*A real-life example of Binary Search:*

Problem statement: Assume there is a dictionary and we have to find the word “raj”.

Method 1: One of the many ways is to check every possible page of the entire dictionary and see if we can find the word “raj”. This technique is known as linear search.

Basically, we can traverse from the first till the end to find the target value in the search space i.e. the entire dictionary in our example

Method 2: In this case, we will optimize our search by using the property of a dictionary i.e. a dictionary is always in the sorted order.

We will first try to open the dictionary in such a way that it is roughly divided into two parts. Then, we will check the left page. Now, assume the words on the left page starts with ‘s’. We can certainly say that our target word i.e. “raj” definitely comes before the words start with ‘s’. So, now, we need not search in the entire dictionary rather we will only search in the left half.

Now, we will do the same thing with the left half. First, we will divide it into 2 halves and then try to locate which half contains the word “raj”. Eventually, after certain steps, we will end up finding the word “raj”.

This is a typical real-life example of binary search.

Note:

Binary search is only applicable in a sorted search space.

The sorted search space does not necessarily have to be a sorted array. It can be anything but the search space must be sorted.

In binary search, we generally divide the search space into two equal halves and then try to locate which half contains the target. According to that, we shrink the search space size.

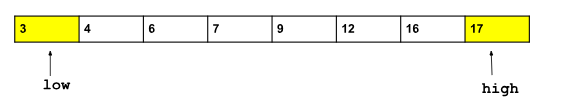
*BS ON 1D ARRAYS*

1. **Binary Search to find X in sorted array.**

**Iterative Implementation :**

Algorithm / Intuition

We will use a couple of pointers i.e. low and high to apply binary search. Initially, the low pointer should point to the first index and the high pointer should point to the last index.



Search space: The entire area between the low and the high pointer(including them) is considered the search space. Here, the search space is sorted.

Algorithm:

Now, we will apply the binary search algorithm in the given array:

Step 1: Divide the search space into 2 halves:

In order to divide the search space, we need to find the middle point of it. So, we will take a ‘mid’ pointer and do the following:

mid = (low+high) // 2

Step 2: Compare the middle element with the target:

In this step, we can observe 3 different cases:

If arr[mid] == target: We have found the target. From this step, we can return the index of the target possibly.

If target > arr[mid]: This case signifies our target is located on the right half of the array. So, the next search space will be the right half.

If target < arr[mid]: This case signifies our target is located on the left half of the array. So, the next search space will be the left half.

Step 3: Trim down the search space:

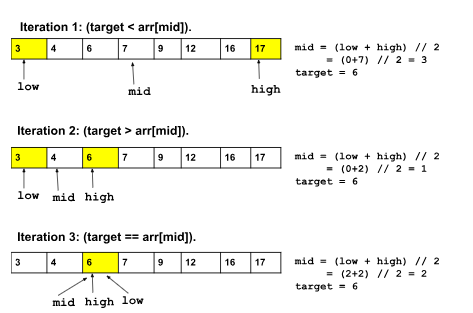
Based on the probable location of the target we will trim down the search space.

If the target occurs on the left, we should set the high pointer to mid-1. Thus the left half will be the next search space.

Similarly, if the target occurs on the right, we should set the low pointer to mid+1. Thus the right half will be the next search space.

The above steps will continue until either we found the target or the search space becomes invalid i.e. high < low. By definition of search space, it will lose its existence if the high pointer is appearing before the low pointer.

**Dry-run:**



Note: If the target is not present in the array, low and high will cross each other.

public static int search(int[] arr,int target)  
{  
 int start = 0;  
 int end = arr.length-1;  
 int mid = 0;  
  
 while(start<=end)  
 {  
 mid = (start+end)/2;  
 if(target == arr[mid])  
 {  
 return mid;  
 }  
 else if(arr[mid]<target)  
 {  
 start = mid+1;  
 }  
 else  
 {  
 end = mid-1;  
 }  
 }  
  
 return -1;  
}

**TIME COMPLEXITY:** O(logN)

Let’s derive the number of divisions mathematically,

If a number n can be divided by 2 for x times:

2^x = n

Therefore, x = logn (base is 2)

So the overall time complexity is O(logN), where N = size of the given array.

**SPACE COMPLEXITY**: O(1)

**Recursive Implementation :**

Algorithm / Intuition

Recursive implementation:

Pre-requisite: Recursion section

Approach:

Assume, the recursive function will look like this: binarySearch(nums, low, high). It basically takes 3 parameters i.e. the array, the low pointer, and the high pointer. In each recursive call, we will change the value of low and high pointers to trim down the search space. Except for this, the rest of the steps will be the same.

The steps are as follows:

Step 1: Divide the search space into 2 halves:

In order to divide the search space, we need to find the middle point of it. So, we will take a ‘mid’ pointer and do the following:

mid = (low+high) // 2 ( ‘//’ refers to integer division)

Step 2: Compare the middle element with the target and trim down the search space:

In this step, we can observe 3 different cases:

If arr[mid] == target: We have found the target. From this step, we can return the index of the target, and the recursion will end.

If target > arr[mid]: This case signifies our target is located on the right half of the array. So, the next recursion call will be binarySearch(nums, mid+1, high).

If target < arr[mid]: This case signifies our target is located on the left half of the array. So, the next recursion call will be binarySearch(nums, low, mid-1).

Base case: The base case of the recursion will be low > high. If (low > high), the search space becomes invalid which means the target is not present in the array.

public static int search2(int[] arr,int target,int high,int low)  
{  
 if(low>high)  
 return -1;  
 int mid = (low+high)/2;  
 if(arr[mid] == target)  
 return mid;  
 else if(arr[mid]<target)  
 return *search2*(arr,target,high,mid+1);  
 return *search2*(arr,target,mid-1,low);  
}

**TIME COMPLEXITY:** O(logN)

Let’s derive the number of divisions mathematically,

If a number n can be divided by 2 for x times:

2^x = n

Therefore, x = logn (base is 2)

So the overall time complexity is O(logN), where N = size of the given array.

**SPACE COMPLEXITY**: O(logN)

Each recursive call adds a new frame to the call stack, and since binary search makes at most O(log n) recursive calls (because it halves the search space at each step), the space complexity is O(log n) due to the recursion stack.

**2.** **Implement Lower Bound.**

The lower bound algorithm finds the first or the smallest index in a sorted array where the value at that index is greater than or equal to a given key i.e. x.2

The lower bound is the smallest index, ind, where arr[ind] >= x. But if any such index is not found, the lower bound algorithm returns n i.e. size of the given array.

**Example 1:**

Input Format:

N = 4, arr[] = {1,2,2,3}, x = 2

Result:

1

Explanation:

Index 1 is the smallest index such that arr[1] >= x.

**Example 2:**

Input Format:

N = 5, arr[] = {3,5,8,15,19}, x = 9

Result:

3

Explanation:

Index 3 is the smallest index such that arr[3] >= x.

**Brute – Force Approach:**

Naive approach (Using linear search):

Let’s understand how we can find the answer using the linear search algorithm. With the knowledge that the array is sorted, our approach involves traversing the array starting from the beginning. During this traversal, each element will be compared with the target value, x. The index, i, where the condition arr[i] >= x is first satisfied, will be the answer.

public static int lowerBound(int []arr, int n, int x) {  
 for (int i = 0; i < n; i++) {  
 if (arr[i] >= x) {  
 // lower bound found:  
 return i;  
 }  
 }  
 return n;  
}

**TIME COMPLEXITY**: O(N), where N = size of the given array.

Reason: In the worst case, we have to travel the whole array. This is basically the time complexity of the linear search algorithm.

**SPACE COMPLEXITY**: O(1) as we are using no extra space.

**Optimal Approach:**

As the array is sorted, we will apply the Binary Search algorithm to find the index. The steps are as follows:

We will declare the 2 pointers and an ‘ans’ variable initialized to n i.e. the size of the array(as If we don’t find any index, we will return n).

Place the 2 pointers i.e. low and high: Initially, we will place the pointers like this: low will point to the first index, and high will point to the last index.

Calculate the ‘mid’: Now, we will calculate the value of mid using the following formula:

mid = (low+high) / 2 .

Compare arr[mid] with x: With comparing arr[mid] to x, we can observe 2 different cases:

Case 1 - If arr[mid] >= x: This condition means that the index mid may be an answer. So, we will update the ‘ans’ variable with mid and search in the left half if there is any smaller index that satisfies the same condition. Here, we are eliminating the right half.

Case 2 - If arr[mid] < x: In this case, mid cannot be our answer and we need to find some bigger element. So, we will eliminate the left half and search in the right half for the answer.

public static int LowerBound(int[] arr, int n, int x)  
{  
 int ans = n;  
 int low = 0;  
 int high = arr.length-1;  
 while(low<=high)  
 {  
 int mid = (low+high)/2;  
 if(arr[mid] >= x)  
 {  
 ans = mid;  
 high = mid-1;  
 }  
 else {  
 low = mid+1;  
 }  
 }  
 return ans;  
}

**TIME COMPLEXITY**: O(logN), where N = size of the given array.

Reason: We are basically using the Binary Search algorithm.

**SPACE COMPLEXITY**: O(1) as we are using no extra space.

**3.** **Implement Upper Bound.**

The upper bound algorithm finds the first or the smallest index in a sorted array where the value at that index is greater than the given key i.e. x.

The upper bound is the smallest index, ind, where arr[ind] > x.

But if any such index is not found, the upper bound algorithm returns n i.e. size of the given array. The main difference between the lower and upper bound is in the condition. For the lower bound the condition was arr[ind] >= x and here, in the case of the upper bound, it is arr[ind] > x.

**Input Format:**

Example 1:

Input Format:

N = 4, arr[] = {1,2,2,3}, x = 2

Result:

3

Explanation:

Index 3 is the smallest index such that arr[3] > x.

Example 2:

Input Format:

N = 6, arr[] = {3,5,8,9,15,19}, x = 9

Result:

4

Explanation:

Index 4 is the smallest index such that arr[4] > x

**Brute – Force Approach:**

Naive approach (Using linear search):

Let’s understand how we can find the answer using the linear search algorithm. With the knowledge that the array is sorted, our approach involves traversing the array starting from the beginning. During this traversal, each element will be compared with the target value, x. The index, i, where the condition arr[i] > x is first satisfied, will be the answer.

public static int UpperBound(int []arr, int n, int x) {

for (int i = 0; i < n; i++) {  
 if (arr[i] > x) {  
 // lower bound found:  
 return i;  
 }  
 }  
 return n;  
}

**TIME COMPLEXITY**: O(N), where N = size of the given array.

Reason: In the worst case, we have to travel the whole array. This is basically the time complexity of the linear search algorithm.

**SPACE COMPLEXITY**: O(1) as we are using no extra space.

**Optimal Approach:**

As the array is sorted, we will apply the Binary Search algorithm to find the index. The steps are as follows:

We will declare the 2 pointers and an ‘ans’ variable initialized to n i.e. the size of the array(as If we don’t find any index, we will return n).

Place the 2 pointers i.e. low and high: Initially, we will place the pointers like this: low will point to the first index and high will point to the last index.

Calculate the ‘mid’: Now, we will calculate the value of mid using the following formula:

mid = (low+high) // 2 ( ‘//’ refers to integer division)

Compare arr[mid] with x: With comparing arr[mid] to x, we can observe 2 different cases:

Case 1 - If arr[mid] > x: This condition means that the index mid may be an answer. So, we will update the ‘ans’ variable with mid and search in the left half if there is any smaller index that satisfies the same condition. Here, we are eliminating the right half.

Case 2 - If arr[mid] <= x: In this case, mid cannot be our answer and we need to find some bigger element. So, we will eliminate the left half and search in the right half for the answer.

The above process will continue until the pointer low crosses high.

public static int UpperBound(int[] arr, int n, int x)  
{  
 int ans = n;  
 int low = 0;  
 int high = arr.length-1;  
 while(low<=high)  
 {  
 int mid = (low+high)/2;  
 if(arr[mid] > x)  
 {  
 ans = mid;  
 high = mid-1;  
 }  
 else {  
 low = mid+1;  
 }  
 }  
 return ans;  
}

**TIME COMPLEXITY**: O(logN), where N = size of the given array.

Reason: We are basically using the Binary Search algorithm.

**SPACE COMPLEXITY**: O(1) as we are using no extra space.

**4.** **Search Insert Position.**

*Problem Statement*:

You are given a sorted array arr of distinct values and a target value x. You need to search for the index of the target value in the array.

If the value is present in the array, then return its index. Otherwise, determine the index where it would be inserted in the array while maintaining the sorted order.

Testcases:

**Example 1:**

Input Format: arr[] = {1,2,4,7}, x = 6

Result: 3

Explanation: 6 is not present in the array. So, if we will insert 6 in the 3rd index(0-based indexing), the array will still be sorted. {1,2,4,6,7}.

**Example 2:**

Input Format: arr[] = {1,2,4,7}, x = 2

Result: 1

Explanation: 2 is present in the array and so we will return its index i.e. 1.

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.

On deep introspection of the given problem, we can easily understand that we have to find the correct position or the existing position of the target number in the given array.

Now, if the element is not present, we have to find the nearest greater number of the target number. So, basically, we are trying to find an element arr[ind] >= x and hence the lower bound of the target number i.e. x.

The lower bound algorithm returns the first occurrence of the target number if the number is present and otherwise, it returns the nearest greater element of the target number.

public static int LowerBound(int[] arr, int n, int x)  
{  
 int ans = n;  
 int low = 0;  
 int high = arr.length-1;  
 while(low<=high)  
 {  
 int mid = (low+high)/2;  
 if(arr[mid] >= x)  
 {  
 ans = mid;  
 high = mid-1;  
 }  
 else {  
 low = mid+1;  
 }  
 }  
 return ans;  
}

***Time Complexity:***O(logN), where N = size of the given array.

Reason: We are basically using the Binary Search algorithm.

***Space Complexity:*** O(1) as we are using no extra space.

**5.** **Floor/Ceil of an array.**

***Problem Statement***:

You're given an sorted array arr of n integers and an integer x. Find the floor and ceiling of x in arr[0..n-1].

The **floor** of x is the largest element in the array which is smaller than or equal to x.

The **ceiling** of x is the smallest element in the array greater than or equal to x.

Testcases:

Example 1:

Input Format: n = 6, arr[] ={3, 4, 4, 7, 8, 10}, x= 5

Result: 4 7

Explanation: The floor of 5 in the array is 4, and the ceiling of 5 in the array is 7.

Example 2:

Input Format: n = 6, arr[] ={3, 4, 4, 7, 8, 10}, x= 8

Result: 8 8

Explanation: The floor of 8 in the array is 8, and the ceiling of 8 in the array is also 8.

The floor of x is the largest element in the array which is smaller than or equal to x( i.e. largest element in the array <= x).

The ceiling of x is the smallest element in the array greater than or equal to x( i.e. smallest element in the array >= x).

From the definitions, we can easily understand that the ceiling of x is basically the lower bound of x. The lower bound algorithm returns the index of x if x is present in the array and otherwise, it returns the index of the smallest element in the array greater than x.

The implementation of Ceil will be the same as the lower bound algorithm.

But we have no such algorithm prepared for the floor. So, we will implement the floor algorithm based on the Binary Search algorithm. The only difference in the algorithm compared to the lower bound algorithm will be the conditions. In this case,

If arr[mid] <= x: arr[mid] is a possible answer. So, we will store it and will try to find a larger number that satisfies the same condition. That is why we will remove the left half and try to find the number in the right half.

If arr[mid] > x: The arr[mid] is definitely not the answer and we need a smaller number. So, we will reduce the search space to the left half by removing the right half.

The rest of the part of the algorithm will be exactly the same.

public static int getFloor(int[] a,int n,int x)  
{  
 int low = 0;  
 int high = a.length-1;  
 int ans = -1;  
 while(low<=high)  
 {  
 int mid = (low+high)/2;  
 if(a[mid] <= x)  
 {  
 ans = a[mid];  
 low = mid+1;  
 }  
 else  
 {  
 high = mid-1;  
 }  
 }  
 return ans;  
}

public static int getCeil(int[] a,int n,int x)  
{  
 int low = 0;  
 int high = a.length-1;  
 int ans = -1;  
 while(low<=high)  
 {  
 int mid = (low+high)/2;  
 if(a[mid] >= x)  
 {  
 ans =a[mid];  
 high = mid-1;  
 }  
 else  
 {  
 low = mid+1;  
 }  
 }  
  
 return ans;  
}

***Time Complexity:*** O(logN), where N = size of the given array.

Reason: We are basically using the Binary Search algorithm.

