# Binary Search - Find Target

The traditional binary search says, given a sorted array, and a target number and you search if the target number is in the array or not. You start with the first element and last element and get the middle position between them if the target number is smaller or equal than the middle number you search the range between the first and middle, inclusive, if the target number is greater than the middle you search the range middle + 1 and last, inclusive. You keep on doing so until first equal to the last. The second search must start with middle + 1 to avoid you loop forever between the position P and P + 1 when P is even.

However I would like you to think more from a different perspective, Instead of searching which range the target number can be found, you say discard the range where the target number not possible to be found, the algorithm is exactly same, but such think will help you to resolve many binary search problem.

## 34. Find First and Last Position of Element in Sorted Array

Medium

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

Your algorithm's runtime complexity must be in the order of *O*(log *n*).

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

**Example 1:**

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

**Output:** [3,4]

**Example 2:**

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

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

### Analysis:

This is a tradition binary search. You just need to reduce the search range. Keep the first anchor in the position where the value is smaller or equal to the target.

/// <summary>

/// Leet code #34. Find First and Last Position of Element in Sorted Array

///

/// Given an array of integers nums sorted in ascending order, find the

/// starting and ending position of a given target value.

/// Your algorithm's runtime complexity must be in the order of O(log n).

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

/// Example 1:

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

/// Output: [3,4]

///

/// Example 2:

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

/// Output: [-1,-1]

/// </summary>

vector<int> LeetCodeBinarySearch::searchRange(vector<int>& nums, int target)

{

vector<int> result = { -1, -1 };

if (nums.empty()) return result;

int first = 0, last = nums.size() - 1;

while (first < last)

{

int middle = first + (last - first) / 2;

if (target > nums[middle])

{

first = middle + 1;

}

else

{

last = middle;

}

}

if (first < (int)nums.size() && nums[first] == target)

{

last = first;

while ((last < (int)(nums.size() - 1)) && (nums[last + 1] == target))

{

last++;

}

result = { first, last };

}

return result;

}

## 162. Find Peak Element

Medium

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

Given an input array nums, where nums[i] ≠ nums[i+1], find a peak element and return its index.

The array may contain multiple peaks, in that case return the index to any one of the peaks is fine.

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:** 1 or 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.

**Note:**

Your solution should be in logarithmic complexity.

### Analysis:

For any random position, we can compare the neighbor to pick the higher one, please notice that we are not target for the maximum number in the array, just some value which is greater than its neighbors.

/// <summary>

/// Leet code #162. Find Peak Element

///

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

///

/// Given an input array nums, where nums[i] ≠ nums[i+1], find a peak element

/// and return its index.

///

/// The array may contain multiple peaks, in that case return the index to

/// any one of the peaks is fine.

///

/// You may imagine that nums[-1] = nums[n] = INT\_MIN.

///

/// 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: 1 or 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.

///

/// Note:

/// Your solution should be in logarithmic complexity.

/// </summary>

int LeetCodeBinarySearch::findPeakElement(vector<int>& nums)

{

int first = 0;

int last = nums.size() - 1;

int middle = last;

while (first < last)

{

middle = first + (last - first) / 2;

if (nums[middle] < nums[middle + 1])

{

first = middle + 1;

}

else

{

last = middle;

}

}

return last;

}

## 1060. Missing Element in Sorted Array

Medium

Given a sorted array A of **unique** numbers, find the *K*-th missing number starting from the leftmost number of the array.

**Example 1:**

**Input:** A = [4,7,9,10], K = 1

**Output:** 5

**Explanation:**

The first missing number is 5.

**Example 2:**

**Input:** A = [4,7,9,10], K = 3

**Output:** 8

**Explanation:**

The missing numbers are [5,6,8,...], hence the third missing number is 8.

**Example 3:**

**Input:** A = [1,2,4], K = 3

**Output:** 6

**Explanation:**

The missing numbers are [3,5,6,7,...], hence the third missing number is 6.

**Note:**

1. 1 <= A.length <= 50000
2. 1 <= A[i] <= 1e7
3. 1 <= K <= 1e8

### Analysis:

You can try on any position, and calculate the gap between the distance between value and the first value against index, so you will know how many number missing before the current number. You conduct such search as a binary search.

Just watch when you calculate the number you should start with the position where you do not have K missing number ahead, and assume from first position to this position, there is no missing number. It will make your formula easy.

/// <summary>

/// Leet code 1060. Missing Element in Sorted Array

///

/// Given a sorted array A of unique numbers, find the K-th missing number

/// starting from the leftmost number of the array.

