Aggressive Cows : Detailed Solution

Problem Statement: You are given an array **‘arr’** of size **‘n’** which denotes the position of stalls.
You are also given an integer **‘k’** which denotes the number of aggressive cows.
You are given the task of assigning stalls to **‘k’** cows such that the minimum distance between any two of them is the maximum possible.
Find the maximum possible minimum distance.

Example 1:

Input Format: N = 6, k = 4, arr[] = {0,3,4,7,10,9}

Result: 3

Explanation: The maximum possible minimum distance between any two cows will be 3 when 4 cows are placed at positions {0, 3, 7, 10}. Here the distances between cows are 3, 4, and 3 respectively. We cannot make the minimum distance greater than 3 in any ways.

Example 2:

Input Format: N = 5, k = 2, arr[] = {4,2,1,3,6}

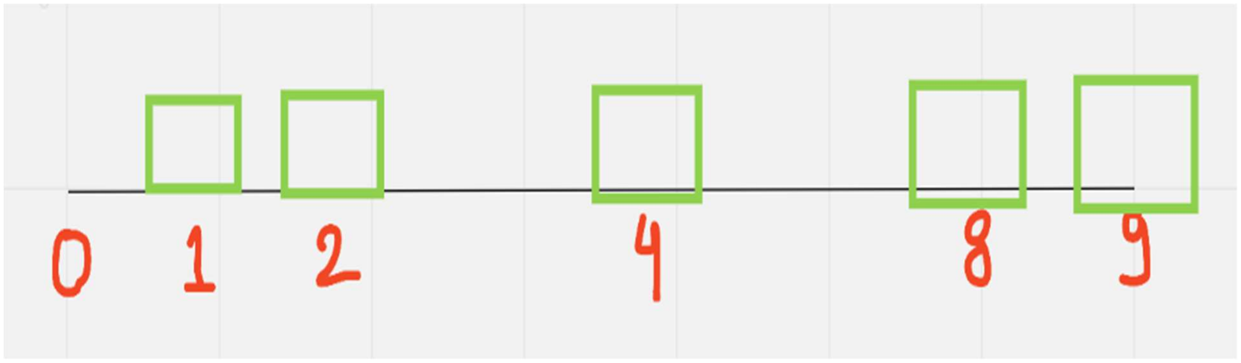
Result: 5

Explanation: The maximum possible minimum distance between any two cows will be 5 when 2 cows are placed at positions {1, 6}.

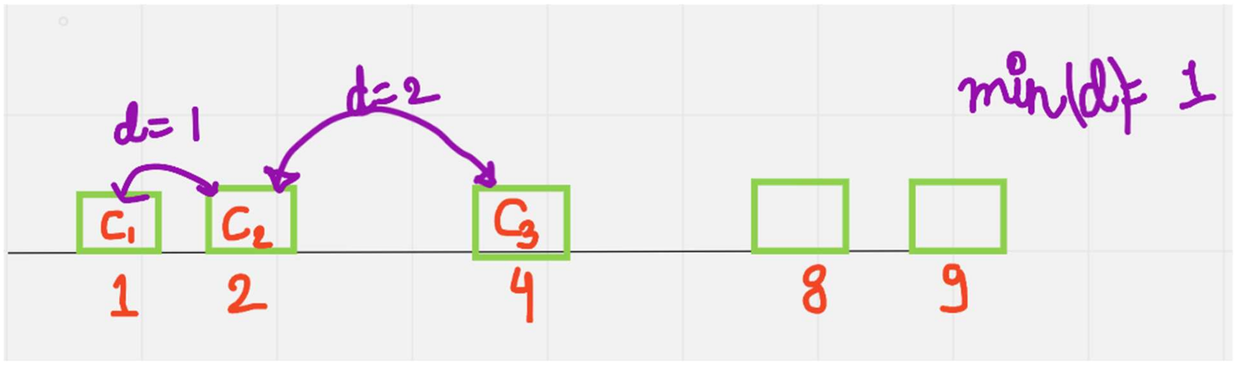
Why do we need to sort the stalls?

