TUTORIAL 3

Ans 1)

```
while (low <= high)
{
    mid = (low + high) / 2;
    if (arr [mid] == key)
        return true;
    else if (arr[mid] > key)
        high = mid - 1;
    else
        low = mid + 1;
}
return false;
```

Ans 2) Iterative insertion sort:

```
for (int i = 1; i < n; i++)
{
    j = i - 1;
    x = A[i];
    while (j > -1 && A[j] > n)
    {
        A[j+1] = A[j]
        j--;
    }
    A[j+1] = n
}
```

Recursive insertion sort

```
void insertionsort (int arr[], int n)
{
    if (n <=1)
        return;
    insertion (arr, n-1);
    int last = arr[n-1];
    i = n-2;
    while (j>=0 && arr[j] > last)
    {
        arr[j+1] = arr[j];
        j--;
    }
    arr[j+1] =last;
}
```

Insertion sort is online sorting because whenever a new element comes, insertion sort defines its right place.

Ans 3) Bubble sort — $O(n^2)$

Insertion sort — $O(n^2)$

Selection sort — $O(n^2)$

Merge sort — $O(n+\log n)$

Quick sort — $O(n \log n)$

~~Count sort~~ Count sort — $O(n)$

Bucket sort — $O(n)$

Ans 4) Online sorting — Insertion sort

Stable sorting — Merge sort, Insertion sort, Bubble sort

Inplace sorting — Bubble sort, Insertion sort, Selection sort

Ans 5) Iterative Binary Search

```
while (low <= high)
{
    int mid = (low + high) /2
    if (arr [mid] == key)
        return true;
    else if (arr [mid] > key)
        high = mid -1;
    else
        low = mid +1;
}
```

O (logn)

Recursive Binary Search

```
while (low <= high)
{
    int mid = (low +high) /2
    if (arr [mid] == key)
        return true;
    else if (arr [mid] > key)
        Binary Search (arr, low, mid -1);
    else
        Binarysearch (arr, mid+1, high);
}
return false;
```

O (log n)

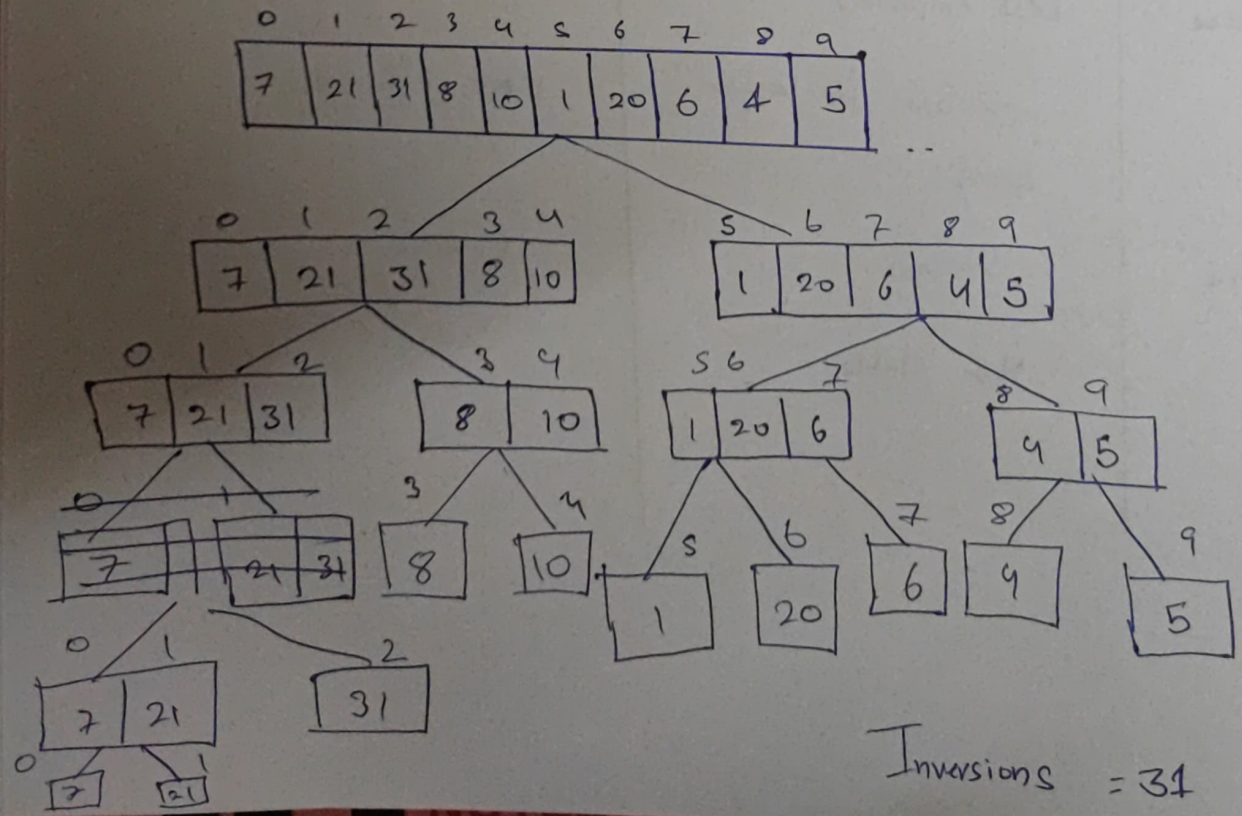
Ans 6) $T(n) = T(n/2) + T(n/2) + C$

Ans 7)

```
map <int, int> m;
for (int i=0; i<arr.size(); i++)
{
    if (m.find(target - arr[i]) = m.end())
        m[arr[i]] = 1;
    else
    {
        cout << i << " " << map[arr[i]];
    }
}
```

Ans 8) Quick sort is the fastest general purpose. In most practical solution, quicksort is the method of choice. If stability is important and space is available, merge sort might be best.

Ans 9) Inversion indicators — how far or close the array is from being sorted



Inversions = 31

Ans 10) Worst Case: The worst case occurs when the picked pivot is always an extreme (smallest or largest) element. This happen when input array is sorted or reverse sorted and either first or last element is picked as pivot. $O(n^2)$

Best Case: Best case occurs when pivot element is the middle element or near to the middle element $O(n \log n)$

Ans 11) Merge Sort: $T(n) = 2T\left(\frac{n}{2}\right) + O(n)$

Quick Sort: $T(n) = 2T\left(\frac{n}{2}\right) + n+1$

Basis	Quick Sort	Merge Sort
1. Partition	Splitting is done in any ratio	Array is ~~parted into just~~ parted into just two halves
2. Works well on	Smaller array	Fine on any size of array
3. Additional space	Less (inplace)	More (not inplace)
4. Efficient	inefficient for larger array	More efficient
5. Sorting method	Internal	External
6. Stability	Not Stable	Stable

Ans 14) We will use Merge Sort because we can divide the 4 GB data into 4 packets of 1 GB and sort them separately and combine them later.

- Internal Sorting: all the data to sort is stored in memory at all times while sorting is in progress

- External Sorting: all the data is stored is outside memory and only loaded into memory in small chunks.