Page Date

Tutorial - 3

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;
```

2. Iterative insertion sort :

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

Recursive insertion sort :

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

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

Insertion sort is called online sorting as whenever a new element comes, insertion sort define its right place.

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 - $O(n)$

4. Online sorting - Insertion sort
Inplace sorting - Bubble sort, Insertion sort, Selection sort
Stable sorting - Bubble sort, Insertion sort, Merge sort

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;
}
```

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;
```

- Binary search time complexity = $O(\log n)$
- Binary search space complexity:
  - → iterative = $O(1)$
  - → recursive = $O(\log n)$
- Linear search time complexity = $O(n)$
- linear search space complexity = $O(1)$

6. $T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + C$

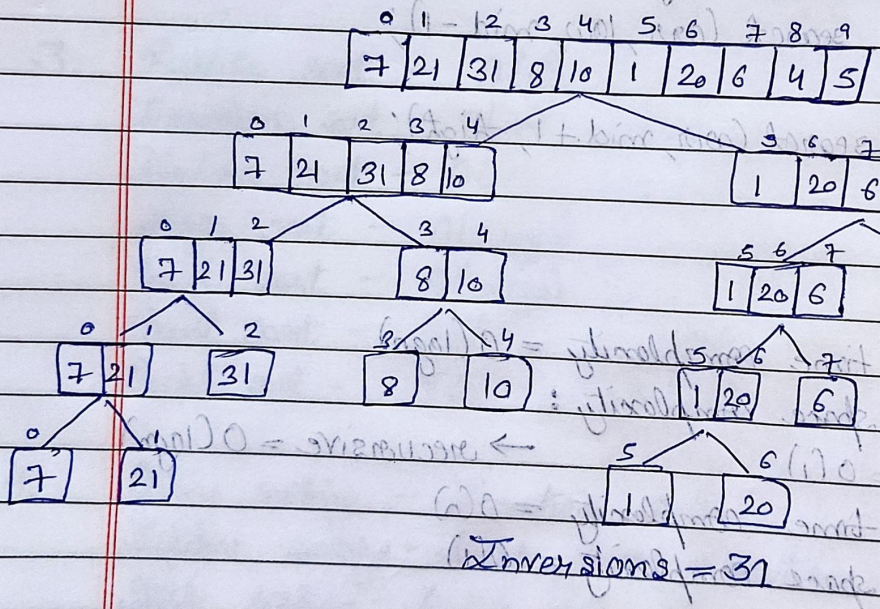
7.

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

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

9 Inversion indicates how far or close the array is from being sorted



10 Worst case → occurs when the pivot is always an extreme (smallest or largest) element. This happens when i/p array is sorted or reverse sorted and either first or last element is picked as pivot. $O(n^2)$
Best case → occurs when pivot element is middle or next to middle. $O(n \log n)$

Page Date

11. Merge sort : $T(n) = 2T(n/2) + O(n)$
Quick sort : $T(n) = 2T(n/2) + n + 1$

Basic	Quick Sort	Merge Sort
• Partition	splitting is done in any ratio	array is parted into just 2 halves
• Works well on	smaller array	any size of array
• Efficient	inefficient for larger array	more efficient
• Sorting Method	Internal	External
• Stability	Not stable	stable

12. Selection sort can be made stable if instead of swapping, the minimum element is placed in its position without swapping i.e. by placing the number in its position by pushing every element one step forward.

```
void stableselectionsort (int a[], int n) {
    for (int i=0; i<n; i++) {
        int min=i;
        for (int j=i+1; j<n; j++)
            if (a[min]>a[j])
                min=j;
        int key = a[min];
        while (min>i) {
            a[min] = a[min-1];
            min--; } a[i] = key; }
```

23. The easiest way to do this is to use external sorting (merge sort). We divide our source file into temporary files of size equal to the size of the RAM & first sort these files.

- External Sorting → If the input data is such that it cannot be adjusted in the m/m entirely at once, it needs to be sorted in a hard disk, floppy disk or any any other storage device. This is called external sorting.

- Internal Sorting → If the input data is such that it can be adjusted in the main m/m at once, it is called internal sorting.