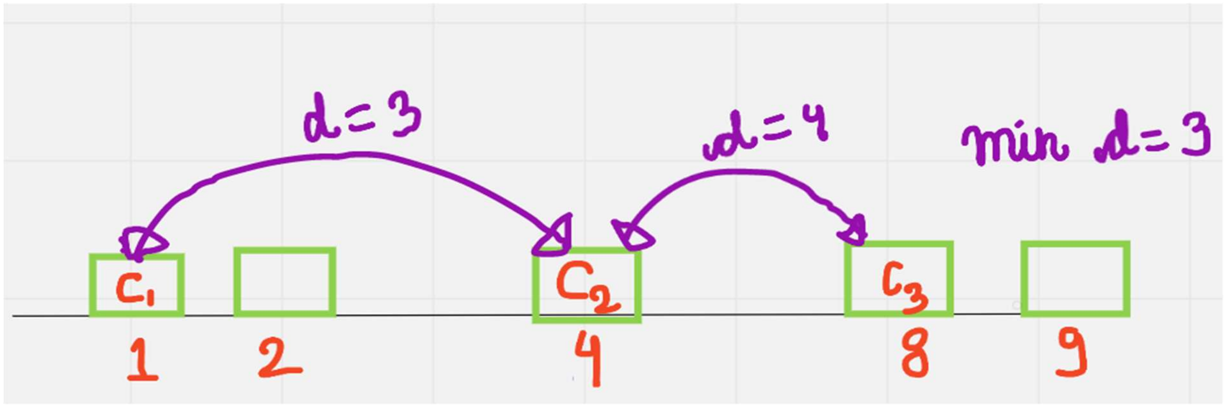
To arrange the cows in a consecutive manner while ensuring a certain distance between them, the initial step is to sort the stalls based on their positions. In a sorted array, the minimum distance will always be obtained from any two consecutive cows. Arranging the cows in a consecutive manner does not necessarily mean placing them in consecutive stalls.

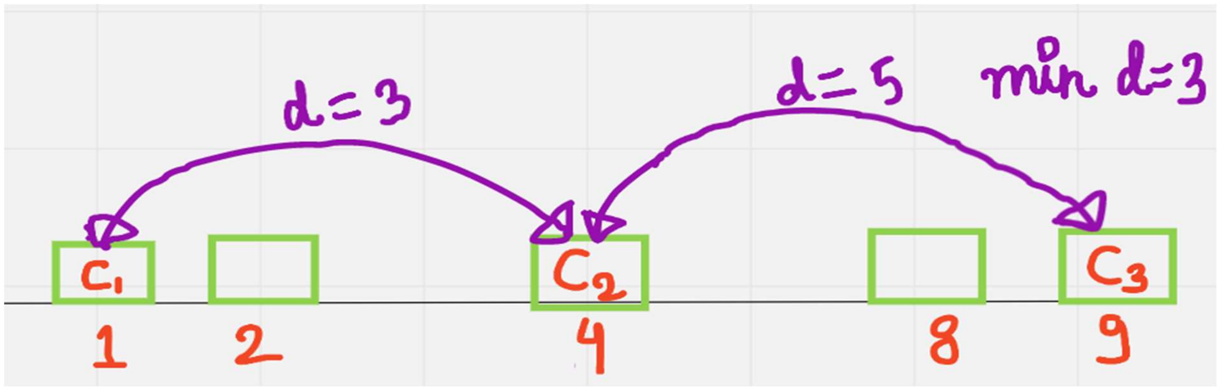
Assume the given stalls array is: {1,2,8,4,9} and after sorting it will be {1, 2, 4, 8, 9}. The given number of cows is 3.



We have to fit three cows in these 5 stalls. Each stall can accommodate only one. Our task is to **maximize** the **minimum** distance between two stalls. Let’s look at some arrangements:







In the first arrangement, the minimum distance between the cows is 1. Now, in the later cases, we have tried to place the cows in a manner so that the minimum distance can be increased. This is done in the second and third cases. It’s not possible to get a minimum distance of more than 3 in any arrangement, so we output 3.

Observation:

- **Minimum possible distance between 2 cows:** The minimum possible distance between two cows is 1 as the minimum distance between 2 consecutive stalls is 1.
- **Maximum possible distance between 2 cows:** The maximum possible distance between two cows is = max(stalls[])-min(stalls[]). This case occurs when we place 2 cows at two ends of the sorted stalls array.

From the observations, ***we can conclude that our answer lies in the range***
[1, max(stalls[])-min(stalls[])].

How to place cows with maintaining a certain distance, ‘dist’, in the sorted stalls:

To begin, we will position the first cow in the very first stall. Next, we will iterate through the array, starting from the second stall. If the distance between the current stall and the last stall where a cow was placed is greater than or equal to the value 'dist', we will proceed to place the next cow in the current stall. Thus we will try to place the cows and finally, we will check if we have placed all the cows maintaining the distance, ‘dist’.