///

/// Example 1:

/// Input: A = [4,7,9,10], K = 1

/// Output: 5

/// Explanation:

/// The first missing number is 5.

///

/// Example 2:

/// Input: A = [4,7,9,10], K = 3

/// Output: 8

/// Explanation:

/// The missing numbers are [5,6,8,...], hence the third missing number is 8.

///

/// Example 3:

/// Input: A = [1,2,4], K = 3

/// Output: 6

/// Explanation:

/// The missing numbers are [3,5,6,7,...], hence the third missing number is 6.

///

/// Note:

/// 1. 1 <= A.length <= 50000

/// 2. 1 <= A[i] <= 1e7

/// 3. 1 <= K <= 1e8

/// </summary>

int LeetCode::missingElement(vector<int>& nums, int k)

{

int first = 0;

int last = nums.size() - 1;

int pos = first;

while (first <= last)

{

int mid = first + (last - first) / 2;

if (nums[mid] - mid - nums[0] < k)

{

pos = mid;

first = mid + 1;

}

else

{

last = mid - 1;

}

}

return k + pos + nums[0];

}

## 33. Search in Rotated Sorted Array

Medium

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]).

You are given a target value to search. If found in the array return its index, otherwise return -1.

You may assume no duplicate exists in the array.

Your algorithm's runtime complexity must be in the order of *O*(log *n*).

**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

### Analysis:

For a rotated array, you will always have one part is in order and another may or may not be in order. You can compare if the target number is within the range of the ordered part, if yes, discard the other part, is no, discard the ordered part.

/// <summary>

/// Leet code #33. Search in Rotated Sorted Array

///

/// 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]).

///

/// You are given a target value to search. If found in the array return

/// its index, otherwise return -1.

///

/// You may assume no duplicate exists in the array.

///

/// Your algorithm's runtime complexity must be in the order of O(log n).

///

/// 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

/// </summary>

int LeetCodeBinarySearch::search(vector<int>& nums, int target)

{

int first = 0, last = nums.size() - 1;

while (first <= last)

{

size\_t middle = first + (last - first) / 2;

if (target == nums[middle])

{

return middle;

}

// because a range can be one item, so = is important.

else if (nums[first] <= nums[middle])

{

// if the target is not within ordered range

if ((target < nums[first]) || (target > nums[middle]))

{

first = middle + 1;

}

else // if not in another range.

{

last = middle - 1;

}

}

else

{

if ((target > nums[middle]) && (target <= nums[last]))

{

first = middle + 1;

}

else

{

last = middle - 1;

}

}

}

return -1;

}

## 81. Search in Rotated Sorted Array II

Medium

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

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

You are given a target value to search. If found in the array return true, otherwise return false.

**Example 1:**

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

**Output:** true

**Example 2:**

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

**Output:** false

**Follow up:**

* This is a follow up problem to [Search in Rotated Sorted Array](https://leetcode.com/problems/search-in-rotated-sorted-array/description/), where nums may contain duplicates.
* Would this affect the run-time complexity? How and why?

### Analysis:

For a rotated array, you will always have one part is in order and another may or may not be in order. You can compare if the target number is within the range of the ordered part, if yes, discard the other part, is no, discard the ordered part. Please notice that if you see an disordered part it is useless, because we cannot determine whether the target number is in there or not. If we can not find any ordered part, we have to discard first or last until we see the ordered part.

One optimization you can consider for the following implementation is that you use recursive call to resolve the problem with given first, last. This may speed up in the case where you have large number of duplicated number either in front or in back.

/// <summary>

/// Leetcode #81. Search in Rotated Sorted Array II

///

/// Suppose an array sorted in ascending order is rotated at some pivot

/// unknown to you beforehand.

///

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

///

/// You are given a target value to search. If found in the array return true,

/// otherwise return false.

///

/// Example 1:

///

/// Input: nums = [2,5,6,0,0,1,2], target = 0

/// Output: true

///

/// Example 2:

///

/// Input: nums = [2,5,6,0,0,1,2], target = 3

/// Output: false

/// Follow up:

///

/// This is a follow up problem to Search in Rotated Sorted Array, where nums

/// may contain duplicates.

/// Would this affect the run-time complexity? How and why?

