Strivers A2Z DSA Course/Sheet

takeuforward.org/strivers-a2z-dsa-course/strivers-a2z-dsa-course-sheet-2

June 20, 2022

A2Z DSA Sheet Overview ›

Note: The Series doesn't focus on any specific programming language. Instead, it emphasizes logic and uses pseudocode. The first two basic videos might use C++, but for Java tutorials, you can watch other YouTube videos. When tackling core problems of data structures and algorithms (DSA), the Series uses pseudocode that isn't tied to any particular programming language. However, you can find code examples in your preferred language in the notes and articles provided.

(03/455)
Marked for Revision: **0**

▶ **Step 1**: Learn the basics
(0/31)
▶ **Step 2**: Learn Important Sorting Techniques
(0/7)
▶ **Step 3**: Solve Problems on Arrays [Easy -> Medium -> Hard]
(0/40)
▼ **Step 4**: Binary Search [1D, 2D Arrays, Search Space]
(0/32)
▼ **Step 4.1**: BS on 1D Arrays

(0/13)

Action	Problem [Articles, Codes]	PL-1	Solution	PL-2	Notes	Company
	Binary Search to find X in sorted a…					
	Implement Lower Bound					
	Implement Upper Bound					
	Search Insert Position					
	Floor/Ceil in Sorted Array					
	Find the first or last occurrence o…					
	Count occurrences of a number in a …					
	Search in Rotated Sorted Array I					
	Search in Rotated Sorted Array II					
	Find minimum in Rotated Sorted Arra…					
	Find out how many times has an arra…					
	Single element in a Sorted Array					
	Find peak element					

▼ **Step 4.2**: BS on Answers

(0/14)

Action	Problem [Articles, Codes]	PL-1	Solution	PL-2	Notes	Company
	Find square root of a number in log…					
	Find the Nth root of a number using…					
	Koko Eating Bananas					

Action	Problem [Articles, Codes]	PL-1	Solution	PL-2	Notes	Company
	Minimum days to make M bouquets	E		E		
	Find the smallest Divisor	E		E		
	Capacity to Ship Packages within D …	E		E		
	Kth Missing Positive Number	E		E		
	Aggressive Cows	E		E		
	Book Allocation Problem	E		E		
	Split array - Largest Sum	E		E		
	Painter's partition	E				
	Minimize Max Distance to Gas Statio…	E		E		
	Median of 2 sorted arrays	E		E		
	Kth element of 2 sorted arrays	E				

▼ **Step 4.3**: BS on 2D Arrays
(0/5)

Action	Problem [Articles, Codes]	PL-1	Solution	PL-2	Notes	Company
	Find the row with maximum number of…	E				
	Search in a 2 D matrix	E		E		
	Search in a row and column wise sor…	E		E		
	Find Peak Element (2D Matrix)	E		E		
	Matrix Median	E				

▶ **Step 5**: Strings [Basic and Medium]
(0/15)

▶ **Step 6**: Learn LinkedList [Single LL, Double LL, Medium, Hard Problems]

(0/31)
▶ **Step 7**: Recursion [PatternWise]
(0/25)
▶ **Step 8**: Bit Manipulation [Concepts & Problems]
(0/18)
▶ **Step 9**: Stack and Queues [Learning, Pre-In-Post-fix, Monotonic Stack, Implementation]
(0/30)
▶ **Step 10**: Sliding Window & Two Pointer Combined Problems
(0/12)
▶ **Step 11**: Heaps [Learning, Medium, Hard Problems]
(0/17)
▶ **Step 12**: Greedy Algorithms [Easy, Medium/Hard]
(3/16)
▶ **Step 13**: Binary Trees [Traversals, Medium and Hard Problems]
(0/39)
▼ **Step 14**: Binary Search Trees [Concept and Problems]
(0/16)
▶ **Step 14.1**: Concepts
(0/3)
▶ **Step 14.2**: Practice Problems
(0/13)
▶ **Step 15**: Graphs [Concepts & Problems]
(0/54)
▶ **Step 16**: Dynamic Programming [Patterns and Problems]
(0/56)
▶ **Step 17**: Tries
(0/7)
▶ **Step 18**: Strings
(0/9)
▶ **Step 19**: Request from Striver
Frequently Asked Questions (FAQs)
Are these many questions enough for clearing any DSA round? ›
Is this a language specific course? ›
How do I get my doubts resolved? ›
What is the difference between Strivers SDE Sheet vs Strivers A2Z DSA Course? ›
Do I need to pay anything? ›

> Hurrah!! You are ready for your placement after some months of hard work! All the best, keep striving…
>
> *Striver*

Share the course/sheet with your friends, created with love for takeUforward fam!

If you find any mistakes in the sheet, it can be a wrong link as well, please fill out the google form here, our team will check it on a weekly basis, thanks.

Binary Search: Explained

takeuforward.org/data-structure/binary-search-explained

January 28, 2022

This is the very first article of the Binary Search series. Until now, we have learned the linear search algorithm. Now, in this article, we will discuss another search algorithm i.e. the Binary Search algorithm. The flow of this article will be the following:

- **A real-life example of Binary Search**
- **Coding problem example**
- **Iterative code implementation of Binary Search**
- **Recursive code implementation of Binary Search**
- **Time complexity**
- **Overflow case**

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.

Coding problem example:

Problem statement: You are given a sorted array of integers and a target, your task is to search for the target in the given array. Assume the given array does not contain any duplicate numbers.

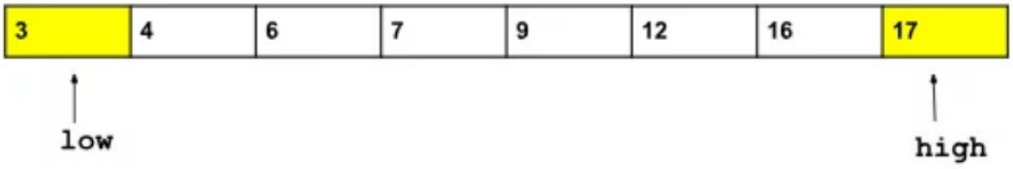
Let's say the given array is = {3, 4, 6, 7, 9, 12, 16, 17} and target = 6.

Solution:

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

Iterative Implementation Recursive Implementation

▼ 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.



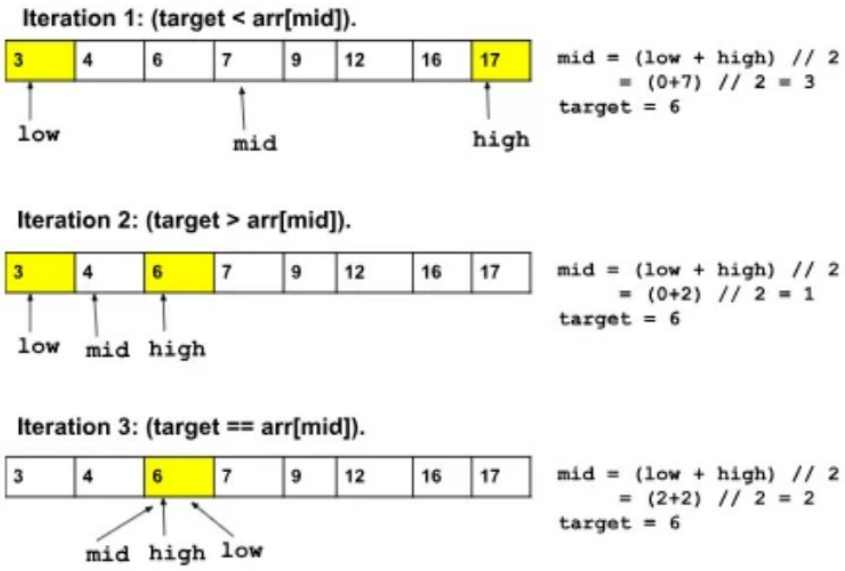
Search space: The entire area between the low and the high pointer(*including them*) is considered the search space. Here, the search space is sorted.

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 ('//' refers to integer division)
- **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.

Note: *For a better understanding, please watch the video at the bottom of the page.*

Iterative implementation:

- Initially, the pointers low and high will be 0 and n-1(*where n = size of the given array*) respectively.
- Now inside a loop, we will perform the 3 steps discussed above in the algorithm section.
- The loop will run until either we fount the target or any of the pointers crosses the other.

▼ Code ›

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

int binarySearch(vector<int>& 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[mid] == target) return mid;
        else if (target > nums[mid]) low = mid + 1;
        else high = mid - 1;
    }
    return -1;
}

int main()
{
    vector<int> a = {3, 4, 6, 7, 9, 12, 16, 17};
    int target = 6;
    int ind = binarySearch(a, target);
    if (ind == -1) cout << "The target is not present." << endl;
    else cout << "The target is at index: "
                << ind << endl;
    return 0;
}
```

Output: The target is at index: 2

▼ Complexity Analysis ›

Time Complexity:

In the algorithm, in every step, we are basically dividing the search space into 2 equal halves. This is actually equivalent to dividing the size of the array by 2, every time. After a certain number of divisions, the size will reduce to such an extent that we will not be able to divide that anymore and the process will stop. The number of total divisions will be equal to the time complexity.

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.

▼ Recursive Approach ›
▼ 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:

1. **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)
2. **Step 2: Compare the middle element with the target and trim down the search space:**
   In this step, we can observe 3 different cases:
   1. **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.
   2. **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)**.
   3. **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)**.
3. **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.

Note: *For a better understanding, please watch the video at the bottom of the page.*

▼ Code ›

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

int binarySearch(vector<int>& nums, int low, int high, int target) {

    if (low > high) return -1; //Base case.

    // Perform the steps:
    int mid = (low + high) / 2;
    if (nums[mid] == target) return mid;
    else if (target > nums[mid])
        return binarySearch(nums, mid + 1, high, target);
    return binarySearch(nums, low, mid - 1, target);
}

int search(vector<int>& nums, int target) {
    return binarySearch(nums, 0, nums.size() - 1, target);
}

int main()
{
    vector<int> a = {3, 4, 6, 7, 9, 12, 16, 17};
    int target = 6;
    int ind = search(a, target);
    if (ind == -1) cout << "The target is not present." << endl;
    else cout << "The target is at index: "
                << ind << endl;
    return 0;
}
```

Output: The target is at index: 2

▼ Complexity Analysis ›

Time Complexity:

In the algorithm, in every step, we are basically dividing the search space into 2 equal halves. This is actually equivalent to dividing the size of the array by 2, every time. After a certain number of divisions, the size will reduce to such an extent that we will not be able to divide that anymore and the process will stop. The number of total divisions will be equal to the time complexity.

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.

▶ Video Explanation >

> *Special thanks to **Sai bargav Nellepalli** and **KRITIDIPTA GHOSH** for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam,* please check out this article

Implement Lower Bound

takeuforward.org/arrays/implement-lower-bound-bs-2

June 10, 2023

Problem Statement: Given a sorted array of **N integers** and an integer x, write a program to find the lower bound of x.

Pre-requisite: Binary Search algorithm

▼ **Examples** ›

```
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.
```

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

Solution:

In the preceding article, we comprehensively explored the implementation of the Binary Search algorithm.

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 this article, we will learn how to implement the lower bound algorithm using a slight modification of the Binary Search algorithm.

Let's start.

What is 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.

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.

Brute Force Approach Optimal Approach

▼ Brute Force Approach ›

▼ Algorithm / Intuition ›

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.

▼ Code ›

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

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

int main()
{
    vector<int> arr = {3, 5, 8, 15, 19};
    int n = 5, x = 9;
    int ind = lowerBound(arr, n, x);
    cout << "The lower bound is the index: " << ind << "\n";
    return 0;
}
```

Output: The lower bound is the index: 3

▼ Complexity Analysis ›
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 ›
▼ Algorithm / Intuition ›

Optimal Approach (Using Binary Search):

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*).

1. **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.

2. **Calculate the 'mid':** Now, we will calculate the value of mid using the following formula:
   mid = (low+high) // 2 ('//' refers to integer division)
3. **Compare arr[mid] with x:** With comparing arr[mid] to x, we can observe 2 different cases:
   1. **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.
   2. **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.

Dry run: *Please refer video for it.*

▼ Code ›

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

int lowerBound(vector<int> 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[mid] >= x) {
            ans = mid;
            //look for smaller index on the left
            high = mid - 1;
        }
        else {
            low = mid + 1; // look on the right
        }
    }
    return ans;
}

int main()
{
    vector<int> arr = {3, 5, 8, 15, 19};
    int n = 5, x = 9;
    int ind = lowerBound(arr, n, x);
    cout << "The lower bound is the index: " << ind << "\n";
    return 0;
}
```

Output: The lower bound is the index: 3

▼ Complexity Analysis ›

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.

▶ Video Explanation ›

> *Special thanks to **KRITIDIPTA GHOSH** for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam,* please check out this article

Implement Upper Bound

takeuforward.org/arrays/implement-upper-bound

June 11, 2023

Problem Statement: Given a sorted array of **N integers** and an integer x, write a program to find the upper bound of x.

Pre-requisite: Binary Search algorithm

▼ **Examples** ›

```
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.
```

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

Solution:

In the preceding article, we comprehensively explored the implementation of the Binary Search algorithm and the lower bound algorithm.

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 this article, we will learn how to implement the upper bound algorithm using a slight modification of the Binary Search algorithm.

What is 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.***

Brute Force Approach Optimal Approach

▼ Brute Force Approach ›

▼ Algorithm / Intuition ›

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.

▼ Code ›

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

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

int main()
{
    vector<int> arr = {3, 5, 8, 9, 15, 19};
    int n = 6, x = 9;
    int ind = upperBound(arr, x, n);
    cout << "The upper bound is the index: " << ind << "\n";
    return 0;
}
```

Output: The upper bound is the index: 4

▼ Complexity Analysis ›
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 ›
▼ Algorithm / Intuition ›

Optimal Approach (Using Binary Search):

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*).

1. **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.

2. **Calculate the 'mid':** Now, we will calculate the value of mid using the following formula:
   mid = (low+high) // 2 ('//' refers to integer division)
3. **Compare arr[mid] with x:** With comparing arr[mid] to x, we can observe 2 different cases:
   1. **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.
   2. **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.

Dry run: *Please refer video for it.*

▼ Code ›

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

int upperBound(vector<int> &arr, int x, int n) {
    int low = 0, high = n - 1;
    int ans = n;

    while (low <= high) {
        int mid = (low + high) / 2;
        // maybe an answer
        if (arr[mid] > x) {
            ans = mid;
            //look for smaller index on the left
            high = mid - 1;
        }
        else {
            low = mid + 1; // look on the right
        }
    }
    return ans;
}

int main()
{
    vector<int> arr = {3, 5, 8, 9, 15, 19};
    int n = 6, x = 9;
    int ind = upperBound(arr, x, n);
    cout << "The upper bound is the index: " << ind << "\n";
    return 0;
}
```

Output: The upper bound is the index: 4

▼ Complexity Analysis ›

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.

▶ Video Explanation ›

upper_bound() in C++ STL:*Please refer to this article and go through the Upper Bound section.*

> *Special thanks to **KRITIDIPTA GHOSH** for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam,* please check out this article

Search Insert Position

takeuforward.org/arrays/search-insert-position

June 12, 2023

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.

Pre-requisite: Lower Bound & Binary Search

```
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.
```

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

Problem Link.

Solution:

We will solve this problem using the lower-bound algorithm which is basically a modified version of the classic Binary Search algorithm.

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.

Approach:

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*).

1. **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.
2. **Calculate the 'mid':** Now, we will calculate the value of mid using the following formula:
   mid = (low+high) // 2 ('//' refers to integer division)
3. **Compare arr[mid] with x:** With comparing arr[mid] to x, we can observe 2 different cases:
   1. **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.
   2. **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.

Dry run: *Please refer video for it.*

Code:

- C++ Code
- Java Code
- Python Code
- JavaScript Code

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

int searchInsert(vector<int>& arr, int x) {
    int n = arr.size(); // size of the array
    int low = 0, high = n - 1;
    int ans = n;

    while (low <= high) {
        int mid = (low + high) / 2;
        // maybe an answer
        if (arr[mid] >= x) {
            ans = mid;
            //look for smaller index on the left
            high = mid - 1;
        }
        else {
            low = mid + 1; // look on the right
        }
    }
    return ans;
}


int main()
{
    vector<int> arr = {1, 2, 4, 7};
    int x = 6;
    int ind = searchInsert(arr, x);
    cout << "The index is: " << ind << "\n";
    return 0;
}
```

Output: The index is: 3

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.

> *Special thanks to* **KRITIDIPTA GHOSH** *for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam,* please check out this article

Watch Video At: https://youtu.be/6zhGS79oQ4k

Floor and Ceil in Sorted Array

takeuforward.org/arrays/floor-and-ceil-in-sorted-array

June 12, 2023

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.

Pre-requisite: Lower Bound & Binary Search

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.

Solution

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

Problem Link.

Solution:

We are going to solve this problem using the concepts of Lower Bound and 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.

What is the floor of x?

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***).

What is the ceiling of 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.

Note:*If the algorithm returns invalid indices which means there is no floor or ceiling of x, we will return -1.*

Note: *Remember, here we are told to find the numbers not the indices. So, we will store the numbers instead of indices in the ans variable.*

Algorithm / Intuition:

Ceil:

We will declare the 2 pointers and an 'ans' variable initialized to -1(*If we don't find any index, we will return -1*).

1. **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.
2. **Calculate the 'mid':** Now, we will calculate the value of mid using the following formula:
   mid = (low+high) // 2 ('//' refers to integer division)
3. **Compare arr[mid] with x:** With comparing arr[mid] to x, we can observe 2 different cases:
   1. **Case 1 – If arr[mid] >= x:** This condition means that the index arr[mid] may be an answer. So, we will update the 'ans' variable with arr[mid] and search in the left half if there is any smaller number that satisfies the same condition. Here, we are eliminating the right half.
   2. **Case 2 – If arr[mid] < x:** In this case, arr[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.

Floor:

We will declare the 2 pointers and an 'ans' variable initialized to -1 (*If we don't find any index, we will return -1*).

4. **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.
5. **Calculate the 'mid':** Now, we will calculate the value of mid using the following formula:
   mid = (low+high) // 2 ('//' refers to integer division)
6. **Compare arr[mid] with x:** With comparing arr[mid] to x, we can observe 2 different cases:
   1. **Case 1 – If arr[mid] <= x:** The index 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.
   2. **Case 2 – If arr[mid] > x:** 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 above process will continue until the pointer low crosses high.

Dry run: *Please refer video for it.*

Code:

- C++ Code
- Java Code
- Python Code
- JavaScript Code

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

int findFloor(int arr[], int n, int x) {
        int low = 0, high = n - 1;
        int ans = -1;

        while (low <= high) {
                int mid = (low + high) / 2;
                // maybe an answer
                if (arr[mid] <= x) {
                        ans = arr[mid];
                        //look for smaller index on the left
                        low = mid + 1;
                }
                else {
                        high = mid - 1; // look on the right
                }
        }
        return ans;
}

int findCeil(int arr[], int n, int x) {
        int low = 0, high = n - 1;
        int ans = -1;

        while (low <= high) {
                int mid = (low + high) / 2;
                // maybe an answer
                if (arr[mid] >= x) {
                        ans = arr[mid];
                        //look for smaller index on the left
                        high = mid - 1;
                }
                else {
                        low = mid + 1; // look on the right
                }
        }
        return ans;
}

pair<int, int> getFloorAndCeil(int arr[], int n, int x) {
        int f = findFloor(arr, n, x);
        int c = findCeil(arr, n, x);
        return make_pair(f, c);
}

int main() {
        int arr[] = {3, 4, 4, 7, 8, 10};
        int n = 6, x = 5;
        pair<int, int> ans = getFloorAndCeil(arr, n, x);
        cout << "The floor and ceil are: " << ans.first
```

```
            << " " << ans.second << endl;
        return 0;
}
```

Output: The floor and ceil are: 4 7

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.

Follow-up Question:

Why are we not comparing the numbers to get the largest or smallest while calculating the floor or ceiling?

- In the floor algorithm, if we get a possible answer, we are reducing the search space to the right half. And the right half certainly contains larger numbers than the current answer. So, every time we are getting larger numbers as answers.
- Similarly, in the ceil algorithm, if we get a possible answer, we are reducing the search space to the left half. And the left half certainly contains smaller numbers than the current answer. So, every time we are getting smaller numbers as answers.
- Basically, in each case, we are moving in a direction such that we get the largest or the smalleset as per our need. That is why we are not checking for the largest or the smallest explicitly.

Note: *Here, we have utilized a variable called 'ans' to store the answers. However, in various instances, you may encounter a different approach where no separate variable is used. Instead, either the 'low' or 'high' pointer is employed as the answer itself. As we progress further in this topic, we will delve into this technique and explore its implementation in detail.*

> *Special thanks to* **KRITIDIPTA GHOSH** *for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam,* please check out this article

Watch Video At: https://youtu.be/6zhGS79oQ4k

Last occurrence in a sorted array

takeuforward.org/data-structure/last-occurrence-in-a-sorted-array

April 10, 2022

Given a sorted array of **N integers**, write a program to find the index of the last occurrence of the target key. If the target is not found then return -1.

Note: Consider 0 based indexing

Examples:

```
Example 1:
Input: N = 7, target=13, array[] = {3,4,13,13,13,20,40}
Output: 4
Explanation: As the target value is 13 , it appears for the first time at index
number 2.

Example 2:
Input: N = 7, target=60, array[] = {3,4,13,13,13,20,40}
Output: -1
Explanation: Target value 60 is not present in the array
```

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

Solution 1: Naive solution

- As the array is already sorted, start traversing the array 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

Code:

- C++ Code
- Java Code

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

using namespace std;

int solve(int n, int key, vector < int > & v) {
  int res = -1;
  for (int i = n - 1; i >= 0; i--) {
    if (v[i] == key) {
      res = i;
      break;
    }
  }
  return res;
}

int main() {
  int n = 7;
  int key = 13;
  vector < int > v = {3,4,13,13,13,20,40};

  // returning the last occurrence index if the element is present otherwise -1
  cout << solve(n, key, v) << "\n";

  return 0;
}
```

Output: 4

Time Complexity: O(n)

Space Complexity: O(1) not considering the given array

Solution 2: Binary search solution (optimised)

As given in the question, the array is already sorted

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

Code:

- C++ Code
- Java Code

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

using namespace std;

int solve(int n, int key, vector < int > & v) {
  int start = 0;
  int end = n - 1;
  int res = -1;

  while (start <= end) {
    int mid = start + (end - start) / 2;
    if (v[mid] == key) {
      res = mid;
      start = mid + 1;

    } else if (key < v[mid]) {
      end = mid - 1;
    } else {
      start = mid + 1;
    }
  }
  return res;
}

int main() {
  int n = 7;
  int key = 13;
  vector < int > v = {3,4,13,13,13,20,40};

  // returning the last occurrence index if the element is present otherwise -1
  cout << solve(n, key, v) << "\n";

  return 0;
}
```

Output: 4

Time Complexity: O(log n)

Space Complexity: O(1)

> *Special thanks to **Rishiraj Girmal** for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam,* please check out this article

Count Occurrences in Sorted Array

takeuforward.org/data-structure/count-occurrences-in-sorted-array

March 7, 2022

Problem Statement: You are given a sorted array containing N integers and a number X, you have to find the occurrences of X in the given array.

▼ **Examples** ›

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.

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

Brute Force Approach Optimal Approach

▼ Brute Force Approach ›
▼ Algorithm / Intuition ›

Approach:

The approach is simple. We will linearly search the entire array, and try to increase the counter whenever we get the target value in the array. Using a for loop that runs from 0 to n – 1, containing an if the condition that checks whether the value at that index equals target. If true then increase the counter, at last return the counter.

▼ Code ›

```
#include<bits/stdc++.h>
using namespace std;
int count(vector<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;
}

int main()
{
    vector<int> arr =  {2, 4, 6, 8, 8, 8, 11, 13};
    int n = 8, x = 8;
    int ans = count(arr, n, x);
    cout << "The number of occurrences is: "
         << ans << "\n";
    return 0;
}
```

Output: The number of occurrences is: 3

▼ Complexity Analysis ›
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 ›
▼ Algorithm / Intuition ›

Optimal Approach(Binary Search):

In the previous article, we discussed how to find the first and the last occurrences of a number in a sorted array 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.

Now in order to solve this problem, we are going to use the previous concept. We will find the first and the last occurrences and figure out the number of occurrences like the following:

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

Algorithm:

- We will get the first and the last occurrences of the number using the function **firstAndLastPosition()**. For the implementation details of the function, please refer to the previous article.
- After getting the indices, we will check the following cases:
    - **If the first index == -1:** This means that the target value is not present in the array. So, we will return 0 as the answer.
    - **Otherwise:** We will find the total number of occurrences like this:
      The total number of occurrences = (last index – first index + 1) and return this length as the answer.

