Nth Root of a Number using Binary Search

Problem Statement: Given two numbers N and M, find the Nth root of M.
The nth root of a number M is defined as a number X when raised to the power N equals M.

Example 1:

```
Input: N=3 M=27
Output: 3
Explanation: The cube root of 27 is 3.
```

Example 2:

```
Input: N=2 M=16
Output: 4
Explanation: The square root of 16 is 4
```

Example 3:

```
Input: N=5 M=243
Output: 3
Explaination: The 5th root of 243 is 3
```

Disclaimer: *Don't jump directly to the solution, try it out yourself first.*

Solution:

Nth root of a number M is defined as a number X when raised to the power N equals to M.

Approach: In order to find the Nth root of any given integer we are gonna use Binary Search.
Step 1: Take low and high. Low will be equal to 1 and high will be M. We will take the mid of low and high such that the searching space is reduced using low + high / 2.
Step 2: Make a separate function to multiply mid N times.
Step 3: Run the while loop till (high – low > eps). Take eps as 1e-6, since we have to find answers to 6 decimal places.
Step 4: If the mid is less than M, then we will reduce search space to low and mid. Else, if it is greater than M then search space will be reduced to mid and high.
Step 5: After the loop breaks we will have our answer as low or high.
We have to find the answer to 6 decimals. So, we will have a double 1e-6. We will run the while loop till (high – low > eps). When we will come out of the loop we will have the answer which will be equal to low as well as high.

Code:

```
#include <bits/stdc++.h>
using namespace std;
double multiply(double number, int n) {
    double ans = 1.0;
    for(int i = 1;i<=n;i++) {
        ans = ans * number;
    }
    return ans;
}
```

```
void getNthRoot(int n, int m) {
    double low = 1;
    double high = m;
    double eps = 1e-6;

    while((high - low) > eps) {
        double mid = (low + high) / 2.0;
        if(multiply(mid, n) < m) {
            low = mid;
        }
        else {
            high = mid;
        }
    }

    cout <<n<<"th root of "<<m<<" is "<<low<<endl;

}
int main() {
     int n=3, m=27;
     getNthRoot(n, m);
     return 0;
}
```

Output: 3th root of 27 is 3
Time Complexity: N x log(M x 10^d)
Space Complexity: O(1)

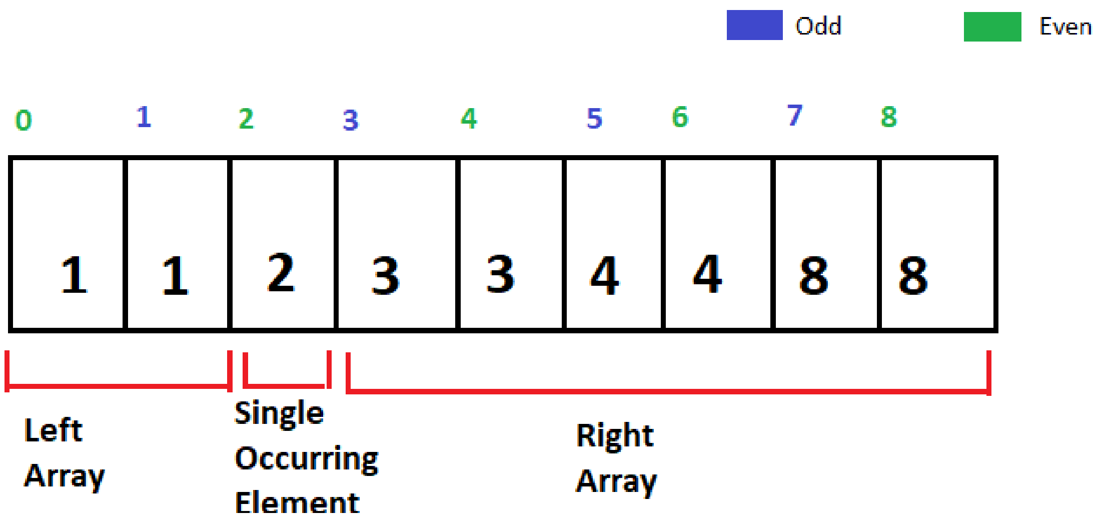
Search Single Element in a sorted array

Problem Statement: Given a sorted array of N integers, where every element except one appears exactly twice and one element appears only once. Search Single Element in a sorted array.

Example 1:
Input: N = 9, array[] = {1,1,2,3,3,4,4,8,8}
Output: 2
Explanation: Every element in this sorted array except 2 appears twice, therefore the answer returned will be 2.
Example 2:
Input: N = 7, array[] = {11,22,22,34,34,57,57}
Output: 11
Explanation: Every element in this sorted array except 11 appears twice, therefore the answer returned will be 11.

Solution
Disclaimer: *Don't jump directly to the solution, try it out yourself first.*
Solution 1: Using XOR(^)
Approach: As every number in the array repeats twice and only one number occurs once, we can take advantage of the XOR(^) operator. These are two properties of the XOR operator which will be helpful.
If p is a number then,
p^p=0
p^0=p
If we find the XOR of every element of the array, we will get the answer.
Code:
```
#include<bits/stdc++.h>

using namespace std;
class Solution {
    public:
        int findSingleElement(vector < int > & nums) {

            int n = nums.size();
            int elem = 0;
            for (int i = 0; i < n; i++) {
                elem = elem ^ nums[i];
            }

            return elem;
        }
};

int main() {
    Solution obj;
    vector < int > v {1,1,2,3,3,4,4,8,8};

    int elem = obj.findSingleElement(v);

    cout << "The single occurring element is "
         << elem << endl;
}
```
Output:
The single occurring element is 2
Time Complexity: O(N)
Space Complexity: O(1)

Solution 2:
Approach: Using Binary Search
As the elements are sorted, twice occurring elements will be placed together in the input array. Moreover, the input array has one element occurring once, therefore a general input can be thought of like this.



Now in this left array, the first instance of every element is occurring on the even index and the second instance on the odd index. Similarly in the right array, the first instance of every element is occurring on the odd index and the second index is occurring on the even index. This is summarized below.

Left Half	Right Half
First Instance => Even Index	First Instance => Odd Index
Second Instance => Odd Index	Second Instance => Even Index

So the algorithmic approach will be to use binary search. The intuition is that when we see an element, if we know its index and whether it is the first instance or the second instance, we can decide whether we are presently in the left array or right array. Moreover, we can decide which direction we need to move to find the breakpoint. We need to find this breakpoint between our left array and the right array.
We will check our mid element, if it is in the left array, we will shrink our left array to the right of this mid element, if it is in the right array, we will shrink the right array to the left of this mid

element. This binary search process will continue till the right array surpasses our left one and the low is pointing towards the breakpoint.

Dry Run: In case you want to see the dry run of this approach, please watch the video at the bottom.

