**Binary Search**

**Introduction**

In this card, we are going to help you understand the general concept of Binary Search.

What is Binary Search?

Binary Search is one of the most fundamental and useful algorithms in Computer Science. It describes the process of searching for a specific value in an ordered collection.

Terminology used in Binary Search:

* Target - the value that you are searching for
* Index - the current location that you are searching
* Left, Right - the indicies from which we use to maintain our search Space
* Mid - the index that we use to apply a condition to determine if we should search left or right

Other Binary Search Defintions:

([Wikipedia](https://en.wikipedia.org/wiki/Binary_search_algorithm))

Background

 How does it work?

 Binary Search

 Identification and Template Introduction

Template I

 Binary Search Template I

 Sqrt(x)

 Guess Number Higher or Lower

 Search in Rotated Sorted Array

Template II

 Binary Search Template II

 First Bad Version

 Find Peak Element

 Find Minimum in Rotated Sorted Array

Template III

 Binary Search Template III

 Search for a Range

 Find K Closest Elements

 Find Peak Element

Template Analysis

 Binary Search Template Analysis

 Closest Binary Search Tree Value

 Search in a Sorted Array of Unknown Size

Conclusion

 Pow(x, n)

 Valid Perfect Square

 Find Smallest Letter Greater Than Target

More Practices

 Find Minimum in Rotated Sorted Array

 Find Minimum in Rotated Sorted Array II

 Intersection of Two Arrays

 Intersection of Two Arrays II

 Two Sum II - Input array is sorted

More Practices II

 Find the Duplicate Number

 Median of Two Sorted Arrays

 Find K-th Smallest Pair Distance

 Split Array Largest Sum

**How does it work?**

In its simplest form, Binary Search operates on a contiguous sequence with a specified left and right index. This is called the Search Space. Binary Search maintains the left, right, and middle indicies of the search space and compares the search target or applies the search condition to the middle value of the collection; if the condition is unsatisfied or values unequal, the half in which the target cannot lie is eliminated and the search continues on the remaining half until it is successful. If the search ends with an empty half, the condition cannot be fulfilled and target is not found.

In the following chapters, we will review how to identify Binary Search problems, reasons why we use Binary Search, and the 3 different Binary Search templates that you might be previously unaware of. Since Binary Search is a common interview topic, we will also categorize practice problems to different templates so you can practice using each.

**Note:**

Binary Search can take many alternate forms and might not always be as straight forward as searching for a specific value. Sometimes you will have to apply a specific condition or rule to determine which side (left or right) to search next.

We will provide examples in the coming chapters. First, could you try write a binary search algorithm yourself?

**Binary Search**

Given a **sorted** (in ascending order) integer array nums of n elements and a target value, write a function to search target in nums. If target exists, then return its index, otherwise return -1.

**Example 1:**

**Input:** nums = [-1,0,3,5,9,12], target = 9

**Output:** 4

**Explanation:** 9 exists in nums and its index is 4

**Example 2:**

**Input:** nums = [-1,0,3,5,9,12], target = 2

**Output:** -1

**Explanation:** 2 does not exist in nums so return -1

**Note:**

1. You may assume that all elements in nums are unique.
2. n will be in the range [1, 10000].
3. The value of each element in nums will be in the range [-9999, 9999].

## Solution

#### **Approach 1: Binary Search**

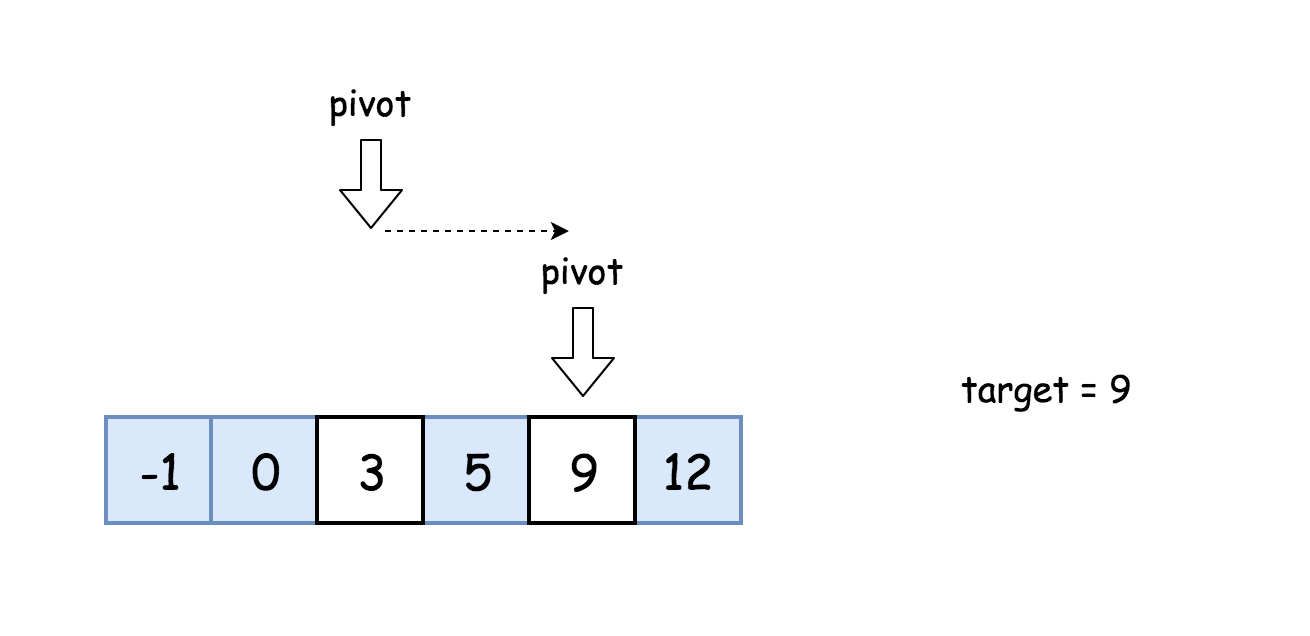
**Intuition**

[Binary search](https://leetcode.com/explore/learn/card/binary-search/) is a textbook algorithm based on the idea to compare the target value to the middle element of the array.

If the target value is equal to the middle element - we're done.

If the target value is smaller - continue to search on the left.

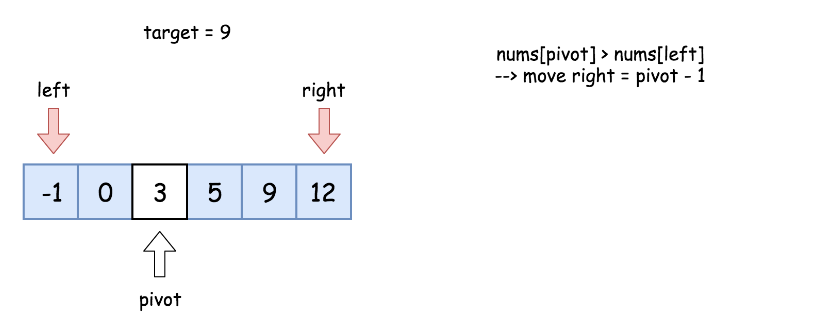
If the target value is larger - continue to search on the right.

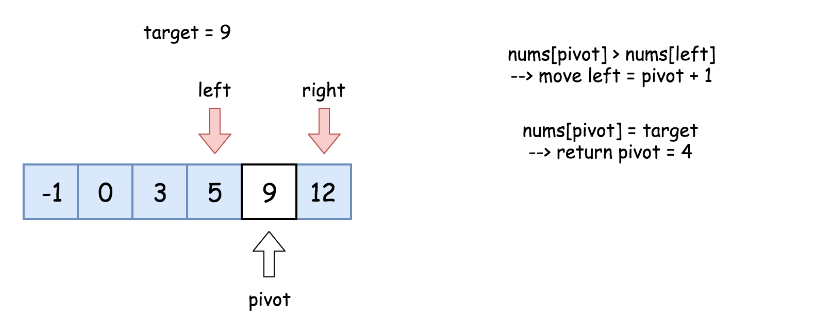


**Algorithm**

* Initialise left and right pointers : left = 0, right = n - 1.
* While left <= right :
  + Compare middle element of the array nums[pivot] to the target value target.
    - If the middle element is the target target = nums[pivot] : return pivot.
    - If the target is not yet found :
      * If target < nums[pivot], continue the search on the left right = pivot - 1.
      * Else continue the search on the right left = pivot + 1.

**Implementation**





|  |
| --- |
| class Solution {  public int search(int[] nums, int target) {  int pivot, left = 0, right = nums.length - 1;  while (left <= right) {  pivot = left + (right - left) / 2;  if (nums[pivot] == target) return pivot;  if (target < nums[pivot]) right = pivot - 1;  else left = pivot + 1;  }  return -1;  }  } |

**Complexity Analysis**

* Time complexity : O(log*N*).

Let's compute time complexity with the help of [master theorem](https://en.wikipedia.org/wiki/Master_theorem_(analysis_of_algorithms)) . The equation represents dividing the problem up into a*a* subproblems of size  time. Here at step there is only one subproblem a = 1, its size is a half of the initial problem b = 2, and all this happens in a constant time d = 0. That means that log*b*​*a*=*d* and hence we're dealing with [case 2](https://en.wikipedia.org/wiki/Master_theorem_(analysis_of_algorithms)#Case_2_example) that results in  time complexity.

* Space complexity : O(1) since it's a constant space solution.

**Identification and Template Introduction**

**How do we identify Binary Search?**

As mentioned in earlier, Binary Search is an algorithm that *divides the search space in 2* after every comparison. Binary Search should be considered every time you need to search for an index or element in a collection. If the collection is unordered, we can always sort it first before applying Binary Search.

**3 Parts of a Successful Binary Search**

Binary Search is generally composed of 3 main sections:

1. ***Pre-processing*** - Sort if collection is unsorted.
2. ***Binary Search*** - Using a loop or recursion to divide search space in half after each comparison.
3. ***Post-processing*** - Determine viable candidates in the remaining space.

**3 Templates for Binary Search**

When we first learned Binary Search, we might struggle. We might study hundreds of Binary Search problems online and each time we looked at a developer's code, it seemed to be implemented slightly differently. Although each implementation divided the problem space in 1/2 at each step, one had numerous questions:

* Why was it implemented slightly differently?
* What was the developer thinking?
* Which way was easier?
* Which way is better?

After many failed attempts and lots of hair-pulling, we found 3 main templates for Binary Search. To prevent hair-pulling and to make it easier for new developers to learn and understand, we have provided them in the next chapter.

## Template I

This chapter shows a snippet of code for Template #1. It gives a brief explanation of when to use the template and highlights the key syntax differences between the 3 templates.

**Binary Search Template I**

|  |
| --- |
| int binarySearch(int[] nums, int target){  if(nums == null || nums.length == 0)  return -1;  int left = 0, right = nums.length - 1;  while(left <= right){  // Prevent (left + right) overflow  int mid = left + (right - left) / 2;  if(nums[mid] == target){ return mid; }  else if(nums[mid] < target) { left = mid + 1; }  else { right = mid - 1; }  }  // End Condition: left > right  return -1;  } |

Template #1 is the most basic and elementary form of Binary Search. It is the standard Binary Search Template that most high schools or universities use when they first teach students computer science. Template #1 is used to search for an element or condition which can be determined by *accessing a single index* in the array.

**Key Attributes:**

* Most basic and elementary form of Binary Search
* Search Condition can be determined without comparing to the element's neighbors (or use specific elements around it)
* No post-processing required because at each step, you are checking to see if the element has been found. If you reach the end, then you know the element is not found

**Distinguishing Syntax:**

* Initial Condition: left = 0, right = length-1
* Termination: left > right
* Searching Left: right = mid-1
* Searching Right: left = mid+1

**Sqrt(x)**

Given a non-negative integer x, compute and return *the square root of* x.

Since the return type is an integer, the decimal digits are **truncated**, and only **the integer part** of the result is returned.

**Example 1:**

**Input:** x = 4

**Output:** 2

**Example 2:**

**Input:** x = 8

**Output:** 2

**Explanation:** The square root of 8 is 2.82842..., and since the decimal part is truncated, 2 is returned.

**Constraints:**

* 0 <= x <= 231 - 1

   Hide Hint #1

Try exploring all integers. (Credits: @annujoshi)

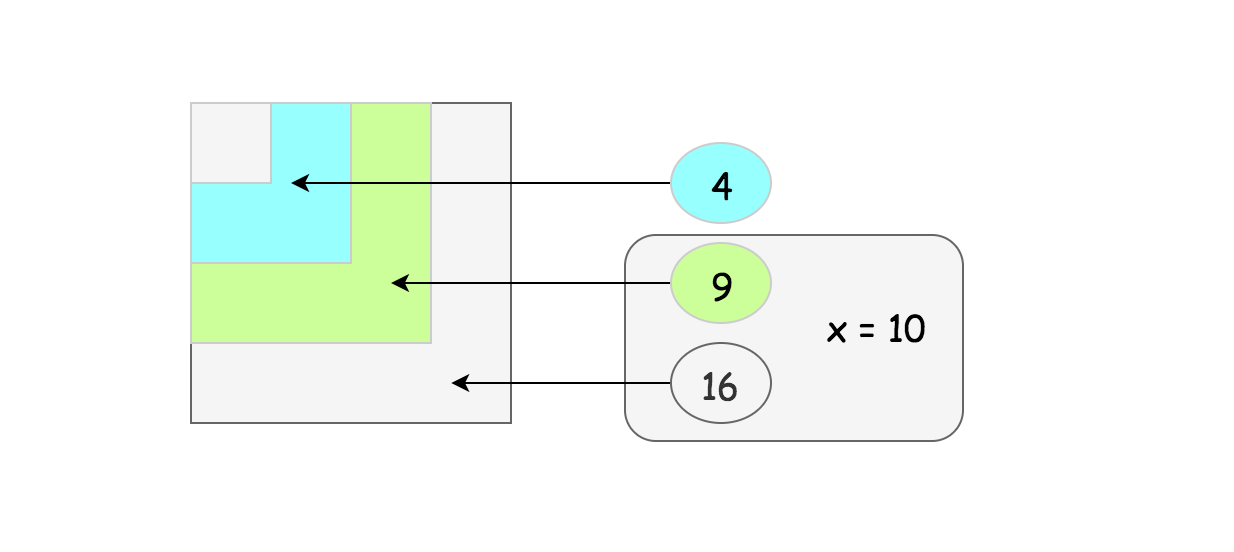
   Hide Hint #2

Use the sorted property of integers to reduced the search space. (Credits: @annujoshi)

## Solution

#### **Integer Square Root**

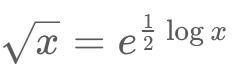
The value a*a* we're supposed to compute could be defined as a^2 \le x < (a + 1)^2*a*2≤*x*<(*a*+1)2. It is called integer square root. From geometrical points of view, it's the side of the largest integer-side square with a surface less than x.



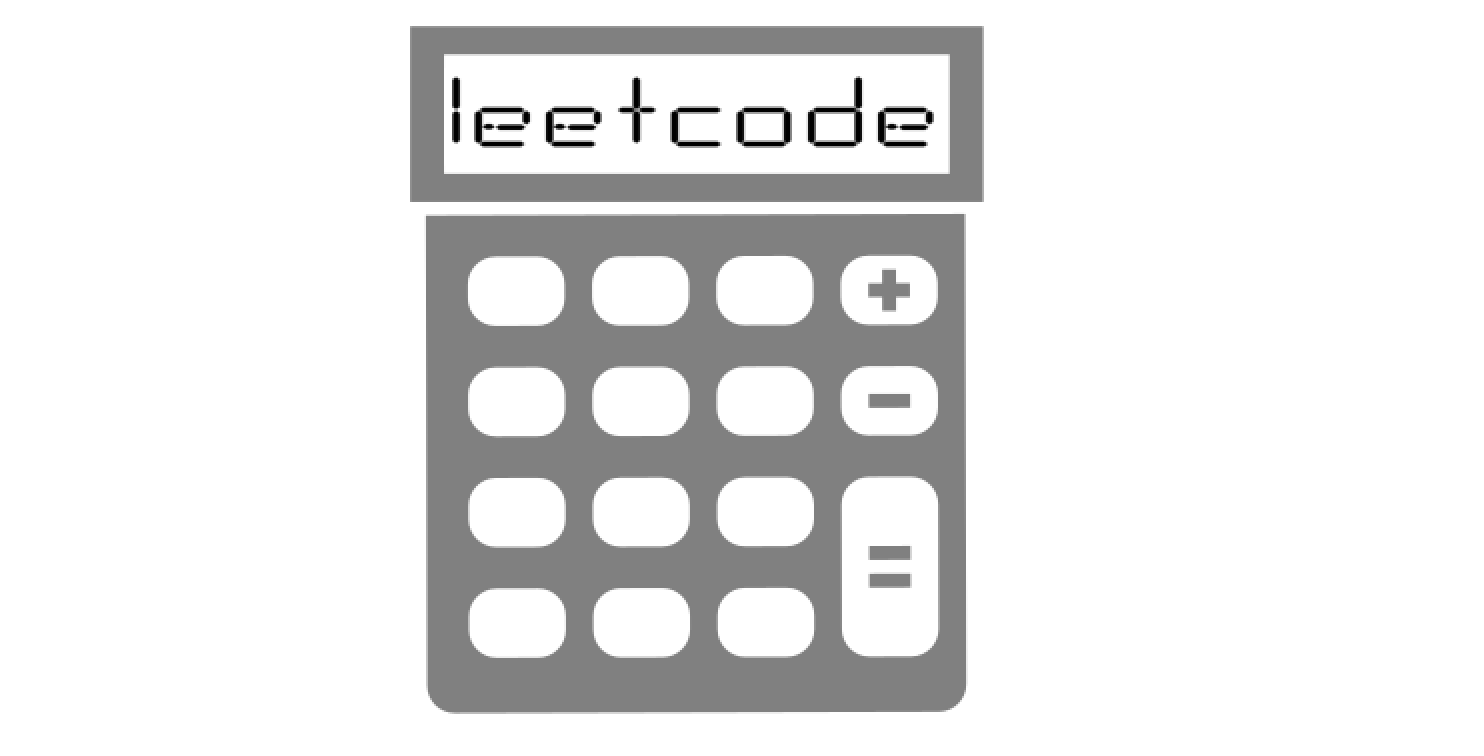
#### **Approach 1: Pocket Calculator Algorithm**

Before going to the serious stuff, let's first have some fun and implement the [algorithm used by the pocket calculators](https://en.wikipedia.org/wiki/Methods_of_computing_square_roots#Exponential_identity).

Usually a pocket calculator computes well exponential functions and natural logarithms by having logarithm tables hardcoded or by the other means. Hence the idea is to reduce the square root computation to these two algorithms as well



That's some sort of cheat because of non-elementary function usage but it's how that actually works in a real life.



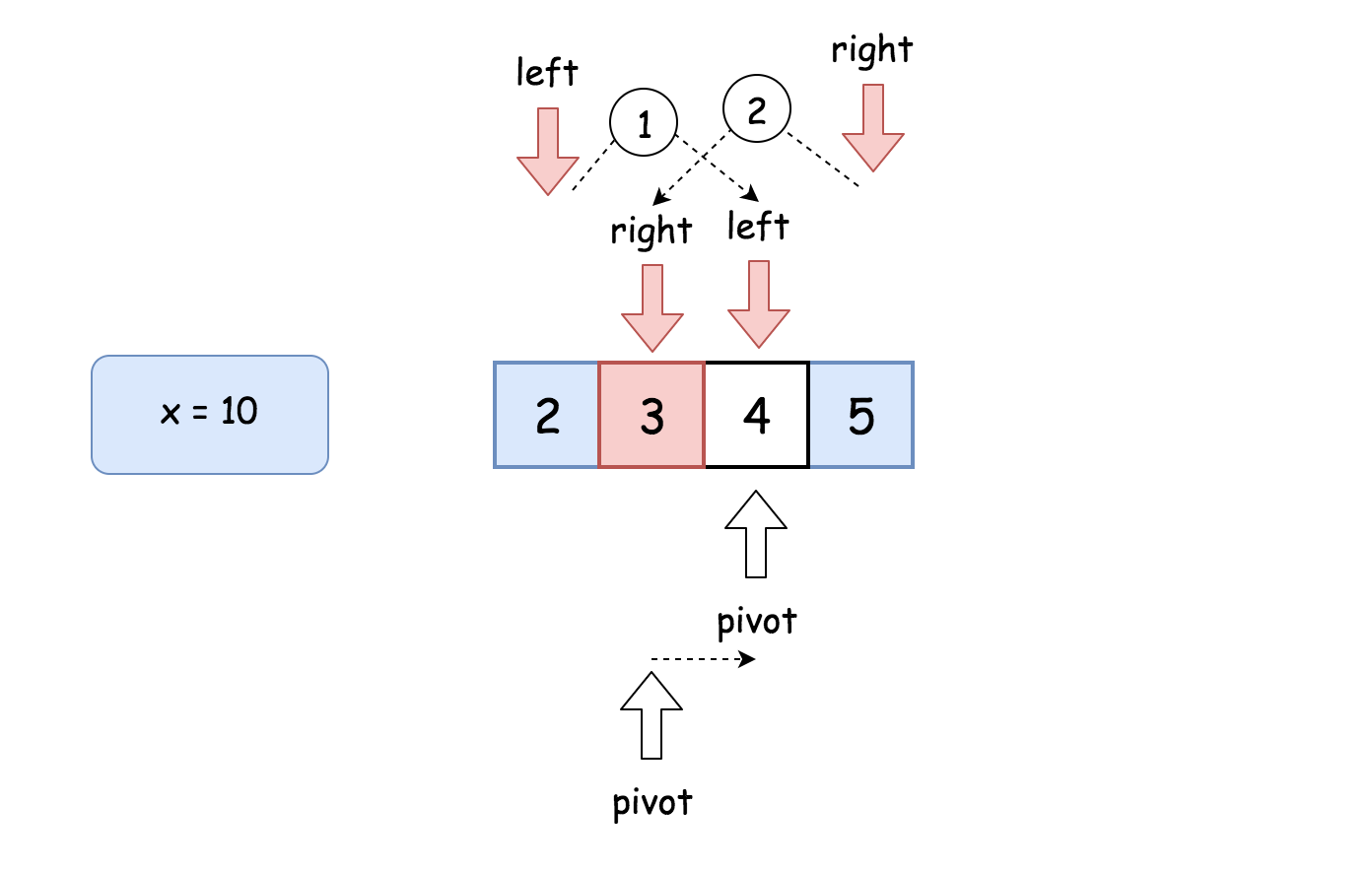
**Implementation**

|  |
| --- |
| class Solution {  public int mySqrt(int x) {  if (x < 2) return x;  int left = (int)Math.pow(Math.E, 0.5 \* Math.log(x));  int right = left + 1;  return (long)right \* right > x ? left : right;  }  } |

#### **Approach 2: Binary Search**

**Intuition**

Let's go back to the interview context. For x \ge 2*x*≥2 the square root is always smaller than x / 2*x*/2 and larger than 0 : 0 < a < x / 20<*a*<*x*/2.  
Since a*a* is an integer, the problem goes down to the iteration over the sorted set of integer numbers. Here the binary search enters the scene.



**Algorithm**

* If x < 2, return x.
* Set the left boundary to 2, and the right boundary to x / 2.
* While left <= right:
  + Take num = (left + right) / 2 as a guess. Compute num \* num and compare it with x:
    - If num \* num > x, move the right boundary right = pivot -1
    - Else, if num \* num < x, move the left boundary left = pivot + 1
    - Otherwise num \* num == x, the integer square root is here, let's return it
* Return right

**Implementation**

|  |
| --- |
| class Solution {  public int mySqrt(int x) {  if (x < 2) return x;  long num;  int pivot, left = 2, right = x / 2;  while (left <= right) {  pivot = left + (right - left) / 2;  num = (long)pivot \* pivot;  if (num > x) right = pivot - 1;  else if (num < x) left = pivot + 1;  else return pivot;  }  return right;  }  } |

**Complexity Analysis**

* Time complexity :  O(log*N*).
* Space complexity :  O(1).

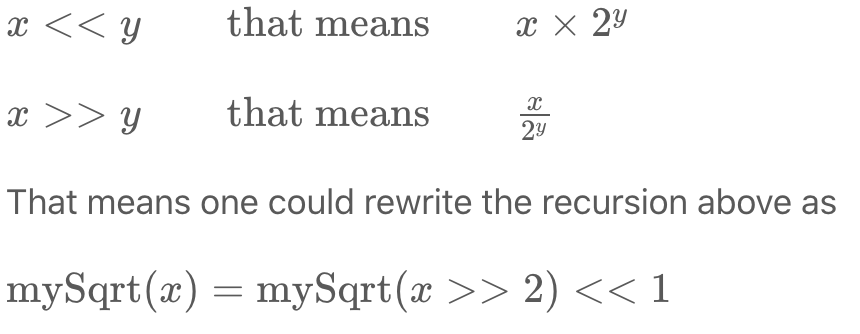
#### **Approach 3: Recursion + Bit Shifts**

**Intuition**

Let's use recursion. Bases cases are . Now the idea is to decrease x*x* recursively at each step to go down to the base cases.

How to go down?

For example, let's notice that , and hence square root could be computed recursively as

  
One could already stop here, but let's use [left and right shifts](https://wiki.python.org/moin/BitwiseOperators), which are quite fast manipulations with bits  


in order to fasten up the computations.

**Implementation**

|  |
| --- |
| class Solution {  public int mySqrt(int x) {  if (x < 2) return x;  int left = mySqrt(x >> 2) << 1;  int right = left + 1;  return (long)right \* right > x ? left : right;  }  } |

**Complexity Analysis**

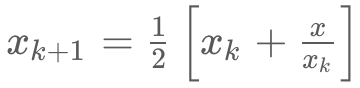
* Time complexity : O(log*N*).
* Space complexity : O(log*N*) to keep the recursion stack.

#### **Approach 4: Newton's Method**

**Intuition**

One of the best and widely used methods to compute sqrt is [Newton's Method](https://en.wikipedia.org/wiki/Newton%27s_method). Here we'll implement the version without the seed trimming to keep things simple. However, seed trimming is a bit of math and lot of fun, so [here is a link](https://en.wikipedia.org/wiki/Methods_of_computing_square_roots#Rough_estimation) if you'd like to dive in.

Let's keep the [mathematical proofs](https://en.wikipedia.org/wiki/Newton%27s_method) outside of the article and just use the textbook fact that the set



converges to . Then the things are straightforward: define that error should be less than 1 and proceed iteratively.

|  |
| --- |
| class Solution {  public int mySqrt(int x) {  if (x < 2) return x;  double x0 = x;  double x1 = (x0 + x / x0) / 2.0;  while (Math.abs(x0 - x1) >= 1) {  x0 = x1;  x1 = (x0 + x / x0) / 2.0;  }  return (int)x1;  }  } |

**Complexity Analysis**

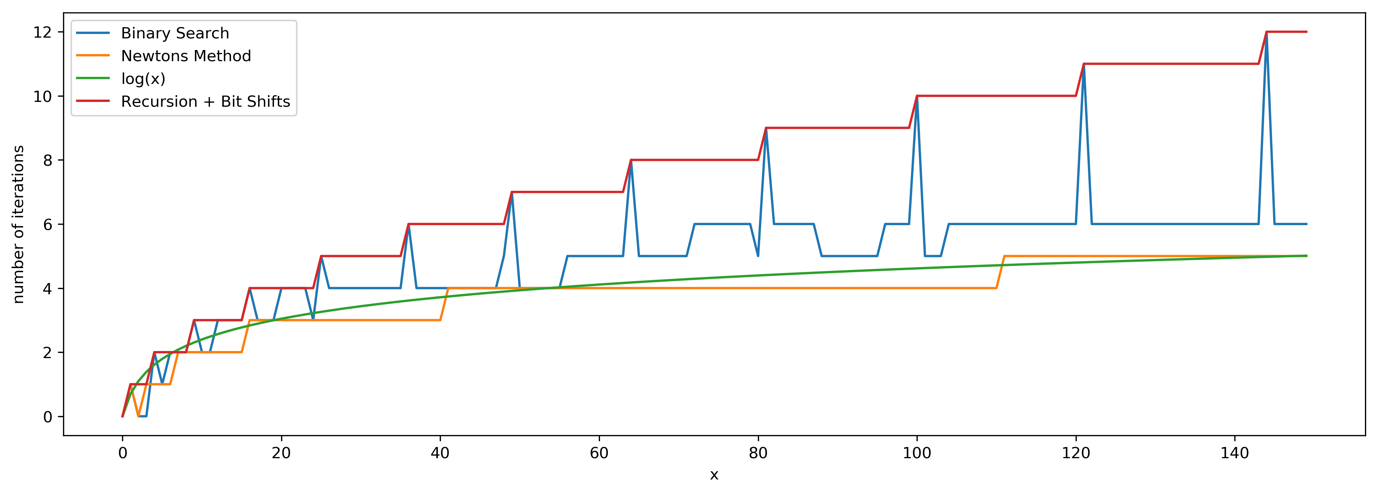
* Time complexity :  O(log*N*) since the set converges quadratically.
* Space complexity :  O(1).

#### **Compare Approaches 2, 3 and 4**

Here we have three algorithms with a time performance  O(log*N*), and it's a bit confusing.

Which one is performing less iterations?

Let's run tests for the range of x in order to check that. Here are the results. The best one is Newton's method.



**Guess Number Higher or Lower**

We are playing the Guess Game. The game is as follows:

I pick a number from 1 to n. You have to guess which number I picked.

Every time you guess wrong, I will tell you whether the number I picked is higher or lower than your guess.

You call a pre-defined API int guess(int num), which returns 3 possible results:

* -1: The number I picked is lower than your guess (i.e. pick < num).
* 1: The number I picked is higher than your guess (i.e. pick > num).
* 0: The number I picked is equal to your guess (i.e. pick == num).

Return *the number that I picked*.

**Example 1:**

**Input:** n = 10, pick = 6

**Output:** 6

**Example 2:**

**Input:** n = 1, pick = 1

**Output:** 1

**Example 3:**

**Input:** n = 2, pick = 1

**Output:** 1

**Example 4:**

**Input:** n = 2, pick = 2

**Output:** 2

**Constraints:**

* 1 <= n <= 231 - 1
* 1 <= pick <= n

## Solution

#### **Approach 1: Brute Force**

We check every number from 1 to n-1 and pass it to the guess*guess* function. The number for which a 0 is returned is the required answer.

|  |
| --- |
| /\* The guess API is defined in the parent class GuessGame.  @param num, your guess  @return -1 if my number is lower, 1 if my number is higher, otherwise return 0  int guess(int num); \*/  public class Solution extends GuessGame {  public int guessNumber(int n) {  for (int i = 1; i < n; i++)  if (guess(i) == 0)  return i;  return n;  }  } |

**Complexity Analysis**

* Time complexity : *O*(*n*). We scan all the numbers from 1 to n.
* Space complexity : *O*(1). No extra space is used.

#### **Approach 2: Using Binary Search**

**Algorithm**

We can apply Binary Search to find the given number. We start with the mid number. We pass that number to the *guess* function. If it returns a -1, it implies that the guessed number is larger than the required one. Thus, we use Binary Search for numbers lower than itself. Similarly, if it returns a 1, we use Binary Search for numbers higher than itself.

|  |
| --- |
| /\* The guess API is defined in the parent class GuessGame.  @param num, your guess  @return -1 if my number is lower, 1 if my number is higher, otherwise return 0  int guess(int num); \*/  public class Solution extends GuessGame {  public int guessNumber(int n) {  int low = 1;  int high = n;  while (low <= high) {  int mid = low + (high - low) / 2;  int res = guess(mid);  if (res == 0)  return mid;  else if (res < 0)  high = mid - 1;  else  low = mid + 1;  }  return -1;  }  } |

**Complexity Analysis**

* Time complexity : *O*(log2​*n*). Binary Search is used.
* Space complexity : *O*(1). No extra space is used.

#### **Approach 3: Ternary Search**

**Algorithm**

In Binary Search, we choose the middle element as the pivot in splitting. In Ternary Search, we choose two pivots (say *m*1 and *m*2) such that the given range is divided into three equal parts. If the required number *num* is less than *m*1 then we apply ternary search on the left segment of *m*1. If *num* lies between *m*1 and *m*2, we apply ternary search between *m*1 and *m*2. Otherwise we will search in the segment right to *m*2.

|  |
| --- |
| /\* The guess API is defined in the parent class GuessGame.  @param num, your guess  @return -1 if my number is lower, 1 if my number is higher, otherwise return 0  int guess(int num); \*/  public class Solution extends GuessGame {  public int guessNumber(int n) {  int low = 1;  int high = n;  while (low <= high) {  int mid1 = low + (high - low) / 3;  int mid2 = high - (high - low) / 3;  int res1 = guess(mid1);  int res2 = guess(mid2);  if (res1 == 0)  return mid1;  if (res2 == 0)  return mid2;  else if (res1 < 0)  high = mid1 - 1;  else if (res2 > 0)  low = mid2 + 1;  else {  low = mid1 + 1;  high = mid2 - 1;  }  }  return -1;  }  } |

**Complexity Analysis**

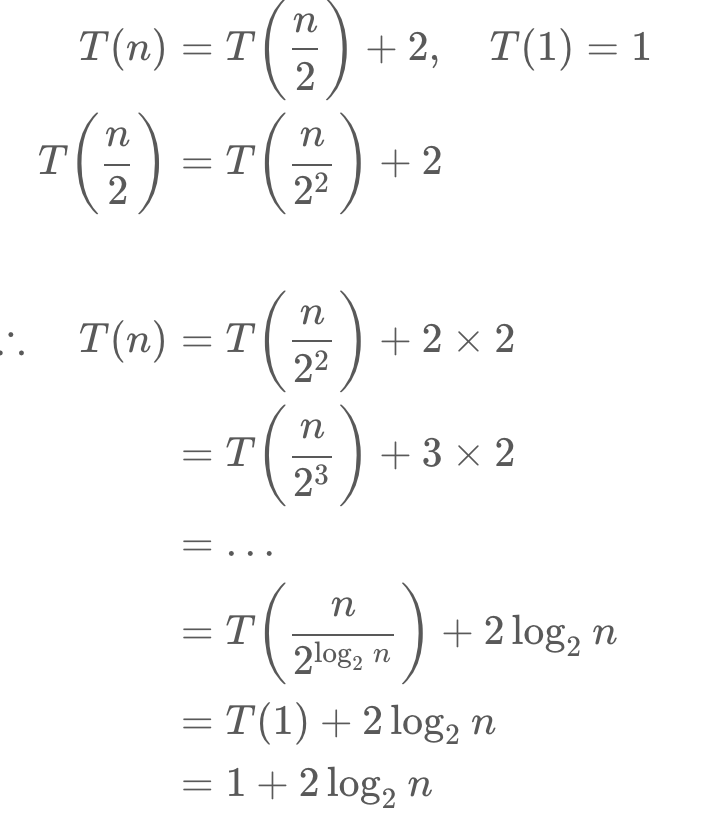
* Time complexity : *O*(log3​*n*). Ternary Search is used.
* Space complexity : *O*(1). No extra space is used.

## Follow up

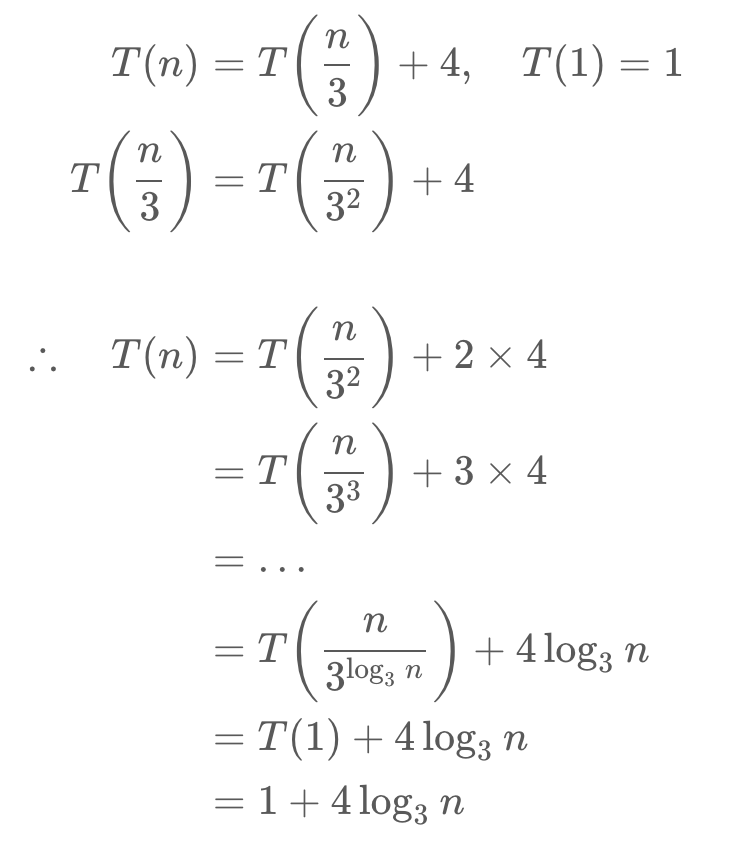
It seems that ternary search is able to terminate earlier compared to binary search. But why is binary search more widely used?

#### **Comparisons between Binary Search and Ternary Search**

Ternary Search is worse than Binary Search. The following outlines the recursive formula to count comparisons of Binary Search in the worst case.



The following outlines the recursive formula to count comparisons of Ternary Search in the worst case.



As shown above, the total comparisons in the worst case for ternary and binary search are 1+4log3​*n* and 1+2log2​*n* comparisons respectively. To determine which is larger, we can just look at the expression 2log3​*n* and log2​*n* . The expression 2log3​*n* can be written as . Since the value of ​ is greater than one, Ternary Search does more comparisons than Binary Search in the worst case.

**Search in Rotated Sorted Array**

You are given an integer array nums sorted in ascending order (with **distinct** values), and an integer target.

Suppose that nums is rotated at some pivot unknown to you beforehand (i.e., [0,1,2,4,5,6,7] might become [4,5,6,7,0,1,2]).

*If target is found in the array return its index, otherwise, return -1.*

**Example 1:**

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

**Output:** 4

**Example 2:**

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

**Output:** -1

**Example 3:**

**Input:** nums = [1], target = 0

**Output:** -1

**Constraints:**

* 1 <= nums.length <= 5000
* -104 <= nums[i] <= 104
* All values of nums are **unique**.
* nums is guaranteed to be rotated at some pivot.
* -104 <= target <= 104

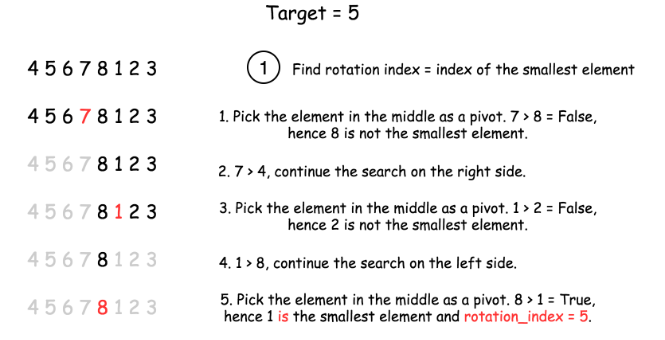
## Solution

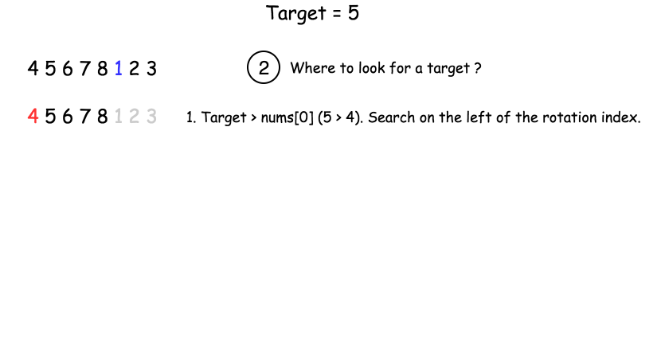
#### **Approach 1: Binary search**

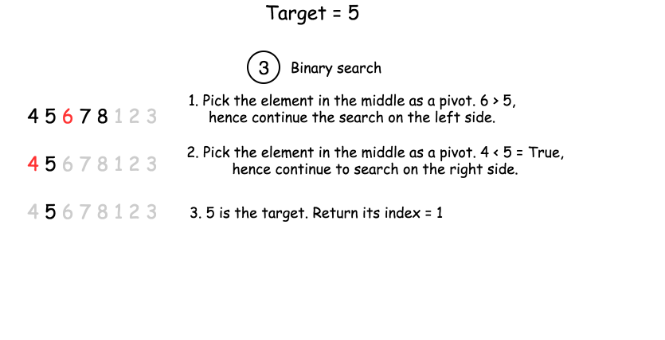
The problem is to implement a search in O(log*N*) time that gives an idea to use a binary search.