▼ Code ›

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

int firstOccurrence(vector<int> &arr, int n, int k) {
    int low = 0, high = n - 1;
    int first = -1;

    while (low <= high) {
        int mid = (low + high) / 2;
        // maybe an answer
        if (arr[mid] == k) {
            first = mid;
            //look for smaller index on the left
            high = mid - 1;
        }
        else if (arr[mid] < k) {
            low = mid + 1; // look on the right
        }
        else {
            high = mid - 1; // look on the left
        }
    }
    return first;
}

int lastOccurrence(vector<int> &arr, int n, int k) {
    int low = 0, high = n - 1;
    int last = -1;

    while (low <= high) {
        int mid = (low + high) / 2;
        // maybe an answer
        if (arr[mid] == k) {
            last = mid;
            //look for larger index on the right
            low = mid + 1;
        }
        else if (arr[mid] < k) {
            low = mid + 1; // look on the right
        }
        else {
            high = mid - 1; // look on the left
        }
    }
    return last;
}


pair<int, int> firstAndLastPosition(vector<int>& arr, int n, int k) {
    int first = firstOccurrence(arr, n, k);
    if (first == -1) return { -1, -1};
```

```
    int last = lastOccurrence(arr, n, k);
    return {first, last};
}

int count(vector<int>& arr, int n, int x) {
    pair<int, int> ans = firstAndLastPosition(arr, n, x);
    if (ans.first == -1) return 0;
    return (ans.second - ans.first + 1);
}

int main()
{
    vector<int> arr =  {2, 4, 6, 8, 8, 8, 11, 13};
    int n = 8, x = 8;
    int ans = count(arr, n, x);
    cout << "The number of occurrences is: "
         << ans << "\n";
    return 0;
}
```

Output: The number of occurrences is: 3

▼ Complexity Analysis ›

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.

▶ Video Explanation ›

> *Special thanks to Rushikesh Yadav and **KRITIDIPTA GHOSH** for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam,* please check out this article

Search Element in a Rotated Sorted Array

takeuforward.org/data-structure/search-element-in-a-rotated-sorted-array

November 29, 2021

Problem Statement: Given an integer array arr of size N, sorted in ascending order (with distinct values) and a target value k. Now the array is rotated at some pivot point unknown to you. Find the index at which k is present and if k is not present return -1.

▼ **Examples** >

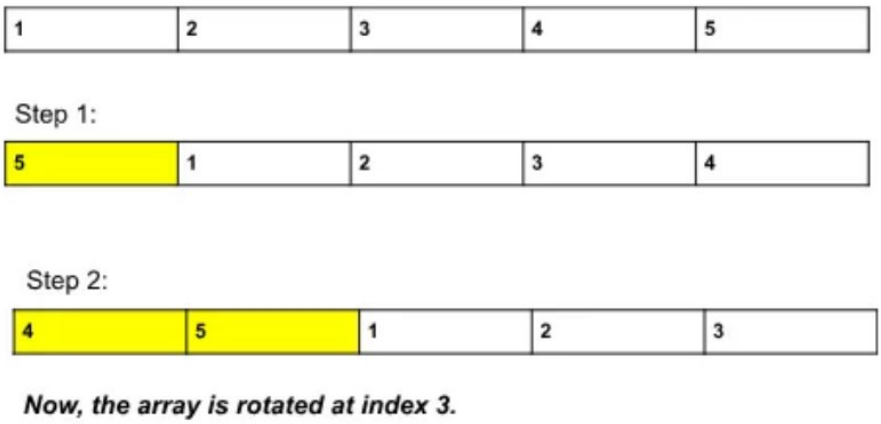
```
Example 1:
Input Format: arr = [4,5,6,7,0,1,2,3], k = 0
Result: 4
Explanation: Here, the target is 0. We can see that 0 is present in the given rotated sorted array, nums.
Thus, we get output as 4, which is the index at which 0 is present in the array.

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.
```

Solution:

How does the rotation occur in a sorted array?

Let's consider a sorted array: {1, 2, 3, 4, 5}. If we rotate this array at index 3, it will become: {4, 5, 1, 2, 3}. In essence, we moved the element at the last index to the front, while shifting the remaining elements to the right. We performed this process twice.



Now, the array is rotated at index 3.

Practice:

Solve Problem

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

Brute Force Approach Optimal Approach

▼ Brute Force Approach >

▼ Algorithm / Intuition >

Naive Approach (Brute force):

One straightforward approach we can consider is using the linear search algorithm. Using this method, we will traverse the array to find the location of the target value. If it is found we will simply return the index and otherwise, we will return -1.

Algorithm:

- We will traverse the array and check every element if it is equal to k. If we find any element, we will return its index.
- **Otherwise,** we will return -1.

▼ Code >

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

int search(vector<int>& arr, int n, int k) {
    for (int i = 0; i < n; i++) {
        if (arr[i] == k)
            return i;
    }
    return -1;
}

int main()
{
    vector<int> arr = {7, 8, 9, 1, 2, 3, 4, 5, 6};
    int n = 9, k = 1;
    int ans = search(arr, n, k);
    if (ans == -1)
        cout << "Target is not present.\n";
    else
        cout << "The index is: " << ans << "\n";
    return 0;
}
```

Output: The index is: 3

▼ Complexity Analysis ›
Time Complexity: O(N), N = size of the given array.
Reason: We have to iterate through the entire array to check if the target is present in the array.

Space Complexity: O(1)
Reason: We have not used any extra data structures, this makes space complexity, even in the worst case as O(1).

▼ Optimal Approach ›
▼ Algorithm / Intuition ›

Optimal Approach(Using Binary Search):

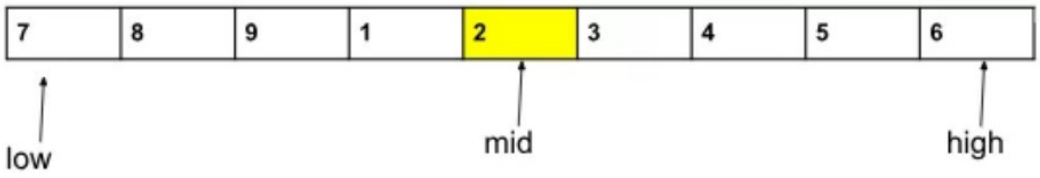
Here, we can easily observe, that we have to search in a sorted array. That is why, we can think of using the Binary Search algorithm to solve this problem.

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:



target = 8,
Considering the comparison made, such as target > arr[mid] (e.g., 8 > 2), we would expect the target to be in the right half. However, due to the array rotation, the number 8 is actually situated in the left half. This rotation creates a challenge in the elimination process.

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:

1. **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.
2. **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)
3. **Check if arr[mid] == target:** If it is, return the index mid.

4. Identify the sorted half, check where the target is located, and then eliminate one half accordingly:
    1. **If arr[low] <= arr[mid]:** This condition ensures that the left part is sorted.
        1. **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***).
        2. **Otherwise,** the target does not exist in the sorted half. So, we will eliminate this left half by doing **low = mid+1**.
    2. **Otherwise, if the right half is sorted:**
        1. **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***).
        2. **Otherwise,** the target does not exist in this sorted half. So, we will eliminate this right half by doing **high = mid-1**.
5. Once, the 'mid' points to the target, the index will be returned.
6. 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.

Dry-run: *Please refer to the video for it.*

▼ Code ›

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

int search(vector<int>& arr, int n, int k) {
    int low = 0, high = n - 1;
    while (low <= high) {
        int mid = (low + high) / 2;

        //if mid points the target
        if (arr[mid] == k) return mid;

        //if left part is sorted:
        if (arr[low] <= arr[mid]) {
            if (arr[low] <= k && k <= arr[mid]) {
                //element exists:
                high = mid - 1;
            }
            else {
                //element does not exist:
                low = mid + 1;
            }
        }
        else { //if right part is sorted:
            if (arr[mid] <= k && k <= arr[high]) {
                //element exists:
                low = mid + 1;
            }
            else {
                //element does not exist:
                high = mid - 1;
            }
        }
    }
    return -1;
}

int main()
{
    vector<int> arr = {7, 8, 9, 1, 2, 3, 4, 5, 6};
    int n = 9, k = 1;
    int ans = search(arr, n, k);
    if (ans == -1)
        cout << "Target is not present.\n";
    else
        cout << "The index is: " << ans << "\n";
    return 0;
}
```

Output: The index is: 3

▼ Complexity Analysis ›

Time Complexity: O(logN), N = size of the given array.
Reason: We are using binary search to search the target.

Space Complexity: O(1)
Reason: We have not used any extra data structures, this makes space complexity, even in the worst case as O(1).

▶ Video Explanation ›

> *Special thanks to **Dewanshi Paul**, **Sudip Ghosh** and **KRITIDIPTA GHOSH** for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam,* please check out this article

Search Element in Rotated Sorted Array II

takeuforward.org/arrays/search-element-in-rotated-sorted-array-ii

June 20, 2023

Problem Statement: Given an integer array **arr** of size **N**, sorted in ascending order (**may contain duplicate values**) and a target value **k**. Now the array is rotated at some pivot point unknown to you. Return True if **k** is present and otherwise, return False.

Pre-requisite: Search Element in Rotated Sorted Array I **&** Binary Search algorithm

▼ **Examples** ›

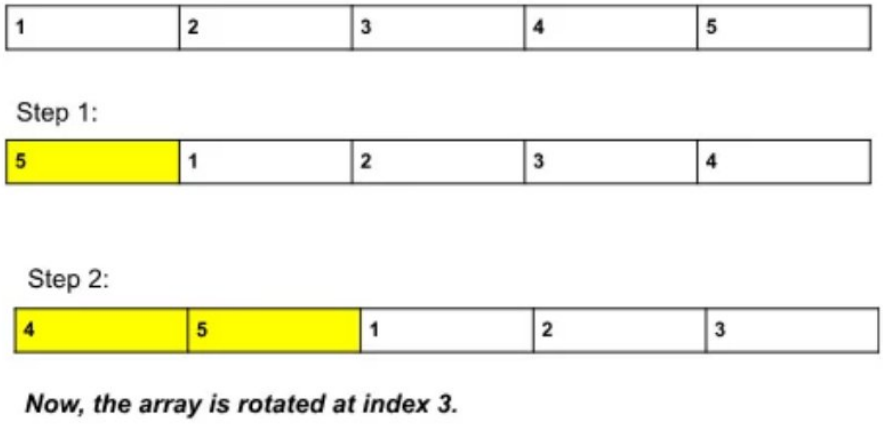
```
Example 1:
Input Format: arr = [7, 8, 1, 2, 3, 3, 3, 4, 5, 6], k = 3
Result: True
Explanation: The element 3 is present in the array. So, the answer is True.

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.
```

Solution:

How does the rotation occur in a sorted array?

Let's consider a sorted array: {1, 2, 3, 4, 5}. If we rotate this array at index 3, it will become: {4, 5, 1, 2, 3}. In essence, we moved the element at the last index to the front, while shifting the remaining elements to the right. We performed this process twice.



Now, the array is rotated at index 3.

Practice:
Solve Problem

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

Brute Force Approach Optimal Approach

▼ Brute Force Approach ›

▼ Algorithm / Intuition ›

Naive Approach (Brute force):

One straightforward approach we can consider is using the linear search algorithm. Using this method, we will traverse the array to check if the target is present in the array. If it is found we will simply return True and otherwise, we will return False.

Algorithm:

- We will traverse the array and check every element if it is equal to k. If we find any element, we will return True.
- **Otherwise,** we will return False.

▼ Code ›

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

bool searchInARotatedSortedArrayII(vector<int>&arr, int k) {
    int n = arr.size(); // size of the array.
    for (int i = 0; i < n; i++) {
        if (arr[i] == k) return true;
    }
    return false;
}

int main()
{
    vector<int> arr = {7, 8, 1, 2, 3, 3, 3, 4, 5, 6};
    int k = 3;
    bool ans = searchInARotatedSortedArrayII(arr, k);
    if (!ans)
        cout << "Target is not present.\n";
    else
        cout << "Target is present in the array.\n";
    return 0;
}
```

Output: Target is present in the array.

▼ Complexity Analysis ›
Time Complexity: O(N), N = size of the given array.
Reason: We have to iterate through the entire array to check if the target is present in the array.

Space Complexity: O(1)
Reason: We have not used any extra data structures, this makes space complexity, even in the worst case as O(1).

▼ Optimal Approach ›
▼ Algorithm / Intuition ›

Optimal Approach(Using Binary Search):

Like the previous problem, we will use the Binary Search algorithm to solve this problem.

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.

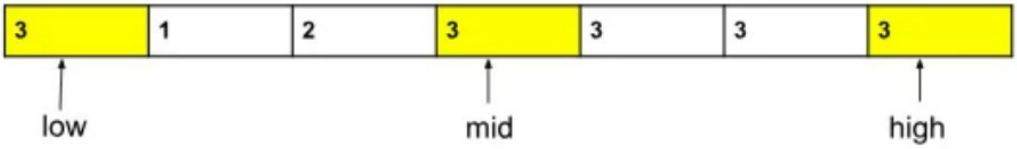
This check was effective in the previous problem, where there were no duplicate numbers. However, in the current problem, the array may contain duplicates. ***Consequently, the previous approach will not work when arr[low] = arr[mid] = arr[high].***

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

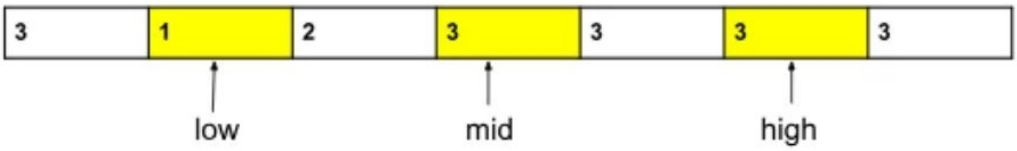
In the algorithm, we first check if arr[mid] is the target before identifying the sorted half. If arr[mid] is not our target, we encounter this edge case. In this scenario, since arr[mid] = arr[low] = arr[high], it means that neither arr[low] nor arr[high] can be the target. To handle

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.



Now, we will remove arr[low] and arr[high] from the search space



Now, 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:

1. **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.
2. **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)
3. **Check if arr[mid] = target:** If it is, return True.
4. **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***.

5. Identify the sorted half, check where the target is located, and then eliminate one half accordingly:
    1. **If arr[low] <= arr[mid]:** This condition ensures that the left part is sorted.
        1. **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***).
        2. **Otherwise,** the target does not exist in the sorted half. So, we will eliminate this left half by doing **low = mid+1**.
    2. **Otherwise, if the right half is sorted:**
        1. **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***).
        2. **Otherwise,** the target does not exist in this sorted half. So, we will eliminate this right half by doing **high = mid-1**.
6. Once, the 'mid' points to the target, we will return True.
7. 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.

Dry-run: *Please refer to the video for it.*

▼ Code ›

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

bool searchInARotatedSortedArrayII(vector<int>&arr, int k) {
    int n = arr.size(); // size of the array.
    int low = 0, high = n - 1;
    while (low <= high) {
        int mid = (low + high) / 2;

        //if mid points the target
        if (arr[mid] == k) return true;

        //Edge case:
        if (arr[low] == arr[mid] && arr[mid] == arr[high]) {
            low = low + 1;
            high = high - 1;
            continue;
        }

        //if left part is sorted:
        if (arr[low] <= arr[mid]) {
            if (arr[low] <= k && k <= arr[mid]) {
                //element exists:
                high = mid - 1;
            }
            else {
                //element does not exist:
                low = mid + 1;
            }
        }
        else { //if right part is sorted:
            if (arr[mid] <= k && k <= arr[high]) {
                //element exists:
                low = mid + 1;
            }
            else {
                //element does not exist:
                high = mid - 1;
            }
        }
    }
    return false;
}

int main()
{
    vector<int> arr = {7, 8, 1, 2, 3, 3, 3, 4, 5, 6};
    int k = 3;
    bool ans = searchInARotatedSortedArrayII(arr, k);
    if (!ans)
        cout << "Target is not present.\n";
```

```
    else
        cout << "Target is present in the array.\n";
    return 0;
}
```

Output: Target is present in the array.

▼ Complexity Analysis ›

Time Complexity: O(logN) for the best and average case. **O(N/2)** for the worst case. Here, N = size of the given array.

Reason: In the best and average scenarios, the binary search algorithm is primarily utilized and hence the time complexity is O(logN). However, in the worst-case scenario, where all array elements are the same but not the target (e.g., given array = {3, 3, 3, 3, 3, 3, 3}), we continue to reduce the search space by adjusting the low and high pointers until they intersect. This worst-case situation incurs a time complexity of O(N/2).

Space Complexity: O(1)

Reason: We have not used any extra data structures, this makes space complexity, even in the worst case as O(1).

▶ Video Explanation ›

> *Special thanks to **KRITIDIPTA GHOSH** for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam,* please check out this article

Minimum in Rotated Sorted Array

takeuforward.org/data-structure/minimum-in-rotated-sorted-array

July 17, 2022

Problem Statement: Given an integer array **arr** of size **N**, sorted in ascending order (**with distinct values**). Now the array is rotated between 1 to N times which is unknown. Find the minimum element in the array.

Pre-requisites: Search in Rotated Sorted Array I, Search in Rotated Sorted Array II & Binary Search algorithm

▼ **Examples** ›

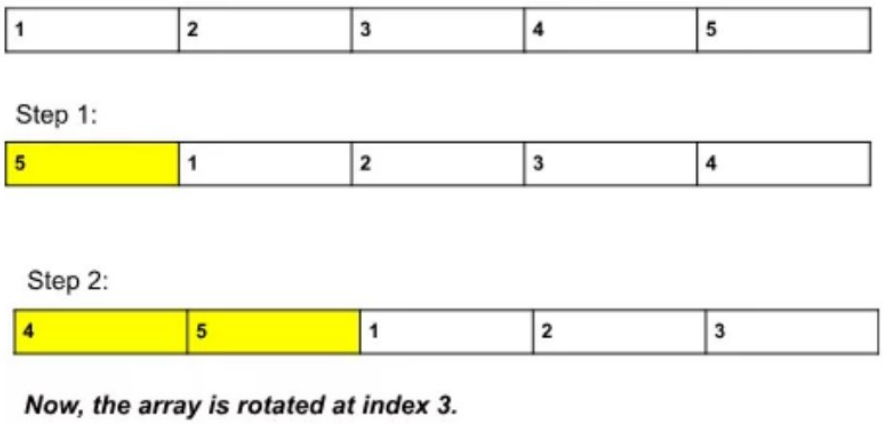
```
Example 1:
Input Format: arr = [4,5,6,7,0,1,2,3]
Result: 0
Explanation: Here, the element 0 is the minimum element in the array.

Example 2:
Input Format: arr = [3,4,5,1,2]
Result: 1
Explanation: Here, the element 1 is the minimum element in the array.
```

Solution:

How does the rotation occur in a sorted array?

Let's consider a sorted array: {1, 2, 3, 4, 5}. If we rotate this array at index 3, it will become: {4, 5, 1, 2, 3}. In essence, we moved the element at the last index to the front, while shifting the remaining elements to the right. We performed this process twice.



Now, the array is rotated at index 3.

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

Solution 1 Solution 2 Solution 3

▼ Solution 1: ›
▼ Algorithm / Intuition ›

Naive Approach (Brute force):

One straightforward approach, we can consider is using the linear search algorithm. Using this method, we will find the minimum number from the array.

Algorithm:

- First, we will declare a 'mini' variable initialized with a large number.
- After that, we will traverse the array and compare each element with the 'mini' variable. Each time the 'mini' variable will be updated with the minimum value i.e. min(mini, arr[i]).
- Finally, we will return 'mini' as our answer.

▼ Code ›

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

int findMin(vector<int>& arr) {
    int n = arr.size(); // size of the array.
    int mini = INT_MAX;
    for (int i = 0; i < n; i++) {
        // Always keep the minimum.
        mini = min(mini, arr[i]);
    }
    return mini;
}

int main()
{
    vector<int> arr = {4, 5, 6, 7, 0, 1, 2, 3};
    int ans = findMin(arr);
    cout << "The minimum element is: " << ans << "\n";
    return 0;
}
```

Output: The minimum element is: 0

▼ Complexity Analysis ›
Time Complexity: O(N), N = size of the given array.
Reason: We have to iterate through the entire array to check if the target is present in the array.

Space Complexity: O(1)
Reason: We have not used any extra data structures, this makes space complexity, even in the worst case as O(1).

▼ Solution 2: ›
▼ Algorithm / Intuition ›

Optimal Approach(Using Binary Search):

Here, we can easily observe, that we have to find the minimum in a sorted array. That is why, we can think of using the Binary Search algorithm to solve this problem.

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.

Key Observation: If an array is rotated and sorted, we already know that for every index, one of the 2 halves of the array will always be sorted.

Based on this observation, we adopted a straightforward two-step process to eliminate one-half of the rotated sorted array.

- 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.

Let's observe how we will find the minimum element:

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.*

Algorithm:

The steps are as follows:

We will declare the 'ans' variable and initialize it with the largest value possible. With that, as usual, we will declare 2 pointers i.e. low and high.

1. **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.
2. **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)
3. Identify the sorted half, and after picking the leftmost element, eliminate that half.
    1. **If arr[low] <= arr[mid]:** This condition ensures that the left part is sorted. So, we will pick the leftmost element i.e. **arr[low]**. Now, we will compare it with 'ans' and update 'ans' with the smaller value (i.e., **min(ans, arr[low])**). Now, we will eliminate this left half(i.e. low = mid+1).
    2. **Otherwise, if the right half is sorted:** This condition ensures that the right half is sorted. So, we will pick the leftmost element i.e. **arr[mid]**. Now, we will compare it with 'ans' and update 'ans' with the smaller value (i.e., **min(ans, arr[mid])**). Now, we will eliminate this right half(i.e. high = mid-1).
4. This process will be inside a loop and the loop will continue until low crosses high. Finally, we will return the 'ans' variable that stores the minimum element.

Dry-run: *Please refer to the video for a detailed explanation.*

Example:{4,5,1,2,3}
low=0,high=4,mid=2
Check if arr[low] <= arr[mid], its not,
So right part is sorted.
We take ans=min(ans,arr[2]) => ans=1, and high = mid-1.

low=0,high=1,mid=0;
arr[low]<=arr[mid] = true.
So we update ans as min(ans,arr[0]) => ans=1;
Since the left part was sorted low=mid+1. Which makes low = high = 1.

low=1,high=1,mid=1
arr[low] <= arr[mid] = true.
So we update ans as min(ans,arr[1]) => ans=1;
Since the left part was sorted low=mid+1. Which makes low = 2. Loop Stops.

▼ Code ›

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

int findMin(vector<int>& arr) {
    int low = 0, high = arr.size() - 1;
    int ans = INT_MAX;
    while (low <= high) {
        int mid = (low + high) / 2;

        //if left part is sorted:
        if (arr[low] <= arr[mid]) {
            // keep the minimum:
            ans = min(ans, arr[low]);

            // Eliminate left half:
            low = mid + 1;
        }
        else { //if right part is sorted:

            // keep the minimum:
            ans = min(ans, arr[mid]);

            // Eliminate right half:
            high = mid - 1;
        }
    }
    return ans;
}

int main()
{
    vector<int> arr = {4, 5, 6, 7, 0, 1, 2, 3};
    int ans = findMin(arr);
    cout << "The minimum element is: " << ans << "\n";
    return 0;
}
```

Output: The minimum element is: 0

▼ Complexity Analysis ›

Time Complexity: O(logN), N = size of the given array.
Reason: We are basically using binary search to find the minimum.

Space Complexity: O(1)
Reason: We have not used any extra data structures, this makes space complexity, even in the worst case as O(1).

▼ Solution 3: ›

▼ Algorithm / Intuition ›

Further Optimization(Using Binary Search):

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.

Algorithm:

The steps are as follows:

We will declare the 'ans' variable and initialize it with the largest value possible. With that, as usual, we will declare 2 pointers i.e. low and high.

1. **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.
2. **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)
3. **If arr[low] <= arr[high]:** In this case, the array from index low to high is completely sorted. Therefore, we can 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. So, we will **break** from this step.
4. Identify the sorted half, and after picking the leftmost element, eliminate that half.
    1. **If arr[low] <= arr[mid]:** This condition ensures that the left part is sorted. So, we will pick the leftmost element i.e. **arr[low]**. Now, we will compare it with 'ans' and update 'ans' with the smaller value (i.e., **min(ans, arr[low])**). Now, we will eliminate this left half(i.e. low = mid+1).
    2. **Otherwise, if the right half is sorted:** This condition ensures that the right half is sorted. So, we will pick the leftmost element i.e. **arr[mid]**. Now, we will compare it with 'ans' and update 'ans' with the smaller value (i.e., **min(ans, arr[mid])**). Now, we will eliminate this right half(i.e. high = mid-1).
5. This process will be inside a loop and the loop will continue until low crosses high. Finally, we will return the 'ans' variable that stores the minimum element.

Dry-run: *Please refer to the video for a detailed explanation.*

Note: *Though the time complexity of the following code is the same as the previous one, this code will run slightly faster.*

▼ Code ›

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

int findMin(vector<int>& arr) {
    int low = 0, high = arr.size() - 1;
    int ans = INT_MAX;
    while (low <= high) {
        int mid = (low + high) / 2;
        //search space is already sorted
        //then arr[low] will always be
        //the minimum in that search space:
        if (arr[low] <= arr[high]) {
            ans = min(ans, arr[low]);
            break;
        }

        //if left part is sorted:
        if (arr[low] <= arr[mid]) {
            // keep the minimum:
            ans = min(ans, arr[low]);

            // Eliminate left half:
            low = mid + 1;
        }
        else { //if right part is sorted:

            // keep the minimum:
            ans = min(ans, arr[mid]);

            // Eliminate right half:
            high = mid - 1;
        }
    }
    return ans;
}

int main()
{
    vector<int> arr = {4, 5, 6, 7, 0, 1, 2, 3};
    int ans = findMin(arr);
    cout << "The minimum element is: " << ans << "\n";
    return 0;
}
```

Output: The minimum element is: 0

▼ Complexity Analysis ›

Time Complexity: O(logN), N = size of the given array.
Reason: We are basically using binary search to find the minimum.