To serve this purpose, we will write a function **canWePlace()** that takes the distance, 'dist', as a parameter and returns true if we can place all the cows maintaining a minimum distance of 'dist'. Otherwise, it returns false.

canWePlace(stalls[], dist, k):

1. We will declare two variables, 'cntCows' and 'last'. **'cntCows' will store the number of cows placed**, and **'last' will store the position of the last placed cow**.
2. First, we will place the first cow in the very first stall. So, we will set 'cntCows' to 1 and 'last' to stall[0].
3. Then, using a loop we will start iterating the array from index 1. Inside the loop, we will do the following:
    1. **If stalls[i] - 'last' >= dist:** This means the current stall is at least 'dist' distance away from the last stall. So, we can place the next cow here. We will increase the value 'cntCows' by 1 and set 'last' to the current stall.
    2. **If cntCows >= k:** This means we have already placed k cows with maintaining the minimum distance 'dist'. So, we will return true from this step.
4. If we are outside the loop, we cannot place k cows with a minimum distance of 'dist'. So, we will return false.

Naive Approach:

The extremely naive approach is to check all possible distances from 1 to max(stalls[])-min(stalls[]). The maximum distance for which the function **canWePlace()** returns true, will be our answer.

Algorithm:

1. First, we will sort the given stalls[] array.
2. We will use a loop(say **i**) to check all possible distances.
3. Next, inside the loop, we will send each distance, i, to the function **canWePlace()** function to check if it is possible to place the cows.

1. We will return (i-1), where i is the minimum distance for which the function **canWePlace()** returns false. Because (i-1) is the maximum distance for which we can place all the cows and for the distances >= i, it becomes impossible.

4. Finally, if we are outside the loop, we can conclude the minimum possible distance should be max(stalls[])-min(stalls[]). And we will return this value.

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

bool canWePlace(vector<int> &stalls, int dist, int cows) {
    int n = stalls.size(); //size of array
    int cntCows = 1; //no. of cows placed
    int last = stalls[0]; //position of last placed cow.
    for (int i = 1; i < n; i++) {
        if (stalls[i] - last >= dist) {
            cntCows++; //place next cow.
            last = stalls[i]; //update the last location.
        }
        if (cntCows >= cows) return true;
    }
    return false;
}
int aggressiveCows(vector<int> &stalls, int k) {
    int n = stalls.size(); //size of array
    //sort the stalls[]:
    sort(stalls.begin(), stalls.end());
    int limit = stalls[n - 1] - stalls[0];
    for (int i = 1; i <= limit; i++) {
        if (canWePlace(stalls, i, k) == false) {
            return (i - 1);
        }
    }
    return limit;
}