The algorithm is quite straightforward :

* Find a rotation index rotation\_index, i.e. index of the smallest element in the array. Binary search works just perfect here.
* rotation\_index splits array in two parts. Compare nums[0] and target to identify in which part one has to look for target.
* Perform a binary search in the chosen part of the array.







|  |
| --- |
| class Solution {  int [] nums;  int target;  public int find\_rotate\_index(int left, int right) {  if (nums[left] < nums[right])  return 0;  while (left <= right) {  int pivot = (left + right) / 2;  if (nums[pivot] > nums[pivot + 1])  return pivot + 1;  else {  if (nums[pivot] < nums[left])  right = pivot - 1;  else  left = pivot + 1;  }  }  return 0;  }  public int search(int left, int right) {  /\*  Binary search  \*/  while (left <= right) {  int pivot = (left + right) / 2;  if (nums[pivot] == target)  return pivot;  else {  if (target < nums[pivot])  right = pivot - 1;  else  left = pivot + 1;  }  }  return -1;  }  public int search(int[] nums, int target) {  this.nums = nums;  this.target = target;  int n = nums.length;  if (n == 1)  return this.nums[0] == target ? 0 : -1;  int rotate\_index = find\_rotate\_index(0, n - 1);  // if target is the smallest element  if (nums[rotate\_index] == target)  return rotate\_index;  // if array is not rotated, search in the entire array  if (rotate\_index == 0)  return search(0, n - 1);  if (target < nums[0])  // search in the right side  return search(rotate\_index, n - 1);  // search in the left side  return search(0, rotate\_index);  }  } |

**Complexity Analysis**

* Time complexity :  O(log*N*).
* Space complexity :  O(1).

#### **Approach 2: One-pass Binary Search**

Instead of going through the input array in two passes, we could achieve the goal in one pass with an revised binary search.

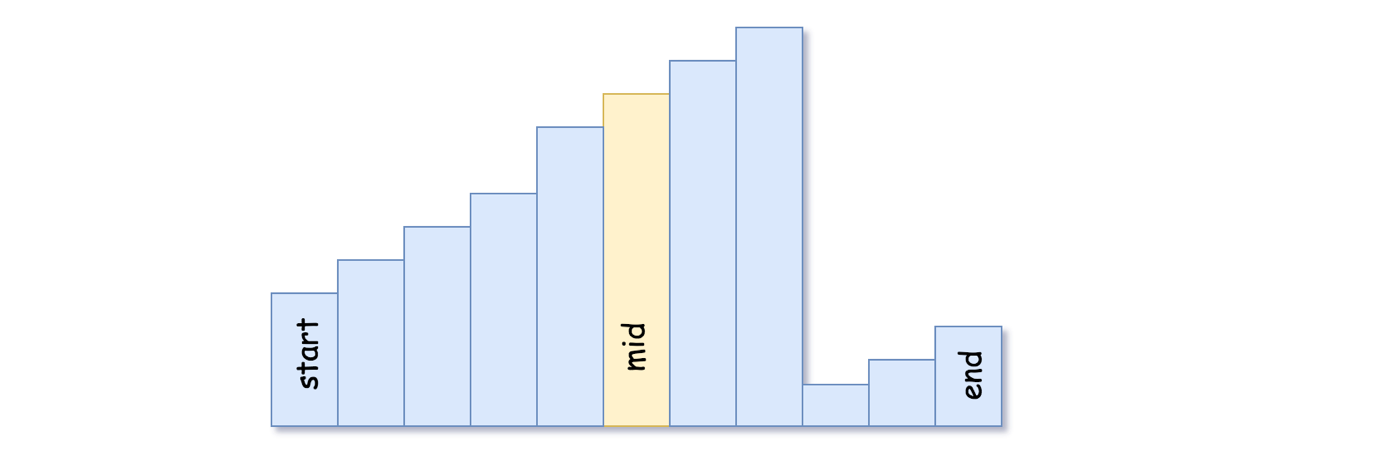
The idea is that we add some additional condition checks in the normal binary search in order to better narrow down the scope of the search.

**Algorithm**

As in the normal binary search, we keep two pointers (i.e. start and end) to track the search scope. At each iteration, we reduce the search scope into half, by moving either the start or end pointer to the middle (i.e. mid) of the previous search scope.

Here are the detailed breakdowns of the algorithm:

* Initiate the pointer start to 0, and the pointer end to n - 1.
* Perform standard binary search. While start <= end:
  + Take an index in the middle mid as a pivot.
  + If nums[mid] == target, the job is done, return mid.
  + Now there could be two situations:
    - Pivot element is larger than the first element in the array, i.e. the subarray from the first element to the pivot is non-rotated, as shown in the following graph.

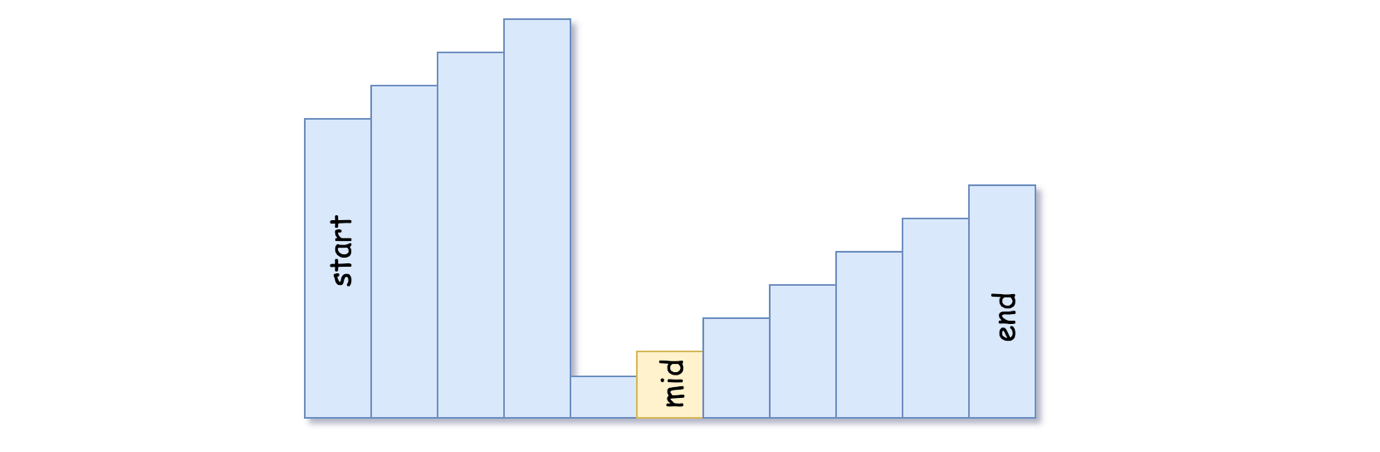


- If the target is located in the non-rotated subarray:

go left: `end = mid - 1`.

- Otherwise: go right: `start = mid + 1`.

* + - Pivot element is smaller than the first element of the array, i.e. the rotation index is somewhere between 0 and mid. It implies that the sub-array from the pivot element to the last one is non-rotated, as shown in the following graph.



- If the target is located in the non-rotated subarray:

go right: `start = mid + 1`.

- Otherwise: go left: `end = mid - 1`.

* We're here because the target is not found. Return -1.

**Implementation**

|  |
| --- |
| class Solution {  public int search(int[] nums, int target) {  int start = 0, end = nums.length - 1;  while (start <= end) {  int mid = start + (end - start) / 2;  if (nums[mid] == target) return mid;  else if (nums[mid] >= nums[start]) {  if (target >= nums[start] && target < nums[mid]) end = mid - 1;  else start = mid + 1;  }  else {  if (target <= nums[end] && target > nums[mid]) start = mid + 1;  else end = mid - 1;  }  }  return -1;  }  } |

**Complexity Analysis**

* Time complexity:  O(log*N*).
* Space complexity:  O(1).

## Template II

This chapter shows a snippet of code for Template #2. It gives a brief explanation of when to use the template and highlights the key syntax differences between the 3 templates.

**Binary Search Template II**

|  |
| --- |
| int binarySearch(int[] nums, int target){  if(nums == null || nums.length == 0)  return -1;  int left = 0, right = nums.length;  while(left < right){  // Prevent (left + right) overflow  int mid = left + (right - left) / 2;  if(nums[mid] == target){ return mid; }  else if(nums[mid] < target) { left = mid + 1; }  else { right = mid; }  }  // Post-processing:  // End Condition: left == right  if(left != nums.length && nums[left] == target) return left;  return -1;  } |

Template #2 is an advanced form of Binary Search. It is used to search for an element or condition which requires *accessing the current index and its immediate right neighbor's index* in the array.

**Key Attributes:**

* An advanced way to implement Binary Search.
* Search Condition needs to access element's immediate right neighbor
* Use element's right neighbor to determine if condition is met and decide whether to go left or right
* Gurantees Search Space is at least 2 in size at each step
* Post-processing required. Loop/Recursion ends when you have 1 element left. Need to assess if the remaining element meets the condition.

**Distinguishing Syntax:**

* Initial Condition: left = 0, right = length
* Termination: left == right
* Searching Left: right = mid
* Searching Right: left = mid+1

**First Bad Version**

You are a product manager and currently leading a team to develop a new product. Unfortunately, the latest version of your product fails the quality check. Since each version is developed based on the previous version, all the versions after a bad version are also bad.

Suppose you have n versions [1, 2, ..., n] and you want to find out the first bad one, which causes all the following ones to be bad.

You are given an API bool isBadVersion(version) which returns whether version is bad. Implement a function to find the first bad version. You should minimize the number of calls to the API.

**Example 1:**

**Input:** n = 5, bad = 4

**Output:** 4

**Explanation:**

call isBadVersion(3) -> false

call isBadVersion(5) -> true

call isBadVersion(4) -> true

Then 4 is the first bad version.

**Example 2:**

**Input:** n = 1, bad = 1

**Output:** 1

**Constraints:**

* 1 <= bad <= n <= 231 - 1

## Summary

This is a very simple problem. There is a subtle trap that you may fall into if you are not careful. Other than that, it is a direct application of a very famous algorithm.

## Solution

#### **Approach #1 (Linear Scan) [Time Limit Exceeded]**

The straight forward way is to brute force it by doing a linear scan.

|  |
| --- |
| public int firstBadVersion(int n) {  for (int i = 1; i < n; i++) {  if (isBadVersion(i)) {  return i;  }  }  return n;  } |

**Complexity analysis**

* Time complexity : *O*(*n*). Assume that *isBadVersion*(*version*) takes constant time to check if a version is bad. It takes at most *n*−1 checks, therefore the overall time complexity is *O*(*n*).
* Space complexity : *O*(1).

#### **Approach #2 (Binary Search) [Accepted]**

It is not difficult to see that this could be solved using a classic algorithm - Binary search. Let us see how the search space could be halved each time below.

Scenario #1: isBadVersion(mid) => false

1 2 3 4 5 6 7 8 9

G G G G G G B B B G = Good, B = Bad

| | |

left mid right

Let us look at the first scenario above where *isBadVersion*(*mid*)⇒*false*. We know that all versions preceding and including *mid* are all good. So we set *left*=*mid*+1 to indicate that the new search space is the interval [*mid*+1,*right*] (inclusive).

Scenario #2: isBadVersion(mid) => true

1 2 3 4 5 6 7 8 9

G G G B B B B B B G = Good, B = Bad

| | |

left mid right

The only scenario left is where *isBadVersion*(*mid*)⇒*true*. This tells us that *mid* may or may not be the first bad version, but we can tell for sure that all versions after *mid* can be discarded. Therefore we set *right*=*mid* as the new search space of interval [*left*,*mid*] (inclusive).

In our case, we indicate *left* and *right* as the boundary of our search space (both inclusive). This is why we initialize *left*=1 and *right*=*n*. How about the terminating condition? We could guess that *left* and *right* eventually both meet and it must be the first bad version, but how could you tell for sure?

The formal way is to [prove by induction](http://www.cs.cornell.edu/courses/cs211/2006sp/Lectures/L06-Induction/binary_search.html), which you can read up yourself if you are interested. Here is a helpful tip to quickly prove the correctness of your binary search algorithm during an interview. We just need to test an input of size 2. Check if it reduces the search space to a single element (which must be the answer) for both of the scenarios above. If not, your algorithm will never terminate.

If you are setting  *mid*=(*left*+*right)/2*​, you have to be very careful. Unless you are using a language that does not overflow such as [Python](https://www.reddit.com/r/Python/comments/36xu5z/can_integer_operations_overflow_in_python/), *left*+*right* could overflow. One way to fix this is to use left + (right – left)/2 ​ instead.

If you fall into this subtle overflow bug, you are not alone. Even Jon Bentley's own implementation of binary search had this [overflow bug](https://en.wikipedia.org/wiki/Binary_search_algorithm#Implementation_issues) and remained undetected for over twenty years.

|  |
| --- |
| public int firstBadVersion(int n) {  int left = 1;  int right = n;  while (left < right) {  int mid = left + (right - left) / 2;  if (isBadVersion(mid)) {  right = mid;  } else {  left = mid + 1;  }  }  return left;  } |

**Complexity analysis**

* Time complexity : *O*(log*n*). The search space is halved each time, so the time complexity is *O*(log*n*).
* Space complexity : *O*(1).

**Find Peak Element**

A peak element is an element that is strictly greater than its neighbors.

Given an integer array nums, find a peak element, and return its index. If the array contains multiple peaks, return the index to **any of the peaks**.

You may imagine that nums[-1] = nums[n] = -∞.

**Example 1:**

**Input:** nums = [1,2,3,1]

**Output:** 2

**Explanation:** 3 is a peak element and your function should return the index number 2.

**Example 2:**

**Input:** nums = [1,2,1,3,5,6,4]

**Output:** 5

**Explanation:** Your function can return either index number 1 where the peak element is 2, or index number 5 where the peak element is 6.

**Constraints:**

* 1 <= nums.length <= 1000
* -231 <= nums[i] <= 231 - 1
* nums[i] != nums[i + 1] for all valid i.

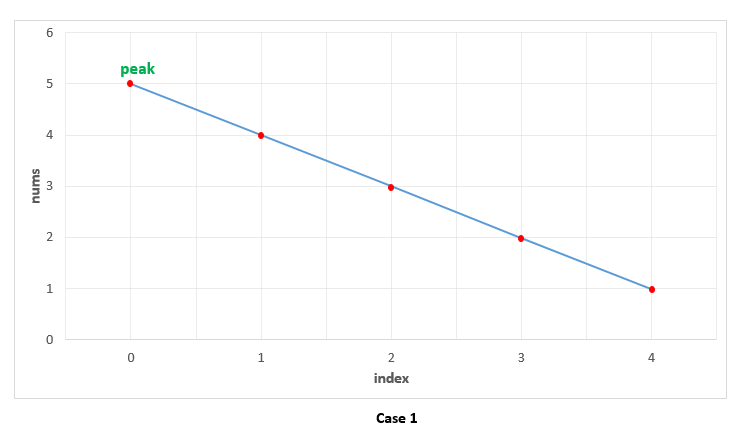
**Follow up:** Could you implement a solution with logarithmic complexity?

## Solution

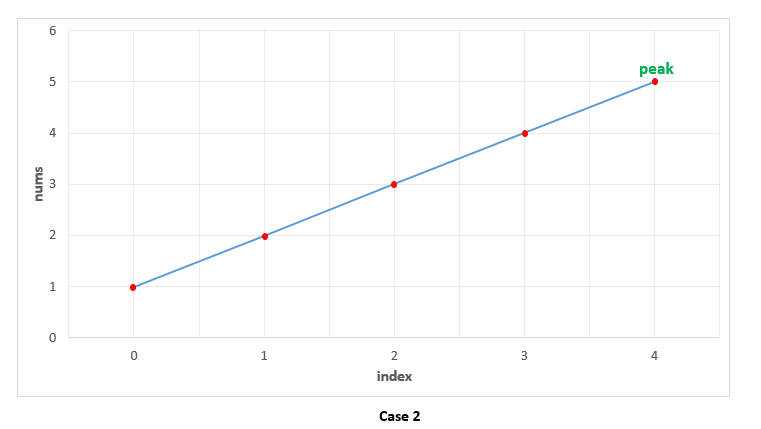
#### **Approach 1: Linear Scan**

In this approach, we make use of the fact that two consecutive numbers *nums*[*j*] and *nums*[*j*+1] are never equal. Thus, we can traverse over the *nums* array starting from the beginning. Whenever, we find a number *nums*[*i*], we only need to check if it is larger than the next number *nums*[*i*+1] for determining if *nums*[*i*] is the peak element. The reasoning behind this can be understood by taking the following three cases which cover every case into which any problem can be divided.

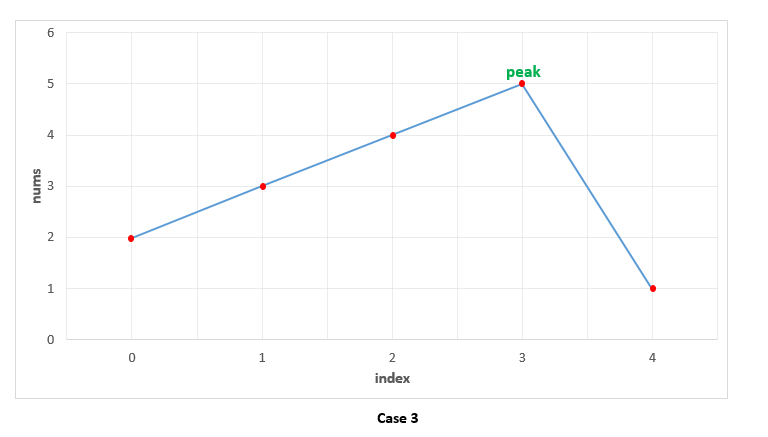
Case 1. All the numbers appear in a descending order. In this case, the first element corresponds to the peak element. We start off by checking if the current element is larger than the next one. The first element satisfies this criteria, and is hence identified as the peak correctly. In this case, we didn't reach a point where we needed to compare *nums*[*i*] with *nums*[*i*−1] also, to determine if it is the peak element or not.



Case 2. All the elements appear in ascending order. In this case, we keep on comparing *nums*[*i*] with *nums*[*i*+1] to determine if *nums*[*i*] is the peak element or not. None of the elements satisfy this criteria, indicating that we are currently on a rising slope and not on a peak. Thus, at the end, we need to return the last element as the peak element, which turns out to be correct. In this case also, we need not compare *nums*[*i*] with *nums*[*i*−1], since being on the rising slope is a sufficient condition to ensure that *nums*[*i*] isn't the peak element.



Case 3. The peak appears somewhere in the middle. In this case, when we are traversing on the rising edge, as in Case 2, none of the elements will satisfy *nums*[*i*]>*nums*[*i*+1]. We need not compare *nums*[*i*] with *nums*[*i*−1] on the rising slope as discussed above. When we finally reach the peak element, the condition *nums*[*i*]>*nums*[*i*+1] is satisfied. We again, need not compare *nums*[*i*] with *nums*[*i*−1]. This is because, we could reach *nums*[*i*] as the current element only when the check *nums*[*i*]>*nums*[*i*+1] failed for the previous((*i*−1)*th* element, indicating that *nums*[*i*−1]<*nums*[*i*]. Thus, we are able to identify the peak element correctly in this case as well.



|  |
| --- |
| public class Solution {  public int findPeakElement(int[] nums) {  for (int i = 0; i < nums.length - 1; i++) {  if (nums[i] > nums[i + 1])  return i;  }  return nums.length - 1;  }  } |

**Complexity Analysis**

* Time complexity : *O*(*n*). We traverse the *nums* array of size *n* once only.
* Space complexity : *O*(1). Constant extra space is used.

#### **Approach 2: Recursive Binary Search**

**Algorithm**

We can view any given sequence in *nums* array as alternating ascending and descending sequences. By making use of this, and the fact that we can return any peak as the result, we can make use of Binary Search to find the required peak element.

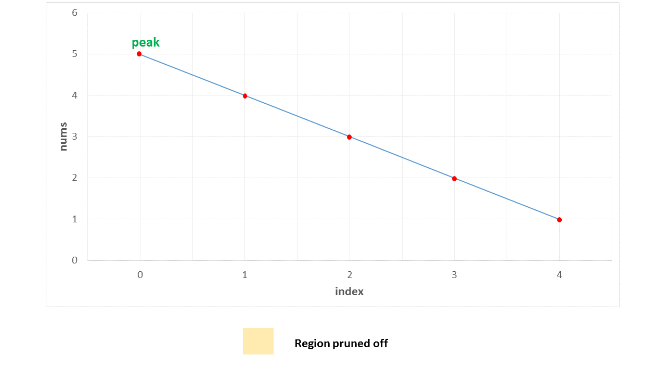
In case of simple Binary Search, we work on a sorted sequence of numbers and try to find out the required number by reducing the search space at every step. In this case, we use a modification of this simple Binary Search to our advantage. We start off by finding the middle element, *mid* from the given *nums* array. If this element happens to be lying in a descending sequence of numbers. or a local falling slope(found by comparing *nums*[*i*] to its right neighbour), it means that the peak will always lie towards the left of this element. Thus, we reduce the search space to the left of *mid*(including itself) and perform the same process on left subarray.

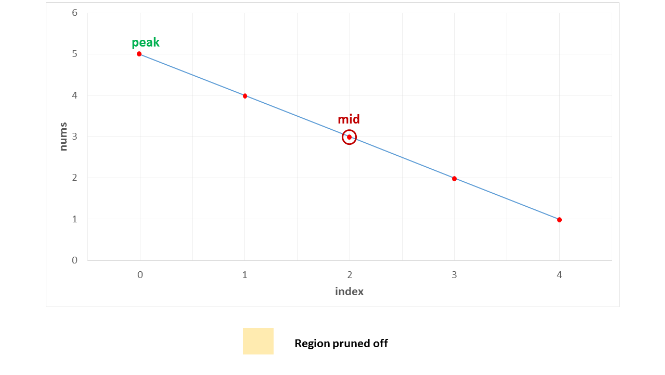
If the middle element, *mid* lies in an ascending sequence of numbers, or a rising slope(found by comparing *nums*[*i*] to its right neighbour), it obviously implies that the peak lies towards the right of this element. Thus, we reduce the search space to the right of *mid* and perform the same process on the right subarray.

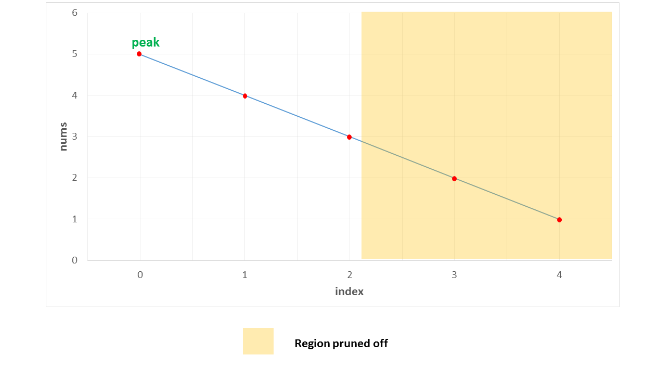
In this way, we keep on reducing the search space till we eventually reach a state where only one element is remaining in the search space. This single element is the peak element.

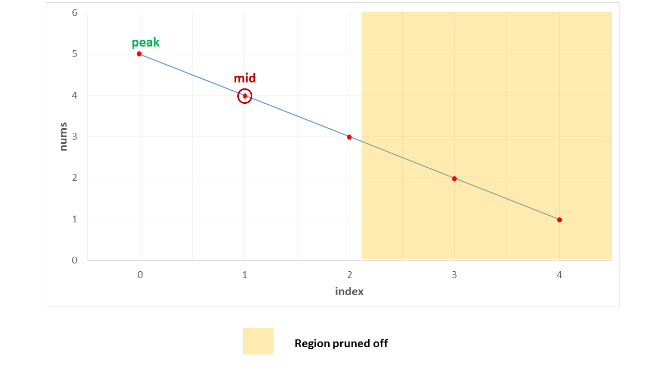
To see how it works, let's consider the three cases discussed above again.

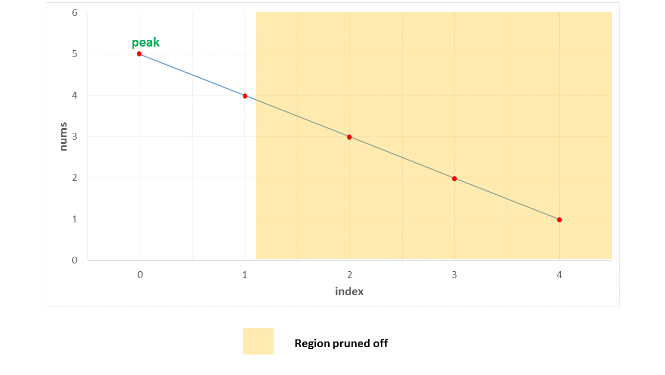
Case 1. In this case, we firstly find 33 as the middle element. Since it lies on a falling slope, we reduce the search space to [1, 2, 3]. For this subarray, 22 happens to be the middle element, which again lies on a falling slope, reducing the search space to [1, 2]. Now, 11 acts as the middle element and it lies on a falling slope, reducing the search space to [1] only. Thus, 11 is returned as the peak correctly.

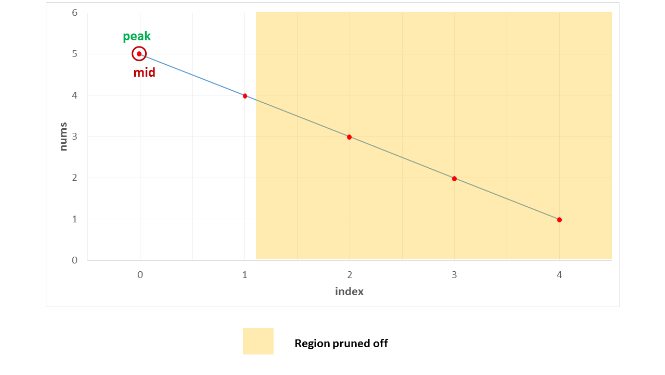


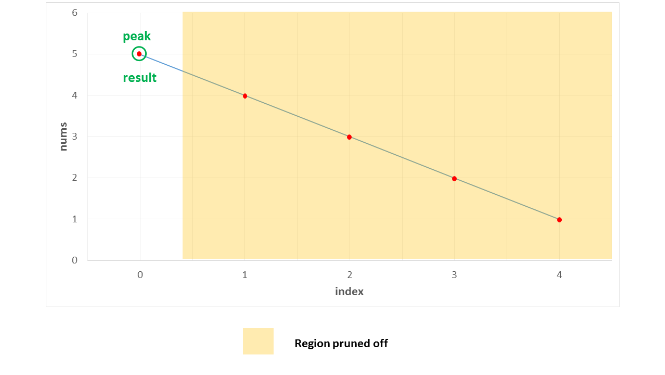




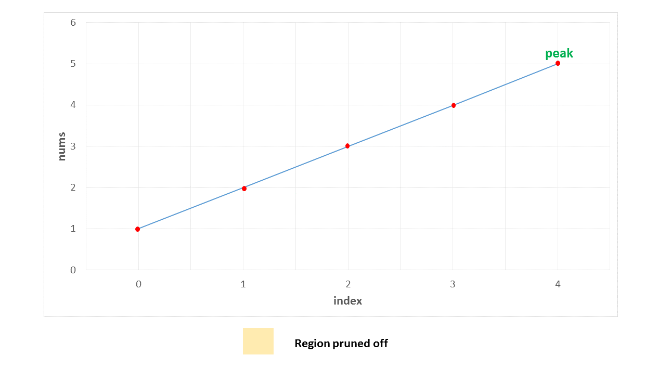


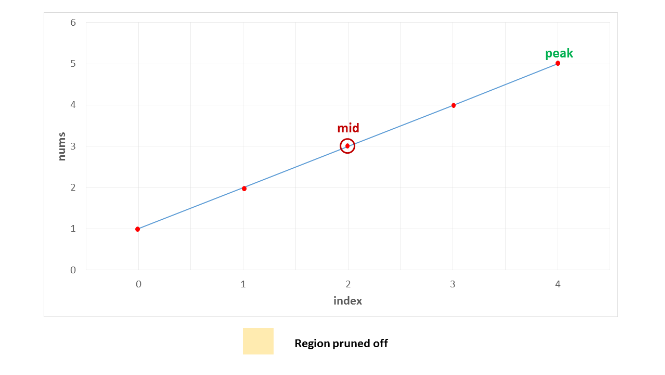


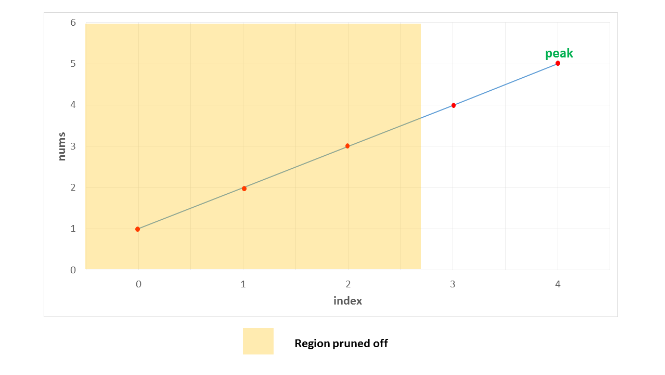


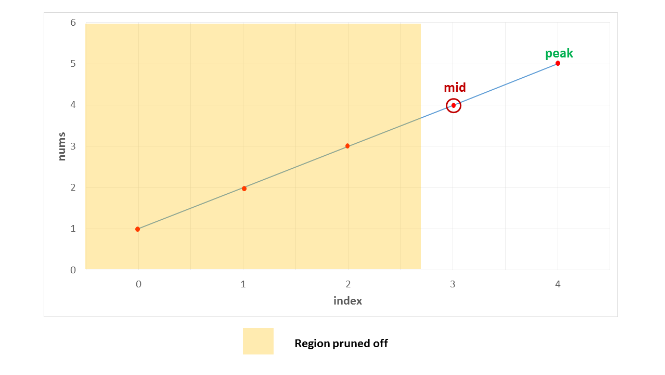


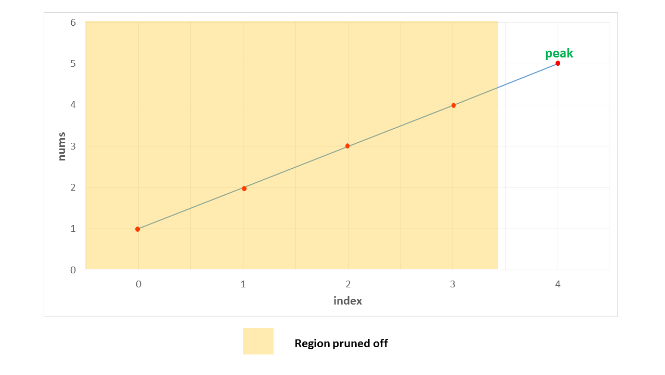
Case 2. In this case, we firstly find 3 as the middle element. Since it lies on a rising slope, we reduce the search space to [4, 5]. Now, 4 acts as the middle element for this subarray and it lies on a rising slope, reducing the search space to [5] only. Thus, 5 is returned as the peak correctly.

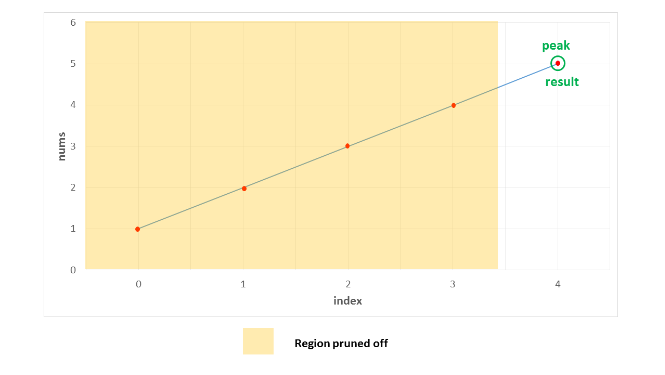




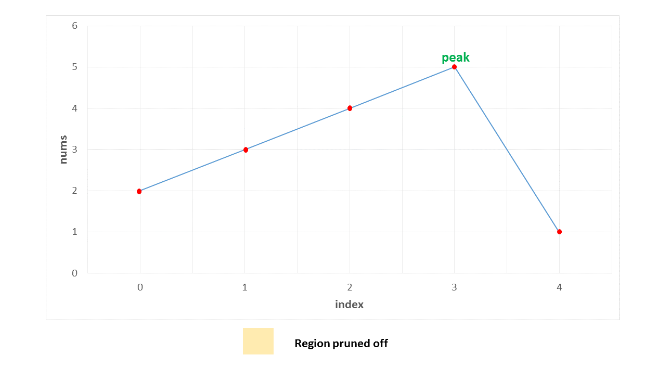




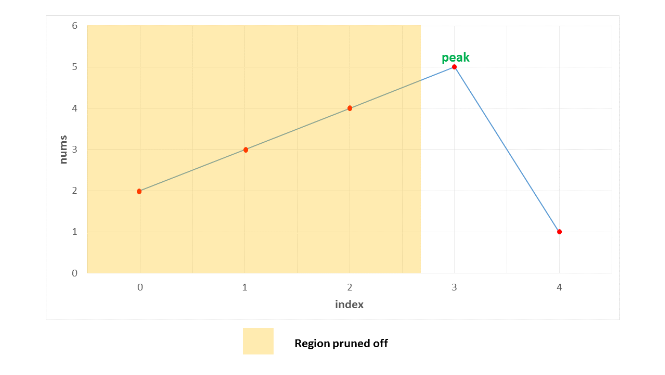


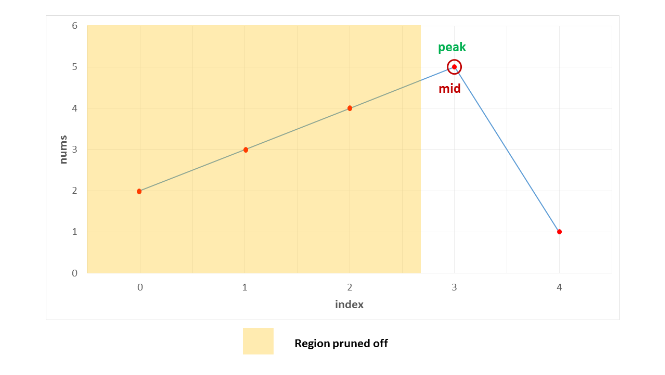


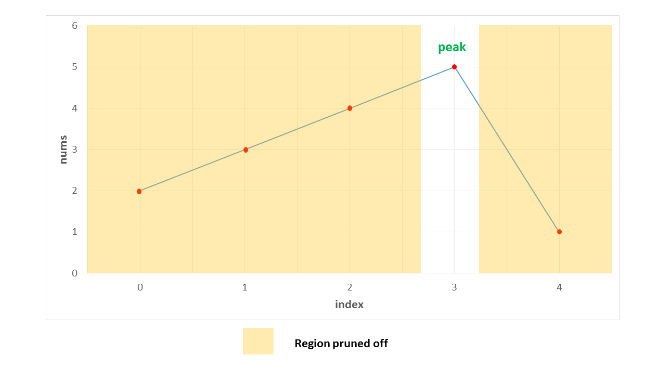
Case 3. In this case, the peak lies somewhere in the middle. The first middle element is 4. It lies on a rising slope, indicating that the peak lies towards its right. Thus, the search space is reduced to [5, 1]. Now, 5 happens to be the on a falling slope(relative to its right neighbour), reducing the search space to [5] only. Thus, 5 is identified as the peak element correctly.

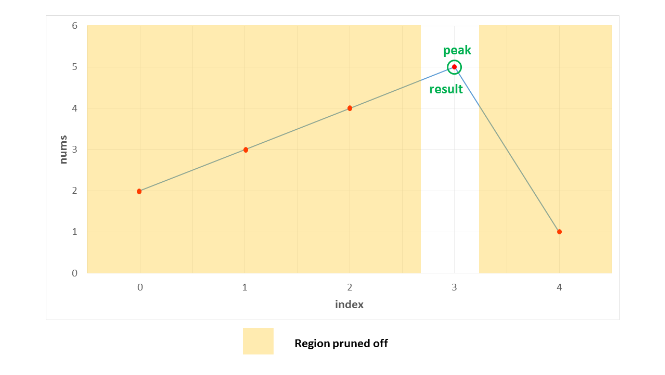












|  |
| --- |
| public class Solution {  public int findPeakElement(int[] nums) {  return search(nums, 0, nums.length - 1);  }  public int search(int[] nums, int l, int r) {  if (l == r)  return l;  int mid = (l + r) / 2;  if (nums[mid] > nums[mid + 1])  return search(nums, l, mid);  return search(nums, mid + 1, r);  }  } |

**Complexity Analysis**

* Time complexity : *O*(*log*2​(*n*)). We reduce the search space in half at every step. Thus, the total search space will be consumed in *log*2​(*n*) steps. Here, *n* refers to the size of *nums* array.
* Space complexity : *O*(*log*2​(*n*)). We reduce the search space in half at every step. Thus, the total search space will be consumed in *log*2​(*n*) steps. Thus, the depth of recursion tree will go upto *log*2​(*n*).

#### **Approach 3: Iterative Binary Search**

**Algorithm**

The binary search discussed in the previous approach used a recursive method. We can do the same process in an iterative fashion also. This is done in the current approach.

|  |
| --- |
| public class Solution {  public int findPeakElement(int[] nums) {  int l = 0, r = nums.length - 1;  while (l < r) {  int mid = (l + r) / 2;  if (nums[mid] > nums[mid + 1])  r = mid;  else  l = mid + 1;  }  return l;  }  } |

**Complexity Analysis**

* Time complexity :  *O*(*log*2​(*n*)). We reduce the search space in half at every step. Thus, the total search space will be consumed in *log*2​(*n*) steps. Here, *n* refers to the size of *nums* array.
* Space complexity : *O*(1). Constant extra space is used.

**Find Minimum in Rotated Sorted Array**

Suppose an array of length n sorted in ascending order is **rotated** between 1 and n times. For example, the array nums = [0,1,2,4,5,6,7] might become:

* [4,5,6,7,0,1,2] if it was rotated 4 times.
* [0,1,2,4,5,6,7] if it was rotated 7 times.

Notice that **rotating** an array [a[0], a[1], a[2], ..., a[n-1]] 1 time results in the array [a[n-1], a[0], a[1], a[2], ..., a[n-2]].

Given the sorted rotated array nums, return the minimum element of this array.

**Example 1:**

**Input:** nums = [3,4,5,1,2]

**Output:** 1

**Explanation:** The original array was [1,2,3,4,5] rotated 3 times.

**Example 2:**

**Input:** nums = [4,5,6,7,0,1,2]

**Output:** 0

**Explanation:** The original array was [0,1,2,4,5,6,7] and it was rotated 4 times.

**Example 3:**

**Input:** nums = [11,13,15,17]

**Output:** 11

**Explanation:** The original array was [11,13,15,17] and it was rotated 4 times.

**Constraints:**

* n == nums.length
* 1 <= n <= 5000
* -5000 <= nums[i] <= 5000
* All the integers of nums are **unique**.
* nums is sorted and rotated between 1 and n times.

   Hide Hint #1

Array was originally in ascending order. Now that the array is rotated, there would be a point in the array where there is a small deflection from the increasing sequence. eg. The array would be something like [4, 5, 6, 7, 0, 1, 2].

   Hide Hint #2

You can divide the search space into two and see which direction to go. Can you think of an algorithm which has O(logN) search complexity?

   Hide Hint #3

1. All the elements to the left of inflection point > first element of the array.
2. All the elements to the right of inflection point < first element of the array.

## Solution

#### **Approach 1: Binary Search**

**Intuition**

A very brute way of solving this question is to search the entire array and find the minimum element. The time complexity for that would be *O*(*N*) given that N is the size of the array.

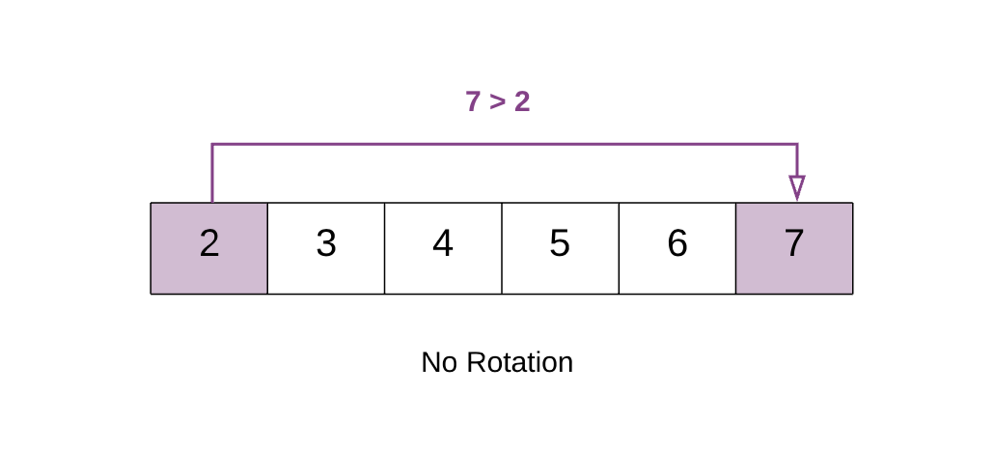
A very cool way of solving this problem is using the Binary Search algorithm. In binary search we find out the mid point and decide to either search on the left or right depending on some condition.

Since the given array is sorted, we can make use of binary search. However, the array is rotated. So simply applying the binary search won't work here.