Space Complexity: O(1)
Reason: We have not used any extra data structures, this makes space complexity, even in the worst case as O(1).

▶ Video Explanation ›

> *Special thanks to **KRITIDIPTA GHOSH** for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam,* please check out this article

Find out how many times the array has been rotated

takeuforward.org/arrays/find-out-how-many-times-the-array-has-been-rotated

June 27, 2023

Problem Statement: Given an integer array **arr** of size **N**, sorted in ascending order (**with distinct values**). Now the array is rotated between 1 to N times which is unknown. Find how many times the array has been rotated.

Pre-requisites: Find minimum in Rotated Sorted Array, Search in Rotated Sorted Array II & Binary Search algorithm

▼ **Examples** ›

```
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.
```

Solution:

How does the rotation occur in a sorted array?

Let's consider a sorted array: {1, 2, 3, 4, 5}. If we rotate this array 2 times, it will become: {4, 5, 1, 2, 3}. In essence, we moved the element at the last index to the front, while shifting the remaining elements to the right. We performed this process twice.

Step 1:

Step 2:

Now, the array is rotated 2 times. And the minimum element is at index 2.

Observation:

- 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.***

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

Brute Force Approach Optimal Approach

▼ Brute Force Approach ›
▼ Algorithm / Intuition ›

Algorithm:

- First, we will declare an 'ans' and an 'index' variable initialized with a large number and -1 respectively.
- Next, we will iterate through the array and compare each element with the variable called 'ans'. Whenever we encounter an element 'arr[i]' that is smaller than 'ans', we will update 'ans' with the value of 'arr[i]' and also update the 'index' variable with the corresponding index value, 'i'.
- Finally, we will return 'index' as our answer.

▼ Code ›

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

int findKRotation(vector<int> &arr) {
    int n = arr.size(); //size of array.
    int ans = INT_MAX, index = -1;
    for (int i = 0; i < n; i++) {
        if (arr[i] < ans) {
            ans = arr[i];
            index = i;
        }
    }
    return index;
}

int main()
{
    vector<int> arr = {4, 5, 6, 7, 0, 1, 2, 3};
    int ans = findKRotation(arr);
    cout << "The array is rotated " << ans << " times.\n";
    return 0;
}
```

Output: The array is rotated 4 times.

▼ Complexity Analysis ›
Time Complexity: O(N), N = size of the given array.
Reason: We have to iterate through the entire array to check if the target is present in the array.

Space Complexity: O(1)
Reason: We have not used any extra data structures, this makes space complexity, even in the worst case as O(1).

▼ Optimal Approach ›
▼ Algorithm / Intuition ›

Optimal Approach(Using Binary Search):

We are going to use the binary search algorithm to solve this problem.

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 article, find the minimum in a rotated sorted array., we have discussed how to find the minimum element in a rotated and sorted array using Binary search. In this problem, we will employ the same algorithm to determine the index of the minimum element. In the previous problem, we only stored the minimum element itself. However, in this updated approach, we will additionally keep track of the index. By making this small adjustment, we can effectively solve the problem using the existing algorithm.

Algorithm:

The steps are as follows:

To begin, we will declare the variable 'ans' and initialize it with the largest possible value. Additionally, we will have two pointers, 'low' and 'high', as usual. In this case, we will also introduce an 'index' variable and initialize it with -1.

1. **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.
2. **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)
3. **If arr[low] <= arr[high]:** In this case, the array from index low to high is completely sorted. Therefore, we can select the minimum element, arr[low].
   Now, ***if arr[low] < ans, we will update 'ans' with the value arr[low] and 'index' with the corresponding index low.***
   Once this is done, there is no need to continue with the binary search algorithm. So, we will **break** from this step.
4. Identify the sorted half, and after picking the leftmost element, eliminate that half.
   1. **If arr[low] <= arr[mid]:**
      This condition ensures that the left part is sorted. So, we will pick the leftmost element i.e. **arr[low]**.
      Now, ***if arr[low] < ans, we will update 'ans' with the value arr[low] and 'index' with the corresponding index low.***
      After that, we will eliminate this left half(i.e. low = mid+1).
   2. **Otherwise, if the right half is sorted:** This condition ensures that the right half is sorted. So, we will pick the leftmost element i.e. **arr[mid]**.
      Now, ***if arr[mid] < ans, we will update 'ans' with the value arr[mid] and 'index' with the corresponding index mid.***
      After that, we will eliminate this right half(i.e. high = mid-1).
5. This process will be inside a loop and the loop will continue until low crosses high. Finally, we will return the 'index' variable that stores the index of the minimum element.

Dry-run: *Please refer to the video for a detailed explanation.*

▼ Code ›

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

int findKRotation(vector<int> &arr) {
    int low = 0, high = arr.size() - 1;
    int ans = INT_MAX;
    int index = -1;
    while (low <= high) {
        int mid = (low + high) / 2;
        //search space is already sorted
        //then arr[low] will always be
        //the minimum in that search space:
        if (arr[low] <= arr[high]) {
            if (arr[low] < ans) {
                index = low;
                ans = arr[low];
            }
            break;
        }

        //if left part is sorted:
        if (arr[low] <= arr[mid]) {
            // keep the minimum:
            if (arr[low] < ans) {
                index = low;
                ans = arr[low];
            }

            // Eliminate left half:
            low = mid + 1;
        }
        else { //if right part is sorted:

            // keep the minimum:
            if (arr[mid] < ans) {
                index = mid;
                ans = arr[mid];
            }

            // Eliminate right half:
            high = mid - 1;
        }
    }
    return index;
}

int main()
{
    vector<int> arr = {4, 5, 6, 7, 0, 1, 2, 3};
```

```
    int ans = findKRotation(arr);
    cout << "The array is rotated " << ans << " times.\n";
    return 0;
}
```

Output: The array is rotated 4 times.

▼ Complexity Analysis ›
Time Complexity: O(logN), N = size of the given array.
Reason: We are basically using binary search to find the minimum.

Space Complexity: O(1)
Reason: We have not used any extra data structures, this makes space complexity, even in the worst case as O(1).

▶ Video Explanation ›

Search Single Element in a sorted array

takeuforward.org/data-structure/search-single-element-in-a-sorted-array

November 22, 2021

Problem Statement: Given an array of N integers. Every number in the array except one appears twice. Find the single number in the array.

Pre-requisite: Binary Search Algorithm

▼ **Examples** ›

```
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.
```

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

Brute Force Approach 1: Brute Force Approach 2: Optimal Approach:

▼ Brute Force Approach 1: ›

▼ Algorithm / Intuition ›

Naive Approach(Brute force):

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

Edge Cases:

1. **If n == 1:** This means the array size is 1. If the array contains only one element, we will return that element only.
2. **If i == 0:** This means this is the very first element of the array. The only condition, we need to check is: **arr[i] != arr[i+1].**
3. **If i == n-1:** This means this is the last element of the array. The only condition, we need to check is: **arr[i] != arr[i-1].**

So, we will traverse the array and we will check for the above conditions.

Algorithm:

The steps are as follows:

1. At first, we will check if the array contains only 1 element. If it is, we will simply return that element.
2. We will start traversing the array. Then for every element, we will check the following.
3. **If i == 0:** If we are at the first index, we will check if the next element is equal.
    1. **If arr[i] != arr[i+1]:** This means arr[i] is the single element and so we will return arr[i].
4. **If i == n-1:** If we are at the last index, we will check if the previous element is equal.
    1. **If arr[i] != arr[i-1]:** This means arr[i] is the single element and so we will return arr[i].
5. For the elements other than the first and last, we will check:
   If arr[i] != arr[i-1] and arr[i] != arr[i+1]: If this condition is true for any element, arr[i], we can conclude this is the single element. And we should return arr[i].

Dry-run: *Please refer to the video for the dry-run.*

▼ Code ›

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

int singleNonDuplicate(vector<int>& arr) {
    int n = arr.size(); //size of the array.
    if (n == 1) return arr[0];

    for (int i = 0; i < n; i++) {

        //Check for first index:
        if (i == 0) {
            if (arr[i] != arr[i + 1])
                return arr[i];
        }
        //Check for last index:
        else if (i == n - 1) {
            if (arr[i] != arr[i - 1])
                return arr[i];
        }
        else {
            if (arr[i] != arr[i - 1] && arr[i] != arr[i + 1])
                return arr[i];
        }
    }

    // dummy return statement:
    return -1;
}

int main()
{
    vector<int> arr = {1, 1, 2, 2, 3, 3, 4, 5, 5, 6, 6};
    int ans = singleNonDuplicate(arr);
    cout << "The single element is: " << ans << "\n";
    return 0;
}
```

Output: The single element is: 4

▼ Complexity Analysis ›

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

Reason: We are traversing the entire array.

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

▼ Brute Force Approach 2: ›
▼ Algorithm / Intuition ›

Naive Approach(Using XOR):

We can simplify the above approach using the XOR operation. We need to remember 2 important properties of XOR:

- a ^ a = 0, XOR of two same numbers results in 0.
- a ^ 0 = a, XOR of a number with 0 always results in that number.

Now, if we XOR all the array elements, all the duplicates will result in 0 and we will be left with a single element.

Algorithm:

1. We will declare an ‘ans’ variable initialized with 0.
2. We will traverse the array and XOR each element with the variable ‘ans’.
3. After complete traversal, the ‘ans’ variable will store the single element and we will return it.

▼ Code ›

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

int singleNonDuplicate(vector<int>& arr) {
    int n = arr.size(); //size of the array.
    int ans = 0;
    // XOR all the elements:
    for (int i = 0; i < n; i++) {
        ans = ans ^ arr[i];
    }
    return ans;
}

int main()
{
    vector<int> arr = {1, 1, 2, 2, 3, 3, 4, 5, 5, 6, 6};
    int ans = singleNonDuplicate(arr);
    cout << "The single element is: " << ans << "\n";
    return 0;
}
```

Output: The single element is: 4

▼ Complexity Analysis ›
Time Complexity: O(N), N = size of the given array.
Reason: We are traversing the entire array.

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

▼ Optimal Approach ›
▼ Algorithm / Intuition ›

Optimal Approach(Using Binary Search):

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

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

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:

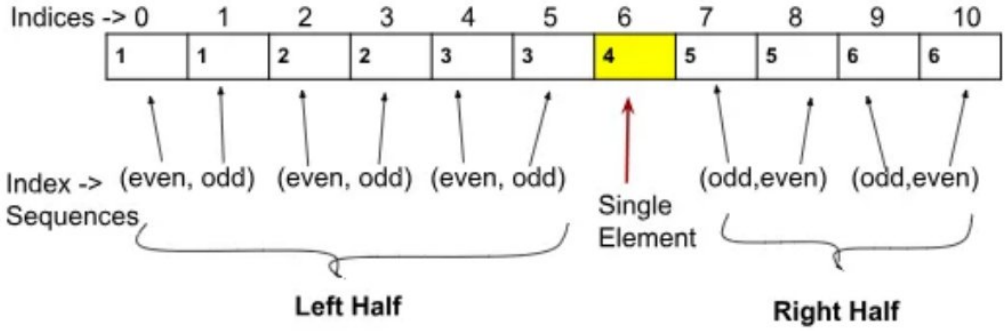
- **If the array size is 1.**
- **If 'mid' points to 0 i.e. the first index.**
- **If 'mid' points to n-1 i.e. the last index.**

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:

1. **If n == 1:** This means the array size is 1. If the array contains only one element, we will return that element only.
2. **If arr[0] != arr[1]:** This means the very first element of the array is the single element. So, we will return arr[0].
3. **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.

1. 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:
    1. **If the current index is even, the element at the next odd index will be the same as the current element.**
    2. **Similarly, If the current index is odd, the element at the preceding even index will be the same as the current element.**
2. 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:
    1. **If the current index is even, the element at the preceding odd index will be the same as the current element.**
    2. **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:

1. **If n == 1:** This means the array size is 1. If the array contains only one element, we will return that element only.
2. **If arr[0] != arr[1]:** This means the very first element of the array is the single element. So, we will return arr[0].
3. **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].
4. **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.
5. **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)
6. **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].
7. **If (mid % 2 == 0 and arr[mid] == arr[mid+1])**
   or (mid%2 == 1 and arr[mid] == arr[mid-1]): This meanswe 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.
8. **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.**Dry-run:** *Please refer to the video for a detailed explanation.*

▼ Code ›

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

int singleNonDuplicate(vector<int>& arr) {
    int n = arr.size(); //size of the array.

    //Edge cases:
    if (n == 1) return arr[0];
    if (arr[0] != arr[1]) return arr[0];
    if (arr[n - 1] != arr[n - 2]) return arr[n - 1];

    int low = 1, high = n - 2;
    while (low <= high) {
        int mid = (low + high) / 2;

        //if arr[mid] is the single element:
        if (arr[mid] != arr[mid + 1] && arr[mid] != arr[mid - 1]) {
            return arr[mid];
        }

        //we are in the left:
        if ((mid % 2 == 1 && arr[mid] == arr[mid - 1])
                || (mid % 2 == 0 && arr[mid] == arr[mid + 1])) {
            //eliminate the left half:
            low = mid + 1;
        }
        //we are in the right:
        else {
            //eliminate the right half:
            high = mid - 1;
        }
    }

    // dummy return statement:
    return -1;
}

int main()
{
    vector<int> arr = {1, 1, 2, 2, 3, 3, 4, 5, 5, 6, 6};
    int ans = singleNonDuplicate(arr);
    cout << "The single element is: " << ans << "\n";
    return 0;
}
```

Output: The single element is: 4

▼ Complexity Analysis ›

Time Complexity: O(logN), N = size of the given array.
Reason: We are basically using the Binary Search algorithm.

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

▶ Video Explanation ›

> *Special thanks to **KRITIDIPTA GHOSH** for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam,* please check out this article

Peak element in Array

takeuforward.org/data-structure/peak-element-in-array

March 12, 2022

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.

Note: *For the first element, the previous element should be considered as negative infinity as well as for the last element, the next element should be considered as negative infinity.*

Pre-requisite: Binary Search Algorithm

▼ **Examples** ›

```
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.
```

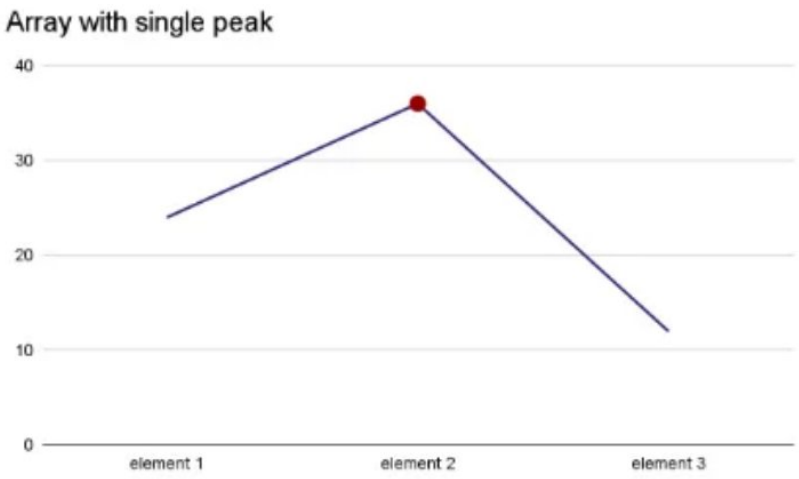
Solution:

What is a peak element?

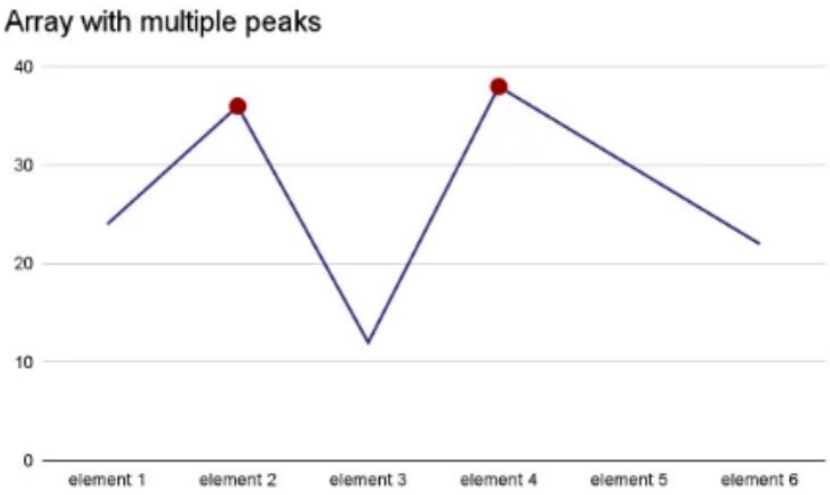
A peak element in an array refers to the element that is greater than both of its neighbors. Basically, if arr[i] is the peak element, arr[i] > arr[i-1] and arr[i] > arr[i+1].

Now if we want to visualize an array with the peak elements from the graphical point of view, it must be one of the following:

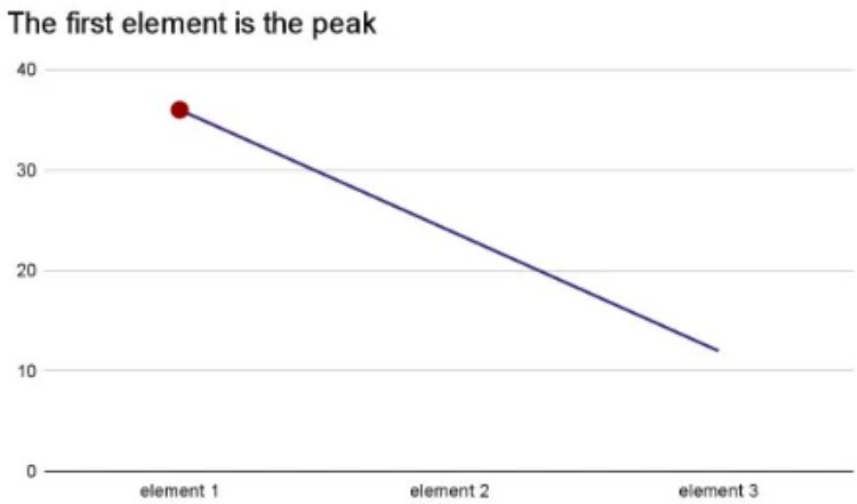
An array with a single peak number:



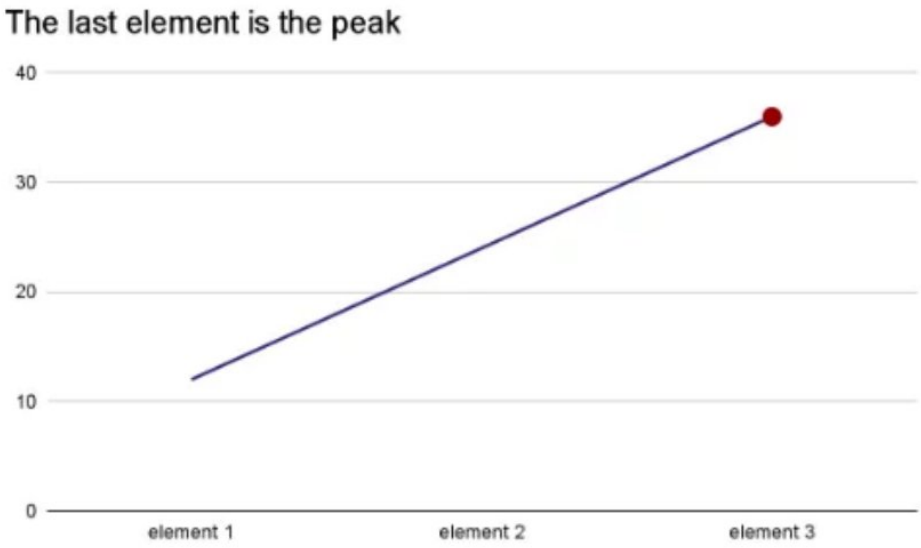
An array with multiple peaks:



An array where the first element is the peak:



An array where the last element is the peak:



Note: In all the above images, the peak is marked with a red circle.

How to identify if an element arr[i] is a peak:

We know that if arr[i] is the peak, **arr[i] > arr[i-1] and arr[i] > arr[i+1]**. So, we can check this condition for all the elements and identify the peak. But there are the following edge cases:

- **If n == 1:** The aforementioned condition will not be applicable. In this scenario, when the array size is 1, the single element within the array serves as the peak, and thus we should return its index. Prior information specifies that for the first element, we should treat the previous element as negative infinity, and similarly, for the last element, we should consider the next element as negative infinity.

- **If i == 0:** The aforementioned condition will not be applicable as arr[i-1] will refer to arr[-1] which is invalid. So, in this case, we should check if **arr[0] > arr[1].** If this condition holds, we can conclude that arr[0] is a peak. Prior information specifies that for the first element, we should treat the previous element as negative infinity.
- **If i == n-1:** The aforementioned condition will not be applicable as arr[i+1] will refer to arr[n] which is again invalid. So, in this case, we should check if **arr[n-1] > arr[n-2].** If this condition holds, we can conclude that arr[n-1] is a peak. Prior information specifies that for the last element, we should treat the next element as negative infinity.

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

Brute Force Approach Optimal Approach

▼ Brute Force Approach ›
▼ Algorithm / Intuition ›

Naive 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.

Algorithm:

- We will start traversing the array and for every index, we will check the below condition.
- **If((i == 0 or arr[i-1] < arr[i]) and (i == n-1 or arr[i] > arr[i+1])):** whenever this condition is true for an element, we should return its index.

Dry-run: *Please refer to the video for the dry-run.*

also update the 'index' variable with the corresponding index value, 'i'.
- Finally, we will return 'index' as our answer.

▼ Code ›

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

int findPeakElement(vector<int> &arr) {
    int n = arr.size(); //Size of array.

    for (int i = 0; i < n; i++) {
        //Checking for the peak:
        if ((i == 0 || arr[i - 1] < arr[i])
                && (i == n - 1 || arr[i] > arr[i + 1])) {
            return i;
        }
    }
    // Dummy return statement
    return -1;
}

int main()
{
    vector<int> arr = {1, 2, 3, 4, 5, 6, 7, 8, 5, 1};
    int ans = findPeakElement(arr);
    cout << "The peak is at index: " << ans << "\n";
    return 0;
}
```

Output: The peak is at index: 7

▼ Complexity Analysis ›
Time Complexity: O(N), N = size of the given array.
Reason: We are traversing the entire array.

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

▼ Optimal Approach ›
▼ Algorithm / Intuition ›

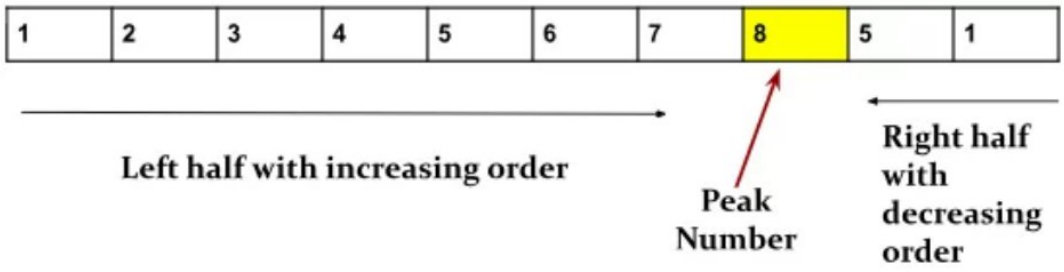
Optimal Approach(Using Binary Search):

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

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

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]**.

