Agenda of today's live session

Counting of number of inversions

Problem Statement :

Input : An array of n elements is given

Output : Find the counting of number of inversions in the given array

Constraints of Inversion :

i < j (where i and j are the indexes of an array)

a[i] > a[j] (elements of i and j index in an array)

And if both of above conditions satisfies, in that case we count it as an inversion.

3	**8**	**0**	**-4**	**1**
1	**2**	**3**	**4**	**5**

(1,3)	**(2,3)**	**(3,4)**
(1,4)	**(2,4)**	
(1,5)	**(2,5)**	

Overall count of number of inversions in an array - 7

70	**50**	**60**	**10**	**20**	**30**	**80**	**15**
1	**2**	**3**	**4**	**5**	**6**	**7**	**8**

Using Divide and Conquer Approach, we see that the number of inversions came out to be 17

Approach : Divide and Conquer Approach

1. Small Problem : If an array is having single element in that case count of number of inversions will return as 0.

2. Big Problem : If an array contains more than one element in that case it will be consider as a big problem and here we apply our Divide and Conquer Strategy.

Implementation :

```
public class Inversions {
  private static int merge_countInversions(int[] arr, int l, int m, int r)      O(n)
  {
    int[] left = Arrays.copyOfRange(arr, l, m + 1);
    int[] right = Arrays.copyOfRange(arr, m + 1, r + 1);
    int i = 0, j = 0, k = l, swaps = 0;
    while (i < left.length && j < right.length) {
      if (left[i] <= right[j])
        arr[k++] = left[i++];
      else {
        arr[k++] = right[j++];
        swaps += (m + 1) - (l + i);
      }
    }
    return swaps;
  }
  // Merge Sort function
  private static int mergeSort_countInversions(int[] arr, int l, int r)
  {
    int count = 0;
    if (l < r) {
      int m = (l + r) / 2;          O(1)
```

```
            count += mergeSort_countInversions(arr, l, m);     // 0                    T(n/2)
            count += mergeSort_countInversions(arr, m + 1, r); // 0 + 0 = 0      T(n/2)
            count += merge_countInversions(arr, l, m, r);        // 0 + 0 + 1 = 1
        }
        return count;
    }
   public static void main(String[] args)
    {
        Scanner s = new Scanner(System.in);
        int n = s.nextInt();
        int arr[] = new int[n];
        for(int i=0;i<n;i++){
            arr[i] = s.nextInt();
        }
        System.out.println(mergeSort_countInversions(arr, 0, n - 1));
    }
}
```

Recurrence Relation :

T(n) = 2T(n/2) + O(n) = O(nlogn)

Strassen's Matrix Multiplication :

Recurrence Relation :

$$T(n) = 7T(n/2) + n^2$$

$$= O(n^{2.81})$$

Problems to be discussed :

Problem 1 :

```
A(n) {
    if(n <= 1)
        return 1;
    else
        return (A(n/2) + A(n/2) + n);
}
```

What is the time complexity of above program ?

a. O(nlogn)

b. O(n^2)

c. O(n^3)

d. O(n) Correct

Problem 2 :

Assume M(n) function is taking O(n^2) as the time complexity

```
A(n) {
    if(n < 1)
        return (n^2+n+1);
    else
        return (5A(n/2) + 3A(n/2) + M(n));
}
```

M(n) -> n^2

a. O(nlogn)

b. O(n^2)

c. O(n^3)

d. O(n)

Problem 3 :

```
int  DoSomething (int n) {
   if (n <= 2)
      return 1;
   else
      return (DoSomething (floor (sqrt(n))) + n);
}
```

T(n) = O(loglogn)