***Space Complexity***: O(1) as we are using no extra space.

**6.** **Find the first or last occurrence of a given number in a sorted array.**

Given an array of integers nums sorted in non-decreasing order, find the starting and ending position of a given target value.

If target is not found in the array, return [-1, -1].

**Example 1:**

Input: nums = [5,7,7,8,8,10], target = 8

Output: [3,4]

**Example 2:**

Input: nums = [5,7,7,8,8,10], target = 6

Output: [-1,-1]

**Example 3:**

Input: nums = [], target = 0

Output: [-1,-1]

**Brute – Force Approach:**

As the array is already sorted, start traversing the array from the front and from the back using a for loop and check whether the element is present or not.

If the target element is present, break out of the loop and print the resulting index.

If the target element is not present inside the array, then print -1

public static int first(int n, int key, int[] v) {  
 int res = -1;  
 for (int i = 0; i < n ; i++) {  
 if (v[i] == key) {  
 res = i;  
 break;  
 }  
 }  
 return res;  
}

public static int last(int n, int key, int[] v) {  
 int res = -1;  
 for (int i = n - 1; i >= 0; i--) {  
 if (v[i] == key) {  
 res = i;  
 break;  
 }  
 }  
 return res;  
}

***Time Complexity***: O(n)

***Space Complexity***: O(1)

**Optimal Approach:**

**Using Binary Search**

Whenever the word “sorted” or other similar terminologies are used in an array question, BINARY SEARCH can be one of the approaches.

Initially consider the start=0 and the end=n-1 pointers and the result as -1.

Till start does not crossover end pointer compare the mid element

If the mid element is equal to the key value, store the mid-value in the result and move the start pointer to mid+1(move back for last) and mid-1(move front for first)

Else if the key value is less than the mid element then end= mid-1(move leftward)

Else do start = mid+1 (move rightwards)

public static int[] searchRange(int[] nums, int target)  
{  
 int first = *first*(nums,target);   
 int last = *last*(nums,target);  
 return new int[]{first,last};  
}  
  
public static int first(int[] nums,int target)  
{  
 int low = 0;  
 int high = nums.length-1;  
 int ans = -1;  
 while(low<=high)  
 {  
 int mid = (low+high)/2;  
 if(nums[mid] == target)  
 {  
 ans = mid;  
 high = mid-1;  
 }  
 else if(nums[mid] < target)  
 {  
 low = mid+1;  
 }  
 else  
 {  
 high = mid-1;  
 }  
 }  
 return ans;  
}  
  
public static int last(int[] nums,int target)  
{  
 int low = 0;  
 int high = nums.length-1;  
 int ans = -1;  
 while(low<=high)  
 {  
 int mid = (low+high)/2;  
 if(nums[mid] == target)  
 {  
 ans = mid;  
 low=mid+1;  
 }  
 else if(nums[mid] < target)  
 {  
 low = mid+1;  
 }  
 else  
 {  
 high = mid-1;  
 }  
 }  
 return ans;  
}

**Time Complexity:** O(2\*log n)6

**Space Complexity:** O(1)

**7.** **Count occurrences of a number in a sorted array with duplicates**

**Example 1:**

Input:

N = 7, X = 3 , array[] = {2, 2 , 3 , 3 , 3 , 3 , 4}

Output

: 4

Explanation:

3 is occurring 4 times in the given array so it is our answer.

**Example 2:**

Input:

N = 8, X = 2 , array[] = {1, 1, 2, 2, 2, 2, 2, 3}

Output

: 5

Explanation:

2 is occurring 5 times in the given array so it is our answer.

**Brute – Force Approach:**

Linear Search

public static int count(int[] arr, int n, int x) {  
 int cnt = 0;  
 for (int i = 0; i < n; i++) {  
  
 // counting the occurrences:  
 if (arr[i] == x) cnt++;  
 }  
 return cnt;  
}

**Time Complexity**: O(N), N = size of the given array

Reason: We are traversing the whole array.

**Space Complexity**: O(1) as we are not using any extra space.

**Optimal Approach: Binary Search**

Total number of occurrences = last occurrence - first occurrence + 1

public static int count(int arr[], int n, int x)  
{  
 int first = *first*(arr,x);  
 int last = *last*(arr,x);  
 if(first == 0 && last == 0)return 0;  
 return last-first+1;  
}  
  
public static int first(int[] arr,int x)  
{  
 int low = 0;  
 int high = arr.length-1;  
 int ans = 0;  
 while(low<=high)  
 {  
 int mid = (low+high)/2;  
 if(arr[mid] == x)  
 {  
 ans = mid;  
 high = mid-1;  
 }  
 else if(arr[mid]<x)  
 {  
 low = mid+1;  
 }  
 else  
 {  
 high = mid - 1;  
 }  
 }  
 return ans;  
}  
public static int last(int[] arr,int x)  
{  
 int low = 0;  
 int high = arr.length-1;  
 int ans = 0;  
 while(low<=high)  
 {  
 int mid = (low+high)/2;  
 if(arr[mid] == x)  
 {  
 ans = mid;  
 low=mid+1;  
 }  
 else if(arr[mid]<x)  
 {  
 low = mid+1;  
 }  
 else  
 {  
 high = mid - 1;  
 }  
 }  
 return ans;  
}

**Time Complexity:** O(2\*logN), where N = size of the given array.

Reason: We are basically using the binary search algorithm twice.

**Space Complexity**: O(1) as we are using no extra space.

**8.** **Single element in a Sorted array**

**Example 1:**

Input Format:

arr[] = {1,1,2,2,3,3,4,5,5,6,6}

Result:

4

Explanation:

Only the number 4 appears once in the array.

**Example 2:**

Input Format:

arr[] = {1,1,3,5,5}

Result:

3

Explanation:

Only the number 3 appears once in the array.

Your solution must run in O(log n) time and O(1) space.

**Brute – Force Approach 1:**

public static int singleNonDuplicate(ArrayList<Integer> arr) {  
 int n = arr.size();  
 if (n == 1)  
 return arr.get(0);  
  
 for (int i = 0; i < n; i++) {  
 if (i == 0) {  
 if (!arr.get(i).equals(arr.get(i + 1)))  
 return arr.get(i);  
 }  
 else if (i == n - 1) {  
 if (!arr.get(i).equals(arr.get(i - 1)))  
 return arr.get(i);  
 } else {  
 if (!arr.get(i).equals(arr.get(i - 1)) && !arr.get(i).equals(arr.get(i + 1)))  
 return arr.get(i);  
 }  
 }  
 return -1;  
}

**Time Complexity**: O(N), N = size of the given array.

**Space Complexity**: O(1) as we are not using any extra space.

**Brute – Force Approach 2:**

Using XOR

public static int singleNonDuplicate(ArrayList<Integer> arr) {  
 int n = arr.size(); //size of the array.  
 int ans = 0;  
 // XOR all the elements:  
 for (Integer integer : arr) {  
 ans = ans ^ integer;  
 }  
 return ans;  
}

**Time Complexity**: O(N), N = size of the given array.

**Space Complexity**: O(1) as we are not using any extra space.

**Optimal Approach(Using Binary Search):**

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.

We need to consider 2 different cases while using Binary Search in this problem. Binary Search works by reducing the search space by half. So, at first, we need to identify the halves and then eliminate them accordingly. In addition to that, we need to check if the current element i.e. arr[mid] is the ‘single element’.

If we can resolve these two cases, we can easily apply Binary Search in this algorithm.

How to check if arr[mid] i.e. the current element is the single element:

A crucial observation to note is that if an element appears twice in a sequence, either the preceding or the subsequent element will also be the same. But only for the single element, this condition will not be satisfied. So, to check this, the condition will be the following:

If arr[mid] != arr[mid-1] and arr[mid] != arr[mid+1]: If this condition is true for arr[mid], we can conclude arr[mid] is the single element.

The above condition will throw errors in the following 3 cases:

If the array size is 1.

If ‘mid’ points to 0 i.e. the first index.

If ‘mid’ points to n-1 i.e. the last index.

Note: At the start of the algorithm, we address the above edge cases without requiring separate conditions during the check for arr[mid] inside the loop. And the search space will be from index 1 to n-2 as indices 0 and n-1 have already been checked.

Resolving edge cases:

If n == 1: This means the array size is 1. If the array contains only one element, we will return that element only.

If arr[0] != arr[1]: This means the very first element of the array is the single element. So, we will return arr[0].

If arr[n-1] != arr[n-2]: This means the last element of the array is the single element. So, we will return arr[n-1].

How to identify the halves:

By observing the above image, we can clearly notice a striking distinction between the index sequences of the left and right halves of the single element in the array.

The index sequence of the duplicate numbers in the left half is always (even, odd). That means one of the following conditions will be satisfied if we are in the left half:

If the current index is even, the element at the next odd index will be the same as the current element.

Similarly, If the current index is odd, the element at the preceding even index will be the same as the current element.

The index sequence of the duplicate numbers in the right half is always (odd, even). That means one of the following conditions will be satisfied if we are in the right half:

If the current index is even, the element at the preceding odd index will be the same as the current element.

Similarly, If the current index is odd, the element at the next even index will be the same as the current element.

Now, we can easily identify the left and right halves, just by checking the sequence of the current index, i, like the following:

If (i % 2 == 0 and arr[i] == arr[i+1]) or (i%2 == 1 and arr[i] == arr[i-1]), we are in the left half.

If (i % 2 == 0 and arr[i] == arr[i-1]) or (i%2 == 1 and arr[i] == arr[i+1]), we are in the right half.

Note: In our case, the index i refers to the index ‘mid’.

How to eliminate the halves:

If we are in the left half of the single element, we have to eliminate this left half (i.e. low = mid+1). Because our single element appears somewhere on the right side.

If we are in the right half of the single element, we have to eliminate this right half (i.e. high = mid-1). Because our single element appears somewhere on the left side.

Now, we have resolved the problems and we can use the binary search accordingly.

Algorithm:

The steps are as follows:

If n == 1: This means the array size is 1. If the array contains only one element, we will return that element only.

If arr[0] != arr[1]: This means the very first element of the array is the single element. So, we will return arr[0].

If arr[n-1] != arr[n-2]: This means the last element of the array is the single element. So, we will return arr[n-1].

Place the 2 pointers i.e. low and high: Initially, we will place the pointers excluding index 0 and n-1 like this: low will point to index 1, and high will point to index n-2 i.e. the second last index.

Calculate the ‘mid’: Now, inside a loop, we will calculate the value of ‘mid’ using the following formula:

mid = (low+high) // 2 ( ‘//’ refers to integer division)

Check if arr[mid] is the single element:

If arr[mid] != arr[mid-1] and arr[mid] != arr[mid+1]: If this condition is true for arr[mid], we can conclude arr[mid] is the single element. We will return arr[mid].

If (mid % 2 == 0 and arr[mid] == arr[mid+1])

or (mid%2 == 1 and arr[mid] == arr[mid-1]): This means we are in the left half and we should eliminate it as our single element appears on the right. So, we will do this:

low = mid+1.

Otherwise, we are in the right half and we should eliminate it as our single element appears on the left. So, we will do this: high = mid-1.

The steps from 5 to 8 will be inside a loop and the loop will continue until low crosses high.

public int singleNonDuplicate(int[] nums)  
{  
 if(nums.length==1 || nums[0] != nums[1])  
 {  
 return nums[0];  
 }  
 if(nums[nums.length-1] != nums[nums.length-2])  
 {  
 return nums[nums.length-1];  
 }  
 int low = 0;  
 int high = nums.length-1;  
 while(low<=high)  
 {  
 int mid = (low+high)/2;  
 if(nums[mid] != nums[mid-1] && nums[mid] != nums[mid+1] )  
 {  
 return nums[mid];  
 }  
 else if(nums[mid] == nums[mid-1] && mid%2 == 1 || nums[mid] == nums[mid+1] && mid%2 == 0)  
 {  
 low = mid+1;  
 }  
 else  
 {  
 high = mid-1;  
 }  
 }  
 return -1;  
}

**Time Complexity:** O(logN), N = size of the given array.

**Space Complexity**: O(1) as we are not using any extra space.

**9.** **Peak element**

**Problem Statement**: Given an array of length N. Peak element is defined as the element greater than both of its neighbors. Formally, if 'arr[i]' is the peak element, 'arr[i - 1]' < 'arr[i]' and 'arr[i + 1]' < 'arr[i]'. Find the index(0-based) of a peak element in the array. If there are multiple peak numbers, return the index of any peak number.

You may imagine that nums[-1] = nums[n] = -∞. In other words, an element is always considered to be strictly greater than a neighbor that is outside the array.

**Example 1:**

Input Format: arr[] = {1,2,3,4,5,6,7,8,5,1}

Result: 7

Explanation: In this example, there is only 1 peak that is at index 7.

**Example 2:**

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

Result: 1

Explanation: In this example, there are 2 peak numbers that are at indices 1 and 5. We can consider any of them.

**Example 3:**

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

Result: 4

Explanation: In this example, there is only 1 peak that is at the index 4.

**Example 4:**

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

Result: 0

Explanation: In this example, there is only 1 peak that is at the index 0.

**What is a peak element?**

A peak element in an array refers to the element that is greater than both of its neighbors. Basically, if arr[i] is the peak element, arr[i] > arr[i-1] and arr[i] > arr[i+1].

**Brute – Force Approach :**

A simple approach involves iterating through the array and checking specific conditions for each element to determine the peak. By considering all the necessary conditions, including edge cases, our final condition can be summarized as follows:

If ((i == 0 or arr[i-1] < arr[i]) and (i == n-1 or arr[i] > arr[i+1])), we have found a peak. In such cases, we can return the index of the element satisfying this condition.

public static int findPeakElement(ArrayList<Integer> arr) {  
 int n = arr.size(); // Size of array.  
  
 for (int i = 0; i < n; i++) {  
 // Checking for the peak:  
 if ((i == 0 || arr.get(i - 1) < arr.get(i))  
 && (i == n - 1 || arr.get(i) > arr.get(i + 1))) {  
 return i;  
 }  
 }  
 // Dummy return statement  
 return -1;  
}

**Time Complexity**: O(N), N = size of the given array.

**Space Complexity**: O(1) as we are not using any extra space.

**Optimal Approach(Using Binary Search):**

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.

Until now, we have found how to identify if an element is a peak. But since binary search works by reducing the search space by half, we have to find a way to identify the halves and then eliminate them accordingly.

How to identify the halves:

By observing the above image, we can clearly notice a striking distinction between the left and right halves of the peak element in the array.

The left half of the peak element has an increasing order. This means for every index i, arr[i-1] < arr[i].

On the contrary, the right half of the peak element has a decreasing order. This means for every index i, arr[i+1] < arr[i].

Now, using the above observation, we can easily identify the left and right halves, just by checking the property of the current index, i, like the following:

If arr[i] > arr[i-1]: we are in the left half.

If arr[i] > arr[i+1]: we are in the right half.

How to eliminate the halves accordingly:

If we are in the left half of the peak element, we have to eliminate this left half (i.e.

low = mid+1). Because our peak element appears somewhere on the right side.

If we are in the right half of the peak element, we have to eliminate this right half (i.e. high = mid-1). Because our peak element appears somewhere on the left side.

p>Now, let’s see if these conditions are enough to handle the array with multiple peaks. Based on the observation, in an array with multiple peaks, an index has four possible positions as follows:

The index is a common point where a decreasing sequence ends and an increasing sequence begins.

The index might be on the left half.

The index might be the peak itself.

The index might be on the right half.

Until now, we have found how to identify if an element is a peak and how to identify the halves and then eliminate them accordingly. So, the last 3 cases have been resolved. We have to find out how the first case should be handled.

If an index is a common point where a decreasing sequence ends and an increasing sequence begins, we can actually eliminate either the left or right half. Because both halves of such an index contain a peak.