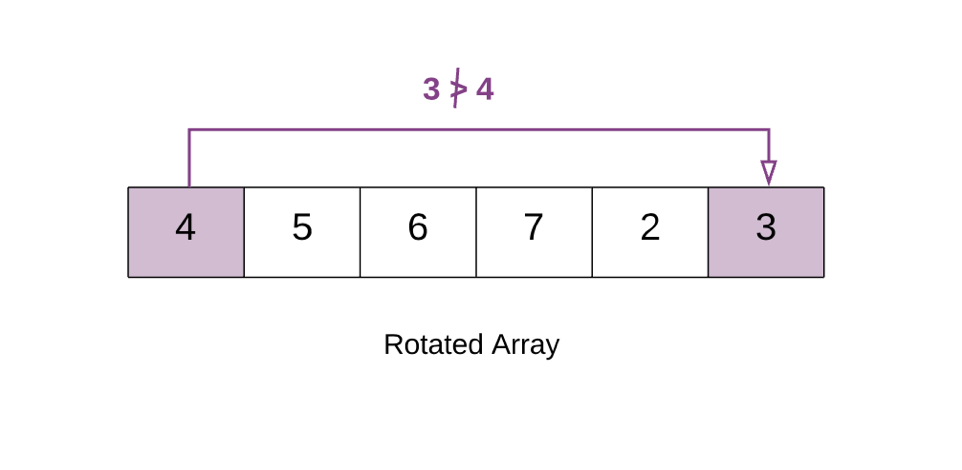
In this question we would essentially apply a modified version of binary search where the condition that decides the search direction would be different than in a standard binary search.

We want to find the smallest element in a rotated sorted array. What if the array is not rotated? How do we check that?

If the array is not rotated and the array is in ascending order, then last element > first element.

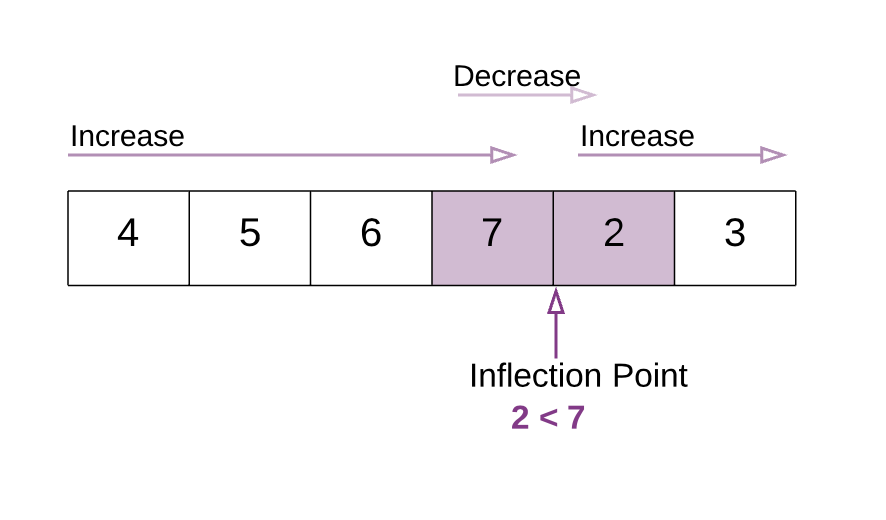


In the above example 7 > 2. This means that the array is still sorted and has no rotation.



In the above example 3 < 4. Hence the array is rotated. This happens because the array was initially [2, 3 ,4 ,5 ,6 ,7]. But after the rotation the smaller elements[2,3] go at the back. i.e. [4, 5, 6, 7, 2, 3]. Because of this the first element [4] in the rotated array becomes greater than the last element.

This means there is a point in the array at which you would notice a change. This is the point which would help us in this question. We call this the Inflection Point.

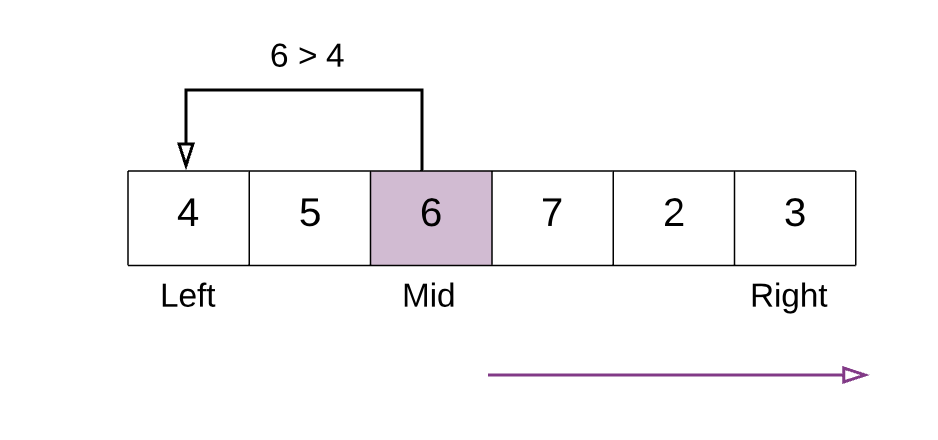


In this modified version of binary search algorithm, we are looking for this point. In the above example notice the Inflection Point .

All the elements to the left of inflection point > first element of the array.  
All the elements to the right of inflection point < first element of the array.

**Algorithm**

1. Find the mid element of the array.
2. If mid element > first element of array this means that we need to look for the inflection point on the right of mid.
3. If mid element < first element of array this that we need to look for the inflection point on the left of mid.

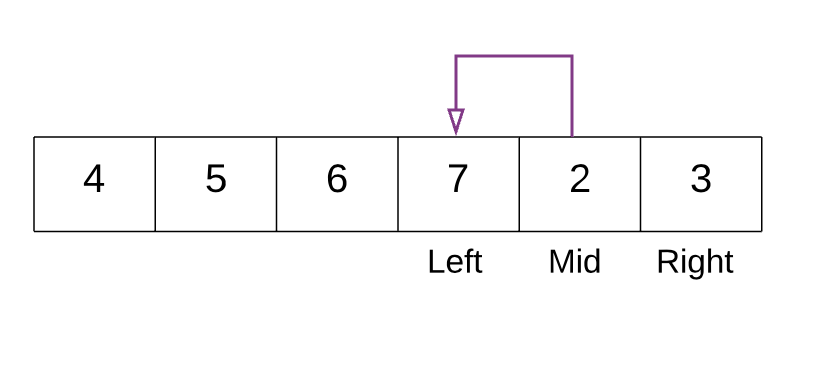


In the above example mid element 6 is greater than first element 4. Hence we continue our search for the inflection point to the right of mid.

4 . We stop our search when we find the inflection point, when either of the two conditions is satisfied:

nums[mid] > nums[mid + 1] Hence, **mid+1** is the smallest.

nums[mid - 1] > nums[mid] Hence, **mid** is the smallest.



In the above example. With the marked left and right pointers. The mid element is 2. The element just before 2 is 7 and 7>2 i.e. nums[mid - 1] > nums[mid]. Thus we have found the point of inflection and 2 is the smallest element.

|  |
| --- |
| class Solution {  public int findMin(int[] nums) {  // If the list has just one element then return that element.  if (nums.length == 1) {  return nums[0];  }  // initializing left and right pointers.  int left = 0, right = nums.length - 1;  // if the last element is greater than the first element then there is no rotation.  // e.g. 1 < 2 < 3 < 4 < 5 < 7. Already sorted array.  // Hence the smallest element is first element. A[0]  if (nums[right] > nums[0]) {  return nums[0];  }    // Binary search way  while (right >= left) {  // Find the mid element  int mid = left + (right - left) / 2;  // if the mid element is greater than its next element then mid+1 element is the smallest  // This point would be the point of change. From higher to lower value.  if (nums[mid] > nums[mid + 1]) {  return nums[mid + 1];  }  // if the mid element is lesser than its previous element then mid element is the smallest  if (nums[mid - 1] > nums[mid]) {  return nums[mid];  }  // if the mid elements value is greater than the 0th element this means  // the least value is still somewhere to the right as we are still dealing with elements  // greater than nums[0]  if (nums[mid] > nums[0]) {  left = mid + 1;  } else {  // if nums[0] is greater than the mid value then this means the smallest value is somewhere to  // the left  right = mid - 1;  }  }  return -1;  }  } |

**Complexity Analysis**

* Time Complexity : Same as Binary Search *O*(log*N*)
* Space Complexity : *O*(1)

## Template III

This chapter shows a snippet of code for Template #3. It gives a brief explanation of when to use the template and highlights the key syntax differences between the 3 templates.

**Binary Search Template III**

|  |
| --- |
| int binarySearch(int[] nums, int target) {  if (nums == null || nums.length == 0)  return -1;  int left = 0, right = nums.length - 1;  while (left + 1 < right){  // Prevent (left + right) overflow  int mid = left + (right - left) / 2;  if (nums[mid] == target) {  return mid;  } else if (nums[mid] < target) {  left = mid;  } else {  right = mid;  }  }  // Post-processing:  // End Condition: left + 1 == right  if(nums[left] == target) return left;  if(nums[right] == target) return right;  return -1;  } |

Template #3 is another unique form of Binary Search. It is used to search for an element or condition which requires accessing the current index and its immediate left and right neighbor's index in the array.

**Key Attributes:**

* An alternative way to implement Binary Search
* Search Condition needs to access element's immediate left and right neighbors
* Use element's neighbors to determine if condition is met and decide whether to go left or right
* Gurantees Search Space is at least 3 in size at each step
* Post-processing required. Loop/Recursion ends when you have 2 elements left. Need to assess if the remaining elements meet the condition.

**Distinguishing Syntax:**

* Initial Condition: left = 0, right = length-1
* Termination: left + 1 == right
* Searching Left: right = mid
* Searching Right: left = mid

**Search for a Range**

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

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

**Follow up:** Could you write an algorithm with O(log n) runtime complexity?

**Example 1:**

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

**Output:** [3,4]

**Example 2:**

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

**Output:** [-1,-1]

**Example 3:**

**Input:** nums = [], target = 0

**Output:** [-1,-1]

**Constraints:**

* 0 <= nums.length <= 105
* -109 <= nums[i] <= 109
* nums is a non-decreasing array.
* -109 <= target <= 109

#### **Approach 1: Linear Scan**

**Intuition**

Checking every index for target exhausts the search space, so it must work.

**Algorithm**

First, we do a linear scan of nums from the left, breaking when we find an instance of target. If we never break, then target is not present, so we can return the "error code" of [-1, -1] early. Given that we did find a valid left index, we can do a second linear scan, but this time from the right. In this case, the first instance of target encountered will be the rightmost one (and because a leftmost one exists, there is guaranteed to also be a rightmost one). We then simply return a list containing the two located indices.

|  |
| --- |
| class Solution {  public int[] searchRange(int[] nums, int target) {  int[] targetRange = {-1, -1};  // find the index of the leftmost appearance of `target`.  for (int i = 0; i < nums.length; i++) {  if (nums[i] == target) {  targetRange[0] = i;  break;  }  }  // if the last loop did not find any index, then there is no valid range  // and we return [-1, -1].  if (targetRange[0] == -1) {  return targetRange;  }  // find the index of the rightmost appearance of `target` (by reverse  // iteration). it is guaranteed to appear.  for (int j = nums.length-1; j >= 0; j--) {  if (nums[j] == target) {  targetRange[1] = j;  break;  }  }  return targetRange;  }  } |

**Complexity Analysis**

* Time complexity : *O*(*n*)

This brute-force approach examines each of the n elements of nums exactly twice, so the overall runtime is linear.

* Space complexity : *O*(1)

The linear scan method allocates a fixed-size array and a few integers, so it has a constant-size memory footprint.

#### **Approach 2: Binary Search**

**Intuition**

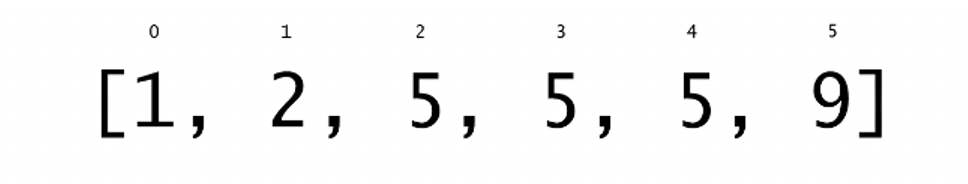
Because the array is sorted, we can use binary search to locate the left and rightmost indices.

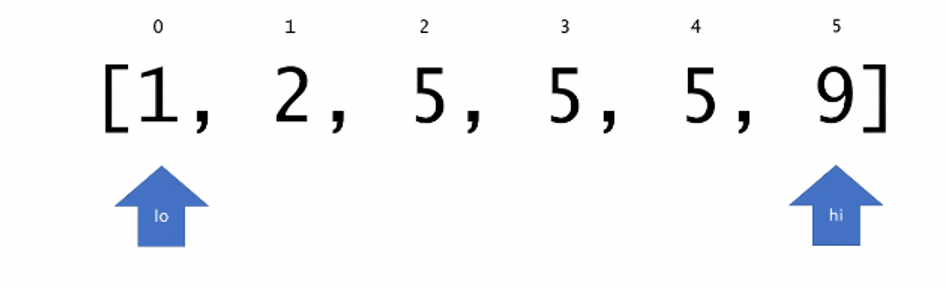
**Algorithm**

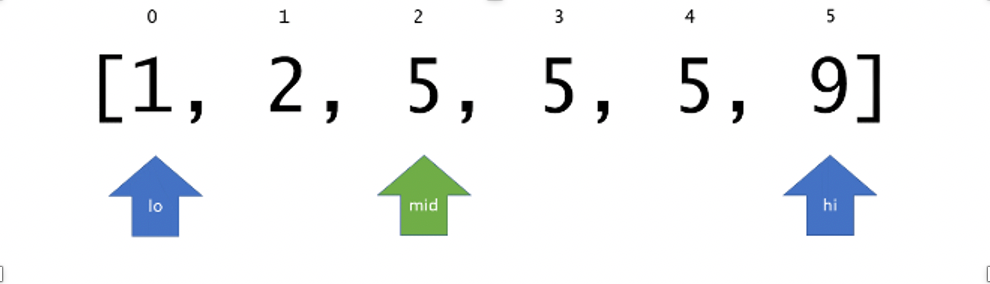
The overall algorithm works fairly similarly to the linear scan approach, except for the subroutine used to find the left and rightmost indices themselves. Here, we use a modified binary search to search a sorted array, with a few minor adjustments. First, because we are locating the leftmost (or rightmost) index containing target (rather than returning true if we find target), the algorithm does not terminate as soon as we find a match. Instead, we continue to search until lo == hi and they contain some index at which target can be found.

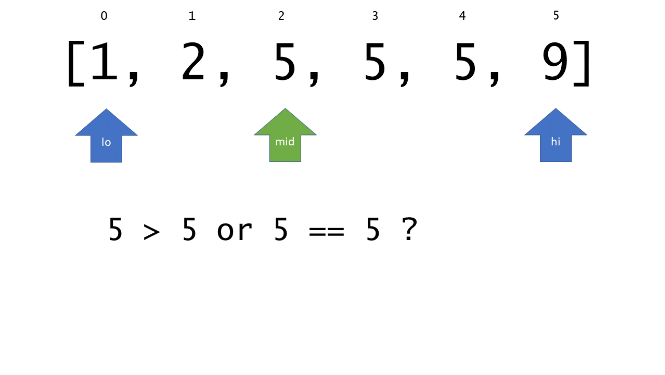
The other change is the introduction of the left parameter, which is a boolean indicating what to do in the event that target == nums[mid]; if left is true, then we "recurse" on the left subarray on ties. Otherwise, we go right. To see why this is correct, consider the situation where we find target at index i. The leftmost target cannot occur at any index greater than i, so we never need to consider the right subarray. The same argument applies to the rightmost index.

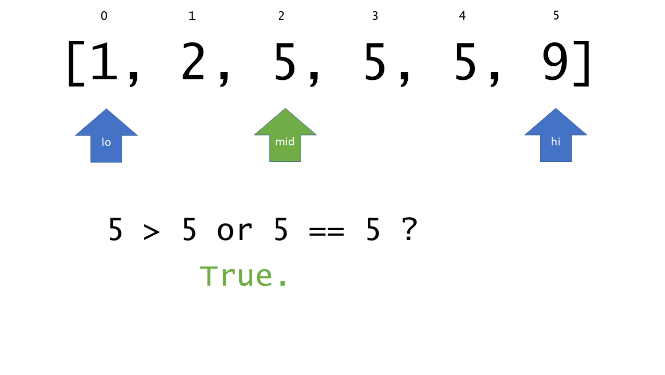
The first animation below shows the process for finding the leftmost index, and the second shows the process for finding the index right of the rightmost index.

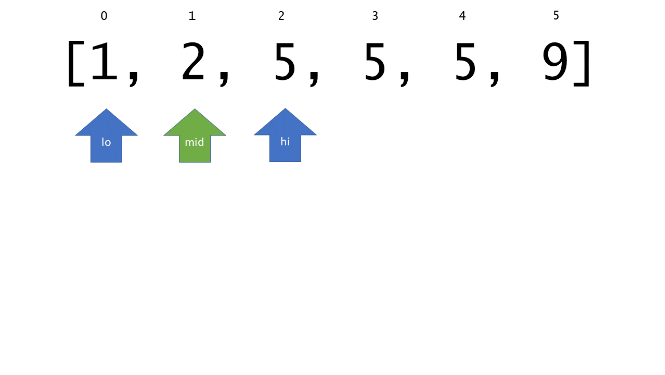


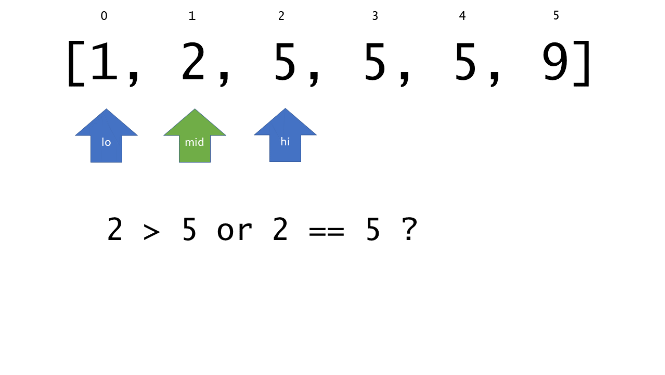


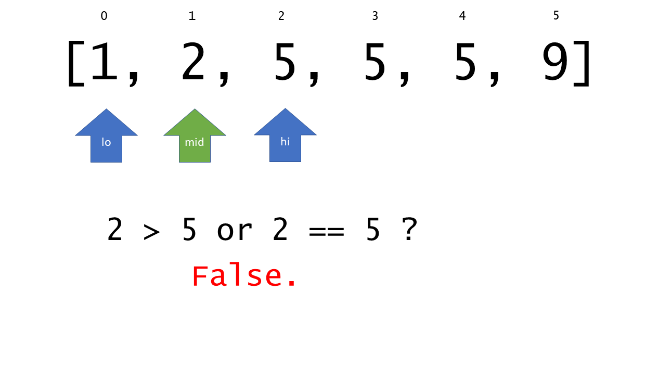


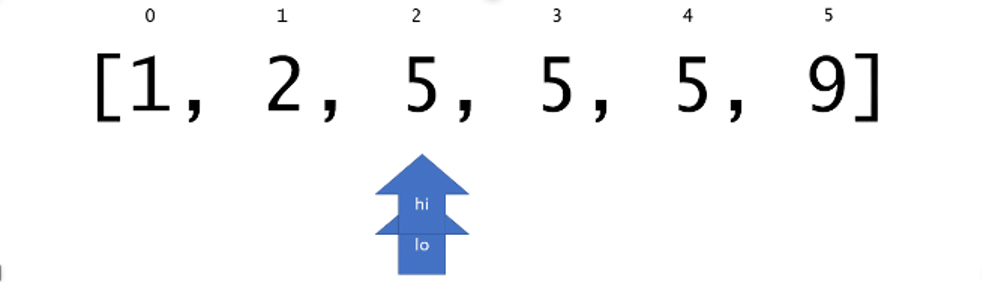


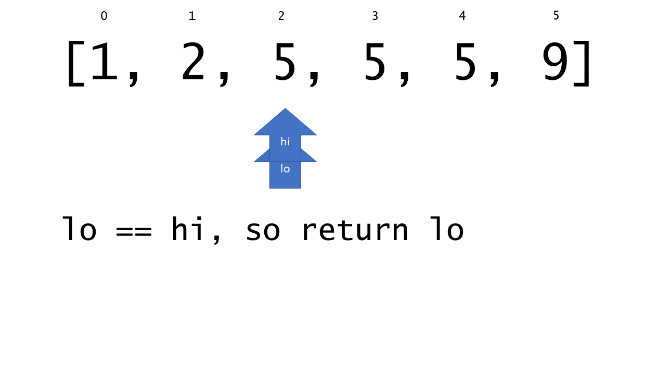




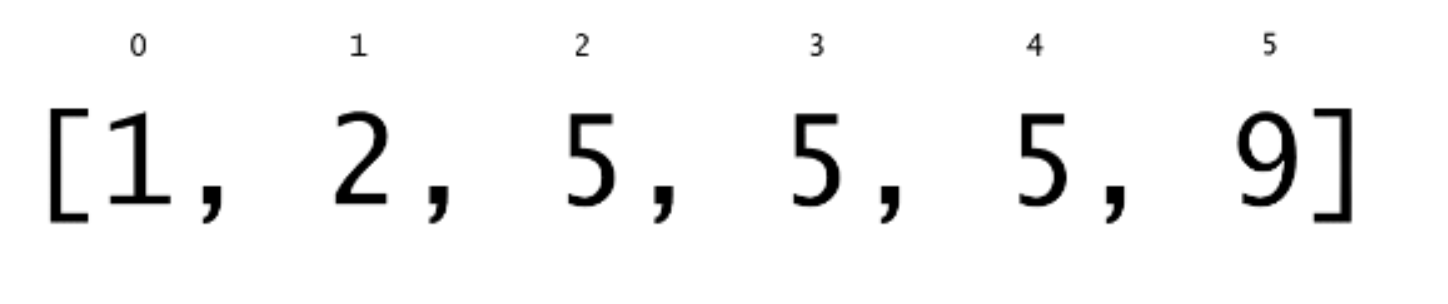


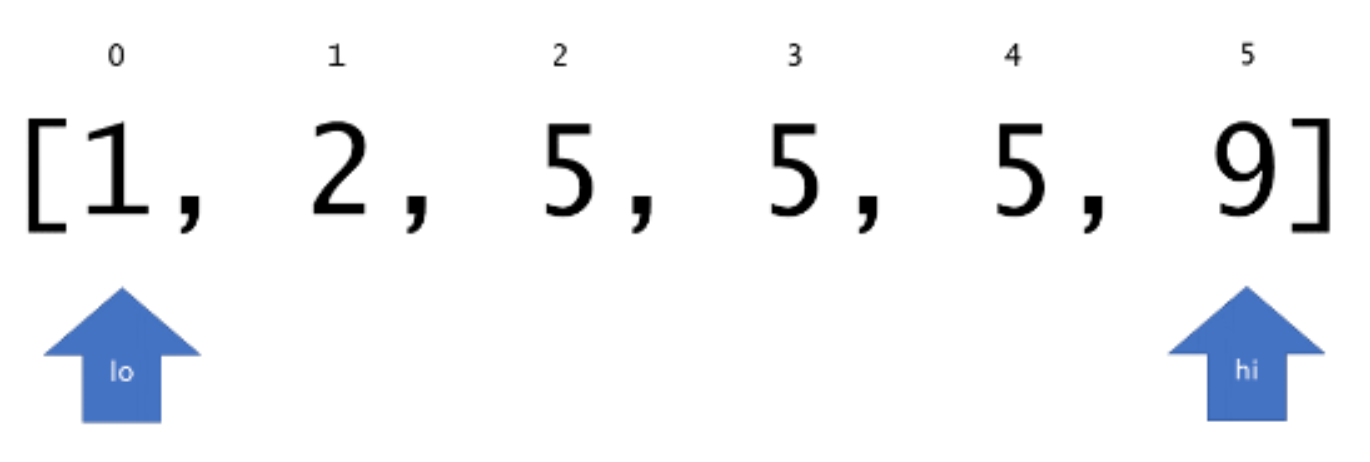




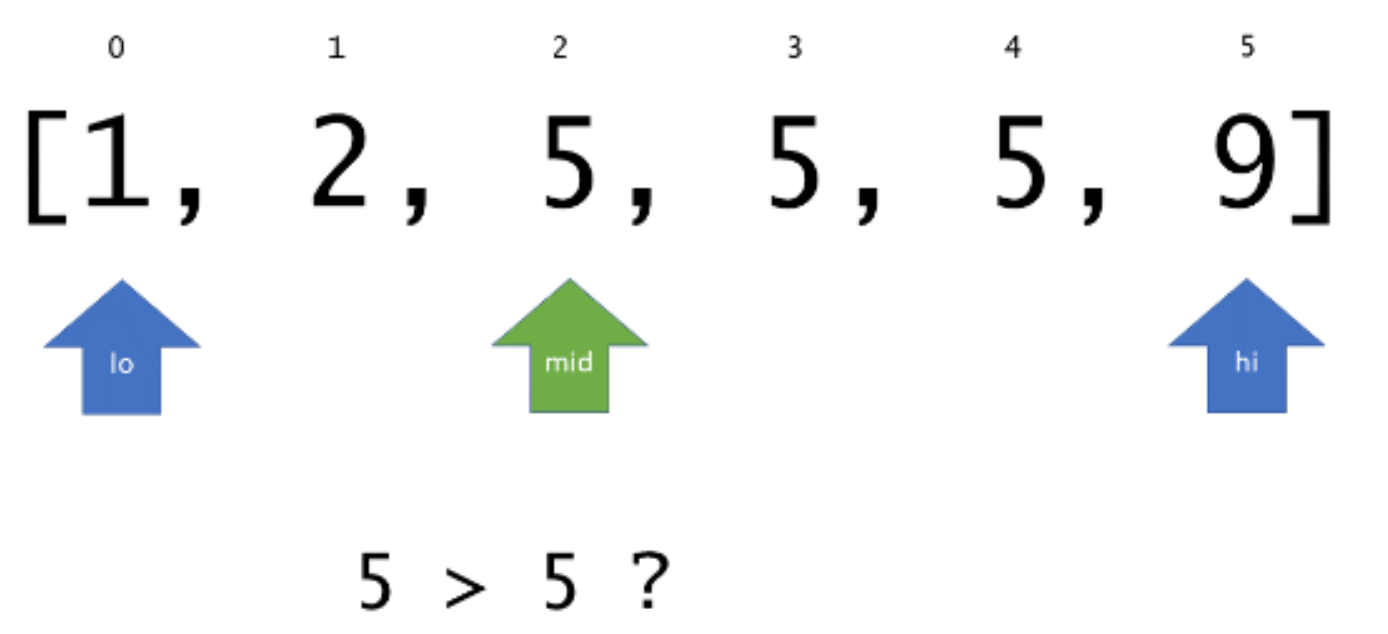


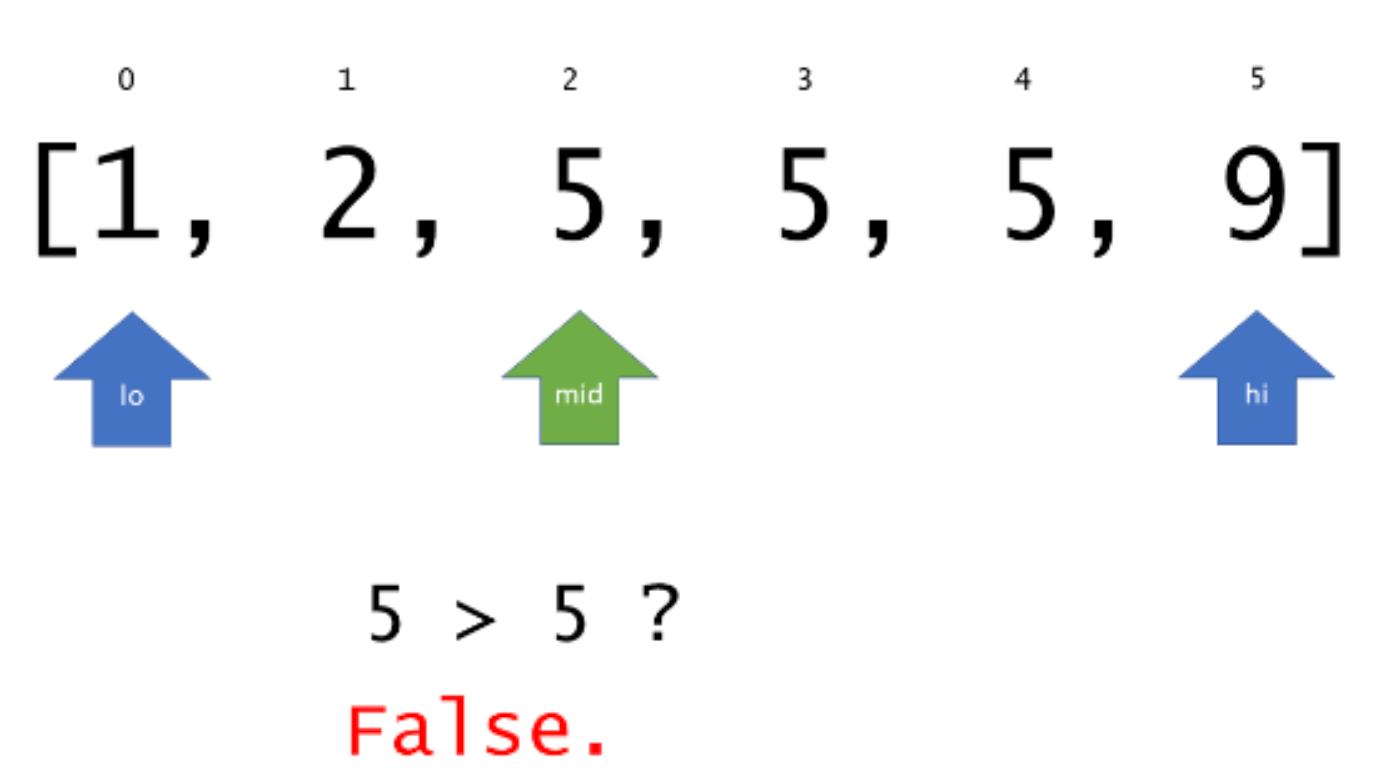
--------------------------------------------------------------------------------------------------------------

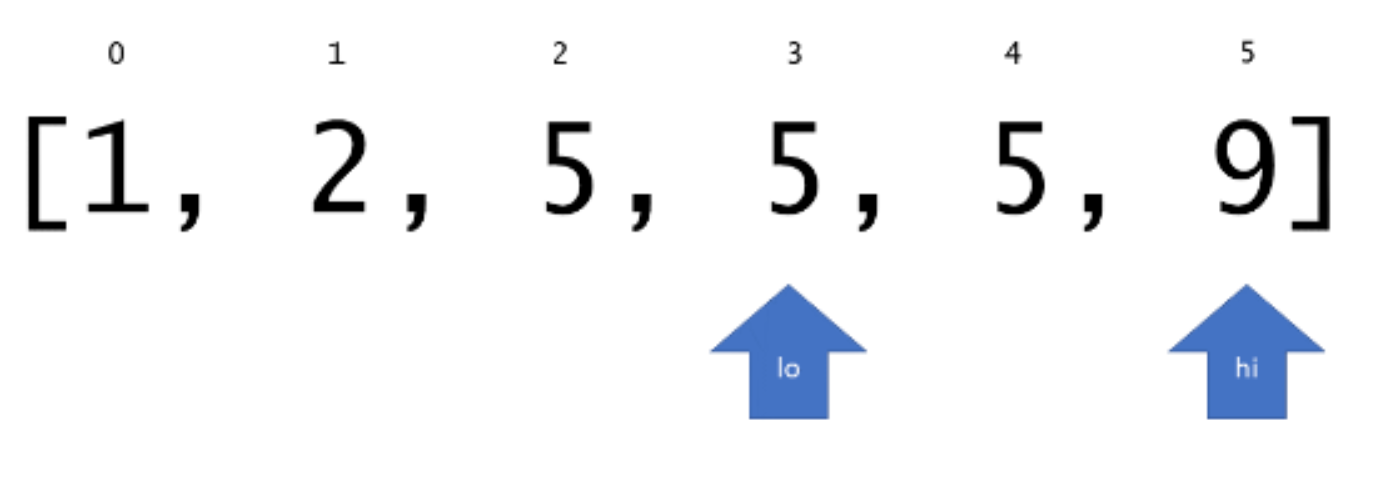


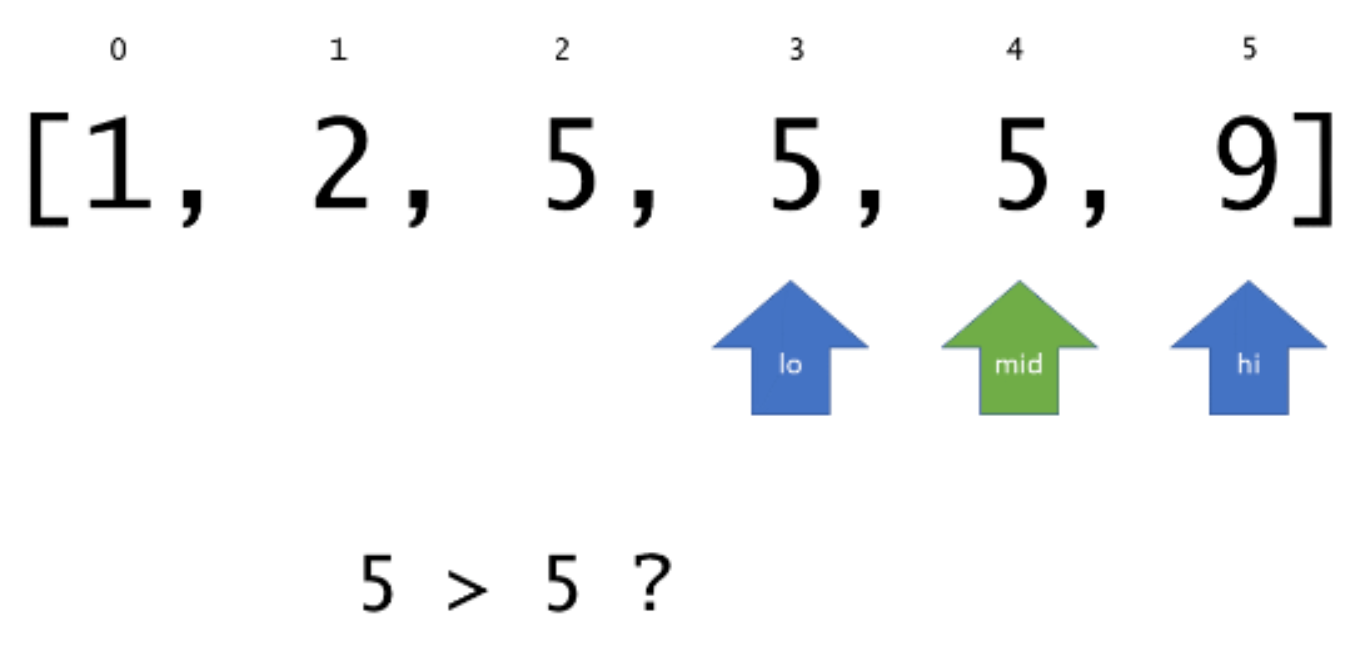
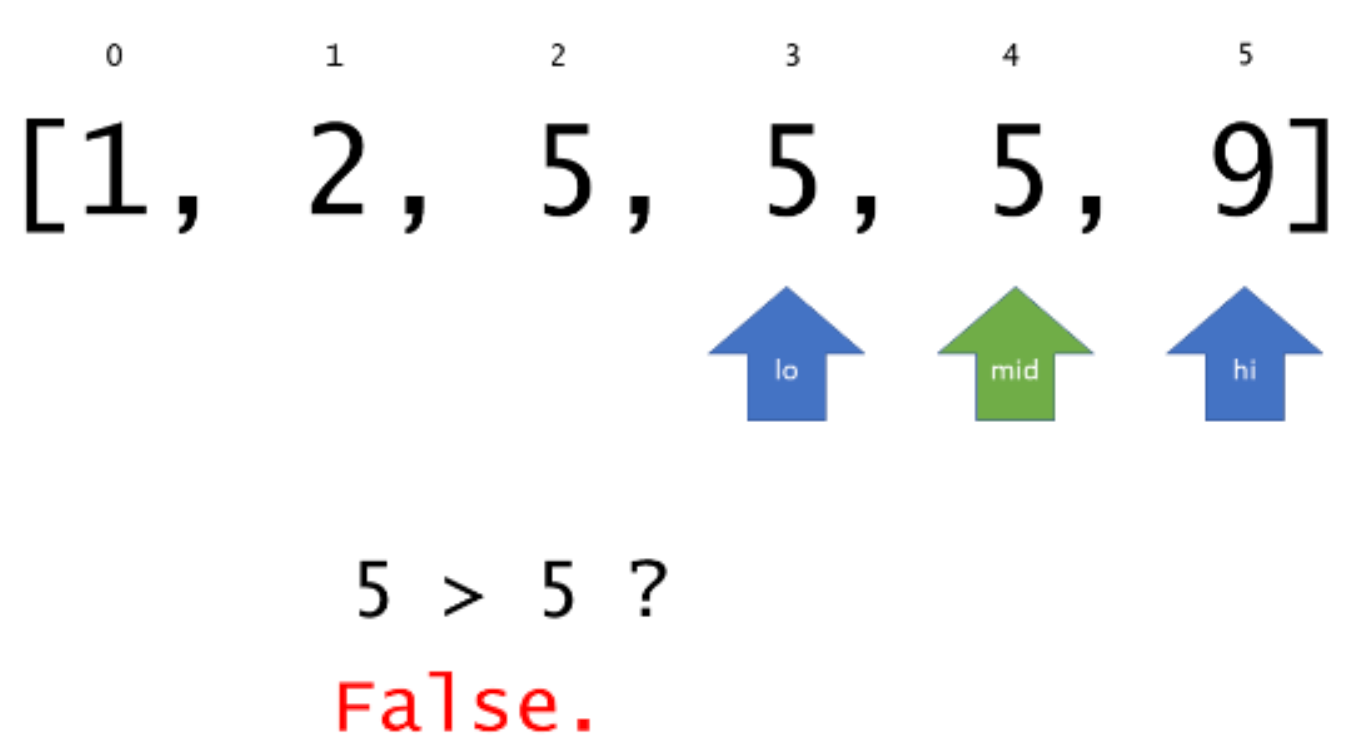


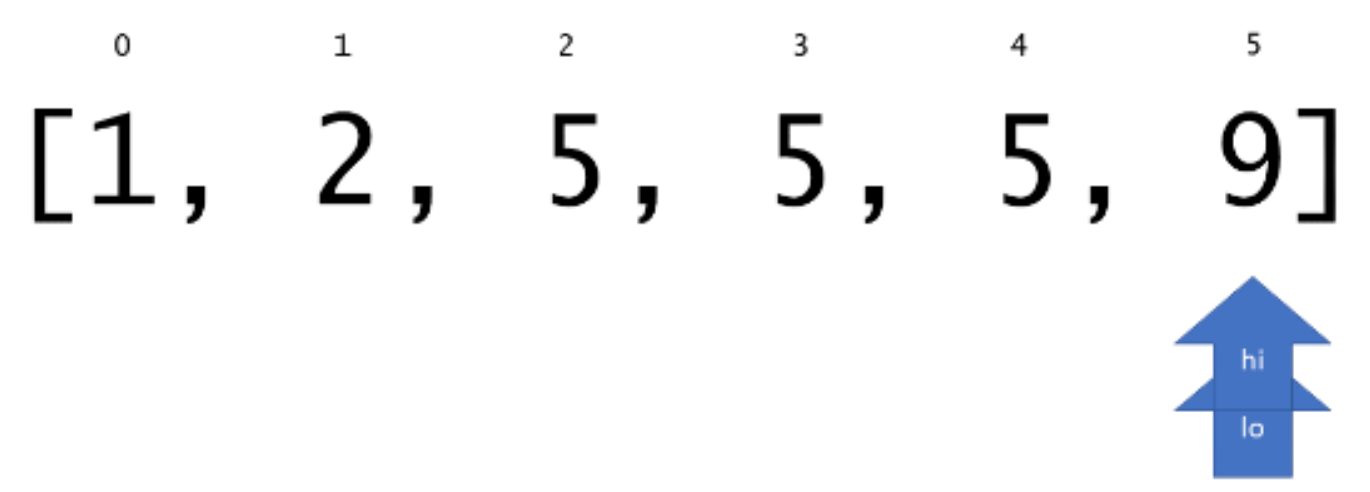


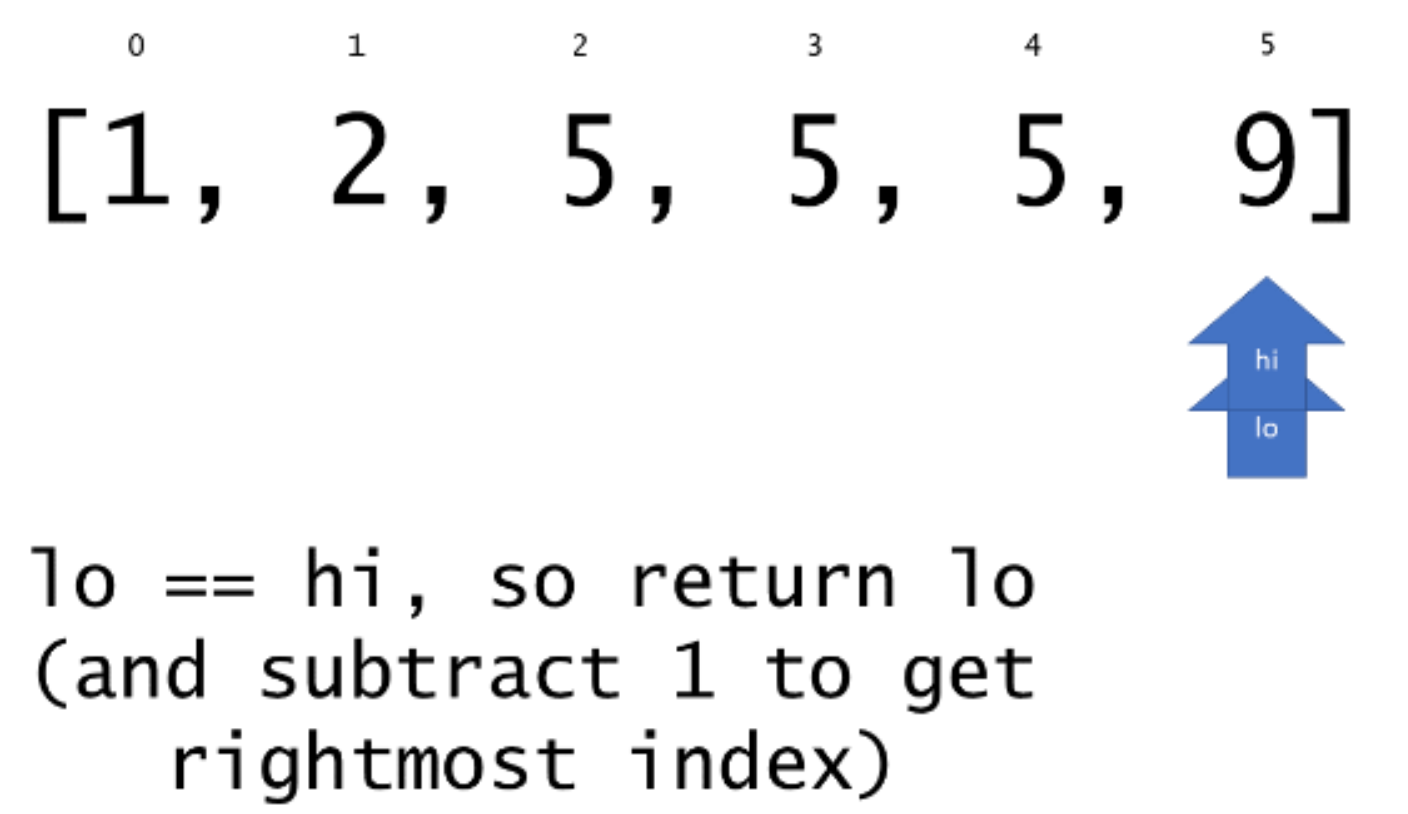










|  |
| --- |
| class Solution {  // returns leftmost (or rightmost) index at which `target` should be  // inserted in sorted array `nums` via binary search.  private int extremeInsertionIndex(int[] nums, int target, boolean left) {  int lo = 0;  int hi = nums.length;  while (lo < hi) {  int mid = (lo + hi) / 2;  if (nums[mid] > target || (left && target == nums[mid])) {  hi = mid;  }  else {  lo = mid+1;  }  }  return lo;  }  public int[] searchRange(int[] nums, int target) {  int[] targetRange = {-1, -1};  int leftIdx = extremeInsertionIndex(nums, target, true);  // assert that `leftIdx` is within the array bounds and that `target`  // is actually in `nums`.  if (leftIdx == nums.length || nums[leftIdx] != target) {  return targetRange;  }  targetRange[0] = leftIdx;  targetRange[1] = extremeInsertionIndex(nums, target, false)-1;  return targetRange;  }  } |

**Complexity Analysis**

* Time complexity : O(log*N*)
* Space complexity : O(1)

All work is done in place, so the overall memory usage is constant.

