Toutorial Sheet 3

Sol1

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

Sol2 Iterative Insertion Sort.

```
void insertion_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.
```

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 required while the algorithm is running.

Sol 3

	Time complexity		Space complexity
	Best case	Worst case	
(i) Selection Sort	$O(n^2)$	$O(n^2)$	$O(1)$
(ii) Insertion Sort	$O(n)$	$O(n^2)$	$O(1)$
(iii) Merge Sort	$O(n\log n)$	$O(n\log n)$	$O(n)$
(iv) Quick Sort	$O(n\log n)$	$O(n^2)$	$O(n)$
(v) Heap Sort	$O(n\log n)$	$O(n\log n)$	$O(1)$
(vi) Bubble Sort	$O(n^2)$	$O(n^2)$	$O(1)$

Sol 4

Sorting	inplace	stable	online
Selection sort	✓	✗	✗
Insertion sort	✓	✓	✓
Merge sort	✗	✓	✗
Quick sort	✓	✗	✗
Heap sort	✓	✗	✗
Bubble sort	✓	✓	✗

Sds (3)

Iterative Binary Search

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

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

Recursive Binary Search

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

Time complexity
Best case - $O(1)$
Avg. case - $O(\log n)$
Wrost case - $O(\log n)$

Sol6 Recursive Relation for binary search.

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

Sol7 $A[i] + A[j] = K$

Sol8 Quick sort is the fastest general purpose sort. In most practical situation, quick-sort is the method of choice. If stability is important and space is available, merge sort might be best.

Sol9: 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 reverse order the inversion count is 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 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) / 2;
```

```
int_ count += merge-sort (arr, temp, left, mid);
inv_count += merge-sort (arr, temp, mid+1, right);
inv-count += merge-so (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 = right;
    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 << "No. 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 picket pivot is always on extreme (smallest or largest) element

This happens when i/p 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 Recurrent 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 and 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.

Sol 12.) Stable Selection Sort :-

```
# include <bits/stdc++.h>
using namespace std;
void stable_selection_sort (int a[], int n)
{
  for (int i =0; i<n-1; i++)
  {
    int min=1;
    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]);
  stable_selection_sort (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 file of size equal to the size of the RAM and first sort these files

→ External Sorting:- If the I/P data is such that it Cannot be adjusted in the memory entirely at once, it needs to be sorted in a hard disk, floppy disk is any other storage device. This is called external sorting.

→ Internal Sorting :- If the i/p data is such that it can be adjusted in the main m/m at once, it is called internal sorting.