So, we decide to merge this case with the condition If arr[i+1] < arr[i]. You can choose otherwise as well.

Algorithm:

Note: At the start of the algorithm, we address the edge cases of identifying the peak element without requiring separate conditions during the check for arr[mid] inside the loop. And the search space will be from index 1 to n-2 as indices 0 and n-1 have already been checked in the edge cases.

The final steps will be as follows:

If n == 1: This means the array size is 1. If the array contains only one element, we will return that index i.e. 0.

If arr[0] > arr[1]: This means the very first element of the array is the peak element. So, we will return the index 0.

If arr[n-1] > arr[n-2]: This means the last element of the array is the peak element. So, we will return the index n-1.

Place the 2 pointers i.e. low and high: Initially, we will place the pointers excluding index 0 and n-1 like this: low will point to index 1, and high will point to index n-2 i.e. the second last index.

Calculate the ‘mid’: Now, inside a loop, we will calculate the value of ‘mid’ using the following formula:

mid = (low+high) // 2 ( ‘//’ refers to integer division)

Check if arr[mid] is the peak element:

If arr[mid] > arr[mid-1] and arr[mid] > arr[mid+1]: If this condition is true for arr[mid], we can conclude arr[mid] is the peak element. We will return the index ‘mid’.

If arr[mid] > arr[mid-1]: This means we are in the left half and we should eliminate it as our peak element appears on the right. So, we will do this:

low = mid+1.

Otherwise, we are in the right half and we should eliminate it as our peak element appears on the left. So, we will do this: high = mid-1. This case also handles the case for the index ‘mid’ being a common point of a decreasing and increasing sequence. It will consider the left peak and eliminate the right peak.

The steps from 5 to 8 will be inside a loop and the loop will continue until low crosses high.

public static int findPeakElement(int[] nums)  
{  
 if(nums.length==1 || nums[0]>nums[1])  
 {  
 return 0;  
 }  
 if(nums[nums.length-1] > nums[nums.length-2])  
 {  
 return nums.length-1;  
 }  
 int low = 1;  
 int high = nums.length-2;  
 while(low<=high)  
 {  
 int mid = (low+high)/2;  
 if(nums[mid]>nums[mid-1] && nums[mid]>nums[mid+1])  
 {  
 return mid;  
 }  
 else if(nums[mid]>nums[mid-1])  
 {  
 low = mid+1;  
 }  
 else  
 {  
 high = mid-1;  
 }  
 }  
  
 return -1;  
}

**Time Complexity**: O(logN), N = size of the given array.

**Space Complexity**: O(1)

**10.** **Search in Rotated Sorted Array I**

**Example 1:**

Input Format: arr = [4,5,6,7,0,1,2,3], k = 0

Result: 4

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 Format: arr = [4,5,6,7,0,1,2], k = 3

Result: -1

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

**Brute – Force Approach :**

public static int search(ArrayList<Integer> arr, int n, int k) {  
 for (int i = 0; i < n; i++) {  
 if (arr.get(i) == k)  
 return i;  
 }  
 return -1;  
}

**Time Complexity:** O(N), N = size of the given array.

**Space Complexity:** O(1)

**Optimal Approach(Using Binary Search):**

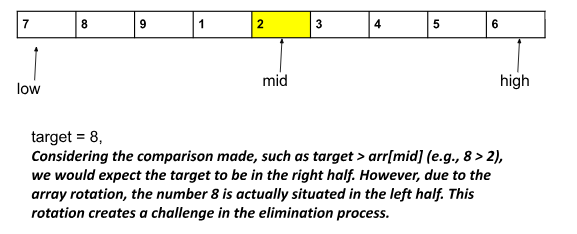
The primary objective of the Binary Search algorithm is to efficiently determine the appropriate half to eliminate, thereby reducing the search space by half. It does this by determining a specific condition that ensures that the target is not present in that half.

Observation:

To utilize the binary search algorithm effectively, it is crucial to ensure that the input array is sorted. By having a sorted array, we guarantee that each index divides the array into two sorted halves. In the search process, we compare the target value with the middle element, i.e. arr[mid], and then eliminate either the left or right half accordingly. This elimination becomes feasible due to the inherent property of the sorted halves(i.e. Both halves always remain sorted).

However, in this case, the array is both rotated and sorted. As a result, the property of having sorted halves no longer holds. This disruption in the sorting order affects the elimination process, making it unreliable to determine the target's location by solely comparing it with arr[mid].

To illustrate this situation, consider the following example:



Key Observation: Though the array is rotated, we can clearly notice that for every index, one of the 2 halves will always be sorted. In the above example, the right half of the index mid is sorted.

So, to efficiently search for a target value using this observation, we will follow a simple two-step process.

First, we identify the sorted half of the array.

Once found, we determine if the target is located within this sorted half.

If not, we eliminate that half from further consideration.

Conversely, if the target does exist in the sorted half, we eliminate the other half.

Algorithm:

The steps are as follows:

Place the 2 pointers i.e. low and high: Initially, we will place the pointers like this: low will point to the first index, and high will point to the last index.

Calculate the ‘mid’: Now, inside a loop, we will calculate the value of ‘mid’ using the following formula:

mid = (low+high) // 2 ( ‘//’ refers to integer division)

Check if arr[mid] == target: If it is, return the index mid.

Identify the sorted half, check where the target is located, and then eliminate one half accordingly:

If arr[low] <= arr[mid]: This condition ensures that the left part is sorted.

If arr[low] <= target && target <= arr[mid]: It signifies that the target is in this sorted half. So, we will eliminate the right half (high = mid-1).

Otherwise, the target does not exist in the sorted half. So, we will eliminate this left half by doing low = mid+1.

Otherwise, if the right half is sorted:

If arr[mid] <= target && target <= arr[high]: It signifies that the target is in this sorted right half. So, we will eliminate the left half (low = mid+1).

Otherwise, the target does not exist in this sorted half. So, we will eliminate this right half by doing high = mid-1.

Once, the ‘mid’ points to the target, the index will be returned.

This process will be inside a loop and the loop will continue until low crosses high. If no index is found, we will return -1.

public int search(int[] arr, int target)  
{  
 int n = arr.length;  
 int low = 0;  
 int high = arr.length - 1;  
 while(low <= high)  
 {  
 int mid = (low+high)/2;  
 if(arr[mid] == target)  
 {  
 return mid;  
 }  
 else if(arr[low]<=arr[mid])  
 {  
 if(arr[low] <= target && target <= arr[mid])  
 high = mid - 1;  
 else  
 low = mid + 1;  
 }  
 else  
 {  
 if(arr[mid] <= target && target <= arr[high])  
 low = mid + 1;  
 else  
 high = mid -1 ;  
 }  
 }  
 return -1;  
}

**Time Complexity:** O(logN)

**Space Complexity:** O(1)

**11.** **Search in Rotated Sorted Array II**

**Example 1:**

Input Format:

arr = [7, 8, 1, 2, 3, 3, 3, 4, 5, 6], k = 3

Result:

True

Explanation:

The element 3 is present in the array. So, the answer is True.

**Example 2:**

Input Format:

arr = [7, 8, 1, 2, 3, 3, 3, 4, 5, 6], k = 10

Result:

False

Explanation:

The element 10 is not present in the array. So, the answer is False.

**Brute – Force Approach :**

public static boolean searchInARotatedSortedArrayII(int []arr, int k) {  
 int n = arr.length; // size of the array.  
 for (int i = 0; i < n; i++) {  
 if (arr[i] == k) return true;  
 }  
 return false;  
}

**Time Complexity:** O(N), N = size of the given array.

**Space Complexity:** O(1)

**Optimal Approach(Using Binary Search):**

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.

In the previous problem, in order to efficiently search for the target value, we followed a simple two-step process.

First, we identify the sorted half of the array.

Once found, we determine if the target is located within this sorted half.

If not, we eliminate that half from further consideration.

Conversely, if the target does exist in the sorted half, we eliminate the other half.

Let’s observe how we identify the sorted half:

We basically compare arr[mid] with arr[low] and arr[high] in the following way:

If arr[low] <= arr[mid]: In this case, we identified that the left half is sorted.

If arr[mid] <= arr[high]: In this case, we identified that the right half is sorted.

This check was effective in the previous problem, where there were no duplicate numbers. However, in the current problem, the array may contain duplicates. Consequently, the previous approach will not work when arr[low] = arr[mid] = arr[high].

How to handle the edge case arr[low] = arr[mid] = arr[high]:

In the algorithm, we first check if arr[mid] is the target before identifying the sorted half. If arr[mid] is not our target, we encounter this edge case. In this scenario, since arr[mid] = arr[low] = arr[high], it means that neither arr[low] nor arr[high] can be the target. To handle this edge case, we simply remove arr[low] and arr[high] from our search space, without affecting the original algorithm.

To eliminate elements arr[low] and arr[high], we can achieve this by simply incrementing the low pointer and decrementing the high pointer by one step. We will continue this process until the condition arr[low] = arr[mid] = arr[high] is no longer satisfied.

Note: As long as this condition is met, we will skip the steps of determining the sorted half and eliminating one of the halves based on the target's location. Instead, we will solely focus on eliminating arr[low] and arr[high].

We will apply the same algorithm as the previous problem by just adding an extra check to handle the above edge case.

Algorithm:

The steps are as follows:

Place the 2 pointers i.e. low and high: Initially, we will place the pointers like this: low will point to the first index, and high will point to the last index.

Calculate the ‘mid’: Now, inside a loop, we will calculate the value of ‘mid’ using the following formula:

mid = (low+high) // 2 ( ‘//’ refers to integer division)

Check if arr[mid] = target: If it is, return True.

Check if arr[low] = arr[mid] = arr[high]: If this condition is satisfied, we will just increment the low pointer and decrement the high pointer by one step. We will not perform the later steps until this condition is no longer satisfied. So, we will continue to the next iteration from this step.

Identify the sorted half, check where the target is located, and then eliminate one half accordingly:

If arr[low] <= arr[mid]: This condition ensures that the left part is sorted.

If arr[low] <= target && target <= arr[mid]: It signifies that the target is in this sorted half. So, we will eliminate the right half (high = mid-1).

Otherwise, the target does not exist in the sorted half. So, we will eliminate this left half by doing low = mid+1.

Otherwise, if the right half is sorted:

If arr[mid] <= target && target <= arr[high]: It signifies that the target is in this sorted right half. So, we will eliminate the left half (low = mid+1).

Otherwise, the target does not exist in this sorted half. So, we will eliminate this right half by doing high = mid-1.

Once, the ‘mid’ points to the target, we will return True.

This process will be inside a loop and the loop will continue until low crosses high. If no element is found, we will return False.

public boolean search(int[] nums, int target)  
{  
 int low = 0;  
 int high = nums.length-1;  
 while(low<=high)  
 {  
 int mid = (low+high)/2;  
 if(nums[mid] == target)  
 {  
 return true;  
 }  
 else if(nums[mid] == nums[low] && nums[mid] == nums[high])  
 {  
 low++;  
 high--;  
 }  
 else if(nums[mid]<=nums[high])  
 {  
 if(nums[mid]<=target && target<=nums[high])  
 {  
 low = mid + 1;  
 }  
 else  
 {  
 high = mid - 1;  
 }  
 }  
 else  
 {  
 if(nums[low]<=target&&target<=nums[mid])  
 {  
 high = mid - 1;  
 }  
 else  
 {  
 low = mid + 1;  
 }  
 }  
 }  
 return false;  
}

**Time Complexity:** O(logN)

**Space Complexity:** O(1)

**12.** **Minimum in Rotated Sorted Array**

**Example 1:**

Input Format:

arr = [4,5,6,7,0,1,2,3]

Result:

0

Explanation:

Here, the element 0 is the minimum element in the array.

**Example 2:**

Input Format:

arr = [3,4,5,1,2]

Result:

1

Explanation:

Here, the element 1 is the minimum element in the array.

**Brute – Force Approach :**

public static int findMin(int []arr) {  
 int n = arr.length; // size of the array.  
 int mini = Integer.*MAX\_VALUE*;  
 for (int i = 0; i < n; i++) {  
 // Always keep the minimum.  
 mini = Math.*min*(mini, arr[i]);  
 }  
 return mini;  
}

**Time Complexity**: O(N)

**Space Complexity**: O(1)

**Optimal Approach(Using Binary Search):**

In this situation, we have two possibilities to consider. The sorted half of the array may or may not include the minimum value. However, we can leverage the property of the sorted half, specifically that the leftmost element of the sorted half will always be the minimum element within that particular half.

During each iteration, we will select the leftmost element from the sorted half and discard that half from further consideration. Among all the selected elements, the minimum value will serve as our answer.

To facilitate this process, we will declare a variable called 'ans' and initialize it with a large number. Then, at each step, after selecting the leftmost element from the sorted half, we will compare it with 'ans' and update 'ans' with the smaller value (i.e., min(ans, leftmost\_element)).

Note: If, at any index, both the left and right halves of the array are sorted, we have the flexibility to select the minimum value from either half and eliminate that particular half (in this case, the left half is chosen first). The algorithm already takes care of this case, so there is no need for explicit handling.

If both the left and right halves of an index are sorted, it implies that the entire search space between the low and high indices is also sorted. In this case, there is no need to conduct a binary search within that segment to determine the minimum value. Instead, we can simply select the leftmost element as the minimum.

The condition to check will be arr[low] <= arr[mid] && arr[mid] <= arr[high]. We can shorten this into arr[low] <= arr[high] as well.

If arr[low] <= arr[high]: In this case, the array from index low to high is completely sorted. Therefore, we can simply select the minimum element, arr[low], and update the 'ans' variable with the minimum value i.e. min(ans, arr[low]). Once this is done, there is no need to continue with the binary search algorithm.

public int findMin(int[] nums)  
{  
 int low = 0;  
 int high = nums.length - 1;  
 int ans = 5001;  
 while(low <= high)  
 {  
 int mid = (low+high)/2;  
 if(nums[low] < nums[mid] && nums[mid] < nums[high])  
 {  
 if(nums[low]<ans)  
 ans = nums[low];  
 break;  
 }  
 else if(nums[low] <= nums[mid])  
 {  
 if(nums[low]<ans)  
 ans = nums[low];  
 low = mid + 1;  
 }  
 else  
 {  
 if(nums[mid]<ans)  
 ans = nums[mid];  
 high = mid - 1;  
 }  
 }  
 return ans;  
}

**Time Complexity:** O(logN)

**Space Complexity:** O(1)

**13.** **Find out how many times has an array been rotated**

**Example 1:**

Input Format:

arr = [4,5,6,7,0,1,2,3]

Result:

4

Explanation:

The original array should be [0,1,2,3,4,5,6,7]. So, we can notice that the array has been rotated 4 times.

**Example 2:**

Input Format:

arr = [3,4,5,1,2]

Result:

3

Explanation:

The original array should be [1,2,3,4,5]. So, we can notice that the array has been rotated 3 times.

**Brute – Force Approach :**

public static int findKRotation(int[] arr) {  
 int n = arr.length; //size of array.  
 int ans = Integer.*MAX\_VALUE*, index = -1;  
 for (int i = 0; i < n; i++) {  
 if (arr[i] < ans) {  
 ans = arr[i];  
 index = i;  
 }  
 }  
 return index;  
}

**Time Complexity**: O(N)

**Space Complexity**: O(1)

**Optimal Approach(Using Binary Search):**

We can easily observe that the number of rotations in an array is equal to the index(0-based index) of its minimum element.

So, in order to solve this problem, we have to find the index of the minimum element.

public static int findRot(int[] nums)  
{  
 int low = 0;  
 int high = nums.length - 1;  
 int ans = 5001;  
 int index = 0;  
 while(low <= high)  
 {  
 int mid = (low+high)/2;  
 if(nums[low] < nums[mid] && nums[mid] < nums[high])  
 {  
 if(nums[low]<ans)  
 {  
 ans = nums[low];  
 index = low;  
 }  
 break;  
 }  
 else if(nums[low] <= nums[mid])  
 {  
 if(nums[low]<ans)  
 {  
 ans = nums[low];  
 index = low;  
 }  
 low = mid + 1;  
 }  
 else  
 {  
 if(nums[mid]<ans)  
 {  
 ans = nums[mid];  
 index = mid;  
 }  
  
 high = mid - 1;  
 }  
 }  
 return index;  
}

**Time Complexity:** O(logN)

**Space Complexity:** O(1)

*BS ON 2D ARRAYS*

**1.** **Search in a sorted 2 D matrix**

**Example 1:**

Input Format:

N = 3, M = 4, target = 8,

mat[] =

1 2 3 4

5 6 7 8

9 10 11 12

Result:

true

Explanation:

The ‘target’ = 8 exists in the 'mat' at index (1, 3).

