Assignment - 2

1. Linear Search (pseudo Code)

```
int   lin_search (int * arr , int n, int key)
{
    for (i>=0  to  n-1)
    {
         if  (arr [i] == Key)
               return i;
    }
  return -1;
}
```

2. Iterative insertion Sort

```
void   insert_Sort (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;
          }
     }
}
```

Reecursive insertion Sort:-

```
Void insert_sort (int arr[] , int n)
{
        if (n<=1)
            return;
        insert_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 sort because it doesn't need to know anything about what values it will sort and the information is requested while the algo is running.

3. (i) Selection Sort :

T.C = Best Case = $O(n^2)$

= Worst Case = $O(n^2)$

S.C = $O(1)$

(ii) Insertion Sort :

T.C = Best Case = $O(n)$

Worst Case = $O(n^2)$

S.C = $O(1)$

(iii) Merge Sort:-

T.C = Best Case = $o(n \log n)$

Worst Case = $o(n \log n)$

S.C = $o(n)$

(iv) Quick Sort :-

T.C = Best Case = $o(n \log n)$

Worst Case = ~~$o(n \log n)$~~ $o(n^2)$

S.C = $o(1)$

(v) Heap Sort :-

T.C = Best Case = $o(n \log n)$

Worst Case = $o(n \log n)$

S.C = $o(1)$

(vi) Bubble Sort :-

T.C = Best Case = $o(n^2)$

Worst Case = $o(n^2)$

S.C = $o(1)$

4.

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

5. Iterative Binary Search:-

```
int    bin-search (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 = $O(1)$

Avg case = $O(\log_2 n)$

Worst Case = $O(\log n)$

Recursive Binary Search

```
int    bin-search (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 bin-search (arr, l , mid-1, x)
          else
              return bin-search (arr, mid+1 , r, x)
      }
    return-1;
}
```

Time Complexity:-

Best case = $O(1)$

Avg case = $O(\log n)$

Worst case = $O(\log n)$

6. Recurrence Relation for binary recursive Search

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

7.

```
Void findTwoIndexes (int arr[], int length , int k)
{
     int left = 0;
     int right = length-1;
     int result = {-1, -1};
   while (left < right)
   {
       int currentSum = arr[left] + arr[right];
       if (currentSum == k)
       {
           result.left = left;
           result.right = right;
           return result;
       }
       else if (currentSum < k)
       {
             left = left +1;
       }
       else {
           right = right -1;
       }
   }
  return result;
}
```

Time Complexity:-

Best Case = O(1)

Avg Case = O(n)

worst case - O(n)

8.

Quick Sort is the fastest general purpose sort. If the array is already sorted, then the inversion count is 0, but if array is soorted in reverse order, the inversion count is max.

9.

If the array is already sorted, then the inversion count is 0, but if array is solved in reverse order. the inversion count is max.

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

```
# include <stdio.h>
int merge (int arr[], int temp-arr[], int left, int mid, int right)
 int i = left;
  int j = mid+1;
  int k = left;
  int inv_count = 0;

 while (i <= mid && j <= right
 {
     if (arr[i] <= arr[j])
     {
         temp_arr[k] = arr[i];
         i++;
     }
     else
     {
         temp-arr[k] = arr[j];
```

```
            j++ ;
            inv_Count += (mid-i+1);
        }
        k++;
    }

    while (i<= mid) {
        temp_arr[k] = arr[i]
        k++ ;
        i++ ;
    }
    while (j<= right)
    {
        temp_arr [k] = arr[j];
        k++ ;
        j++;
    }
    for (int i = left ; i<= right ; i++)
    {
        arr [i] = temp_arr [i];
    }
    return inv_count;
}

int mergeSort (int arr[], int temp_arr[], int left , int right)
{
    int inv_count = 0;
    if (left < right ) {
        int mid = (left + right)/2;
        inv_count += mergesort (arr, temp_arr , left , mid);
        inv_Count += mergeSort (arr, temp_arr , mid+1 , right);
        inv_Count += merge (arr , temp_arr , left , mid , right);
        {
        return inv_count;
    }

int CountInversions (int arr[], int n)
{
      int temp_arr[n];
      return mergeSort (arr, temp-arr, 0, n-1);
  }

int main( )
{
    int arr[] = {7, 21, 31, 8, 10, 1, 20, 6, 4, 5};
    int n = sizeof (arr) / size of (arr[0])
    Printf (" No. of Inversions: %d", CountInversions (arr, n));
    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 on extreme (smallest or largest) element. This happens 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. Reccurance Relation of:

(a) Mege Sort → $T(n) = 2T(n/2)+n$

(b) Quick Sort → $T(n) = 2T(n/2)+n$

→ Merge Sort is more efficient and works faster thon quick Sort in case of larger array size or data set.

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

12. Stable Selection Sort:-

```
void    stab_selSort (int arr[] , 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 = size of (a) / size of (a[0]);
      stab_selSort (a, n);
       for (int i=0 ; i<n ; i++)
       {
             Scanf ("%d", a[i]);
        }
       return 0;
  }
```

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 and first sort the files.

- External Sorting: If the input data is such that it cannot adjust in the memory entirely at once it need 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 adjust in the main memory at once, it is called internal Sorting.