/// </summary>

bool LeetCodeBinarySearch::searchII(vector<int>& nums, int target)

{

int first = 0;

// because we need to compare with right end, so must point to a solid item.

int last = nums.size() - 1;

while (first <= last)

{

int middle = first + (last - first) / 2;

if (target == nums[middle])

{

return true;

}

else if (nums[first] < nums[middle])

{

// first check the target is within the ordered range, if not must be

// in a disordered range

if (target < nums[first] || target > nums[middle])

{

first = middle + 1;

}

else

{

last = middle - 1;

}

}

else if (nums[middle] < nums[last])

{

if (target < nums[middle] || target > nums[last])

{

last = middle - 1;

}

else

{

first = middle + 1;

}

}

// otherwise we move the first or the last until we find a disordered one

else if (nums[first] == nums[middle])

{

first++;

}

else if (nums[last] == nums[middle])

{

last--;

}

}

return false;

}

## 153. Find Minimum in Rotated Sorted Array

Medium

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.

You may assume no duplicate exists in the array.

**Example 1:**

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

**Output:** 1

**Example 2:**

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

**Output:** 0

### Analysis:

For a rotated array, you will always have one part is in order and another may or may not be in order. If you see the part which is disordered, then the lowest number must be there, if you do not see any disordered part, it must be in the lower half.

/// <summary>

/// Leet code #153. Find Minimum in Rotated Sorted Array

///

/// 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.

///

/// You may assume no duplicate exists in the array.

///

/// Example 1:

///

/// Input: [3,4,5,1,2]

/// Output: 1

///

/// Example 2:

///

/// Input: [4,5,6,7,0,1,2]

/// Output: 0

/// </summary>

int LeetCodeBinarySearch::findMin(vector<int>& nums)

{

if (nums.size() == 0) return 0;

int first = 0;

int last = nums.size() - 1;

while (first < last)

{

int middle = first + (last - first) / 2;

// If first is greater than last, then the minimum item must be

// within the range

if (nums[first] > nums[middle])

{

last = middle;

}

else if (nums[middle] > nums[last])

{

first = middle + 1;

}

else

{

last = middle;

}

}

return nums[first];

}

## 154. Find Minimum in Rotated Sorted Array II

Hard

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?

### Analysis:

For a rotated array, the minimum number must be either in the disordered part or the first half of the ordered part. If the second half is either disordered or ordered, we know where to find. The only problem is that the fist, middle and last are same, in this case we keep on discard last one, since our job is not to find all the minimum values, until we have the clue.

/// <summary>

/// Leet code #154. 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.

/// Would allow duplicates affect the run-time complexity? How and why?

/// </summary>

int LeetCodeBinarySearch::findMinII(vector<int>& nums)

{

if (nums.size() == 0) return 0;

int first = 0;

int last = nums.size() - 1;

while (first < last)

{

int middle = first + (last - first) / 2;

// The first part disordered, the minimum is in first part

if (nums[first] > nums[middle])

{

last = middle;

}

// The second part is disordered, the minimum is in the second part.

else if (nums[middle] > nums[last])

{

first = middle + 1;

}

// The second part is in order, the minimum is in the first part.

// regardless first part is in order or not.

else if (nums[middle] < nums[last])

{

last = middle;

}

// We can not determine the second part is in order or not, discard last.

else if (nums[middle] == nums[last])

{

last--;

}

}

return nums[first];

}

## 1095. Find in Mountain Array

Hard

*(This problem is an****interactive problem****.)*

You may recall that an array A is a *mountain array* if and only if:

* A.length >= 3
* There exists some i with 0 < i < A.length - 1 such that:
  + A[0] < A[1] < ... A[i-1] < A[i]
  + A[i] > A[i+1] > ... > A[A.length - 1]

Given a mountain array mountainArr, return the **minimum** index such that mountainArr.get(index) == target.  If such an index doesn't exist, return -1.

**You can't access the mountain array directly.**  You may only access the array using a MountainArray interface:

* MountainArray.get(k) returns the element of the array at index k (0-indexed).
* MountainArray.length() returns the length of the array.

Submissions making more than 100 calls to MountainArray.get will be judged *Wrong Answer*.  Also, any solutions that attempt to circumvent the judge will result in disqualification.

**Example 1:**

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

**Output:** 2

**Explanation:** 3 exists in the array, at index=2 and index=5. Return the minimum index, which is 2.

**Example 2:**

**Input:** array = [0,1,2,4,2,1], target = 3

**Output:** -1

**Explanation:** 3 does not exist in the array, so we return -1.

**Constraints:**

1. 3 <= mountain\_arr.length() <= 10000
2. 0 <= target <= 10^9
3. 0 <= mountain\_arr.get(index) <= 10^9

### Analysis:

When you pick a middle point, you first check if this is a uphill or downhill, if this is a uphill and the target is greater than the middle point or if it is a downhill and value is great than the middle point you can discard the first half or second half. If the value is within the uphill range you search in uphill first if not found then search in the second half, or if the value is within the downhill, you search the downhill first, if not found then search the first half.