Code:

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

using namespace std;
class Solution {
    public:
        int findSingleElement(vector < int > & nums)
        {
            int low = 0;
            int high = n - 2;

            while (low <= high) {
                int mid = (low + high) / 2;

                if (mid % 2 == 0) {
                    if (nums[mid] != nums[mid + 1])
                    //Checking whether we are in right half

                        high = mid - 1; //Shrinking the right half
                    else
                        low = mid + 1; //Shrinking the left half
                } else {

                    //Checking whether we are in right half
                    if (nums[mid] == nums[mid + 1])
                        high = mid - 1; //Shrinking the right half
                    else
                        low = mid + 1; //Shrinking the left half
                }
            }

            return nums[low];
        }
};

int main() {
    Solution obj;
    vector < int > v {1,1,2,3,3,4,4,8,8
    };
    int elem = obj.findSingleElement(v);
    cout << "The single occurring element is " +
    " << elem << endl;

}
```

Output:
The single occurring element is 2
Time Complexity: O(log(N))
Space Complexity: O(1)

Search Element in a Rotated Sorted Array

Problem Statement: There is an integer array **nums** sorted in ascending order (with distinct values). Given the array **nums** after the possible rotation and an integer **target**, return *the* **index of target** if *it is in* nums*,* or -1 *if it is not in* nums. We need to search a given element in a rotated sorted array.
Example 1:

```
Input: nums = [4,5,6,7,0,1,2,3], target = 0

Output: 7

Explanation: Here, the target is 0. We can see that 0 is present in
the given rotated sorted array, nums. Thus, we get output as 4, which
is the index at which 0 is present in the array.
```

Example 2:

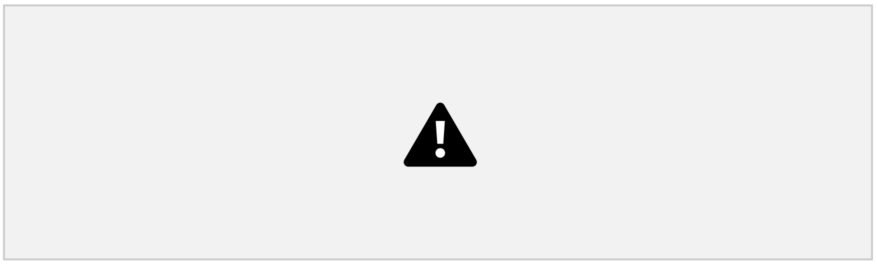
```
Input: nums = [4,5,6,7,0,1,2], target = 3

Output: -1

Explanation: Here, the target is 3. Since 3 is not present in the
given rotated sorted array. Thus, we get output as -1.
```

Solution:
Disclaimer: *Don't jump directly to the solution, try it out yourself first.*
Solution 1: Using Linear Search
Approach :
We will iterate through the array completely. While iterating, we have to check if we have found the target element in the array. If we find it, we will return its index. If we iterate completely and still do not find an element in the array. This indicates the target is not present and hence we return -1 as mentioned in the question.
Code:

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

```
using namespace std;

int search(vector<int>& nums,int target) {
    for(int i=0;i<nums.size();i++)
    {
        if(nums[i]==target)
        return i;
    }
    return -1;

}

int main() {
    vector<int> nums = {4,5,6,7,0,1,2,3};
    int target = 3;
    cout<<"The index in which the target is present is
"<<search(nums,target);

    return 0;
}
```

Time Complexity : O(N)
Reason: We have to iterate through the entire array to check if the target is present in the array.
Space Complexity: O(1)
Reason: We have not used any extra data structures, this makes space complexity, even in the worst case as O(1).
Solution 2: Using Binary Search
Intuition :
It is mentioned that the array given is sorted but in a rotated manner. So, we can divide a given array into two parts that are sorted. This gives us hints to use binary search. We can visualize a given array to be composed of two sorted arrays.



Approach :
We divide the array into parts. It is done using two pointers, low and high, and dividing the range between them by 2. This gives the midpoint of the range. Check if the target is present in the midpoint, calculated before, of the array. If not present, check if the left half of the array is sorted. This implies that binary search can be applied in the left half of the array provided the target lies between the value range. Else check into the right half of the array. Repeat the above

steps until low <= high. If low > high, indicates we have searched array and target is not present hence return -1. Thus, it makes search operations less as we check range first then perform searching in possible ranges which may have target value.

Code:

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

using namespace std;

int search(vector < int > & nums, int target) {
  int low = 0, high = nums.size() - 1; //<---step 1

  while (low <= high) { //<--- step 2
    int mid = (low + high) >> 1; //<----step 3
    if (nums[mid] == target)
      return mid; // <---step 4

    if (nums[low] <= nums[mid]) { //<---step 5
      if (nums[low] <= target && nums[mid] >= target)
        high = mid - 1; //<---step 6
      else
        low = mid + 1; //<---step 7
    } else { //<---step 7
      if (nums[mid] <= target && target <= nums[high])
        low = mid + 1;
      else
        high = mid - 1;
    }
  }
  return -1; //<---step 8
}

int main() {
  vector<int> nums = {4,5,6,7,0,1,2,3};
  int target = 3;
  cout << "The index in which the target is present is " <<
search(nums, target);

  return 0;
}
```

Output: The index in which the target is present is 7

Time Complexity: O(log(N))

Reason: We are performing a binary search, this turns time complexity to O(log(N)) where N is the size of the array.

Space Complexity: O(1)

Reason: We do not use any extra data structure, this turns space complexity to O(1).

Median of Two Sorted Arrays of different sizes

Problem Statement: Given **two sorted arrays** arr1 and arr2 of size m and n respectively, return the **median** of the two sorted arrays.

Example 1:

```
Input format: arr1 = [1,4,7,10,12], arr2 = [2,3,6,15]

Output format : 6.00000

Explanation:
Merge both arrays. Final sorted array is [1,2,3,4,6,7,10,12,15]. We
know that to find the median we find the mid element. Since, the size
of the element is odd. By formula, the median will be at [(n+1)/2]th
position of the final sorted array. Thus, for this example, the
median is at [(9+1)/2]th position which is [5]th = 6.
```

Example 2:

```
Input: arr1 = [1], arr2 = [2]

Output format:
 1.50000

Explanation:

Merge both arrays. Final sorted array is [1,2]. We know that to find
the median we find the mid element. Since, the size of the element is
even. By formula, the median will be the mean of elements at [n/2]th
and  [(n/2)+1]th position of the final sorted array. Thus, for this
example, the median is (1+2)/2 = 3/2 = 1.50000.
```

Solution 1: Naive Approach

Intuition :

The point to observe is that the given arrays are sorted. Our task is to merge them into a sorted array. The word “merge” gives us hints to apply the merge step of merge sort.

Approach :

Take two pointers, each pointing to each array. Take an array of size (m+n) to store the final sorted array. If the first pointed element is smaller than the second one, store that value in an array and move the first pointer ahead by one. Else do the same for the second pointer when the case is vice-versa. Then use the formula to get the median position and return the element present at that position.