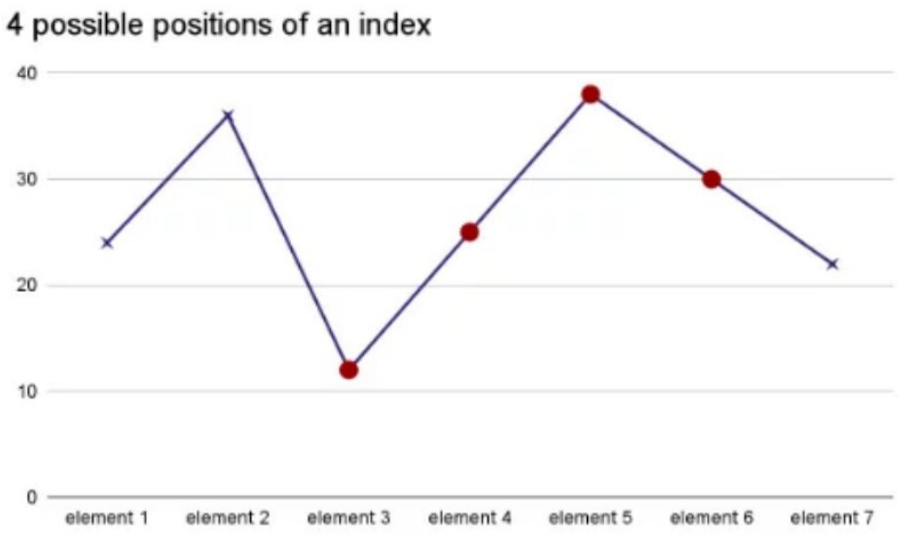
Now, using the above observation, we can easily identify the left and right halves, just by checking the property of the current index, i, like the following:

1. **If arr[i] > arr[i-1]:** we are in the left half.
2. **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:

1. **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.
2. **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.

3. **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.
4. **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.
5. **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)
6. **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'.
7. **If arr[mid] > arr[mid-1]:** This meanswe 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.
8. **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.

Dry-run: *Please refer to the video for the dry-run.*

▼ Code ›

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

int findPeakElement(vector<int> &arr) {
    int n = arr.size(); //Size of array.

    // Edge cases:
    if (n == 1) return 0;
    if (arr[0] > arr[1]) return 0;
    if (arr[n - 1] > arr[n - 2]) return n - 1;

    int low = 1, high = n - 2;
    while (low <= high) {
        int mid = (low + high) / 2;

        //If arr[mid] is the peak:
        if (arr[mid - 1] < arr[mid] && arr[mid] > arr[mid + 1])
            return mid;

        // If we are in the left:
        if (arr[mid] > arr[mid - 1]) low = mid + 1;

        // If we are in the right:
        // Or, arr[mid] is a common point:
        else high = mid - 1;
    }
    // Dummy return statement
    return -1;
}

int main()
{
    vector<int> arr = {1, 2, 3, 4, 5, 6, 7, 8, 5, 1};
    int ans = findPeakElement(arr);
    cout << "The peak is at index: " << ans << "\n";
    return 0;
}
```

Output: The peak is at index: 7

▼ Complexity Analysis ›

Time Complexity: O(logN), N = size of the given array.
Reason: We are basically using binary search to find the minimum.

Space Complexity: O(1)
Reason: We have not used any extra data structures, this makes space complexity, even in the worst case as O(1).

▶ Video Explanation ›

> *Special thanks to **KRITIDIPTA GHOSH** for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam, please check out this article*

Finding Sqrt of a number using Binary Search

takeuforward.org/binary-search/finding-sqrt-of-a-number-using-binary-search

July 3, 2023

Problem Statement: You are given a positive integer n. Your task is to find and return its square root. If ‘n’ is not a perfect square, then return the floor value of ‘sqrt(n)’.

Note: *The question explicitly states that if the given number, n, is not a perfect square, our objective is to find the maximum number, x, such that x squared is less than or equal to n (x*x <= n). In other words, we need to determine the floor value of the square root of n.*

▼ Examples ›

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.

Practice:

Solve Problem

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

Brute Force Approach Optimal Approach 1 Optimal Approach 2

▼ Brute Force Approach >

▼ Algorithm / Intuition >

Naive Approach(Using linear search):

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.

Algorithm:

- We will first declare a variable called 'ans'.
- Then, we will first run a loop(***say i***) from 1 to n.
- Until the value i*i <= n, we will update the variable 'ans', with i.
- Once, the value i*i becomes greater than n, we will break out from the loop as the current number i, or the numbers greater than i, cannot be our answers.
- Finally, our answer should have been stored in 'ans'.

▼ Code >

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

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

int main()
{
    int n = 28;
    int ans = floorSqrt(n);
    cout << "The floor of square root of " << n
         << " is: " << ans << "\n";
    return 0;
}
```

Output: The floor of square root of 28 is: 5

▼ Complexity Analysis ›
Time Complexity: O(N), N = the given number.
Reason: Since we are using linear search, we traverse the entire answer space.

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

▼ Optimal Approach 1: ›
▼ Algorithm / Intuition ›

First Approach(Using in-built sqrt() function):

A straightforward solution to this problem is to utilize the built-in sqrt() function. This approach doesn't require any code implementation but serves as one of the possible solutions.

▼ Code ›

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

int floorSqrt(int n) {
    int ans = sqrt(n);
    return ans;
}

int main()
{
    int n = 28;
    int ans = floorSqrt(n);
    cout << "The floor of square root of " << n
         << " is: " << ans;
    return 0;
}
```

Output: Output: The floor of square root of 28 is: 5

▼ Complexity Analysis ›
Time Complexity: O(logN), N = size of the given array.
Reason: We are basically using the Binary Search algorithm.

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

▼ Optimal Approach 2: ›
▼ Algorithm / Intuition ›

Optimal Approach(Using binary search):

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

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

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.

Algorithm:

The steps are as follows:

We will declare a variable called ‘ans’.

1. **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 n.
2. **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)
3. **Eliminate the halves accordingly:**
   1. **If mid*mid <= n:** On satisfying this condition, we can conclude that the number 'mid' is one of the possible answers. So, we will store 'mid' in the variable 'ans'. But we want the maximum number that holds this condition. So, we will eliminate the left half and consider the right half(i.e. low = mid+1).
   2. **Otherwise,** the value mid is larger than the number we want. This means the numbers greater than 'mid' will not be our answers and the right half of 'mid' consists of such numbers. So, we will eliminate the right half and consider the left half(i.e. high = mid-1).
4. Finally, the 'ans' variable will be storing our answer. In addition to that, the high pointer will also point to the same number i.e. our answer. So, we can return either of the 'ans' or 'high'.

The steps from 2-3 will be inside a loop and the loop will continue until low crosses high.

Dry-run: *Please refer to the video for the dry-run.*

Note: *In this case, the 'high' pointer serves as our answer, eliminating the need for an additional 'ans' variable. Therefore, it is perfectly acceptable to omit the use of an extra variable to store the answer. Consequently, in the following code, no additional variable is utilized for storing the answer.*

▼ Code ›

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

int floorSqrt(int n) {
    int low = 1, high = n;
    //Binary search on the answers:
    while (low <= high) {
        long long mid = (low + high) / 2;
        long long val = mid * mid;
        if (val <= (long long)(n)) {
            //eliminate the left half:
            low = mid + 1;
        }
        else {
            //eliminate the right half:
            high = mid - 1;
        }
    }
    return high;
}

int main()
{
    int n = 28;
    int ans = floorSqrt(n);
    cout << "The floor of square root of " << n
         << " is: " << ans << "\n";
    return 0;
}
```

Output: The floor of square root of 28 is: 5

▼ Complexity Analysis ›

Time Complexity: O(logN), N = size of the given array.
Reason: We are basically using the Binary Search algorithm.

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

▶ Video Explanation ›

> *Special thanks to **KRITIDIPTA GHOSH** for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam,* please check out this article

Nth Root of a Number using Binary Search

takeuforward.org/data-structure/nth-root-of-a-number-using-binary-search

December 1, 2021

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.

▼ **Examples** ›

```
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.
```

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

Brute Force Approach Optimal Approach

▼ Brute Force Approach ›
▼ Algorithm / Intuition ›

Solution:

Approach(Using linear search):

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. x^n.

Algorithm:

- We will first run a loop(***say i***) from 1 to m.
- **Inside the loop we will check the following:**
  - **If func(x, n) == m:** This means x is the number we are looking for. So, we will return x from this step.
  - **If func(x, n) > m:** This means we have got a bigger number than our answer and until now we have not found any number that can be our answer. In this case, our answer does not exist and we will break out from this step and return -1.

Note: We will use the power exponential method to implement the func() function and its time complexity will be O(logN)(*where N = given exponent*).

Dry-run: *Please refer to the video for the dry-run.*

▼ Code ›

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

// Power exponential method:
long long func(int b, int exp) {
    long long  ans = 1;
    long long base = b;
    while (exp > 0) {
        if (exp % 2) {
            exp--;
            ans = ans * base;
        }
        else {
            exp /= 2;
            base = base * base;
        }
    }
    return ans;
}

int NthRoot(int n, int m) {
    //Use linear search on the answer space:
    for (int i = 1; i <= m; i++) {
        long long val = func(i, n);
        if (val == m * 1ll) return i;
        else if (val > m * 1ll) break;
    }
    return -1;
}

int main()
{
    int n = 3, m = 27;
    int ans = NthRoot(n, m);
    cout << "The answer is: " << ans << "\n";
    return 0;
}
```

Output: The answer is: 3

▼ Complexity Analysis ›

Time Complexity: O(M), M = the given number.
Reason: Since we are using linear search, we traverse the entire answer space.

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

▼ Optimal Approach ›

▼ Algorithm / Intuition ›

Optimal Approach(Using Binary Search):

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

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

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 mid^n 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 mid^n.
- 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 equ**al 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.

Algorithm:

1. **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 m.
2. **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)

3. **Eliminate the halves accordingly:**
    1. **If func(n, m, mid) == 1:** On satisfying this condition, we can conclude that the number ‘mid’ is our answer. So, we will return to ‘mid’.
    2. **If func(n, m, mid) == 0:** On satisfying this condition, we can conclude that the number ‘mid’ is smaller than our answer. So, we will eliminate the left half and consider the right half(i.e. low = mid+1).
    3. **If func(n, m, mid) == 2:** the value mid is larger than the number we want. This means the numbers greater than ‘mid’ will not be our answers and the right half of ‘mid’ consists of such numbers. So, we will eliminate the right half and consider the left half(i.e. high = mid-1).
4. Finally, if we are outside the loop, this means no answer exists. So, we will return -1.

The steps from 2-3 will be inside a loop and the loop will continue until low crosses high.

Dry-run: *Please refer to the video for the dry-run.*

▼ Code ›

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

//return 1, if == m:
//return 0, if < m:
//return 2, if > m:
int func(int mid, int n, int m) {
    long long ans = 1;
    for (int i = 1; i <= n; i++) {
        ans = ans * mid;
        if (ans > m) return 2;
    }
    if (ans == m) return 1;
    return 0;
}

int NthRoot(int n, int m) {
    //Use Binary search on the answer space:
    int low = 1, high = m;
    while (low <= high) {
        int mid = (low + high) / 2;
        int midN = func(mid, n, m);
        if (midN == 1) {
            return mid;
        }
        else if (midN == 0) low = mid + 1;
        else high = mid - 1;
    }
    return -1;
}

int main()
{
    int n = 3, m = 27;
    int ans = NthRoot(n, m);
    cout << "The answer is: " << ans << "\n";
    return 0;
}
```

Output: The peak is at index: 7

▼ Complexity Analysis ›

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

Reason: We are basically using binary search to find the minimum.

Space Complexity: O(1)
Reason: We have not used any extra data structures, this makes space complexity, even in the worst case as O(1).

▶ Video Explanation ›

> *Special thanks to **KRITIDIPTA GHOSH** for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam,* please check out this article

Koko Eating Bananas

takeuforward.org/binary-search/koko-eating-bananas

July 3, 2023

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.

▼ **Examples** ›

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.

Practice:

Solve Problem

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

Brute Force Approach Optimal Approach

▼ Brute Force Approach >
▼ Algorithm / Intuition >

Naive Approach(Brute-force):

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:

1. First, we will find the maximum value i.e. **max(a[])** in the given array.
2. We will run a loop(**say i**) from 1 to max(a[]), to check all possible answers.
3. 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.
4. 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.

1. We will iterate every pile of the given array using a loop(**say i**).
2. For every pile i, we will calculate the hour i.e. ceil(v[i] / hourly), and add it to the total hours.
3. Finally, we will return the total hours.

Dry-run: *Please refer to the video for the dry-run.*

▼ Code >

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

int findMax(vector<int> &v) {
    int maxi = INT_MIN;
    int n = v.size();
    //find the maximum:
    for (int i = 0; i < n; i++) {
        maxi = max(maxi, v[i]);
    }
    return maxi;
}

int calculateTotalHours(vector<int> &v, int hourly) {
    int totalH = 0;
    int n = v.size();
    //find total hours:
    for (int i = 0; i < n; i++) {
        totalH += ceil((double)(v[i]) / (double)(hourly));
    }
    return totalH;
}

int minimumRateToEatBananas(vector<int> v, int h) {
    //Find the maximum number:
    int maxi = findMax(v);

    //Find the minimum value of k:
    for (int i = 1; i <= maxi; i++) {
        int reqTime = calculateTotalHours(v, i);
        if (reqTime <= h) {
            return i;
        }
    }

    //dummy return statement
    return maxi;
}

int main()
{
    vector<int> v = {7, 15, 6, 3};
    int h = 8;
    int ans = minimumRateToEatBananas(v, h);
    cout << "Koko should eat atleast " << ans << " bananas/hr.\n";
    return 0;
}
```

Output: Koko should eat atleast 5 bananas/hr.

▼ Complexity Analysis ›
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 ›
▼ Algorithm / Intuition ›

Optimal Approach(Using Binary Search):

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

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

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:

1. **First,** we will find the maximum element in the given array i.e. max(a[]).
2. **Place the 2 pointers i.e. low and high:** Initially, we will place the pointers. The pointer low will point to 1 and the high will point to max(a[]).
3. **Calculate the 'mid':** Now, inside the loop, we will calculate the value of 'mid' using the following formula:
   mid = (low+high) // 2 ('//' refers to integer division)
4. **Eliminate the halves based on the 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):**
    1. **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).
    2. **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).

5. 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.

1. We will iterate every pile of the given array using a loop(**say i**).
2. For every pile i, we will calculate the hour i.e. ceil(v[i] / hourly), and add it to the total hours.
3. Finally, we will return the total hours.

Dry-run: *Please refer to the video for the dry-run.*

▼ Code ›

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

int findMax(vector<int> &v) {
    int maxi = INT_MIN;
    int n = v.size();
    //find the maximum:
    for (int i = 0; i < n; i++) {
        maxi = max(maxi, v[i]);
    }
    return maxi;
}

int calculateTotalHours(vector<int> &v, int hourly) {
    int totalH = 0;
    int n = v.size();
    //find total hours:
    for (int i = 0; i < n; i++) {
        totalH += ceil((double)(v[i]) / (double)(hourly));
    }
    return totalH;
}

int minimumRateToEatBananas(vector<int> v, int h) {
    int low = 1, high = findMax(v);

    //apply binary search:
    while (low <= high) {
        int mid = (low + high) / 2;
        int totalH = calculateTotalHours(v, mid);
        if (totalH <= h) {
            high = mid - 1;
        }
        else {
            low = mid + 1;
        }
    }
    return low;
}

int main()
{
    vector<int> v = {7, 15, 6, 3};
    int h = 8;
    int ans = minimumRateToEatBananas(v, h);
    cout << "Koko should eat atleast " << ans << " bananas/hr.\n";
    return 0;
}
```

Output: Koko should eat atleast 5 bananas/hr.

▼ Complexity Analysis ›

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

Reason: We are applying Binary search for the range [1, max(a[])], and for every value of 'mid', we are traversing the entire array inside the function named **calculateTotalHours().**

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

▶ Video Explanation ›

> *Special thanks to **KRITIDIPTA GHOSH** for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam,* please check out this article

Minimum days to make M bouquets

takeuforward.org/arrays/minimum-days-to-make-m-bouquets

July 3, 2023

Problem Statement: You are given ‘N’ roses and you are also given an array ‘arr’ where ‘arr[i]’ denotes that the ‘ith’ rose will bloom on the ‘arr[i]th’ day.
You can only pick already bloomed roses that are adjacent to make a bouquet. You are also told that you require exactly ‘k’ adjacent bloomed roses to make a single bouquet.
Find the minimum number of days required to make at least ‘m’ bouquets each containing ‘k’ roses. Return -1 if it is not possible.

▼ **Examples** ›

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.

Now, we will write a function **possible(),** that will return true if, on a certain day, we can make at least m bouquets otherwise it will return false. The steps will be the following:

possible(arr[], day, m, k) algorithm:

1. We will declare two variables i.e. 'cnt' and 'noOfB'.
   cnt -> the number of adjacent flowers,
   noOfB -> the number of bouquets.
2. We will run a loop to traverse the array.

3. Inside the loop, we will do the following:
    1. **If arr[i] <= day:** This means the ith flower has bloomed. So, we will increase the number of adjacent flowers i.e. 'cnt' by 1.
    2. **Otherwise,** the flower has not bloomed. Here, we will calculate the number of bouquets we can make with 'cnt' adjacent flowers i.e. floor(cnt/k), and add it to the noOfB. Now, as this ith flower breaks the sequence of the adjacent bloomed flowers, we will set the 'cnt' 0.
4. Lastly, outside the loop, we will calculate the floor(cnt/k) and add it to the noOfB.
5. **If noOfB >= m:** This means, we can make at least m bouquets. So, we will return true.
6. **Otherwise,** We will return false.

Note: *We actually pass a particular day as a parameter to the possible() function. The function returns true if it is possible to make atleast m bouquets on that particular day, otherwise, it returns false.*

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

Brute Force Approach Optimal Approach

▼ Brute Force Approach ›
▼ Algorithm / Intuition ›

Naive Approach(Brute force):

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.

Algorithm:

1. **If m*k > arr.size:** This means we have insufficient flowers. So, it is impossible to make m bouquets and we will return -1.
2. We will run a loop(**say i**) from min(arr[]) to max(arr[]) to check all possible answers.
3. Next, we will pass each potential answer, represented by the variable 'i' (which corresponds to a specific day), to the 'possible()' function. If the function returns true, indicating that we can create 'm' bouquets, we will return the value of 'i'.
4. Finally, if we are outside the loop, we can conclude that is impossible to make m bouquets. So, we will return -1.

Dry-run: *Please refer to the video for the dry-run.*

▼ Code ›

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

bool possible(vector<int> &arr, int day, int m, int k) {
    int n = arr.size(); //size of the array
    int cnt = 0;
    int noOfB = 0;
    // count the number of bouquets:
    for (int i = 0; i < n; i++) {
        if (arr[i] <= day) {
            cnt++;
        }
        else {
            noOfB += (cnt / k);
            cnt = 0;
        }
    }
    noOfB += (cnt / k);
    return noOfB >= m;
}
int roseGarden(vector<int> arr, int k, int m) {
    long long val = m * 1ll * k * 1ll;
    int n = arr.size(); //size of the array
    if (val > n) return -1; //impossible case.
    //find maximum and minimum:
    int mini = INT_MAX, maxi = INT_MIN;
    for (int i = 0; i < n; i++) {
        mini = min(mini, arr[i]);
        maxi = max(maxi, arr[i]);
    }

    for (int i = mini; i <= maxi; i++) {
        if (possible(arr, i, m, k))
            return i;
    }
    return -1;
}

int main()
{
    vector<int> arr = {7, 7, 7, 7, 13, 11, 12, 7};
    int k = 3;
    int m = 2;
    int ans = roseGarden(arr, k, m);
    if (ans == -1)
        cout << "We cannot make m bouquets.\n";
    else
        cout << "We can make bouquets on day " << ans << "\n";
    return 0;
```

```
}
```

Output: We can make bouquets on day 12.

▼ Complexity Analysis ›
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 ›
▼ Algorithm / Intuition ›

Optimal Approach(Using Binary Search):

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

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

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.

Algorithm:

1. **If m*k > arr.size:** This means we have insufficient flowers. So, it is impossible to make m bouquets and we will return -1.
2. **Next,** we will find the maximum element i.e. max(arr[]), and the minimum element i.e. min(arr[]) in the array.
3. **Place the 2 pointers i.e. low and high:** Initially, we will place the pointers. The pointer low will point to min(arr[]) and the high will point to max(arr[]).
4. **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)

5. **Eliminate the halves based on the value returned by possible():**
   We will pass the potential answer, represented by the variable 'mid' (which corresponds to a specific day), to the 'possible()' function.
   1. **If possible() returns true:** 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).
   2. **Otherwise,** the value mid is smaller than the number we want. 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).
6. Finally, outside the loop, we will return the value of low as the pointer will be pointing to the answer.

The steps from 3-5 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.*

Dry-run: *Please refer to the video for the dry-run.*

▼ Code ›

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

bool possible(vector<int> &arr, int day, int m, int k) {
    int n = arr.size(); //size of the array
    int cnt = 0;
    int noOfB = 0;
    // count the number of bouquets:
    for (int i = 0; i < n; i++) {
        if (arr[i] <= day) {
            cnt++;
        }
        else {
            noOfB += (cnt / k);
            cnt = 0;
        }
    }
    noOfB += (cnt / k);
    return noOfB >= m;
}
int roseGarden(vector<int> arr, int k, int m) {
    long long val = m * 1ll * k * 1ll;
    int n = arr.size(); //size of the array
    if (val > n) return -1; //impossible case.
    //find maximum and minimum:
    int mini = INT_MAX, maxi = INT_MIN;
    for (int i = 0; i < n; i++) {
        mini = min(mini, arr[i]);
        maxi = max(maxi, arr[i]);
    }

    //apply binary search:
    int low = mini, high = maxi;
    while (low <= high) {
        int mid = (low + high) / 2;
        if (possible(arr, mid, m, k)) {
            high = mid - 1;
        }
        else low = mid + 1;
    }
    return low;
}

int main()
{
    vector<int> arr = {7, 7, 7, 7, 13, 11, 12, 7};
    int k = 3;
    int m = 2;
    int ans = roseGarden(arr, k, m);
```

```
    if (ans == -1)
        cout << "We cannot make m bouquets.\n";
    else
        cout << "We can make bouquets on day " << ans << "\n";
    return 0;
}
```

Output: We can make bouquets on day 12

▼ Complexity Analysis ›

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}.
Reason: We are applying binary search on our answers that are in the range of [min(arr[]), max(arr[])]. For every possible answer 'mid', 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.

▶ Video Explanation ›

> *Special thanks to **KRITIDIPTA GHOSH** for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam,* please check out this article

Find the Smallest Divisor Given a Threshold

takeuforward.org/arrays/find-the-smallest-divisor-given-a-threshold

July 3, 2023

Problem Statement: You are given an array of integers ‘arr’ and an integer i.e. a threshold value ‘limit’. Your task is to find the smallest positive integer divisor, such that upon dividing all the elements of the given array by it, the sum of the division’s result is less than or equal to the given threshold value.

▼ **Examples** ›

Example 1:
Input Format: N = 5, arr[] = {1,2,3,4,5}, limit = 8
Result: 3
Explanation: We can get a sum of 15(1 + 2 + 3 + 4 + 5) if we choose 1 as a divisor.
The sum is 9(1 + 1 + 2 + 2 + 3) if we choose 2 as a divisor. Upon dividing all the elements of the array by 3, we get 1,1,1,2,2 respectively. Now, their sum is equal to 7 <= 8 i.e. the threshold value. So, 3 is the minimum possible answer.

Example 2:
Input Format: N = 4, arr[] = {8,4,2,3}, limit = 10
Result: 2
Explanation: If we choose 1, we get 17 as the sum. If we choose 2, we get 9(4+2+1+2) <= 10 as the answer. So, 2 is the answer.

Point to remember:

> While dividing the array elements with a chosen number, we will always take the ceiling value. And then we will consider their summation. For example, 3 / 2 = 2.

Observation:

- **Minimum possible divisor:** We can easily consider 1 as the minimum divisor as it is the smallest positive integer.
- **Maximum possible divisor:** If we observe, we can conclude the maximum element in the array i.e. max(arr[]) is the maximum possible divisor. Any number > max(arr[]), will give the exact same result as max(arr[]) does. This divisor will generate the minimum possible result i.e. n(*1 for each element*), where n = size of the array.