**Find K Closest Elements**

Given a **sorted** integer array arr, two integers k and x, return the k closest integers to x in the array. The result should also be sorted in ascending order.

An integer a is closer to x than an integer b if:

* |a - x| < |b - x|, or
* |a - x| == |b - x| and a < b

**Example 1:**

**Input:** arr = [1,2,3,4,5], k = 4, x = 3

**Output:** [1,2,3,4]

**Example 2:**

**Input:** arr = [1,2,3,4,5], k = 4, x = -1

**Output:** [1,2,3,4]

**Constraints:**

* 1 <= k <= arr.length
* 1 <= arr.length <= 104
* arr is sorted in **ascending** order.
* -104 <= arr[i], x <= 104

## Solution

#### **Approach 1: Using Collection.sort()**

**Algorithm**

Intuitively, we can sort the elements in list arr by their absolute difference values to the target x. Then the sublist of the first k elements is the result after sorting the elements by the natural order.

|  |
| --- |
| public List<Integer> findClosestElements(List<Integer> arr, int k, int x) {  Collections.sort(arr, (a,b) -> a == b ? a - b : Math.abs(a-x) - Math.abs(b-x));  arr = arr.subList(0, k);  Collections.sort(arr);  return arr;  } |

**Complexity Analysis**

* Time complexity : *O*(*n*log*n*). Collections.sort() uses binary sort so it has a *O*(*n*log*n*) complexity.
* Space complexity : *O*(*k*). The in-place sorting does not consume any extra space. However, generating a k length sublist will take some space.

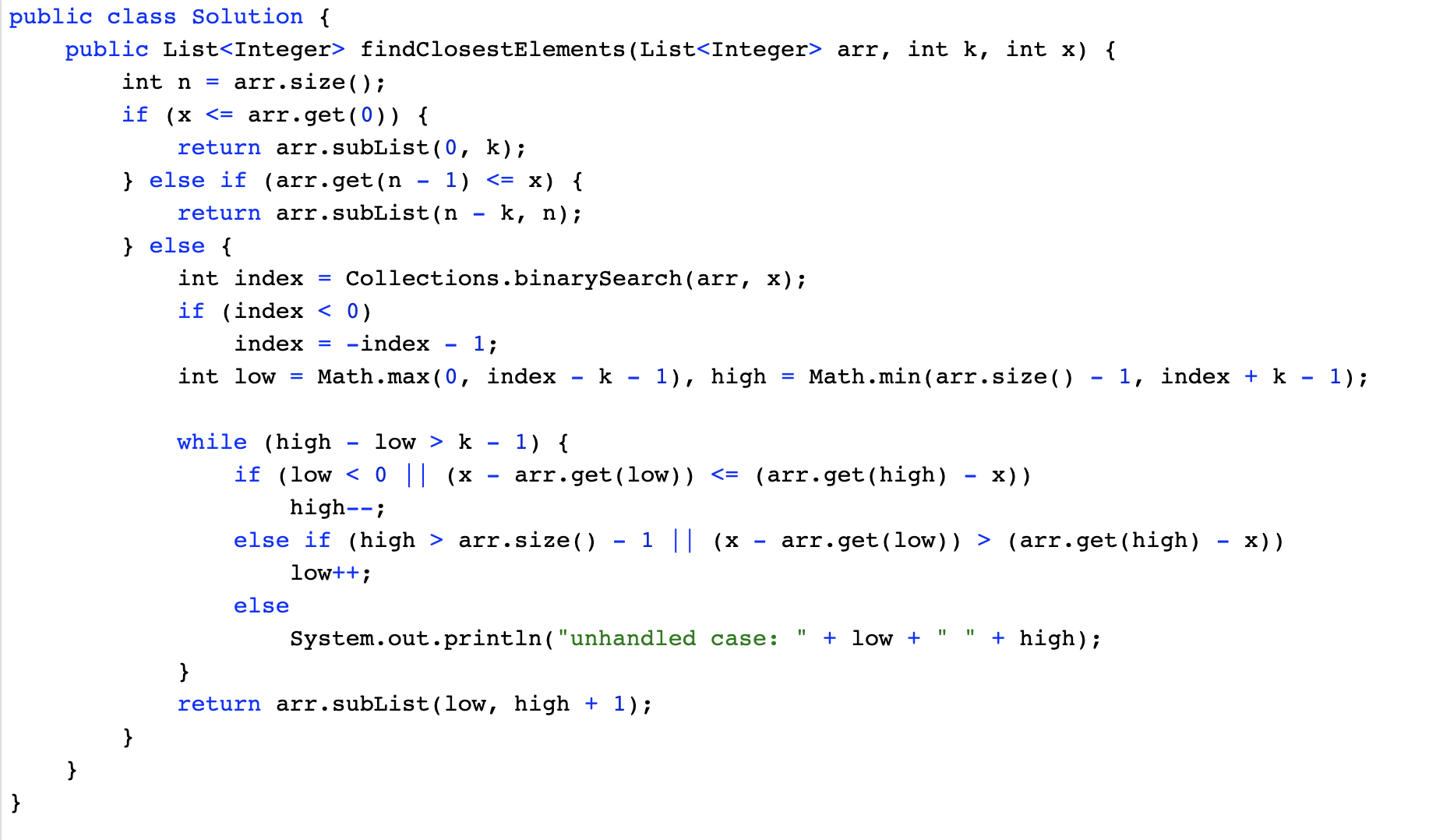
#### **Approach 2: Binary Search and Two Pointers**

**Algorithm**

The original array has been sorted so we can take this advantage by the following steps.

1. If the target x is less or equal than the first element in the sorted array, the first k elements are the result.
2. Similarly, if the target x is more or equal than the last element in the sorted array, the last k elements are the result.
3. Otherwise, we can use binary search to find the index of the element, which is equal (when this list has x) or a little bit larger than x (when this list does not have it). Then set low to its left k-1 position, and high to the right k-1 position of this index as a start. The desired k numbers must in this rang [index-k-1, index+k-1]. So we can shrink this range to get the result using the following rules.
   * If low reaches the lowest index 0 or the low element is closer to x than the high element, decrease the high index.
   * If high reaches to the highest index arr.size()-1 or it is nearer to x than the low element, increase the low index.
   * The looping ends when there are exactly k elements in [low, high], the subList of which is the result.

|  |
| --- |
| public class Solution {  public List<Integer> findClosestElements(List<Integer> arr, int k, int x) {  int n = arr.size();  if (x <= arr.get(0)) {  return arr.subList(0, k);  } else if (arr.get(n - 1) <= x) {  return arr.subList(n - k, n);  } else {  int index = Collections.binarySearch(arr, x);  if (index < 0)  index = -index - 1;  int low = Math.max(0, index - k - 1), high = Math.min(arr.size() - 1, index + k - 1);  while (high - low > k - 1) {  if (low < 0 || (x - arr.get(low)) <= (arr.get(high) - x))  high--;  else if (high > arr.size() - 1 || (x - arr.get(low)) > (arr.get(high) - x))  low++;  else  System.out.println("unhandled case: " + low + " " + high);  }  return arr.subList(low, high + 1);  }  }  } |



**Complexity Analysis**

* Time complexity : *O*(log*n*+*k*). *O*(log*n*) is for the time of binary search, while *O*(*k*) is for shrinking the index range to k elements.
* Space complexity : *O*(*k*). It is to generate the required sublist.

**Find Peak Element**

A peak element is an element that is strictly greater than its neighbors.

Given an integer array nums, find a peak element, and return its index. If the array contains multiple peaks, return the index to **any of the peaks**.

You may imagine that nums[-1] = nums[n] = -∞.

**Example 1:**

**Input:** nums = [1,2,3,1]

**Output:** 2

**Explanation:** 3 is a peak element and your function should return the index number 2.

**Example 2:**

**Input:** nums = [1,2,1,3,5,6,4]

**Output:** 5

**Explanation:** Your function can return either index number 1 where the peak element is 2, or index number 5 where the peak element is 6.

**Constraints:**

* 1 <= nums.length <= 1000
* -231 <= nums[i] <= 231 - 1
* nums[i] != nums[i + 1] for all valid i.

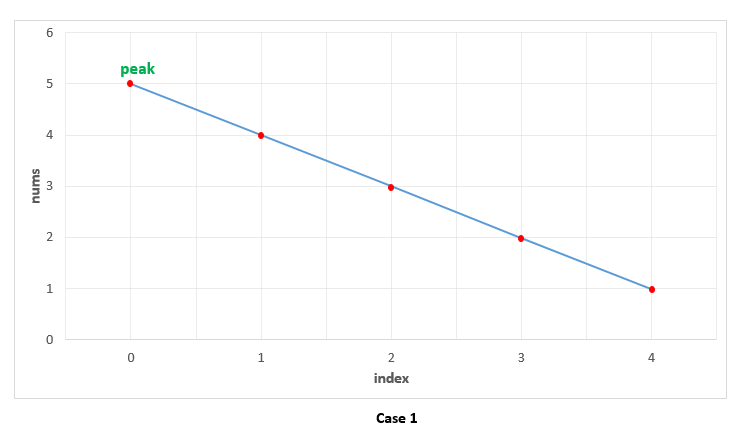
**Follow up:** Could you implement a solution with logarithmic complexity?

## Solution

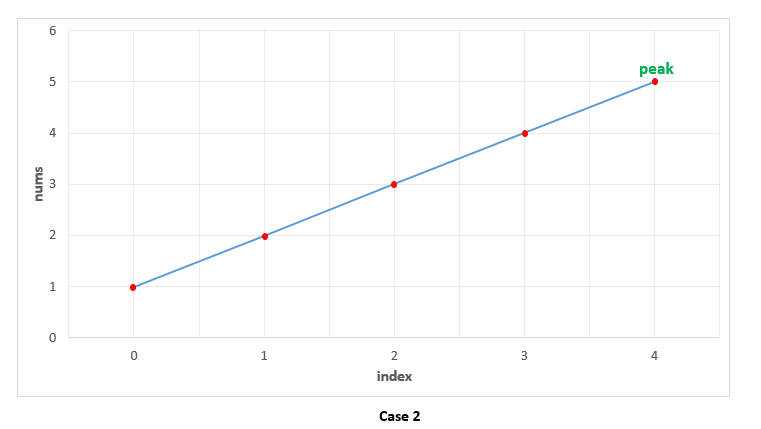
#### **Approach 1: Linear Scan**

In this approach, we make use of the fact that two consecutive numbers *nums*[*j*] and *nums*[*j*+1] are never equal. Thus, we can traverse over the *nums* array starting from the beginning. Whenever, we find a number *nums*[*i*], we only need to check if it is larger than the next number *nums*[*i*+1] for determining if *nums*[*i*] is the peak element. The reasoning behind this can be understood by taking the following three cases which cover every case into which any problem can be divided.

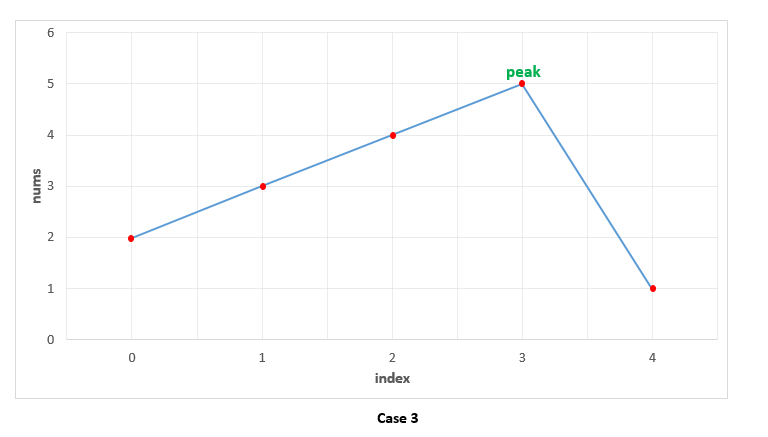
Case 1. All the numbers appear in a descending order. In this case, the first element corresponds to the peak element. We start off by checking if the current element is larger than the next one. The first element satisfies this criteria, and is hence identified as the peak correctly. In this case, we didn't reach a point where we needed to compare *nums*[*i*] with *nums*[*i*−1] also, to determine if it is the peak element or not.



Case 2. All the elements appear in ascending order. In this case, we keep on comparing *nums*[*i*] with *nums*[*i*+1] to determine if *nums*[*i*] is the peak element or not. None of the elements satisfy this criteria, indicating that we are currently on a rising slope and not on a peak. Thus, at the end, we need to return the last element as the peak element, which turns out to be correct. In this case also, we need not compare *nums*[*i*] with *nums*[*i*−1], since being on the rising slope is a sufficient condition to ensure that *nums*[*i*] isn't the peak element.



Case 3. The peak appears somewhere in the middle. In this case, when we are traversing on the rising edge, as in Case 2, none of the elements will satisfy *nums*[*i*]>*nums*[*i*+1]. We need not compare *nums*[*i*] with *nums*[*i*−1] on the rising slope as discussed above. When we finally reach the peak element, the condition *nums*[*i*]>*nums*[*i*+1] is satisfied. We again, need not compare *nums*[*i*] with *nums*[*i*−1]. This is because, we could reach *nums*[*i*] as the current element only when the check *nums*[*i*]>*nums*[*i*+1] failed for the previous((*i*−1)*th* element, indicating that *nums*[*i*−1]<*nums*[*i*]. Thus, we are able to identify the peak element correctly in this case as well.



|  |
| --- |
| public class Solution {  public int findPeakElement(int[] nums) {  for (int i = 0; i < nums.length - 1; i++) {  if (nums[i] > nums[i + 1])  return i;  }  return nums.length - 1;  }  } |

**Complexity Analysis**

* Time complexity : *O*(*n*). We traverse the *nums* array of size *n* once only.
* Space complexity : *O*(1). Constant extra space is used.

#### **Approach 2: Recursive Binary Search**

**Algorithm**

We can view any given sequence in *nums* array as alternating ascending and descending sequences. By making use of this, and the fact that we can return any peak as the result, we can make use of Binary Search to find the required peak element.

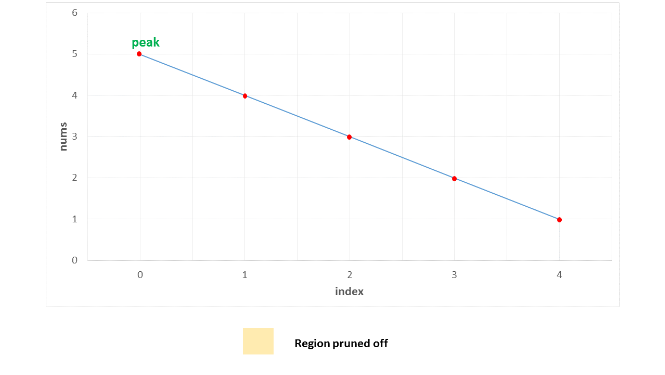
In case of simple Binary Search, we work on a sorted sequence of numbers and try to find out the required number by reducing the search space at every step. In this case, we use a modification of this simple Binary Search to our advantage. We start off by finding the middle element, *mid* from the given *nums* array. If this element happens to be lying in a descending sequence of numbers. or a local falling slope(found by comparing *nums*[*i*] to its right neighbour), it means that the peak will always lie towards the left of this element. Thus, we reduce the search space to the left of mid*mid*(including itself) and perform the same process on left subarray.

If the middle element, *mid* lies in an ascending sequence of numbers, or a rising slope(found by comparing *nums*[*i*] to its right neighbour), it obviously implies that the peak lies towards the right of this element. Thus, we reduce the search space to the right of *mid* and perform the same process on the right subarray.

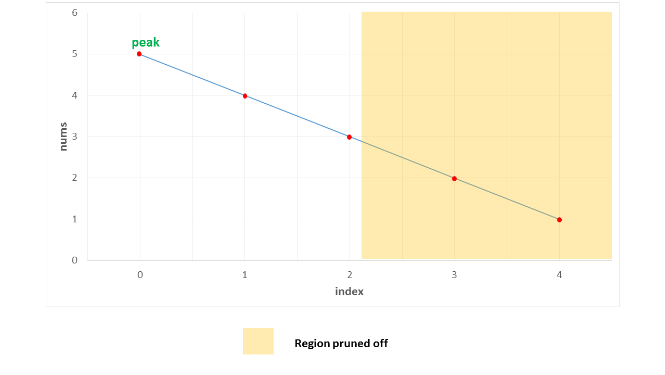
In this way, we keep on reducing the search space till we eventually reach a state where only one element is remaining in the search space. This single element is the peak element.

To see how it works, let's consider the three cases discussed above again.

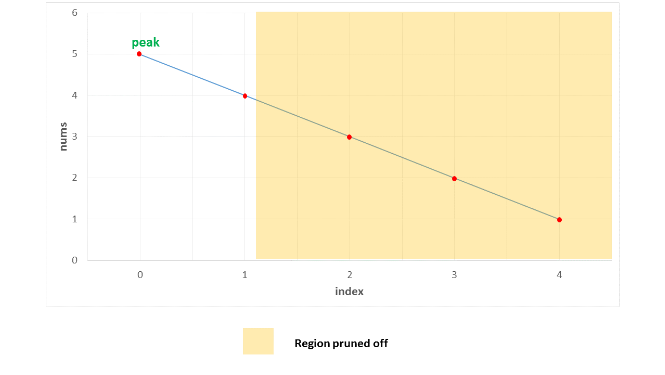
Case 1. In this case, we firstly find 33 as the middle element. Since it lies on a falling slope, we reduce the search space to [1, 2, 3]. For this subarray, 22 happens to be the middle element, which again lies on a falling slope, reducing the search space to [1, 2]. Now, 11 acts as the middle element and it lies on a falling slope, reducing the search space to [1] only. Thus, 11 is returned as the peak correctly.

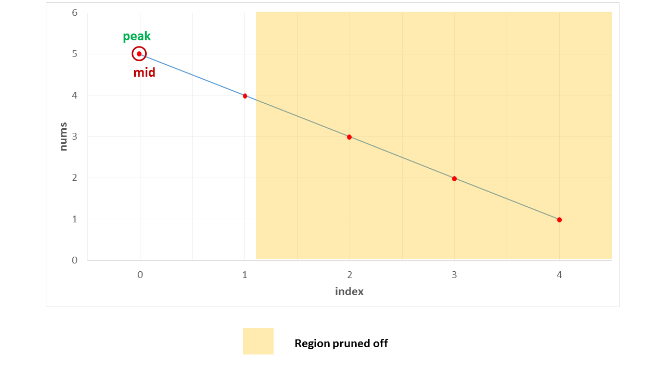


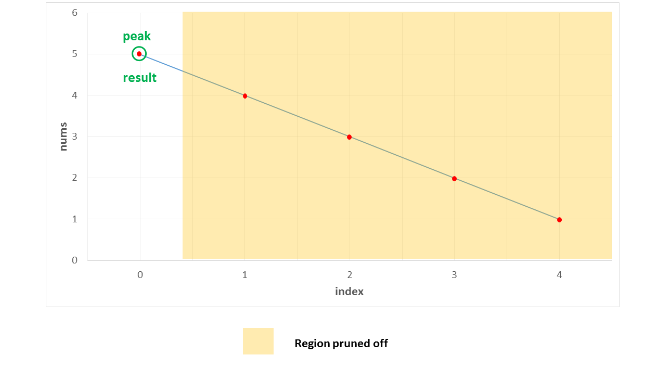




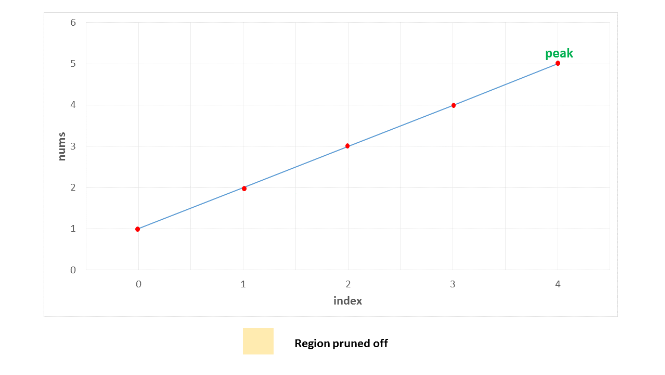


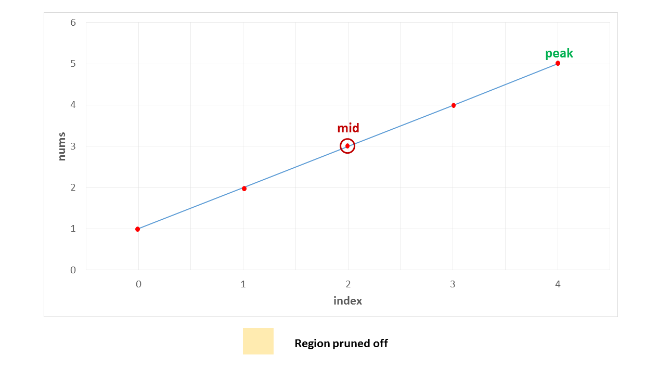


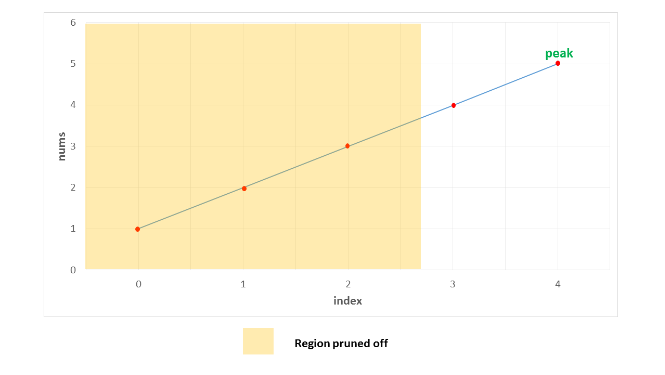


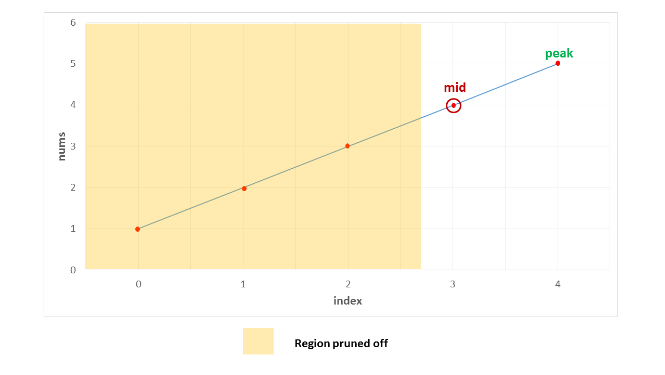


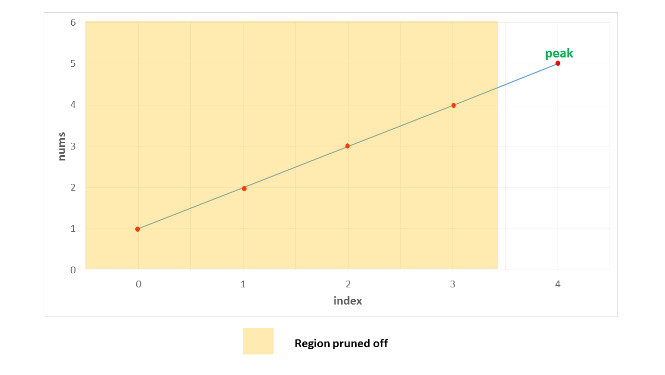
Case 2. In this case, we firstly find 3 as the middle element. Since it lies on a rising slope, we reduce the search space to [4, 5]. Now, 4 acts as the middle element for this subarray and it lies on a rising slope, reducing the search space to [5] only. Thus, 5 is returned as the peak correctly.





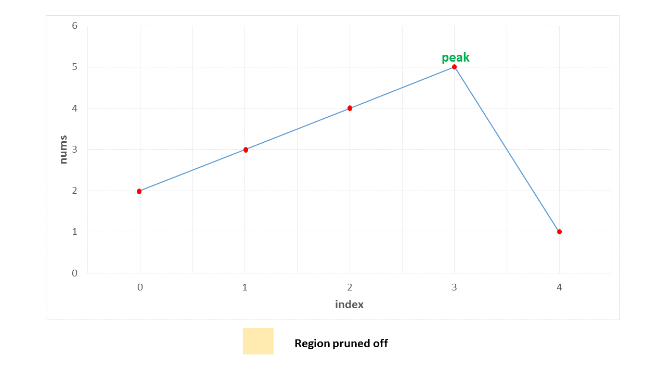




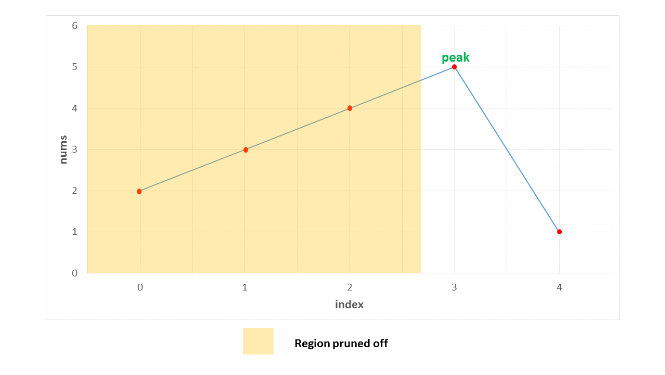


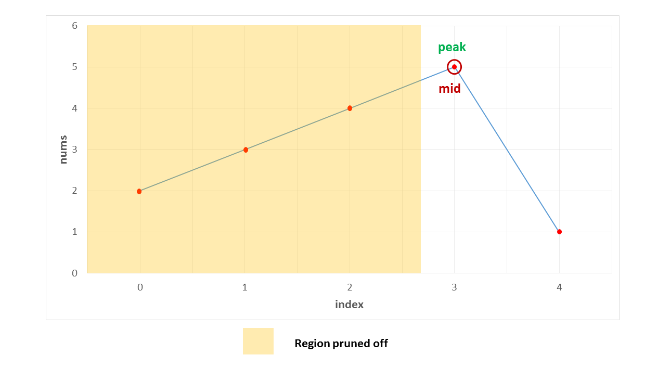


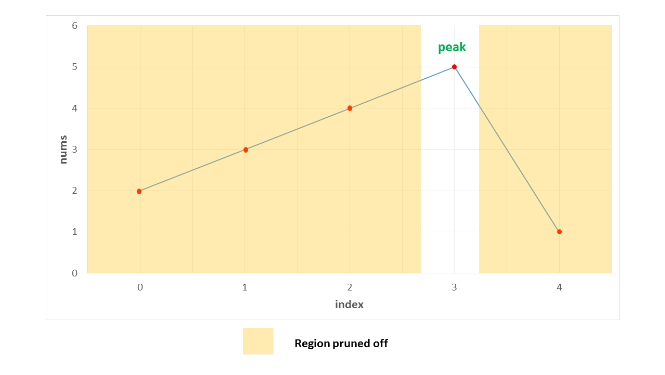
Case 3. In this case, the peak lies somewhere in the middle. The first middle element is 4. It lies on a rising slope, indicating that the peak lies towards its right. Thus, the search space is reduced to [5, 1]. Now, 5 happens to be the on a falling slope(relative to its right neighbour), reducing the search space to [5] only. Thus, 5 is identified as the peak element correctly.

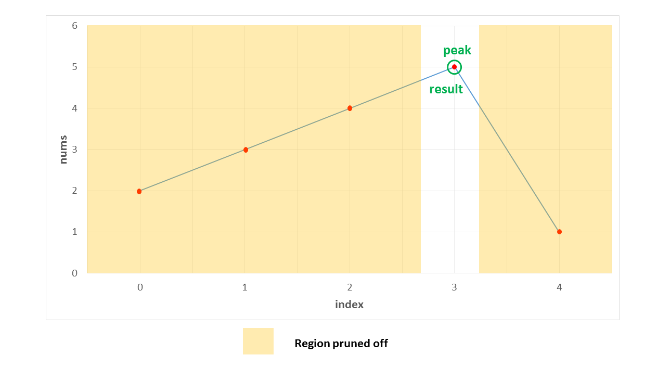












|  |
| --- |
| public class Solution {  public int findPeakElement(int[] nums) {  return search(nums, 0, nums.length - 1);  }  public int search(int[] nums, int l, int r) {  if (l == r)  return l;  int mid = (l + r) / 2;  if (nums[mid] > nums[mid + 1])  return search(nums, l, mid);  return search(nums, mid + 1, r);  }  } |

**Complexity Analysis**

* Time complexity : *O*(*log*2​(*n*)). We reduce the search space in half at every step. Thus, the total search space will be consumed in *log*2​(*n*) steps. Here, *n* refers to the size of *nums* array.
* Space complexity : *O*(*log*2​(*n*)). We reduce the search space in half at every step. Thus, the total search space will be consumed in *log*2​(*n*) steps. Thus, the depth of recursion tree will go upto *log*2​(*n*).

#### **Approach 3: Iterative Binary Search**

**Algorithm**

The binary search discussed in the previous approach used a recursive method. We can do the same process in an iterative fashion also. This is done in the current approach.

|  |
| --- |
| public class Solution {  public int findPeakElement(int[] nums) {  int l = 0, r = nums.length - 1;  while (l < r) {  int mid = (l + r) / 2;  if (nums[mid] > nums[mid + 1])  r = mid;  else  l = mid + 1;  }  return l;  }  } |

**Complexity Analysis**

* Time complexity : *O*(*log*2​(*n*)). We reduce the search space in half at every step. Thus, the total search space will be consumed in *log*2​(*n*) steps. Here, *n* refers to the size of *nums* array.
* Space complexity : *O*(1). Constant extra space is used.

## Template Analysis

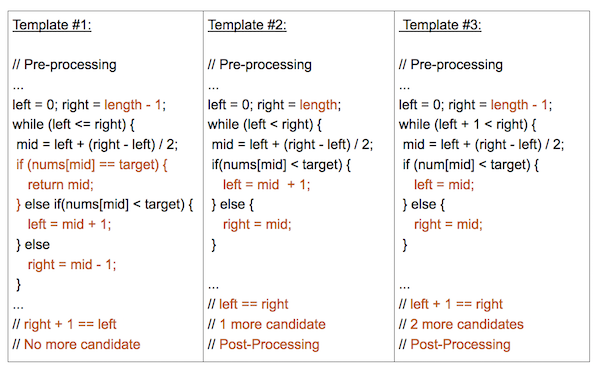
This chapter sums up the 3 different templates that we introduced earlier and analyzes them for specific use cases. Furthermore, a brief explanation of their key differences is emphasized.

**Binary Search Template Analysis**

**Template Explanation:**

99% of binary search problems that you see online will fall into 1 of these 3 templates. Some problems can be implemented using multiple templates, but as you practice more, you will notice that some templates are more suited for certain problems than others.

**Note:** The templates and their differences have been colored coded below.



These 3 templates differ by their:

* left, mid, right index assignments
* loop or recursive termination condition
* necessity of post-processing

Template 1 and 3 are the most commonly used and almost all binary search problems can be easily implemented in one of them. Template 2 is a bit more advanced and used for certain types of problems.

Each of these 3 provided templates provide a specific use case:

**Template #1** (left <= right):

* Most basic and elementary form of Binary Search
* Search Condition can be determined without comparing to the element's neighbors (or use specific elements around it)
* No post-processing required because at each step, you are checking to see if the element has been found. If you reach the end, then you know the element is not found

**Template #2** (left < right):

* An advanced way to implement Binary Search.
* Search Condition needs to access element's immediate right neighbor
* Use element's right neighbor to determine if condition is met and decide whether to go left or right
* Gurantees Search Space is at least 2 in size at each step
* Post-processing required. Loop/Recursion ends when you have 1 element left. Need to assess if the remaining element meets the condition.

**Template #3** (left + 1 < right):

* An alternative way to implement Binary Search
* Search Condition needs to access element's immediate left and right neighbors
* Use element's neighbors to determine if condition is met and decide whether to go left or right
* Gurantees Search Space is at least 3 in size at each step
* Post-processing required. Loop/Recursion ends when you have 2 elements left. Need to assess if the remaining elements meet the condition.

**Time and Space Complexity:**

**Runtime:** O(log n) -- Logorithmic Time

Because Binary Search operates by applying a condition to the value in the middle of our search space and thus cutting the search space in half, in the worse case, we will have to make O(log n) comparisons, where n is the number of elements in our collection.

Why log n?

* Binary search is performed by dividing the existing array in half.
* So every time you a call the subroutine ( or complete one iteration ) the size reduced to half of the existing part.
* First N become N/2, then it become N/4 and go on till it find the element or size become 1.
* The maximum no of iterations is log N (base 2).

**Space:** O(1) -- Constant Space

Although, Binary Search does require keeping track of 3 indicies, the iterative solution does not typically require any other additional space and can be applied directly on the collection itself, therefore warrants O(1) or constant space.

**Other Types of Binary Search:**

**Below, we have provided another type of Binary Search for practice.**

Binary Search With 2 Arrays -- In this problem, we need to compare values from 2 arrays to determine our search space: [LC #4: Median of Two Sorted Arrays](https://leetcode.com/problems/median-of-two-sorted-arrays/)

**Closest Binary Search Tree Value**

Given a non-empty binary search tree and a target value, find the value in the BST that is closest to the target.

**Note:**

* Given target value is a floating point.
* You are guaranteed to have only one unique value in the BST that is closest to the target.

**Example:**

**Input:** root = [4,2,5,1,3], target = 3.714286

4

/ \

2 5

/ \

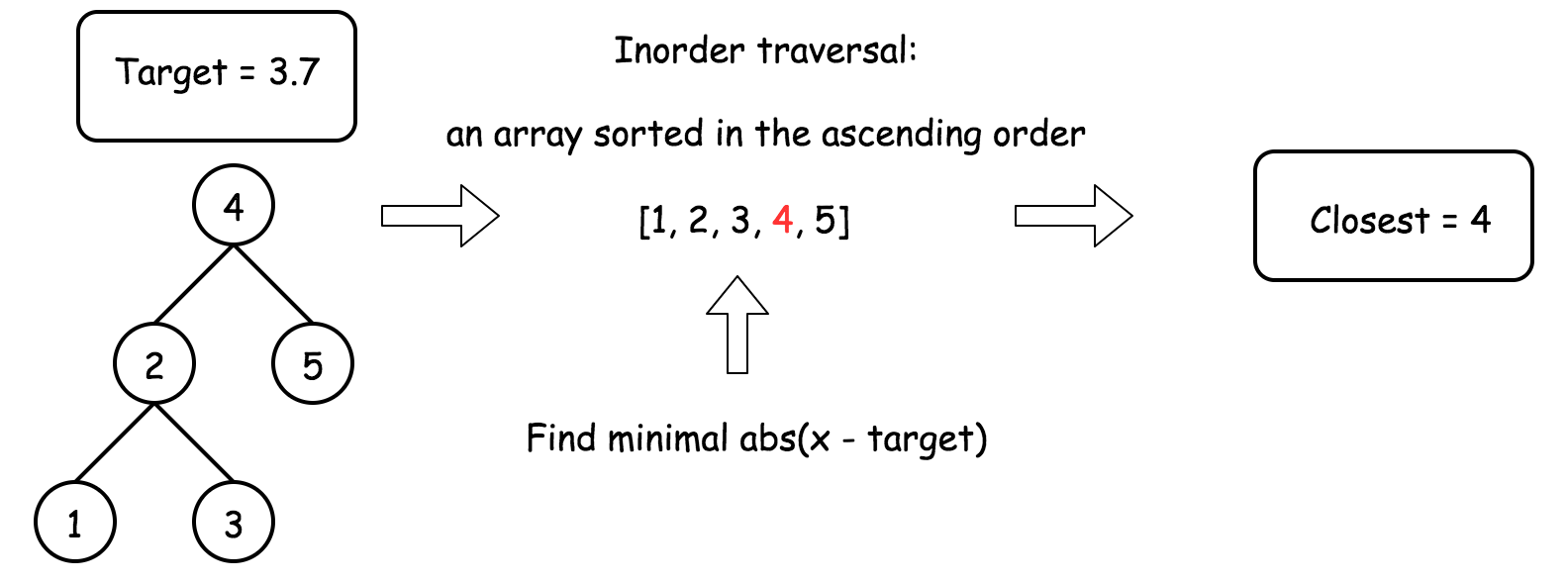
1 3

**Output:** 4

#### **Approach 1: Recursive Inorder + Linear search, O(N) time**

**Intuition**

The simplest approach (3 lines in Python) is to build inorder traversal and then find the closest element in a sorted array with built-in function min.



This approach is simple stupid, and serves to identify the subproblems.

**Algorithm**

* Build an inorder traversal array.
* Find the closest to target element in that array.

**Implementation**

|  |
| --- |
| class Solution {  public void inorder(TreeNode root, List<Integer> nums) {  if (root == null) return;  inorder(root.left, nums);  nums.add(root.val);  inorder(root.right, nums);  }  public int closestValue(TreeNode root, double target) {  List<Integer> nums = new ArrayList();  inorder(root, nums);  return Collections.min(nums, new Comparator<Integer>() {  @Override  public int compare(Integer o1, Integer o2) {  return Math.abs(o1 - target) < Math.abs(o2 - target) ? -1 : 1;  }  });  }  } |

**Complexity Analysis**

* Time complexity : O(*N*) because to build inorder traversal and then to perform linear search takes linear time.
* Space complexity : O(*N*) to keep inorder traversal.