Dry Run :

Code:

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

float median(int nums1[],int nums2[],int m,int n) {
    int finalArray[n+m];
    int i=0,j=0,k=0;
    while(i<m && j<n) {
        if(nums1[i]<nums2[j]) {
            finalArray[k++] = nums1[i++];
        }
        else {
            finalArray[k++] = nums2[j++];
        }
    }
    if(i<m) {
        while(i<m)
            finalArray[k++] = nums1[i++];
    }
    if(j<n) {
        while(j<n)
            finalArray[k++] = nums2[j++];
    }
    n = n+m;
    if(n%2==1)
        return finalArray[((n+1)/2)-1];
    else return
((float)finalArray[(n/2)-1]+(float)finalArray[(n/2)])/2;
}
```

```
int main() {
    int nums1[] = {1,4,7,10,12};
    int nums2[] = {2,3,6,15};
    int m = sizeof(nums1)/sizeof(nums1[0]);
    int n = sizeof(nums2)/sizeof(nums2[0]);
    cout<<"The median of two sorted array is
"<<fixed<<setprecision(5)
    <<median(nums1,nums2,m,n);
    return 0;
}
```

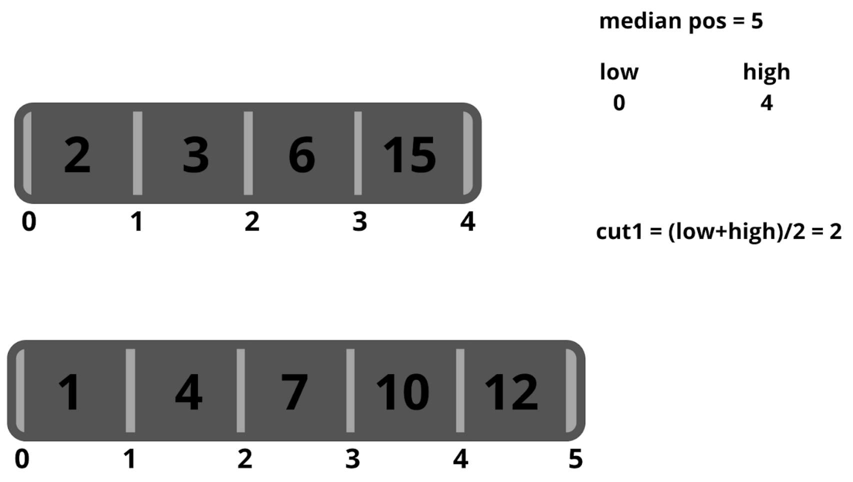
The Median of two sorted arrays is 6.00000
Time Complexity :
O(m+n)
Reason – We traverse through both the arrays linearly.
Space Complexity :
O(m+n)
Reason – We store the final array whose size is m+n.

Solution 2: Optimised Naive Approach
We can optimize in space complexity.
Approach :
Similar to the naive approach, instead of storing the final merged sorted array, we can keep a counter to keep track of the required position where the median will exist. First, using the median formula, get a position where the median will exist. Then, follow the above approach and instead of storing elements in another array, we will increase the counter value. When the counter value is equal to the median positions, return element.
Time Complexity : O(m+n)
Reason – We are still traversing both the arrays linearly.
Space Complexity: O(1)
Reason – We do not use any extra array.
Solution 3: Efficient solution
Intuition :
We came up with a naive solution because of the hint that two arrays are sorted and we want elements from merged sorted arrays. If we look into the word “sorted arrays”, we can think of a binary solution. Hence, we move to an efficient solution using binary search. But how to apply binary search? Let’s look into the thought process.
We know that we will get answers only from the final merged sorted arrays. We figured it out with the naive approach discussed above. We will partition both the arrays in such a way that the left half of the partition will contain elements, which will be there when we merge them, till the median element and rest in the other right half. This partitioning of both arrays will be done by binary search.

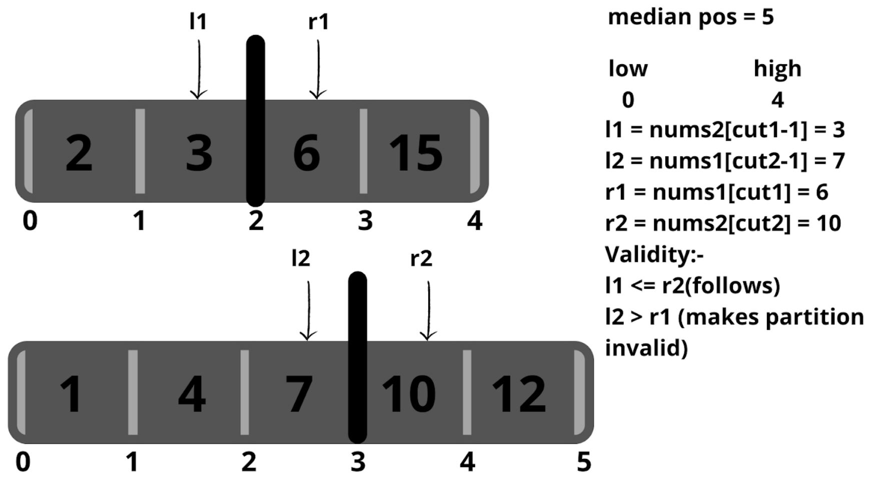
Approach :
We will do a binary search in one of the arrays which have a minimum size among the two. Firstly, calculate the median position with (n+1)/2. Point two-pointer, say low and high, equal to 0 and size of the array on which we are applying binary search respectively. Find the partition of the array. Check if the left half has a total number of elements equal to the median position. If not, get the remaining elements from the second array. Now, check if partitioning is valid. This is only when l1<=r2 and l2<=r1. If valid, return max(l1,l2)(when odd number of elements present) else return (max(l1,l2)+min(r1,r2))/2.
If partitioning is not valid, move ranges. When l1>r2, move left and perform the above operations again. When l2>r2, move right and perform the above operations.
Code :
```
#include<bits/stdc++.h>
using namespace std;
float median(int num 1[],int num2[],int m,int n) {
    if(m>n)
        return median(nums2,nums1,n,m);//ensuring that binary search
happens on minimum size array
    int low=0,high=m,medianPos=((m+n)+1)/2;
    while(low<=high) {
        int cut1 = (low+high)>>1;
        int cut2 = medianPos - cut1;
        int l1 = (cut1 == 0)? INT_MIN:nums1[cut1-1];
        int l2 = (cut2 == 0)? INT_MIN:nums2[cut2-1];
        int r1 = (cut1 == m)? INT_MAX:nums1[cut1];
        int r2 = (cut2 == n)? INT_MAX:nums2[cut2];
        if(l1<=r2 && l2<=r1) {
            if((m+n)%2 != 0)
                return max(l1,l2);
            else
                return (max(l1,l2)+min(r1,r2))/2.0;
        }
        else if(l1>r2) high = cut1-1;
        else low = cut1+1;
    }
    return 0.0;
}
int main() {
    int nums1[] = {1,4,7,10,12};
    int nums2[] = {2,3,6,15};
    int m = sizeof(nums1)/sizeof(nums1[0]);
    int n = sizeof(nums2)/sizeof(nums2[0]);
    cout<<"The Median of two sorted arrays
is"<<fixed<<setprecision(5)
    <<median(nums1,nums2,m,n);
    return 0;
}
```

Output:
The Median of two sorted arrays is 6.00000
Time Complexity : O(log(m,n))
Reason – We are applying binary search on the array which has a minimum size.
Space Complexity: O(1)
Reason – No extra array is used.

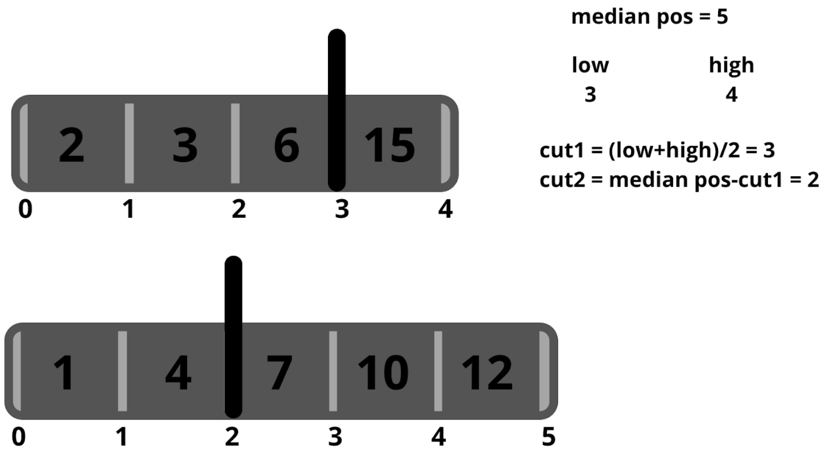
Dry Run :
Let’s look into the dry run. For example 1, the total size is equal to 9 which is odd. Applying the formula, the median will be at (5+1)/2 = 10/2 = 5th position of the final merged sorted array. The size of nums2 is less than the size of nums1. Thus, apply binary search in nums2.



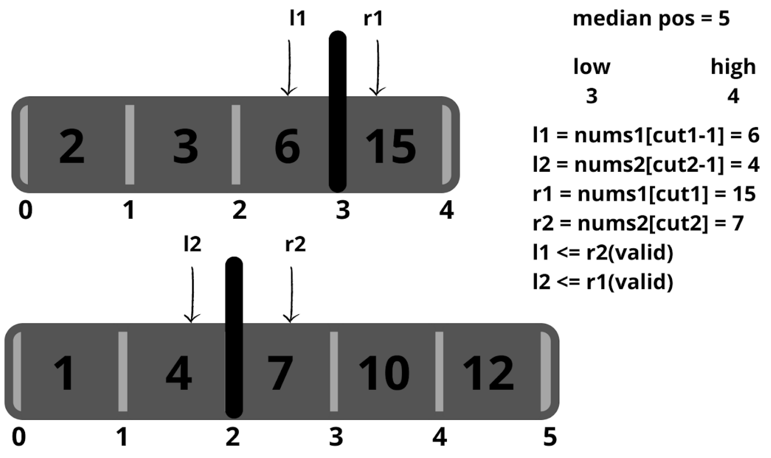
We will cut1 to be at pos 2. So, the total number of elements at the left of cut1 is 2. So, we will choose the remaining 3 elements from num2. We will make cut2 at 3.



Partitioning is invalid. l2>r1 and to make the left half valid, we have to decrease the value of l2. We have to move right by moving low to cut1+1.



We moved towards the right and got a new partitioning.



We can see it is a valid left half. Thus, we get our median as max(l1,l2).

K-th Element of two sorted arrays

Problem Statement: Given **two sorted arrays** of size **m** and **n** respectively, you are tasked with finding the element that would be at the **kth position** of the **final sorted array**.

Examples :