With these observations, ***we can surely say that our answer will lie in the range [1, max(arr[])]***.

Practice:

Solve Problem

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

Brute Force Approach Optimal Approach

▼ Brute Force Approach ›

▼ Algorithm / Intuition ›

Naive Approach(Brute-force):

The extremely naive approach is to check all possible divisors from 1 to max(arr[]). The minimum number for which the result <= threshold value, will be our answer.

Algorithm:

- We will run a loop(say **d**) from 1 to max(arr[]) to check all possible divisors.
- To calculate the result, we will iterate over the given array using a loop. Within this loop, we will divide each element in the array by the current divisor, d, and sum up the obtained ceiling values.
- Inside the outer loop, **If result <= threshold:** We will return d as our answer.
- Finally, if we are outside the nested loops, we will return -1.

Dry-run: *Please refer to the video for the dry-run.*

▼ Code ›

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

int smallestDivisor(vector<int>& arr, int limit) {
    int n = arr.size(); //size of array.
    //Find the maximum element:
    int maxi = *max_element(arr.begin(), arr.end());

    //Find the smallest divisor:
    for (int d = 1; d <= maxi; d++) {
        //Find the summation result:
        int sum = 0;
        for (int i = 0; i < n; i++) {
            sum += ceil((double)(arr[i]) / (double)(d));
        }
        if (sum <= limit)
            return d;
    }
    return -1;
}

int main()
{
    vector<int> arr = {1, 2, 3, 4, 5};
    int limit = 8;
    int ans = smallestDivisor(arr, limit);
    cout << "The minimum divisor is: " << ans << "\n";
    return 0;
}
```

Output: The minimum divisor is: 3

▼ Complexity Analysis ›
Time Complexity: O(max(arr[])*N), where max(arr[]) = maximum element in the array, N = size of the array.
Reason: We are using nested loops. The outer loop runs from 1 to max(arr[]) 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 ›
▼ Algorithm / Intuition ›

Optimal Approach(Using Binary Search):

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

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

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 [1, max(arr[])], 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.

Algorithm:

1. **If n > threshold:** If the minimum summation i.e. n > threshold value, the answer does not exist. In this case, we will return -1.
2. **Next,** we will find the maximum element i.e. max(arr[]) in the given array.
3. **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(arr[]).
4. **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)

5. **Eliminate the halves based on the summation of division results:**
   We will pass the potential divisor, represented by the variable 'mid', to the '**sumByD()**' function. This function will return the summation result of the division values.
   1. **If result <= threshold:** 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).
   2. **Otherwise,** the value mid is smaller than the number we want. 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).
6. Finally, outside the loop, we will return the value of low as the pointer will be pointing to the answer.

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

Note: *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.*

The algorithm for **sumByD()** is given below:

sumByD(arr[], div):

arr[] -> the given array, div -> the divisor.

1. We will run a loop to iterate over the array.
2. We will divide each element by 'div', and consider the ceiling value.
3. With that, we will sum up the ceiling values as well.
4. Finally, we will return the summation.

Dry-run: *Please refer to the video for the dry-run.*

▼ Code ›

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

int sumByD(vector<int> &arr, int div) {
    int n = arr.size(); //size of array
    //Find the summation of division values:
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum += ceil((double)(arr[i]) / (double)(div));
    }
    return sum;
}

int smallestDivisor(vector<int>& arr, int limit) {
    int n = arr.size();
    if (n > limit) return -1;
    int low = 1, high = *max_element(arr.begin(), arr.end());

    //Apply binary search:
    while (low <= high) {
        int mid = (low + high) / 2;
        if (sumByD(arr, mid) <= limit) {
            high = mid - 1;
        }
        else {
            low = mid + 1;
        }
    }
    return low;
}

int main()
{
    vector<int> arr = {1, 2, 3, 4, 5};
    int limit = 8;
    int ans = smallestDivisor(arr, limit);
    cout << "The minimum divisor is: " << ans << "\n";
    return 0;
}
```

Output: The minimum divisor is: 3

▼ Complexity Analysis ›

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

Reason: We are applying binary search on our answers that are in the range of [1,

max(arr[])]. For every possible divisor ‘mid’, we call the sumByD() function. Inside that 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.

▶ Video Explanation ›

> *Special thanks to **KRITIDIPTA GHOSH** for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam,* please check out this article

Capacity to Ship Packages within D Days

takeuforward.org/arrays/capacity-to-ship-packages-within-d-days

July 5, 2023

Problem Statement: You are the owner of a Shipment company. You use conveyor belts to ship packages from one port to another. The packages must be shipped within ‘d’ days.
The weights of the packages are given in an array ‘of weights’. The packages are loaded on the conveyor belts every day in the same order as they appear in the array. The loaded weights must not exceed the maximum weight capacity of the ship.
Find out the least-weight capacity so that you can ship all the packages within ‘d’ days.

▼ **Examples** ›

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):

1. 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.
2. **Next,** we will use a loop(***say i***) to iterate over the weights. For each weight, weights[i], we will check the following:
    1. **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]).
    2. **Otherwise,** We will just add the current weight to the load(i.e. load = load+weights[i]).
3. Finally, we will return 'days' which represents the number of days required.

Practice:

Solve Problem

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

Brute Force Approach Optimal Approach

▼ Brute Force Approach >

▼ Algorithm / Intuition >

Naive 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.

Algorithm:

1. We will use a loop(say **cap**) to check all possible capacities.
2. Next, inside the loop, we will send each capacity to the **findDays()** function to get the number of days required for that particular capacity.
3. The minimum number, for which the number of days <= d, will be the answer.

Dry-run: *Please refer to the video for the dry-run.*

▼ Code >

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

int findDays(vector<int> &weights, int cap) {
    int days = 1; //First day.
    int load = 0;
    int n = weights.size(); //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;
}

int leastWeightCapacity(vector<int> &weights, int d) {
    //Find the maximum and the summation:
    int maxi = *max_element(weights.begin(), weights.end());
    int sum = accumulate(weights.begin(), weights.end(), 0);

    for (int i = maxi; i <= sum; i++) {
        if (findDays(weights, i) <= d) {
            return i;
        }
    }
    //dummy return statement:
    return -1;
}

int main()
{
    vector<int> weights = {5, 4, 5, 2, 3, 4, 5, 6};
    int d = 5;
    int ans = leastWeightCapacity(weights, d);
    cout << "The minimum capacity should be: " << ans << "\n";
    return 0;
}
```

Output: The minimum capacity should be: 9.

▼ Complexity Analysis ›

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.
Reason: We are using a loop from max(weights[]) to sum(weights[]) to check all possible weights. Inside the loop, we are calling findDays() function for each weight. 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.

▼ Optimal Approach ›
▼ Algorithm / Intuition ›

Optimal Approach(Using Binary Search):

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

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

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.

Algorithm:

1. **First,** we will find the maximum element i.e. max(weights[]), and the summation i.e. sum(weights[]) of the given array.
2. **Place the 2 pointers i.e. low and high:** Initially, we will place the pointers. The pointer low will point to max(weights[]) and the high will point to sum(weights[]).
3. **Calculate the 'mid':** Now, inside the loop, we will calculate the value of 'mid' using the following formula:
   mid = (low+high) // 2 ('//' refers to integer division)

4. **Eliminate the halves based on the number of days required for the capacity 'mid':**
   We will pass the potential capacity, represented by the variable 'mid', to the '**findDays()**' function. This function will return the number of days required to ship all the weights for the particular capacity, 'mid'.
   1. **If munerOfDays <= d:** 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).
   2. **Otherwise,** the value mid is smaller than the number we want. 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).
5. Finally, outside the loop, we will return the value of low as the pointer will be pointing to the answer.

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

Note: *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.*

Dry-run: *Please refer to the video for the dry-run.*

▼ Code ›

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

int findDays(vector<int> &weights, int cap) {
    int days = 1; //First day.
    int load = 0;
    int n = weights.size(); //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;
}

int leastWeightCapacity(vector<int> &weights, int d) {
    //Find the maximum and the summation:
    int low = *max_element(weights.begin(), weights.end());
    int high = accumulate(weights.begin(), weights.end(), 0);
    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;
}

int main()
{
    vector<int> weights = {5, 4, 5, 2, 3, 4, 5, 6};
    int d = 5;
    int ans = leastWeightCapacity(weights, d);
    cout << "The minimum capacity should be: " << ans << "\n";
    return 0;
}
```

Output: The minimum capacity should be: 9.

▼ Complexity Analysis ›

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.

▶ Video Explanation ›

> *Special thanks to **KRITIDIPTA GHOSH** for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam,* please check out this article

Kth Missing Positive Number

takeuforward.org/arrays/kth-missing-positive-number

July 5, 2023

Problem Statement: You are given a strictly increasing array ‘vec’ and a positive integer ‘k’. Find the ‘kth’ positive integer missing from ‘vec’.

▼ **Examples** >

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.

Practice:
Solve Problem
Disclaimer: *Don’t jump directly to the solution, try it out yourself first.*

Brute Force Approach Optimal Approach

▼ Brute Force Approach ›
▼ Algorithm / Intuition ›

Naive Approach:

There might be many brute-force approaches to solve this problem. But we are going to use the following simple steps to solve the problem.

Algorithm:

- We will use a loop to traverse the array.
- Inside the loop,
  - **If vec[i] <= k:** we will simply increase the value of k by 1.
  - **Otherwise,** we will break out of the loop.
- Finally, we will return the value of k.

Note: *The main idea is to shift k by 1 step if the current element is smaller or equal to k. And whenever we get a number > k, we can conclude that k is the missing number.*

Dry-run: *Please refer to the video for the dry-run.*

▼ Code ›

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

int missingK(vector < int > vec, int n, int k) {
    for (int i = 0; i < n; i++) {
        if (vec[i] <= k) k++; //shifting k
        else break;
    }
    return k;
}


int main()
{
    vector<int> vec = {4, 7, 9, 10};
    int n = 4, k = 4;
    int ans = missingK(vec, n, k);
    cout << "The missing number is: " << ans << "\n";
    return 0;
}
```

Output: The missing number is: 5.

▼ Complexity Analysis ›
Time Complexity: O(N), N = size of the given array.
Reason: We are using a loop that traverses the entire given array in the worst case.

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

▼ Optimal Approach ›
▼ Algorithm / Intuition ›

Optimal Approach(Using Binary Search):

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

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

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.

Note: *Please make sure to refer to the video and try out some test cases of your own to understand, how the pointer 'high' will be always pointing to the preceding closest neighbor in this case.*

Algorithm:

1. **Place the 2 pointers i.e. low and high:** Initially, we will place the pointers. The pointer low will point to index 0 and the high will point to index n-1 i.e. the last index.
2. **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)

3. **Eliminate the halves based on the number of missing numbers up to index 'mid':** We will calculate the number of missing numbers using the above-said formula like this: missing_numbers = vec[mid] - (mid+1).
    1. **If missing_numbers < k:** On satisfying this condition, we can conclude that we are currently at a smaller index. But we want a larger index. So, we will eliminate the left half and consider the right half(i.e. low = mid+1).
    2. **Otherwise,** we have to consider smaller indices. So, we will eliminate the right half and consider the left half(i.e. high = mid-1).
4. Finally, when we are outside the loop, we will return the value of (k+high+1) i.e. the kth missing number.

The steps from 2-3 will be inside a loop and the loop will continue until low crosses high.

Dry-run: *Please refer to the video for the dry-run.*

▼ Code ›

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

int missingK(vector < int > vec, int n, int k) {
    int low = 0, high = n - 1;
    while (low <= high) {
        int mid = (low + high) / 2;
        int missing = vec[mid] - (mid + 1);
        if (missing < k) {
            low = mid + 1;
        }
        else {
            high = mid - 1;
        }
    }
    return k + high + 1;
}

int main()
{
    vector<int> vec = {4, 7, 9, 10};
    int n = 4, k = 4;
    int ans = missingK(vec, n, k);
    cout << "The missing number is: " << ans << "\n";
    return 0;
}
```

Output: Output: The missing number is: 5.

▼ Complexity Analysis ›

Time Complexity: O(logN), N = size of the given array.
Reason: We are using the simple binary search algorithm.

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

▶ Video Explanation ›

> *Special thanks to **KRITIDIPTA GHOSH** for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam,* please check out this article

Aggressive Cows : Detailed Solution

takeuforward.org/data-structure/aggressive-cows-detailed-solution

December 3, 2021

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.

▼ **Examples** ›

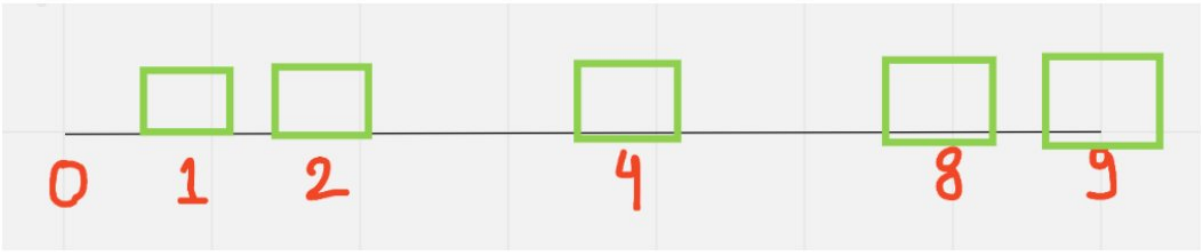
Example 1:
Input Format: N = 6, k = 4, arr[] = {0,3,4,7,10,9}
Result: 3
Explanation: The maximum possible minimum distance between any two cows will be 3 when 4 cows are placed at positions {0, 3, 7, 10}. Here the distances between cows are 3, 4, and 3 respectively. We cannot make the minimum distance 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}.

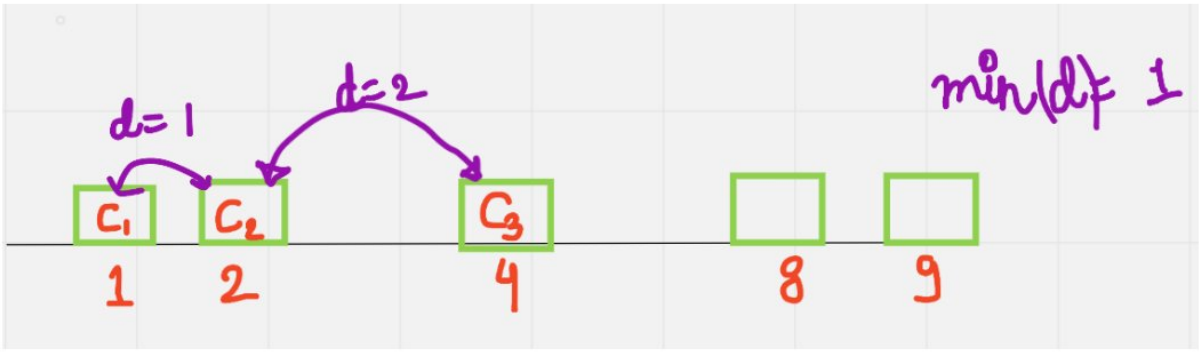
Why do we need to sort the stalls?

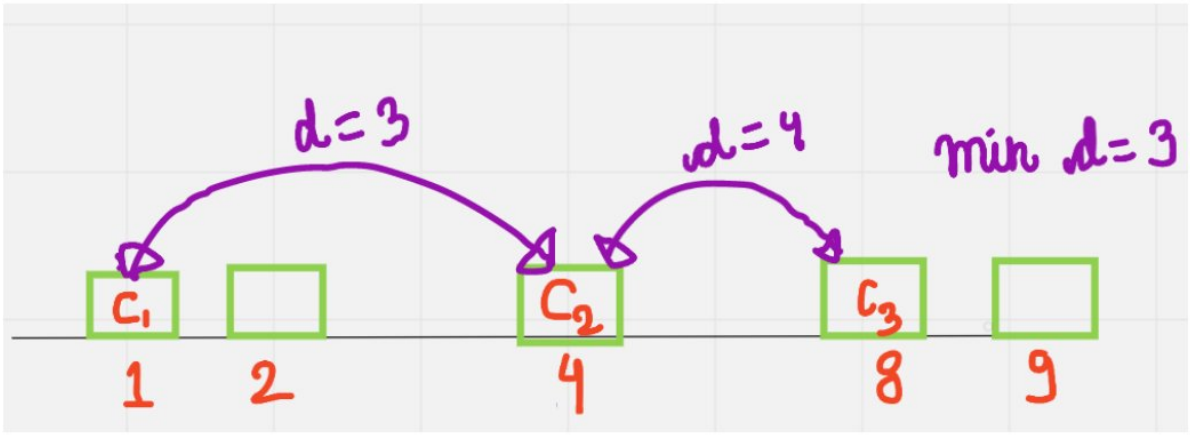
To arrange the cows in a consecutive manner while ensuring a certain distance between them, the initial step is to sort the stalls based on their positions. In a sorted array, the minimum distance will always be obtained from any two consecutive cows. Arranging the cows in a consecutive manner does not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

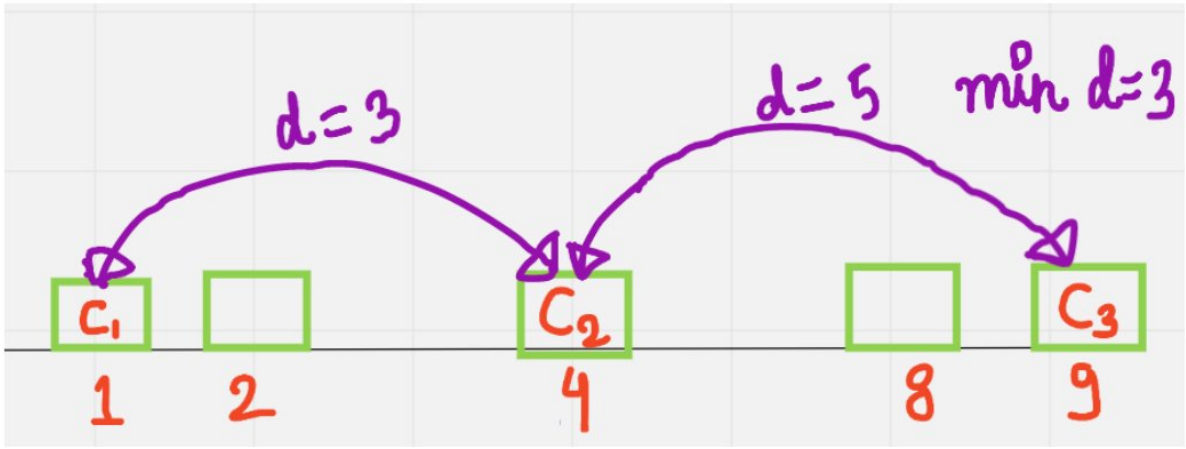
Assume the given stalls array is: {1,2,8,4,9} and after sorting it will be {1, 2, 4, 8, 9}. The given number of cows is 3.



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 arrarangements:







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

Observation:

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

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

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

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

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

canWePlace(stalls[], dist, k):

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

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

Brute Force Approach Optimal Approach

▼ Brute Force Approach ›
▼ Algorithm / Intuition ›

Naive Approach:

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

Algorithm:

1. First, we will sort the given stalls[] array.
2. We will use a loop(say **i**) to check all possible distances.
3. Next, inside the loop, we will send each distance, i, to the function **canWePlace()** function to check if it is possible to place the cows.
    1. We will return (i-1), where i is the minimum distance for which the function **canWePlace()** returns false. Because (i-1) is the maximum distance for which we can place all the cows and for the distances >= i, it becomes impossible.

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

Dry-run: *Please refer to the video for the dry-run.*

▼ Code ›

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

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

    int limit = stalls[n - 1] - stalls[0];
    for (int i = 1; i <= limit; i++) {
        if (canWePlace(stalls, i, k) == false) {
            return (i - 1);
        }
    }
    return limit;
}

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

Output: The maximum possible minimum distance is: 3.

▼ Complexity Analysis ›

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

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

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

▼ Optimal Approach ›

▼ Algorithm / Intuition ›

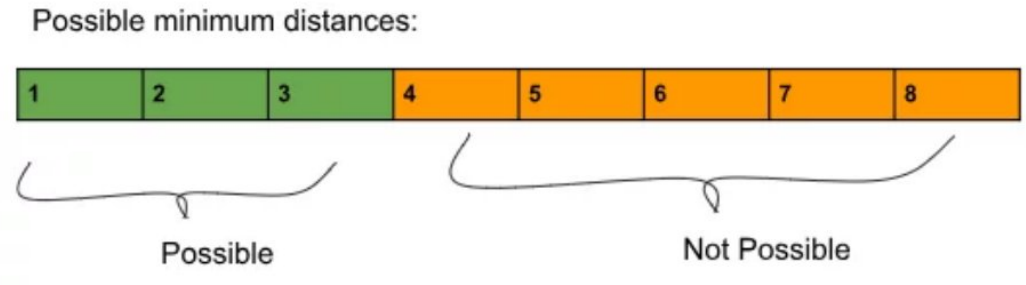
Optimal Approach(Using Binary Search):

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

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

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

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



Algorithm:

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

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

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

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

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

Dry-run: *Please refer to the video for the dry-run.*

▼ Code ›

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

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

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

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

Output: The maximum possible minimum distance is: 3.

▼ Complexity Analysis ›

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

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

▶ Video Explanation ›

> *Special thanks to **Harshit Garg**, **Sudip Ghosh** and **KRITIDIPTA GHOSH** for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam, please check out this article*

Allocate Minimum Number of Pages

takeuforward.org/data-structure/allocate-minimum-number-of-pages

December 2, 2021

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:

1. Each student gets at least one book.
2. Each book should be allocated to only one student.
3. 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

▼ **Examples** ›

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):

1. 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.

2. We will start traversing the given array.
3. **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.
4. **Otherwise,** we will move to the next student(*i.e. students += 1*) and allocate the book.

Finally, we will return the value of 'students'.

Practice:

Solve Problem

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

Brute Force Approach Optimal Approach

▼ Brute Force Approach ›

▼ Algorithm / Intuition ›

Naive Approach:

The extremely naive approach is to check all possible pages from max(arr[]) to sum(arr[]). The minimum pages for which we can allocate all the books to M students will be our answer.

Algorithm:

1. **If m > n:** In this case, book allocation is not possible and so, we will return -1.
2. Next, we will find the maximum element and the summation of the given array.
3. We will use a loop(say **pages**) to check all possible pages from max(arr[]) to sum(arr[]).
4. Next, inside the loop, we will send each 'pages', to the function **countStudents()** function to get the number of students to whom we can allocate the books.
    1. The first number of pages, 'pages', for which the number of students will be equal to 'm', will be our answer. So, we will return that particular 'pages'.
5. Finally, if we are out of the loop, we will return max(arr[]) as there cannot exist any answer smaller than that.

Dry-run: *Please refer to the video for the dry-run.*

▼ Code ›

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

int countStudents(vector<int> &arr, int pages) {
    int n = arr.size(); //size of array.
    int students = 1;
    long long pagesStudent = 0;
    for (int i = 0; i < n; i++) {
        if (pagesStudent + arr[i] <= pages) {
            //add pages to current student
            pagesStudent += arr[i];
        }
        else {
            //add pages to next student
            students++;
            pagesStudent = arr[i];
        }
    }
    return students;
}

int findPages(vector<int>& arr, int n, int m) {
    //book allocation impossible:
    if (m > n) return -1;

    int low = *max_element(arr.begin(), arr.end());
    int high = accumulate(arr.begin(), arr.end(), 0);

    for (int pages = low; pages <= high; pages++) {
        if (countStudents(arr, pages) == m) {
            return pages;
        }
    }
    return low;
}

int main()
{
    vector<int> arr = {25, 46, 28, 49, 24};
    int n = 5;
    int m = 4;
    int ans = findPages(arr, n, m);
    cout << "The answer is: " << ans << "\n";
    return 0;
}
```

Output: The answer is: 71.

▼ Complexity Analysis >

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 >
▼ Algorithm / Intuition >

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.

Upon closer observation, we can recognize that our answer space, represented as [max(arr[]), sum(arr[])], 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.

Algorithm:

1. **If m > n:** In this case, book allocation is not possible and so, we will return -1.
2. **Place the 2 pointers i.e. low and high:** Initially, we will place the pointers. The pointer low will point to max(arr[]) and the high will point to sum(arr[]).
3. **Calculate the 'mid':** Now, inside the loop, we will calculate the value of 'mid' using the following formula:
   mid = (low+high) // 2 ('//' refers to integer division)
4. **Eliminate the halves based on the number of students returned by countStudents():**
   We will pass the potential number of pages, represented by the variable 'mid', to the '**countStudents()**' function. This function will return the number of students to whom we can allocate the books.
   1. **If students > m:** On satisfying this condition, we can conclude that the number 'mid' is smaller than our answer. So, we will eliminate the left half and consider the right half(i.e. low = mid+1).
   2. **Otherwise,** the value mid is one of the possible answers. But we want the minimum value. So, we will eliminate the right half and consider the left half(i.e. high = mid-1).

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

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

Note: *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.*

Dry-run: *Please refer to the video for the dry-run.*

▼ Code ›

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

int countStudents(vector<int> &arr, int pages) {
    int n = arr.size(); //size of array.
    int students = 1;
    long long pagesStudent = 0;
    for (int i = 0; i < n; i++) {
        if (pagesStudent + arr[i] <= pages) {
            //add pages to current student
            pagesStudent += arr[i];
        }
        else {
            //add pages to next student
            students++;
            pagesStudent = arr[i];
        }
    }
    return students;
}

int findPages(vector<int>& arr, int n, int m) {
    //book allocation impossible:
    if (m > n) return -1;

    int low = *max_element(arr.begin(), arr.end());
    int high = accumulate(arr.begin(), arr.end(), 0);
    while (low <= high) {
        int mid = (low + high) / 2;
        int students = countStudents(arr, mid);
        if (students > m) {
            low = mid + 1;
        }
        else {
            high = mid - 1;
        }
    }
    return low;
}

int main()
{
    vector<int> arr = {25, 46, 28, 49, 24};
    int n = 5;
    int m = 4;
    int ans = findPages(arr, n, m);
    cout << "The answer is: " << ans << "\n";
    return 0;
```

```
}
```

Output: The answer is: 71.