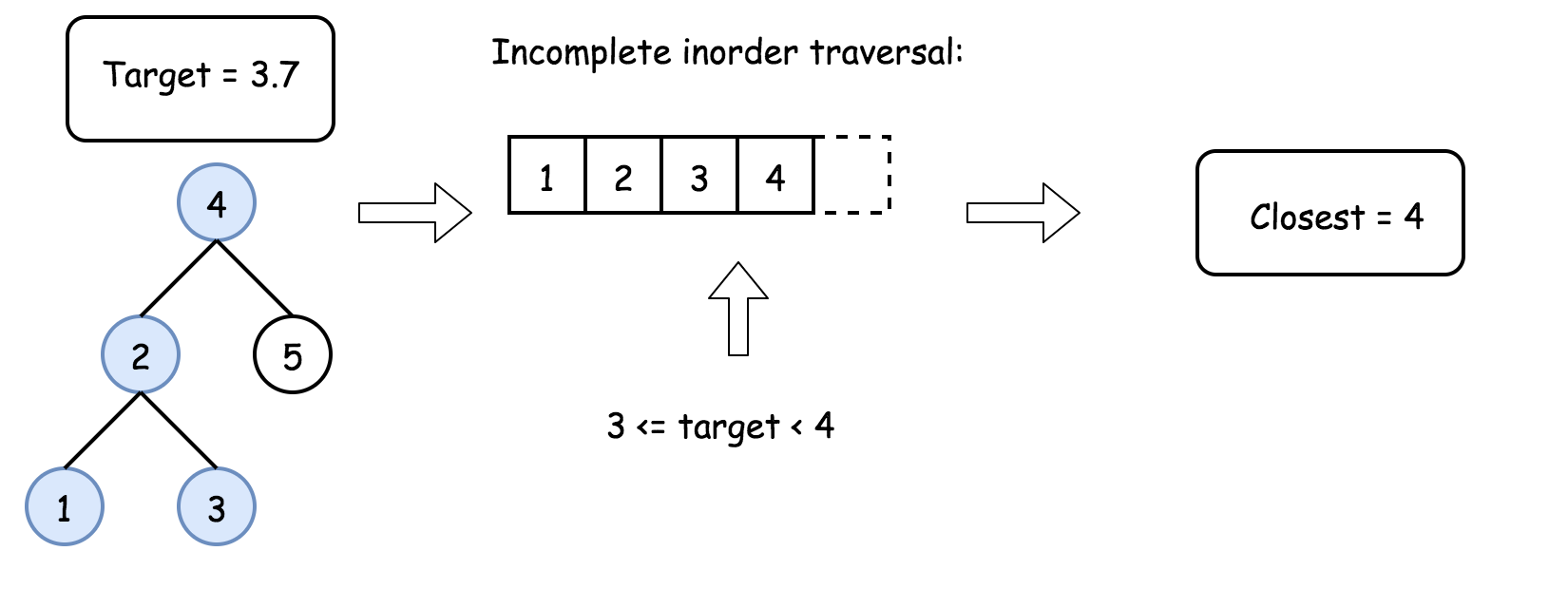
#### **Approach 2: Iterative Inorder, O(k) time**

**Intuition**

Let's optimise Approach 1 in the case when index k of the closest element is much smaller than the tree heigh H.

First, one could merge both steps by traversing the tree and searching the closest value at the same time.

Second, one could stop just after identifying the closest value, there is no need to traverse the whole tree. The closest value is found if the target value is in-between of two inorder array elements nums[i] <= target < nums[i + 1]. Then the closest value is one of these elements.



**Algorithm**

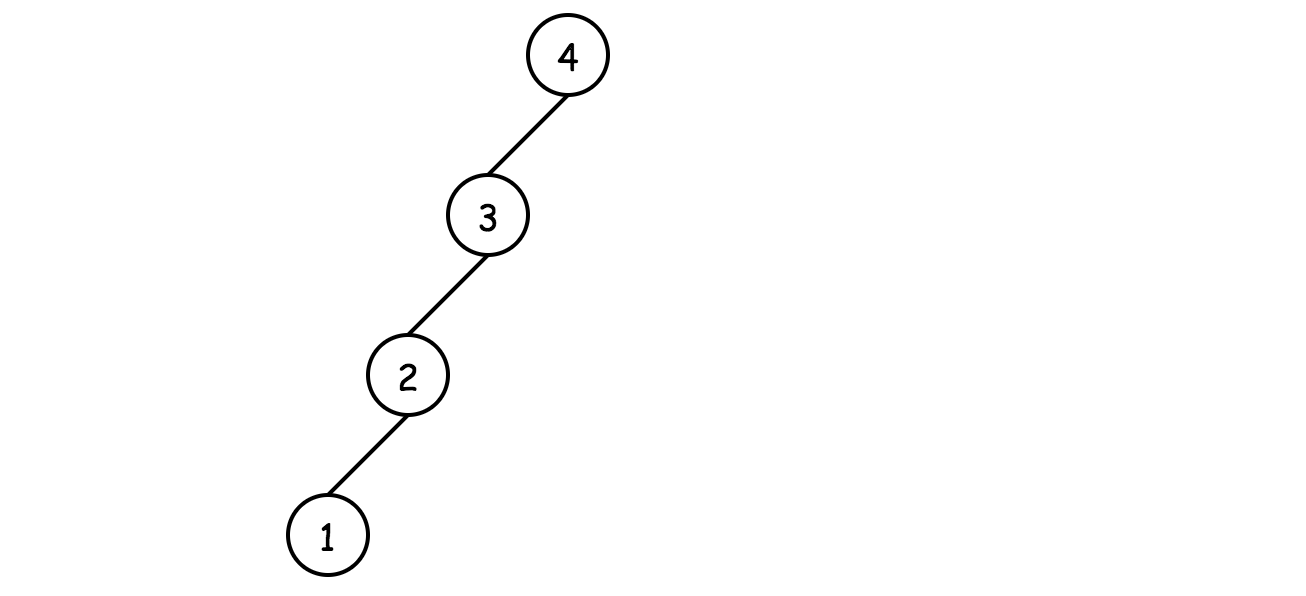
* Initiate stack as an empty array and predecessor value as a very small number.
* While root is not null:
  + To build an inorder traversal iteratively, go left as far as you can and add all nodes on the way into stack.
  + Pop the last element from stack root = stack.pop().
  + If target is in-between of pred and root.val, return the closest between these two elements.
  + Set predecessor value to be equal to root.val and go one step right: root = root.right.
* We're here because during the loop one couldn't identify the closest value. That means that the closest value is the last value in the inorder traversal, i.e. current predecessor value. Return it.

**Implementation**

|  |
| --- |
| class Solution {  public int closestValue(TreeNode root, double target) {  LinkedList<TreeNode> stack = new LinkedList();  long pred = Long.MIN\_VALUE;  while (!stack.isEmpty() || root != null) {  while (root != null) {  stack.add(root);  root = root.left;  }  root = stack.removeLast();  if (pred <= target && target < root.val)  return Math.abs(pred - target) < Math.abs(root.val - target) ? (int)pred : root.val;  pred = root.val;  root = root.right;  }  return (int)pred;  }  } |

**Complexity Analysis**

* Time complexity : O(*k*) in the average case and O(*H*+*k*) in the worst case, where k is an index of closest element. It's known that [average case is a balanced tree](https://pages.cpsc.ucalgary.ca/~jacobs/Courses/cpsc331/F08/notes/lecture17.pdf), in that case stack always contains a few elements, and hence one does 2*k* operations to go to kth element in inorder traversal (k times to push into stack and then k times to pop out of stack). That results in O(*k*) time complexity. The worst case is a completely unbalanced tree, then you first push H elements into stack and then pop out k elements, that results in O(*H*+*k*) time complexity.



* Space complexity : up to O(*H*) to keep the stack in the case of non-balanced tree.

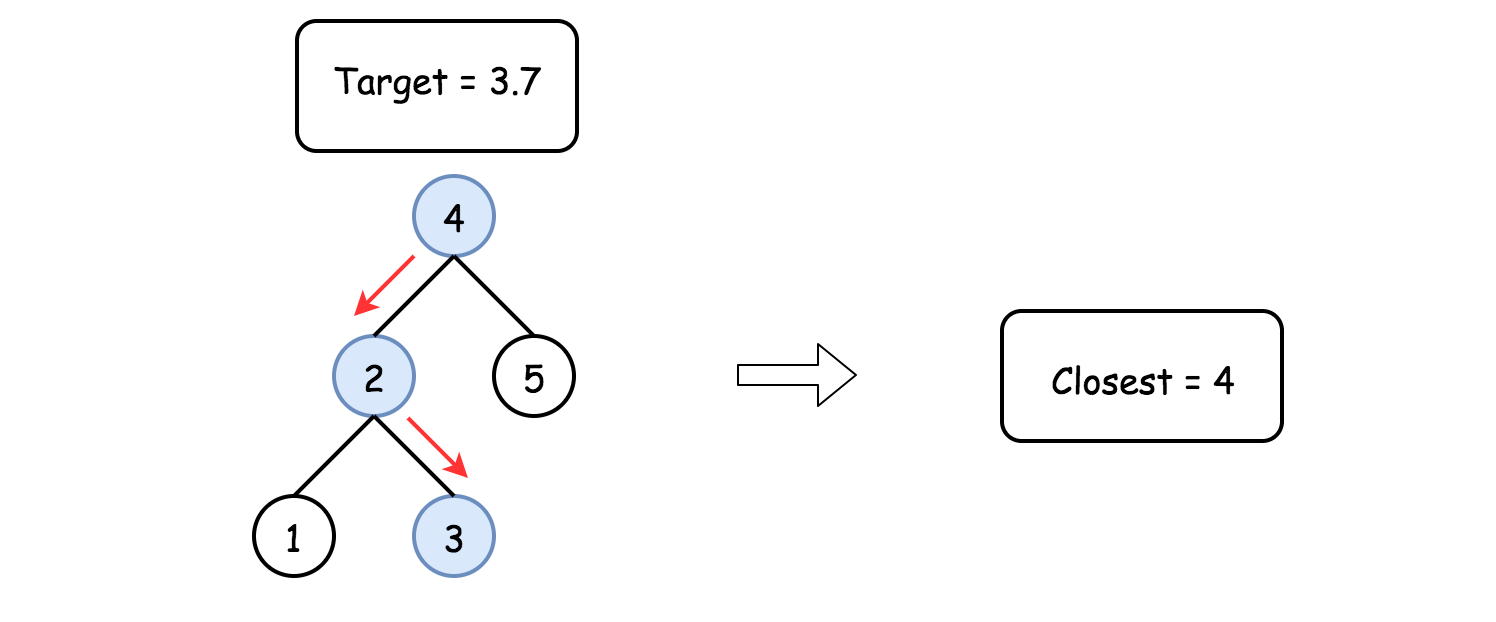
#### **Approach 3: Binary Search, O(H) time**

**Intuition**

Approach 2 works fine when index k of closest element is much smaller than the tree height H.

Let's now consider another limit and optimise Approach 1 in the case of relatively large k, comparable with N.

Then it makes sense to use a binary search: go left if target is smaller than current root value, and go right otherwise. Choose the closest to target value at each step.



Kudos for this solution go to @[stefanpochmann](https://leetcode.com/stefanpochmann/).

**Implementation**

|  |
| --- |
| class Solution {  public int closestValue(TreeNode root, double target) {  int val, closest = root.val;  while (root != null) {  val = root.val;  closest = Math.abs(val - target) < Math.abs(closest - target) ? val : closest;  root = target < root.val ? root.left : root.right;  }  return closest;  }  } |

**Complexity Analysis**

* Time complexity : O(*H*) since here one goes from root down to a leaf.
* Space complexity : O(1).

**Search in a Sorted Array of Unknown Size**

Given an integer array sorted in ascending order, write a function to search target in nums.  If target exists, then return its index, otherwise return -1. **However, the array size is unknown to you**. You may only access the array using an ArrayReader interface, where ArrayReader.get(k) returns the element of the array at index k (0-indexed).

You may assume all integers in the array are less than 10000, and if you access the array out of bounds, ArrayReader.get will return 2147483647.

**Example 1:**

**Input:** array = [-1,0,3,5,9,12], target = 9

**Output:** 4

**Explanation:** 9 exists in nums and its index is 4

**Example 2:**

**Input:** array = [-1,0,3,5,9,12], target = 2

**Output:** -1

**Explanation:** 2 does not exist in nums so return -1

**Constraints:**

* You may assume that all elements in the array are unique.
* The value of each element in the array will be in the range [-9999, 9999].
* The length of the array will be in the range [1, 10^4].

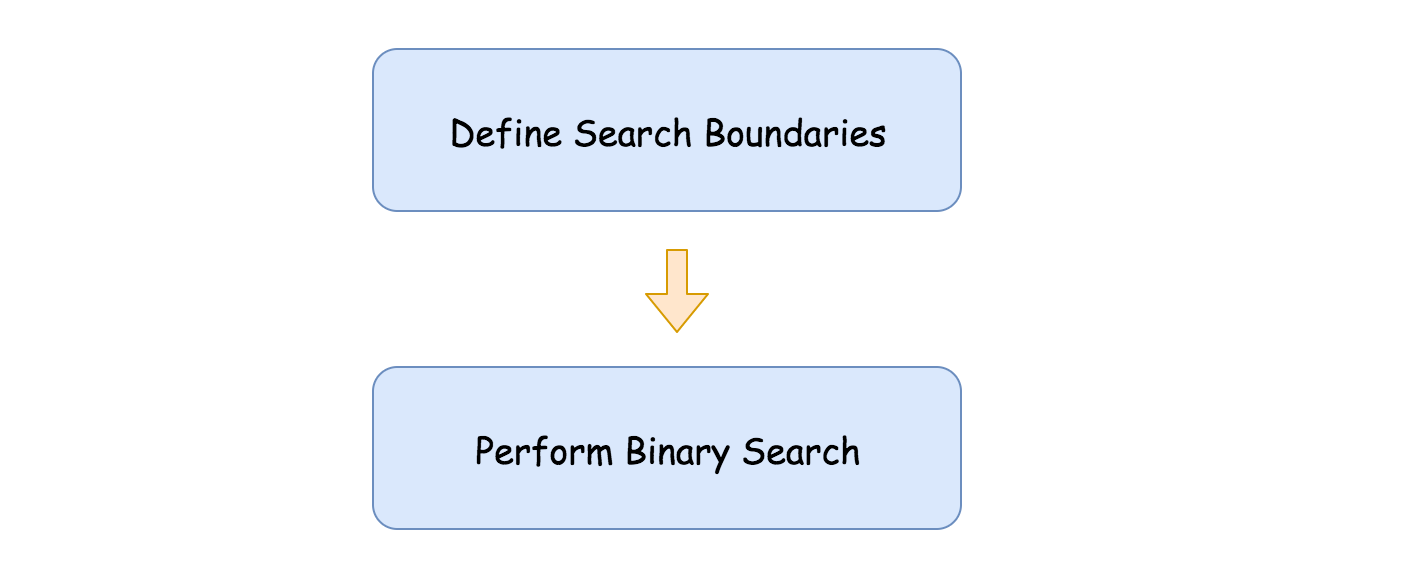
## Solution

#### **Approach 1: Binary Search**

**Split into Two Subproblems**

The array is sorted, i.e. one could try to fit into a logarithmic time complexity. That means two subproblems, and both should be done in a logarithmic time:

* Define search limits, i.e. left and right boundaries for the search.
* Perform binary search in the defined boundaries.

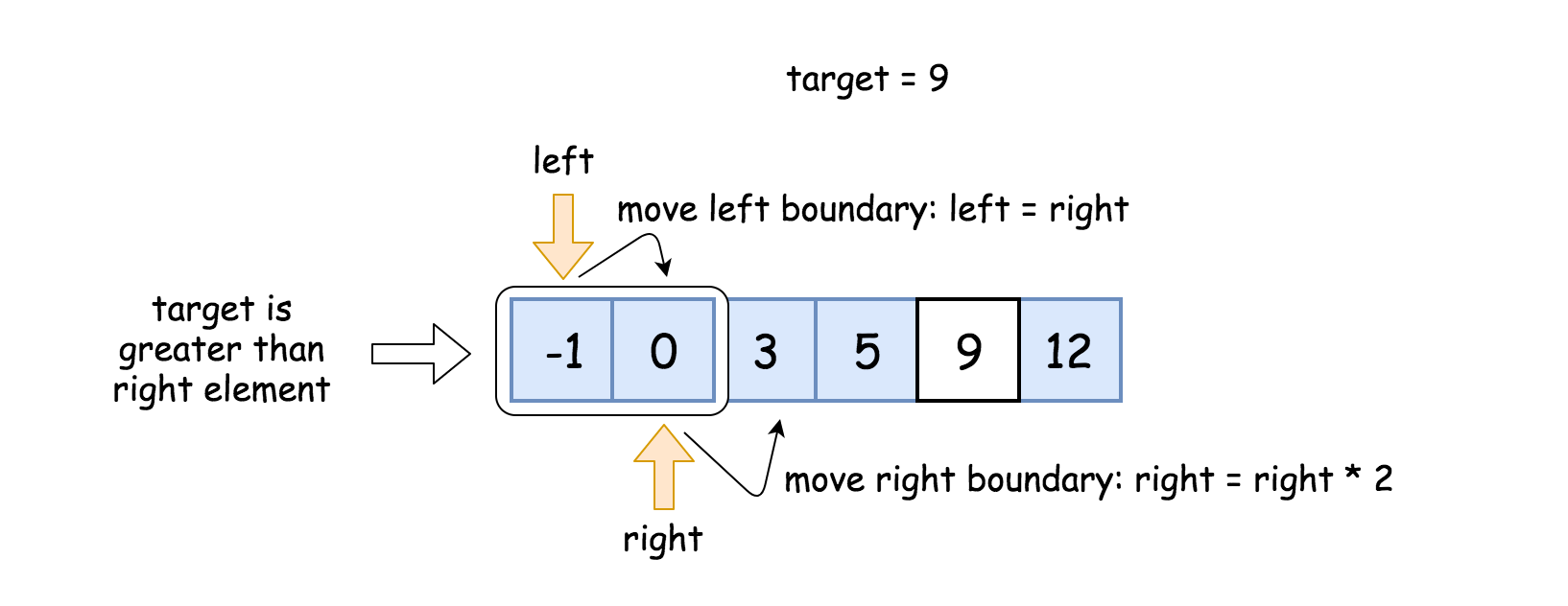


**Define Search Boundaries**

This is a key subproblem here.

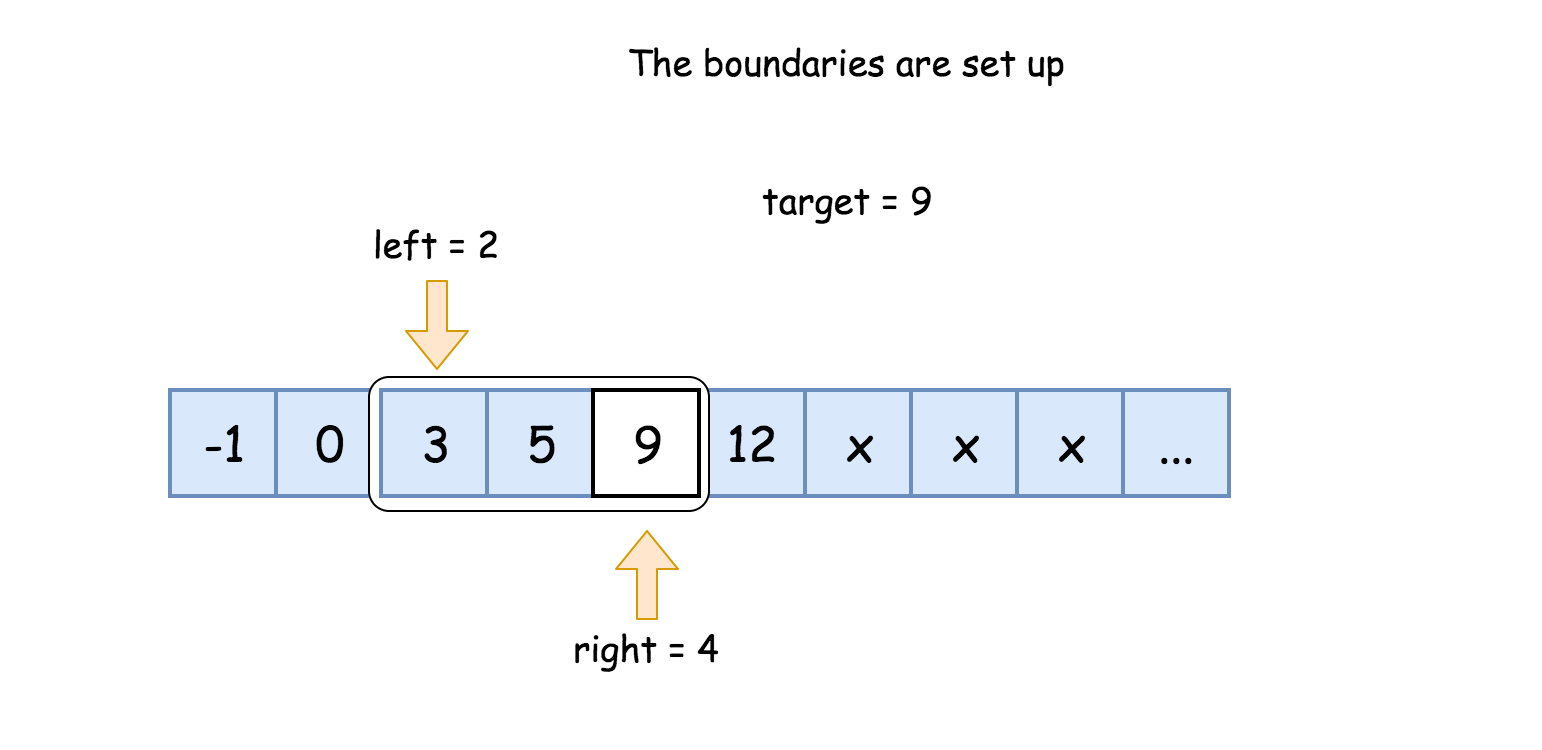
The idea is quite simple. Let's take two first indexes, 0 and 1, as left and right boundaries. If the target value is not among these zeroth and the first element, then it's outside the boundaries, on the right.

That means that the left boundary could moved to the right, and the right boundary should be extended. To keep logarithmic time complexity, let's extend it twice as far: right = right \* 2.



If the target now is less than the right element, we're done, the boundaries are set. If not, repeat these two steps till the boundaries are established:

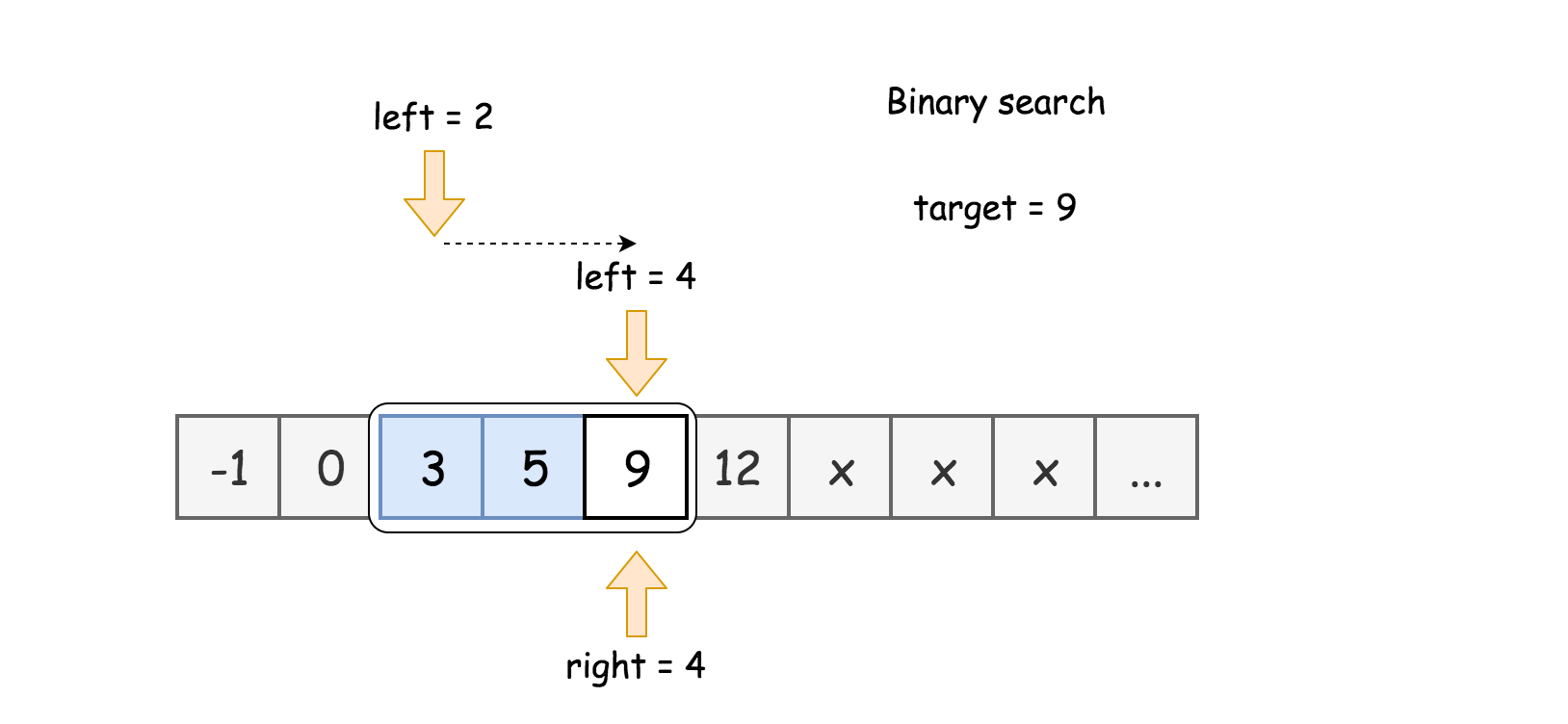
* Move the left boundary to the right: left = right.
* Extend the right boundary: right = right \* 2.



**Binary Search**

[Binary search](https://leetcode.com/explore/learn/card/binary-search/) is a textbook algorithm with a logarithmic time complexity. It's based on the idea to compare the target value to the middle element of the array.

* If the target value is equal to the middle element - we're done.
* If the target value is smaller - continue to search on the left.
* If the target value is larger - continue to search on the right.



**Prerequisites: left and right shifts**

To speed up, one could use here [bitwise shifts](https://wiki.python.org/moin/BitwiseOperators):

* Left shift: x << 1. The same as multiplying by 2: x \* 2.
* Right shift: x >> 1. The same as dividing by 2: x / 2.

**Algorithm**

Define boundaries:

* Initiate left = 0 and right = 1.
* While target is on the right to the right boundary: reader.get(right) < target:
  + Set left boundary equal to the right one: left = right.
  + Extend right boundary: right \*= 2. To speed up, use right shift instead of multiplication: right <<= 1.
* Now the target is between left and right boundaries.

Binary Search:

* While left <= right:
  + Pick a pivot index in the middle: pivot = (left + right) / 2. To avoid overflow, use the form pivot = left + ((right - left) >> 1) instead of straightforward expression above.
  + Retrieve the element at this index: num = reader.get(pivot).
  + Compare middle element num to the target value.
    - If the middle element is the target num == target: return pivot.
    - If the target is not yet found:
      * If num > target, continue to search on the left right = pivot - 1.
      * Else continue to search on the right left = pivot + 1.
* We're here because target is not found. Return -1.

**Implementation**

|  |
| --- |
| class Solution {  public int search(ArrayReader reader, int target) {  if (reader.get(0) == target) return 0;  // search boundaries  int left = 0, right = 1;  while (reader.get(right) < target) {  left = right;  right <<= 1;  }  // binary search  int pivot, num;  while (left <= right) {  pivot = left + ((right - left) >> 1);  num = reader.get(pivot);  if (num == target) return pivot;  if (num > target) right = pivot - 1;  else left = pivot + 1;  }  // there is no target element  return -1;  }  } |

**Complexity Analysis**

* Time complexity : O(log*T*), where *T* is an index of target value.

There are two operations here: to define search boundaries and to perform binary search.

Let's first find the number of steps k to setup the boundaries. On the first step, the boundaries are 20 .. 20 + 1 on the second step 21 .. 21 + 1, etc. When everything is done, the boundaries are 2k .. 2k + 1 and 2k < T ≤ 2k + 1. That means one needs *k*=log*T* steps to setup the boundaries, that means O(log*T*) time complexity.

Now let's discuss the complexity of the binary search. There are 2k + 1 - 2k = 2k elements in the boundaries, i.e. 2log T = T elements. [As discussed](https://leetcode.com/articles/binary-search/), binary search has logarithmic complexity, that results in O(log*T*) time complexity.

* Space complexity : O(1) since it's a constant space solution.

## Conclusion

Binary Search is an immensely useful technique used to tackle different algorithmic problems. Practice identifying Binary Search Problems and applying different templates to different search conditions. Improve your approach to tackling problems, notice the patterns and repeat!

This chapter concludes our Binary Search learnings and summarizes key concepts. Below you can find some problems to help you practice Binary Search!

**Pow(x, n)**

Implement [pow(x, n)](http://www.cplusplus.com/reference/valarray/pow/), which calculates x raised to the power n (i.e. xn).

**Example 1:**

**Input:** x = 2.00000, n = 10

**Output:** 1024.00000

**Example 2:**

**Input:** x = 2.10000, n = 3

**Output:** 9.26100

**Example 3:**

**Input:** x = 2.00000, n = -2

**Output:** 0.25000

**Explanation:** 2-2 = 1/22 = 1/4 = 0.25

**Constraints:**

* -100.0 < x < 100.0
* -231 <= n <= 231-1
* -104 <= xn <= 104

#### **Approach 1: Brute Force**

**Intuition**

Just simulate the process, multiply x for n times.

If *n*<0, we can substitute *x*,*n* with 1/x, -n to make sure *n*≥0. This restriction can simplify our further discussion.

But we need to take care of the corner cases, especially different range limits for negative and positive integers.

**Algorithm**

We can use a straightforward loop to compute the result.

|  |
| --- |
| class Solution {  public double myPow(double x, int n) {  long N = n;  if (N < 0) {  x = 1 / x;  N = -N;  }  double ans = 1;  for (long i = 0; i < N; i++)  ans = ans \* x;  return ans;  }  }; |

**Complexity Analysis**

* Time complexity : *O*(*n*). We will multiply x for n times.
* Space complexity : *O*(1). We only need one variable to store the final product of x.

#### **Approach 2: Fast Power Algorithm Recursive**

**Intuition**

Assuming we have got the result of x ^ n, how can we get x ^ {2 \* n} ? Obviously we do not need to multiply x for another n times. Using the formula (x ^ n) ^ 2 = x ^ {2 \* n}, we can get x ^ {2 \* n} at the cost of only one computation. Using this optimization, we can reduce the time complexity of our algorithm.

**Algorithm**

Assume we have got the result of x ^ {n / 2}, and now we want to get the result of x ^ n. Let A be result of x ^ {n / 2}, we can talk about x ^ n based on the parity of n respectively. If n is even, we can use the formula (x ^ n) ^ 2 = x ^ {2 \* n} to get x ^ n = A \* A. If n is odd, then A \* A = x ^ {n - 1}. Intuitively, We need to multiply another x*x* to the result, so x ^ n = A \* A \* x. This approach can be easily implemented using recursion. We call this method "**Fast Power**", because we only need at most *O*(log*n*) computations to get x ^ n.

|  |
| --- |
| class Solution {  private double fastPow(double x, long n) {  if (n == 0) {  return 1.0;  }  double half = fastPow(x, n / 2);  if (n % 2 == 0) {  return half \* half;  } else {  return half \* half \* x;  }  }  public double myPow(double x, int n) {  long N = n;  if (N < 0) {  x = 1 / x;  N = -N;  }  return fastPow(x, N);  }  }; |

**Complexity Analysis**

* Time complexity : *O*(log*n*). Each time we apply the formula (x ^ n) ^ 2 = x ^ {2 \* n}, *n* is reduced by half. Thus we need at most *O*(log*n*) computations to get the result.
* Space complexity : *O*(log*n*). For each computation, we need to store the result of x ^ {n / 2}. We need to do the computation for *O*(log*n*) times, so the space complexity is *O*(log*n*).

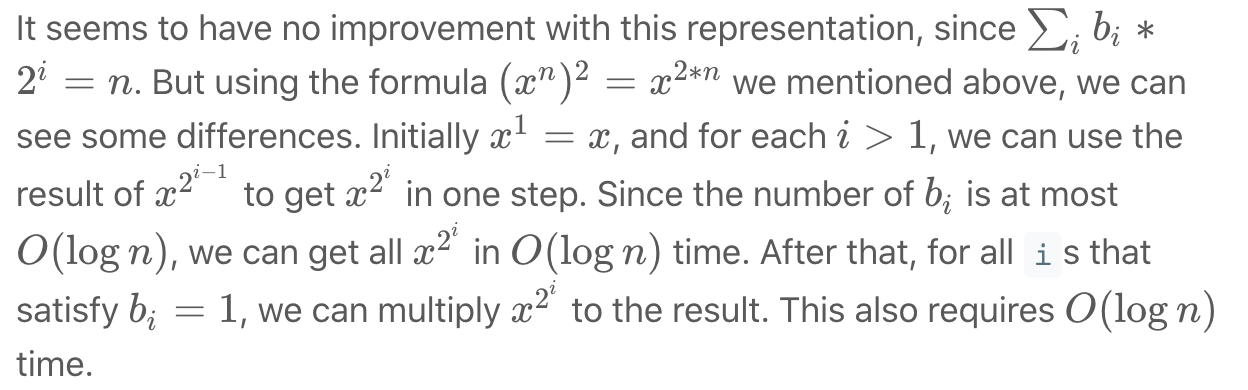
#### **Approach 3: Fast Power Algorithm Iterative**

**Intuition**

Using the formula x ^ {a + b} = x ^ a \* x ^ b*xa*+*b*=*xa*∗*xb*, we can write n as a sum of positive integers, n = \sum\_i b\_i*n*=∑*i*​*bi*​. If we can get the result of x ^ {b\_i}*xbi*​ quickly, the total time for computing x ^ n*xn* will be reduced.

**Algorithm**

We can use the binary representation of n to better understand the problem. Let the binary representation of n to be *b*1​,*b*2​,...,*blength*\_*limit*​, from the Least Significant Bit(LSB) to the Most Significant Bit(MSB). For the ith bit, if *bi*​=1, it means we need to multiply the result by x ^ {2 ^ i}.



Using fast power recursively or iteratively are actually taking different paths towards the same goal. For more information about fast power algorithm, you can visit its wiki[[1]](https://leetcode.com/problems/powx-n/solution/#fn1).

|  |
| --- |
| class Solution {  public double myPow(double x, int n) {  long N = n;  if (N < 0) {  x = 1 / x;  N = -N;  }  double ans = 1;  double current\_product = x;  for (long i = N; i > 0; i /= 2) {  if ((i % 2) == 1) {  ans = ans \* current\_product;  }  current\_product = current\_product \* current\_product;  }  return ans;  }  }; |

**Complexity Analysis**

* Time complexity : *O*(log*n*). For each bit of n 's binary representation, we will at most multiply once. So the total time complexity is *O*(log*n*).
* Space complexity : *O*(1). We only need two variables for the current product and the final result of x.

**Valid Perfect Square**

Given a **positive** integer *num*, write a function which returns True if *num* is a perfect square else False.

**Follow up:** **Do not** use any built-in library function such as sqrt.

**Example 1:**

**Input:** num = 16

**Output:** true

**Example 2:**

**Input:** num = 14

**Output:** false

**Constraints:**

* 1 <= num <= 2^31 - 1

## Solution

#### **Overview**

Square root related problems usually could be solved in logarithmic time. There are three standard logarithmic time approaches, listed here from the worst to the best:

* Recursion. The slowest one.
* Binary Search. The simplest one.
* Newton's Method. The fastest one, and therefore widely used in dynamical simulations.

The last two algorithms are interesting ones, let's discuss them in details.

These solutions have the same starting point. If one knows an [integer square](https://en.wikipedia.org/wiki/Integer_square_root) *x* of num, the answer is straightforward: num is a perfect square if  *x*∗*x*==num. Hence the problem is to compute this integer square.

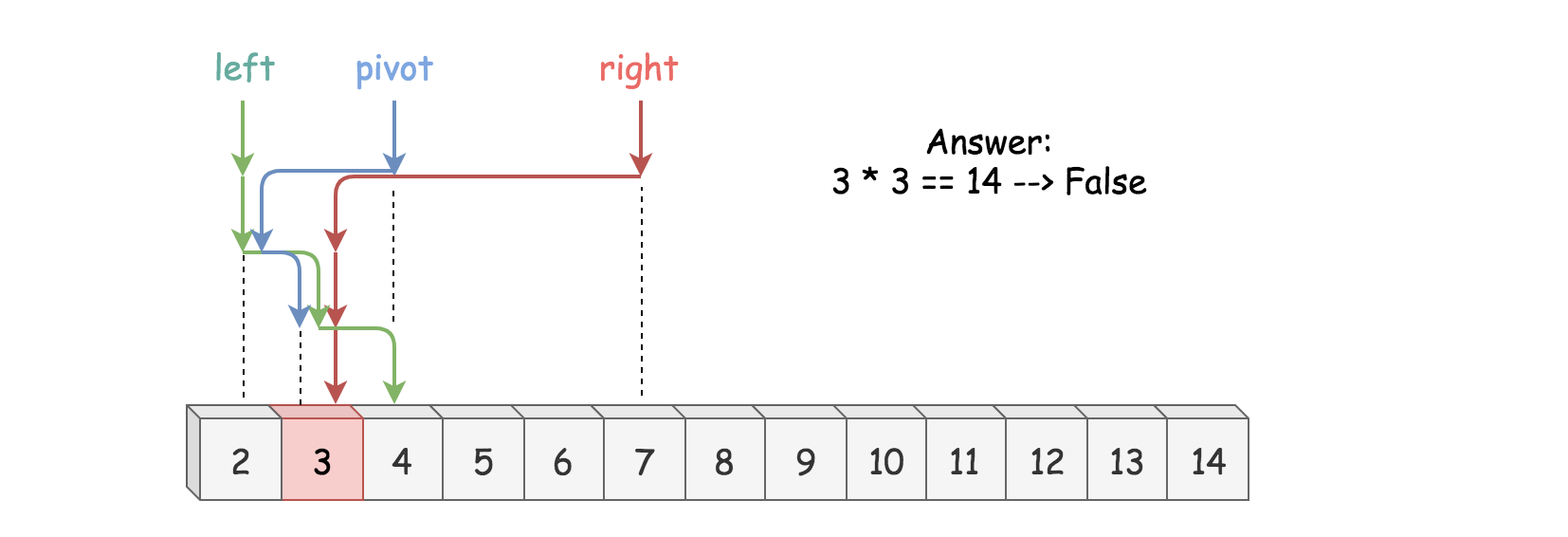
#### **Approach 1: Binary Search**

For num>2 the square root a*a* is always less than num/2 and greater than 1: 1<*x*<num/2. Since x*x* is an integer, the problem goes down to the search in the sorted set of integer numbers. Binary search is a standard way to proceed in such a situation.

**Algorithm**

* If num < 2, return True.
* Set the left boundary to 2, and the right boundary to num / 2.
* While left <= right:
  + Take x = (left + right) / 2 as a guess. Compute guess\_squared = x \* x and compare it with num:
    - If guess\_squared == num, then the perfect square is right here, return True.
    - If guess\_squared > num, move the right boundary right = x - 1.
    - Otherwise, move the left boundary left = x + 1.
* If we're here, the number is not a prefect square. Return False.

**Implementation**



|  |
| --- |
| class Solution {  public boolean isPerfectSquare(int num) {  if (num < 2) {  return true;  }  long left = 2, right = num / 2, x, guessSquared;  while (left <= right) {  x = left + (right - left) / 2;  guessSquared = x \* x;  if (guessSquared == num) {  return true;  }  if (guessSquared > num) {  right = x - 1;  } else {  left = x + 1;  }  }  return false;  }  } |

**Complexity Analysis**

* Time complexity : O(log*N*).
* Space complexity : O(1).

