# Assignment - 2.

(1) Linear ~~Search~~ Search (pseudo code)

```
int lin_search (int *ar, 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 && arr[j] > temp)
            arr[j+1] = ar[j];
            j = j-1;
        arr[j+1] = temp;
```

(3) Recursion 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
```

⇒ ~~Install~~ 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 = $O(n^2)$
= Least Case = $O(n^2)$

S.C = $O(1)$

(ii) Insertion Sort

T.C = Best Case = $O(n)$
= Least 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(1)~~ $O(n)$

(iv) Quick Sort

TC = Best Case = $O(n \log n)$
Worst Case = $O(n^2)$

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;
        }
}
```

T.C  B.C = O(1)
Avg C = O($\log_2 n$)
Worst C = O(log n)

```
        return -1
    }
```

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;
}
```

(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;
}
```

T.C -

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 sorted in reverse order, its 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.

```
# 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= 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)/ sizeof(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 an 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 if

(a) MergeSort – $T(n) = 2T(n/2) + n$

(b) QuickSort – $T(n) = 2T(n/2) + n$

→ Merge Sort is more efficient and works faster than 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 StabSelSort (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 = sizeof(a)/sizeof(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 be 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 device is such that it can adjust in the main memory at once, it is called internal sorting.