```
Input: m = 5
       n = 4
       array1 = [2,3,6,7,9]
       array2 = [1,4,8,10]
       k = 5

Output:
 6
```

```
Explanation: Merging both arrays and sorted. Final array will be -
 [1,2,3,4,6,7,8,9,10]
We can see at k = 5 in the final array has 6.


Input:
 m = 1
       n = 4
       array1 = [0]
       array2 = [1,4,8,10]
       k = 2

Output:
 4

Explanation:
 Merging both arrays and sorted. Final array will be -
 [1,4,8,10]
We can see at k = 2 in the final array has 4
```

Solution:

Disclaimer: *Don't jump directly to the solution, try it out yourself first.*

Solution 1: Naive Solution

Intuition:

It is given that both arrays are sorted. We need to kth element which will be present when both are merged in a sorted manner. It gives us hints of approaching a solution with merge sort. Why so? If we see an algorithm of merge sort. It includes the following steps.

1. Divide the array into two halves.
2. Merge them in a sorted way.

So, we can use the method of merging two sorted arrays.

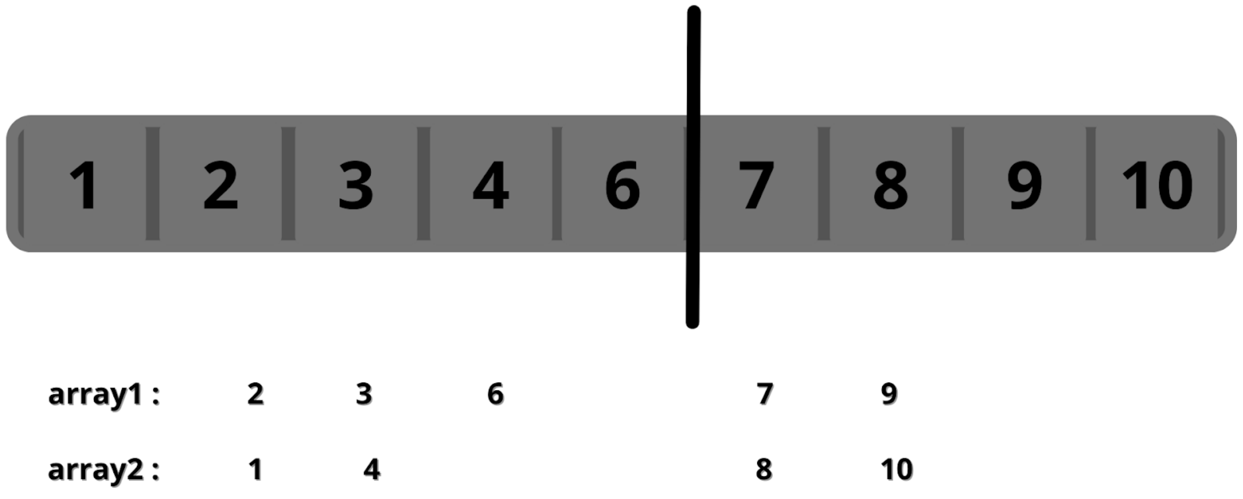
Approach :

We will keep two pointers, say p1 and p2, each in two arrays. A counter to keep track of whether we have reached the kth position. Start iterating through both arrays. If array1[p1] < array2[p2], move p1 pointer ahead and increase counter value. If array2[p2] <array1[p1], move p2 pointer ahead and increase counter. When the count is equal to k, return the element in which condition makes the counter value equal to k.

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

int kthelement(int array1[],int array2[],int m,int n,int k) {
    int p1=0,p2=0,counter=0,answer=0;

    while(p1<m && p2<n) {
        if(counter == k) break;
        else if(array1[p1]<array2[p2]) {
            answer = array1[p1];
            ++p1;
        }
        else {
            answer = array2[p2];
            ++p2;
        }
        ++counter;
    }
    if(counter != k) {
        if(p1 != m-1)
            answer = array1[k-counter];
        else
            answer = array2[k-counter];
    }
    return answer;
}

int main() {
    int array1[] = {2,3,6,7,9};
    int array2[] = {1,4,8,10};
    int m = sizeof(array1)/sizeof(array1[0]);
    int n = sizeof(array2)/sizeof(array2[0]);
    int k = 5;
    cout<<"The element at the kth position in the final sorted array
is "
    <<kthelement(array1,array2,m,n,k);
    return 0;
}
```

Output: The element at the kth position in the final sorted array is 6
Time Complexity :
We iterate at total k times. This makes time complexity to O(k)

Space Complexity :
We do not use any extra data structure and hence, the time complexity is O(1).
Approach 2: Optimal Solution
Intuition :
It is mentioned that given arrays are sorted. This gives us some hints to use binary search in them.
If we look into the final merged sorted array.