#### **Approach 2: Newton's Method**

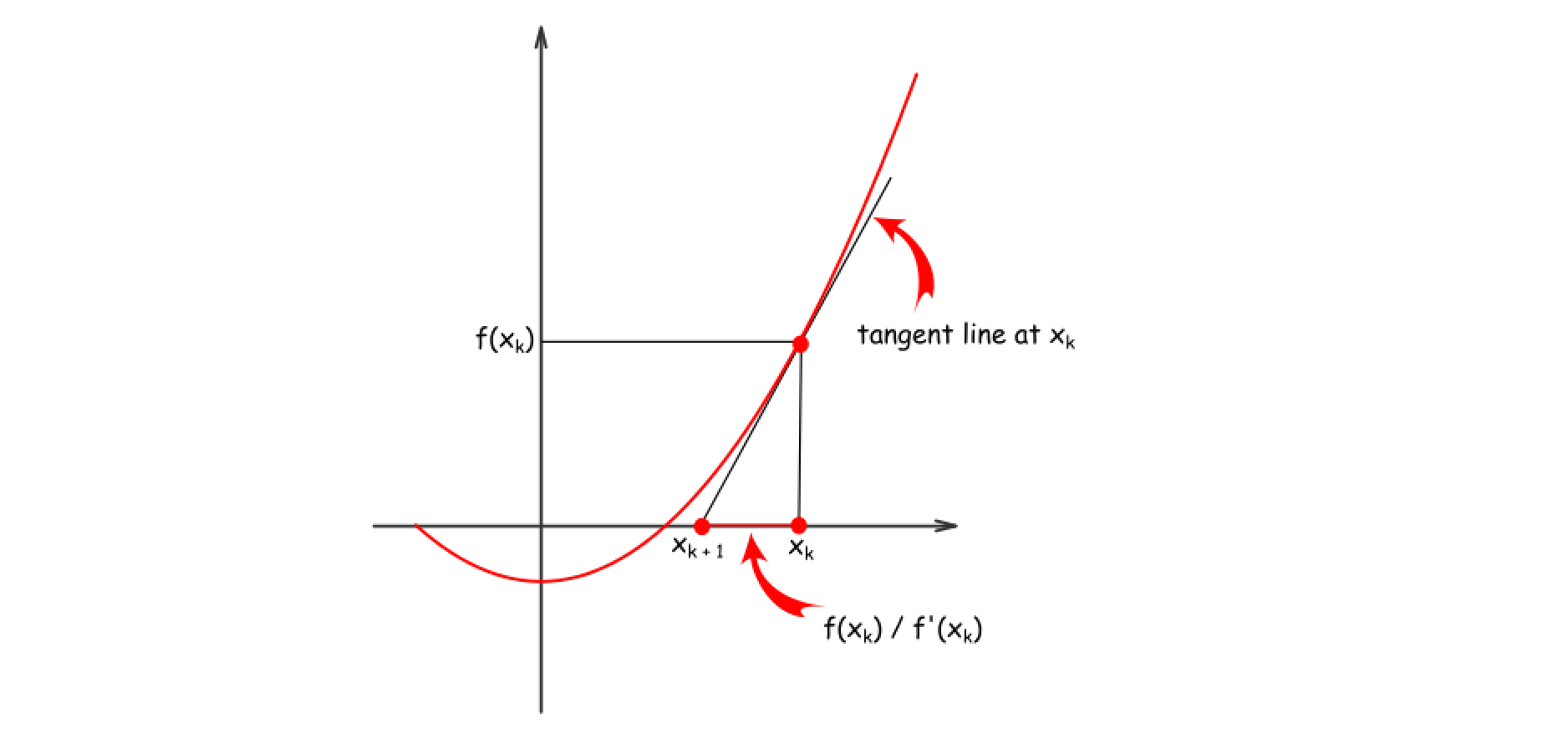
**Newton's Algorithm: How to Figure out the Formula**

Let's do a very rough derivation of Newton's sequence which could be done in two minutes during the interview. Please note that it's more a way to memorize than a strict mathematical proof.

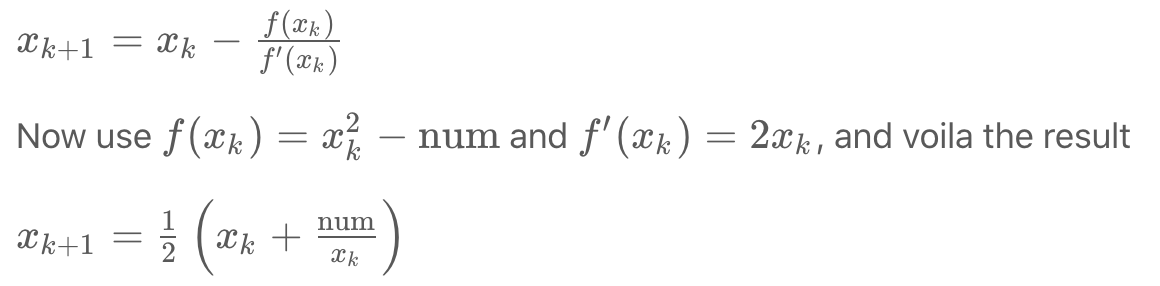
The problem is to find a root of

*f*(*x*)=*x*2−num=0

The idea of Newton's algorithm is to start from a seed (= initial guess) and then to compute a root as a sequence of improved guesses.



For example, there is a guess xk​. To compute next guess x{k + 1}​, let's approximate f(xk) by its tangent line, that would result in



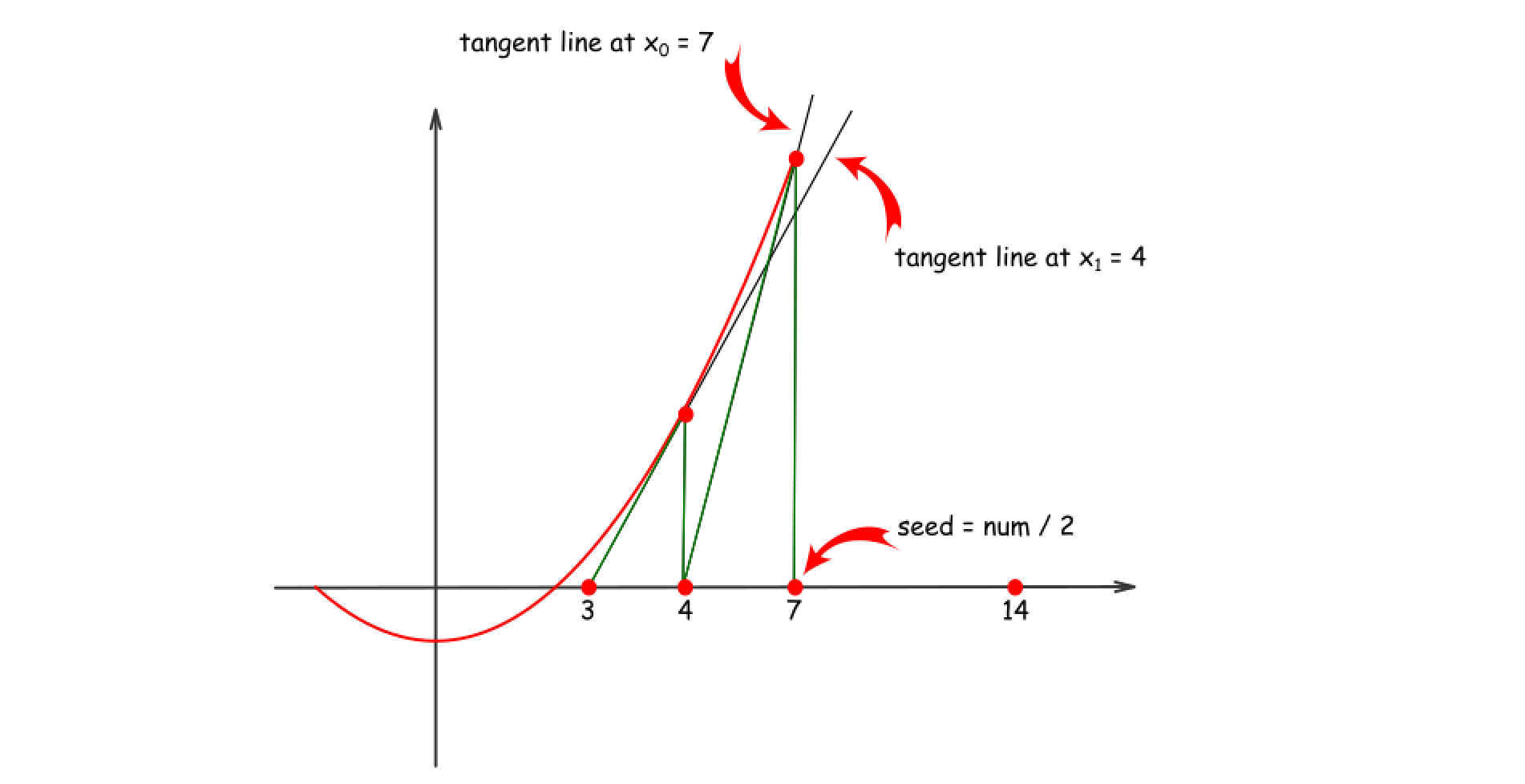
**Choose a seed**

How to choose a seed? Since the function f(x) = x^2 −num is monotonous, the smaller seed the better, so let's take num/2.

**Algorithm**

* Take num / 2 as a seed.
* While x \* x > num, compute the next guess using Newton's method: x = 
* Return x \* x == num

**Implementation**



|  |
| --- |
| class Solution {  public boolean isPerfectSquare(int num) {  if (num < 2) return true;  long x = num / 2;  while (x \* x > num) {  x = (x + num / x) / 2;  }  return (x \* x == num);  }  } |

**Complexity Analysis**

* Time complexity : O(log*N*) because [guess sequence converges quadratically](https://en.wikipedia.org/wiki/Newton%27s_method#Proof_of_quadratic_convergence_for_Newton's_iterative_method).
* Space complexity : O(1).

**Find Smallest Letter Greater Than Target**

Given a list of sorted characters letters containing only lowercase letters, and given a target letter target, find the smallest element in the list that is larger than the given target.

Letters also wrap around. For example, if the target is target = 'z' and letters = ['a', 'b'], the answer is 'a'.

**Examples:**

**Input:**

letters = ["c", "f", "j"]

target = "a"

**Output:** "c"

**Input:**

letters = ["c", "f", "j"]

target = "c"

**Output:** "f"

**Input:**

letters = ["c", "f", "j"]

target = "d"

**Output:** "f"

**Input:**

letters = ["c", "f", "j"]

target = "g"

**Output:** "j"

**Input:**

letters = ["c", "f", "j"]

target = "j"

**Output:** "c"

**Input:**

letters = ["c", "f", "j"]

target = "k"

**Output:** "c"

**Note:**

1. letters has a length in range [2, 10000].
2. letters consists of lowercase letters, and contains at least 2 unique letters.
3. target is a lowercase letter.

   Hide Hint #1

Try to find whether each of 26 next letters are in the given string array.

#### **Approach #1: Record Letters Seen [Accepted]**

**Intuition and Algorithm**

Let's scan through letters and record if we see a letter or not. We could do this with an array of size 26, or with a Set structure.

Then, for every next letter (starting with the letter that is one greater than the target), let's check if we've seen it. If we have, it must be the answer.

|  |
| --- |
| class Solution {  public char nextGreatestLetter(char[] letters, char target) {  boolean[] seen = new boolean[26];  for (char c: letters)  seen[c - 'a'] = true;  while (true) {  target++;  if (target > 'z') target = 'a';  if (seen[target - 'a']) return target;  }  }  } |

#### **Approach #2: Linear Scan [Accepted]**

**Intuition and Algorithm**

Since letters are sorted, if we see something larger when scanning form left to right, it must be the answer. Otherwise, (since letters is non-empty), the answer is letters[0].

|  |
| --- |
| class Solution {  public char nextGreatestLetter(char[] letters, char target) {  for (char c: letters)  if (c > target) return c;  return letters[0];  }  } |

**Complexity Analysis**

* Time Complexity: *O*(*N*), where *N* is the length of letters. We scan every element of the array.
* Space Complexity: *O*(1), as we maintain only pointers.

#### **Approach #3: Binary Search [Accepted]**

**Intuition and Algorithm**

Like in Approach #2, we want to find something larger than target in a sorted array. This is ideal for a binary search: Let's find the rightmost position to insert target into letters so that it remains sorted.

Our binary search (a typical one) proceeds in a number of rounds. At each round, let's maintain the loop invariant that the answer must be in the interval [lo, hi]. Let mi = (lo + hi) / 2. If letters[mi] <= target, then we must insert it in the interval [mi + 1, hi]. Otherwise, we must insert it in the interval [lo, mi].

At the end, if our insertion position says to insert target into the last position letters.length, we return letters[0] instead. This is what the modulo operation does.

|  |
| --- |
| class Solution {  public char nextGreatestLetter(char[] letters, char target) {  int lo = 0, hi = letters.length;  while (lo < hi) {  int mi = lo + (hi - lo) / 2;  if (letters[mi] <= target) lo = mi + 1;  else hi = mi;  }  return letters[lo % letters.length];  }  } |

**Complexity Analysis**

* Time Complexity: *O*(log*N*), where *N* is the length of letters. We peek only at log*N* elements in the array.
* Space Complexity: *O*(1), as we maintain only pointers.

**Find Minimum in Rotated Sorted Array**

Suppose an array of length n sorted in ascending order is **rotated** between 1 and n times. For example, the array nums = [0,1,2,4,5,6,7] might become:

* [4,5,6,7,0,1,2] if it was rotated 4 times.
* [0,1,2,4,5,6,7] if it was rotated 7 times.

Notice that **rotating** an array [a[0], a[1], a[2], ..., a[n-1]] 1 time results in the array [a[n-1], a[0], a[1], a[2], ..., a[n-2]].

Given the sorted rotated array nums, return the minimum element of this array.

**Example 1:**

**Input:** nums = [3,4,5,1,2]

**Output:** 1

**Explanation:** The original array was [1,2,3,4,5] rotated 3 times.

**Example 2:**

**Input:** nums = [4,5,6,7,0,1,2]

**Output:** 0

**Explanation:** The original array was [0,1,2,4,5,6,7] and it was rotated 4 times.

**Example 3:**

**Input:** nums = [11,13,15,17]

**Output:** 11

**Explanation:** The original array was [11,13,15,17] and it was rotated 4 times.

**Constraints:**

* n == nums.length
* 1 <= n <= 5000
* -5000 <= nums[i] <= 5000
* All the integers of nums are **unique**.
* nums is sorted and rotated between 1 and n times.

   Hide Hint #1

Array was originally in ascending order. Now that the array is rotated, there would be a point in the array where there is a small deflection from the increasing sequence. eg. The array would be something like [4, 5, 6, 7, 0, 1, 2].

   Hide Hint #2

You can divide the search space into two and see which direction to go. Can you think of an algorithm which has O(logN) search complexity?

   Hide Hint #3

1. All the elements to the left of inflection point > first element of the array.
2. All the elements to the right of inflection point < first element of the array.

## Solution

#### **Approach 1: Binary Search**

**Intuition**

A very brute way of solving this question is to search the entire array and find the minimum element. The time complexity for that would be *O*(*N*) given that N is the size of the array.

A very cool way of solving this problem is using the Binary Search algorithm. In binary search we find out the mid point and decide to either search on the left or right depending on some condition.

Since the given array is sorted, we can make use of binary search. However, the array is rotated. So simply applying the binary search won't work here.

In this question we would essentially apply a modified version of binary search where the condition that decides the search direction would be different than in a standard binary search.

We want to find the smallest element in a rotated sorted array. What if the array is not rotated? How do we check that?

If the array is not rotated and the array is in ascending order, then last element > first element.

Table

Description automatically generated

In the above example 7 > 2. This means that the array is still sorted and has no rotation.

Table

Description automatically generated

In the above example 3 < 4. Hence the array is rotated. This happens because the array was initially [2, 3 ,4 ,5 ,6 ,7]. But after the rotation the smaller elements[2,3] go at the back. i.e. [4, 5, 6, 7, 2, 3]. Because of this the first element [4] in the rotated array becomes greater than the last element.

This means there is a point in the array at which you would notice a change. This is the point which would help us in this question. We call this the Inflection Point.

A picture containing chart

Description automatically generated

In this modified version of binary search algorithm, we are looking for this point. In the above example notice the Inflection Point .

All the elements to the left of inflection point > first element of the array.  
All the elements to the right of inflection point < first element of the array.

**Algorithm**

1. Find the mid element of the array.
2. If mid element > first element of array this means that we need to look for the inflection point on the right of mid.
3. If mid element < first element of array this that we need to look for the inflection point on the left of mid.

Chart, histogram

Description automatically generated

In the above example mid element 6 is greater than first element 4. Hence we continue our search for the inflection point to the right of mid.

4 . We stop our search when we find the inflection point, when either of the two conditions is satisfied:

nums[mid] > nums[mid + 1] Hence, **mid+1** is the smallest.

nums[mid - 1] > nums[mid] Hence, **mid** is the smallest.

A picture containing table

Description automatically generated

In the above example. With the marked left and right pointers. The mid element is 2. The element just before 2 is 7 and 7>2 i.e. nums[mid - 1] > nums[mid]. Thus we have found the point of inflection and 2 is the smallest element.

|  |
| --- |
| class Solution {  public int findMin(int[] nums) {  // If the list has just one element then return that element.  if (nums.length == 1) {  return nums[0];  }  // initializing left and right pointers.  int left = 0, right = nums.length - 1;  // if the last element is greater than the first element then there is no rotation.  // e.g. 1 < 2 < 3 < 4 < 5 < 7. Already sorted array.  // Hence the smallest element is first element. A[0]  if (nums[right] > nums[0]) {  return nums[0];  }    // Binary search way  while (right >= left) {  // Find the mid element  int mid = left + (right - left) / 2;  // if the mid element is greater than its next element then mid+1 element is the smallest  // This point would be the point of change. From higher to lower value.  if (nums[mid] > nums[mid + 1]) {  return nums[mid + 1];  }  // if the mid element is lesser than its previous element then mid element is the smallest  if (nums[mid - 1] > nums[mid]) {  return nums[mid];  }  // if the mid elements value is greater than the 0th element this means  // the least value is still somewhere to the right as we are still dealing with elements  // greater than nums[0]  if (nums[mid] > nums[0]) {  left = mid + 1;  } else {  // if nums[0] is greater than the mid value then this means the smallest value is somewhere to  // the left  right = mid - 1;  }  }  return -1;  }  } |

**Complexity Analysis**

* Time Complexity : Same as Binary Search *O*(log*N*)
* Space Complexity : *O*(1)

**Find Minimum in Rotated Sorted Array II**

Suppose an array sorted in ascending order is rotated at some pivot unknown to you beforehand.

(i.e.,  [0,1,2,4,5,6,7] might become  [4,5,6,7,0,1,2]).

Find the minimum element.

The array may contain duplicates.

**Example 1:**

**Input:** [1,3,5]

**Output:** 1

**Example 2:**

**Input:** [2,2,2,0,1]

**Output:** 0

**Note:**

* This is a follow up problem to [Find Minimum in Rotated Sorted Array](https://leetcode.com/problems/find-minimum-in-rotated-sorted-array/description/).
* Would allow duplicates affect the run-time complexity? How and why?

## Solution

#### **Approach 1: Variant of Binary Search**

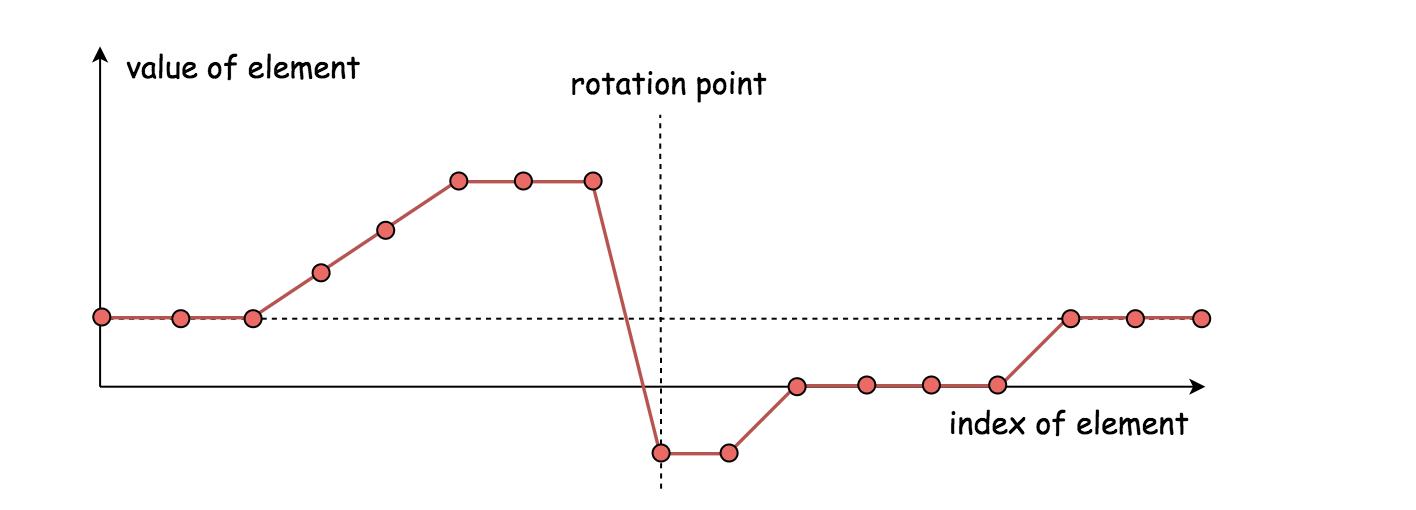
**Intuition**

Given a sorted array in ascending order (denoted as L[i]), the array is then rotated over certain unknown pivot, (denoted as L'[i]). We are asked to find the minimum value of this sorted and rotated array, which is to find the value of the first element in the original array, i.e. L[0].

The problem resembles a common problem of finding a given value from a sorted array, to which problem one could apply the **binary search** algorithm. Intuitively, one might wonder if we could apply a variant of binary search algorithm to solve our problem here.

Indeed, this is the right intuition, though the tricky part is to figure out a ***concise solution*** that could work for all cases.

To illustrate the algorithm, we draw the array in a 2D dimension in the following graph, where the X axis indicates the index of each element in the array and the Y axis indicates the value of the element.



The main structure of our algorithm remains the same as the classical binary search algorithm. As a reminder, we summarize it briefly as follows:

* We keep two pointers, i.e. low, high which point to the lowest and highest boundary of our search scope.
* We then reduce the search scope by moving either of pointers, according to various situations. Usually we shift one of pointers to the mid point between low and high, (i.e. pivot = (low+high)/2), which reduces the search scope down to half. This is also where the name of the algorithm comes from.
* The reduction of the search scope would stop, either we find the desired element or the two pointers converge (i.e. low == high).

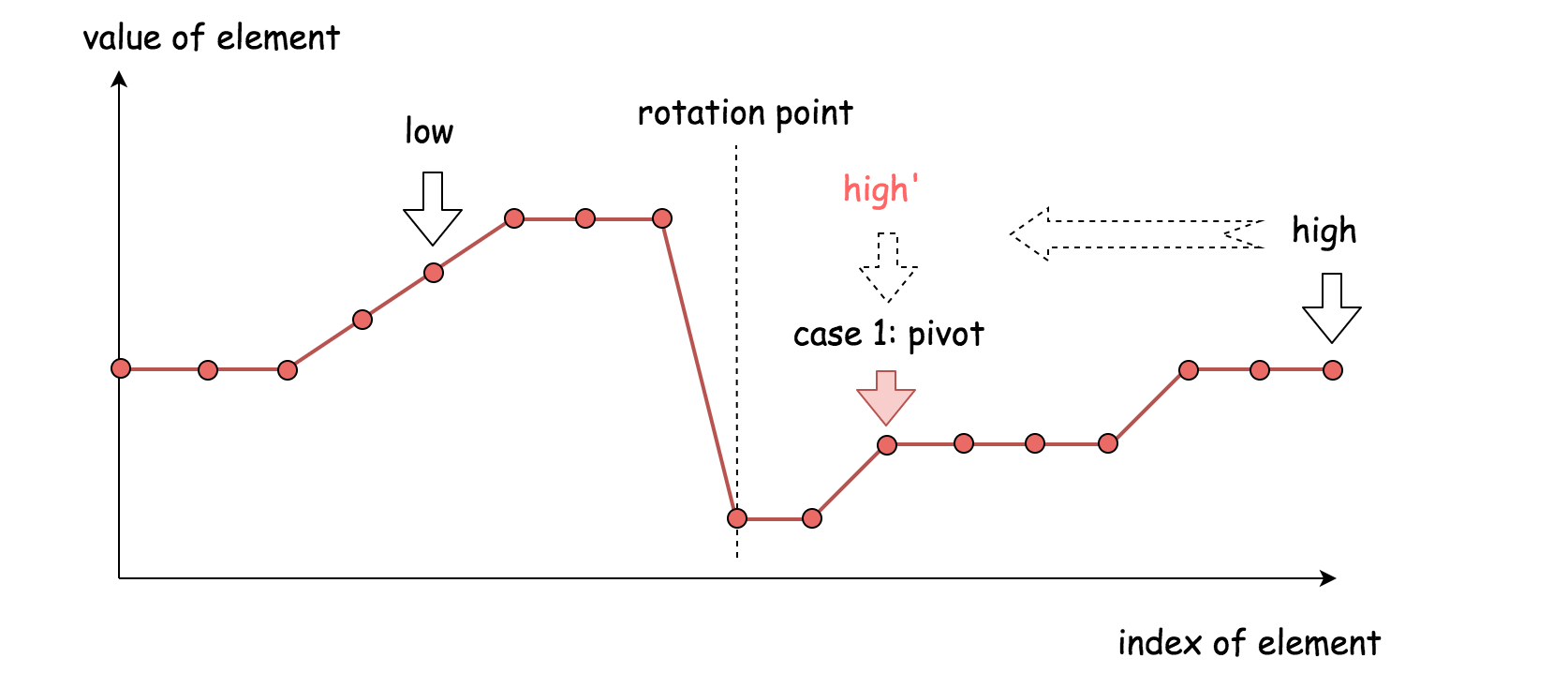
**Algorithm**

In the classical binary search algorithm, we would compare the pivot element (i.e. nums[pivot]) with the value that we would like to locate. In our case, however, we would compare the pivot element to the element pointed by the upper bound pointer (i.e. nums[high]).

Following the structure of the binary search algorithm, the essential part remained is to design the cases on how to update the two pointers.

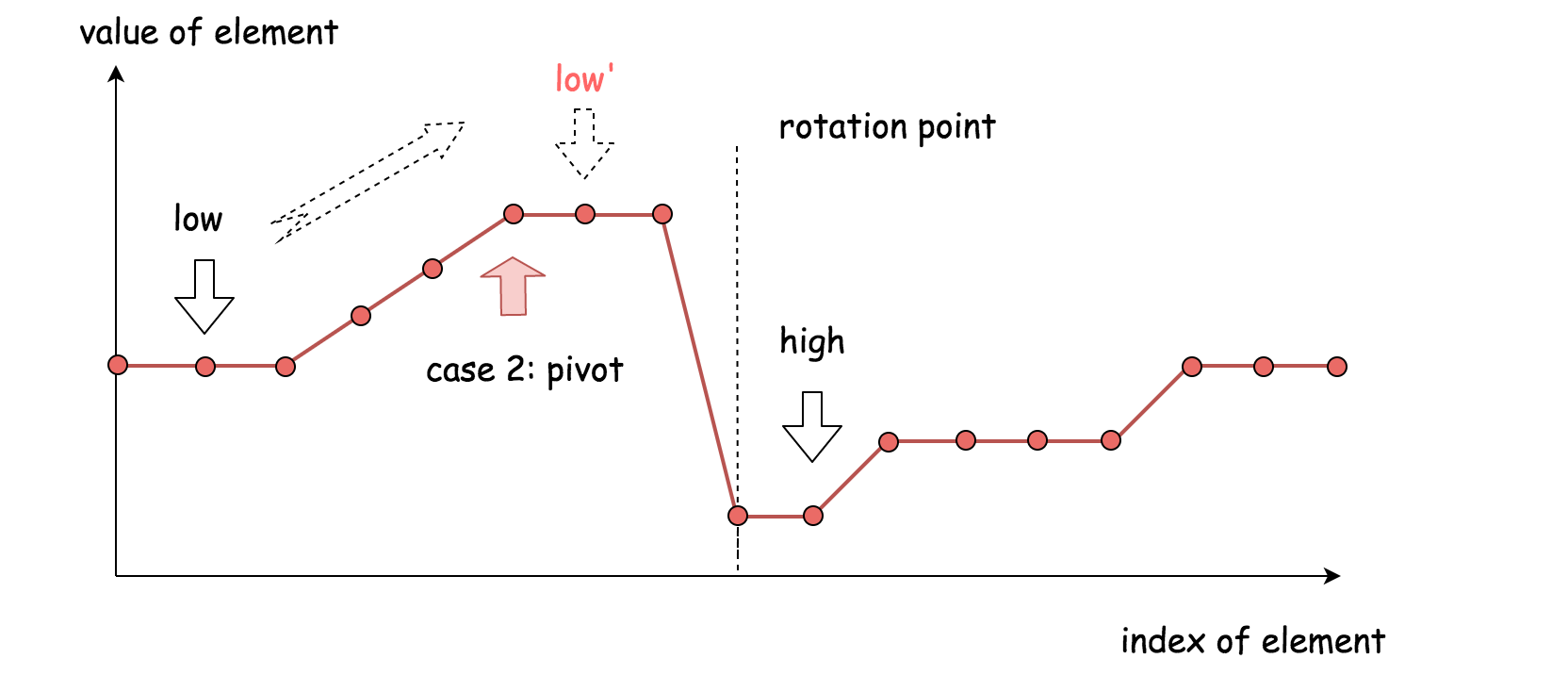
Here we give one example on how we can break it down ***concisely*** into three cases. Note that given the array, we consider the element pointed by the low index to be on the left-hand side of the array, and the element pointed by the high index to be on the right-hand side.

Case 1). nums[pivot] < nums[high]



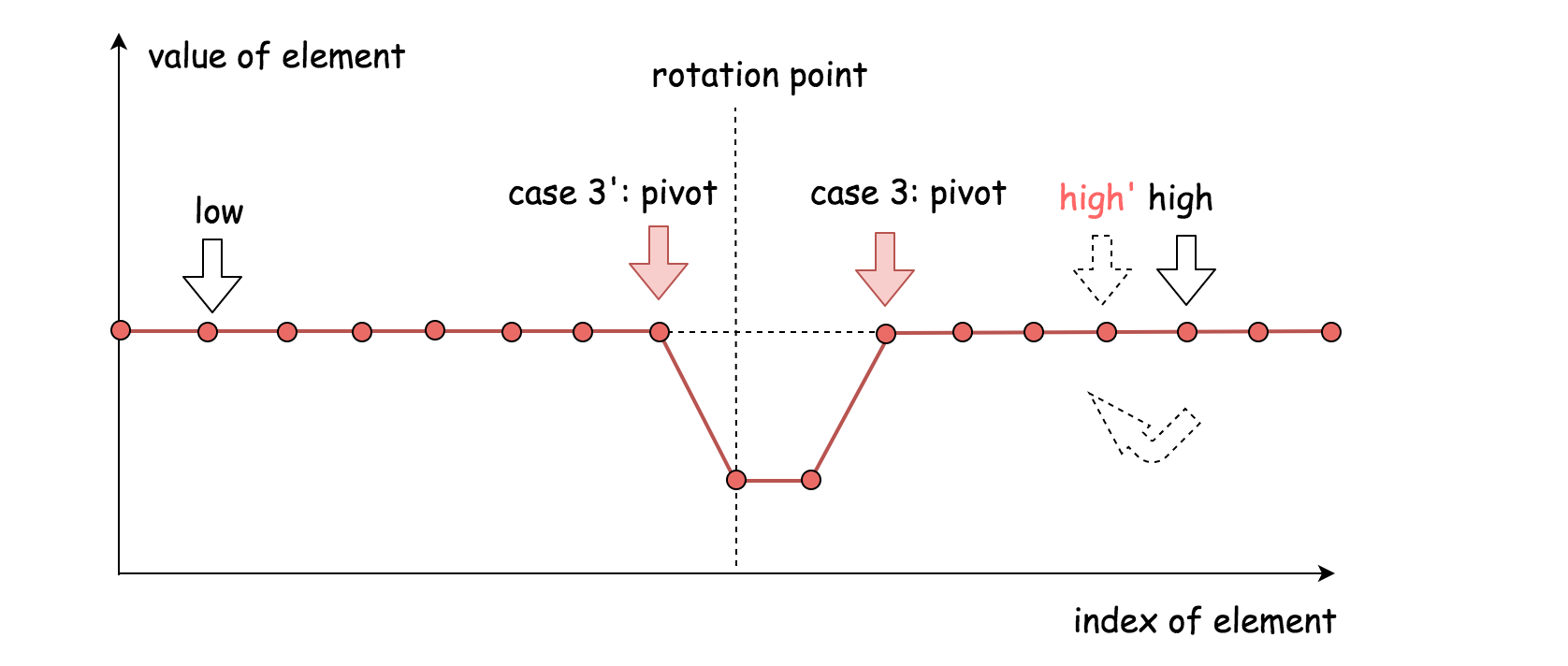
* The pivot element resides in the same half as the upper bound element.
* Therefore, the desired minimum element should reside to the **left-hand side** of pivot element. As a result, we then move the upper bound down to the pivot index, i.e. high = pivot.

Case 2). nums[pivot] > nums[high]



* The pivot element resides in the different half of array as the upper bound element.
* Therefore, the desired minium element should reside to the **right-hand side** of the pivot element. As a result, we then move the lower bound up next to the pivot index, i.e. low = pivot + 1.

Case 3). nums[pivot] == nums[high]



* In this case, we are not sure which side of the pivot that the desired minimum element would reside.
* To further reduce the search scope, a safe measure would be to reduce the upper bound by one (i.e. high = high - 1), rather than moving aggressively to the pivot point.
* The above strategy would prevent the algorithm from stagnating (i.e. endless loop). More importantly, it maintains the **correctness** of the procedure, i.e. we would not end up with skipping the desired element.

To summarize, this algorithm differs to the classical binary search algorithm in two parts:

* We use the upper bound of search scope as the reference for the comparison with the pivot element, while in the classical binary search the reference would be the desired value.
* When the result of comparison is equal (i.e. Case #3), we further move the upper bound, while in the classical binary search normally we would return the value immediately.

Here are some sample implementations based on the above algorithm. Note: the idea is inspired by the post from [sheehan](https://leetcode.com/problems/find-minimum-in-rotated-sorted-array-ii/discuss/48808/My-pretty-simple-code-to-solve-it) in the discussion forum.

|  |
| --- |
| class Solution {  public int findMin(int[] nums) {  int low = 0, high = nums.length - 1;  while (low < high) {  int pivot = low + (high - low) / 2;  if (nums[pivot] < nums[high])  high = pivot;  else if (nums[pivot] > nums[high])  low = pivot + 1;  else  high -= 1;  }  return nums[low];  }  } |

**Complexity Analysis**

* Time complexity: on average O(log2​*N*) where *N* is the length of the array, since in general it is a binary search algorithm. However, in the worst case where the array contains identical elements (i.e. case #3 nums[pivot]==nums[high]), the algorithm would deteriorate to iterating each element, as a result, the time complexity becomes O(*N*).
* Space complexity : O(1), it's a constant space solution.

**Discussion**

The problem is a follow-up to the problem of [153. Find Minimum in Rotated Sorted Array](https://leetcode.com/problems/find-minimum-in-rotated-sorted-array/). The difference is that in this problem the array can contain duplicates. So the question is "Would allow duplicates affect the run-time complexity? How and why?"

First of all, the problem of [153. Find Minimum in Rotated Sorted Array](https://leetcode.com/problems/find-minimum-in-rotated-sorted-array/) can be considered as a specific case of this problem, where it just happens that the array does not contain any duplicate. As a result, the very solutions of this problem would work for the problem of [#153](https://leetcode.com/problems/find-minimum-in-rotated-sorted-array/) as well. It is just that we would never come cross the case #3 (i.e. nums[pivot] == nums[high]) in the problem of [#153](https://leetcode.com/problems/find-minimum-in-rotated-sorted-array/).

It is due to the fact that there might exist some duplicates in the array, that we come up the case #3 which eventually render the time complexity of the algorithm to be linear \mathcal{O}(N)O(*N*), rather than \mathcal{O}(\log\_{2}{N})O(log2​*N*).

One might wonder that whether it works in case #3 if we move the lower boundary (i.e. low += 1), rather than the upper boundary (i.e. high -= 1).

The short answer is that it could work for some cases, but not for all. For instance, given the input [1, 3, 3], by moving the lower boundary, we would skip the correct answer.

While we do low = pivot + 1 to reduce the search scope, then why not do high = pivot - 1 instead of high = pivot? Or a similar question would be "why don't we do check of *low <= high* rather than *low < high*"?

As a matter of fact, the binary search algorithm has several [forms of implementation](https://en.wikipedia.org/wiki/Binary_search_algorithm), regarding how we set the boundaries and the loop conditions. One can refer to the [Explore card of Binary Search](https://leetcode.com/explore/learn/card/binary-search/) in LeetCode for more details. As simple as the idea of binary search might seem to be, it is tricky to make it work for all cases.

As one would discover from the card, the above implementation of binary search complies with the [template II](https://leetcode.com/explore/learn/card/binary-search/126/template-ii/937/) of binary search. And by replacing high = pivot with high = pivot - 1, the algorithm will not work.

As subtle as it looks like, the update of the pointers should be consistent with the conditions of the loop. As a rule of thumb, it is advised to stick with one form of binary search, and not to mix them up.

One might notice that we are calculating the pivot with the formula of pivot = low + (high-low)/2, rather than the more intuitive term pivot = (high+low)/2.

Actually, this is done intentionally to prevent the numeric overflow issue, since the sum of two integers could exceed the limit of the integer number. As a fun fact, the above mistake prevails in many implementations of binary search, as revealed from a post titled ["Nearly All Binary Searches and Mergesorts are Broken"](https://ai.googleblog.com/2006/06/extra-extra-read-all-about-it-nearly.html) from googleblog in 2006.

**Intersection of Two Arrays**

Given two arrays, write a function to compute their intersection.

**Example 1:**

**Input:** nums1 = [1,2,2,1], nums2 = [2,2]

**Output:** [2]

**Example 2:**

**Input:** nums1 = [4,9,5], nums2 = [9,4,9,8,4]

**Output:** [9,4]

**Note:**

* Each element in the result must be unique.
* The result can be in any order.

## Solution

#### **Approach 1: Two Sets**

**Intuition**

The naive approach would be to iterate along the first array nums1 and to check for each value if this value in nums2 or not. If yes - add the value to output. Such an approach would result in a pretty bad O(*n*×*m*) time complexity, where n and m are arrays' lengths.

To solve the problem in linear time, let's use the structure set, which provides in/contains operation in O(1) time in average case.

The idea is to convert both arrays into sets, and then iterate over the smallest set checking the presence of each element in the larger set. Time complexity of this approach is O(*n*+*m*) in the average case.

Background pattern

Description automatically generated with low confidence

A picture containing chart

Description automatically generated

A picture containing diagram

Description automatically generated

A picture containing diagram

Description automatically generated

A picture containing diagram

Description automatically generated

A picture containing diagram

Description automatically generated

|  |
| --- |
| class Solution {  public int[] set\_intersection(HashSet<Integer> set1, HashSet<Integer> set2) {  int [] output = new int[set1.size()];  int idx = 0;  for (Integer s : set1)  if (set2.contains(s)) output[idx++] = s;  return Arrays.copyOf(output, idx);  }  public int[] intersection(int[] nums1, int[] nums2) {  HashSet<Integer> set1 = new HashSet<Integer>();  for (Integer n : nums1) set1.add(n);  HashSet<Integer> set2 = new HashSet<Integer>();  for (Integer n : nums2) set2.add(n);  if (set1.size() < set2.size()) return set\_intersection(set1, set2);  else return set\_intersection(set2, set1);  }  } |

**Complexity Analysis**

* Time complexity : O(*n*+*m*), where n and m are arrays' lengths. O(*n*) time is used to convert nums1 into set, O(*m*) time is used to convert nums2, and contains/in operations are O(1) in the average case.
* Space complexity : O(*m*+*n*) in the worst case when all elements in the arrays are different.

#### **Approach 2: Built-in Set Intersection**

**Intuition**

There are built-in intersection facilities, which provide O(*n*+*m*) time complexity in the average case and O(*n*×*m*) time complexity in the worst case.

In Python it's [intersection operator](https://wiki.python.org/moin/TimeComplexity#set), in Java - [retainAll() function](https://docs.oracle.com/javase/8/docs/api/java/util/AbstractCollection.html#retainAll-java.util.Collection-).

**Implementation**

|  |
| --- |
| class Solution {  public int[] intersection(int[] nums1, int[] nums2) {  HashSet<Integer> set1 = new HashSet<Integer>();  for (Integer n : nums1) set1.add(n);  HashSet<Integer> set2 = new HashSet<Integer>();  for (Integer n : nums2) set2.add(n);  set1.retainAll(set2);  int [] output = new int[set1.size()];  int idx = 0;  for (int s : set1) output[idx++] = s;  return output;  }  } |

**Complexity Analysis**

* Time complexity : O(*n*+*m*) in the average case and O(*n*×*m*) [in the worst case when load factor is high enough](https://wiki.python.org/moin/TimeComplexity#set).
* Space complexity : O(*n*+*m*) in the worst case when all elements in the arrays are different.

**Intersection of Two Arrays II**

Given two arrays, write a function to compute their intersection.