**Example 2:**

Input Format:

N = 3, M = 3, target = 78,

mat[] =

1 2 4

6 7 8

9 10 34

Result:

false

Explanation:

The ‘target' = 78 does not exist in the 'mat'. Therefore in the output, we see 'false'.

**Brute – Force Approach:**

The extremely naive approach is to get the answer by checking all the elements of the given matrix. So, we will traverse the matrix and check every element if it is equal to the given ‘target’.

public static boolean searchMatrix(ArrayList<ArrayList<Integer>> matrix, int target) {  
 int n = matrix.size(), m = matrix.get(0).size();  
  
 // traverse the matrix:  
 for (int i = 0; i < n; i++) {  
 for (int j = 0; j < m; j++) {  
 if (matrix.get(i).get(j) == target)  
 return true;  
 }  
 }  
 return false;  
}

**Time Complexity**: O(N X M)

**Space Complexity**: O(1)

**Better Approach:**

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.

The question specifies that each row in the given matrix is sorted. Therefore, to determine if the target is present in a specific row, we don't need to search column by column. Instead, we can efficiently use the binary search algorithm.

To make the time complexity even better, we won't use binary search on every row. We'll focus only on the particular row where the target might be located.

How to check if a specific row is containing the target:

If the target lies between the first and last element of the row, i (i.e. matrix[i][0] <= target && target <= matrix[i][m-1]), we can conclude that the target might be present in that specific row.

Once we locate the potentially relevant row containing the 'target', we need to confirm its presence. To accomplish this, we will utilize the Binary search algorithm, effectively reducing the time complexity.

public static boolean binarySearch(ArrayList<Integer> nums, int target) {  
 int n = nums.size(); //size of the array  
 int low = 0, high = n - 1;  
  
 // Perform the steps:  
 while (low <= high) {  
 int mid = (low + high) / 2;  
 if (nums.get(mid) == target) return true;  
 else if (target > nums.get(mid)) low = mid + 1;  
 else high = mid - 1;  
 }  
 return false;  
}  
public static boolean searchMatrix(ArrayList<ArrayList<Integer>> matrix, int target) {  
 int n = matrix.size();  
 int m = matrix.get(0).size();  
  
 for (int i = 0; i < n; i++) {  
 if (matrix.get(i).get(0) <= target && target <= matrix.get(i).get(m - 1)) {  
 return *binarySearch*(matrix.get(i), target);  
 }  
 }  
 return false;  
}

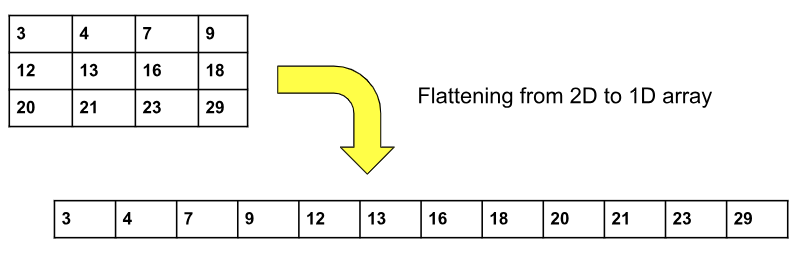
**Time Complexity**: O(N + logM), where N = given row number, M = given column number.

Reason: We are traversing all rows and it takes O(N) time complexity. But for all rows, we are not applying binary search rather we are only applying it once for a particular row. That is why the time complexity is O(N + logM) instead of O(N\*logM).

**Space Complexity**: O(1) as we are not using any extra space.

**Optimal Approach:**

If we flatten the given 2D matrix to a 1D array, the 1D array will also be sorted. By utilizing binary search on this sorted 1D array to locate the 'target' element, we can further decrease the time complexity. The flattening will be like the following:



But if we really try to flatten the 2D matrix, it will take O(N x M) time complexity and extra space to store the 1D array. In that case, it will not be the optimal solution anymore.

How to apply binary search on the 1D array without actually flattening the 2D matrix:

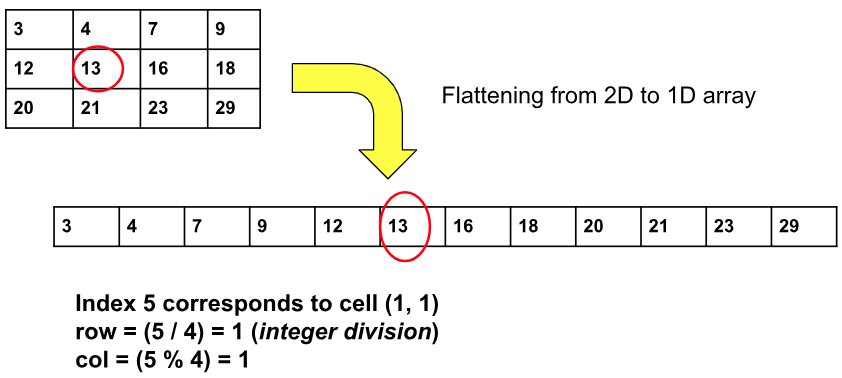
If we can figure out how to convert the index of the 1D array into the corresponding cell number in the 2D matrix, our task will be complete. In this scenario, we will use the binary search with the indices of the imaginary 1D array, ranging from 0 to (NxM)-1(total no. of elements in the 1D array = NxM). When comparing elements, we will convert the index to the cell number and retrieve the element. Thus we can apply binary search in the imaginary 1D array.

How to convert 1D array index to the corresponding cell of the 2D matrix:

We will use the following formula:

If index = i, and no. of columns in the matrix = m, the index i corresponds to the cell with

row = i / m and col = i % m. More formally, the cell is (i / m, i % m)(0-based indexing).



The range of the indices of the imaginary 1D array is [0, (NxM)-1] and in this range, we will apply binary search.

public boolean searchMatrix(int[][] matrix, int target)  
{  
 int row = matrix.length;  
 int col = matrix[0].length;  
 int low = 0;  
 int high = (row\*col)-1;  
 while(low<=high)  
 {  
 int mid = (low+high)/2;  
 int r = mid/col;  
 int c = mid%col;  
 if(matrix[r][c] == target)  
 {  
 return true;  
 }  
 else if(matrix[r][c] < target)  
 {  
 low = mid + 1;  
 }  
 else  
 {  
 high = mid - 1;  
 }  
 }  
 return false;  
}

**Time Complexity**: O(log(NxM)), where N = given row number, M = given column number.

Reason: We are applying binary search on the imaginary 1D array of size NxM.

**Space Complexity**: O(1) as we are not using any extra space.

1. **Search in a row and column wise sorted matrix**

**Example 1:**

Input Format:

N = 5, M = 5, target = 14

mat[] =

Result:

true

Explanation:

Target 14 is present in the cell (3, 2)(0-based indexing) of the matrix. So, the answer is true.

**Example 2:**

Input Format:

N = 3, M = 3, target = 12,

mat[] =

Result:

false

Explanation:

As target 12 is not present in the matrix, the answer is false.

**Brute – Force Approach:**

same as above.

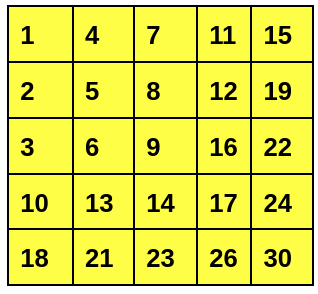
**Better Approach:**

Same as above.

**Optimal Approach:**

We can enhance this method by adjusting how we move through the matrix. Let's take a look at the four corners: (0, 0), (0, m-1), (n-1, 0), and (n-1, m-1). By observing these corners, we can identify variations in how we traverse the matrix.

Assume the given ‘target’ = 14 and given matrix =



Observations:

Cell (0, 0): Assume we are starting traversal from (0, 0) and we are searching for 14. Now, this row and column are both sorted in increasing order. So, we cannot determine, how to move i.e. row-wise or column-wise. That is why, we cannot start traversal from (0, 0).

Cell (0, m-1): Assume we are starting traversal from (0, m-1) and we are searching for 14. Now, in this case, the row is in decreasing order and the column is in increasing order. Therefore, if we start traversal from (0, m-1), in the following way, we can easily determine how we should move.

If matrix[0][m-1] > target: We should move row-wise.

If matrix[0][m-1] < target: We need bigger elements and so we should move column-wise.

Cell (n-1, m-1): Assume we are starting traversal from (n-1, m-1) and we are searching for 14. Now, this row and column are both sorted in decreasing order. So, we cannot determine, how to move i.e. row-wise or column-wise. That is why, we cannot start traversal from (n-1, m-1).

Cell (n-1, 0): Assume we are starting traversal from (n-1, 0) and we are searching for 14. Now, in this case, the row is in increasing order and the column is in decreasing order. Therefore, if we start traversal from (n-1, 0), in the following way, we can easily determine how we should move.

If matrix[n-1][0] < target: We should move row-wise.

If matrix[n-1][0] > target: We need smaller elements and so we should move column-wise.

From the above observations, it is quite clear that we should start the matrix traversal from either the cell (0, m-1) or (n-1, 0).

Note: Here in this approach, we have chosen the cell (0, m-1) to start with. You can choose otherwise.

Using the above observations, we will start traversal from the cell (0, m-1) and every time we will compare the target with the element at the current cell. After comparing we will either eliminate the row or the column accordingly like the following:

If current element > target: We need the smaller elements to reach the target. But the column is in increasing order and so it contains only greater elements. So, we will eliminate the column by decreasing the current column value by 1(i.e. col--) and thus we will move row-wise.

If current element < target: In this case, We need the bigger elements to reach the target. But the row is in decreasing order and so it contains only smaller elements. So, we will eliminate the row by increasing the current row value by 1(i.e. row++) and thus we will move column-wise.

Algorithm:

As we are starting from the cell (0, m-1), the two variables i.e. ‘row’ and ‘col’ will point to 0 and m-1 respectively.

We will do the following steps until row < n and col >= 0(i.e. while(row < n && col >= 0)):

If matrix[row][col] == target: We have found the target and so we will return true.

If matrix[row][col] > target: We need the smaller elements to reach the target. But the column is in increasing order and so it contains only greater elements. So, we will eliminate the column by decreasing the current column value by 1(i.e. col--) and thus we will move row-wise.

If matrix[row][col] < target: In this case, We need the bigger elements to reach the target. But the row is in decreasing order and so it contains only smaller elements. So, we will eliminate the row by increasing the current row value by 1(i.e. row++) and thus we will move column-wise.

If we are outside the loop without getting any matching element, we will return false.

public boolean searchMatrix(int[][] matrix, int target)  
{  
 int row = matrix.length;  
 int col = matrix[0].length;  
 int r = 0;  
 int c = col - 1;  
 while(r>=0 && r<row && c>=0 && c<col)  
 {  
 if(matrix[r][c] == target)  
 {  
 return true;  
 }  
 else if(matrix[r][c]>target)  
 {  
 c--;  
 }  
 else  
 {  
 r++;  
 }  
 }  
 return false;  
}

**Time Complexity**: O(N+M)

**Space Complexity**: O(1)

**3.Find Peak Element**

A peak element in a 2D grid is an element that is strictly greater than all of its adjacent neighbors to the left, right, top, and bottom.

Given a 0-indexed m x n matrix mat where no two adjacent cells are equal, find any peak element mat[i][j] and return the length 2 array [i,j].

You may assume that the entire matrix is surrounded by an outer perimeter with the value -1 in each cell.

You must write an algorithm that runs in O(m log(n)) or O(n log(m)) time.

**Example 1:**

Input: mat = [[1,4],[3,2]]

Output: [0,1]

Explanation: Both 3 and 4 are peak elements so [1,0] and [0,1] are both acceptable answers.

**Example 2:**

Input: mat = [[10,20,15],[21,30,14],[7,16,32]]

Output: [1,1]

Explanation: Both 30 and 32 are peak elements so [1,1] and [2,2] are both acceptable answers.

**Brute – Force Approach:**

same as above.

**Optimal Approach:**

Intuition

A more efficient way is to apply binary search on columns instead of rows.

The idea is :

We find the maximum element in the middle column.

Then, we check if it is a peak.

If not, we move left or right based on the larger adjacent element.

Algorithm :

Perform a binary search on columns.

Find the row index of the maximum element in the middle column.

Check if this maximum element is a peak:

If it is greater than both left and right neighbors, return its position.

If the left neighbor is greater, move left.

If the right neighbor is greater, move right.

Continue binary searching on columns until a peak is found.

public static int[] findPeakGrid(int[][] mat)  
{  
 int row = mat.length;  
 int col = mat[0].length;  
 int low = 0;  
 int high = col-1;  
 while(low<=high)  
 {  
 int mid = (low+high)/2;  
 int r = *search*(mat,mid);  
 int left = mid-1>=0?mat[r][mid-1]:-1;  
 int right = mid+1<col?mat[r][mid+1]:-1;  
 if(mat[r][mid] >= left && mat[r][mid] >= right)  
 {  
 return new int[]{r,mid};  
 }  
 else if(mat[r][mid]<left)  
 {  
 high = mid - 1;  
 }  
 else  
 {  
 low = mid + 1;  
 }  
 }  
 return new int[]{-1,-1};  
}  
  
public static int search(int[][] mat,int col)  
{  
 int row = -1;  
 int value = 0;  
 for(int i=0;i<mat.length;i++)  
 {  
 if(mat[i][col]>value)  
 {  
 value = mat[i][col];  
 row = i;  
 }  
 }  
 return row;  
}

**Time Complexity:**

Finding the maximum in a column: O(m) , Binary search on columns: O(log n)

Total: O(m log n)

**Space Complexity:**

O(1) since no extra space is used.

**4.Median of Row Wise Sorted Matrix**

**Example 1:**

Input Format:M = 3, N = 3, matrix[][] =

1 4 9

2 5 6

3 7 8

Result: 5

Explanation: If we find the linear sorted array, the array becomes 1 2 3 4 5 6 7 8 9. So, median = 5

**Example 2:**

Input Format:M = 3, N = 3, matrix[][] =

1 3 8

2 3 4

1 2 5

Result: 3

Explanation: If we find the linear sorted array, the array becomes 1 1 2 2 3 3 4 5 7 8. So, median = 3

 **If the number of elements (n) is odd:**

* The median is the middle element.
* Formula: Median=arr[n/2]

 **If the number of elements (n) is even:**

* The median is the average of the two middle elements.
* Formula: Median=arr[n−1/2]+arr[n/2]/2

**Brute – Force Approach:**

The extremely naive approach is to use a linear array/list to store the elements of the given matrix. Now, we will sort the list and return the middle element.

public static int median(int matrix[][], int m, int n) {  
 List<Integer> lst = new ArrayList<>();  
  
 for (int i = 0; i < m; i++) {  
 for (int j = 0; j < n; j++) {  
 lst.add(matrix[i][j]);  
 }  
 }  
  
 // Sort the list:  
 Collections.*sort*(lst);  
 return lst.get((m \* n) / 2);  
}

**Time Complexity**: O(MXN) + O(MXN(log(MXN))), where M = number of row in the given matrix, N = number of columns in the given matrix

Reason: At first, we are traversing the matrix to copy the elements. This takes O(MXN) time complexity. Then we are sorting the linear array of size (MXN), that takes O(MXN(log(MXN))) time complexity

**Space Complexity**: O(MXN) as we are using a temporary list to store the elements of the matrix.

**Optimal Approach:**

Now, if we wish to further optimize the previous approach we cannot afford to check every element. So, we have to eliminate some parts to reduce the time complexity under O(MXN). This is where the binary search algorithm comes in

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.

Observations:

What is the search space where we will apply binary search?

