Assignment – 2.

1) function linearSearch(arr, target)
if n == 0;
return "Array is empty."
for (int i = 0 to n-1)
if (arr[i] == target)
return "element found" + i;
else if arr[i] > target
return ("Element not found");

return "Element not found";

2) Iterative insertion sort –
iterative_insertion (arr)
n = length of arr.
for i from 1 to n-1;
key = arr[i]
j = i-1;
while((j >= 0) && (arr[j] > key))
arr[j+1] = arr[j];
j = j-1;
arr[j+1] = key.

Recursive insertion sort
recursive_insertion (arr, n);
if n <= 1;
return insertion (arr, n-1)
last = arr[n-1];
j = n-2
while(j >= 0 && arr[j] > last)
arr[j+1] = arr[j]
j = j-1;
arr[j+1] = last;

Insertion sort is sometimes called "online sorting", as it sorts elements one at a time as they are presented, which makes it suitable for sorting data as it arrive, hence "online".

3.)

Bubble	Best	Avg.	Worst	SC
Bubble	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Selection	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
Insertion	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Merge	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$
Quick	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$ to $O(n)$
Heap	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$
Counting	$O(n+k)$			$O(n+k)$
Radix	$O((n+b) * \log_b(k))$			$O(n+b)$

4) Inplace algorithm —
1. selection sort
2. insertion sort
3. bubble sort
4. quick sort

Online algorithm —
1. insertion

Stable algorithm —
1. Merge.
2. insertion
3. bubble
4. count.
5. radix.

5) Recursive Binary Search —

```
function binarySearch (arr, low, high, target)
if high >= low.
mid = low + (high - low) /2;
if arr [mid] == target
return mid
elseif arr [mid] > target
return binarySearch (arr, low, mid - 1, target)
else
return binarySearch (arr, mid + 1, high, target)
else
return -1;
```

Iterative Binary Search —

```
function binarySearch (arr, target)
low = 0
high = length of arr - 1.
while (low <= high)
mid = low + (high - low) /2;
if (arr [mid] == target)
return mid
else if (arr [mid] < target)
low = mid + 1;
else
high = mid - 1;
return -1,
```

Linear Search — T.C = O(n)
S.C = O(1)

Binary Search — TC = O(log n)
SC = O(log n)

6.) Recurrence relation ——

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

where,

$T(n)$ = time taken to search within array of size n.

$T(n/2)$ = time taken to search within half of array.

$O(1)$ = const. time taken for comparison & other operations.

7.)
```
#include<iostream>
using namespace std;
void findIndexes(int A[], int size, int K)
{
  for(int i=0; i<size-1; ++i)
  {
    for(int j=i+1; j<size; ++j)
    {
      if(A[i]+A[j]==K)
      {
        cout<<"Indexes"<<i<<","<<j<<endl;
        cout<<"Numbers"<<A[i]<<","<A[j]<<endl;
        return;
      }
    }
  }
  cout<<"No such pair exists"<<endl;
}
```

8.) Insertion sort —

- it is stable & in-place, making it suitable for memory usage is a concern.

Selection sort —

- suitable for small datasets or situations where simplicity is more imp. than performance.

9.) In this context of arrays, an inversion refers to a pair of elements (arr[i], arr[j]) such that i, j, but (arr[i] > arr[j]). In simpler terms, it indicates that the elements are out of order relative to each other.

```
#include <iostream>
using namespace std;
long long merge(int arr[], int temp[], int left, int mid, int
                right)
{
  int i = left;
  int j = mid;
  int k = left;
  long long inversions = 0;
  while(i<mid && j<=right)
  {
    if(arr[i] <= arr[j])
    {
      temp[k++] = arr[i++];
    }
    else
    {
      temp[k++] = arr[j++];
      inversions += mid - 1;
    }
```

```
        }
    }
    while (i < mid)
    {
        temp[k++] = arr[j++];
    }
    for (i = left; i <= right; i++)
    {
        arr[i] = temp[i];
    }
    return inversions;
}

long long mergeSort(int arr[], int temp[], int left, int right)
{
    long long inversions = 0;
    if (left < right)
    {
        int mid = left + (right - left) / 2;
        inversions += mergeSort(arr, temp, left, mid);
        inversions += mergeSort(arr, temp, mid+1, right);
        inversions += mergeSort(arr, temp, left, mid+1, right);
    }
    return inversions;
}
```

10.) The best case complexity (TC) occurs when the partition process always picks the middle element as the pivot.

$$\boxed{TC = O(n \log n)}$$

The worst case time complexity happens when the pivot selection consistently results in highly unbalanced partitions, such as when the smallest or largest element is always chosen as the pivot.

$$\boxed{TC = O(n^2)}$$

11.) Merge Sort —

Best Case — $T(n) = 2T(n/2) + O(n)$

Worst Case — $T(n) = 2T(n/2) + O(n)$

Quick Sort —

Best Case — $T(n) = 2T(n/2) + O(n)$

Worst Case — $T(n) = T(n-1) + O(n)$

Similarities —

- both have best TC = $O(n \log n)$
  worst TC = $O(n \log n)$
- both use divide & conquer technique.
- both are comparison-based sorting algorithms.

Differences —

1. merge sort is stable sort, whereas Quick Sort is not.
2. Merge sort requires more space, whereas Quick sort is inplace sorting algorithms.

12.) To make a stable version of selection sort, we need to modify the selection process so that it doesn't swap equal elements unnecessarily.

```
void stableSelection(int arr[], int n)
{
  for (int i=0; i<n-1; ++i)
  {
    int min_index = i;
    for (int j= i+1; j< n; ++j)
    {
      if (arr[j] < arr[min_index])
      {
        min_index = j;
      }
    }
    while (min_index > i)
    {
      arr[min_index] = arr[min_index - 1];
      -- min_index;
    }
    min[i] = min_value;
  }
}
```

13.)
```
#include <iostream>
void bubbleSort(int arr[], int n)
{
  for (int i=0; i<n; i++)
  {
    bool swapped = false;
    for (int j=0; j<n-i-1; j++)
    {
      if (arr[j] > arr[j+1])
      {
        int temp = arr[j];
        arr[j] = arr[j+1];
        arr[j+1] = temp;
        swapped = true;
      }
    }
    if (!swapped)
    {
      break;
    }
  }
}
```