int main()
{
    vector<int> stalls = {0, 3, 4, 7, 10, 9};
    int k = 4;
    int ans = aggressiveCows(stalls, k);
    cout << "The maximum possible minimum distance is: " << ans << "\n";
    return 0;
}
```

Complexity Analysis

Time Complexity: O(NlogN) + O(N *(max(stalls[])-min(stalls[]))), where N = size of the array, max(stalls[]) = maximum element in stalls[] array, min(stalls[]) = minimum element in stalls[] array.

Reason: O(NlogN) for sorting the array. We are using a loop from 1 to max(stalls[])-min(stalls[]) to check all possible distances. Inside the loop, we are calling canWePlace() function for each distance. Now, inside the canWePlace() function, we are using a loop that runs for N times.

Space Complexity: O(1) as we are not using any extra space to solve this problem.

Algorithm / Intuition

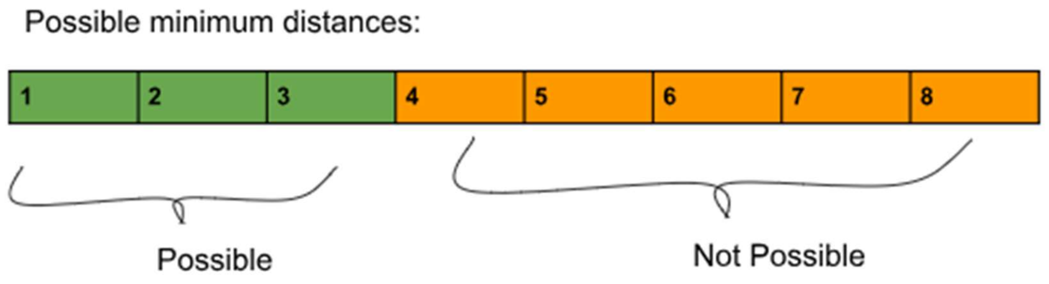
Optimal Approach(Using Binary Search):

We are going to use the Binary Search algorithm to optimize the approach.

The primary objective of the Binary Search algorithm is to efficiently determine the appropriate half to eliminate, thereby reducing the search space by half. It does this by determining a specific condition that ensures that the target is not present in that half.

Upon closer observation, we can recognize that our answer space, represented as [1, (max(stalls[])-min(stalls[]))], is actually sorted. Additionally, we can identify a pattern that allows us to divide this space into two halves: one consisting of potential answers and the other of non-viable options. So, we will apply binary search on the answer space.

For example, the given array is {1, 2, 8, 4, 9}. The possible distances are the following:



Algorithm:

1. **First,** we will sort the given stalls[] array.

2. **Place the 2 pointers i.e. low and high:** Initially, we will place the pointers. The pointer low will point to 1 and the high will point to (stalls[n-1]-stalls[0]). As the 'stalls[]' is sorted, 'stalls[n-1]' refers to the maximum, and 'stalls[0]' is the minimum element.
3. **Calculate the 'mid':** Now, inside the loop, we will calculate the value of 'mid' using the following formula:
   mid = (low+high) // 2 ('//' refers to integer division)
4. **Eliminate the halves based on the boolean value returned by canWePlace():**
   We will pass the potential distance, represented by the variable 'mid', to the **'canWePlace()'** function. This function will return true if it is possible to place all the cows with a minimum distance of 'mid'.
   1. **If the returned value is true:** On satisfying this condition, we can conclude that the number 'mid' is one of our possible answers. But we want the maximum number. So, we will eliminate the left half and consider the right half(i.e. low = mid+1).
   2. **Otherwise,** the value mid is greater than the distance we want. This means the numbers greater than 'mid' should not be considered and the right half of 'mid' consists of such numbers. So, we will eliminate the right half and consider the left half(i.e. high = mid-1).
5. Finally, outside the loop, we will return the value of high as the pointer will be pointing to the answer.

The steps from 3-4 will be inside a loop and the loop will continue until low crosses high.

Note: *It is always the opposite polarity. Initially the pointer 'high' was in the 'not-possible' half and so it ends up in the 'possible' half. Similarly, 'low' was initially in the 'possible' part and it ends up in the 'not-possible' part.*

Note: *Please make sure to refer to the video and try out some test cases of your own to understand, how the pointer 'high' will be always pointing to the answer in this case. This is also the reason we have not used any extra variable here to store the answer.*

```
#include <bits/stdc++.h>

using namespace std;
```

```
bool canWePlace(vector<int> &stalls, int dist, int cows) {
    int n = stalls.size(); //size of array
    int cntCows = 1; //no. of cows placed
    int last = stalls[0]; //position of last placed cow.
    for (int i = 1; i < n; i++) {
        if (stalls[i] - last >= dist) {
            cntCows++; //place next cow.
            last = stalls[i]; //update the last location.
        }
        if (cntCows >= cows) return true;
    }
    return false;
}
int aggressiveCows(vector<int> &stalls, int k) {
    int n = stalls.size(); //size of array
    //sort the stalls[]:
    sort(stalls.begin(), stalls.end());

    int low = 1, high = stalls[n - 1] - stalls[0];
    //apply binary search:
    while (low <= high) {
        int mid = (low + high) / 2;
        if (canWePlace(stalls, mid, k) == true) {
            low = mid + 1;
        }
        else high = mid - 1;
    }
    return high;
}


int main()

{

   vector<int> stalls = {0, 3, 4, 7, 10, 9};

   int k = 4;

   int ans = aggressiveCows(stalls, k);

   cout << "The maximum possible minimum distance is: " << ans << "\n";

   return 0;

}
```

Complexity Analysis

Time Complexity: O(NlogN) + O(N * log(max(stalls[])-min(stalls[]))), where N = size of the array, max(stalls[]) = maximum element in stalls[] array, min(stalls[]) = minimum element in stalls[] array.
Reason: O(NlogN) for sorting the array. We are applying binary search on [1, max(stalls[])-min(stalls[])]. Inside the loop, we are calling canWePlace() function for each distance, 'mid'. Now, inside the canWePlace() function, we are using a loop that runs for N times.

Space Complexity: O(1) as we are not using any extra space to solve this problem.