**Example 1:**

**Input:** nums1 = [1,2,2,1], nums2 = [2,2]

**Output:** [2,2]

**Example 2:**

**Input:** nums1 = [4,9,5], nums2 = [9,4,9,8,4]

**Output:** [4,9]

**Note:**

* Each element in the result should appear as many times as it shows in both arrays.
* The result can be in any order.

**Follow up:**

* What if the given array is already sorted? How would you optimize your algorithm?
* What if *nums1*'s size is small compared to *nums2*'s size? Which algorithm is better?
* What if elements of *nums2* are stored on disk, and the memory is limited such that you cannot load all elements into the memory at once?

## Solution

If an interviewer gives you this problem, your first question should be - how should I handle duplicates? Your second question, perhaps, can be about the order of inputs and outputs. Such questions manifest your problem-solving skills, and help you steer to the right solution.

The [solution](https://leetcode.com/problems/intersection-of-two-arrays/solution/) for the previous problem, [349. Intersection of Two Arrays](https://leetcode.com/problems/intersection-of-two-arrays/), talks about approaches when each number in the output must be unique. For this problem, we need to adapt those approaches so that numbers in the result appear as many times as they do in both arrays.

#### **Approach 1: Hash Map**

For the previous problem, we used a hash set to achieve a linear time complexity. Here, we need to use a hash map to track the count for each number.

We collect numbers and their counts from one of the arrays into a hash map. Then, we iterate along the second array, and check if the number exists in the hash map and its count is positive. If so - add the number to the result and decrease its count in the hash map.



It's a good idea to check array sizes and use a hash map for the smaller array. It will reduce memory usage when one of the arrays is very large.

**Algorithm**

1. If nums1 is larger than nums2, swap the arrays.
2. For each element in nums1:
   * Add it to the hash map m.
     + Increment the count if the element is already there.
3. Initialize the insertion pointer (k) with zero.
4. Iterate along nums2:
   * If the current number is in the hash map and count is positive:
     + Copy the number into nums1[k], and increment k.
     + Decrement the count in the hash map.
5. Return first k elements of nums1.

For our solutions here, we use one of the arrays to store the result. As we find common numbers, we copy them to the first array starting from the beginning. This idea is from [this solution](https://leetcode.com/problems/intersection-of-two-arrays-ii/discuss/82405/Simple-Java-Solution) by [sankitgupta](https://leetcode.com/sankitgupta/).

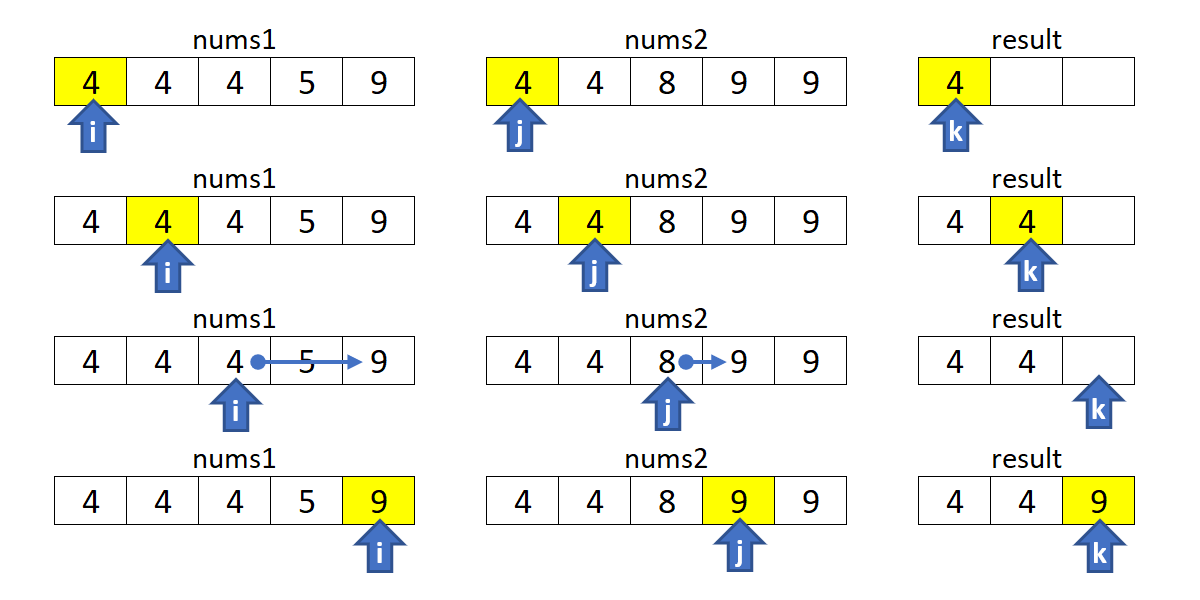
|  |
| --- |
| public int[] intersect(int[] nums1, int[] nums2) {  if (nums1.length > nums2.length) {  return intersect(nums2, nums1);  }  HashMap<Integer, Integer> m = new HashMap<>();  for (int n : nums1) {  m.put(n, m.getOrDefault(n, 0) + 1);  }  int k = 0;  for (int n : nums2) {  int cnt = m.getOrDefault(n, 0);  if (cnt > 0) {  nums1[k++] = n;  m.put(n, cnt - 1);  }  }  return Arrays.copyOfRange(nums1, 0, k);  } |

**Complexity Analysis**

* Time Complexity: O(*n*+*m*), where *n* and m*m* are the lengths of the arrays. We iterate through the first, and then through the second array; insert and lookup operations in the hash map take a constant time.
* Space Complexity: O(min(*n*,*m*)). We use hash map to store numbers (and their counts) from the smaller array.

#### **Approach 2: Sort**

You can recommend this method when the input is sorted, or when the output needs to be sorted. Here, we sort both arrays (assuming they are not sorted) and use two pointers to find common numbers in a single scan.



**Algorithm**

1. Sort nums1 and nums2.
2. Initialize i, j and k with zero.
3. Move indices i along nums1, and j through nums2:
   * Increment i if nums1[i] is smaller.
   * Increment j if nums2[j] is smaller.
   * If numbers are the same, copy the number into nums1[k], and increment i, j and k.
4. Return first k elements of nums1.

|  |
| --- |
| public int[] intersect(int[] nums1, int[] nums2) {  Arrays.sort(nums1);  Arrays.sort(nums2);  int i = 0, j = 0, k = 0;  while (i < nums1.length && j < nums2.length) {  if (nums1[i] < nums2[j]) {  ++i;  } else if (nums1[i] > nums2[j]) {  ++j;  } else {  nums1[k++] = nums1[i++];  ++j;  }  }  return Arrays.copyOfRange(nums1, 0, k);  } |

**Complexity Analysis**

* Time Complexity: O(*n*log*n*+*m*log*m*), where n*n* and m*m* are the lengths of the arrays. We sort two arrays independently, and then do a linear scan.
* Space Complexity: from O(log*n*+log*m*) to O(*n*+*m*), depending on the implementation of the sorting algorithm. For the complexity analysis purposes, we ignore the memory required by inputs and outputs.

#### **Approach 3: Built-in Intersection**

This is similar to [Approach 2](https://leetcode.com/problems/intersection-of-two-arrays-ii/solution/#approach-2-sort). Instead of iterating with two pointers, we use a built-in function to find common elements. In C++, we can use set\_intersection for sorted arrays (or multisets).

The [retainAll](https://docs.oracle.com/javase/8/docs/api/java/util/AbstractCollection.html#retainAll-java.util.Collection-) method in Java, unfortunately, does not care how many times an element occurs in the other collection. You can use the [retainOccurrences](https://guava.dev/releases/23.0/api/docs/com/google/common/collect/Multisets.html#retainOccurrences-com.google.common.collect.Multiset-com.google.common.collect.Multiset-) method of the multiset implementation in [Guava](https://guava.dev/releases/16.0/api/docs/com/google/common/collect/Multiset.html).

**Algorithm**

Note that set\_intersection returns the position past the end of the produced range, so it can be used as an input for the erase function. The idea is from [this solution](https://leetcode.com/problems/intersection-of-two-arrays-ii/discuss/82269/Short-Python-C%2B%2B) by [StefanPochmann](https://leetcode.com/stefanpochmann/).

|  |
| --- |
| vector<int> intersect(vector<int>& nums1, vector<int>& nums2) {  sort(begin(nums1), end(nums1));  sort(begin(nums2), end(nums2));  nums1.erase(set\_intersection(begin(nums1), end(nums1),  begin(nums2), end(nums2), begin(nums1)), end(nums1));  return nums1;  } |

**Complexity Analysis**

* Same as for [approach 2](https://leetcode.com/problems/intersection-of-two-arrays-ii/solution/#approach2complexity) above.

#### **Follow-up Questions**

1. What if the given array is already sorted? How would you optimize your algorithm?
   * We can use either [Approach 2](https://leetcode.com/problems/intersection-of-two-arrays-ii/solution/#approach-2-sort) or [Approach 3](https://leetcode.com/problems/intersection-of-two-arrays-ii/solution/#approach-3-built-in-intersection), dropping the sort of course. It will give us linear time and constant memory complexity.
2. What if nums1's size is small compared to nums2's size? Which algorithm is better?
   * [Approach 1](https://leetcode.com/problems/intersection-of-two-arrays-ii/solution/#approach-1-hash-map) is a good choice here as we use a hash map for the smaller array.
3. What if elements of nums2 are stored on disk, and the memory is limited such that you cannot load all elements into the memory at once?
   * If nums1 fits into the memory, we can use [Approach 1](https://leetcode.com/problems/intersection-of-two-arrays-ii/solution/#approach-1-hash-map) to collect counts for nums1 into a hash map. Then, we can sequentially load and process nums2.
   * If neither of the arrays fit into the memory, we can apply some partial processing strategies:
     + Split the numeric range into subranges that fits into the memory. Modify [Approach 1](https://leetcode.com/problems/intersection-of-two-arrays-ii/solution/#approach-1-hash-map) to collect counts only within a given subrange, and call the method multiple times (for each subrange).
     + Use an external sort for both arrays. Modify [Approach 2](https://leetcode.com/problems/intersection-of-two-arrays-ii/solution/#approach-2-sort) to load and process arrays sequentially.

**Two Sum II - Input array is sorted**

Given an array of integers numbers that is already **sorted in ascending order**, find two numbers such that they add up to a specific target number.

Return the indices of the two numbers (**1-indexed**) as an integer array answer of size 2, where 1 <= answer[0] < answer[1] <= numbers.length.

You may assume that each input would have **exactly one solution** and you **may not** use the same element twice.

**Example 1:**

**Input:** numbers = [2,7,11,15], target = 9

**Output:** [1,2]

**Explanation:** The sum of 2 and 7 is 9. Therefore index1 = 1, index2 = 2.

**Example 2:**

**Input:** numbers = [2,3,4], target = 6

**Output:** [1,3]

**Example 3:**

**Input:** numbers = [-1,0], target = -1

**Output:** [1,2]

**Constraints:**

* 2 <= numbers.length <= 3 \* 104
* -1000 <= numbers[i] <= 1000
* numbers is sorted in **increasing order**.
* -1000 <= target <= 1000
* **Only one valid answer exists.**

## 

## Solution

#### **Approach 1: Two Pointers**

**Algorithm**

We can apply [Two Sum's solutions](https://leetcode.com/articles/two-sum/) directly to get O(n^2) time, *O*(1) space using brute force and *O*(*n*) time, *O*(*n*) space using hash table. However, both existing solutions do not make use of the property where the input array is sorted. We can do better.

We use two indexes, initially pointing to the first and last element respectively. Compare the sum of these two elements with target. If the sum is equal to target, we found the exactly only solution. If it is less than target, we increase the smaller index by one. If it is greater than target, we decrease the larger index by one. Move the indexes and repeat the comparison until the solution is found.

Let [...,*a*,*b*,*c*,...,*d*,*e*,*f*,...] be the input array that is sorted in ascending order and the element *b*, *e* be the exactly only solution. Because we are moving the smaller index from left to right, and the larger index from right to left, at some point one of the indexes must reach either one of *b* or *e*. Without loss of generality, suppose the smaller index reaches *b* first. At this time, the sum of these two elements must be greater than target. Based on our algorithm, we will keep moving the larger index to its left until we reach the solution.

|  |
| --- |
| class Solution {  public:  vector<int> twoSum(vector<int>& numbers, int target) {  int low = 0, high = numbers.size() - 1;  while (low < high) {  int sum = numbers[low] + numbers[high];  if (sum == target)  return {low + 1, high + 1};  else if (sum < target)  ++low;  else  --high;  }  return {-1, -1};  }  }; |

Do we need to consider if *numbers*[*low*]+*numbers*[*high*] overflows? The answer is no. Even if adding two elements in the array may overflow, because there is exactly one solution, we will reach the solution first.

**Complexity analysis**

* Time complexity : *O*(*n*). Each of the *n* elements is visited at most once, thus the time complexity is *O*(*n*).
* Space complexity :*O*(1). We only use two indexes, the space complexity is *O*(1).

**Find the Duplicate Number**

Given an array of integers nums containing n + 1 integers where each integer is in the range [1, n] inclusive.

There is only **one repeated number** in nums, return *this repeated number*.

**Example 1:**

**Input:** nums = [1,3,4,2,2]

**Output:** 2

**Example 2:**

**Input:** nums = [3,1,3,4,2]

**Output:** 3

**Example 3:**

**Input:** nums = [1,1]

**Output:** 1

**Example 4:**

**Input:** nums = [1,1,2]

**Output:** 1

#### **Note**

The first two approaches mentioned do not satisfy the constraints given in the prompt, but they are solutions that you might be likely to come up with during a technical interview. As an interviewer, I personally would not expect someone to come up with the cycle detection solution unless they have heard it before.

#### **Proof**

Proving that at least one duplicate must exist in nums is simple application of the [pigeonhole principle](https://en.wikipedia.org/wiki/Pigeonhole_principle). Here, each number in nums is a "pigeon" and each distinct number that can appear in nums is a "pigeonhole". Because there are *n*+1 numbers are *n* distinct possible numbers, the pigeonhole principle implies that at least one of the numbers is duplicated.

#### **Approach 1: Sorting**

**Intuition**

If the numbers are sorted, then any duplicate numbers will be adjacent in the sorted array.

**Algorithm**

Given the intuition, the algorithm follows fairly simply. First, we sort the array, and then we compare each element to the previous element. Because there is exactly one duplicated element in the array, we know that the array is of at least length 2, and we can return the duplicate element as soon as we find it.

|  |
| --- |
| class Solution {  public int findDuplicate(int[] nums) {  Arrays.sort(nums);  for (int i = 1; i < nums.length; i++) {  if (nums[i] == nums[i-1]) {  return nums[i];  }  }  return -1;  }  } |

**Complexity Analysis**

* Time complexity : O(*nlgn*)

The sort invocation costs O(*nlgn*) time in Python and Java, so it dominates the subsequent linear scan.

* Space complexity : O(1) (or O(*n*))

Here, we sort nums in place, so the memory footprint is constant. If we cannot modify the input array, then we must allocate linear space for a copy of nums and sort that instead.

#### **Approach 2: Set**

**Intuition**

If we store each element as we iterate over the array, we can simply check each element as we iterate over the array.

**Algorithm**

In order to achieve linear time complexity, we need to be able to insert elements into a data structure (and look them up) in constant time. A Set satisfies these constraints nicely, so we iterate over the array and insert each element into seen. Before inserting it, we check whether it is already there. If it is, then we found our duplicate, so we return it.

|  |
| --- |
| class Solution {  public int findDuplicate(int[] nums) {  Set<Integer> seen = new HashSet<Integer>();  for (int num : nums) {  if (seen.contains(num)) {  return num;  }  seen.add(num);  }  return -1;  }  } |

**Complexity Analysis**

* Time complexity : O(*n*)

Set in both Python and Java rely on underlying hash tables, so insertion and lookup have amortized constant time complexities. The algorithm is therefore linear, as it consists of a for loop that performs constant work *n* times.

* Space complexity : O(*n*)

In the worst case, the duplicate element appears twice, with one of its appearances at array index *n*−1. In this case, seen will contain *n*−1 distinct values, and will therefore occupy O(*n*) space.

#### **Approach 3: Floyd's Tortoise and Hare (Cycle Detection)**

**Intuition**

The idea is to reduce the problem to [Linked List Cycle II](https://leetcode.com/problems/linked-list-cycle-ii/solution/):

Given a linked list, return the node where the cycle begins.

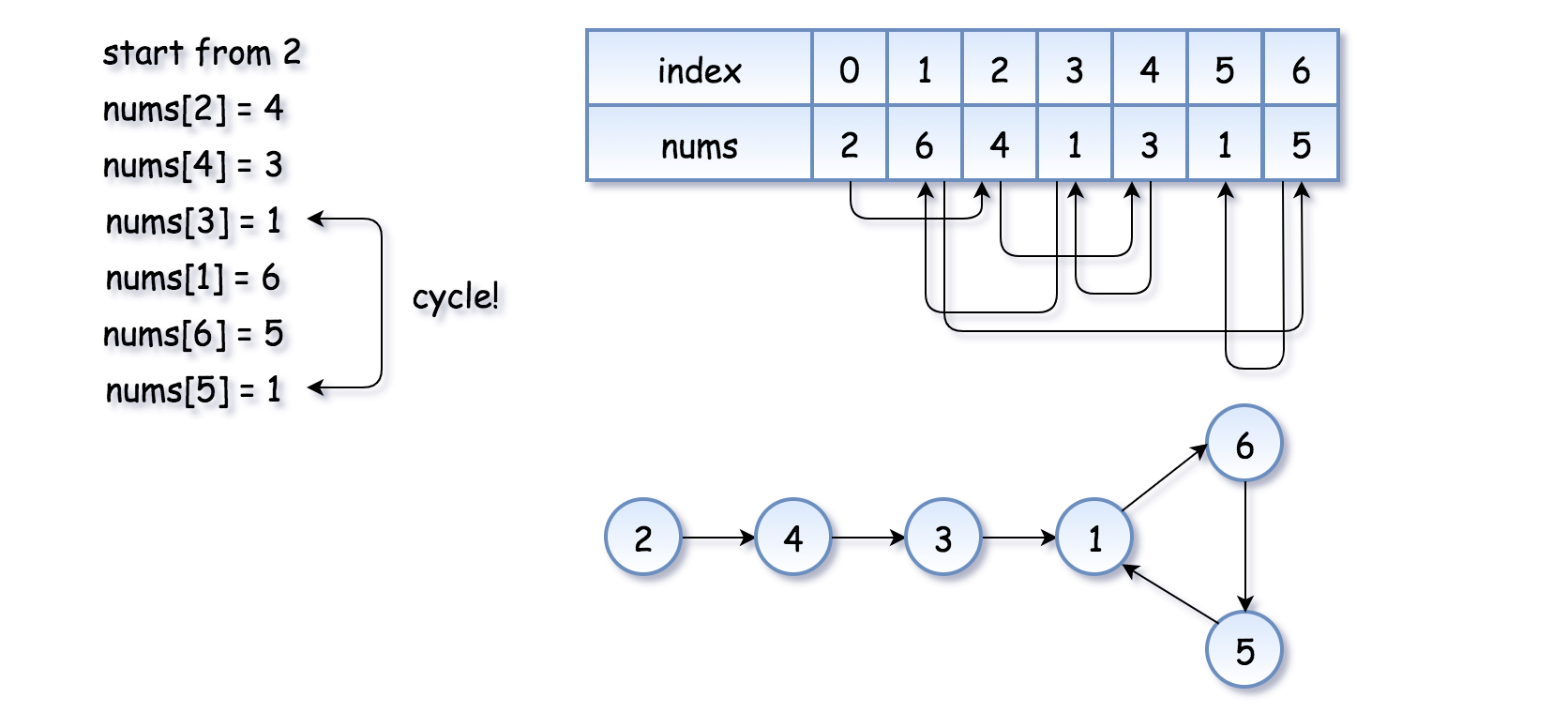
First of all, where does the cycle come from? Let's use the function f(x) = nums[x] to construct the sequence: x, nums[x], nums[nums[x]], nums[nums[nums[x]]], ....

Each new element in the sequence is an element in nums at the index of the previous element.

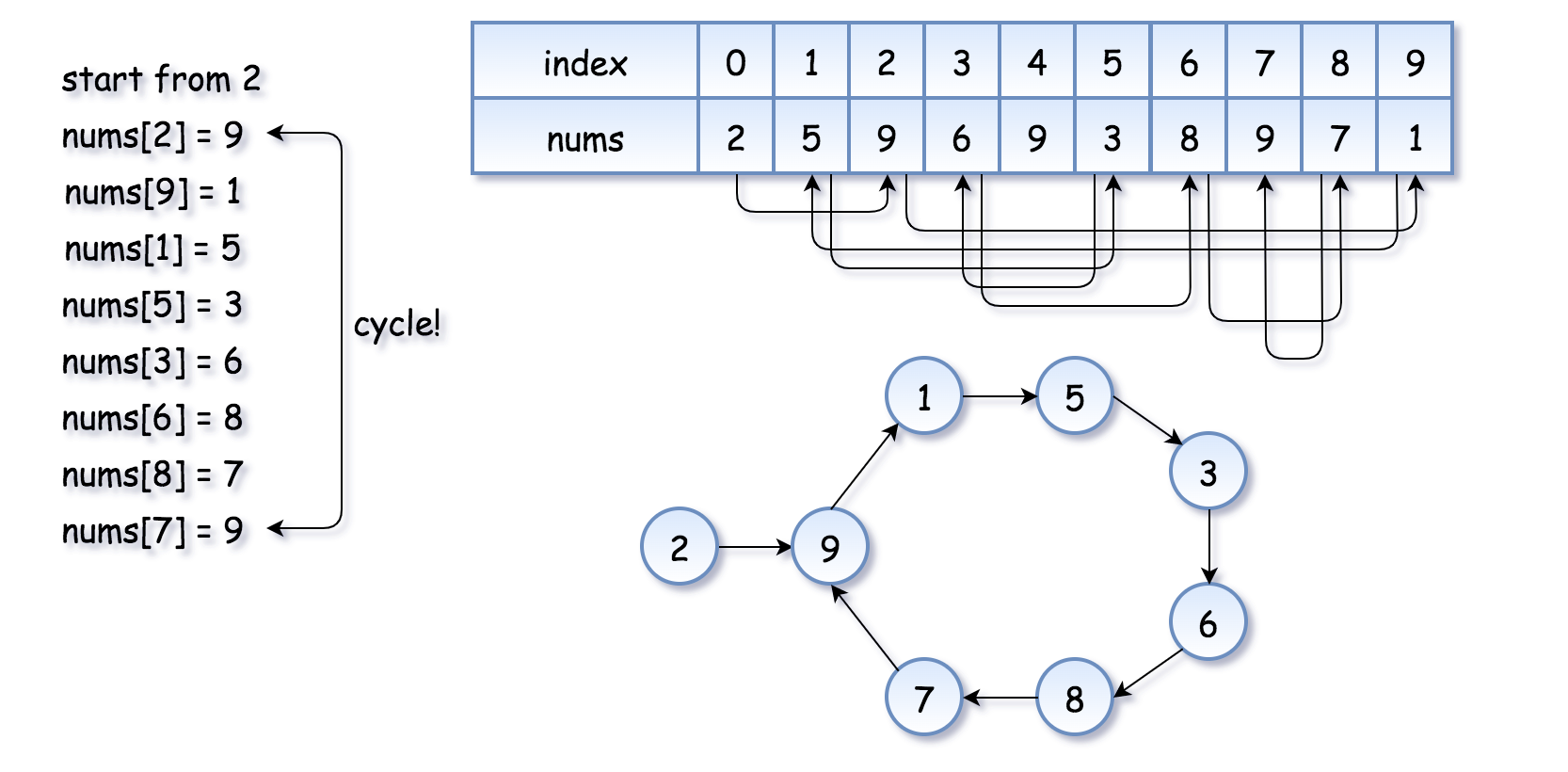
If one starts from x = nums[0], such a sequence will produce a linked list with a cycle.

The cycle appears because nums contains duplicates. The duplicate node is a cycle entrance.

Here is how it works:



The example above is simple because the loop is small. Here is a more interesting example (special thanks to @[sushant\_chaudhari](https://leetcode.com/sushant_chaudhari))



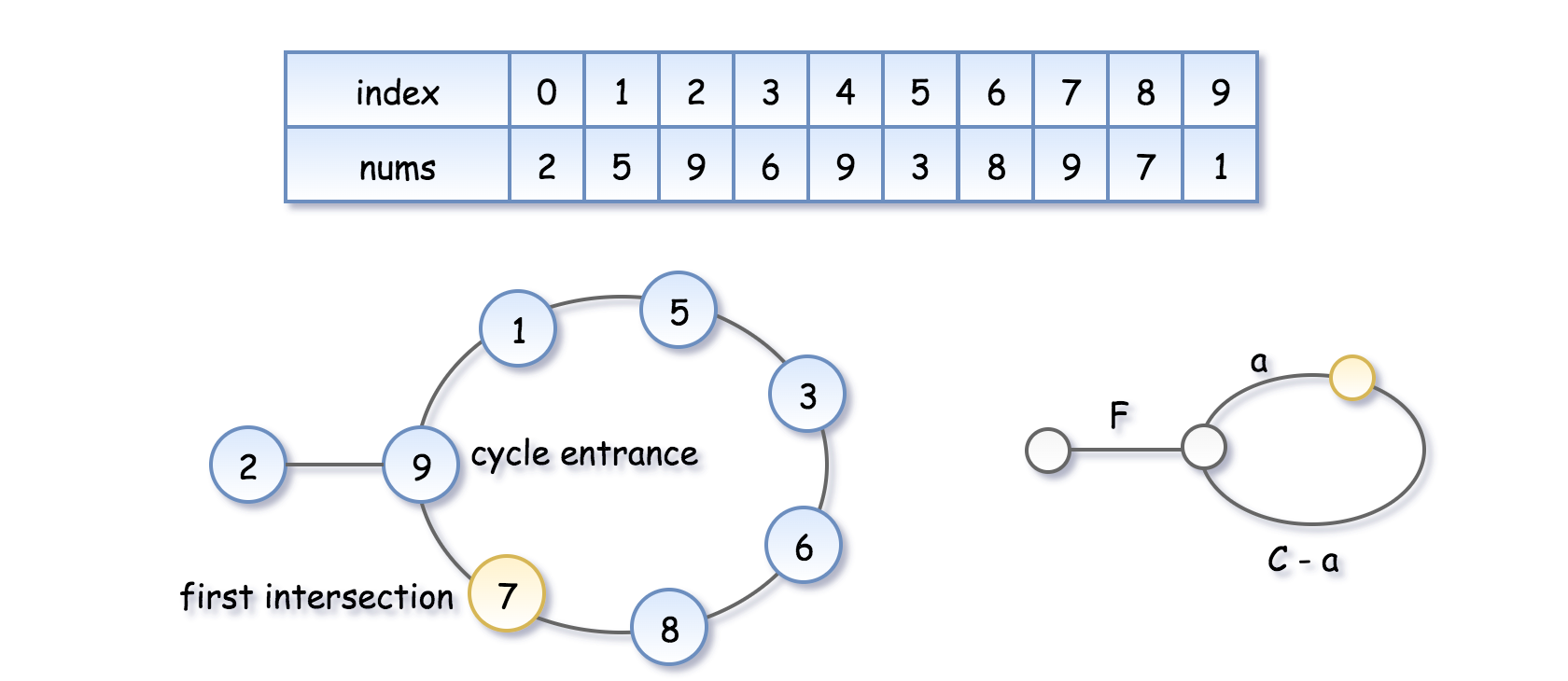
Now the problem is to find the entrance of the cycle.

**Algorithm**

[Floyd's algorithm](https://en.wikipedia.org/wiki/The_Tortoise_and_the_Hare) consists of two phases and uses two pointers, usually called tortoise and hare.

**In phase 1**, hare = nums[nums[hare]] is twice as fast as tortoise = nums[tortoise]. Since the hare goes fast, it would be the first one who enters the cycle and starts to run around the cycle. At some point, the tortoise enters the cycle as well, and since it's moving slower the hare catches the tortoise up at some intersection point. Now phase 1 is over, and the tortoise has lost.

Note that the intersection point is not the cycle entrance in the general case.

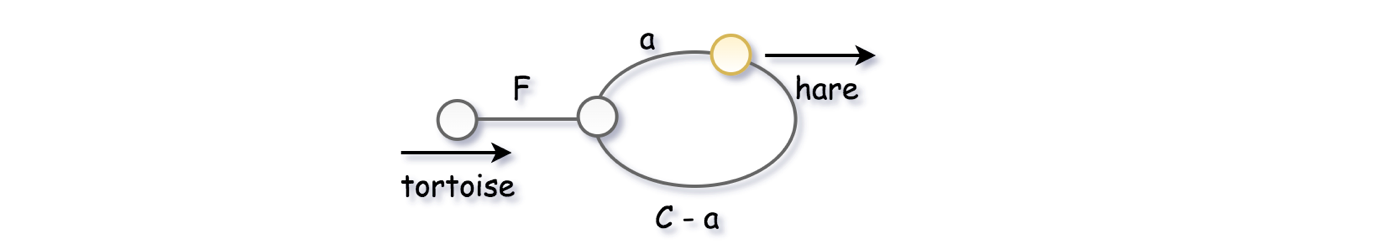


To compute the intersection point, let's note that the hare has traversed twice as many nodes as the tortoise, i.e. 2*d*(tortoise)=*d*(hare), that means

2(*F*+*a*)=*F*+*nC*+*a*, where *n* is some integer.

Hence the coordinate of the intersection point is *F*+*a*=*nC*.

**In phase 2**, we give the tortoise a second chance by slowing down the hare, so that it now moves with the speed of tortoise: tortoise = nums[tortoise], hare = nums[hare]. The tortoise is back at the starting position, and the hare starts from the intersection point.



Let's show that this time they meet at the cycle entrance after *F* steps.

* The tortoise started from zero, so its position after *F* steps is *F*.
* The hare started at the intersection point *F*+*a*=*nC*, so its position after F steps is *C*+*F*, that is the same point as *F*.
* So the tortoise and the (slowed down) hare will meet at the entrance of the cycle.

A picture containing diagram

Description automatically generated

A picture containing diagram

Description automatically generated

A picture containing diagram

Description automatically generated

A picture containing diagram

Description automatically generated

A picture containing diagram

Description automatically generated

A picture containing diagram

Description automatically generated

A picture containing diagram

Description automatically generated

A picture containing diagram

Description automatically generated

A picture containing diagram

Description automatically generated

A picture containing diagram

Description automatically generated

|  |
| --- |
| class Solution {  public int findDuplicate(int[] nums) {  // Find the intersection point of the two runners.  int tortoise = nums[0];  int hare = nums[0];  do {  tortoise = nums[tortoise];  hare = nums[nums[hare]];  } while (tortoise != hare);  // Find the "entrance" to the cycle.  tortoise = nums[0];  while (tortoise != hare) {  tortoise = nums[tortoise];  hare = nums[hare];  }  return hare;  }  } |

**Complexity Analysis**

* Time complexity : O(*n*) For detailed analysis, refer to [Linked List Cycle II](https://leetcode.com/problems/linked-list-cycle-ii/solution/#approach-2-floyds-tortoise-and-hare-accepted).
* Space complexity : O(1) For detailed analysis, refer to [Linked List Cycle II](https://leetcode.com/problems/linked-list-cycle-ii/solution/#approach-2-floyds-tortoise-and-hare-accepted).

**Median of Two Sorted Arrays**

Given two sorted arrays nums1 and nums2 of size m and n respectively, return **the median** of the two sorted arrays.

**Follow up:** The overall run time complexity should be O(log (m+n)).

**Example 1:**

**Input:** nums1 = [1,3], nums2 = [2]

**Output:** 2.00000

**Explanation:** merged array = [1,2,3] and median is 2.

**Example 2:**

**Input:** nums1 = [1,2], nums2 = [3,4]

**Output:** 2.50000

**Explanation:** merged array = [1,2,3,4] and median is (2 + 3) / 2 = 2.5.

**Example 3:**

**Input:** nums1 = [0,0], nums2 = [0,0]

**Output:** 0.00000

**Example 4:**

**Input:** nums1 = [], nums2 = [1]

**Output:** 1.00000

**Example 5:**

**Input:** nums1 = [2], nums2 = []

**Output:** 2.00000

**Constraints:**

* nums1.length == m
* nums2.length == n
* 0 <= m <= 1000
* 0 <= n <= 1000
* 1 <= m + n <= 2000
* -106 <= nums1[i], nums2[i] <= 106

# Finding the Median of 2 Sorted Arrays in Logarithmic Time

[hamid](https://medium.com/@hazemu?source=post_page-----1d3f2ecbeb46--------------------------------)

#### **[hamid](https://medium.com/@hazemu?source=post_page-----1d3f2ecbeb46--------------------------------)**

#### **[Mar 11, 2019·10 min read](https://medium.com/@hazemu/finding-the-median-of-2-sorted-arrays-in-logarithmic-time-1d3f2ecbeb46?source=post_page-----1d3f2ecbeb46--------------------------------)**

This problem is featured on [LeetCode](https://leetcode.com/problems/median-of-two-sorted-arrays/) along with a [fairly clever solution](https://leetcode.com/problems/median-of-two-sorted-arrays/solution/) that is explained in a somewhat intricate way. This post is an attempt to explain the general intuition behind that solution in simple terms.

# **Main Challenge**

The crux of this problem is finding what two arrays would look like when they are merged, without actually merging them since this would take O(n+m) time.

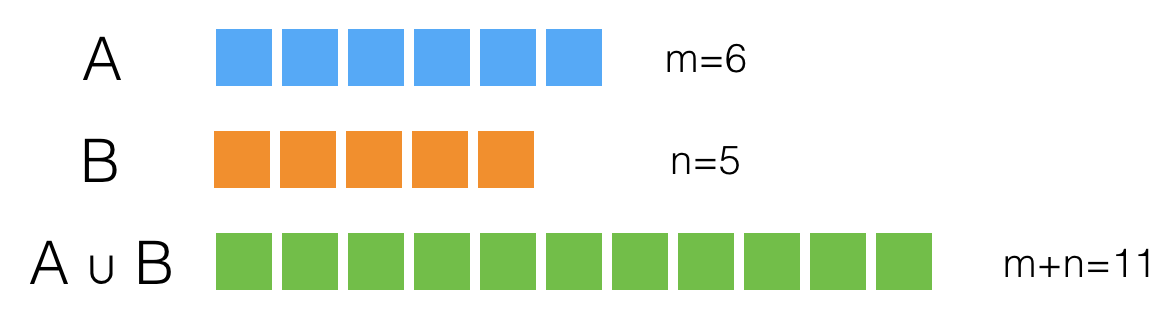


Fig. 1 — Two sorted arrays, A and B, whose lengths are m and n, respectively. A ∪ B is a third array that represents the result of merging A and B.

Practically speaking, we are only interested in knowing what the left half of the merged array, A ∪ B, would be, because this is the subarray that ends with the median.

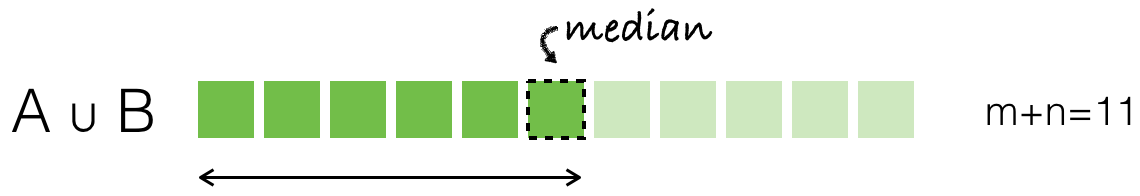


Fig. 2 — We are only interested in the left half (size = 6) of the merged array.

# **Key Questions**

## Question 1

Is there a way we can guess what the values in the left half of A ∪ B would be without merging A and B?

Let’s think about it. What do we know about this half? We know:

* It will contain six values.
* These six values could be coming from A, B, or both.

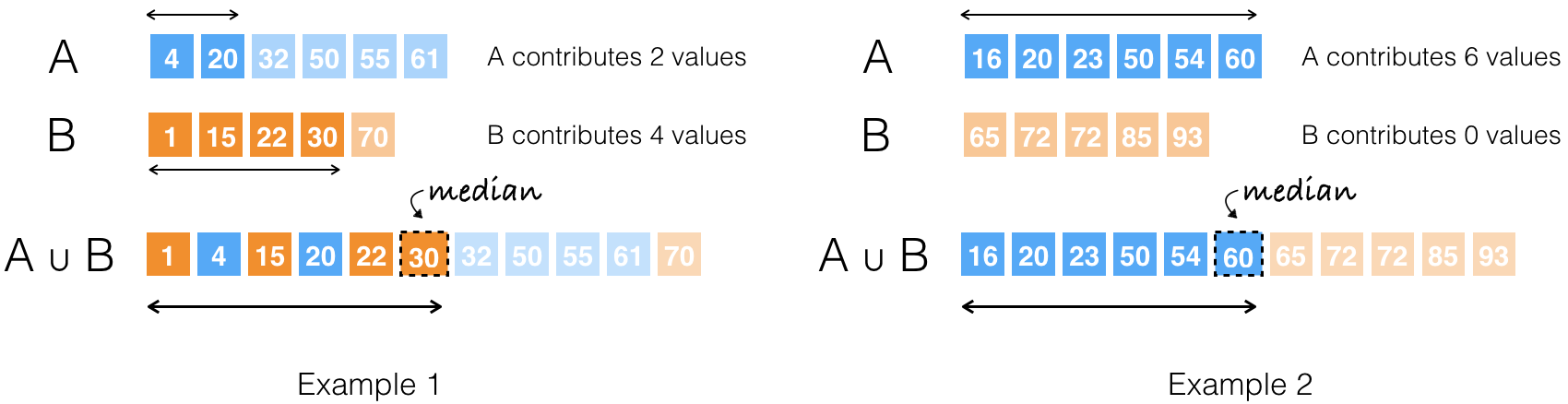


Fig. 3 — A couple of examples illustrating how the values within A and B determine the number of elements each array contributes to the left half of A ∪ B.

Not knowing anything about their values, A and B could contribute to the left half of A ∪ B in six different ways.

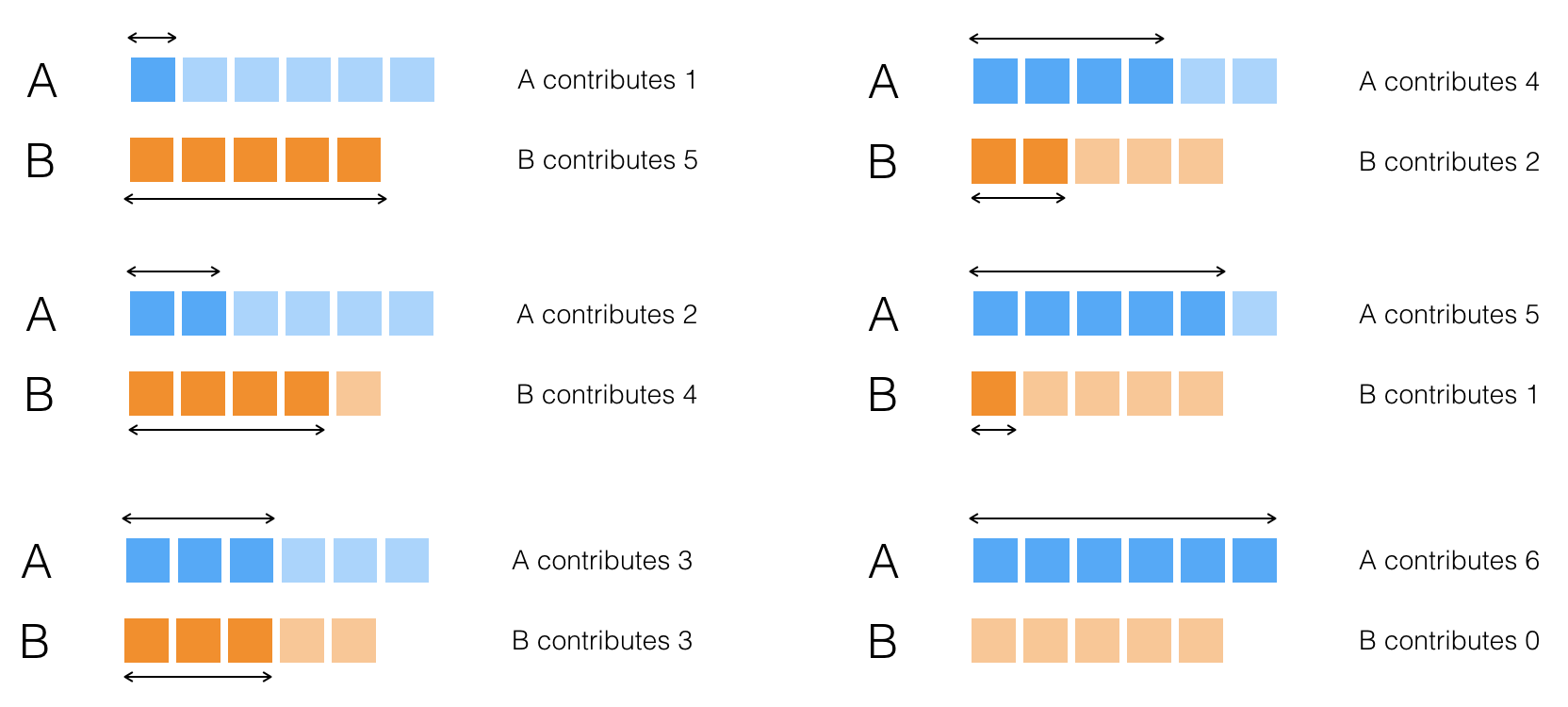


Fig. 4— An exhaustive list of the different ways A and B could contribute to the left half of their merged result.