Please notice that the original problem is represented as a design problem which is not aligned with the latest signature.

/// <summary>

/// Leet code #1095. Find in Mountain Array

///

/// You may recall that an array A is a mountain array if and only if:

/// A.length >= 3

/// There exists some i with 0 < i < A.length - 1 such that:

/// A[0] < A[1] < ... A[i-1] < A[i]

/// A[i] > A[i+1] > ... > A[A.length - 1]

/// Given a mountain array mountainArr, return the minimum index such that

/// mountainArr.get(index) == target. If such an index doesn't exist,

/// return -1.

///

/// You can't access the mountain array directly. You may only access the

/// array using a MountainArray interface:

///

/// MountainArray.get(k) returns the element of the array at index k

/// (0-indexed).

/// MountainArray.length() returns the length of the array.

/// Submissions making more than 100 calls to MountainArray.get will be

/// judged Wrong Answer. Also, any solutions that attempt to circumvent

/// the judge will result in disqualification.

///

/// Example 1:

///

/// Input: array = [1,2,3,4,5,3,1], target = 3

/// Output: 2

/// Explanation: 3 exists in the array, at index=2 and index=5. Return

/// the minimum index, which is 2.

///

/// Example 2:

///

/// Input: array = [0,1,2,4,2,1], target = 3

/// Output: -1

/// Explanation: 3 does not exist in the array, so we return -1.

///

/// Constraints:

/// 1. 3 <= mountain\_arr.length() <= 10000

/// 2. 0 <= target <= 10^9

/// 3. 0 <= mountain\_arr.get(index) <= 10^9

/// </summary>

class MountainArray

{

private:

vector<int> m\_array;

int length()

{

return m\_array.size();

}

int get(int k)

{

return m\_array[k];

}

public:

MountainArray()

{

}

MountainArray(vector<int>& a)

{

m\_array = a;

}

int findInMountainArray(int target, MountainArray& mountainArr, int first, int last)

{

int result = -1;

if (first == last)

{

if (mountainArr.get(first) == target) result = first;

}

else if (first + 1 == last)

{

if (mountainArr.get(first) == target) result = first;

else if (mountainArr.get(last) == target) result = last;

}

else

{

int mid = first + (last - first) / 2;

if (mountainArr.get(mid - 1) < mountainArr.get(mid) &&

mountainArr.get(mid) < mountainArr.get(mid + 1))

{

if (target <= mountainArr.get(mid))

{

result = findInMountainArray(target, mountainArr, first, mid);

}

if (result == -1)

{

result = findInMountainArray(target, mountainArr, mid + 1, last);

}

}

else if (mountainArr.get(mid - 1) > mountainArr.get(mid) &&

mountainArr.get(mid) > mountainArr.get(mid + 1))

{

result = findInMountainArray(target, mountainArr, first, mid);

if (result == -1)

{

if (target <= mountainArr.get(mid + 1))

{

result = findInMountainArray(target, mountainArr, mid + 1, last);

}

}

}

else

{

if (target <= mountainArr.get(mid))

{

result = findInMountainArray(target, mountainArr, first, mid);

}

if (result == -1)

{

if (target <= mountainArr.get(mid + 1))

{

result = findInMountainArray(target, mountainArr, mid + 1, last);

}

}

}

}

return result;

}

int findInMountainArray(int target, MountainArray& mountainArr)

{

int first = 0;

int last = mountainArr.length() - 1;

int result = findInMountainArray(target, mountainArr, first, last);

return result;

}

};

## 240. Search a 2D Matrix II

Medium

Write an efficient algorithm that searches for a value in an *m* x *n* matrix. This matrix has the following properties:

* Integers in each row are sorted in ascending from left to right.
* Integers in each column are sorted in ascending from top to bottom.

**Example:**

Consider the following matrix:

[

[1, 4, 7, 11, 15],

[2, 5, 8, 12, 19],

[3, 6, 9, 16, 22],

[10, 13, 14, 17, 24],

[18, 21, 23, 26, 30]

]

Given target = 5, return true.

Given target = 20, return false.

