Searching & Sorting Applications

1. Aggressive Cows

Problem Statement:

Farmer John has built a new long barn, with N (2 <= N <= 100,000) stalls. The stalls are located along a straight line at positions x1,...,xN (0 <= xi <= 1,000,000,000).

His C (2 <= C <= N) cows don't like this barn layout and become aggressive towards each other once put into a stall. To prevent the cows from hurting each other, FJ wants to assign the cows to the stalls, such that the minimum distance between any two of them is as large as possible. What is the largest minimum distance?

Explanation:

Approach 1:

We are given **N stalls (2 <= N** <= **10^6)** and **C cows (2 <= C <= N)** and we have to place the cows in these stalls such that there is largest minimum distance between them. So suppose we are given the input **N = 5** and **C = 3** and the position of the stalls are at **1 2 8 4 9**.

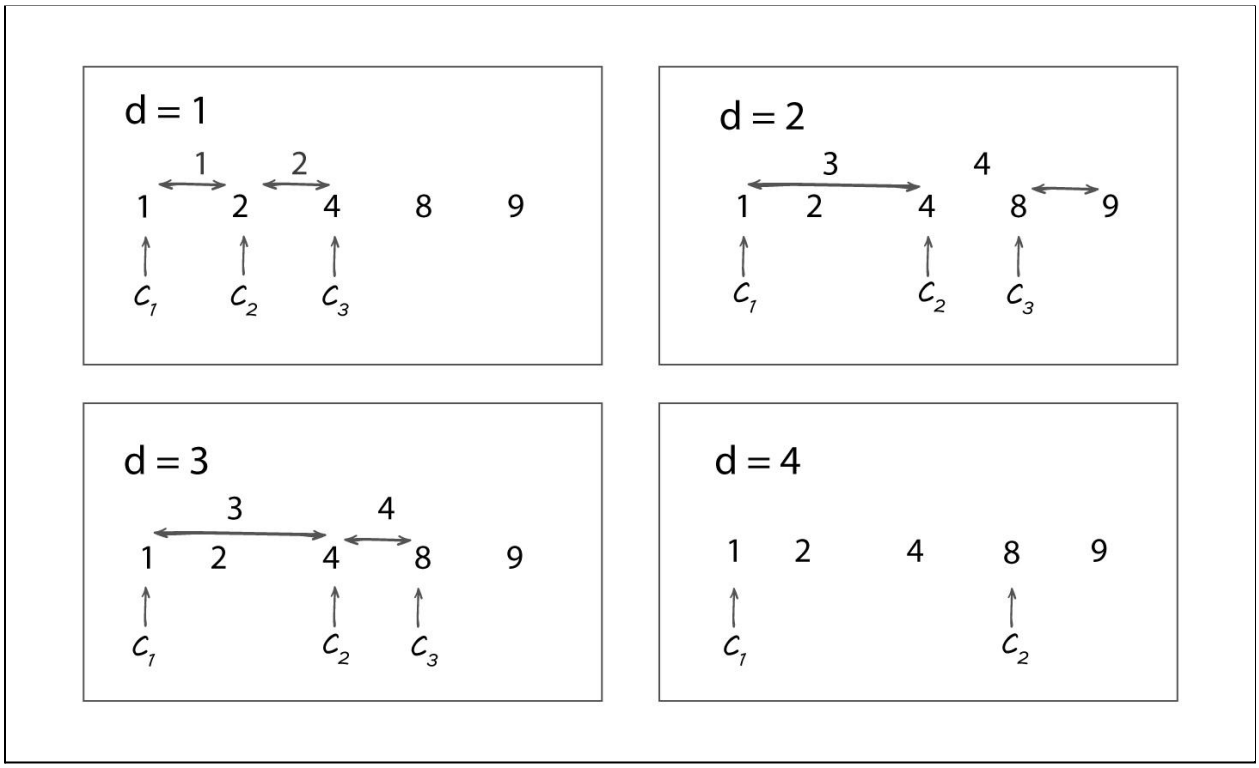
One approach that comes to mind is to arrange these positions of stalls in a **sorted array [1, 2, 4, 8, 9]**. Now, the **min_distance = 0** if we place C1 and C2 at positions 1 & 2; and the **max_distance = 9 - 1 = 8** if the two cows are at the ends of the array. That is, **0 <= distance (d) <= 8**

0 <= d <= position[N-1] - position[0]

Now, for every possible **d,** we check whether we can place C = 3 cows or not and amongst the possible answers we choose the maximum value as our answer.

