4rd Jan 2023

FIRST :-

Maximum Profit By Choosing A Subset Of Intervals : Medium

Given a list **intervals** of **n** intervals, the **ith** element **[s, e, p]** denotes the starting point **s**, ending point **e**, and the profit **p** earned by choosing the **ith** interval. Find the maximum profit one can achieve by choosing a subset of non-overlapping intervals.

Two intervals **[s1, e1, p1]** and **[s2, e2, p2]** are said to be non-overlapping if **[e1 <= s2]** and **[s1 < s2]**.

Example 1:

```
Input: n = 3
intervals = {
{1, 2, 4},
{1, 5, 7},
{2, 4, 4}
}
Output:
8
Explanation:
One can choose intervals [1, 2, 4] and [2, 4, 4] for a
profit of 8.
```

Example 2:

```
Input: n = 3
intervals = {
{1, 4, 4},
{2, 3, 7},
{2, 3, 4}
}
Output:
7
Explanation:
One can choose interval [2, 3, 7] for a profit of 7.
```

Your Task:

You don't need to print or output anything. Complete the function **maximum_profit()** which takes an integer **n** and a 2D integer array **intervals** and returns an integer, denoting the maximum profit which one can get by choosing the non-overlapping intervals.

Constraints:

- 1 <= n and n <= 10^4
- 1 <= starting point of **ith** interval < ending point of **ith** interval <= 10^5
- 1 <= profit earned by choosing **ith** interval <= 10^5

CODE SECTION:-

```
class Solution {
public:
    int maximum_profit(int n, vector<vector<int>> &intervals) {
        // Write your code here.
sort(intervals.begin(),intervals.end());
vector<vector<int>>dp(n+1,vector<int>(n+1));


        for(int i=n-1;i>=0;i--) {
for(int j=-1;j<i;j++) {                int one
= dp[i+1][j+1], two = 0;
               if(j == -1 or intervals[i][0] >= intervals[j][1])
two = intervals[i][2] + dp[i+1][i+1];                dp[i][j+1] =
max(one, two);
           }        }
return dp[0][0];
    }
};
```

SECOND:-

Count Inversions

Given an array of integers. Find the Inversion Count in the array.

Inversion Count: For an array, inversion count indicates how far (or close) the array is from being sorted. If array is already sorted then the inversion count is 0. If an array is sorted in the reverse order then the inversion count is the maximum.
Formally, two elements a[i] and a[j] form an inversion if a[i] > a[j] and i < j.

Example 1:

```
Input: N = 5, arr[] = {2, 4, 1, 3, 5}
Output: 3
Explanation: The sequence 2, 4, 1, 3, 5  has
three inversions (2, 1), (4, 1), (4, 3).
```

Example 2:

```
Input: N = 5
```

```
arr[] = {2, 3, 4, 5, 6}
Output: 0
Explanation: As the sequence is already
sorted so there is no inversion count.
```

Example 3:

```
Input: N = 3, arr[] = {10, 10, 10}
Output: 0
Explanation: As all the elements of array
are same, so there is no inversion count.
```

Your Task:
You don't need to read input or print anything. Your task is to complete the function **inversionCount()** which takes the array arr[] and the size of the array as inputs and returns the inversion count of the given array.

Expected Time Complexity: O(NLogN).
Expected Auxiliary Space: O(N).

Constraints:
$1 \leq N \leq 5*10^5$
$1 \leq arr[i] \leq 10^{18}$

```cpp
class Solution
{
public:
    // arr[]: Input Array
    // N : Size of the Array arr[]
    // Function to count inversions in the array.
```

```
void merge(long long arr[], long i, long m, long j, long &ans)
{
    long len1 = m - i + 1;
    long len2 = j - m;
    long *first = new long[len1];
    long *second = new long[len2];

    int k = i;
    for (int a = 0; a < len1; a++)
    {
        first[a] = arr[k++];
    }
    for (int a = 0; a < len2; a++)
    {
        second[a] = arr[k++];
    }
    int a = 0;
    int b = 0;
    k = i;
    while (a < len1 and b < len2)
    {
        if (first[a] <= second[b])
        {
            arr[k++] = first[a++];
        }
        else
        {
            arr[k++] = second[b++];
            ans = ans + len1 - a;
        }
    }
    while (a < len1)
    {
        arr[k++] = first[a++];
    }
    while (b < len2)
    {
        arr[k++] = second[b++];
    }
    delete[] first;
    delete[] second;
}
void mergeSort(long long arr[], long i, long j, long &ans)
{
    if (j <= i)
    {
        return;
    }
```

```
        int mid = i + (j - i) / 2;
        mergeSort(arr, i, mid, ans);
        mergeSort(arr, mid + 1, j, ans);
        merge(arr, i, mid, j, ans);
    }

    long long int inversionCount(long long arr[], long long N)
    {
        long ans = 0;
        mergeSort(arr, 0, N - 1, ans);
        return ans;
    }
};
```

HELP SECTION:-

1. This question can be done using two for loop very easily ,but it will take $O(N^2)$ {brute forece method}.
2. We can optimize this question using merge sort ,and then it can be solved in O(nlogn)
    a. Apply merge sort
    b. And count how much time j is greater than i
    c. Return the count

-: DONE FOR TODAY:-