If we carefully observe, our answer lies between the smallest and the largest number in the given matrix. So, the search space will be [min(matrix), max(matrix)].

While applying binary search how to check if an element ‘x’ is a possible median?

If ‘x’ is the median, the number of elements smaller or equal to ‘x’ will be surely greater than (MXN) // 2 (integer division).

How to check how many numbers are smaller or equal to an element ‘mid’?

One of the ways is to traverse the whole matrix and count the numbers. But in that case, the time complexity will be high. So, we have to find other ways. It is given that the matrix is row-wise sorted. So, we can apply the concept of upper bound

For every particular row, we will find the upper bound of the current element ‘mid’. The index returned will be the number of smaller or equal elements in that row.

We will do it for each row and add them to get the total number for the whole matrix.

Mathematically, smaller\_equal\_in\_row = upperBound(matrix[row], mid)

We will just convert the above observation into code.

public static int findMedian(int matrix[][], int m, int n)  
{  
 int ans = 0;  
 int low = matrix[0][0];  
 int high = -1;  
 for(int i=0;i<m;i++)  
 {  
 if(matrix[i][0] < low)  
 {  
 low = matrix[i][0];  
 }  
 if(matrix[i][n-1] > high)  
 {  
 high = matrix[i][n-1];  
 }  
 }  
  
 while(low<=high)  
 {  
 int mid = (low+high)/2;  
 int lesserOrEqual = *blackBox*(matrix,m,n,mid);  
 if(lesserOrEqual <= (m\*n)/2)  
 {  
 low = mid+1;  
 }  
 else  
 {  
 high = mid - 1;  
 ans = mid;  
 }  
 }  
 return ans;  
}  
  
public static int blackBox(int[][] matrix,int rows,int cols,int mid)  
{  
 int cnt = 0;  
 for(int i=0;i<rows;i++)  
 {  
 cnt += *upperBound*(matrix[i],mid);  
 }  
 return cnt;  
}  
  
public static int upperBound(int[] arr,int value)  
{  
 int low = 0;  
 int high = arr.length-1;  
 while(low<=high)  
 {  
 int mid = (low+high)/2;  
 if(arr[mid]>value)  
 {  
 high = mid - 1;  
 }  
 else  
 {  
 low = mid + 1;  
 }  
 }  
 return low;  
}

**Time Complexity**: O(log(109)) X O(M(logN)), where M = number of rows in the given matrix, N = number of columns in the given matrix

Reason: Our search space lies between [0, 109] as the min(matrix) can be 0 and the max(matrix) can be 109. We are applying binary search in this search space and it takes O(log(109)) time complexity. Then we call countSmallEqual() function for every ‘mid’ and this function takes O(M(logN)) time complexity.

**Space Complexity** : O(1) as we are not using any extra space

**5.** **Find the row with maximum number of 1's**

**Example 1:**

Input Format:

n = 3, m = 3,

mat[] =

1 1 1

0 0 1

0 0 0

Result:

0

Explanation:

The row with the maximum number of ones is 0 (0 - indexed).

**Example 2:**

Input Format:

n = 2, m = 2 ,

mat[] =

0 0

0 0

Result:

-1

Explanation:

The matrix does not contain any 1. So, -1 is the answer.

**Brute – Force Approach:**

The extremely naive approach is to traverse the matrix as usual using nested loops and for every single row count the number of 1’s. Finally, we will return the row with the maximum no. of 1’s. If multiple rows contain the maximum no. of 1’s we will return the row with the minimum index.

**Time Complexity:** O(n X m), where n = given row number, m = given column number.

**Space Complexity:** O(1) as we are not using any extra space.

**Optimal Approach:**

We cannot optimize the row traversal but we can optimize the counting of 1’s for each row.

Instead of counting the number of 1’s, we can use the following formula to calculate the number of 1’s:

Number\_of\_ones = m(number of columns) - first occurrence of 1(0-based index).

As, each row is sorted, we can find the first occurrence of 1 in each row using any of the following approaches:

lowerBound(1) (ref: Implement Lower Bound)

upperBound(0) (ref: Implement Upper Bound)

firstOccurrence(1) (ref: First and Last Occurrences in Array)

Note: Here, we are going to use the lowerBound() function to find the first occurrence. You can use the other methods as well.

public static int lowerBound(ArrayList<Integer> arr, int n, int x) {  
 int low = 0, high = n - 1;  
 int ans = n;  
  
 while (low <= high) {  
 int mid = (low + high) / 2;  
 // maybe an answer  
 if (arr.get(mid) >= x) {  
 ans = mid;  
 // look for smaller index on the left  
 high = mid - 1;  
 } else {  
 low = mid + 1; // look on the right  
 }  
 }  
 return ans;  
}  
public static int rowWithMax1s(ArrayList<ArrayList<Integer>> matrix, int n, int m) {  
 int cnt\_max = 0;  
 int index = -1;  
  
 // traverse the rows:  
 for (int i = 0; i < n; i++) {  
 // get the number of 1's:  
 int cnt\_ones = m - *lowerBound*(matrix.get(i), m, 1);  
 if (cnt\_ones > cnt\_max) {  
 cnt\_max = cnt\_ones;  
 index = i;  
 }  
 }  
 return index;  
}

**Time Complexity**: O(n X logm), where n = given row number, m = given column number.

Reason: We are using a loop running for n times to traverse the rows. Then we are applying binary search on each row with m columns.

**Space Complexity:** O(1) as we are not using any extra space.

*BS ON ANSWERS*

**1. Koko Eating Bananas**

Problem Statement: A monkey is given ‘n’ piles of bananas, whereas the 'ith' pile has ‘a[i]’ bananas. An integer ‘h’ is also given, which denotes the time (in hours) for all the bananas to be eaten.

Each hour, the monkey chooses a non-empty pile of bananas and eats ‘k’ bananas. If the pile contains less than ‘k’ bananas, then the monkey consumes all the bananas and won’t eat any more bananas in that hour

Find the minimum number of bananas ‘k’ to eat per hour so that the monkey can eat all the bananas within ‘h’ hours.

**Example 1:**

Input Format:

N = 4, a[] = {7, 15, 6, 3}, h = 8

Result:

5

Explanation:

If Koko eats 5 bananas/hr, he will take 2, 3, 2, and 1 hour to eat the piles accordingly. So, he will take 8 hours to complete all the piles.

**Example 2:**

Input Format:

N = 5, a[] = {25, 12, 8, 14, 19}, h = 5

Result:

25

Explanation:

If Koko eats 25 bananas/hr, he will take 1, 1, 1, 1, and 1 hour to eat the piles accordingly. So, he will take 5 hours to complete all the piles.

Before moving on to the solution, let’s understand how Koko will eat the bananas. Assume, the given array is {3, 6, 7, 11} and the given time i.e. h is 8.

First of all, Koko cannot eat bananas from different piles. He should complete the pile he has chosen and then he can go for another pile.

Now, Koko decides to eat 2 bananas/hour. So, in order to complete the first he will take

3 / 2 = 2 hours. Though mathematically, he should take 1.5 hrs but it is clearly stated in the question that after completing a pile Koko will not consume more bananas in that hour. So, for the first pile, Koko will eat 2 bananas in the first hour and then he will consume 1 banana in another hour.

From here we can conclude that we have to take ceil of (3/2). Similarly, we will calculate the times for other piles.

1st pile: ceil(3/2) = 2 hrs

2nd pile: ceil(6/2) = 3 hrs

3rd pile: ceil(7/2) = 4 hrs

4th pile: ceil(11/2) = 6 hrs

Koko will take 15 hrs in total to consume all the bananas from all the piles.

Observation: Upon observation, it becomes evident that the maximum number of bananas (represented by 'k') that Koko can consume in an hour is obtained from the pile that contains the largest quantity of bananas. Therefore, the maximum value of 'k' corresponds to the maximum element present in the given array.

So, our answer i.e. the minimum value of ‘k’ lies between 1 and the maximum element in the array i.e. max(a[]).

Now, let’s move on to the solution.

**Brute – Force Approach:**

The extremely naive approach is to check all possible answers from 1 to max(a[]). The minimum number for which the required time <= h, is our answer.

Algorithm:

First, we will find the maximum value i.e. max(a[]) in the given array.

We will run a loop(say i) from 1 to max(a[]), to check all possible answers.

For each number i, we will calculate the hours required to consume all the bananas from the pile. We will do this using the function calculateTotalHours(), discussed below.

The first i, for which the required hours <= h, we will return that value of i.

calculateTotalHours(a[], hourly):

a[] -> the given array

Hourly -> the possible number of bananas, Koko will eat in an hour.

We will iterate every pile of the given array using a loop(say i).

For every pile i, we will calculate the hour i.e. ceil(v[i] / hourly), and add it to the total hours.

public static int findMax(int[] v) {  
 int maxi = Integer.*MIN\_VALUE*;;  
 int n = v.length;  
 for (int i = 0; i < n; i++) {  
 maxi = Math.*max*(maxi, v[i]);  
 }  
 return maxi;  
}  
  
public static int calculateTotalHours(int[] v, int hourly) {  
 int totalH = 0;  
 int n = v.length;  
   
 for (int i = 0; i < n; i++) {  
 totalH += Math.*ceil*((double)(v[i]) / (double)(hourly));  
 }  
 return totalH;  
}  
  
public static int minimumRateToEatBananas(int[] v, int h) {  
 int maxi = *findMax*(v);  
   
 for (int i = 1; i <= maxi; i++) {  
 int reqTime = *calculateTotalHours*(v, i);  
 if (reqTime <= h) {  
 return i;  
 }  
 }  
   
 return maxi;  
}

**Time Complexity:** O(max(a[]) \* N), where max(a[]) is the maximum element in the array and N = size of the array.

Reason: We are running nested loops. The outer loop runs for max(a[]) times in the worst case and the inner loop runs for N times.

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

**Optimal Approach:**

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.

Now, we are not given any sorted array on which we can apply binary search. But, if we observe closely, we can notice that our answer space i.e. [1, max(a[])] is sorted. So, we will apply binary search on the answer space.

Algorithm:

First, we will find the maximum element in the given array i.e. max(a[]).

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 max(a[]).

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)

Eliminate the halves based on the time required if Koko eats ‘mid’ bananas/hr:

We will first calculate the total time(required to consume all the bananas in the array) i.e. totalH using the function calculateTotalHours(a[], mid):

If totalH <= h: On satisfying this condition, we can conclude that the number ‘mid’ is one of our possible answers. But we want the minimum number. So, we will eliminate the right half and consider the left half(i.e. high = mid-1).

Otherwise, the value mid is smaller than the number we want(as the totalH > h). This means the numbers greater than ‘mid’ should be considered and the right half of ‘mid’ consists of such numbers. So, we will eliminate the left half and consider the right half(i.e. low = mid+1).

Finally, outside the loop, we will return the value of low as the pointer will be pointing to the answer.

The steps from 2-4 will be inside a loop and the loop will continue until low crosses high.

Note: Please make sure to refer to the video and try out some test cases of your own to understand, how the pointer ‘low’ 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.

calculateTotalHours(a[], hourly):

a[] -> the given array

Hourly -> the possible number of bananas, Koko will eat in an hour.

We will iterate every pile of the given array using a loop(say i).

For every pile i, we will calculate the hour i.e. ceil(v[i] / hourly), and add it to the total hours.

Finally, we will return the total hours.

public static int minEatingSpeed(int[] piles, int h)  
{  
 int low = 0;  
 int high = *maximum*(piles);  
  
 while(low<=high)  
 {  
 int mid = (low+high)/2;  
 int totHours = *totalHours*(piles,mid);  
 if(totHours<=h)  
 {  
 high = mid - 1;  
 }  
 else  
 {  
 low = mid + 1;  
 }  
 }  
 return low;  
}  
  
public static int totalHours(int[] piles,int h)  
{  
 int total = 0;  
 for(int i=0;i<piles.length;i++)  
 {  
 total += Math.*ceil*((double)piles[i]/(double)h);  
 }  
 return total;  
}  
  
public static int maximum(int[] piles)  
{  
 int high = 0;  
 for(int i=0;i<piles.length;i++)  
 {  
 if(piles[i]>high)  
 {  
 high = piles[i];  
 }  
 }  
 return high;  
}

**Time Complexity**: O(N \* log(max(a[]))), where max(a[]) is the maximum element in the array and N = size of the array.

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

**2.Find the smallest Divisor**

Same as 1. Koko Eating Bananas

**3.** **Finding Sqrt of a number using Binary Search  
  
Example 1:**

Input Format:

n = 36

Result:

6

Explanation:

6 is the square root of 36.

**Example 2:**

Input Format:

n = 28

Result:

5

Explanation:

Square root of 28 is approximately 5.292. So, the floor value will be 5.

**Brute – Force Approach:**

We can guarantee that our answer will lie between the range from 1 to n i.e. the given number. So, we will perform a linear search on this range and we will find the maximum number x, such that x\*x <= n.

public static int floorSqrt(int n) {  
 int ans = 0;  
 // linear search on the answer space  
 for (long i = 1; i <= n; i++) {  
 long val = i \* i;  
 if (val <= (long) n) {  
 ans = (int) i;  
 } else {  
 break;  
 }  
 }  
 return ans;  
}

**Time Complexity**: O(N), N = the given number.

**Space Complexity**: O(1) as we are not using any extra space.

**Optimal Approach:**

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.

Now, we are not given any sorted array on which we can apply binary search. But, if we observe closely, we can notice that our answer space i.e. [1, n] is sorted. So, we will apply binary search on the answer space.

public int mySqrt(int x)  
{  
 long low = 1;  
 long high = x;  
 int ans = 0;  
 while(low<=high)  
 {  
 long mid = (low+high)/2;  
 long value = mid \* mid;  
 if(value <= (long)x)  
 {  
 ans = (int)mid;  
 low = mid + 1;  
 }  
 else  
 {  
 high = mid - 1;  
 }  
 }  
 return ans;  
}

**Time Complexity**: O(logN), N = size of the given array.

**Space Complexity**: O(1) as we are not using any extra space.

**4.** **Finding nth rt 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. If the 'nth root is not an integer, return -1.

**Example 1:**

Input Format:

N = 3, M = 27

Result:

3

Explanation:

The cube root of 27 is equal to 3.

**Example 2:**

Input Format:

N = 4, M = 69

Result:

-1

Explanation:

The 4th root of 69 does not exist. So, the answer is -1.

**Brute – Force Approach:**

We can guarantee that our answer will lie between the range from 1 to m i.e. the given number. So, we will perform a linear search on this range and we will find the number x, such that

func(x, n) = m. If no such number exists, we will return -1.

Note: func(x, n) returns the value of x raised to the power n i.e. xn.

public static long func(int b, int exp) {  
 long ans = 1;  
 long base = b;  
 while (exp > 0) {  
 if (exp % 2 == 1) {  
 exp--;  
 ans = ans \* base;  
 } else {  
 exp /= 2;  
 base = base \* base;  
 }  
 }  
 return ans;  
}  
  
public static int NthRoot(int n, int m) {  
 //Use linear search on the answer space:  
 for (int i = 1; i <= m; i++) {  
 long val = *func*(i, n);  
 if (val == (long)m) return i;  
 else if (val > (long)m) break;  
 }  
 return -1;  
}

**Time Complexity:** O(M), M = the given number.

**Space Complexity:** O(1) as we are not using any extra space.

**Optimal Approach:**

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.

Now, we are not given any sorted array on which we can apply binary search. But, if we observe closely, we can notice that our answer space i.e. [1, n] is sorted. So, we will apply binary search on the answer space.

Edge case: How to eliminate the halves:

Our first approach should be the following:

After placing low at 1 and high m, we will calculate the value of ‘mid’.

Now, based on the value of ‘mid’ raised to the power n, we will check if ‘mid’ can be our answer, and based on this value we will also eliminate the halves. If the value is smaller than m, we will eliminate the left half and if greater we will eliminate the right half.

But, if the given numbers m and n are big enough, we cannot store the value midn in a variable. So to resolve this problem, we will implement a function like the following:

func(n, m, mid):

We will first declare a variable ‘ans’ to store the value midn.

Now, we will run a loop for n times to multiply the ‘mid’ n times with ‘ans’.