We can part it in such a way that our kth element will be at the end of the left half array. Let's make some observations. The left portion of the array is made of some elements of both array1 and array2. We can see that all elements of the right half of the array are always larger than the left ones. So, with help of binary search, we will divide arrays into partitions with keeping k elements in the left half. We have to keep in mind that l1 <= r2 and l2 <= r1. Why so? This ensures that left-half elements are always lesser than right elements.
Approach :
Apply binary search in an array with a small size. Start iterating with two pointers, say left and right. Find the middle of the range. Take elements from low to middle of array1 and the remaining elements from the second array. Then using the condition mentioned above, check if the left half is valid. If valid, print the maximum of both array's last element. If not, move the range towards the right if l2 > r1, else move the range towards the left if l1 > r2.

Code:
```
#include<bits/stdc++.h>
using namespace std;
int kthelement(int arr1[], int arr2[], int m, int n, int k) {
    if(m > n) {
        return kthelement(arr2, arr1, n, m, k);
    }

    int low = max(0,k-m), high = min(k,n);

    while(low <= high) {
        int cut1 = (low + high) >> 1;
        int cut2 = k - cut1;
        int l1 = cut1 == 0 ? INT_MIN : arr1[cut1 - 1];
        int l2 = cut2 == 0 ? INT_MIN : arr2[cut2 - 1];
        int r1 = cut1 == n ? INT_MAX : arr1[cut1];
        int r2 = cut2 == m ? INT_MAX : arr2[cut2];

        if(l1 <= r2 && l2 <= r1) {
            return max(l1, l2);
        }
        else if (l1 > r2) {
            high = cut1 - 1;
        }
        else {
            low = cut1 + 1;
        }
    }
    return 1;
}
int main() {
    int array1[] = {2,3,6,7,9};
    int array2[] = {1,4,8,10};
    int m = sizeof(array1)/sizeof(array1[0]);
    int n = sizeof(array2)/sizeof(array2[0]);
    int k = 5;
    cout<<"The element at the kth position in the final sorted array
is "
    <<kthelement(array1,array2,m,n,k);
    return 0;
}
```

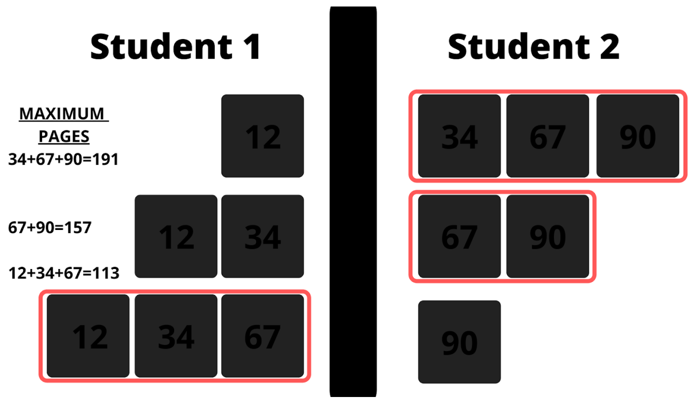
Output: The element at the kth position in the final sorted array is 6

Time Complexity : log(min(m,n))
Reason: We are applying binary search in the array with minimum size among the two. And we know the time complexity of the binary search is log(N) where N is the size of the array. Thus, the time complexity of this approach is log(min(m,n)), where m,n are the sizes of two arrays.
Space Complexity: O(1)
Reason: Since no extra data structure is used, making space complexity to O(1).

Allocate Minimum Number of Pages

Problem Statement: Given an array of integers A of size N and an integer B.
The College library has N bags, the ith book has A[i] number of pages.
You have to allocate books to B number of students so that the maximum number of pages allocated to a student is minimum.
Conditions given :
A book will be allocated to exactly one student.
Each student has to be allocated at least one book.
Allotment should be in contiguous order, for example, A student cannot be allocated book 1 and book 3, skipping book 2.
Calculate and return the **minimum possible number**. Return -1 if a valid assignment is not possible.
Examples:

```
Example 1:

Input: A = [12, 34, 67, 90]
       B = 2

Output: 113

Explaination: Let's see all possible cases of how books can be
allocated for each student.
```



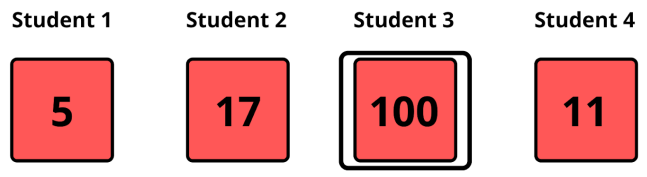
```
So, the maximum number of pages allocated in each case is
[191,157,113]. So, the minimum number among them is 113. Hence, our
result is 113.
```

Example 2:

Input: `A = [5, 17, 100, 11]`
`B = 4`

Output: `100`

Explaination:

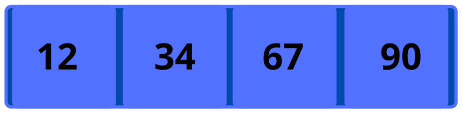


It is the only possible way to allocate books to each student.
Hence, the maximum page allocated is [100]. So, minimum of the maximum pages allocated is 100.

***Disclaimer**: Don't jump directly to the solution, try it out yourself first.*
Solution: Using Binary Search
Intuition :
Let's analyze a case.
We are given A = [12, 34, 67, 90] and B = 1. So, for one student we can allocate all books to this student. Why so? Because we have to maximize the number of pages allocated to a student and then find the minimum out of it. So, this fact gives us an idea that a single student will be allocated the sum of all pages available.
Let's analyze another case.
We are required to find the minimum number of pages among all possible maximum number of pages of allocations. So, in the worst case, the minimum possible will be minimum pages among all given books.
Now, we know the lowest possible answer and the maximum possible answer and for general cases, the answer will lie in between these edge cases. Thus, we have a search space. This search space will be sorted. Guess what? We reached our approach to use a binary solution.
Approach :
We will set a search space. The lower boundary will be of minimal value among all the books given. The upper boundary will be the sum of all book pages given. Then apply binary search. How to change the range of searching? While searching, allocate pages to each student in such a way that the sum of allocated pages of each student is not greater than the mid-value of

search space. If allocating students increases more than the number of students provided, this shows that mid-value should be more, and hence, we move right by restricting our lower boundary as mid+1. If an allocation is possible then reduce the search upper boundary by mid-1. Also, an edge case to check while allocating, each book page should not be greater than mid-value chosen as a barrier.