## Question 2

How is knowing the number of values contributed by A and B to the left half of A ∪ B going to help us find the median?

Well,

* We could just compare the last value contributed by A with the last value contributed by B. The greater of the two would be the median.
* In cases where either array contributes zero elements — like the one in the lower right corner of Fig. 4 — the median will be the last value contributed by the other array.

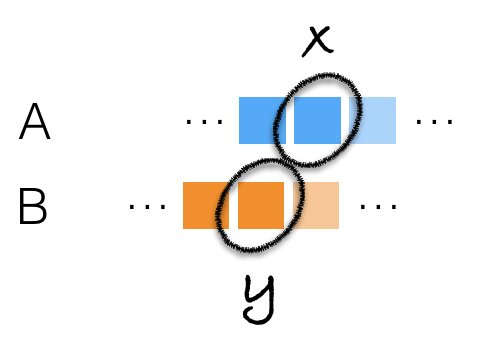


Fig. 6 — The greater of the 2 values contributed by A and B, labeled x and y, will be the median if A and B are merged.

## Question 3

What if m + n is an even number?

We would still need to know the last value in the left half of A ∪ B. The only difference is that we will need to know just one more value thereafter in order to compute the final value of the median.

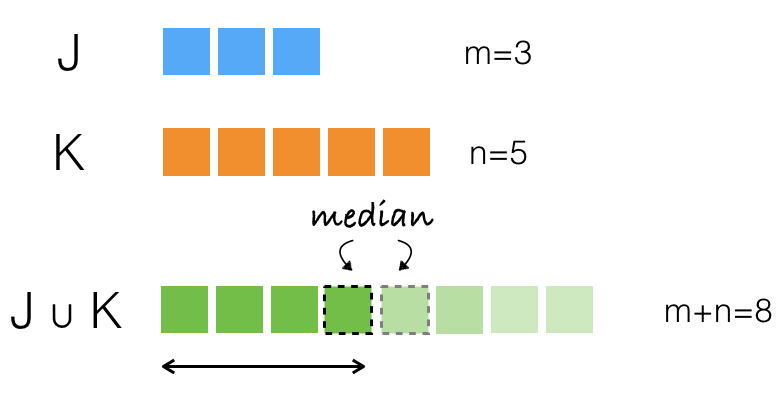


Fig. 7 — Another example with two arrays, J and K, whose combined size is even. We still need to identify the last value in the left half of their merged result. We’ll just need to find out the value next to it.

## Question 4

If there are six different ways A and B can contribute values to the left half of A ∪ B, how do we know the correct one?

To answer this question, let’s look at a few examples.

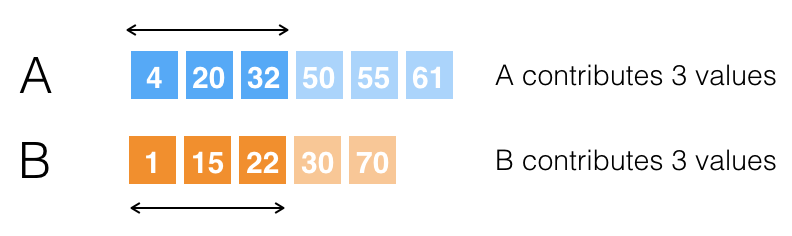


Fig. 8 — An example where we examine the possibility of arrays A and B contributing 3 values each, to the left half of A ∪ B.

We know based on the lengths of A and B that the left half of A ∪ B is of length six. We also know that the median will be the sixth value in A ∪ B. Let’s assume A and B will contribute three elements each, to the left half of A ∪ B, as shown in Fig. 8.

To verify this assumption, we examine the greatest value contributed by each of A and B, i.e. 32 and 22. Since 32 is greater, we expect it to show after 22 in A ∪ B. So, is it safe to say 32 will be the sixth value and the median? Unfortunately not. The reason is that when A and B are merged, 32 will not appear until after the value 30 from array B, since the merged array has to be sorted.

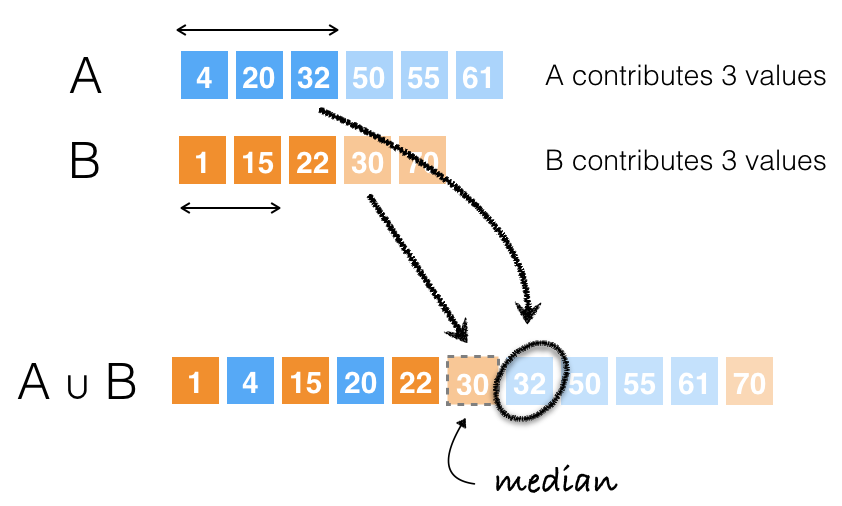


Fig. 9 — The value 32 will not appear in A ∪ B until after the value 30 from array B.

It turns out it wasn’t sufficient to compare 32 and 22. We should have also compared 32 to the value next to 22, i.e. 30, to make sure 32 won’t be pushed any further in A ∪ B. Generally speaking, it is not enough to compare the greatest values contributed by A and B, but we also need to make sure the greater of the two won’t be pushed further away by some value in the other array.

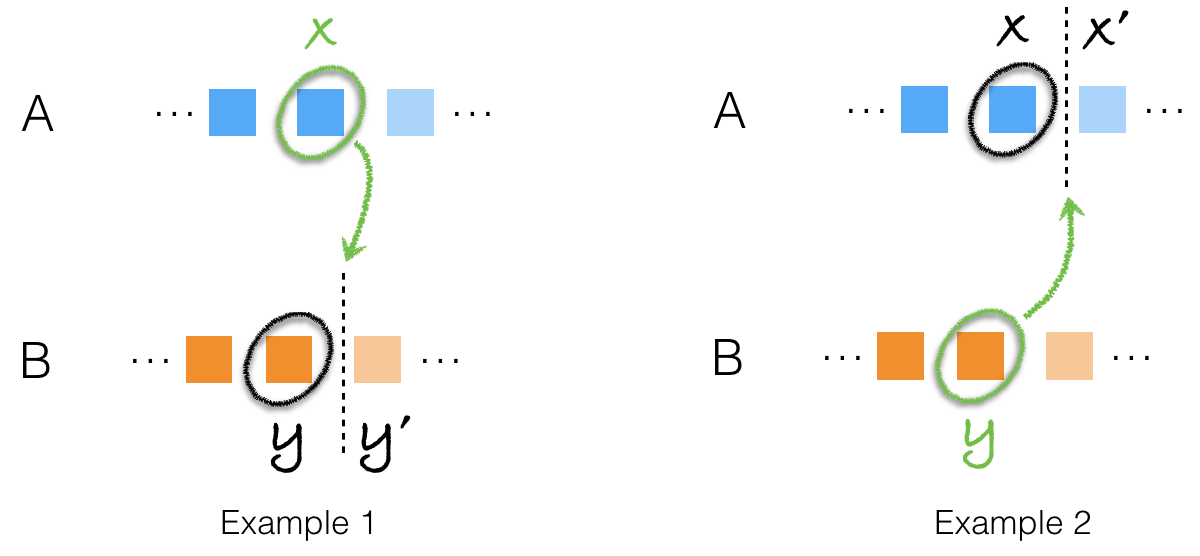


Fig. 10 — Example 1 illustrates the case where the greatest value contributed by A, labelled x, is greater than the greatest value contributed by B, labelled y. To confirm x is the median, we need to compare it with the value next to y, labelled y’, so we are certain x lies between y and y’, and will not be pushed further along in A ∪ B. Example 2 illustrates the case where y is greater than x.

Going back to the example in Fig. 9, 32 is greater than 22 but it is also greater than 22’s successor, 30. This implies that 32 is not the median.

Knowing that B has a smaller value to offer than A’s greatest contributed value, 32, strongly suggests B will contribute more values to the left half of A ∪ B. Therefore, we should consider increasing the number of values contributed by B, thus decreasing the number of values contributed by A. As show in Fig. 11, allowing B to contribute four values instead of three causes A’s contribution to shrink and, more importantly, reveals that 30 is the median.



Fig. 11 — When A and B contribute 2 and 4 values, respectively, the greatest value contributed by B, y = 30, is greater than the greatest value contributed by A, x = 20. y is also smaller than x’ = 32, which indicates y is indeed the median.

## Question 5

Now that we have a way to identify the correct split between A and B, can’t we just try all the possible splits to find the median?

We can, but examining every possible split means the amount of work we will be doing is still linearly proportional to m + n. For instance, in the example we have been studying, there were six different ways A (size = 6) and B (size = 5) could contribute to the left half of A ∪ B. We would like our algorithm to have logarithmic runtime complexity.

## Question 6

How do we find the correct split in logarithmic time?

We use the concept of binary search to reduce the number of possibilities we consider. It may not be very obvious how binary search is relevant, so let’s take a moment to understand this.

In essence, what we’re trying to find is the number of values each of A and B will contribute to the left half of A ∪ B. But since we know the size of this half in advance, (m + n)/2, we can simplify our objective by saying we’re only interested in the number of values A is contributing. For instance, in our example, if we know A is contributing four values, then it follows that B is contributing two, since the left half of A ∪ B has a total length of six.

This leads us to the following question: what is the minimum and maximum number of values can A contribute? In our example, A must contribute at least one value; the size of the left half of A ∪ B is six, and B has five values only. On the other hand, A can contribute all of its six values to the left half of A ∪ B, which could happen if all the values in A were smaller than those in B. This is to say we can find the median of A ∪ B if we know A’s contribution size, which is an integer in the range [1, 6]. Now instead of trying out all the possible sizes from 1 to 6, we can use binary search, i.e.

1. Set A’s minimum and maximum contribution sizes to 1 and 6, respectively (min = 1, max = 6).
2. Consider the midpoint between min and max, mid = (1 + 6)/2 = 3. Check to see if our conditions for finding the median are met if A’s contribution size is equal to mid (by performing the comparisons we discussed in the answer to question 4). If so, then we found the solution, and we know the median is the greater of the greatest values contributed by A and B.
3. Otherwise, we can adjust min to mid + 1 or max to mid − 1 based on comparing A’s greatest contributed value, B’s greatest contributed value, and the value that succeeds the smaller of the two.

## Question 7

How do we know whether to increase or decrease A’s contribution size?

* If y < x ≤ y’ then we found the solution, and x is the median.
* If x < y ≤ x’ then we found the solution, and y is the median.
* If x > y and x > y’ then we need to decrease A’s contribution size because x will end up beyond the left half of A ∪ B. It’s useful to observe that if x > y’ then x > y must be true since y’ > y.
* If y > x and y > x’ then we need to decrease B’s contribution size, i.e. increase A’s contribution size, because y will end up beyond the left half of A ∪ B. It’s also useful to observe that checking if y > x’ should be sufficient.



Fig. 12 — x is the median if it lies between y and y’. y is the median if it lies between x and x’. Otherwise, contributions of A/B need to be readjusted.

# **General Approach**

Listing 1 below shows an initial implementation of the solution we have discussed so far, written in C#. Certain operations have been deliberately abstracted behind functions whose implementations are not included. These operations are discussed in detail in the following sections. But before we move on, please take a moment to look at the code below and understand it thoroughly.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | public double FindMedianSortedArrays(int[]A, int[] B) | | { | | | | // TODO: Check for corner cases | | | |  | | | | int aLen = A.Length; | | | | int bLen = B.Length; | | | int leftHalfLen = GetLeftHalfLength(aLen + bLen); | | |  | | | // aMinCount and aMaxCount are the min and max number | | | // of values A can contribute to the left half of A ∪ B, | | | // respectively. | | | (int aMinCount, int aMaxCount) = GetMinMaxCounts(aLen, bLen); | | |  | | | while (aMinCount <= aMaxCount) | | | { | | | // aCount is the number of values A will contribute to left half of A ∪ B | | | int aCount = (aMinCount + aMaxCount) / 2; | | |  | | | // bCount is the number of values B will contribute to left half of A ∪ B. | | | // B will contribute as many values as necessary to fill however many remaining | | | // slots in the left half. | | | int bCount = leftHalfLen - aCount; | | |  | | | int x = A[aCount - 1]; // Last value contributed by A to left half of A ∪ B | | | int y = B[bCount - 1]; // Last value contributed by B to left half of A ∪ B | | | int xP = A[aCount]; // x' (value right after x) | | | int yP = B[bCount]; // y' (value right after y) | | |  | | | if (x > yP) | | | { | | | // Decrease A's contribution size; x lies in the right half of A ∪ B. | | | aMaxCount = aCount - 1; | | | } | | | else if (y > xP) | | | { | | | // Decrease B's contribution size, i.e. increase A's contribution size; | | | // y lies in the right half of A ∪ B. | | | aMinCount = aCount + 1; | | | } | | | else | | | { | | | // | | | // Neither x nor y lie beyond the left half of A ∪ B. This implies we | | | // found the right aCount. We don't know how x and y compare to each | | | // other yet though. | | | // | | |  | | | // | | | // If x > y then x is the median because x <= yP (see line 21) ⟹ y < x ≤ yP | | | // | | | if (x > y) | | | { | | | return x; | | | } | | |  | | | // | | | // If y > x then y is the median because y <= xP (see line 26) ⟹ x < y ≤ xP | | | // | | | if (y > x) | | | { | | | return y; | | | } | | |  | | | // x and y are equal. We can return either. | | | return x; | | | } | | | } | | |  | | | // TODO: Report invalid input | | | return -1; | | |  | } | | | |

The similarities in the overall structure between this algorithm and binary search should be clear by now. This implementation has a number of issues that we need to address nonetheless.

## Input Validation

The code in Listing 1 does not check if the input arrays are null or empty. Checking for null is trivial, and so is checking if both arrays are empty. If only one of them is empty, we can directly compute the median of the other.

## Computing Left Half Length

In the example we have been studying so far, we chose a left half of length 6 for two input arrays whose combined size is 11. The main advantage of this decision is that the median becomes the last element in this half. We also discussed how we can generalize this to work in case of even lengths.

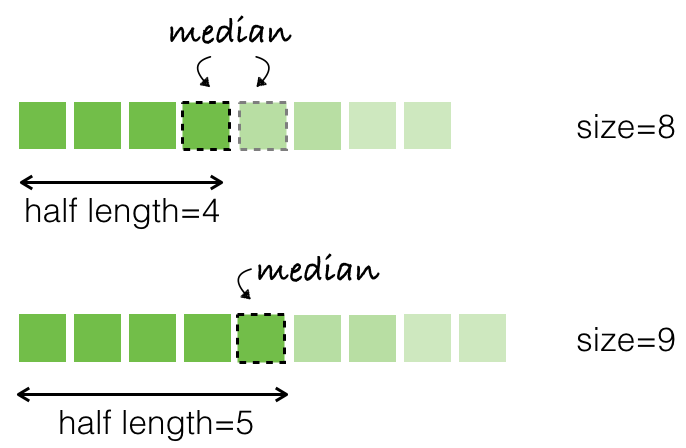


Fig. 13– Two example arrays of even and odd lengths. Left half length is 4 for the first array and 5 for the second.

To tackle both cases, we can compute the left half length using either of the two equations below.

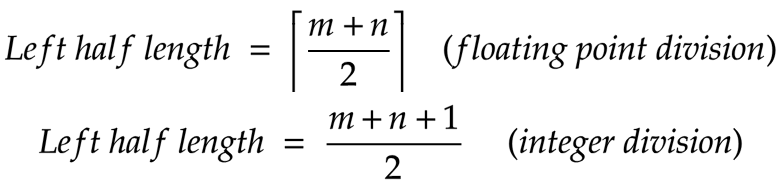


Fig. 14 — Equations for computing the left half length of the merged array, A ∪ B.

## Min and Max Number of Values to Contribute

Deciding the initial values of aMinCount and aMaxCount is both important and subtle. In the example of arrays A and B, we observed that aMinCount and aMaxCount were equal to 1 and 6, respectively. A couple of things worth highlighting in this example are:

* Zero isn’t a valid value for aMinCount since B has five values only. There is no way B can contribute enough values to fill all six slots of the left half of A ∪ B on its own.
* It’s very useful to observe we can conduct the search in B instead of A, thus defining bMinCount and bMaxCount rather than aMinCount and aMaxCount. However, it’s also important to realize bMinCount and bMaxCount won’t have the same initial range, [1, 6], as aMinCount and aMaxCount. This follows from the fact that, unlike A, B may contribute no values to the left half of A ∪ B (i.e. if all values in A are smaller than those in B), or all of its five values (i.e. if all values in B are smaller than those in A). In addition to having the simpler range of [0, m=5] for bMinCount and bMaxCount, searching B has the added benefit of having fewer values to examine.

## Computing Midpoint

We compute aCount as (aMinCount + aMaxCount)/2, which is susceptible to overflows if the values of aMinCount and aMaxCount are close to the maximum allowable integer value, e.g. when we’re searching the far right extents of a very large array. This is a general issue that all algorithm implementations based on binary search are susceptible to. You can read more about it in [this article](https://ai.googleblog.com/2006/06/extra-extra-read-all-about-it-nearly.html) by [Joshua Bloch](https://en.wikipedia.org/wiki/Joshua_Bloch).

## Guarding Against Invalid Index Errors

An important detail we need to take into account as we compare x, y, x’, and y’, is that some of them may be non-existent/undefined. For instance, if A is contributing all of its six values to the left half of A ∪ B, then

* y will be undefined since B is not contributing any values, and
* x’ will also be undefined because there are no elements left in A.

In this case, expressions like B[bCount − 1] and A[aCount] will yield index out of bounds/range errors, since bCount and aCount will have the values 0 and A.Length, respectively.

We certainly need some additional checks to account for such cases. Alternatively, the code in Listing 2 addresses this issue by utilizing a nifty C# feature — nullable types. If you are not familiar with C#, it’s sufficient to know that:

1. A nullable integer, as its name suggests, can be null.
2. Any less-than/greater-than comparisons involving a null nullable, directly evaluate to false.

|  |
| --- |
| public class Solution  {  public double FindMedianSortedArrays(int[] A, int[] B)  {  if (Object.ReferenceEquals(A, null) || Object.ReferenceEquals(B, null))  {  throw new ArgumentNullException();  }  int aLen = A.Length;  int bLen = B.Length;  // Make sure we always search the shorter array.  if (aLen > bLen)  {  Swap(ref A, ref B);  Swap(ref aLen, ref bLen);  }  int leftHalfLen = (aLen + bLen + 1) / 2;  // Since A is guaranteed to be either shorter or of  // the same length as B, we know it can contribute  // 0 or all of its values to the left half of A ∪ B.  int aMinCount = 0;  int aMaxCount = aLen;  while (aMinCount <= aMaxCount)  {  int aCount = aMinCount + ((aMaxCount - aMinCount) / 2);  int bCount = leftHalfLen - aCount;  //  // x can be null if A is not contributing any values to left half.  // e.g. A = [10, 11], B = [3, 4]  // ⟹ left half = [3, 4], aCount = 0, bCount = 2.  //  int? x = (aCount > 0) ? A[aCount - 1] : (int?)null;  //  // y can be null if B is not contributing any values to left half.  // e.g. A = [3, 4], B = [10, 11]  // ⟹ left half = [3, 4], aCount = 2, bCount = 0.  //  int? y = (bCount > 0) ? B[bCount - 1] : (int?)null;  //  // xP can be null if A is contributing all of its values to left half,  // i.e. aCount = A.Length.  // e.g. A = [3, 4], B = [10, 11]  // ⟹ left half = [3, 4], aCount = 2, bCount = 0.  //  int? xP = (aCount < aLen) ? A[aCount] : (int?)null;  //  // yP can be null if B is contributing all of its values to left half,  // i.e. bCount = B.Length.  // e.g. A = [10, 11], B = [3, 4]  // ⟹ left half = [3, 4], aCount = 0, bCount = 2.  //  int? yP = (bCount < bLen) ? B[bCount] : (int?)null;  if (x > yP)  {  // Decrease A's contribution size; x lies in the right half.  aMaxCount = aCount - 1;  }  else if (y > xP)  {  // Decrease B's contribution size, i.e. increase A's contribution size;  // y lies in the right half.  aMinCount = aCount + 1;  }  else  {  //  // Neither x nor y lie beyond the left half. We found the right aCount.  // We don't know how x and y compare to each other yet though.  //  //  // If aLen + bLen is odd, the median is the greater of x and y.  // Remember that either x or y can be null (if A or B is not contributing).  //  int leftHalfEnd = (x == null)  ? y.Value  : (y == null)  ? x.Value  : Math.Max(x.Value, y.Value);  if (IsOdd(aLen + bLen))  {  return leftHalfEnd;  }  //  // aLen + bLen is even. To compute the median, we need to find  // the first element in the right half, which will be the smaller  // of xP and yP. Remember that either xP or yP can be null (if all  // the values of A or B are in the left half).  //  int rightHalfStart = (xP == null)  ? yP.Value  : (yP == null)  ? xP.Value  : Math.Min(xP.Value, yP.Value);  return (leftHalfEnd + rightHalfStart) / 2.0;  }  }  throw new InvalidOperationException("Unexpected code path reached");  }  private void Swap<T>(ref T x, ref T y)  {  T temp = x;  x = y;  y = temp;  }  // The least significant bit of any odd number is 1.  private bool IsOdd(int x) => (x & 1) == 1;  } |

Listing 2 — The final implementation

Of course you can replace these nullables with simple index checks if you’re solving this problem using a different language. An example is provided in Listing 3 below.

|  |
| --- |
| public class Solution  {  public double FindMedianSortedArrays(int[] A, int[] B)  {  if (Object.ReferenceEquals(A, null) || Object.ReferenceEquals(B, null))  {  throw new ArgumentNullException();  }  int aLen = A.Length;  int bLen = B.Length;  // Make sure we always search the shorter array.  if (aLen > bLen)  {  Swap(ref A, ref B);  Swap(ref aLen, ref bLen);  }  int leftHalfLen = (aLen + bLen + 1) / 2;  // Since A is guaranteed to be the shorter array,  // we know it can contribute 0 or all of its values.  int aMinCount = 0;  int aMaxCount = aLen;  while (aMinCount <= aMaxCount)  {  int aCount = aMinCount + ((aMaxCount - aMinCount) / 2);  int bCount = leftHalfLen - aCount;  //  // Make sure aCount is greater than 0 (because A can contribute 0 values;  // remember that A is either shorter or of the same length as B). This also  // implies bCount will be less than B.Length since it won't be possible  // for B to contribute all of its values if A has contributed at least 1  // value.  //  if (aCount > 0 && A[aCount - 1] > B[bCount])  {  // Decrease A's contribution size; x lies in the right half.  aMaxCount = aCount - 1;  }    //  // Make sure aCount is less than A.Length since A can actually contribute  // all of its values (remember that A is either shorter or of the same  // length as B). This also implies bCount > 0 because B has to contribute  // at least 1 value if aCount < A.Length.  //  else if (aCount < aLen && B[bCount - 1] > A[aCount])  {  // Decrease B's contribution size, i.e. increase A's contribution size;  // y lies in the right half.  aMinCount = aCount + 1;  }  else  {  //  // Neither x nor y lie beyond the left half. We found the right aCount.  // We don't know how x and y compare to each other yet though.  //  //  // If aLen + bLen is odd, the median is the greater of x and y.  //  int leftHalfEnd =  (aCount == 0) // A not contributing?  ? B[bCount - 1] // aCount = 0 implies bCount > 0  : (bCount == 0) // B is not contributing?  ? A[aCount - 1] // bCount = 0 implies aCount > 0  : Math.Max(A[aCount - 1], B[bCount - 1]);  if (IsOdd(aLen + bLen))  {  return leftHalfEnd;  }  //  // aLen + bLen is even. To compute the median, we need to find  // the first element in the right half, which will be the smaller  // of A[aCount] and B[bCount]. Remember that aCount could be equal  // to A.Length, bCount could be equal to B.Length (if all the values  // of A or B are in the left half).  //  int rightHalfStart =  (aCount == aLen) // A is all in the left half?  ? B[bCount] // aCount = aLen implies bCount < B.Length  : (bCount == bLen) // B is all in the left half?  ? A[aCount] // bCount = B.Length implies aCount < A.Length  : Math.Min(A[aCount], B[bCount]);  return (leftHalfEnd + rightHalfStart) / 2.0;  }  }  throw new InvalidOperationException("Unexpected code path reached");  }  private void Swap<T>(ref T x, ref T y)  {  T temp = x;  x = y;  y = temp;  }  // The least significant bit of any odd number is 1.  private bool IsOdd(int x) => (x & 1) == 1;  } |

<https://medium.com/@hazemu/finding-the-median-of-2-sorted-arrays-in-logarithmic-time-1d3f2ecbeb46>

**Find K-th Smallest Pair Distance**

Given an integer array, return the k-th smallest **distance** among all the pairs. The distance of a pair (A, B) is defined as the absolute difference between A and B.

**Example 1:**

**Input:**

nums = [1,3,1]

k = 1

**Output: 0**

**Explanation:**

Here are all the pairs:

(1,3) -> 2

(1,1) -> 0

(3,1) -> 2

Then the 1st smallest distance pair is (1,1), and its distance is 0.

**Note:**

1. 2 <= len(nums) <= 10000.
2. 0 <= nums[i] < 1000000.
3. 1 <= k <= len(nums) \* (len(nums) - 1) / 2.

   Hide Hint #1

Binary search for the answer. How can you check how many pairs have distance <= X?

#### **Approach #1: Heap [Time Limit Exceeded]**

**Intuition and Algorithm**

Sort the points. For every point with index i, the pairs with indexes (i, j) [by order of distance] are (i, i+1), (i, i+2), ..., (i, N-1).

Let's keep a heap of pairs, initially heap = [(i, i+1) for all i], and ordered by distance (the distance of (i, j) is nums[j] - nums[i].) Whenever we use a pair (i, x) from our heap, we will add (i, x+1) to our heap when appropriate.

|  |
| --- |
| class Solution {  public int smallestDistancePair(int[] nums, int k) {  Arrays.sort(nums);  PriorityQueue<Node> heap = new PriorityQueue<Node>(nums.length,  Comparator.<Node> comparingInt(node -> nums[node.nei] - nums[node.root]));  for (int i = 0; i + 1 < nums.length; ++i) {  heap.offer(new Node(i, i+1));  }  Node node = null;  for (; k > 0; --k) {  node = heap.poll();  if (node.nei + 1 < nums.length) {  heap.offer(new Node(node.root, node.nei + 1));  }  }  return nums[node.nei] - nums[node.root];  }  }  class Node {  int root;  int nei;  Node(int r, int n) {  root = r;  nei = n;  }  } |

**Complexity Analysis**

* Time Complexity: *O*((*k*+*N*)log*N*), where *N* is the length of nums. As k = O(N^2), this is *O*(*N*2log*N*) in the worst case. The complexity added by our heap operations is either *O*((*k*+*N*)log*N*) in the Java solution, or *O*(*k*log*N*+*N*) in the Python solution because the heapq.heapify operation is linear time. Additionally, we add *O*(*N*log*N*) complexity due to sorting.
* Space Complexity: *O*(*N*), the space used to store our heap of at most N-1 elements.

#### **Approach #2: Binary Search + Prefix Sum [Accepted]**

**Intuition**

Let's binary search for the answer. It's definitely in the range [0, W], where W = max(nums) - min(nums)].

Let possible(guess) be true if and only if there are k or more pairs with distance less than or equal to guess. We will focus on evaluating our possible function quickly.

**Algorithm**

Let prefix[v] be the number of points in nums less than or equal to v. Also, let multiplicity[j] be the number of points i with i < j and nums[i] == nums[j]. We can record both of these with a simple linear scan.

Now, for every point i, the number of points j with i < j and nums[j] - nums[i] <= guess is prefix[x+guess] - prefix[x] + (count[i] - multiplicity[i]), where count[i] is the number of ocurrences of nums[i] in nums. The sum of this over all i is the number of pairs with distance <= guess.

Finally, because the sum of count[i] - multiplicity[i] is the same as the sum of multiplicity[i], we could just replace that term with multiplicity[i] without affecting the answer. (Actually, the sum of multiplicities in total will be a constant used in the answer, so we could precalculate it if we wanted.)

In our Java solution, we computed possible = count >= k directly in the binary search instead of using a helper function.

|  |
| --- |
| class Solution {  public int smallestDistancePair(int[] nums, int k) {  Arrays.sort(nums);  int WIDTH = 2 \* nums[nums.length - 1];  //multiplicity[i] = number of nums[j] == nums[i] (j < i)  int[] multiplicity = new int[nums.length];  for (int i = 1; i < nums.length; ++i) {  if (nums[i] == nums[i-1]) {  multiplicity[i] = 1 + multiplicity[i - 1];  }  }  //prefix[v] = number of values <= v  int[] prefix = new int[WIDTH];  int left = 0;  for (int i = 0; i < WIDTH; ++i) {  while (left < nums.length && nums[left] == i) left++;  prefix[i] = left;  }  int lo = 0;  int hi = nums[nums.length - 1] - nums[0];  while (lo < hi) {  int mi = (lo + hi) / 2;  int count = 0;  for (int i = 0; i < nums.length; ++i) {  count += prefix[nums[i] + mi] - prefix[nums[i]] + multiplicity[i];  }  //count = number of pairs with distance <= mi  if (count >= k) hi = mi;  else lo = mi + 1;  }  return lo;  }  } |

**Complexity Analysis**

* Time Complexity: *O*(*W*+*N*log*W*+*N*log*N*), where *N* is the length of nums, and *W* is equal to nums[nums.length - 1] - nums[0]. We do *O*(*W*) work to calculate prefix initially. The log*W* factor comes from our binary search, and we do *O*(*N*) work inside our call to possible (or to calculate count in Java). The final *O*(*N*log*N*) factor comes from sorting.
* Space Complexity: *O*(*N*+*W*), the space used to store multiplicity and prefix.

#### **Approach #3: Binary Search + Sliding Window [Accepted]**

**Intuition**

As in Approach #2, let's binary search for the answer, and we will focus on evaluating our possible function quickly.

**Algorithm**

We will use a sliding window approach to count the number of pairs with distance <= guess.

For every possible right, we maintain the loop invariant: left is the smallest value such that nums[right] - nums[left] <= guess. Then, the number of pairs with right as it's right-most endpoint is right - left, and we add all of these up.

|  |
| --- |
| class Solution {  public int smallestDistancePair(int[] nums, int k) {  Arrays.sort(nums);  int lo = 0;  int hi = nums[nums.length - 1] - nums[0];  while (lo < hi) {  int mi = (lo + hi) / 2;  int count = 0, left = 0;  for (int right = 0; right < nums.length; ++right) {  while (nums[right] - nums[left] > mi) left++;  count += right - left;  }  //count = number of pairs with distance <= mi  if (count >= k) hi = mi;  else lo = mi + 1;  }  return lo;  }  } |

**Complexity Analysis**

* Time Complexity: *O*(*N*log*W*+*N*log*N*), where *N* is the length of nums, and *W* is equal to nums[nums.length - 1] - nums[0]. The log*W* factor comes from our binary search, and we do *O*(*N*) work inside our call to possible (or to calculate count in Java). The final *O*(*N*log*N*) factor comes from sorting.
* Space Complexity: *O*(1). No additional space is used except for integer variables.

**Split Array Largest Sum**

Given an array nums which consists of non-negative integers and an integer m, you can split the array into m non-empty continuous subarrays.

Write an algorithm to minimize the largest sum among these m subarrays.

**Example 1:**

**Input:** nums = [7,2,5,10,8], m = 2

**Output:** 18

**Explanation:**

There are four ways to split nums into two subarrays.

The best way is to split it into [7,2,5] and [10,8],

where the largest sum among the two subarrays is only 18.

**Example 2:**

**Input:** nums = [1,2,3,4,5], m = 2

**Output:** 9

**Example 3:**

**Input:** nums = [1,4,4], m = 3

**Output:** 4

**Constraints:**

* 1 <= nums.length <= 1000
* 0 <= nums[i] <= 106
* 1 <= m <= min(50, nums.length)

#### **Approach #1 Brute Force [Time Limit Exceeded]**

**Intuition**

Check all possible splitting plans to find the minimum largest value for subarrays.

**Algorithm**

We can use depth-first search to generate all possible splitting plan. For each element in the array, we can choose to append it to the previous subarray or start a new subarray starting with that element (if the number of subarrays does not exceed m). The sum of the current subarray can be updated at the same time.

|  |
| --- |
| class Solution {  private int ans;  private int n, m;  private void dfs(int[] nums, int i, int cntSubarrays, int curSum, int curMax) {  if (i == n && cntSubarrays == m) {  ans = Math.min(ans, curMax);  return;  }  if (i == n) {  return;  }  if (i > 0) {  dfs(nums, i + 1, cntSubarrays, curSum + nums[i], Math.max(curMax, curSum + nums[i]));  }  if (cntSubarrays < m) {  dfs(nums, i + 1, cntSubarrays + 1, nums[i], Math.max(curMax, nums[i]));  }  }  public int splitArray(int[] nums, int M) {  ans = Integer.MAX\_VALUE;  n = nums.length;  m = M;  dfs(nums, 0, 0, 0, 0);  return ans;  }  } |

**Complexity Analysis**

* Time complexity : O(n^m). To split n elements into m parts, we can have {m - 1}(*m*−1*n*−1​) different solutions. This is equivalent to n ^ m.
* Space complexity : *O*(*n*). We only need the space to store the array.

#### **Approach #2 Dynamic Programming [Accepted]**

**Intuition**

The problem satisfies the non-aftereffect property. We can try to use dynamic programming to solve it.

The non-aftereffect property means, once the state of a certain stage is determined, it is not affected by the state in the future. In this problem, if we get the largest subarray sum for splitting nums[0..i] into j parts, this value will not be affected by how we split the remaining part of nums.

To know more about non-aftereffect property, this link may be helpful : <http://www.programering.com/a/MDOzUzMwATM.html>

**Algorithm**

Let's define f[i][j] to be the minimum largest subarray sum for splitting nums[0..i] into j parts.

Consider the jth subarray. We can split the array from a smaller index k to i to form it. Thus f[i][j] can be derived from max(f[k][j - 1], nums[k + 1] + ... + nums[i]). For all valid index k, f[i][j] should choose the minimum value of the above formula.

The final answer should be f[n][m], where n is the size of the array.

For corner situations, all the invalid f[i][j] should be assigned with INFINITY, and f[0][0] should be initialized with 0.

#### **Approach #2 Dynamic Programming [Accepted]**

**Intuition**

The problem satisfies the non-aftereffect property. We can try to use dynamic programming to solve it.

The non-aftereffect property means, once the state of a certain stage is determined, it is not affected by the state in the future. In this problem, if we get the largest subarray sum for splitting nums[0..i] into j parts, this value will not be affected by how we split the remaining part of nums.

To know more about non-aftereffect property, this link may be helpful : <http://www.programering.com/a/MDOzUzMwATM.html>

**Algorithm**

Let's define f[i][j] to be the minimum largest subarray sum for splitting nums[0..i] into j parts.

Consider the jth subarray. We can split the array from a smaller index k to i to form it. Thus f[i][j] can be derived from max(f[k][j - 1], nums[k + 1] + ... + nums[i]). For all valid index k, f[i][j] should choose the minimum value of the above formula.

The final answer should be f[n][m], where n is the size of the array.

For corner situations, all the invalid f[i][j] should be assigned with INFINITY, and f[0][0] should be initialized with 0.

|  |
| --- |
| class Solution {  public int splitArray(int[] nums, int m) {  int n = nums.length;  int[][] f = new int[n + 1][m + 1];  int[] sub = new int[n + 1];  for (int i = 0; i <= n; i++) {  for (int j = 0; j <= m; j++) {  f[i][j] = Integer.MAX\_VALUE;  }  }  for (int i = 0; i < n; i++) {  sub[i + 1] = sub[i] + nums[i];  }  f[0][0] = 0;  for (int i = 1; i <= n; i++) {  for (int j = 1; j <= m; j++) {  for (int k = 0; k < i; k++) {  f[i][j] = Math.min(f[i][j], Math.max(f[k][j - 1], sub[i] - sub[k]));  }  }  }  return f[n][m];  }  } |

**Complexity Analysis**

* Time complexity : *O*(*n*2*m*). The total number of states is *O*(*n*∗*m*). To compute each state f[i][j], we need to go through the whole array to find the optimum k. This requires another *O*(*n*) loop. So the total time complexity is O(n ^ 2 \* m).
* Space complexity : *O*(*n*∗*m*). The space complexity is equivalent to the number of states, which is *O*(*n*∗*m*).

#### **Approach #3 Binary Search + Greedy [Accepted]**

**Intuition**

We can easily find a property for the answer:

If we can find a splitting method that ensures the maximum largest subarray sum will not exceed a value x, then we can also find a splitting method that ensures the maximum largest subarray sum will not exceed any value y that is greater than x.

Lets define this property as F(x) for the value x. F(x) is true means we can find a splitting method that ensures the maximum largest subarray sum will not exceed x.

From the discussion above, we can find out that for x ranging from -INFINITY to INFINITY, F(x) will start with false, then from a specific value x0, F(x) will turn to true and stay true forever.

Obviously, the specific value x0 is our answer.

**Algorithm**

We can use Binary search to find the value x0. Keeping a value mid = (left + right) / 2. If F(mid) is false, then we will search the range [mid + 1, right]; If F(mid) is true, then we will search [left, mid - 1].

For a given x, we can get the answer of F(x) using a greedy approach. Using an accumulator sum to store the sum of the current processing subarray and a counter cnt to count the number of existing subarrays. We will process the elements in the array one by one. For each element num, if sum + num <= x, it means we can add num to the current subarray without exceeding the limit. Otherwise, we need to make a cut here, start a new subarray with the current element num. This leads to an increment in the number of subarrays.

After we have finished the whole process, we need to compare the value cnt to the size limit of subarrays m. If cnt <= m, it means we can find a splitting method that ensures the maximum largest subarray sum will not exceed x. Otherwise, F(x) should be false.

|  |
| --- |
| class Solution {  public int splitArray(int[] nums, int m) {  long l = 0;  long r = 0;  int n = nums.length;  for (int i = 0; i < n; i++) {  r += nums[i];  if (l < nums[i]) {  l = nums[i];  }  }  long ans = r;  while (l <= r) {  long mid = (l + r) >> 1;  long sum = 0;  int cnt = 1;  for (int i = 0; i < n; i++) {  if (sum + nums[i] > mid) {  cnt ++;  sum = nums[i];  } else {  sum += nums[i];  }  }  if (cnt <= m) {  ans = Math.min(ans, mid);  r = mid - 1;  } else {  l = mid + 1;  }  }  return (int)ans;  }  } |

\*\*Complexity Analysis\*\*

* Time complexity : *O*(*n*∗*log*(*sumofarray*)). The binary search costs *O*(*log*(*sumofarray*)), where sum of array is the sum of elements in nums. For each computation of F(x), the time complexity is *O*(*n*) since we only need to go through the whole array.
* Space complexity : *O*(1) space complexity without taking the output list into account, and *O*(*n*) to store the output list.
* Set the search range between min=(largest single value) and max=(sum of all values).  
  The min starts there because we're looking for the sum of the largest group in the final set of groups. And no matter what groups you create, the largest value has to be in it, so the largest group can't be smaller than that. (This assumes no negative numbers.)
* Calculate the midpoint between min and max. This midpoint is the group size we're going to try out to see how well it performs.
* Split the nums list into groups such that no group has a value larger than the chosen midpoint.  
  Note that we may end up with too many or too few groups. That's fine.
* Compare the number of groups we created against the target m. If we created too many groups, we know the final answer must be between mid+1 and max. That's because we need fewer groups and the way to achieve fewer groups is to increase the allowed maximum sum in each group.
* On the other hand, if the number of groups is too small, we know the final answer is between min and mid-1 because we need to increase the number of groups which means the target sum is something smaller than the one we used. This is actually also a possible answer assuming m is valid because you can always take any group and split it up to make more groups, so the mid value you targeted is at worst, higher than the real value.
* On the third hand, if the number of groups is just right, we have a possible answer, so remember that answer. However, we should keep searching just in case there is a better answer. We're ultimately looking for smaller maximum sums, so the potentially better answer is between min and mid-1.
* Repeat the process until there is nothing else to search. Then use the minimum value we found during the above process.