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

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 insertion sort (int arr[ ], int n)
{
    if ( n <=1)
        return ;
insertion sort ( 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;
}
```

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 - $O(n)$

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;
}
```

$O(\log n)$

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

O(log n)

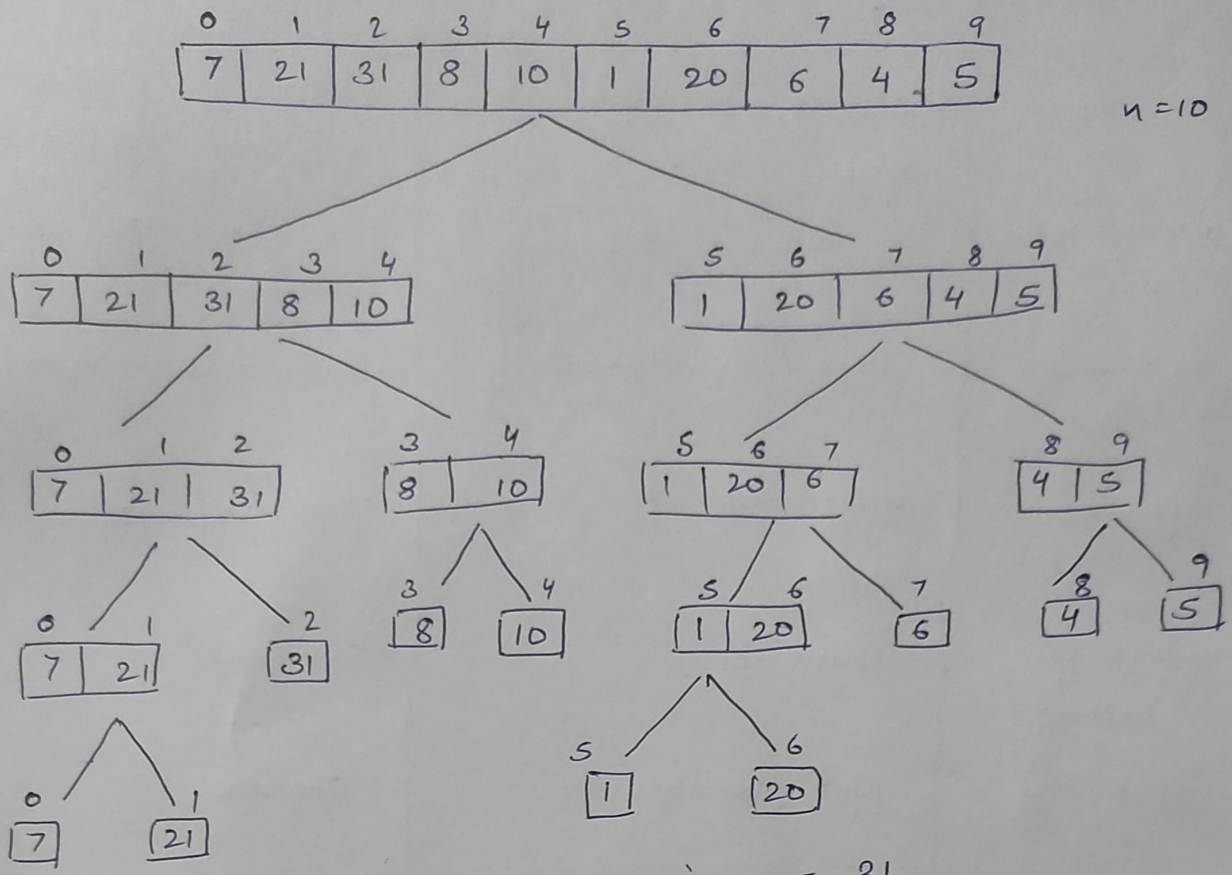
Ans 6 → $T(n) = T(n/2) + T(n/2) + c$

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 << "  " << mp [arr [i] ];
}
}
```

Ans 8 → 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, merge sort might be best.

Ans 9 → Inversion indicates - 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 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 near to the middle element $O(n \log n)$

Ans 11 →

Merge sort : $T(n) = 2T(\frac{n}{2}) + O(n)$

Quick sort : $T(n) = 2T(\frac{n}{2}) + n + 1$

	Quick Sort	Merge Sort
• Partition	Splitting is done in any ratio	array is parted into just 2 halves.
• Works well on	Smaller array	fine on any-size of array.
• Additional space	Less (in-place)	More (Not in-place)
• Efficient	inefficient for larger array	More efficient
• Sorting method	Internal	External
• 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 latter.

- Internal sorting → all the data to sort is sorted 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.