▼ Complexity Analysis ›

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.
Reason: We are applying binary search on [max(arr[]), sum(arr[])]. Inside the loop, we are calling the countStudents() function for the value of 'mid'. 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.

▶ Video Explanation ›

> *Special thanks to **Dewanshi Paul**, **Sudip Ghosh** and **KRITIDIPTA GHOSH** for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam, please check out this article*

Split Array – Largest Sum

takeuforward.org/arrays/split-array-largest-sum

July 13, 2023

Problem Statement: Given an integer array ‘A’ of size ‘N’ and an integer ‘K’. Split the array ‘A’ into ‘K’ non-empty subarrays such that the largest sum of any subarray is minimized. Your task is to return the minimized largest sum of the split.
A subarray is a contiguous part of the array.

Pre-requisite: BS-18. Allocate Books or Book Allocation | Hard Binary Search

▼ **Examples** ›

Example 1:
Input Format: N = 5, a[] = {1,2,3,4,5}, k = 3
Result: 6
Explanation: There are many ways to split the array a[] into k consecutive subarrays. The best way to do this is to split the array a[] into [1, 2, 3], [4], and [5], where the largest sum among the three subarrays is only 6.

Example 2:
Input Format: N = 3, a[] = {3,5,1}, k = 3
Result: 5
Explanation: There is only one way to split the array a[] into 3 subarrays, i.e., [3], [5], and [1]. The largest sum among these subarrays is 5.

Upon close observation, we can understand that this problem is similar to the problem: BS-18. Allocate Books or Book Allocation | Hard Binary Search. In that case, we had to allocate books to the students. But actually, we were dividing that given array based on the subarray sum. We will do the same in this case.

Assume the given array is {10, 20, 30, 40} and k = 2. Now, we can split the array in the following ways:

- 10 | 20, 30, 40 → Maximum subarray sum = 90
- 10, 20 | 30, 40 → Maximum subarray sum = 70
- 10, 20, 30 | 40 → Maximum subarray sum = 60

From the above allocations, we can clearly observe that in the last case, the maximum subarray sum is the minimum possible. So, 60 will be the answer.

Observations:

- **Minimum possible answer:** We will get the minimum answer when we split the array into n subarrays(*i.e. Each subarray will have a single element*). Now, in this case, the maximum subarray sum 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 put all n elements into a single subarray. The maximum subarray sum will be 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 subarrays we need to make if the maximum subarray sum can be at most 'maxSum':

In order to calculate the number of subarrays we will write a function, **countPartitions()**. This function will take the array and 'maxSum' as parameters and return the number of partitions.

countPartitions(arr[], maxSum):

1. We will first declare two variables i.e. 'partitions'(*stores the no. of partitions*), and 'subarraySum'(*stores the sum of the current subarray*). As we are starting with the first subarray, 'partitions' should be initialized with 1.
2. We will start traversing the given array.
3. **If subarraySum + arr[i] <= maxSum:** If upon adding the current element with 'subarraySum' does not exceed 'maxSum', we can insert this i-th element to the current subarray.
4. **Otherwise,** we will move to the next subarray(*i.e. partitions += 1*) and insert the i-th element into that.

Finally, we will return the value of *'partitions'*.

Practice:

Solve Problem

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

Brute Force Approach Optimal Approach

▼ Brute Force Approach ›

▼ Algorithm / Intuition ›

Naive Approach:

The extremely naive approach is to check all possible answers from max(arr[]) to sum(arr[]). The minimum value for which we can make k subarrays will be our answer.

Algorithm:

1. First, we will find the maximum element and the summation of the given array.
2. We will use a loop(say **maxSum**) to check all possible answers from max(arr[]) to sum(arr[]).
3. Next, inside the loop, we will send 'maxSum', to the function **countPartitions()** function to get the number of partitions.
    1. The first value of 'maxSum', for which the number of partitions will be equal to 'k', will be our answer. So, we will return that particular value of 'maxSum'.
4. Finally, if we are out of the loop, we will return max(arr[]) as there cannot exist any answer smaller than that.

Dry-run: *Please refer to the video for the dry-run.*

▼ Code ›

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

int countPartitions(vector<int> &a, int maxSum) {
    int n = a.size(); //size of array.
    int partitions = 1;
    long long subarraySum = 0;
    for (int i = 0; i < n; i++) {
        if (subarraySum + a[i] <= maxSum) {
            //insert element to current subarray
            subarraySum += a[i];
        }
        else {
            //insert element to next subarray
            partitions++;
            subarraySum = a[i];
        }
    }
    return partitions;
}

int largestSubarraySumMinimized(vector<int> &a, int k) {
    int low = *max_element(a.begin(), a.end());
    int high = accumulate(a.begin(), a.end(), 0);

    for (int maxSum = low; maxSum <= high; maxSum++) {
        if (countPartitions(a, maxSum) == k)
            return maxSum;
    }
    return low;
}

int main()
{
    vector<int> a = {10, 20, 30, 40};
    int k = 2;
    int ans = largestSubarraySumMinimized(a, k);
    cout << "The answer is: " << ans << "\n";
    return 0;
}
```

Output: The answer is: 60

▼ Complexity Analysis >

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 values of time. Inside the loop, we are calling the countPartitions() function for each number. Now, inside the countPartitions() 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 >
▼ Algorithm / Intuition >

Optimal Approach(Using Binary Search):

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

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

Upon closer observation, we can recognize that our answer space, represented as [max(arr[]), sum(arr[])], 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.

Algorithm:

1. **Place the 2 pointers i.e. low and high:** Initially, we will place the pointers. The pointer low will point to max(arr[]) and the high will point to sum(arr[]).
2. **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)
3. **Eliminate the halves based on the number of subarrays returned by countPartitions():**
   We will pass the potential value of 'maxSum', represented by the variable 'mid', to the **'countPartitions()'** function. This function will return the number of partitions we can make.
   1. **If partitions > k:** On satisfying this condition, we can conclude that the number 'mid' is smaller than our answer. So, we will eliminate the left half and consider the right half(i.e. low = mid+1).
   2. **Otherwise,** the value mid is one of the possible answers. But we want the minimum value. So, we will eliminate the right half and consider the left half(i.e. high = mid-1).

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

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

Note: *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.*

Dry-run: *Please refer to the video for the dry-run.*

▼ Code ›

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

int countPartitions(vector<int> &a, int maxSum) {
    int n = a.size(); //size of array.
    int partitions = 1;
    long long subarraySum = 0;
    for (int i = 0; i < n; i++) {
        if (subarraySum + a[i] <= maxSum) {
            //insert element to current subarray
            subarraySum += a[i];
        }
        else {
            //insert element to next subarray
            partitions++;
            subarraySum = a[i];
        }
    }
    return partitions;
}

int largestSubarraySumMinimized(vector<int> &a, int k) {
    int low = *max_element(a.begin(), a.end());
    int high = accumulate(a.begin(), a.end(), 0);
    //Apply binary search:
    while (low <= high) {
        int mid = (low + high) / 2;
        int partitions = countPartitions(a, mid);
        if (partitions > k) {
            low = mid + 1;
        }
        else {
            high = mid - 1;
        }
    }
    return low;
}

int main()
{
    vector<int> a = {10, 20, 30, 40};
    int k = 2;
    int ans = largestSubarraySumMinimized(a, k);
    cout << "The answer is: " << ans << "\n";
    return 0;
}
```

Output: The answer is: 60.

▼ Complexity Analysis ›

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.
Reason: We are applying binary search on [max(arr[]), sum(arr[])]. Inside the loop, we are calling the countPartitions() function for the value of 'mid'. Now, inside the countPartitions() 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.

▶ Video Explanation ›

> *Special thanks to **KRITIDIPTA GHOSH** for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam,* please check out this article

Painter's Partition Problem

takeuforward.org/arrays/painters-partition-problem

July 13, 2023

Problem Statement: Given an array/list of length 'N', where the array/list represents the boards and each element of the given array/list represents the length of each board. Some 'K' numbers of painters are available to paint these boards. Consider that each unit of a board takes 1 unit of time to paint. You are supposed to return the area of the minimum time to get this job done of painting all the 'N' boards under the constraint that any painter will only paint the continuous sections of boards.

Pre-requisite: BS-18. Allocate Books or Book Allocation | Hard Binary Search

▼ **Examples** ›

Example 1:
Input Format: N = 4, boards[] = {5, 5, 5, 5}, k = 2
Result: 10
Explanation: We can divide the boards into 2 equal-sized partitions, so each painter gets 10 units of the board and the total time taken is 10.

Example 2:
Input Format: N = 4, boards[] = {10, 20, 30, 40}, k = 2
Result: 60
Explanation: We can divide the first 3 boards for one painter and the last board for the second painter.

We can allocate the boards to the painters in several ways but it is clearly said in the question that we have to allocate the boards in such a way that the painters can paint all the boards in the minimum possible time. The painters will work simultaneously.

Note: *Upon close observation, we can understand that this problem is similar to the previous problem: BS-18. Allocate Books or Book Allocation | Hard Binary Search. There we had to allocate books to the students and here we need to allocate walls to the painters.*

Assume the given array is {10, 20, 30, 40} and number of painters, k = 2. Now, we can allocate these boards in different ways. Some of them are the following:

- 10 | 20, 30, 40 → Minimum time required to paint all the boards = 90
- 10, 20 | 30, 40 → Minimum time required to paint all the boards = 70
- 10, 20, 30 | 40 → Minimum time required to paint all the boards = 60

From the above allocations, we can clearly observe that in the last case, the first painter will paint the first 3 walls in 60 units of time and the second painter will take 40 units of time. So, the minimum time required to paint all the boards is 60.

Observations:

- **Minimum possible answer:** We will get the minimum answer when we give n boards of the array to n painters(*i.e. Each painter will be allocated 1 board*). Now, in this case, the minimum time required to paint all the boards 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 boards to a single painter. The total time required 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 painters we need if we have to paint all the walls within 'time' units of time:

In order to calculate the number of painters we will write a function, **countPainters()**. This function will take the array and 'time' as parameters and return the number of painters to whom we can allocate the boards.

countPainters(arr[], time):

1. We will first declare two variables i.e. 'painters'(*stores the no. of painters*), and 'boardsPainter'(*stores the unit of boards, a painter will paint*). As we are starting with the first painter, 'painters' should be initialized with 1.
2. We will start traversing the given array.
3. **If boardsPainter + arr[i] <= time:** If upon adding the current board with 'boardsPainter' does not exceed the time limit, we can allocate this i-th board to the current painter.
4. **Otherwise,** we will move to the next painter(*i.e. painters += 1*) and allocate the i-th board.

Finally, we will return the value of 'painters'.

Practice:

Solve Problem

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

Brute Force Approach Optimal Approach

▼ Brute Force Approach ›

▼ Algorithm / Intuition ›

Naive Approach:

The extremely naive approach is to check all possible answers from max(arr[]) to sum(arr[]). The minimum time for which we can paint all the boards will be our answer.

Algorithm:

1. First, we will find the maximum element and the summation of the given array.
2. We will use a loop(say **time**) to check all possible answers from max(arr[]) to sum(arr[]).
3. Next, inside the loop, we will send 'time', to the function **countPainters()** function to get the number of painters to whom we can allocate the boards.
    1. The first value of 'time', for which the number of painters will be lesser or equal to 'k', will be our answer. So, we will return that particular value of 'time'.
4. Finally, if we are out of the loop, we will return max(arr[]) as there cannot exist any answer smaller than that.

Dry-run: *Please refer to the video for the dry-run.*

▼ Code ›

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

int countPainters(vector<int> &boards, int time) {
    int n = boards.size(); //size of array.
    int painters = 1;
    long long boardsPainter = 0;
    for (int i = 0; i < n; i++) {
        if (boardsPainter + boards[i] <= time) {
            //allocate board to current painter
            boardsPainter += boards[i];
        }
        else {
            //allocate board to next painter
            painters++;
            boardsPainter = boards[i];
        }
    }
    return painters;
}

int findLargestMinDistance(vector<int> &boards, int k) {
    int low = *max_element(boards.begin(), boards.end());
    int high = accumulate(boards.begin(), boards.end(), 0);

    for (int time = low; time <= high; time++) {
        if (countPainters(boards, time) <= k) {
            return time;
        }
    }
    return low;
}

int main()
{
    vector<int> boards = {10, 20, 30, 40};
    int k = 2;
    int ans = findLargestMinDistance(boards, k);
    cout << "The answer is: " << ans << "\n";
    return 0;
}
```

Output: The answer is: 60

▼ Complexity Analysis ›

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 values of time. Inside the loop, we are calling the countPainters() function for each number. Now, inside the countPainters() 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 >
▼ Algorithm / Intuition >
Optimal Approach(Using Binary Search):

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

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

Upon closer observation, we can recognize that our answer space, represented as [max(arr[]), sum(arr[])], 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.

Algorithm:

1. **Place the 2 pointers i.e. low and high:** Initially, we will place the pointers. The pointer low will point to max(arr[]) and the high will point to sum(arr[]).
2. **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)
3. **Eliminate the halves based on the number of painters returned by countPainters():**
   We will pass the potential value of time, represented by the variable 'mid', to the '**countPainters()**' function. This function will return the number of painters we need to paint all the boards.
   1. **If painters > k:** On satisfying this condition, we can conclude that the number 'mid' is smaller than our answer. So, we will eliminate the left half and consider the right half(i.e. low = mid+1).
   2. **Otherwise,** the value mid is one of the possible answers. But we want the minimum value. So, we will eliminate the right half and consider the left half(i.e. high = mid-1).
4. Finally, outside the loop, we will return the value of low as the pointer will be pointing to the answer.

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

Note: *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.*

Dry-run: *Please refer to the video for the dry-run.*

▼ Code ›

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

int countPainters(vector<int> &boards, int time) {
    int n = boards.size(); //size of array.
    int painters = 1;
    long long boardsPainter = 0;
    for (int i = 0; i < n; i++) {
        if (boardsPainter + boards[i] <= time) {
            //allocate board to current painter
            boardsPainter += boards[i];
        }
        else {
            //allocate board to next painter
            painters++;
            boardsPainter = boards[i];
        }
    }
    return painters;
}

int findLargestMinDistance(vector<int> &boards, int k) {
    int low = *max_element(boards.begin(), boards.end());
    int high = accumulate(boards.begin(), boards.end(), 0);
    //Apply binary search:
    while (low <= high) {
        int mid = (low + high) / 2;
        int painters = countPainters(boards, mid);
        if (painters > k) {
            low = mid + 1;
        }
        else {
            high = mid - 1;
        }
    }
    return low;
}

int main()
{
    vector<int> boards = {10, 20, 30, 40};
    int k = 2;
    int ans = findLargestMinDistance(boards, k);
    cout << "The answer is: " << ans << "\n";
    return 0;
}
```

Output: The answer is: 60.

▼ Complexity Analysis ›

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.
Reason: We are applying binary search on [max(arr[]), sum(arr[])]. Inside the loop, we are calling the countPainters() function for the value of 'mid'. Now, inside the countPainters() 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.

▶ Video Explanation ›

> *Special thanks to **KRITIDIPTA GHOSH** for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam,* please check out this article

Minimise Maximum Distance between Gas Stations

takeuforward.org/arrays/minimise-maximum-distance-between-gas-stations

July 25, 2023

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.

▼ **Examples** ›

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}.

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

Brute Force Approach Better Approach Optimal Approach

▼ Brute Force Approach ›
▼ Algorithm / Intuition ›

Naive 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:

1. First, we will declare an array **'howMany[]'** of size n-1, to keep track of the number of placed gas stations.
2. Next, using a loop we will pick k gas stations one at a time.
3. 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]++).
4. 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)
5. Among all the distances, the maximum one will be the answer.

Dry-run: *Please refer to the video for the dry-run.*

▼ Code ›

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

long double minimiseMaxDistance(vector<int> &arr, int k) {
    int n = arr.size(); //size of array.
    vector<int> howMany(n - 1, 0);

    //Pick and place k gas stations:
    for (int gasStations = 1; gasStations <= k; gasStations++) {
        //Find the maximum section
        //and insert the gas station:
        long double maxSection = -1;
        int maxInd = -1;
        for (int i = 0; i < n - 1; i++) {
            long double diff = arr[i + 1] - arr[i];
            long double sectionLength =
                diff / (long 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:
    long double maxAns = -1;
    for (int i = 0; i < n - 1; i++) {
        long double diff = arr[i + 1] - arr[i];
        long double sectionLength =
            diff / (long double)(howMany[i] + 1);
        maxAns = max(maxAns, sectionLength);
    }
    return maxAns;
}

int main()
{
    vector<int> arr = {1, 2, 3, 4, 5};
    int k = 4;
    long double ans = minimiseMaxDistance(arr, k);
    cout << "The answer is: " << ans << "\n";
    return 0;
}
```

Output: The answer is: 0.5

▼ Complexity Analysis >

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 >

▼ Algorithm / Intuition >

Better Approach(Using Heap):

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:

1. 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.
2. We will insert the first n-1 indices with the respective distance value, arrr[i+1]-arr[i] for every index.
3. Next, using a loop we will pick k gas stations one at a time.
4. 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'.**
5. 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)

6. After that, we will again insert the pair <new_section_length, secInd> into the priority queue for further consideration.
7. 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.

Dry-run: *Please refer to the video for the dry-run.*

▼ Code ›

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

long double minimiseMaxDistance(vector<int> &arr, int k) {
    int n = arr.size(); //size of array.
    vector<int> howMany(n - 1, 0);
    priority_queue<pair<long double, int>> pq;

    //insert the first n-1 elements into pq
    //with respective distance values:
    for (int i = 0; i < n - 1; i++) {
        pq.push({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:
        auto tp = pq.top();
        pq.pop();
        int secInd = tp.second;

        //insert the current gas station:
        howMany[secInd]++;

        long double inidiff = arr[secInd + 1] - arr[secInd];
        long double newSecLen = inidiff / (long double)(howMany[secInd] + 1);
        pq.push({newSecLen, secInd});
    }

    return pq.top().first;
}

int main()
{
    vector<int> arr = {1, 2, 3, 4, 5};
    int k = 4;
    long double ans = minimiseMaxDistance(arr, k);
    cout << "The answer is: " << ans << "\n";
    return 0;
}
```

Output: The answer is: 0.5

▼ Complexity Analysis ›

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 ›
▼ Algorithm / Intuition ›

Optimal Approach(Using Binary Search):

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

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

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[]):

1. We will use a loop(say i) that will run from 1 to n.
2. For each section between i and i-1, we will do the following:
   No. of stations = (arr[i]-arr[i-1]) / dist
3. **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.
4. Now, we will add the no. of stations regarding all the sections and the total will be the answer.

Algorithm:

1. **First,** we will find the maximum distance between two consecutive gas stations i.e. max(dist).
2. **Place the 2 pointers i.e. low and high:** Initially, we will place the pointers. The pointer low will point to 0 and the high will point to max(dist).
3. **Now,** we will use the 'while' loop like this: **while(high – low > 10^(-6)).**
4. **Calculate the 'mid':** Now, inside the loop, we will calculate the value of 'mid' using the following formula:
   mid = (low+high) / 2.0

5. **Eliminate the halves based on the number of stations returned by numberOfGasStationsRequired():**
   We will pass the potential value of 'dist', represented by the variable 'mid', to the '**numberOfGasStationsRequired()**' function. This function will return the number of gas stations we can place.
   1. **If result > k:** On satisfying this condition, we can conclude that the number 'mid' is smaller than our answer. So, we will eliminate the left half and consider the right half(i.e. low = mid).
   2. **Otherwise,** the value mid is one of the possible answers. But we want the minimum value. So, we will eliminate the right half and consider the left half(i.e. high = mid).
6. Finally, outside the loop, we can return either low or high as their difference is beyond 10^(-6). They both can be the possible answer. Here, we have returned the 'high'.

The steps from 4-5 will be inside a loop and the loop will continue until (low-high <= 10^(-6)).

Dry-run: *Please refer to the video for the dry-run.*

▼ Code >

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

int numberOfGasStationsRequired(long double dist, vector<int> &arr) {
    int n = arr.size(); // size of the array
    int cnt = 0;
    for (int i = 1; i < n; i++) {
        int numberInBetween = ((arr[i] - arr[i - 1]) / dist);
        if ((arr[i] - arr[i - 1]) == (dist * numberInBetween)) {
            numberInBetween--;
        }
        cnt += numberInBetween;
    }
    return cnt;
}

long double minimiseMaxDistance(vector<int> &arr, int k) {
    int n = arr.size(); // size of the array
    long double low = 0;
    long double high = 0;

    //Find the maximum distance:
    for (int i = 0; i < n - 1; i++) {
        high = max(high, (long double)(arr[i + 1] - arr[i]));
    }

    //Apply Binary search:
    long double diff = 1e-6 ;
    while (high - low > diff) {
        long double mid = (low + high) / (2.0);
        int cnt = numberOfGasStationsRequired(mid, arr);
        if (cnt > k) {
            low = mid;
        }
        else {
            high = mid;
        }
    }
    return high;
}

int main()
{
    vector<int> arr = {1, 2, 3, 4, 5};
    int k = 4;
    long double ans = minimiseMaxDistance(arr, k);
    cout << "The answer is: " << ans << "\n";
    return 0;
```

```
}
```

Output: The answer is: 0.5

▼ Complexity Analysis ›

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.

▶ Video Explanation ›

> *Special thanks to **KRITIDIPTA GHOSH** for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam,* please check out this article

Median of Two Sorted Arrays of different sizes

takeuforward.org/data-structure/median-of-two-sorted-arrays-of-different-sizes

December 2, 2021

Problem Statement: Given **two sorted arrays** arr1 and arr2 of size m and n respectively, return the **median** of the two sorted arrays. The median is defined as the middle value of a sorted list of numbers. In case the length of the list is even, the median is the average of the two middle elements.

▼ **Examples** ›

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.

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

Brute Force Approach Better Approach Optimal Approach

▼ Brute Force Approach >

▼ Algorithm / Intuition >

Naive Approach(Brute-force):

The extremely naive approach is to merge the two sorted arrays and then find the median in that merged array.

How to merge two sorted arrays:

The word “merge” suggests applying the merge step of the merge sort algorithm. In that step, we essentially perform the same actions as required by this solution. By using two pointers on two given arrays, we fill up the elements into a third array.

How to find the median:

- **If the length of the merged array (n1+n2) is even:** The median is the average of the two middle elements. index = (n1+n2) / 2, median = (arr3[index] + arr3[index-1]) / 2.0.
- **If the length of the merged array (n1+n2) is odd:** index = (n1+n2) / 2, median = arr3[index].

Algorithm:

1. We will use a third array i.e. arr3[] of size (n1+n2) to store the elements of the two sorted arrays.
2. Now, 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[].
3. 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 insert the smallest element in the third array and increase that specific pointer by 1.
    1. **If arr1[i] < arr2[j]:** Insert arr1[i] into the third array and increase i by 1.
    2. **Otherwise:** Insert arr2[j] into the third array and increase j by 1.
4. After that, the left-out elements from both arrays will be copied as it is into the third array.
5. Now, the third array i.e. arr3[] will be the sorted merged array. Now the median will be the following:
    1. **If the length of arr3[] i.e. (n1+n2) is even:** The median is the average of the two middle elements. index = (n1+n2) / 2, median = (arr3[index] + arr3[index-1]) / 2.0.
    2. **If the length of arr3[] i.e. (n1+n2) is odd:** index = (n1+n2) / 2, median = arr3[index].
6. Finally, we will return the value of the median.

Dry-run: *Please refer to the attached video for a detailed dry-run.*

▼ Code ›

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

double median(vector<int>& a, vector<int>& b) {
    //size of two given arrays:
    int n1 = a.size(), n2 = b.size();

    vector<int> arr3;
    //apply the merge step:
    int i = 0, j = 0;
    while (i < n1 && j < n2) {
        if (a[i] < b[j]) arr3.push_back(a[i++]);
        else arr3.push_back(b[j++]);
    }

    //copy the left-out elements:
    while (i < n1) arr3.push_back(a[i++]);
    while (j < n2) arr3.push_back(b[j++]);

    //Find the median:
    int n = n1 + n2;
    if (n % 2 == 1) {
        return (double)arr3[n / 2];
    }

    double median = ((double)arr3[n / 2] + (double)arr3[(n / 2) - 1]) / 2.0;
    return median;
}

int main()
{
    vector<int> a = {1, 4, 7, 10, 12};
    vector<int> b = {2, 3, 6, 15};
    cout << "The median of two sorted array is " << fixed << setprecision(1)
         << median(a, b) << '\n';
}
```

Output: The median of two sorted arrays is 6.0

▼ Complexity Analysis ›
Time Complexity: O(n1+n2), where n1 and n2 are the sizes of the given arrays.
Reason: We traverse through both arrays linearly.