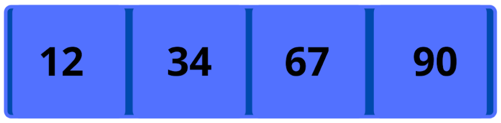
Dry Run:



Student = 2

low = 12(minimum among all given pages)
high = 12+34+67+90 = 203(sum of all pages)
so search space is
[12 . . . . 203]

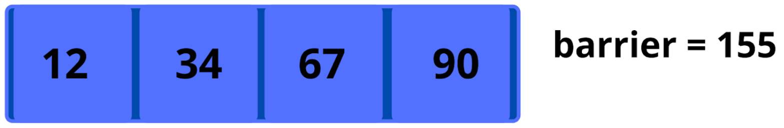
We will find mid of the search space and set it as a barrier. Here, mid = (12+203)/2 =107. Now, we will find if the allocation is possible among two students.



barrier = 107

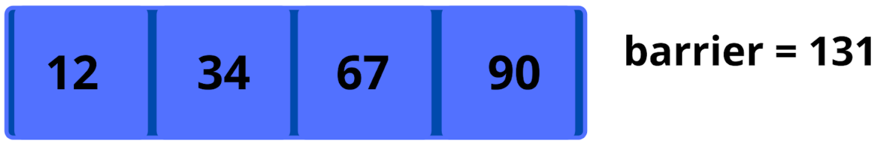
number of students possible for allocation
Student 1 : 12+34 = 46 < 107
Student 2 : 67 < 107
Student 3 : 90 < 107
Since, number of students is greater than given students, hence this allocation is wrong and we increase out barrier my moving low = mid+1

Now, low = 108, high = 203, so mid = (108+203)/2 = 155. Again check if allocation is possible.



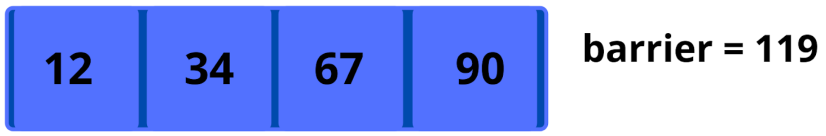
number of students possible for allocation
Student 1 : 12+34+67 = 113 < 155
Student 2 : 90 < 155
Here, allocation is possible. This means it is one of the possible answer. So, we reduce search space by high = mid-1

Now, low = 108, high = 154, so mid = (108+154)/2 = 131. Again check if allocation is possible.



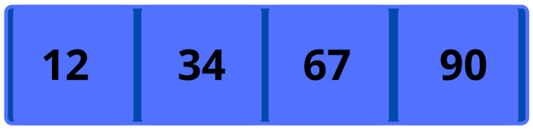
number of students possible for allocation
Student 1 : 12+34+67 = 113 < 131
Student 2 : 90 < 131
Here, allocation is possible. This means it is one of the possible answer. So, we reduce search space by high = mid-1

Now, low = 108, high = 130,so mid = (108+130)/2 = 119. Again check if allocations are possible.



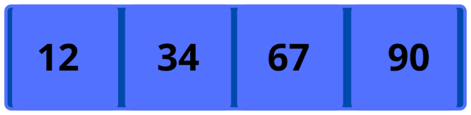
number of students possible for allocation
Student 1 : 12+34+67 = 113 < 119
Student 2 : 90 < 119
Here, allocation is possible. This means it is one of the possible answer. So, we reduce search space by high = mid-1

Now, low = 108, high = 118,so mid = (108+118)/2 = 113. Again check if allocations are possible.



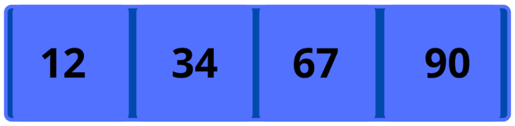
number of students possible for allocation
Student 1 : 12+34+67 = 113 = 113
Student 2 : 90 < 113
Here, allocation is possible. This means it is one of the possible answer. So, we reduce search space by high = mid-1

Now, low = 108, high = 112,so mid = (108+112)/2 = 110. Again check if allocations are possible.



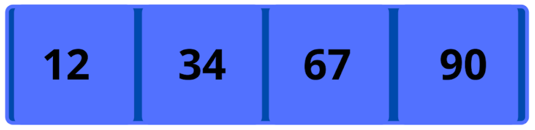
number of students possible for allocation
Student 1 : 12+34 = 46 < 110
Student 2 : 67 < 110
Student 3 : 90 < 110
Since, number of students is greater than given students, hence this allocation is wrong and we increase out barrier my moving low = mid+1

Now, low = 111, high = 112,so mid = (111+112)/2 = 111. Again check if allocations are possible.



number of students possible for allocation
Student 1 : 12+34 = 46 < 111
Student 2 : 67 < 111
Student 3 : 90 < 111
Since, number of students is greater than given students, hence this allocation is wrong and we increase out barrier my moving low = mid+1

Now, low = 112, high = 112,so mid = (112+112)/2 = 112. Again check if allocations are possible.



number of students possible for allocation
Student 1 : 12+34 = 46 < 112
Student 2 : 67 < 112
Student 3 : 90 < 112
Since, number of students is greater than given students, hence this allocation is wrong and we increase out barrier my moving low = mid+1

Now, low = 113, high = 112. Since, low > high, binary search ends and our result is equal to low = 113.

