Date.
Page No.

Assignment -1

1.
```
int   linearSearchSorted ( int arr[], int target) {

      for (int i=0 ; i<arr.size(); ++i) {

            if (arr[i]==target) return i ;
            else if (arr[i] > target ) break ;
      }
      return -1;
}
```

2.
```
void  iterativeInsertionSort (int arr[]) {

      for (int i = 1 ; i<arr.size(); ++i) {
            int key = arr[i];
            int j = i-1 ;
            while (j>=0 && arr[j] > key ) {
               arr [j+1] = arr[j] ;
               --j ;
            }
            arr [j+1] = key ;
      }
}
```

3.
```
void recursiveInsertionSort ( int[] arr , int n) {
      if (n>1) {
            recursiveInsertionSort(arr, n-1) ;
            int key = arr[n-1] ;
               int j = n-2 ;
               while(j>=0 && arr[j]>key) {
                  arr[j+1] = arr[j];
                  --j ;
      }
```

Date. ____________
Page No. ____________

```
        ~~arr[j+1] = arr[j];~~
      }
       arr[i+1] = key ;
      }
    }

4.  int  binarySearch ( int  arr[] , int  target) {

        int  low = 0 ;
         int  high  = arr.size () - 1 ;
        while ( low <= high )  {
                  int  mid = low + (high - low) / 2 ;
                  ~~int~~ .
                  if  (arr[mid] = target ) return mid;
                  elseif (arr[mid] < target) low = mid + 1 ;
                  else high = mid - 1;
            }
            return -1;
        }

5-  BinarySearch ( int  arr[] , int  target, int low , int high) {

        if (low <= high) {
                int mid = low + (high - low) / 2 ;
              if (arr[mid] == target) return mid;
             else if (arr[mid] < target) {
               return binarySearch(arr, target, mid+1, high);
               else return binarySearch(arr, target, low,
                                                   mid-1);

        }
       return -1;
    }
```

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

7.

~~finding two In~~

```
#include<iostream>
~~#~~ ~~in~~.

~~st~~

using namespace std;
pair<int,int> findindices WithSum k( const int arr[], int size, int k)
{
    const    int MAX_VALUE = 10000; //
    int seen [MAX_VALUE] ={0};
    for(int i=0; i<size; i++) {
        int complement = k - arr[i];
        if (complement >=0 && seen[complement!=0)){
            return {seen[complement]-1, i};
        }
        seen [arr [i]] = i+1;
    }
    return {-1,-1}
}

int main(){ 
    const int arr = {2,7,11,15};
    int size = sizeof(arr) / sizeof(arr[0]);
    int  k=9;
    pair <int, int> indices = findIndicesWithSumK(arr, size, k);

    if (indices.first != -1 && indices.second != -1){

        cout << k << indices.first << "," << indices.second;
    }
    else
    {
      cout << k << "-" << endl;
    }
    return 0;
}
```

8. Merge Sort or Quick Sort, as they have good average-case performance and are widely used in practice.

9. ~~Ques~~

- An inversion is a pair(i, j) such that i<j but arr[i] > arr[j].
- Number of inversion in {7, 21, 31, 8, 10, 1, 20, 6, 4, 5} using merge sort is 22.

10.

Best Case : O(nlogn)
when the pivot consistently divides the array into two halves.
Worst Case : O(n^2) when the pivot consistently divide the array into two equal halves.

Date.
Page No.

11. Merge Sort:

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

Quick Sort:

Best case : $T(n) = 2T(n/2) + O(n)$

Worst case: $T(n) = T(n-1) + O(n)$

Similarities : Both divide the array and conquer recursively.
Differences : Quick Sort's pivot selection affects its performance, leading to different best & ~~best~~ worst case scenarios.

12. Maintain a stable selection sort by swapping elements only if the element to be moved is smaller than the current minimum.

13. Introduce a flag to check if any swaps were made during a pass; if not; the array is already sorted.

14. External Sorting:
Use Algorithms like merge sort or ~~External~~ merge sort that minimize the need for loading the entire data into RAM.

External Sorting involves using external storage to manage larger data sets ~~that~~ don't fill entirely into ram.