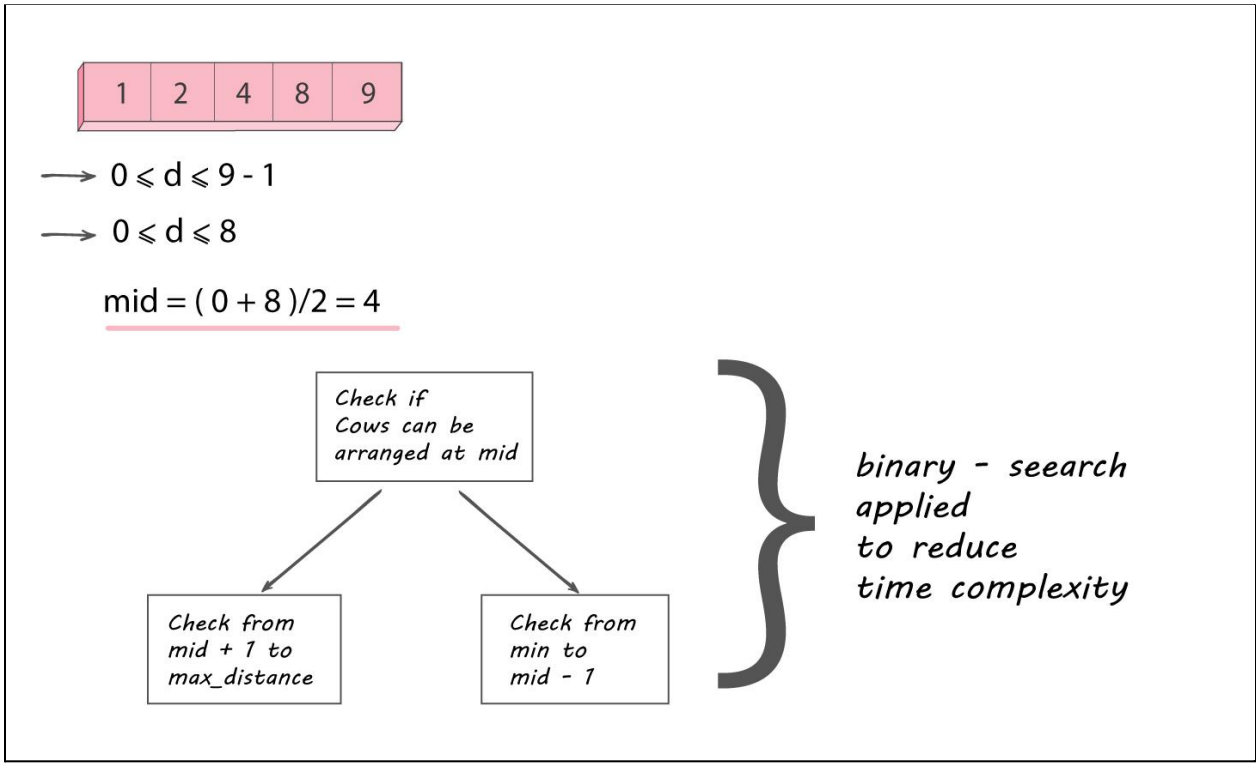
Problem with this approach:

In the above solution we are traversing and solving for every possible value of **d** so the time complexity of this solution is **O(d*N)** which in the worst case will be 10^9 * 10^6 = 10^{14} that is not acceptable. Therefore, we must think of another approach to solve this problem.