Inside the loop, if at any point ‘ans’ becomes greater than m, we will return 2.

Once the loop is completed, if the ‘ans’ is equal to m, we will return 1.

If the value is smaller, we will return 0.

Now, based on the output of the above function, we will check if ‘mid’ is our possible answer or we will eliminate the halves. Thus we can avoid the integer overflow case.

public static int nRoot(int num,int root)  
{  
 int low = 0;  
 int high = num;  
 int ans = 0;  
 while(low<=high)  
 {  
 int mid = (low+high)/2;  
 int value = *nroot*(mid,num,root);  
 if(value == 1)  
 {  
 return mid;  
 }else if(value == 0)  
 {  
 low = mid + 1;  
 }  
 else  
 {  
 high = mid - 1;  
 }  
 }  
 return -1;  
}  
  
public static int nroot(int mid,int num,int root)  
{  
 int ans = 1;  
 for(int i=0;i<root;i++)  
 {  
 ans = ans \* mid;  
 if(ans>num)  
 return 2;  
 }  
 if(ans == num)  
 {  
 return 1;  
 }  
 return 0;  
}

**Time Complexity:** O(logN), N = size of the given array.

**Space Complexity:** O(1)

**5.** **Minimum days to make M bouquets.**

**Example 1:**

Input Format:

N = 8, arr[] = {7, 7, 7, 7, 13, 11, 12, 7}, m = 2, k = 3

Result:

12

Explanation:

On the 12th the first 4 flowers and the last 3 flowers would have already bloomed. So, we can easily make 2 bouquets, one with the first 3 and another with the last 3 flowers.

**Example 2:**

Input Format:

N = 5, arr[] = {1, 10, 3, 10, 2}, m = 3, k = 2

Result:

-1

Explanation:

If we want to make 3 bouquets of 2 flowers each, we need at least 6 flowers. But we are given only 5 flowers, so, we cannot make the bouquets.

Let's grasp the question better with the help of an example. Consider an array: {7, 7, 7, 7, 13, 11, 12, 7}. We aim to create bouquets with k, which is 3 adjacent flowers, and we need to make m, which is 2 such bouquets. Now, if we try to make bouquets on the 11th day, the first 4 flowers and the 6th and the last flowers would have bloomed. So, we will be having 6 flowers in total on the 11th day. However, we require two groups of 3 adjacent flowers each. Although we can form one group with the first 3 adjacent flowers, we cannot create a second group. Therefore, 11 is not the answer in this case.

If we choose the 12th day, we can make 2 such groups, one with the first 3 adjacent flowers and the other with the last 3 adjacent flowers. Hence, we need a minimum of 12 days to make 2 bouquets.

Observation:

Impossible case: To create m bouquets with k adjacent flowers each, we require a minimum of m\*k flowers in total. If the number of flowers in the array, represented by array-size, is less than m\*k, it becomes impossible to form m bouquets even after all the flowers have bloomed. In such cases, where array-size < m\*k, we should return -1.

Maximum possible answer: The maximum potential answer corresponds to the time needed for all the flowers to bloom. In other words, it is the highest value within the given array i.e. max(arr[]).

Minimum possible answer: The minimum potential answer corresponds to the time needed for atleast one flower to bloom. In other words, it is the smallest value within the given array i.e. min(arr[]).

Note: From the above observations, we can conclude that our answer lies between the range [min(arr[]), max(arr[])].

How to calculate the number of bouquets we can make on dth day:

We will count the number of adjacent bloomed flowers(say cnt) and whenever we get a flower that is not bloomed, we will add the number of bouquets we can make with ‘cnt’ adjacent flowers i.e. floor(cnt/k) to the answer. We will follow the process throughout the array.

**Brute – Force Approach:**

The extremely naive approach is to check all possible answers from min(arr[]) to max(arr[]). The minimum number for which possible() returns true, is our answer.

public static int roseGarden(int[] arr, int k, int m) {  
 long val = (long) m \* k;  
 int n = arr.length;  
 if (val > n) return -1;  
 int mini = Integer.*MAX\_VALUE*, maxi = Integer.*MIN\_VALUE*;  
 for (int i = 0; i < n; i++)  
 {  
 mini = Math.*min*(mini, arr[i]);  
 maxi = Math.*max*(maxi, arr[i]);  
 }  
 for (int i = mini; i <= maxi; i++)  
 {  
 if (*possible*(arr, i, m, k))  
 return i;  
 }  
 return -1;  
}

**Time Complexity**: O((max(arr[])-min(arr[])+1) \* N), where {max(arr[]) -> maximum element of the array, min(arr[]) -> minimum element of the array, N = size of the array}.

Reason: We are running a loop to check our answers that are in the range of [min(arr[]), max(arr[])]. For every possible answer, we will call the possible() function. Inside the possible() function, we are traversing the entire array, which results in O(N).

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

**Optimal Approach:**

Now, we are not given any sorted array on which we can apply binary search. But, if we observe closely, we can notice that our answer space i.e. [mini(arr[]), max(arr[])] is sorted. So, we will apply binary search on the answer space.

public int minDays(int[] bloomDay, int m, int k)  
{  
 if(bloomDay.length<m\*k)  
 {  
 return -1;  
 }  
 int result = -1;  
 int low = bloomDay[0];  
 int high = bloomDay[0];  
 for(int i=1;i<bloomDay.length;i++)  
 {  
 low = Math.*min*(bloomDay[i],low);  
 high = Math.*max*(bloomDay[i],high);  
 }  
 while(low<=high)  
 {  
 int mid = (low+high)/2;  
 int ans = check(bloomDay,mid,m,k);  
 if(ans>=m)  
 {  
 result = mid;  
 high = mid - 1;  
 }  
 else if(ans<m)  
 {  
 low = mid + 1;  
 }  
  
 }  
 return result;  
}  
public int check(int[] bloomDay,int mid,int m,int k)  
{  
 int bouquet = 0;  
 int cnt = 0;  
 for(int i=0;i<bloomDay.length;i++)  
 {  
 if(bloomDay[i]<=mid)  
 {  
 cnt++;  
 }  
 else  
 {  
 bouquet += cnt/k;  
 cnt = 0;  
 }  
 }  
 bouquet += cnt/k;  
 return bouquet;  
}

**Time Complexity**: O(log(max(arr[])-min(arr[])+1) \* N), where {max(arr[]) -> maximum element of the array, min(arr[]) -> minimum element of the array, N = size of the array}.

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

**6.** **Capacity to Ship Packages within D Days  
Example 1:**

Input Format:

N = 5, weights[] = {5,4,5,2,3,4,5,6}, d = 5

Result:

9

Explanation:

If the ship capacity is 9, the shipment will be done in the following manner:

Day Weights Total

1 - 5, 4 - 9

2 - 5, 2 - 7

3 - 3, 4 - 7

4 - 5 - 5

5 - 6 - 6

So, the least capacity should be 9.

**Example 2:**

Input Format:

N = 10, weights[] = {1,2,3,4,5,6,7,8,9,10}, d = 1

Result:

55

Explanation:

We have to ship all the goods in a single day. So, the weight capacity should be the summation of all the weights i.e. 55.

Observation:

Minimum ship capacity: The minimum ship capacity should be the maximum value in the given array. Let’s understand using an example. Assume the given weights array is {1, 2, 3, 4, 5, 6, 7, 8, 9, 10} and the ship capacity is 8. Now in the question, it is clearly stated that the loaded weights in the ship must not exceed the maximum weight capacity of the ship. For this constraint, we can never ship the weights 9 and 10, if the ship capacity is 8. That is why, in order to ship all the weights, the minimum ship capacity should be equal to the maximum of the weights array i.e. nax(weights[]).

Maximum capacity: If the ship capacity is equal to the sum of all the weights, we can ship all goods within a single day. Any capacity greater than this will yield the same result. So, the maximum capacity will be the summation of all the weights i.e. sum(weights[]).

From the observations, it is clear that our answer lies in the range

[max(weights[]), sum(weights[])].

How to calculate the number of days required to ship all the weights for a certain ship capacity:

In order to calculate this, we will write a function findDays(). This function accepts the weights array and a capacity as parameters and returns the number of days required for that particular capacity. The steps will be the following:

findDays(weights[], cap):

We will declare to variables i.e. ‘days’(representing the required days) and ‘load’ (representing the loaded weights in the ship). As we are on the first day, ‘days’ should be initialized with 1 and ‘load’ should be initialized with 0.

Next, we will use a loop(say i) to iterate over the weights. For each weight, weights[i], we will check the following:

If load+weights[i] > cap: If upon adding current weight with load exceeds the ship capacity, we will move on to the next day(i.e. day = day+1) and then load the current weight(i.e. Set load to weights[i], load = weights[i]).

Otherwise, We will just add the current weight to the load(i.e. load = load+weights[i]).

Finally, we will return ‘days’ which represents the number of days required.

**Brute – Force Approach:**

The extremely naive approach is to check all possible capacities from max(weights[]) to sum(weights[]). The minimum number for which the required days <= d value, will be our answer.

public static int findDays(int[] weights, int cap) {  
 int days = 1; //First day.  
 int load = 0;  
 int n = weights.length; //size of array.  
 for (int i = 0; i < n; i++) {  
 if (load + weights[i] > cap) {  
 days += 1; //move to next day  
 load = weights[i]; //load the weight.  
 } else {  
 //load the weight on the same day.  
 load += weights[i];  
 }  
 }  
 return days;  
}  
  
public static int leastWeightCapacity(int[] weights, int d) {  
 //Find the maximum and the summation:  
 int maxi = Integer.*MIN\_VALUE*, sum = 0;  
 for (int i = 0; i < weights.length; i++) {  
 sum += weights[i];  
 maxi = Math.*max*(maxi, weights[i]);  
 }  
  
 for (int i = maxi; i <= sum; i++) {  
 if (*findDays*(weights, i) <= d) {  
 return i;  
 }  
 }  
 //dummy return statement:  
 return -1;  
}

**Time Complexity:** O(N \* (sum(weights[]) - max(weights[]) + 1)), where sum(weights[]) = summation of all the weights, max(weights[]) = maximum of all the weights, N = size of the weights array.

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

**Optimal Approach:**

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.

Now, we are not given any sorted array on which we can apply binary search. Upon closer observation, we can recognize that our answer space, represented as [max(weights[]), sum(weights[])], 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.

public static int findDays(int[] weights, int cap) {  
 int days = 1; //First day.  
 int load = 0;  
 int n = weights.length; //size of array.  
 for (int i = 0; i < n; i++) {  
 if (load + weights[i] > cap) {  
 days += 1; //move to next day  
 load = weights[i]; //load the weight.  
 } else {  
 //load the weight on the same day.  
 load += weights[i];  
 }  
 }  
 return days;  
}  
  
public static int leastWeightCapacity(int[] weights, int d) {  
 //Find the maximum and the summation:  
 int low = Integer.*MIN\_VALUE*, high = 0;  
 for (int i = 0; i < weights.length; i++) {  
 high += weights[i];  
 low = Math.*max*(low, weights[i]);  
 }  
  
 while (low <= high) {  
 int mid = (low + high) / 2;  
 int numberOfDays = *findDays*(weights, mid);  
 if (numberOfDays <= d) {  
 //eliminate right half  
 high = mid - 1;  
 } else {  
 //eliminate left half  
 low = mid + 1;  
 }  
 }  
 return low;  
}

**Time Complexity**: O(N \* log(sum(weights[]) - max(weights[]) + 1)), where sum(weights[]) = summation of all the weights, max(weights[]) = maximum of all the weights, N = size of the weights array.

Reason: We are applying binary search on the range [max(weights[]), sum(weights[])]. For every possible answer ‘mid’, we are calling findDays() function. Now, inside the findDays() 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.

**7.** **Kth Missing Positive Number   
Example 1:**

Input Format:

vec[]={4,7,9,10}, k = 1

Result:

1

Explanation:

The missing numbers are 1, 2, 3, 5, 6, 8, 11, 12, ……, and so on. Since 'k' is 1, the first missing element is 1.

**Example 2:**

Input Format:

vec[]={4,7,9,10}, k = 4

Result:

5

Explanation:

The missing numbers are 1, 2, 3, 5, 6, 8, 11, 12, ……, and so on. Since 'k' is 4, the fourth missing element is 5.

**Brute – Force Approach:**

LINEAR SEARCH

public int findKthPositive(int[] arr, int k)  
{  
 if(arr[arr.length-1]==arr.length)  
 {  
 return arr.length+k;  
 }  
 if(arr[0]>1 && k<arr[0])  
 {  
 return k;  
 }  
 int index = 0;  
 for(int i=0;i<arr.length;i++)  
 {  
 if(arr[i]-(i+1) >= k)  
 {  
 break;  
 }  
 else  
 {  
 index = i;  
 }  
 }  
 return arr[index] + (k-(arr[index]-(index+1)));  
}

**Time Complexity**: O(N), N = size of the given array.

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

**Optimal Approach:**

public int findKthPositive(int[] arr, int k)  
{  
 if(arr[arr.length-1]==arr.length)  
 {  
 return arr.length+k;  
 }  
 if(arr[0]>1 && k<arr[0])  
 {  
 return k;  
 }  
 int low = 0;  
 int high = arr.length - 1;  
 int first = 0;  
 int index1 = -1;  
 while(low <= high)  
 {  
 int mid = (low + high)/2;  
 int val = arr[mid] - (mid+1);  
 if(val < k)  
 {  
 first = val;  
 index1 = mid;  
 low = mid + 1;  
 }  
 else  
 {  
 high = mid - 1;  
 }  
 }  
 return arr[index1]+(k-first);  
}

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.

We cannot apply binary search on the answer space here as we cannot assure which missing number has the possibility of being the kth missing number. That is why, we will do something different here. We will try to find the closest neighbors (i.e. Present in the array) for the kth missing number by counting the number of missing numbers for each element in the given array.

Let’s understand it using an example. Assume the given array is {2, 3, 4, 7, 11}. Now, if no numbers were missing the given array would look like {1, 2, 3, 4, 5}. Comparing these 2 arrays, we can conclude the following:

Up to index 0: Only 1 number i.e. 1 is missing in the given array.

Up to index 1: Only 1 number i.e. 1 is missing in the given array.

Up to index 2: Only 1 number i.e. 1 is missing in the given array.

Up to index 3: 3 numbers i.e. 1, 5, and 6 are missing.

Up to index 4: 6 numbers i.e. 1, 5, 6, 8, 9, and 10 are missing.

For a given value of k as 5, we can determine that the answer falls within the range of 7 to 11. Since there are only 3 missing numbers up to index 3, the 5th missing number cannot be before vec[3], which is 7. Therefore, it must be located somewhere to the right of 7. Our actual answer i.e. 9 also supports this theory. So, by following this process we can find the closest neighbors (i.e. Present in the array) for the kth missing number. In our example, the closest neighbors of the 5th missing number are 7 and 11.

How to calculate the number of missing numbers for any index i?

From the above example, we can derive a formula to find the number of missing numbers before any array index, i. The formula is

Number of missing numbers up to index i = vec[i] - (i+1).

The given array, vec, is currently containing the number vec[i] whereas it should contain (i+1) if no numbers were missing. The difference between the current and the ideal element will give the result.

How to apply Binary Search?

We will apply binary search on the indices of the given array. For each index, we will calculate the number of missing numbers and based on it, we will try to eliminate the halves.

How we will get the answer after all these steps?

After completing the binary search on the indices, the pointer high will point to the closest neighbor(present in the array) that is smaller than the kth missing number.

So, in the given array, the preceding neighbor of the kth missing number is vec[high].

Now, we know, up to index ‘high’,

the number of missing numbers = vec[high] - (high+1).

But we want to go further and find the kth number. To extend our objective, we aim to find the kth number in the sequence. In order to determine the number of additional missing values required to reach the kth position, we can calculate this as

more\_missing\_numbers = k - (vec[high] - (high+1)).

Now, we will simply add more\_missing\_numbers to the preceding neighbor i.e. vec[high] to get the kth missing number.

kth missing number = vec[high] + k - (vec[high] - (high+1))

= vec[high] + k - vec[high] + high + 1

= k + high + 1.  
  