Space Complexity: O(n1+n2), where n1 and n2 are the sizes of the given arrays.
Reason: We are using an extra array of size (n1+n2) to solve this problem.

▼ Better Approach ›

▼ Algorithm / Intuition ›

To optimize the space used in the previous approach, we can eliminate the third array used to store the final merged result. After closer examination, we realize that we only need the two middle elements at indices (n1+n2)/2 and ((n1+n2)/2)-1, rather than the entire merged array, to solve the problem effectively.

We will stick to the same basic approach, but instead of storing elements in a separate array, we will use a counter called 'cnt' to represent the imaginary third array's index. As we traverse through the arrays, when 'cnt' reaches either index (n1+n2)/2 or ((n1+n2)/2)-1, we will store that particular element. This way, we can achieve the same goal without using any extra space.

Algorithm:

1. 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.
2. 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[].
3. 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.
   1. **If arr1[i] < arr2[j]:** Check 'cnt' to perform necessary operations and increase i and 'cnt' by 1.
   2. **Otherwise:** Check 'cnt' to perform necessary operations and increase j and 'cnt' by 1.
4. 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.
5. Now, let's call the elements at the required indices as **ind1el(*at ind1*)** and **ind2el(*at ind2*)**:
   1. **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.
   2. **If the total length i.e. (n1+n2) is odd:** median = ind2el.
6. Finally, we will return the value of the median.

Dry-run: *Please refer to the attached video for a detailed dry-run.*

▼ Code ›

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

double median(vector<int>& a, vector<int>& b) {
    //size of two given arrays:
    int n1 = a.size(), n2 = b.size();
    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;
}

int main()
{
    vector<int> a = {1, 4, 7, 10, 12};
    vector<int> b = {2, 3, 6, 15};
    cout << "The median of two sorted array is " << fixed << setprecision(1)
         << median(a, b) << '\n';
}
```

Output: The median of two sorted arrays is 6.0

▼ Complexity Analysis ›
Time Complexity: O(n1+n2), where n1 and n2 are the sizes of the given arrays.
Reason: We traverse through both arrays linearly.

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

▼ Optimal Approach ›
▼ Algorithm / Intuition ›
We are going to use the Binary Search algorithm to optimize the approach.

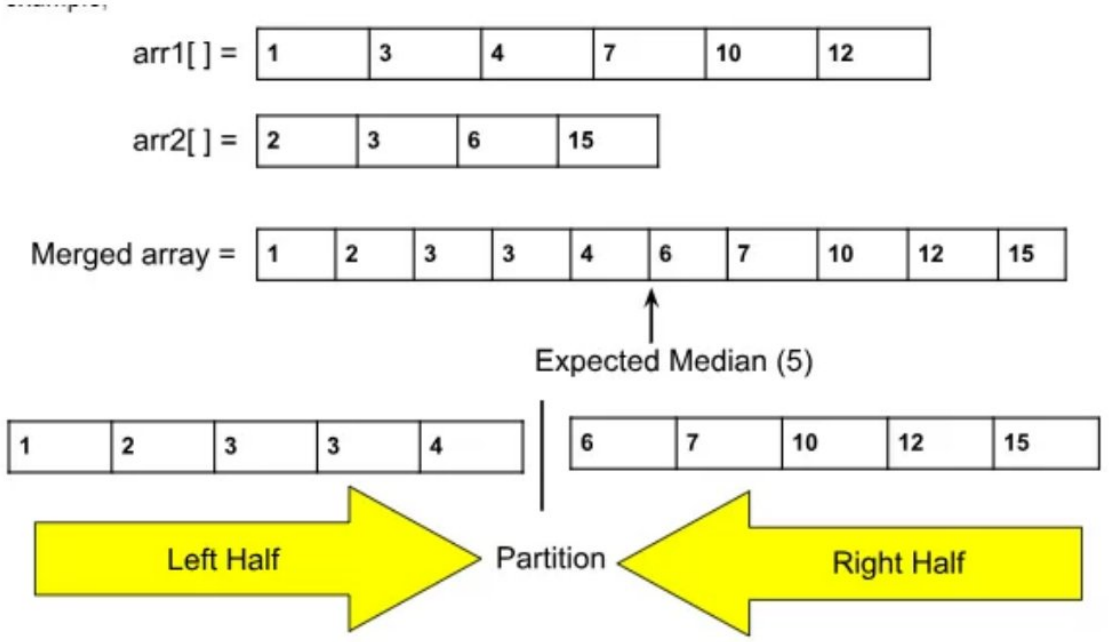
The primary objective of the Binary Search algorithm is to efficiently determine the appropriate half to eliminate, thereby reducing the search space by half. It does this by determining a specific condition that ensures that the target is not present in that half.

Now, 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.

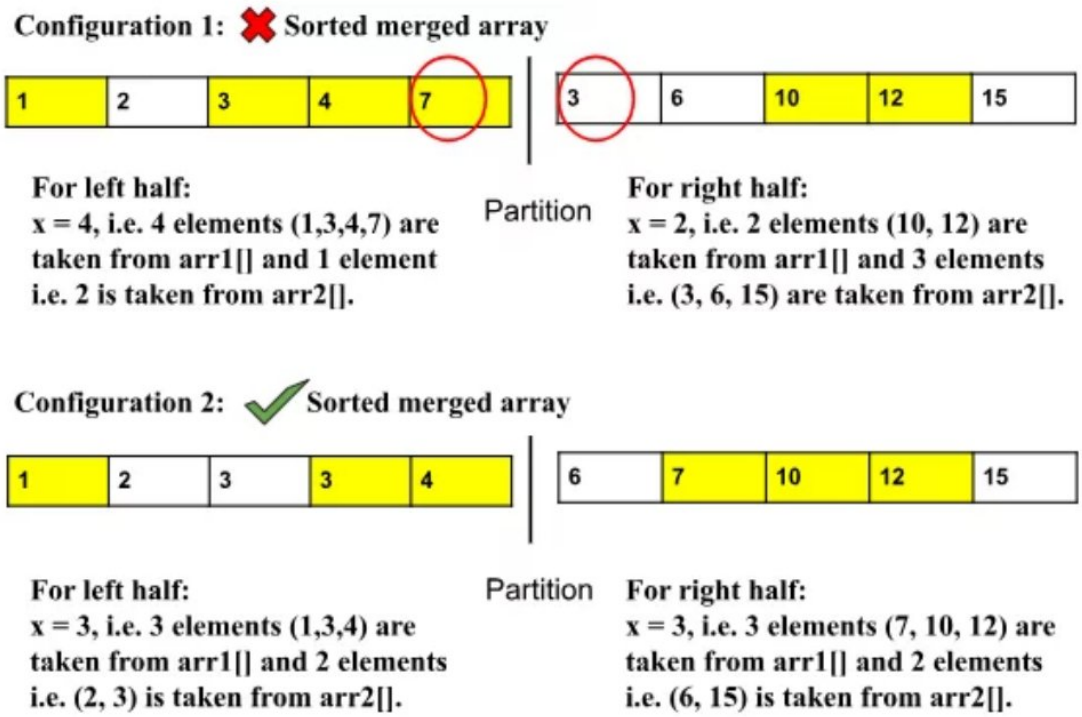
Observations:

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:



The first configuration is not valid as the merged array containing the left and right half is not sorted. But the second one is valid. So, for a valid merged array, the configuration of the left and right halves is unique.

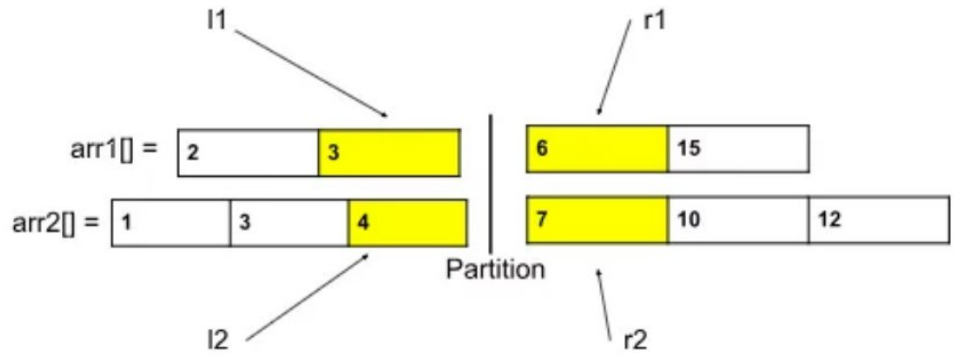
How to solve the problem using the above observations:

- **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,



If max(l1, l2) <= min(r1, r2), then the merged array will always be sorted.

If the value of x is 2 and n/2 is 5, l1 = arr1[1], l2 = arr2[2], r1 = arr1[2], r2 = arr2[3]. Thus we can check if the formed left half is valid or not.

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)].

Algorithm:

1. First, we have to make sure that the arr1[] is the smaller array. If not by default, we will just swap the arrays. Our main goal is to consider the smaller array as arr1[].
2. **Calculate the length of the left half:** left = (n1+n2+1) / 2.
3. **Place the 2 pointers i.e. low and high:** Initially, we will place the pointers. The pointer low will point to 0 and the high will point to n1(*i.e. The size of arr1[]*).
4. **Calculate the 'mid1' i.e. x and 'mid2' i.e. left-x:** Now, inside the loop, we will calculate the value of 'mid1' using the following formula:
   mid1 = (low+high) // 2 ('//' refers to integer division)
   mid2 = left-mid1
5. **Calculate l1, l2, r1, and r2:** Generally,
   1. l1 = arr1[mid1-1]
   2. l2 = arr2[mid2-1]
   3. r1 = arr1[mid1]
   4. r2 = arr2[mid2]

   The possible values of 'mid1' and 'mid2' might be 0 and n1 and n2 respectively. So, to handle these cases, we need to store some default values for these four variables. The default value for l1 and l2 will be **INT_MIN** and for r1 and r2, it will be **INT_MAX.**

6. **Eliminate the halves based on the following conditions:**
    1. **If l1 <= r2 && l2 <= r1:** We have found the answer.
        1. **If (n1+n2) is odd:** Return the median = max(l1, l2).
        2. **Otherwise:** Return median = (max(l1, l2)+min(r1, r2)) / 2.0
    2. **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 = mid1-1).
    3. **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 = mid1+1).
7. Finally, outside the loop, we will include a dummy return statement just to avoid warnings or errors.

The steps from 4-6 will be inside a loop and the loop will continue until low crosses high.

Dry-run: *Please refer to the attached video for the dry run.*

▼ Code ›

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

double median(vector<int>& a, vector<int>& b) {
    int n1 = a.size(), n2 = b.size();
    //if n1 is bigger swap the arrays:
    if (n1 > n2) return median(b, a);

    int n = n1 + n2; //total length
    int left = (n1 + n2 + 1) / 2; //length of left half
    //apply binary search:
    int low = 0, high = n1;
    while (low <= high) {
        int mid1 = (low + high) >> 1;
        int mid2 = left - mid1;
        //calculate l1, l2, r1 and r2;
        int l1 = INT_MIN, l2 = INT_MIN;
        int r1 = INT_MAX, r2 = INT_MAX;
        if (mid1 < n1) r1 = a[mid1];
        if (mid2 < n2) r2 = b[mid2];
        if (mid1 - 1 >= 0) l1 = a[mid1 - 1];
        if (mid2 - 1 >= 0) l2 = b[mid2 - 1];

        if (l1 <= r2 && l2 <= r1) {
            if (n % 2 == 1) return max(l1, l2);
            else return ((double)(max(l1, l2) + min(r1, r2))) / 2.0;
        }

        //eliminate the halves:
        else if (l1 > r2) high = mid1 - 1;
        else low = mid1 + 1;
    }
    return 0; //dummy statement
}

int main()
{
    vector<int> a = {1, 4, 7, 10, 12};
    vector<int> b = {2, 3, 6, 15};
    cout << "The median of two sorted array is " << fixed << setprecision(1)
         << median(a, b) << '\n';
}
```

Output: The median of two sorted array is 6.0

▼ Complexity Analysis ›

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.

▶ Video Explanation ›

> *Special thanks to **Dewanshi Paul** and **KRITIDIPTA GHOSH** for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam, please check out this article*

K-th Element of two sorted arrays

takeuforward.org/data-structure/k-th-element-of-two-sorted-arrays

November 29, 2021

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

Examples :

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

Output:
 6

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


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

Output:
 4

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

Solution:

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

Solution 1: Naive Solution

Intuition:

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

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

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

Approach :

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

Code:

- C++ Code
- Java Code
- Python Code

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

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

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

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

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

Time Complexity :

We iterate at total k times. This makes time complexity to O(k)

Space Complexity :

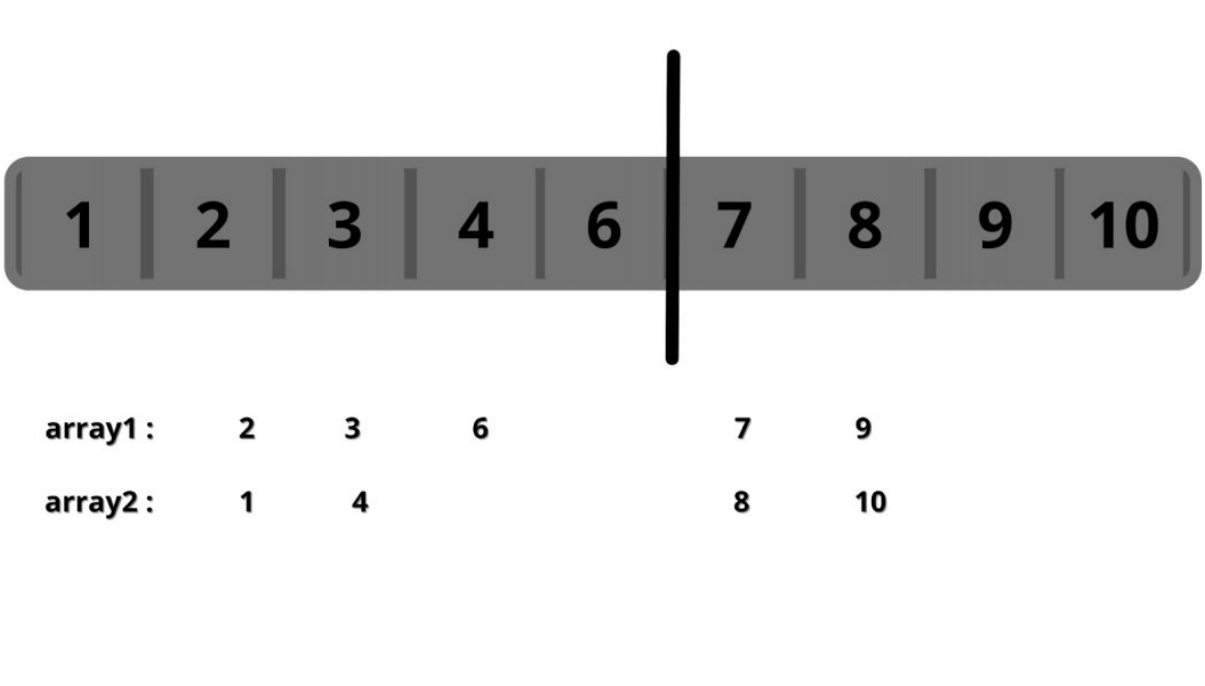
We do not use any extra data structure and hence, the time complexity is O(1).

Approach 2: Optimal Solution

Intuition :

It is mentioned that given arrays are sorted. This gives us some hints to use binary search in them.

If we look into the final merged sorted array.



We can part it in such a way that our kth element will be at the end of the left half array. Let's make some observations. The left portion of the array is made of some elements of both array1 and array2. We can see that all elements of the right half of the array are always larger than the left ones. So, with help of binary search, we will divide arrays into partitions with keeping k elements in the left half. We have to keep in mind that l1 <= r2 and l2 <= r1. Why so? This ensures that left-half elements are always lesser than right elements.

Approach :

Apply binary search in an array with a small size. Start iterating with two pointers, say left and right. Find the middle of the range. Take elements from low to middle of array1 and the remaining elements from the second array. Then using the condition mentioned above, check if the left half is valid. If valid, print the maximum of both array's last element. If not, move the range towards the right if l2 > r1, else move the range towards the left if l1 > r2.

Code:

- C++ Code
- Java Code
- Python Code

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

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

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

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

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

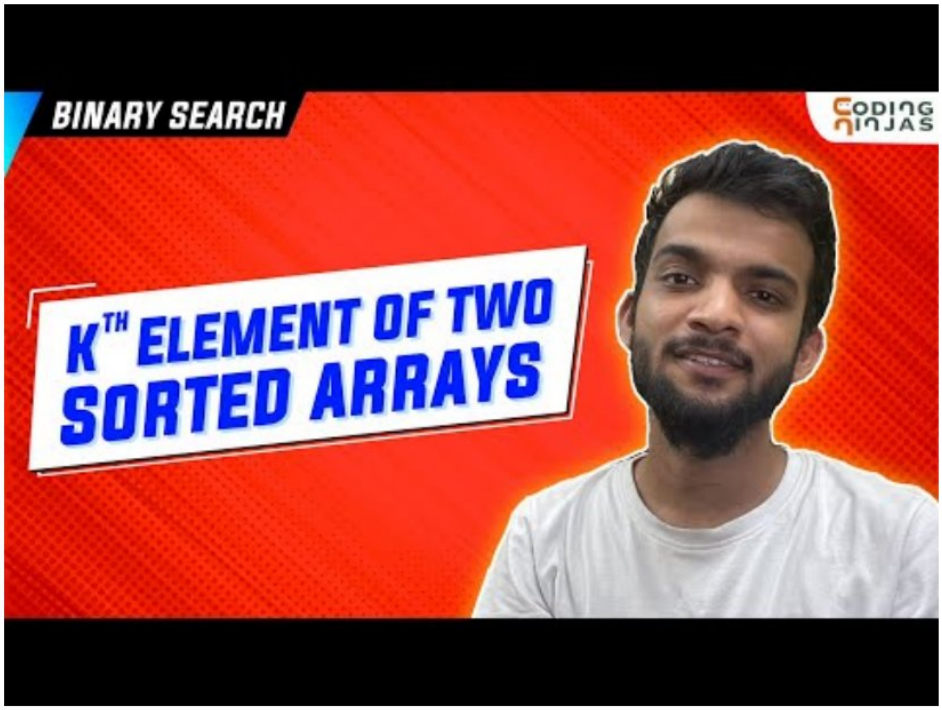
Time Complexity : log(min(m,n))

Reason: We are applying binary search in the array with minimum size among the two. And we know the time complexity of the binary search is log(N) where N is the size of the array. Thus, the time complexity of this approach is log(min(m,n)), where m,n are the sizes of two arrays.

Space Complexity: O(1)

Reason: Since no extra data structure is used, making space complexity to O(1).

> *Special thanks to **Dewanshi Paul** and **Sudip Ghosh** for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam, please check out this article*



Watch Video At: https://youtu.be/nv7F4PiLUzo

Find the row with maximum number of 1’s

takeuforward.org/arrays/find-the-row-with-maximum-number-of-1s

August 24, 2023

Problem Statement: You have been given a non-empty grid **‘mat’** with **‘n’** rows and **‘m’** columns consisting of only 0s and 1s. All the rows are sorted in ascending order.
Your task is to find the index of the row with the maximum number of ones.
Note: If two rows have the same number of ones, consider the one with a smaller index. If there’s no row with at least 1 zero, return -1.

Pre-requisite: Lower Bound implementation, Upper Bound implementation, & Find the first occurrence of a number.

▼ **Examples** ›

```
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.
```

Practice:

Solve Problem

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

Brute Force Approach Optimal Approach

▼ Brute Force Approach >

▼ Algorithm / Intuition >

In the question, it is clearly stated that we should return -1 if the matrix does not contain any 1.

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.

Algorithm:

The steps are as follows:

1. First, we will declare 2 variables i.e. cnt_max(*initialized with 0*), and index(*initialized with -1*). The first variable will store the maximum number of 1's we have got and the 'index' will store the row number.
2. Next, we will start traversing the matrix. We will use a loop(say i) to select each row at a time.
3. Now, for each row i, we will use another loop(say j) and count the number of 1's in that row.
4. After that, we will compare it with cnt_max and if the current number of 1's is greater, we will update cnt_max with the current no. of 1's and 'index' with the current row index.

5. Finally, we will return the variable ‘index’. It will store the index of the row with the maximum no. of 1’s. And otherwise, it will store -1.

Note: As we want the row with the minimum index, we will only update the index if the current number of 1’s is greater than cnt_max (*we will not update if they are equal*).

Dry-run: *Please refer to the attached video for a detailed dry-run.*

▼ Code ›

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

int rowWithMax1s(vector<vector<int>> &matrix, int n, int m) {
    int cnt_max = 0;
    int index = -1;

    //traverse the matrix:
    for (int i = 0; i < n; i++) {
        int cnt_ones = 0;
        for (int j = 0; j < m; j++) {
            cnt_ones += matrix[i][j];
        }
        if (cnt_ones > cnt_max) {
            cnt_max = cnt_ones;
            index = i;
        }
    }
    return index;
}

int main()
{
    vector<vector<int>> matrix = {{1, 1, 1}, {0, 0, 1}, {0, 0, 0}};
    int n = 3, m = 3;
    cout << "The row with maximum no. of 1's is: " <<
         rowWithMax1s(matrix, n, m) << '\n';
}
```

Output: The row with maximum no. of 1’s is: 0

▼ Complexity Analysis ›

Time Complexity: O(n X m), where n = given row number, m = given column number.
Reason: We are using nested loops running for n and m times respectively.

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

▼ Optimal Approach ›
▼ Algorithm / Intuition ›
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 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.

Algorithm:

1. First, we will declare 2 variables i.e. cnt_max(*initialized with 0*), and index(*initialized with -1*). The first variable will store the maximum number of 1's we have got and 'index' will store the row number.
2. Next, we will start traversing the rows. We will use a loop(say i) to select each row at a time.
3. Now, for each row i, we will use lowerBound() to get the first occurrence of 1. Now, using the following formula we will calculate the number of 1's:
   Number_of_ones = m(*number of columns*) – lowerBound(1)(*0-based index*).
4. After that, we will compare it with cnt_max and if the current number of 1's is greater, we will update cnt_max with the current no. of 1's and 'index' with the current row index.
5. Finally, we will return the variable 'index'. It will store the index of the row with the maximum no. of 1's. And if the matrix does not contain any 1, it stores -1.

Note: As we want the row with the minimum index, we will only update the index if the current number of 1's is greater than cnt_max (*we will not update if they are equal*).

Dry-run: *Please refer to the attached video for a detailed dry-run.*

▼ Code ›

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

int lowerBound(vector<int> 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[mid] >= x) {
            ans = mid;
            //look for smaller index on the left
            high = mid - 1;
        }
        else {
            low = mid + 1; // look on the right
        }
    }
    return ans;
}
int rowWithMax1s(vector<vector<int>> &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[i], m, 1);
        if (cnt_ones > cnt_max) {
            cnt_max = cnt_ones;
            index = i;
        }
    }
    return index;
}

int main()
{
    vector<vector<int>> matrix = {{1, 1, 1}, {0, 0, 1}, {0, 0, 0}};
    int n = 3, m = 3;
    cout << "The row with maximum no. of 1's is: " <<
         rowWithMax1s(matrix, n, m) << '\n';
}
```

Output: The row with maximum no. of 1's is: 0

▼ Complexity Analysis ›

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.