Approach 2:

In approach 1 we were checking for every possible **d** but now we will apply the concept of **binary search** to find out upto which d we can get the answer to reduce the searching process to **O(log(d)).** That is, we find
mid = (min_distance + max_distance)/2;
Now if it is possible to place the cows at mid distance from each other then we will apply binary search from mid + 1 to max_distance otherwise we will apply binary search from min to mid - 1.




```cpp
#include<bits/stdc++.h>
using namespace std;

bool check(int cows,long long positions[],int n,long long distance){

    int count = 1;
    long long last_position = positions[0];

    for(int i=1;i<n;i++){
```

```
        if(positions[i] - last_position >= distance){
            last_position = positions[i];
            count++;
        }

        if(count == cows){
            return true;
        }
    }
    return false;
}

int main(){
    int t;
    cin >> t;
    while(t--){
        int n,c;
        cin >> n >> c;

        long long positions[n];
        for(int i=0;i<n;i++){
            cin >> positions[i];
        }
        sort(positions,positions+n);
        long long start = 0;
        long long end = positions[n-1] - positions[0];

        long long ans = -1;

        while(start<=end){
            long long mid = start + (end-start)/2;

            if(check(c,positions,n,mid)){
                ans = mid;
                start = mid+1;
            }else{
                end = mid-1;
            }

        }
        cout << ans <<endl;
```

```
    }

    return 0;
}
```

2. Inversion Count

Problem Statement:

For a given integer array/list of size N, find the total number of 'Inversions' that may exist.

An inversion is defined for a pair of integers in the array/list when the following two conditions are met.

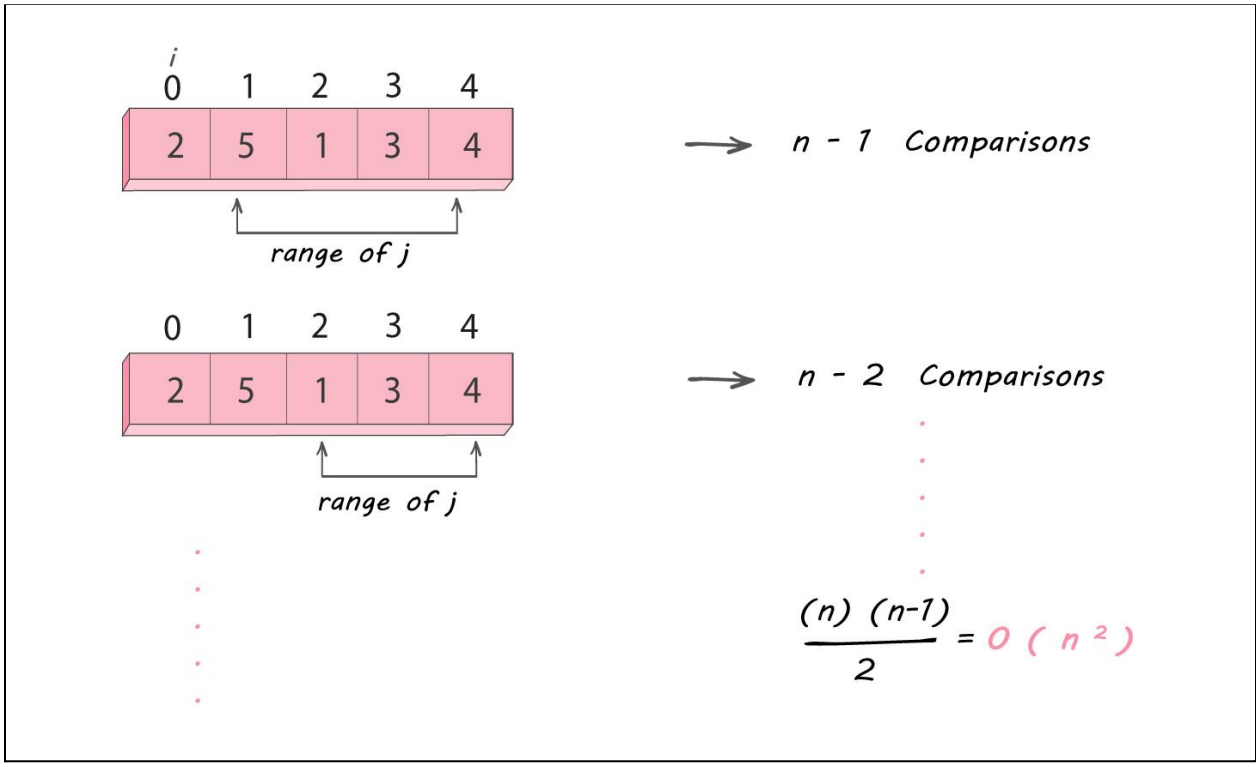
A pair (arr[i], arr[j]) is said to be an inversion when,