**Time Complexity:** O(logN), N = size of the given array.

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

**8.** **Aggressive Cows**

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):

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.

First, we will place the first cow in the very first stall. So, we will set ‘cntCows’ to 1 and ‘last’ to stall[0].

Then, using a loop we will start iterating the array from index 1. Inside the loop, we will do the following:

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.

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.

If we are outside the loop, we cannot place k cows with a minimum distance of ‘dist’. So, we will return false.

<https://leetcode.com/problems/magnetic-force-between-two-balls/description/>

**Brute – Force Approach:**

public static boolean canWePlace(int[] stalls, int dist, int cows) {  
 int n = stalls.length; //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;  
}  
public static int aggressiveCows(int[] stalls, int k) {  
 int n = stalls.length; //size of array  
 //sort the stalls[]:  
 Arrays.*sort*(stalls);  
  
 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;  
}

**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.

**Optimal Approach:**

public int maxDistance(int[] position, int m)  
{  
 Arrays.*sort*(position);  
 int low = 1;  
 int high = position[position.length-1]-position[0];  
 while(low <= high)  
 {  
 int mid = (low+high)/2;  
 if(canPlace(position,mid,m))  
 {  
 low = mid + 1;  
 }  
 else  
 {  
 high = mid - 1;  
 }  
 }  
 return high;  
}  
  
public boolean canPlace(int[] position,int minDis,int cows)  
{  
 int cnt = 1;  
 int prev = position[0];  
 for(int i=1;i<position.length;i++)  
 {  
 if(position[i]-prev >= minDis)  
 {  
 cnt+=1;  
 prev = position[i];  
 }  
 }  
 if(cnt>=cows)  
 return true;  
 else  
 return false;  
}

**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.

**9.** **Allocate Minimum Number of Pages**Problem Statement: Given an array ‘arr of integer numbers, ‘ar[i]’ represents the number of pages in the ‘i-th’ book. There are a ‘m’ number of students, and the task is to allocate all the books to the students.

Allocate books in such a way that:

Each student gets at least one book.

Each book should be allocated to only one student.

Book allocation should be in a contiguous manner.

You have to allocate the book to ‘m’ students such that the maximum number of pages assigned to a student is minimum. If the allocation of books is not possible. return -1

**Example 1:**

Input Format:

n = 4, m = 2, arr[] = {12, 34, 67, 90}

Result:

113

Explanation:

The allocation of books will be 12, 34, 67 | 90. One student will get the first 3 books and the other will get the last one.

**Example 2:**

Input Format:

n = 5, m = 4, arr[] = {25, 46, 28, 49, 24}

Result:

71

Explanation: The allocation of books will be 25, 46 | 28 | 49 | 24.

We can allocate books in several ways but it is clearly said in the question that we have to allocate the books in such a way that the maximum number of pages received by a student should be minimum.

Assume the given array is {25 46 28 49 24} and number of students, M = 4. Now, we can allocate these books in different ways. Some of them are the following:

25 | 46 | 28 | 49, 24 → Maximum no. of pages a student receive = 73

25 | 46 | 28, 49 | 24 → Maximum no. of pages a student receive = 77

25 | 46, 28 | 49 | 24 → Maximum no. of pages a student receive = 74

25, 46 | 28 | 49 | 24 → Maximum no. of pages a student receive = 71

From the above allocations, we can clearly observe that the minimum possible maximum number of pages is 71.

When it is impossible to allocate books:

When the number of books is lesser than the number of students, we cannot allocate books to all the students even if we give only a single book to each student. So, if m > n, we should return -1.

Observations:

Minimum possible answer: We will get the minimum answer when we give n books of the array to n students(i.e. Each student will receive 1 book). Now, in this case, the maximum number of pages will be the maximum element in the array. So, the minimum possible answer is max(arr[]).

Maximum possible answer: We will get the maximum answer when we give all n books to a single student. The maximum no. of pages he/she will receive is the summation of array elements i.e. sum(arr[]). So, the maximum possible answer is sum(arr[]).

From the observations, it is clear that our answer lies in the range [max(arr[]), sum(arr[])].

How to calculate the number of students to whom we can allocate the books if one can receive at most ‘pages’ number of pages:

In order to calculate the number of students we will write a function, countStudents(). This function will take the array and ‘pages’ as parameters and return the number of students to whom we can allocate the books.

countStudents(arr[], pages):

We will first declare two variables i.e. ‘students’(stores the no. of students), and pagesStudent(stores the number of pages of a student). As we are starting with the first student, ‘students’ should be initialized with 1.

We will start traversing the given array.

If pagesStudent + arr[i] <= pages: If upon adding the pages with the existing number of pages does not exceed the limit, we can allocate this i-th book to the current student.

Otherwise, we will move to the next student(i.e. students += 1 ) and allocate the book.

Finally, we will return the value of ‘students’

**Brute – Force Approach:**

public static int countStudents(ArrayList<Integer> arr, int pages) {  
 int n = arr.size(); // size of array  
 int students = 1;  
 long pagesStudent = 0;  
 for (int i = 0; i < n; i++) {  
 if (pagesStudent + arr.get(i) <= pages) {  
 // add pages to current student  
 pagesStudent += arr.get(i);  
 } else {  
 // add pages to next student  
 students++;  
 pagesStudent = arr.get(i);  
 }  
 }  
 return students;  
}  
  
public static int findPages(ArrayList<Integer> arr, int n, int m) {  
 // book allocation impossible  
 if (m > n)  
 return -1;  
  
 int low = Collections.*max*(arr);  
 int high = arr.stream().mapToInt(Integer::intValue).sum();  
  
 for (int pages = low; pages <= high; pages++) {  
 if (*countStudents*(arr, pages) == m) {  
 return pages;  
 }  
 }  
 return low;  
}

**Time Complexity:** O(N \* (sum(arr[])-max(arr[])+1)), where N = size of the array, sum(arr[]) = sum of all array elements, max(arr[]) = maximum of all array elements.

Reason: We are using a loop from max(arr[]) to sum(arr[]) to check all possible numbers of pages. Inside the loop, we are calling the countStudents() function for each number. Now, inside the countStudents() 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.

**Optimal Approach:**

public static int findPages(ArrayList<Integer> arr, int n, int m)  
{  
 int low = arr.get(0);  
 int high = 0;  
 for(int i=0;i<arr.size();i++)  
 {  
 if(arr.get(i)>low)  
 {  
 low = arr.get(i);  
 }  
 high += arr.get(i);  
 }  
 System.*out*.println("high: "+high);  
 while(low<=high)  
 {  
 int mid = (low+high)/2;  
 System.*out*.println("mid: "+mid);  
 if(*allocBook*(arr,mid,m) <= m)  
 {  
 high = mid - 1;  
 }  
 else  
 {  
 low = mid + 1;  
 }  
 }  
 return low;  
}  
  
public static int allocBook(ArrayList<Integer> arr, int maximum, int students)  
{  
 int student = 1;  
 int ans = 0;  
 for(int i=0;i<arr.size();i++)  
 {  
 if(ans+arr.get(i) <= maximum)  
 {  
 ans+=arr.get(i);  
 }  
 else  
 {  
 ans = arr.get(i);  
 student+=1;  
 }  
  
 }  
 return student;  
}

**Time Complexity:** O(N \* log(sum(arr[])-max(arr[])+1)), where N = size of the array, sum(arr[]) = sum of all array elements, max(arr[]) = maximum of all array elements.

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

**10.** **Split Array Largest sum**

Same as above problem.

**11. Painter's Partition Problem**

Same as above problem.

**12.** **Minimise Maximum Distance between Gas Stations**

Problem Statement: You are given a sorted array ‘arr’ of length ‘n’, which contains positive integer positions of ‘n’ gas stations on the X-axis. You are also given an integer ‘k’. You have to place 'k' new gas stations on the X-axis. You can place them anywhere on the non-negative side of the X-axis, even on non-integer positions. Let 'dist' be the maximum value of the distance between adjacent gas stations after adding k new gas stations.

Find the minimum value of ‘dist’.

Note: Answers within 10^-6 of the actual answer will be accepted. For example, if the actual answer is 0.65421678124, it is okay to return 0.654216. Our answer will be accepted if that is the same as the actual answer up to the 6th decimal place.

**Example 1:**

Input Format:

N = 5, arr[] = {1,2,3,4,5}, k = 4

Result:

0.5

Explanation:

One of the possible ways to place 4 gas stations is {1,1.5,2,2.5,3,3.5,4,4.5,5}. Thus the maximum difference between adjacent gas stations is 0.5. Hence, the value of ‘dist’ is 0.5. It can be shown that there is no possible way to add 4 gas stations in such a way that the value of ‘dist’ is lower than this.

**Example 2:**

Input Format:

N = 10, arr[] = {1,2,3,4,5,6,7,8,9,10}, k = 1

Result:

1

Explanation:

One of the possible ways to place 1 gas station is {1,1.5,2,3,4,5,6,7,8,9,10}. Thus the maximum difference between adjacent gas stations is still 1. Hence, the value of ‘dist’ is 1. It can be shown that there is no possible way to add 1 gas station in such a way that the value of ‘dist’ is lower than this.

Let’s understand how to place the new gas stations so that the maximum distance between two consecutive gas stations is reduced.

Let’s consider a small example like this: given gas stations = {1, 7} and k = 2.

Observation: A possible arrangement for placing 2 gas stations is as follows: {1, 7, 8, 9}. In this arrangement, the new gas stations are positioned after the last existing one. Prior to adding the new stations, the maximum distance between stations was 6 (i.e. the distance between 1 and 7). Even after placing the 2 new stations, the maximum distance remains unchanged at 6.

Conclusions:

From the above observation, we can conclude that placing new gas stations before the first existing station or after the last existing station will make no difference to the maximum distance between two consecutive stations.

So, in order to minimize the maximum distance we have to place the new gas stations in between the existing stations.

How to place the gas stations in between so that the maximum distance is minimized:

Until now we have figured out that we have to place the gas stations in between the existing ones. But we have to place them in such a way that the maximum distance between two consecutive stations is the minimum possible.

Let’s understand this considering the previous example. Given gas stations = {1, 7} and k = 2.

If we place the gas stations as follows: {1, 2, 6, 7}, the maximum distance will be 4(i.e. 6-2 = 4). But if we place them like this: {1, 3, 5, 7}, the maximum distance boils down to 2. It can be proved that we cannot make the maximum distance lesser than 2.

To minimize the maximum distance between gas stations, we need to insert new stations with equal spacing. If we have to add 'k' gas stations within a section of length 'section\_length', each station should be placed at a distance of

(section\_length / (k + 1)) from one another.

This way, we maintain a uniform spacing between consecutive gas stations.

For example, the gas stations are = {1, 7} and k = 2. Here, the ‘dist’ is = (7-1) = 6. So, the space between two gas stations will be dis / (k+1) = 6 / (2+1) = 2. The placements will be as follows: {1, 3, 5, 7}.

**Brute – Force Approach:**

We are given n gas stations. Between them, there are n-1 sections where we may insert the new stations to reduce the distance. So, we will create an array of size n-1 and each of its indexes will represent the respective sections between the given gas stations.

In each iteration, we will identify the index 'i' where the distance (arr[i+1] - arr[i]) is the maximum. Then, we will insert new stations into that section to reduce that maximum distance. The number of stations inserted in each section will be tracked using the previously declared array of size n-1.

Finally, after placing all the stations we will find the maximum distance between two consecutive stations. To calculate the distance using the previously discussed formula, we will just do as follows for each section:

distance = section\_length / (number\_of\_stations\_ inserted+1)

Among all the values of ‘distance’, the maximum one will be our answer.

Algorithm:

First, we will declare an array ‘howMany[]’ of size n-1, to keep track of the number of placed gas stations.

Next, using a loop we will pick k gas stations one at a time.

Then, using another loop, we will find the index 'i' where the distance (arr[i+1] - arr[i]) is the maximum and insert the current gas station between arr[i] and arr[i+1] (i.e. howMany[i]++).

Finally, after placing all the new stations, we will find the distance between two consecutive gas stations. For a particular section,

distance = section\_length / (number\_of\_stations\_ inserted+1)

= (arr[i+1]-arr[i]) / (howMany[i]+1)

Among all the distances, the maximum one will be the answer.  
  
public static double minimiseMaxDistance(int[] arr, int k) {  
 int n = arr.length; //size of array.  
 int[] howMany = new int[n - 1];  
  
 //Pick and place k gas stations:  
 for (int gasStations = 1; gasStations <= k; gasStations++) {  
 //Find the maximum section  
 //and insert the gas station:  
 double maxSection = -1;  
 int maxInd = -1;  
 for (int i = 0; i < n - 1; i++) {  
 double diff = arr[i + 1] - arr[i];  
 double sectionLength =  
 diff / (double)(howMany[i] + 1);  
 if (sectionLength > maxSection) {  
 maxSection = sectionLength;  
 maxInd = i;  
 }  
 }  
 //insert the current gas station:  
 howMany[maxInd]++;  
 }  
  
 //Find the maximum distance i.e. the answer:  
 double maxAns = -1;  
 for (int i = 0; i < n - 1; i++) {  
 double diff = arr[i + 1] - arr[i];  
 double sectionLength =  
 diff / (double)(howMany[i] + 1);  
 maxAns = Math.*max*(maxAns, sectionLength);  
 }  
 return maxAns;  
}

**Time Complexity**: O(k\*n) + O(n), n = size of the given array, k = no. of gas stations to be placed.

Reason: O(k\*n) to insert k gas stations between the existing stations with maximum distance. Another O(n) for finding the answer i.e. the maximum distance.

**Space Complexity:** O(n-1) as we are using an array to keep track of placed gas stations.

**Better Approach:**

In the previous approach, for every gas station, we were finding the index i for which the distance between arr[i+1] and arr[i] is maximum. After that, our job was to place the gas station. Instead of using a loop to find the maximum distance, we can simply use the heap data structure i.e. the priority queue.

Priority Queue: Priority queue internally uses the heap data structure. In the max heap implementation, the first element is always the greatest of the elements it contains and the rest elements are in decreasing order.

Note: Please refer to the article: priority\_queue in C++ STL to know more about the data structure.

Thus using a priority queue, we can optimize the search for the maximum distance. We will use the max heap implementation and the elements will be in the form of pairs i.e. <distance, index> as we want the indices sorted based on the distance. As we are using max heap the maximum distance will always be the first element.

Algorithm:

First, we will declare an array ‘howMany[]’ of size n-1, to keep track of the number of placed gas stations and a priority queue that uses max heap.

We will insert the first n-1 indices with the respective distance value, arrr[i+1]-arr[i] for every index.

Next, using a loop we will pick k gas stations one at a time.

Then we will pick the first element of the priority queue as this is the element with the maximum distance. Let’s call the index ‘secInd’.

Now we will place the current gas station at ‘secInd’(howMany[secInd]++) and calculate the new section length,

new\_section\_length = initial\_section\_length / (number\_of\_stations\_ inserted+1)

= (arr[secInd+1] - arr[secInd]) / (howMany[i] + 1)

After that, we will again insert the pair <new\_section\_length, secInd> into the priority queue for further consideration.

After performing all the steps for k gas stations, the distance at the top of the priority queue will be the answer as we want the maximum distance.

public static double minimiseMaxDistance(int[] arr, int k) {  
 int n = arr.length; // size of array.  
 int[] howMany = new int[n - 1];  
 PriorityQueue<Pair> pq = new PriorityQueue<>((a, b) -> Double.*compare*(b.first, a.first));  
  
 // insert the first n-1 elements into pq  
 // with respective distance values:  
 for (int i = 0; i < n - 1; i++) {  
 pq.add(new Pair(arr[i + 1] - arr[i], i));  
 }  
  
 // Pick and place k gas stations:  
 for (int gasStations = 1; gasStations <= k; gasStations++) {  
 // Find the maximum section  
 // and insert the gas station:  
 Pair tp = pq.poll();  
 int secInd = tp.second;  
  
 // insert the current gas station:  
 howMany[secInd]++;  
  
 double inidiff = arr[secInd + 1] - arr[secInd];  
 double newSecLen = inidiff / (double) (howMany[secInd] + 1);  
 pq.add(new Pair(newSecLen, secInd));  
 }  
  
 return pq.peek().first;  
}

**Time Complexity:** O(nlogn + klogn), n = size of the given array, k = no. of gas stations to be placed.

Reason: Insert operation of priority queue takes logn time complexity. O(nlogn) for inserting all the indices with distance values and O(klogn) for placing the gas stations.