▶ Video Explanation ›

> *Special thanks to **KRITIDIPTA GHOSH** for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam,* please check out this article

Search in a sorted 2D matrix

takeuforward.org/data-structure/search-in-a-sorted-2d-matrix

October 24, 2021

Problem Statement: You have been given a 2-D array **'mat'** of size **'N x M'** where **'N'** and **'M'** denote the number of rows and columns, respectively. The elements of each row are sorted in non-decreasing order. Moreover, the first element of a row is greater than the last element of the previous row (if it exists). You are given an integer **'target'**, and your task is to find if it exists in the given 'mat' or not.

▼ **Examples** ›

```
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'.
```

Practice:

Solve Problem

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

Brute Force Approach Better Approach Optimal Approach

▼ Brute Force Approach ›

▼ Algorithm / Intuition ›

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’.

Algorithm:

1. We will use a loop(say i) to select a particular row at a time.
2. Next, for every row, we will use another loop(say j) to traverse each column.
3. Inside the loops, we will check if the element i.e. matrix[i][j] is equal to the ‘target’. If we find any matching element, we will return true.
4. Otherwise, after completing the traversal, we will return false.

Dry-run: *Please refer to the attached video for a detailed dry-run.*

▼ Code ›

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

bool searchMatrix(vector<vector<int>>& matrix, int target) {
    int n = matrix.size(), m = matrix[0].size();

    //traverse the matrix:
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            if (matrix[i][j] == target)
                return true;
        }
    }
    return false;
}

int main()
{
    vector<vector<int>> matrix = {{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}};
    searchMatrix(matrix, 8) == true ? cout << "true\n" : cout << "false\n";
}
```

Output: true

▼ Complexity Analysis ›
Time Complexity: O(N X M), where N = given row number, M = given column number.
Reason: In order to traverse the matrix, we are using nested loops running for n and m times respectively.

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

▼ Better Approach ›
▼ Algorithm / Intuition ›
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.

Algorithm:

1. We will use a loop(say i) to select a particular row at a time.
2. Next, for every row, i, we will check if it contains the target.
    1. **If matrix[i][0] <= target && target <= matrix[i][m-1]:** If this condition is met, we can conclude that row i has the possibility of containing the target.
    So, we will apply binary search on row i, and check if the 'target' is present. If it is present, we will return true from this step. Otherwise, we will return false.
3. Otherwise, after completing the traversal, we will return false.

Dry-run: *Please refer to the attached video for a detailed dry-run.*

▼ Code ›

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

bool binarySearch(vector<int>& 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[mid] == target) return true;
        else if (target > nums[mid]) low = mid + 1;
        else high = mid - 1;
    }
    return false;
}


bool searchMatrix(vector<vector<int>>& matrix, int target) {
    int n = matrix.size();
    int m = matrix[0].size();

    for (int i = 0; i < n; i++) {
        if (matrix[i][0] <= target && target <= matrix[i][m - 1]) {
            return binarySearch(matrix[i], target);
        }
    }
    return false;
}

int main()
{
    vector<vector<int>> matrix = {{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}};
    searchMatrix(matrix, 8) == true ? cout << "true\n" : cout << "false\n";
}
```

Output: true

▼ Complexity Analysis ›

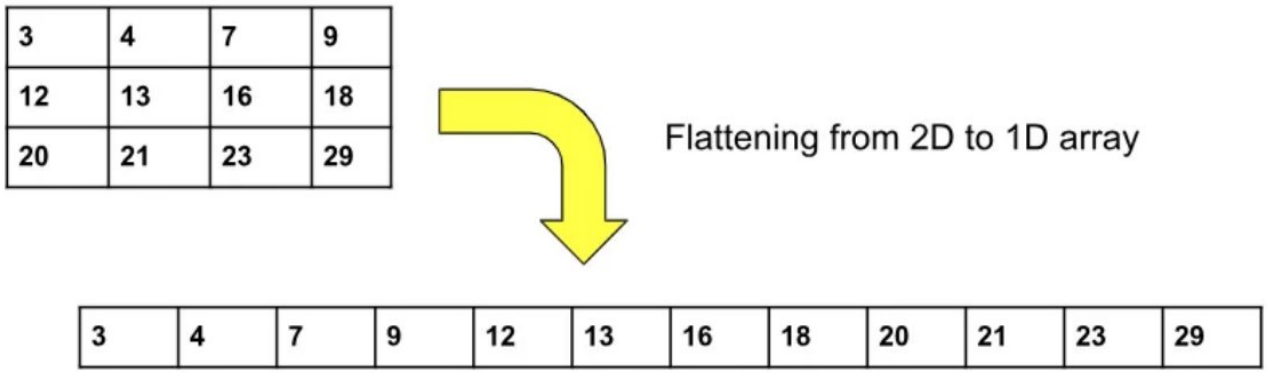
Time Complexity: O(N + logM), where N = given row number, M = given column number.
Reason: We are traversing all rows and it takes O(N) time complexity. But for all rows, we are not applying binary search rather we are only applying it once for a particular row. That is why the time complexity is O(N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

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

▼ Optimal Approach ›

▼ Algorithm / Intuition >

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.

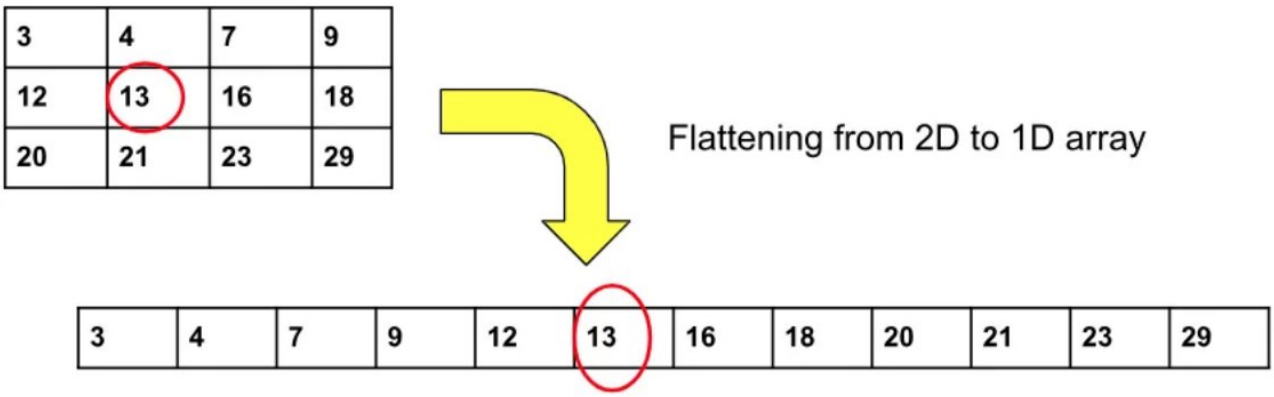
How to apply binary search on the 1D array without actually flattening the 2D matrix:

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*).



Index 5 corresponds to cell (1, 1)
row = (5 / 4) = 1 (*integer division*)
col = (5 % 4) = 1

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

Algorithm:

1. **Place the 2 pointers i.e. low and high:** Initially, we will place the pointers. The pointer low will point to 0 and the high will point to (NxM)-1.
2. **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)
3. **Eliminate the halves based on the element at index mid:** To get the element, we will convert index 'mid' to the corresponding cell using the above formula. Here no. of columns of the matrix = M.
   row = mid / M, col = mid % M.
   1. **If matrix[row][col] == target:** We should return true here, as we have found the 'target'.
   2. **If matrix[row][col] < target:** In this case, we need bigger elements. So, we will eliminate the left half and consider the right half (low = mid+1).
   3. **If matrix[row][col] > target:** In this case, we need smaller elements. So, we will eliminate the right half and consider the left half (high = mid-1).
4. Steps 2-3 will be inside a while loop and the loop will end once low crosses high (*i.e. low > high*). If we are out of the loop, we can say the target does not exist in the matrix. So, we will return false.

Dry-run: *Please refer to the attached video for a detailed dry-run.*

▼ Code ›

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

bool searchMatrix(vector<vector<int>>& matrix, int target) {
    int n = matrix.size();
    int m = matrix[0].size();

    //apply binary search:
    int low = 0, high = n * m - 1;
    while (low <= high) {
        int mid = (low + high) / 2;
        int row = mid / m, col = mid % m;
        if (matrix[row][col] == target) return true;
        else if (matrix[row][col] < target) low = mid + 1;
        else high = mid - 1;
    }
    return false;
}

int main()
{
    vector<vector<int>> matrix = {{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}};
    searchMatrix(matrix, 8) == true ? cout << "true\n" : cout << "false\n";
}
```

Output: true

▼ Complexity Analysis ›

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.

▶ Video Explanation ›

> *Special thanks to **KRITIDIPTA GHOSH** for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam,* please check out this article

Search in a row and column-wise sorted matrix

takeuforward.org/arrays/search-in-a-row-and-column-wise-sorted-matrix

August 23, 2023

Problem Statement: You have been given a 2-D array **'mat'** of size **'N x M'** where **'N'** and **'M'** denote the number of rows and columns, respectively. The elements of each row and each column are sorted in non-decreasing order.
But, the first element of a row is not necessarily greater than the last element of the previous row (if it exists).
You are given an integer **'target'**, and your task is to find if it exists in the given 'mat' or not.

Pre-requisite: Search in a 2D sorted matrix

▼ **Examples** ›

Example 1:
Input Format: N = 5, M = 5, target = 14
mat[] =

1	4	7	11	15
2	5	8	12	19
3	6	9	16	22
10	13	14	17	24
18	21	23	26	30

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[] =

1	3	7
6	13	15
14	20	21

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

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

Brute Force Approach Better Approach Optimal Approach

▼ Brute Force Approach >
▼ Algorithm / Intuition >
One key point to notice here is that the first element of a row is not necessarily greater than the last element of the previous row (if it exists). This means the matrix is not necessarily entirely sorted although each row and column is sorted in non-decreasing order.

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'.

Algorithm:

1. We will use a loop(say i) to select a particular row at a time.
2. Next, for every row, we will use another loop(say j) to traverse each column.
3. Inside the loops, we will check if the element i.e. matrix[i][j] is equal to the 'target'. If we found any matching element, we will return true.
4. Finally, after completing the traversal, if we found no matching element, we will return false.

Dry-run: *Please refer to the attached video for a detailed dry-run.*

▼ Code ›

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

bool searchElement(vector<vector<int>>& matrix, int target) {
    int n = matrix.size(), m = matrix[0].size();

    //traverse the matrix:
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            if (matrix[i][j] == target)
                return true;
        }
    }
    return false;
}

int main()
{
    vector<vector<int>> matrix = {{1, 4, 7, 11, 15}, {2, 5, 8, 12, 19},
        {3, 6, 9, 16, 22}, {10, 13, 14, 17, 24},
        {18, 21, 23, 26, 30}
    };
    searchElement(matrix, 8) == true ? cout << "true\n" : cout << "false\n";
}
```

Output: true

▼ Complexity Analysis ›

Time Complexity: O(N X M), where N = given row number, M = given column number.
Reason: In order to traverse the matrix, we are using nested loops running for n and m times respectively.

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

▼ Better Approach ›
▼ Algorithm / Intuition ›
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.

Algorithm:

1. We will use a loop(say i) to select a particular row at a time.
2. Next, for every row, i, we will check if it contains the target using binary search.
    1. After applying binary search on row, i, if we found any element equal to the target, we will return true. Otherwise, we will move on to the next row.
3. Finally, after completing all the row traversals, if we found no matching element, we will return false.

Dry-run: *Please refer to the attached video for a detailed dry-run.*

▼ Code ›

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

bool binarySearch(vector<int>& 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[mid] == target) return true;
        else if (target > nums[mid]) low = mid + 1;
        else high = mid - 1;
    }
    return false;
}

bool searchElement(vector<vector<int>>& matrix, int target) {
    int n = matrix.size();

    for (int i = 0; i < n; i++) {
        bool flag =  binarySearch(matrix[i], target);
        if (flag) return true;
    }
    return false;
}

int main()
{
    vector<vector<int>> matrix = {{1, 4, 7, 11, 15}, {2, 5, 8, 12, 19},
        {3, 6, 9, 16, 22}, {10, 13, 14, 17, 24},
        {18, 21, 23, 26, 30}
    };
    searchElement(matrix, 8) == true ? cout << "true\n" : cout << "false\n";
}
```

Output: true

▼ Complexity Analysis ›

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. And for all rows, we are applying binary search. So, the total time complexity is O(N*logM).

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

▼ Optimal Approach ›

▼ Algorithm / Intuition ›

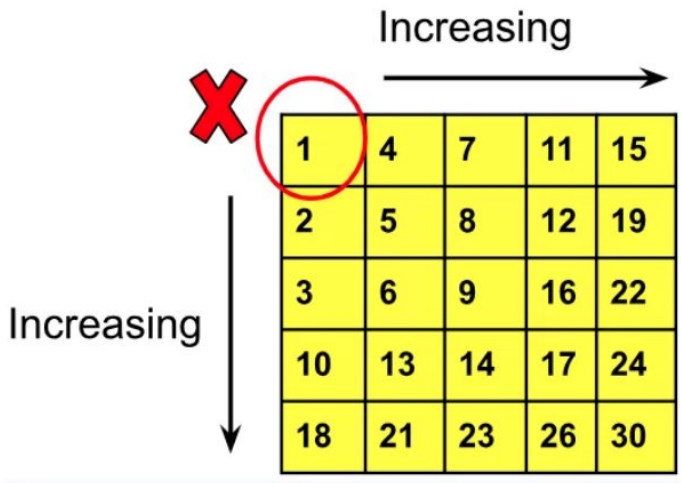
We can enhance this method by adjusting how we move through the matrix. Let's take a look at the four corners: (0, 0), (0, m-1), (n-1, 0), and (n-1, m-1). By observing these corners, we can identify variations in how we traverse the matrix.

Assume the given 'target' = 14 and given matrix =

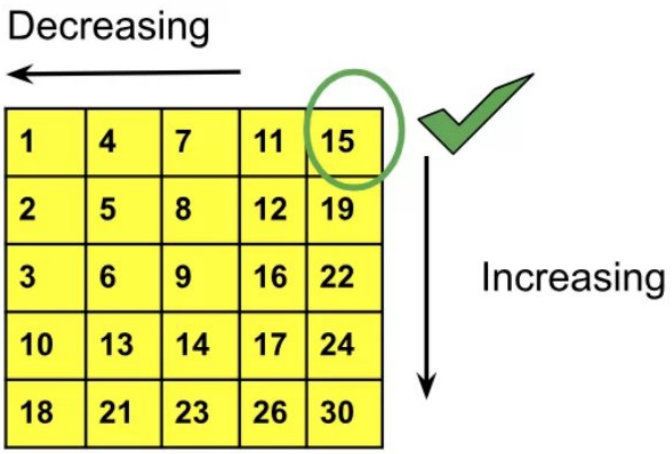
1	4	7	11	15
2	5	8	12	19
3	6	9	16	22
10	13	14	17	24
18	21	23	26	30

Observations:

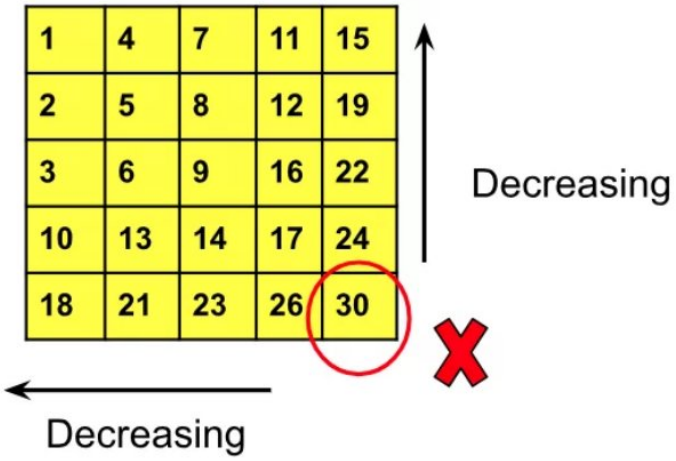
- **Cell (0, 0):** Assume we are starting traversal from (0, 0) and we are searching for 14. Now, this row and column are both sorted in increasing order. So, we cannot determine, how to move i.e. row-wise or column-wise. That is why, ***we cannot start traversal from (0, 0).***



- **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:

1. 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.

2. We will do the following steps until row < n and col >= 0(*i.e. while(row < n && col >= 0)*):
    1. **If matrix[row][col] == target:** We have found the target and so we will return true.
    2. **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.
    3. **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.
3. If we are outside the loop without getting any matching element, we will return false.

Dry-run: *Please refer to the attached video for a detailed dry-run.*

▼ Code ›

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

bool searchElement(vector<vector<int>>& matrix, int target) {
    int n = matrix.size();
    int m = matrix[0].size();
    int row = 0, col = m - 1;

    //traverse the matrix from (0, m-1):
    while (row < n && col >= 0) {
        if (matrix[row][col] == target) return true;
        else if (matrix[row][col] < target) row++;
        else col--;
    }
    return false;
}

int main()
{
    vector<vector<int>> matrix = {{1, 4, 7, 11, 15}, {2, 5, 8, 12, 19},
        {3, 6, 9, 16, 22}, {10, 13, 14, 17, 24},
        {18, 21, 23, 26, 30}
    };
    searchElement(matrix, 8) == true ? cout << "true\n" : cout << "false\n";
}
```

Output: true

▼ Complexity Analysis ›

Time Complexity: O(N+M), where N = given row number, M = given column number.
Reason: We are starting traversal from (0, M-1), and at most, we can end up being in the cell (M-1, 0). So, the total distance can be at most (N+M). So, the time complexity is O(N+M).

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

▶ Video Explanation ›

> *Special thanks to **KRITIDIPTA GHOSH** for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam,* please check out this article

Median of Row Wise Sorted Matrix

takeuforward.org/data-structure/median-of-row-wise-sorted-matrix

January 26, 2022

In this article we will solve the most asked coding interview problem: Median of Row Wise Sorted Matrix

Problem Statement: Given a row-wise sorted matrix of size r*c, where r is no. of rows and c is no. of columns, find the median in the given matrix.

Assume – r*c is odd

Examples:

```
Example 1:
Input:
r = 3 , c = 3
1 4 9
2 5 6
3 8 7
Output: Median = 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:
r = 3 , c = 3
1 3 8
2 3 4
1 2 5
Output: Median = 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
```

Solution

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

Solution 1:

Approach 1: Brute Force Approach

The approach is very simple, just fill all elements in a linear array sort the array using the sort function, and then find the middle value (**Median)**.

For Eg,

For the given matrix

1 3 8

2 3 4

1 2 5

We find the sorted linear array as 1 1 2 2 3 3 4 5 8

Now, the middle element of the array is 3.

Code:

- C++ Code
- Java Code

```
#include <bits/stdc++.h>
using namespace std;
// Function to find median  of row wise sorted matrix
int Findmedian(int arr[3][3], int row, int col)
{
  int median[row * col];
  int index = 0;
  for (int i = 0; i < row; i++)
  {
    for (int j = 0; j < col; j++)
    {
      median[index] = arr[i][j];
      index++;
    }
  }

  return median[(row * col) / 2];
}
int main()
{
  int row = 3, col = 3;
  int arr[3][3] = {{1, 3, 8},
                   {2, 3, 4},
                   {1, 2, 5}};
  cout <<"The median of the row-wise sorted matrix is: "<<Findmedian
  (arr, row, col) << endl;
  return 0;
}
```

Output: The median of the row-wise sorted matrix is: 3

Time Complexity: O(row*col log(row*col)) for sorting the array where r*c denotes the number of elements in the linear array.

Space Complexity: O(row*col) for storing elements in the linear array

Approach 2: Efficient Approach (Using Binary Search)

Step 1: Find the minimum and maximum element in the given array. By just traversing the first column, we find the minimum element and by just traversing the last column, we find the maximum element.

Step 2: Now find the middle element of the array one by one and check in the matrix how many elements are present in the matrix.

Three cases can occur:-

- If count == (r*c+1)/2 , so it may be the answer still we mark max as mid since we are not referring indices , we are referring the elements and 5 elements can be smaller than 6 also and 7 also , so to confirm we mark max as mid.
- If count<(r*c+1)/2 , we mark min as mid+1 since curr element or elements before it cannot be the answer.
- If count>(r*c+1)/2 , we mark max as mid , since elements after this can only be the answer now.

Let's discuss the approach with an example

For eg, the given array is

Row wise sorted Matrix

2	3	5
2	3	4
1	7	9

Min – 1
Max – 9

Step 1 : Mid of min=1 and max=9 is 5

No. of elements smaller than equal to 5 are 1,2,2,3,3,4,5 which are 7
Since 7>5 so , max will be updated as mid = 5

Step 2 : Mid of min=1 and max=5 is 3

No. of elements smaller than and equal to 3 are 1,2,2,3,3 which are 5
Since 5==5 , so update max as mid = 3

Step 3 : Mid of min=1 and max=3 is 2

No. of elements smaller than and equal to 2 are 1,2,2 which are 3
Since 3<5 so , min will be updated as mid+1 = 3

Since min=max we come out of loop and return min =3

Code:

- C++ Code
- Java Code

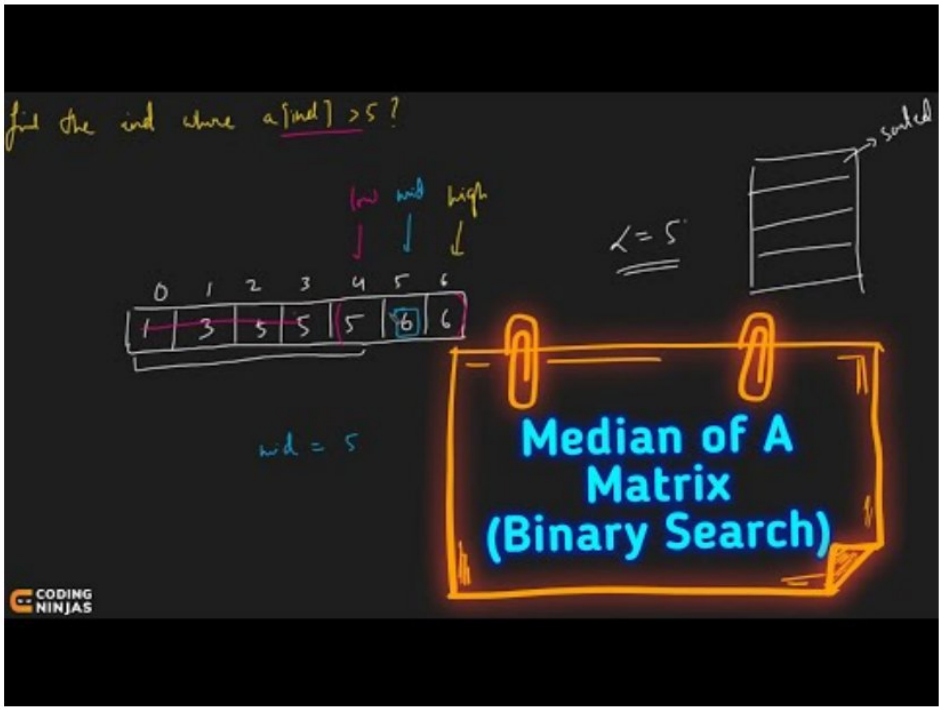
```
#include <bits/stdc++.h>
using namespace std;
int countSmallerThanMid(vector<int> &row, int mid)
{
  int l = 0, h = row.size() - 1;
  while (l <= h)
  {
    int md = (l + h) >> 1;
    if (row[md] <= mid)
    {
      l = md + 1;
    }
    else
    {
      h = md - 1;
    }
  }
  return l;
}
int findMedian(vector<vector<int>> &A)
{
  int low = 1;
  int high = 1e9;
  int n = A.size();
  int m = A[0].size();
  while (low <= high)
  {
    int mid = (low + high) >> 1;
    int cnt = 0;
    for (int i = 0; i < n; i++)
    {
      cnt += countSmallerThanMid(A[i], mid);
    }
    if (cnt <= (n * m) / 2)
      low = mid + 1;
    else
      high = mid - 1;
  }
  return low;
}
int main()
{
  int row = 3, col = 3;
  vector<vector<int>> arr = {{1, 3, 8},
                             {2, 3, 4},
                             {1, 2, 5}};
  cout <<"The median of the row-wise sorted matrix is: "<<findMedian(arr)<<endl;
  return 0;
}
```

Output: The median of the row-wise sorted matrix is: 3

Time Complexity: O(row*log col) since the upper bound function takes log c time.

Space Complexity: O(1) since no extra space is required.

> *Special thanks to* ***Gurmeet Singh*** *for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam,* please check out this article



Watch Video At: https://youtu.be/63fPPOdIr2c