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

using namespace std;
//to check if allocation of books among given students is possible.
int isPossible(vector < int > & A, int pages, int students) {
  int cnt = 0;
  int sumAllocated = 0;
  for (int i = 0; i < A.size(); i++) {
    if (sumAllocated + A[i] > pages) {
      cnt++;
      sumAllocated = A[i];
      if (sumAllocated > pages) return false;
    } else {
      sumAllocated += A[i];
    }
  }
  if (cnt < students) return true;
  return false;
}
int books(vector < int > & A, int B) {
  if (B > A.size()) return -1;
  int low = A[0];
  int high = 0;
  //to find minimum value and sum of all pages
  for (int i = 0; i < A.size(); i++) {
    high = high + A[i];
    low = min(low, A[i]);
  }
  //binary search
  while (low <= high) {
    int mid = (low + high) >> 1;
    if (isPossible(A, mid, B)) {
      high = mid - 1;
    } else {
      low = mid + 1;
    }
  }
  return low;
}
int main() {
  vector<int> A = {12,34,67,90};
  int B = 2;
  cout << "Minimum Possible Number is " << books(A, B);
  return 0;
}
```

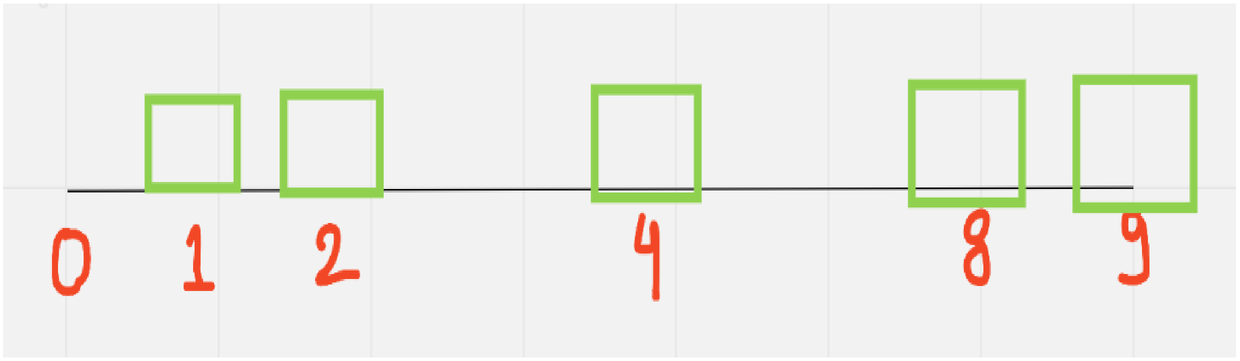
Output: Minimum Possible Number is 113
Time Complexity : O(NlogN)
Reason: Binary search takes O(log N). For every search, we are checking if an allocation is possible or not. Checking for allocation takes O(N).
Space Complexity: O(1)
Reason: No extra data structure is used to store spaces

Aggressive Cows : Detailed Solution

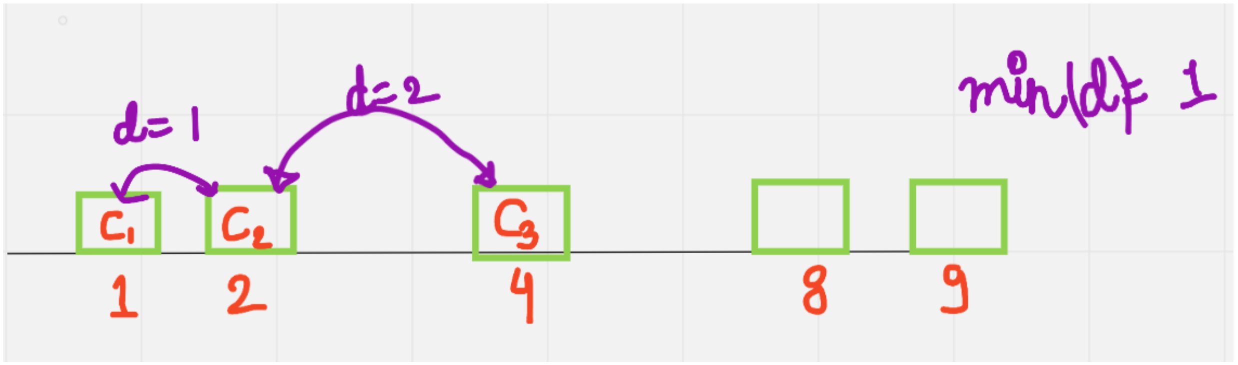
Problem Statement: There is a new barn with N stalls and C cows. The stalls are located on a straight line at positions x1,….,xN (0 <= xi <= 1,000,000,000). We want to assign the cows to the stalls, such that the minimum distance between any two of them is as large as possible. What is the largest minimum distance?
Examples:

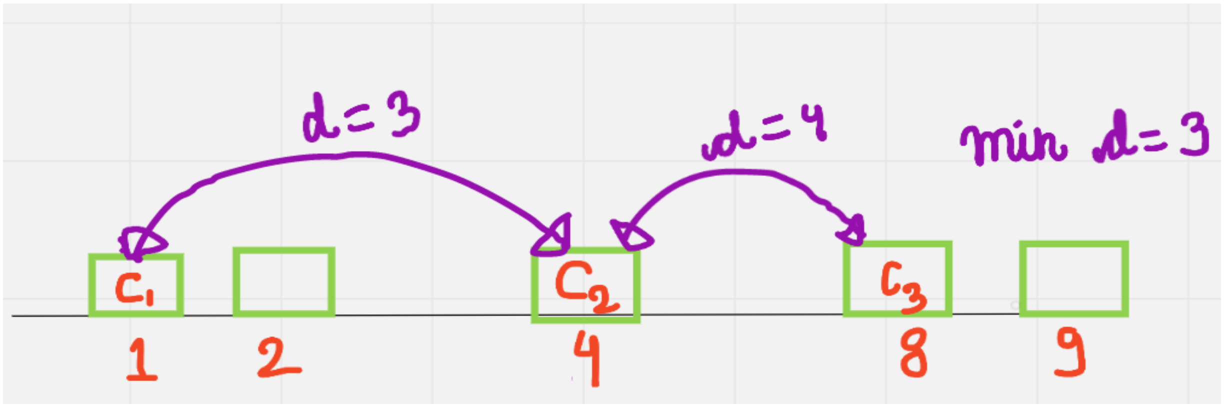
```
Input: No of stalls = 5
       Array: {1,2,8,4,9}
       And number of cows: 3

Output: One integer, the largest minimum distance 3
```

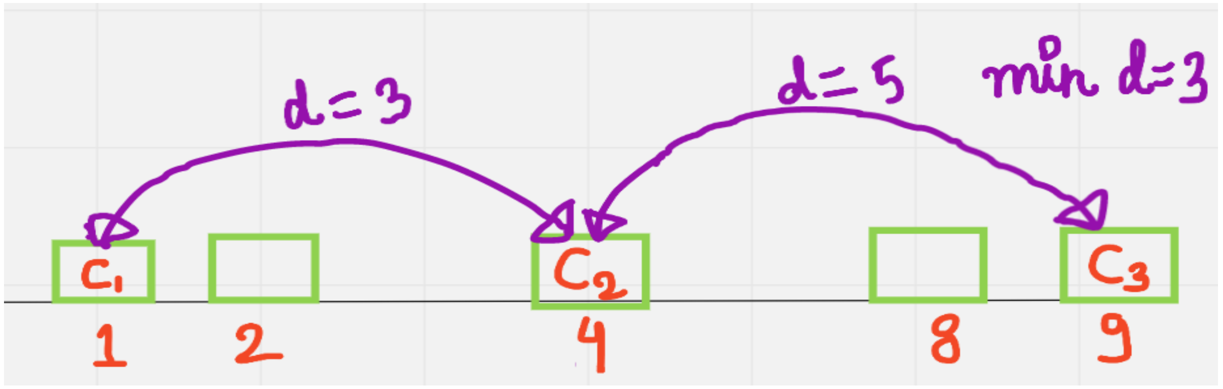
Explanation:



We have to fit in 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 case, the cows were arranged in the first three consecutive stalls, which is not advisable because they require maximum distance between them. So we make sure that there is some **minimum** distance between them so they do not fight. We have to maximize that difference so as to accommodate three cows. This is done in the second and third examples. It's not possible to get a minimum distance of more than 3 in any arrangement, so we output 3.

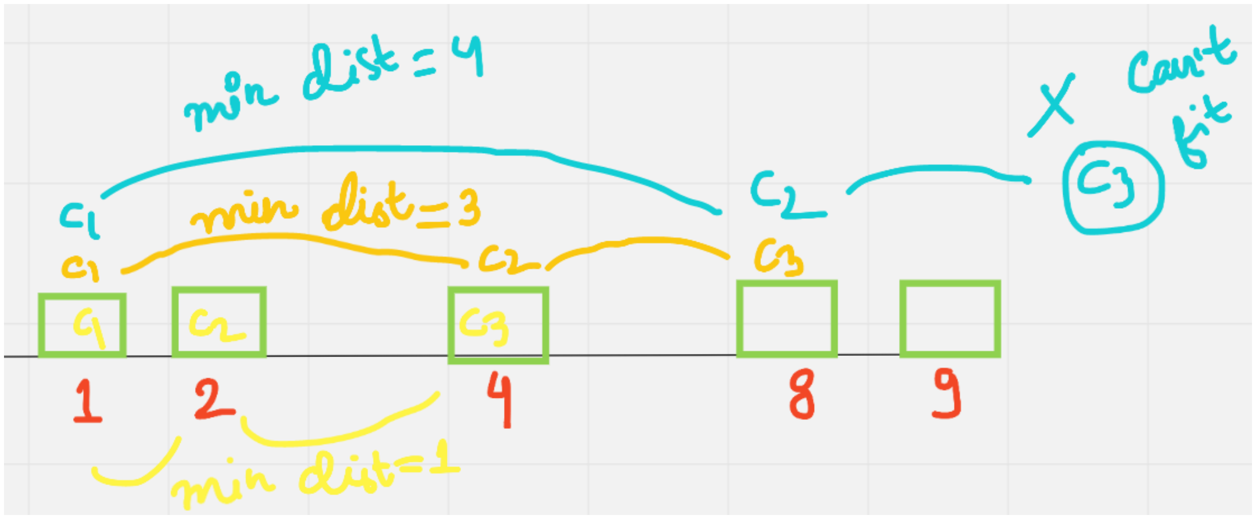
Solution

Disclaimer: *Don't jump directly to the solution, try it out yourself first.*

Solution 1: Brute Force

Intuition:

It's required that we put all the C cows into our stalls. So for a start, let's say we set the minimum distance = 1 and put them consecutively. These cows fit perfectly.
This is too close, so let's increase our minimum distance a bit. Let's increase the distance further to 2. We can again check that the cows can be accommodated.
But we want to reduce the possibility of collision/fighting as much as possible, so we keep on increasing the minimum distance. Here is an illustration:



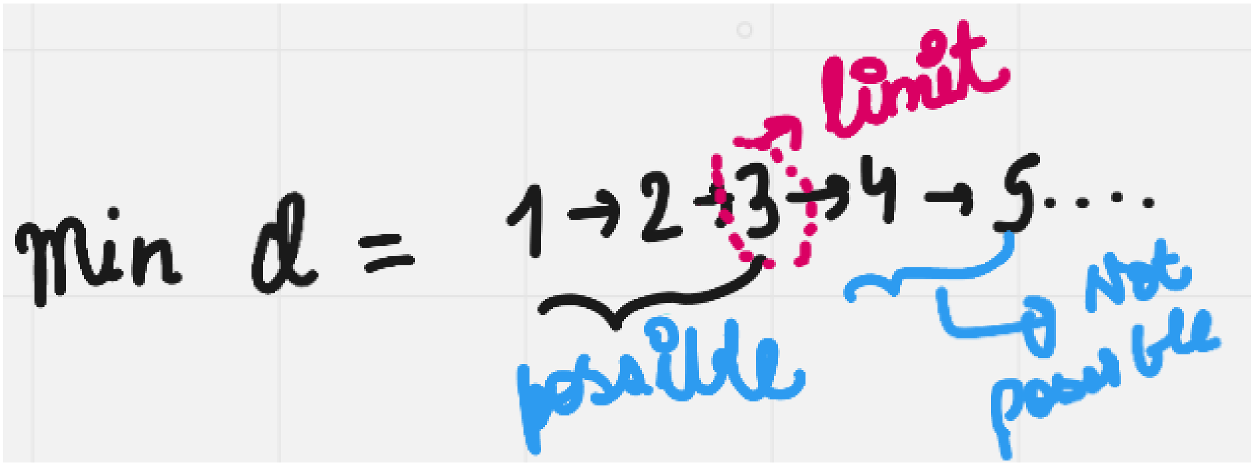
Approach:

After sorting the array, we set a minimum distance, then we keep on increasing until accommodation of all cows is not possible. We stop *just* before that to get our answer, which in this example is 3.
For checking if the cows can fit or not, we are simply iterating over our n stalls, and for every stall with the said minimum distance, we place our cow. After we are done, if all cows have been accommodated, we return true, otherwise false. Let's observe the time complexity of our brute force algorithm here, we are increasing distance in each step (which in the worst case of two cows gets as high as m = array[n-1]-array[0]), and in that step, we are checking if our cows can "fit", so we are iterating again for each step to check.

Code:

