Tutorial-3

Sol 1:- int Linear_Search (int *arr, int n, int key)

```
for i>=0 to n-1
    if arr[i] = key
        return i
return -1.
```

Sol 2:- iterative insertion sort

```
void insertionsort (int arr[], int n)
{
    int i, temp, j;
    for i <- 1 to n
    {
        temp <- arr[i]
        j <- i-1
        while (j>=0 AND arr[j] > temp)
            arr[j+1] <- arr[j]
            j <- j-1.
        arr[j+1] <- temp
```

→ recursive insertion sort

```
void insertion_sort (int arr[], int n)
    if (n<=1)
        return
    insertion_sort(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.

So13:- (i) Selection Sort :-
→ time complexity = Best case :- $O(n^2)$; Worst case = $O(n^2)$
→ space complexity = $O(1)$

(ii) Insertion Sort :-
→ time complexity = Best case - $O(n)$; 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) Quick sort :-
→ time complexity = Best case = $O(n \log n)$ worst case = $O(n^2)$
→ space complexity = $O(n)$

(v) Heap Sort :-
→ time complexity = Best case = $O(n \log n)$ worst case = $O(n \log n)$
→ space complexity = $O(1)$

(vi) Bubble sorting :-
→ time complexity = Best case = $O(n^2)$ worst case = $O(n^2)$
→ space complexity = $O(1)$

So14:-

Sorting	inplace	stable	online	
selection sort	✓			
Insertion sort	✓	✓	✓	
merge sort		✓		
quick sort	✓			
heap sort	✓			
Bubble sort	✓	✓		

Sol5:- • iterative binary search

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

→ Time complexity

Best case = $O(1)$
Average case = $O(\log_2 n)$
Worst case = $O(\log n)$

• Recursive Binary Search

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

→ Time complexity

Best case = $O(1)$
Average case = $O(\log n)$
Worst case = $O(\log n)$

Sol 6:- Recurrence Relation for binary recursive search

$$\boxed{T(n) = T(n/2) + 1}$$

Sol 7:- $A[i] + A[j] = K$

Sol 8:- Quicksort is the fastest general-purpose sort. In most practical situations, quicksort is the method of choice. If stability is important & space is available, mergesort might be best.

Sol 9:- Inversion count for any 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 array is sorted in the reverse order, the inversion count is maximum.

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

```
# include ~~nam~~ <bits/stdc++.h>
using namespace std;
int _mergeSort(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 _mergeSort(arr, temp, 0, array_size - 1);
}
int _mergeSort(int arr[], int temp[], int left, int right)
{
    int mid, inv_count = 0;
    if (right > left)
    {   mid = (right + left) / 2;
```

```
        inv_count += _mergesort(arr, temp, left, mid);
        inv_count += _mergesort(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;
    int 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, 31, 8, 10, 1, 20, 6, 4, 5}
    int n = sizeof(arr)/sizeof(arr[0]);
    int ans = mergeSort(arr, n);
    cout << "Number of inversion are" << ans;
    return 0;
}
```

Sol 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.

Sol 11:- 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 sor 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.

Sol12:- Stable selection sort

```
using namespace std;
void stableselectionsort (int a[], int n)
{
        for (int i = 0 ; i<n-1 ; i++)
        {
                int min = i ;
                for ( int j = i+1 ; j<n ; j++)
                        if (a[min] > a[j])
                                min = j;
                int key = a[min];
                while (min > i)
                {
                        a[min] = a[min-1];
                        min--;
                }
                a[i] = key;
        }
}
int main()
{
        int a[] = {4, 5, 3, 2, 4, 1};
        int n = sizeof(a) / sizeof(a[0]);
        stableselectionsort(a, n);
        for (int i = 0 ; i<n; i++)
                cout << a[i] << " ";
        cout << endl;
        return 0;
}
```

Sol 13:- The easiest way to do this is to use external sorting. we divide our source file into temporary files of size equal 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 needs to be stored 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