1. arr[i] > arr[j]
2. i < j

Where 'arr' is the input array/list, 'i' and 'j' denote the indices ranging from [0, N).

Explanation:

Approach 1: Consider the A = [2, 5, 1, 3, 4]. For this we have to calculate the number of inversions. An inversion is counted when **A[i] > A[j] given that, i < j**. The basic approach that comes to mind is to compare each element with all the elements present in the array from the next index. That is, we use 2 loops and **for each index i we compare A[i] with A[i+1] to A[N-1].**



The time complexity of this approach will be **O(N^2)** and is not acceptable in this problem.

Approach 2: If we divide the array in two parts and then calculate the number of inversions in each part and at the time of merging calculate the newly formed inversions then also we are traversing at almost every element to compare.

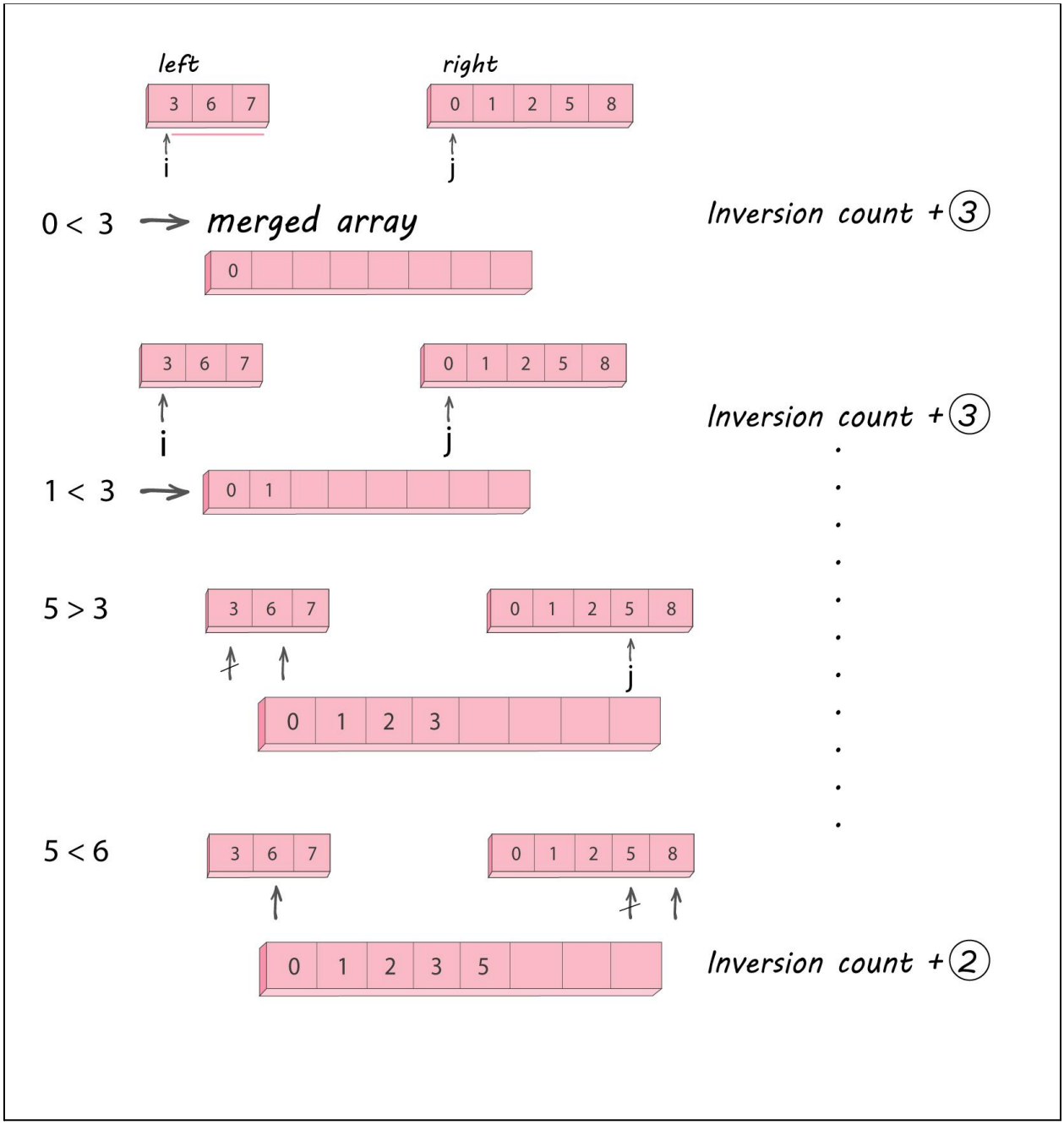
Approach 3: If we add a slight modification to Approach 2 that before merging if we sort the array then it will be easy for us to calculate the newly formed inversions without comparing each and every element.