**Space Complexity**: O(n-1)+O(n-1)

Reason: The first O(n-1) is for the array to keep track of placed gas stations and the second one is for the priority queue.

**Optimal 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.

Observations:

Minimum possible answer: We will get the minimum answer when we place all the gas stations in a single location. Now, in this case, the maximum distance will be 0.

Maximum possible answer: We will not place stations before the first or after the last station rather we will place stations in between the existing stations. So, the maximum possible answer is the maximum distance between two consecutive existing stations.

From the observations, it is clear that our answer lies in the range [0, max(dist)].

Upon closer observation, we can recognize that our answer space 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.

Changes in the binary search algorithm to apply it to the decimal answer space:

The traditional binary search algorithm used for integer answer space won't be effective in this case. As our answer space consists of decimal numbers, we need to adjust some conditions to tailor the algorithm to this specific context. The changes are the following:

while(low <= high): The condition 'while(low <= high)' inside the 'while' loop won't work for decimal answers, and using it might lead to a TLE error. To avoid this, we can modify the condition to 'while(high - low > 10^(-6))'. This means we will only check numbers up to the 6th decimal place. Any differences beyond this decimal precision won't be considered, as the question explicitly accepts answers within 10^-6 of the actual answer.

low = mid+1: We have used this operation to eliminate the left half. But if we apply the same here, we might ignore several decimal numbers and possibly our actual answer. So, we will use this: low = mid.

high = mid-1: Similarly, We have used this operation to eliminate the right half. But if we apply the same here, we might ignore several decimal numbers and possibly the actual answer. So, we will use this: high = mid.

We are applying binary search on the answer i.e. the possible values of distances. So, we have to figure out a way to check the number of gas stations we can place for a particular value of distance.

How to check the number of gas stations we can place with a particular distance ‘dist’:

In order to find out the number of gas stations we will use the following function:

numberOfGasStationsRequired(dist, arr[]):

We will use a loop(say i) that will run from 1 to n.

For each section between i and i-1, we will do the following:

No. of stations = (arr[i]-arr[i-1]) / dist

Let's keep in mind a crucial edge case: if the section\_length (arr[i] - arr[i-1]) is completely divisible by 'dist', the actual number of stations required will be one less than what we calculate.

if (arr[i]-arr[i-1] == (No. of stations\*dist): No. of stations -= 1.

Now, we will add the no. of stations regarding all the sections and the total will be the answer.

public static int numberOfGasStationsRequired(double dist, int[] arr) {  
 int n = arr.length; // size of the array  
 int cnt = 0;  
 for (int i = 1; i < n; i++) {  
 int numberInBetween = (int)((arr[i] - arr[i - 1]) / dist);  
 if ((arr[i] - arr[i - 1]) == (dist \* numberInBetween)) {  
 numberInBetween--;  
 }  
 cnt += numberInBetween;  
 }  
 return cnt;  
}  
  
public static double minimiseMaxDistance(int[] arr, int k) {  
 int n = arr.length; // size of the array  
 double low = 0;  
 double high = 0;  
  
 //Find the maximum distance:  
 for (int i = 0; i < n - 1; i++) {  
 high = Math.*max*(high, (double)(arr[i + 1] - arr[i]));  
 }  
  
 //Apply Binary search:  
 double diff = 1e-6 ;  
 while (high - low > diff) {  
 double mid = (low + high) / (2.0);  
 int cnt = *numberOfGasStationsRequired*(mid, arr);  
 if (cnt > k) {  
 low = mid;  
 } else {  
 high = mid;  
 }  
 }  
 return high;  
}

**Time Complexity:** O(n\*log(Len)) + O(n), n = size of the given array, Len = length of the answer space.

Reason: We are applying binary search on the answer space. For every possible answer, we are calling the function numberOfGasStationsRequired() that takes O(n) time complexity. And another O(n) for finding the maximum distance initially.

**Space Complexity:** O(1) as we are using no extra space to solve this problem.

**13.** **Median of 2 sorted arrays**

**Example 1:**

Input Format:

n1 = 3, arr1[] = {2,4,6}, n2 = 3, arr2[] = {1,3,5}

Result:

3.5

Explanation:

The array after merging 'a' and 'b' will be { 1, 2, 3, 4, 5, 6 }. As the length of the merged list is even, the median is the average of the two middle elements. Here two medians are 3 and 4. So the median will be the average of 3 and 4, which is 3.5.

**Example 2:**

Input Format:

n1 = 3, arr1[] = {2,4,6}, n2 = 2, arr2[] = {1,3}

Result:

3

Explanation:

The array after merging 'a' and 'b' will be { 1, 2, 3, 4, 6 }. The median is simply 3.

**Brute – Force Approach:**

We will call the required indices as ind2 = (n1+n2)/2 and ind1 = ((n1+n2)/2)-1. Now we will declare the counter called ‘cnt’ and initialize it with 0.

Now, as usual, we will take two pointers i and j, where i points to the first element of arr1[] and j points to the first element of arr2[].

Next, using a while loop( while(i < n1 && j < n2)), we will select two elements i.e. arr1[i] and arr2[j], and consider the smallest one among the two. Then, we will increase that specific pointer by 1.

In addition to that, in each iteration, we will check if the counter ‘cnt’ hits the indices ind1 or ind2. when 'cnt' reaches either index ind1 or ind2, we will store that particular element. We will also increase the ‘cnt’ by 1 every time regardless of matching the conditions.

If arr1[i] < arr2[j]: Check ‘cnt’ to perform necessary operations and increase i and ‘cnt’ by 1.

Otherwise: Check ‘cnt’ to perform necessary operations and increase j and ‘cnt’ by 1.

After that, the left-out elements from both arrays will be copied as it is into the third array. While copying we will again check the above-said conditions for the counter, ‘cnt’ and increase it by 1.

Now, let’s call the elements at the required indices as ind1el(at ind1) and ind2el(at ind2):

If the total length i.e. (n1+n2) is even: The median is the average of the two middle elements. median = (ind1el + ind2el) / 2.0.

If the total length i.e. (n1+n2) is odd: median = ind2el.

Finally, we will return the value of the median.  
  
public static double median(int[] a, int[] b) {  
 // Size of two given arrays  
 int n1 = a.length;  
 int n2 = b.length;  
  
 int n = n1 + n2; //total size  
 //required indices:  
 int ind2 = n / 2;  
 int ind1 = ind2 - 1;  
 int cnt = 0;  
 int ind1el = -1, ind2el = -1;  
  
 //apply the merge step:  
 int i = 0, j = 0;  
 while (i < n1 && j < n2) {  
 if (a[i] < b[j]) {  
 if (cnt == ind1) ind1el = a[i];  
 if (cnt == ind2) ind2el = a[i];  
 cnt++;  
 i++;  
 } else {  
 if (cnt == ind1) ind1el = b[j];  
 if (cnt == ind2) ind2el = b[j];  
 cnt++;  
 j++;  
 }  
 }  
  
 //copy the left-out elements:  
 while (i < n1) {  
 if (cnt == ind1) ind1el = a[i];  
 if (cnt == ind2) ind2el = a[i];  
 cnt++;  
 i++;  
 }  
 while (j < n2) {  
 if (cnt == ind1) ind1el = b[j];  
 if (cnt == ind2) ind2el = b[j];  
 cnt++;  
 j++;  
 }  
  
 //Find the median:  
 if (n % 2 == 1) {  
 return (double)ind2el;  
 }  
  
 return (double)((double)(ind1el + ind2el)) / 2.0;  
}

**Time Complexity:** O(n1+n2), where n1 and n2 are the sizes of the given arrays.

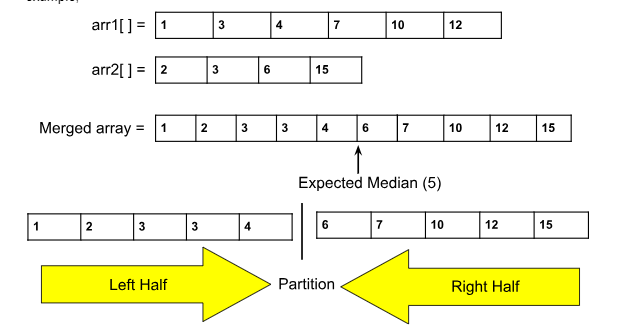
**Space Complexity:** O(1), as we are not using any extra space to solve this problem.

**Optimal Approach:**Now, let’s learn through the following observations how we can apply binary search to this problem. First, we will try to solve this problem where n1+n2 is even and then we will consider the odd scenario.

Observations:

Assume, n = n1+n2 i.e. the total length of the final merged array.

Median creates a partition on the final merged array: Upon closer observation, we can easily show that the median divides the final merged array into two halves. For example,

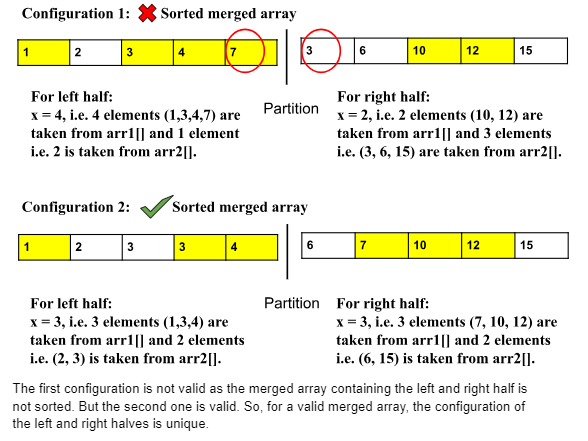


Characteristics of each half:

Each half contains (n/2) elements.

Each half also contains x elements from the first array i.e. arr1[] and (n/2)-x elements from the second array i.e. arr2[]. The value of x might be different for the two halves. For example, in the above array, the left half contains 3 elements from arr1[] and 2 elements from arr2[].

The unique configuration of halves: Considering different values of x, we can get different left and right halves(x = the number of elements taken from arr1[] for a particular half). Some different configurations for the above example are shown below:



How to solve the problem using the above observations:

Try to form the unique left half:

For a valid merged array, the configurations of the two halves are unique. So, we can try to form the halves with different values of x, where x = the number of elements taken from arr1[] for a particular half.

There's no need to construct both halves. Once we have the correct left half, the right half is automatically determined, consisting of the remaining elements not yet considered. Therefore, our focus will solely be on creating the unique left half.

How to form all configurations of the left half: We know that the left half will surely contain x elements from arr1[] and (n/2)-x elements from arr2[]. Here the only variable is x. The minimum possible value of x is 0 and the maximum possible value is n1(i.e. The length of the considered array).

For all the values,[0, n1] of x, we will try to form the left half and then we will check if that half’s configuration is valid.

Check if the formed left half is valid: For a valid left half, the merged array will always be sorted. So, if the merged array containing the formed left half is sorted, the formation is valid.

How to check if the merged array is sorted without forming the array:

In order to check we will consider 4 elements, i.e. l1, l2, r1, r2.

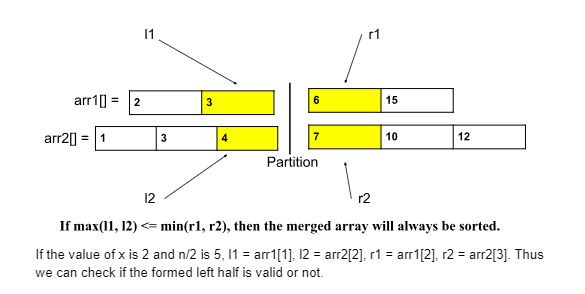
l1 = the maximum element belonging to arr1[] of the left half.

l2 = the maximum element belonging to arr2[] of the left half.

r1 = the minimum element belonging to arr1[] of the right half.

r1 = the minimum element belonging to arr2[] of the right half.

For example,



How to apply Binary search to form the left half:

We will check the formation of the left half for all possible values of x. Now, we know that the minimum possible value of x is 0 and the maximum is n1(i.e. The length of the considered array). Now the range is sorted. So, we will apply the binary search on the possible values of x i.e. [0, n1].

How to eliminate the halves based on the values of x: Binary search works by eliminating the halves in each step. Upon closer observation, we can eliminate the halves based on the following conditions:

If l1 > r2: This implies that we have considered more elements from arr1[] than necessary. So, we have to take less elements from arr1[] and more from arr2[]. In such a scenario, we should try smaller values of x. To achieve this, we will eliminate the right half (high = mid-1).

If l2 > r1: This implies that we have considered more elements from arr2[] than necessary. So, we have to take less elements from arr2[] and more from arr1[]. In such a scenario, we should try bigger values of x. To achieve this, we will eliminate the left half (low = mid+1).

Until now, we have learned how to use binary search but with the assumption that (n1+n2) is even. Let’s generalize this.

If (n1+n2) is odd: In the case of even, we have considered the length of the left half as

(n1+n2) / 2. In this case, that length will be (n1 + n2 + 1) / 2. This much change is enough to handle the case of odd. The rest of the things will be completely the same.

As in the code, division refers to integer division, this modified formula (n1+n2+1) / 2 will be valid for both cases of odd and even.

What will be the answer i.e. the median:

If l1 <= r2 && l2 <= r1: This condition assures that we have found the correct elements.

If (n1+n2) is odd: The median will be max(l1, l2).

Otherwise, median = (max(l1, l2) + min(r1, r2)) / 2.0

Note: We are applying binary search on the possible values of x i.e. [0, n1]. Here n1 is the length of arr1[]. Now, to further optimize it, we will consider the smaller array as arr1[]. So, the actual range will be [0, min(n1, n2)].

public double findMedianSortedArrays(int[] nums1, int[] nums2)  
{  
 int n1 = nums1.length;  
 int n2 = nums2.length;  
 if(n1>n2) return findMedianSortedArrays(nums2,nums1);  
 int low = 0;  
 int high = n1;  
 int left = (n1+n2+1)/2;  
 int n = n1+n2;  
 while(low <= high)  
 {  
 int mid1 = (low+high)/2;  
 int mid2 = left - mid1;  
 int l1 = Integer.*MIN\_VALUE*;  
 int l2 = Integer.*MIN\_VALUE*;  
 int r1 = Integer.*MAX\_VALUE*;  
 int r2 = Integer.*MAX\_VALUE*;  
 if(mid1-1 >= 0)l1 = nums1[mid1-1];  
 if(mid2-1 >= 0)l2 = nums2[mid2-1];  
 if(mid1 < n1)r1 = nums1[mid1];  
 if(mid2 < n2)r2 = nums2[mid2];  
 if(l1 <= r2 && l2 <= r1)  
 {  
 if(n%2 == 1)  
 return Math.*max*(l1,l2);  
 return (Math.*max*(l1,l2)+Math.*min*(r1,r2))/2.0;  
 }  
 else if(l2>r1)  
 {  
 low = mid1 + 1;  
 }  
 else  
 {  
 high = mid1 - 1;  
 }  
 }  
  
 return 0;  
}

**Time Complexity:** O(log(min(n1,n2))), where n1 and n2 are the sizes of two given arrays.

Reason: We are applying binary search on the range [0, min(n1, n2)].

**Space Complexity:** O(1) as no extra space is used.

**14.Kth element of 2 sorted arrays.**

Same as above

Slight changes is that

Left side only k values , so that we can return k easily

And low and high changes accordingly.

low = max(0,k-n2);

high = min(k,n1);

finally, the mystery of why low and high needs to be corrected for this solution?

Let's take an example, m = 3, n = 10, k = 12

low = 0, and high = 3

then mid1 = 1;

means we pick only one element from the first array,

and now the remaining elements need to be picked from the second array.

mid2 = (k - mid1) = 12 - 1 = 11 ????

but there are only 10 elements in the second array .

Hence we can't start our search when we pick no elements from the first array. So our low must be max(k - n, 0) [no of elements at least need to pick for 1st array] .

Similarly, for high, we have to reduce the search space such that it can handle low K values.