], int l, int r,
                                    int key) {
      while (l<=r){
```

Date. ______
Page No. ______

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

T.C.
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, key);
    else
        return binarySearch (arr, mid+1, r, key);
    }
    return -1;
}
```

T.C.
Best Case = $O(1)$
Avg. Case = $O(\log n)$
Worst Case = $O(\log n)$

Linear Search T.C.

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, quicksort is the method of choice. If stability is important & space is available, merge sort might be best.

9. Inversion count for an array indicates – how far (or close) the array is from being sorted. If the array is already 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 right);
int merge (int arr [], int temp[], int left,
                   int mid, int right);

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 right){
    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, right);
        inv_count += merge (arr, temp,
                             left, mid+1, right);
    }
    return inv_count;
}
```

```
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; j <= 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 = size of (arr)/ size of (arr[0]);
```

```
int ans = mergesort (arr, n);
cout << " no of inversion are " << ans;
return 0;
}
```

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

11. Recurrence relation of :
a) Merge Sort $\Rightarrow T(n) = 2T(n/2) + n$
b) Quick Sort $\Rightarrow T(n) = 2T(n/2) + n$

→ Merge sort is more efficient & 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.

Date. ________
Page No. ________

12. 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] = a[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;
}
```

Date.
Page No.

13. The easiest way to do this is to use external sorting. we divide our source file into ~~temperory~~ temporary files of size equal to the size of the RAM & first sort these files.

- External Sorting: If the input data is such that is cannot adjusted in the memory entirely at once it needs to be sorted in a hard disk, floppy disk or any other storage device. This is called external Sorting.

- Internal sorting: If the input data is such that it can adjusted in the main memory at once it is called internal sorting.