### Analysis:

For this problem you cannot apply binary search directly, but you can discard what impossible to contain the target. You start from the right upper corner, if it is less than the value, you discard the current column and move one column left, if it is greater than the current value you discard the current row and move to the next row. Keep on doing so until you find the target or move out of the matrix.

/// <summary>

/// Leet code #240. Search a 2D Matrix II

/// Write an efficient algorithm that searches for a value in an m x n matrix.

/// This matrix has the following properties:

/// Integers in each row are sorted in ascending from left to right.

/// Integers in each column are sorted in ascending from top to bottom.

///

/// For example,

/// Consider the following matrix:

/// [

/// [1, 4, 7, 11, 15],

/// [2, 5, 8, 12, 19],

/// [3, 6, 9, 16, 22],

/// [10, 13, 14, 17, 24],

/// [18, 21, 23, 26, 30]

/// ]

///

/// Given target = 5, return true.

/// Given target = 20, return false.

/// </summary>

bool LeetCode::searchMatrixII(vector<vector<int>>& matrix, int target)

{

if ((matrix.size() == 0) || (matrix[0].size() == 0))

{

return false;

}

// starting from the end of first row

int row = 0;

int col = matrix[0].size() - 1;

while ((row < (int)matrix.size()) && (col >= 0))

{

// if target is greater than the specified position, exclude the row

if (matrix[row][col] < target)

{

row++;

}

// if target is greater than the specified position, exclude the column

else if (matrix[row][col] > target)

{

col--;

}

else

{

return true;

}

}

return false;

}

## 4. Median of Two Sorted Arrays

Hard

There are two sorted arrays **nums1** and **nums2** of size m and n respectively.

Find the median of the two sorted arrays. The overall run time complexity should be O(log (m+n)).

You may assume **nums1** and **nums2** cannot be both empty.

**Example 1:**

nums1 = [1, 3]

nums2 = [2]

The median is 2.0

**Example 2:**

nums1 = [1, 2]

nums2 = [3, 4]

The median is (2 + 3)/2 = 2.5

### Analysis:

To get the middle of two arrays, you can say discard another half of the two arrays. If the total number of the element in two arrays are m and n, you should discard (m + n) /2 and (m+n-1)/2 in two process, and then calculate the average between them. Assume you want to discard k from two arrays, you can check each array k/2 and discard the one with higher value which will guarantee to be safe. When one array become empty you can discard K element from another one.

/// <summary>

/// LeetCode #4. Median of Two Sorted Arrays

/// </summary>

int LeetCodeBinarySearch::findMedianSortedArrays(vector<int>& nums1, vector<int>& nums2, size\_t s1, size\_t s2, size\_t k)

{

if (s1 < s2) return findMedianSortedArrays(nums2, nums1, s2, s1, k);

if (s2 == 0) return nums1[s1 - k - 1];

if (k == 0)

{

if (nums1[s1 - 1] >= nums2[s2 - 1]) return nums1[s1 - 1];

else return nums2[s2 - 1];

}

else

{

size\_t n\_k = 1;

if (k > 1) n\_k = k / 2;

n\_k = min(min(n\_k, s1), s2);

if (nums1[s1 - n\_k] < nums2[s2 - n\_k])

{

return findMedianSortedArrays(nums1, nums2, s1, s2 - n\_k, k - n\_k);

}

else

{

return findMedianSortedArrays(nums1, nums2, s1 - n\_k, s2, k - n\_k);

}

}

}

/// <summary>

/// LeetCode #4 Median of Two Sorted Arrays

///

/// There are two sorted arrays nums1 and nums2 of size m and n respectively.

/// Find the median of the two sorted arrays.The overall run time complexity

/// should be O(log(m + n)).

///

/// Example 1:

/// num1 = [1, 3]

/// nums2 = [2]

/// The median is 2.0

///

/// Example 2:

/// nums1 = [1, 2]

/// nums2 = [3, 4]

/// The median is (2 + 3)/2 = 2.5

/// </summary>

double LeetCodeBinarySearch::findMedianSortedArrays(vector<int>& nums1, vector<int>& nums2)

{

double value;

// transfer the problem to find the kth item (middle point) in two arrays

size\_t size = nums1.size() + nums2.size();

value = (double)findMedianSortedArrays(nums1, nums2, nums1.size(), nums2.size(), size / 2);

value += (double)findMedianSortedArrays(nums1, nums2, nums1.size(), nums2.size(), (size - 1) / 2);

return value / 2;

}