This approach can be divided into the following steps:

Step 1: Divide the array into two parts.

Step 2: Merge Sort left and right arrays + Calculate the inversion count in respective arrays

Step 3: Total Inversion Count = Left inversion count + Right inversion count + New formed inversions during merging



Now, to calculate the inversion counts while merging let us suppose we have **left array = [3, 6, 7] and right array = [0, 1, 2, 5, 8]**

Pointer i and j are at 3 and 0 respectively

First iteration : 0 < 3 therefore, since both arrays are sorted that means 0 will be smaller to every element in the left array so **increase the inversion count by 3**, add 0 to the merged array and move j pointer to 1.
Second iteration : 1 < 3 therefore increase the inversion count by 3, add 1 to the merged array and move j pointer to 2.
Third iteration : 2 < 3 therefore increase the inversion count by 3, add 2 to the merged array and move j pointer to 5.
Fourth iteration : 5 > 3 so now add 3 to the merged array, move i pointer to 6.
Fifth iteration : 5 < 6 therefore increase the inversion count by 2, add 5 to the merged array and move j pointer to 8.
Continue these iterations while(i<mid && j<=right)

```
#include<iostream>
using namespace std;


long long merge(int A[],int left,int mid,int right){

    int i=left,j=mid,k=0;

    int temp[right-left+1];
    long long count = 0;
    while(i<mid && j<=right){
        if(A[i] <= A[j]){
            temp[k++] = A[i++];
        }else{
            temp[k++] = A[j++];
            count += mid - i;
        }
    }
    while(i<mid){
        temp[k++] = A[i++];
    }
```

```
    while(j<=right){
        temp[k++] = A[j++];
    }

    for(int i=left,k=0;i<=right;i++,k++){
        A[i] = temp[k];
    }
    return count;
}
long long merge_sort(int A[],int left,int right){

    long long count = 0;
    if(right > left){
        int mid = (left + right)/2;

        long long countLeft = merge_sort(A,left,mid);
        long long countRight = merge_sort(A,mid+1,right);
        long long myCount = merge(A,left,mid+1,right);

        return myCount + countLeft + countRight;
    }
    return count;

}
long long solve(int A[], int n)
{
    long long ans = merge_sort(A,0,n-1);
    return ans;
}

int main(){
    int A[] = {5,4,2,3,1};
    cout << solve(A,5);
    return 0;
}
```