

Q1) Solution:-

```
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;
```

Q2) Solution:- Iterative insertion sort:-

```
for (int i=1; i<n; i++)
{
    j = i-1;
    x = A[i];
    while (j > 0 && A[j] > x)
    {
        A[j+1] = A[j];
        j--;
    }
    A[j+1] = x;
}
```

Recursive insertion sort:-

```
void insertionSort (int arr[], int n)
{
    if (n <= 1) return;
```

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

```
    insertionSort (arr, n-1);
    int last = arr[n-1];
    j = n-2;
    while (j >= 0 && arr[j] > last)
    {
        arr[j+1] = arr[j];
        j--;
    }
    arr[j+1] = last;
}
```

Q3) Solution:-

- Bubble Sort $\rightarrow O(n^2)$
- Insertion Sort $\rightarrow O(n^2)$
- Selection Sort $\rightarrow O(n^2)$
- Merge Sort $\rightarrow O(n \log n)$
- Quick Sort $\rightarrow O(n \log n)$
- Count Sort $\rightarrow O(n)$
- Bucket Sort $\rightarrow O(n)$

Q4) Solution:- Online Sorting \rightarrow Insertion Sort
 Stable Sorting \rightarrow Merge Sort, Insertion Sort, Bubble Sort.
 Inplace Sorting \rightarrow Bubble Sort, Insertion Sort, Selection Sort.

Q5) Solution:- Iterative Binary Search:
 $O(\log n)$

```

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;
}
    
```

Recursive Binary Search:
 $O(\log n)$

```

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
        Binary-search(arr, mid + 1, high);
}
return false;
    
```

Q6) Solution:- $T(n) = T(n/2) + T(n/2) + C$

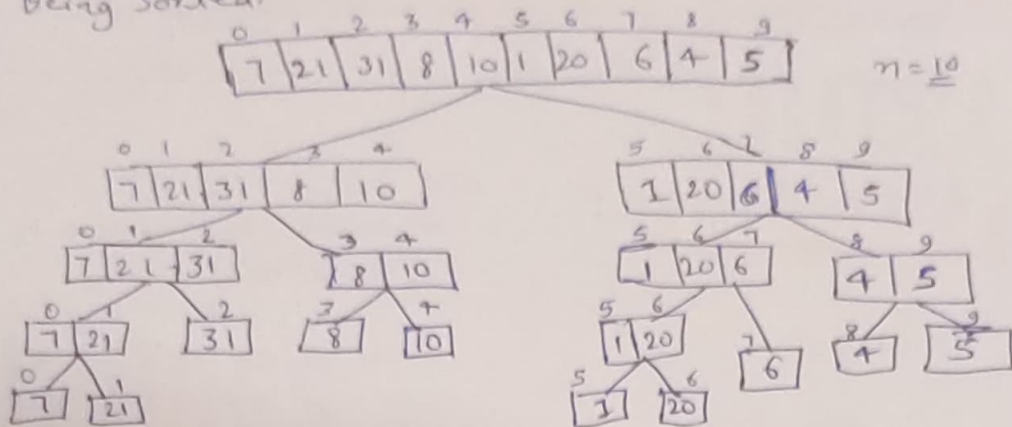
Q7) Solution:-

```

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

Q8) Solution:- Quicksort is the fastest general purpose sort. In most practical situation, quicksort is the method of choice. If stability is important and space is available, mergesort might be best.

Q9) Solution:- Inversion indicates - how far or close the array is from being sorted. 03



Inversion = 31

Q10) Solution:- Worst case:- The worst case occurs when the picked pivot is always an Extreme (Smallest or largest) element. This happens when input array is sorted as 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 as new to the middle element.

$O(n \log n)$

Q11) Solution:- Merge Sort: $T(n) = 2T(n/2) + O(n)$

Quick Sort: $T(n) = 2T(n/2) + n + 1$.

Quick Sort

- Partition
- Works well on
- Additional space
- Efficient
- Sorting Method
- Stability

Splitting is done in any ratio

Smaller array

Less (in-place)

inefficient for large array

Internal

Not stable

Merge Sort

array is parted into just 2 halves

fine on any size of array.

Move (Not in-place)

More efficient

External

Stable.

Q 14 > Solution:- We will use Mergesort 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 outside memory and only loaded into memory in small chunks.