```
#include <bits/stdc++.h>
using namespace std;
bool isCompatible(vector < int > inp, int dist, int cows) {
  int n = inp.size();
  int k = inp[0];
  cows--;
  for (int i = 1; i < n; i++) {
    if (inp[i] - k >= dist) {
      cows--;
      if (!cows) {
        return true;
      }
      k = inp[i];
    }
  }
  return false;
}
int main() {
  int n = 5, m = 3;
  vector<int> inp {1,2,8,4,9};
  sort(inp.begin(), inp.end());
  int maxD = inp[n - 1] - inp[0];
  int ans = INT_MIN;
  for (int d = 1; d <= maxD; d++) {
    bool possible = isCompatible(inp, d, m);
    if (possible) {
      ans = max(ans, d);
    }
  }
  cout << "The largest minimum distance is " << ans << '\n';
}
```

Output: The largest minimum distance is 3
Time complexity: O(n* m)
Space Complexity: O(1)
Solution 2: Binary Search
Intuition:
Observing what we did in the previous case, this time complexity can be improved.
There is a certain maximum “limit” as to what we can increase our minimum distance because we don’t want any cow to be left behind.



In this example, all values including and after 4 are invalid. There is certain “monotonicity” here, speaking in layman’s terms, we are trying to find d for which the minimum distance is maximized, and there is a certain value, here 4 including and after which the solution to this problem is not possible. In these situations, we use the binary search algorithm.

Approach:

(Note, before proceeding, make sure you are familiar with binary search.)

Make sure to sort the array first, because only then it makes sense to use binary search.

For the BS approach, we set low = 1 because the minimum distance is 1 and high =array[n-1] – array[0] . because that’s the maximum possible distance between two stalls.

Let’s calculate our mid-value after this.

mid = low + (high-low)/2

Then we check if the minimum distance(mid-value) is possible by the same method defined in brute force, and if it is not possible, this means we can set our upper bound as high-1, and if it is possible, we store it in an answer variable and set our lower bound as mid+1. We keep on doing this until high and low pointers are equal.

Dry run of this example:

low = 1, high = 9-1 = 8

mid = 1 + (8-1)/2 = 4

Let’s check for 4, but it doesn’t fit(check Fig 4), so now we can reduce our search space by setting the upper bound as 4-1=3 because all numbers greater than equal to 4 are not valid. Now we check for mid = 1 + (3-1)/2 = 2Which is 2, checking if it’s a valid solution and cows can fit in, we find that it is a valid solution, so we store it as a possible answer and because we need a maximum-minimum distance, we set the lower bound as 2+1 =3. We find that 3 is also a possible solution, so we store it in our ans because it’s greater than our previous answer. Our low and high variables are equal now, so we can stop our binary search here.

Code:

```
#include <bits/stdc++.h>
 using namespace std;
 bool isPossible(int a[], int n, int cows, int minDist) {
   int cntCows = 1;
   int lastPlacedCow = a[0];
   for (int i = 1; i < n; i++) {
     if (a[i] - lastPlacedCow >= minDist) {
       cntCows++;
       lastPlacedCow = a[i];
     }
   }
   if (cntCows >= cows) return true;
   return false;
 }
 int main() {
   int n = 5, cows = 3,a[]={1,2,8,4,9};
   sort(a, a + n);
   int low = 1, high = a[n - 1] - a[0];
```

```
      while (low <= high) {
        int mid = (low + high) >> 1;a
        if (isPossible(a, n, cows, mid)) {
          low = mid + 1;
        } else {
          high = mid - 1;
        }      }
      cout << "The largest minimum distance is " << high << endl;
      return 0;    }
```

Output: The largest minimum distance is 3
Time Complexity: O(N*log(M)).
Reason: For binary search in a space of M, O(log(M)) and for each search, we iterate over max N stalls to check. O(N).
Space Complexity: O(1)