Tutorial-3

①
```
int linearSearch(int key, int *arr, int size){
    for(int i=0; i<size; i++){
        if(arr[i] >= key || arr[size-1] < key){
            if(arr[i] = = key)
                return i;

            break;
        }
    }
    return 0;
}
```

② Iterative insertion sort:

```
void insertionSort(int *arr, int size){
    int value;
    for(int i=1; i<n;i++){
        value= arr[i];
        int j =i-1;
        while(j>=0 && arr[j]> value){
            arr[j+1]= arr[j];
            j--;
        }
        arr[j+1]= value;
    }
}
```

Recursive insertion sort:

```
void insertionSort(int *arr, int size){
    int value = arr[i];
    int i, j=i-1;
    while (j>=0 && arr[j] >value){
        arr[j+1]= arr[j];
    }   j--;
    arr[j+1] = value;
  if (i <= size) {
        insertionSort (arr, size);
  }
}
```

Insertion sorting is called online sorting as we put the number at the right position with comparison from already traced elements insted of inserting the element at the end and then shifting all other elements.

Stable Sorting: In this, the relative order of equal elements will not change even after sorting. For eg: Merge sort, bubble sort.

Inplace Sorting: In this, the sorted array occupies the same space as the original one. Hence, the space complexity is O(1). For eg: bubble sort.

External Sorting: This type of sorting algorithm is used when the data to be sorted is so large that the Computer's main memory cannot be used to store it, insted secondary storage devices are used to store the data.
For eg: merge sort.

(3) Bubble Sort:

TC = $O(n^2)$ for all cases.

SC = $O(1)$

Selection Sort:

TC = $O(n^2)$ for all cases

SC = $O(1)$

Insertion Sort:

TC = $O(n)$ for best case
$O(n^2)$ for average & worst cases

SC = $O(1)$

Merge Sort:

TC = $O(n \log n)$ for all cases
SC = $O(n)$

Quick Sort:

TC = $O(n \log n)$ for best & average cases
$O(n^2)$ for worst cases

SC = $O(\log n)$ for best case
$O(n)$ for worst case

Heap Sort:

TC = $O(n \log n)$ for all cases
SC = $O(1)$

Topological Sorting:

TC = $O(V+E)$ for all cases.
SC = $O(V)$

(4) **Inplace Sorting**: bubble sort, selection sort, insertion sort, heap sort, quick sort.

Stable Sorting: merge sort, bubble sort, insertion sort.

Online Sorting: insertion sort

(5) Recursive binary Search:

```
function binarySearch (array, low, high, x)
    if(low <= high)
        mid = low+ (high-low)/2.
        if x == array[mid] return mid
        else if x > array[mid]
            return binarySearch (array, mid+1, high
                                                , x)
        else
            return binarySearch (array, low, mid-1, x)
    end if
end function
```

Time complexity = O(log n)
Space complexity = O(log n)

Iterative binary Search:

```
function binarySearch (array, low, high, x)
    if( low<= high)
        mid = low +(high - low)/2
        if x == array[mid] return mid
        else if x > array[mid]
                        low = mid+1

        else
            high = mid-1
    end if
end function
```

Time Complexity = $O(\log n)$
Space Complexity $= O(1)$

Linear Search:

Time complexity $= O(n)$
Space complexity $= O(1)$

6.

$$T(n) = T(n/2) + 1 \quad ; \; n > 1$$
$$T(1) = 1 \quad ; \; n = 1$$

7. A[i] + A[j] = K using Hash Table :

```
bool checkPair (int *arr, int n, int k){
    Take Hash table H of size n
    for (int i=0; i<n; i++){
        int x= k-arr[i];
        if (H.search(x) is true) return 1;
        H.insert(A[i]);
    }
  return -1;
}
```

8. For practical uses, quick sort is most preferable. It is generally considered as fastest sorting technique because it has the best performance in average ca for most inputs, i.e., $O(n\log n)$. Also, quick sort is an inpalace sorting algorithm.

(10) The quick sort will give the best case time complexity when the partitions are as evenly balanced as possible. ie. The size difference on either side of the pivot element is either 0 or 1.

It will give worst case time complexity when the partitions are mostly unbalanced. In this case the recursive call on the (n-1) elements will take (n-1) time, on (n-2) elements it will take (n-2) time and so on. and original call takes n time.

(11) Quick sort: (worst case)

$$T(n) = n + T(n-1) \quad ; n > 1$$

$$T(0) = T(1) = 0 \quad ; n \leq 0, n = 1$$

(best case)

$$T(n) = 2T(n/2) + n \quad ; n > 1$$

$$T(0) = T(1) = 0 \quad ; n = 0, 1$$

Merge Sort (best and worst case):

$$T(n) = 2T(n/2) + n \quad ; n > 1$$

$$T(1) = 1 \quad ; n = 1$$

Quick sort gives O(n logn) complexity for the best case and so as of merge sort (in all cases). This is because merge sort divides the array into 2 halves for all the cases and then merge the sorted part. Quick sort divides the array into two sub-parts based on the pivot element, which can be chosen randomly. The elements on the left of the pivot element are

Smaller whereas the elements on the right side are greater than the pivot element. This partition on the basis of pivot element is not necessarily balanced always, hence it gives different complexities for different cases. When the partition is balanced (in best case), it gives complexity similar to the merge sort, otherwise, complexity is different.

(12) Bubble sort scans whole array even the array is sorted to swap the adjacent elements until they are sorted. It can be modified by stopping the algorithm if inner loop does not cause any swapping of elements. Hence, the array will not scan the whole array once it is sorted.

```
void bubbleSort (int *arr, int n){
    bool swapped;
    for (int i=0; i<n-1 ;i++){
        swapped= false;
        for(int j= 0; j< n-i-1; j++){
            if (arr[j] > arr[j+1]){
                swap(&arr[j], &arr[j+1]);
                swapped= true;
            }
        }
        if(swapped == false) break;
    }
}
```

(13) If my computer's RAM is of 2GB and I am given an array of 4GB for sorting, then external sorting algorithm will be used because in this sorting algorithm if the data to be sorted is larger than the computer's main memory then the secondary storage devices are used to store the data. Merge sort works on the same principle, therefore it can be used for such problem.

Internal sorting is that type of sorting algorithm that takes place entirely inside the computer's main memory as the data to be sorted is small enough to be fit inside the main memory.

eg: bubble sort, insertion sort.