# Leetcode 题解 - 二分查找

* [Leetcode 题解 - 二分查找](#leetcode-题解---二分查找)
  + [1. 求开方](#1-求开方)
  + [2. 大于给定元素的最小元素](#2-大于给定元素的最小元素)
  + [3. 有序数组的 Single Element](#3-有序数组的-single-element)
  + [4. 第一个错误的版本](#4-第一个错误的版本)
  + [5. 旋转数组的最小数字](#5-旋转数组的最小数字)
  + [6. 查找区间](#6-查找区间)

**正常实现**

Input : [1,2,3,4,5]  
key : 3  
return the index : 2

public int binarySearch(int[] nums, int key) {  
 int l = 0, h = nums.length - 1;  
 while (l <= h) {  
 int m = l + (h - l) / 2;  
 if (nums[m] == key) {  
 return m;  
 } else if (nums[m] > key) {  
 h = m - 1;  
 } else {  
 l = m + 1;  
 }  
 }  
 return -1;  
}

**时间复杂度**

二分查找也称为折半查找，每次都能将查找区间减半，这种折半特性的算法时间复杂度为 O(logN)。

**m 计算**

有两种计算中值 m 的方式：

* m = (l + h) / 2
* m = l + (h - l) / 2

l + h 可能出现加法溢出，也就是说加法的结果大于整型能够表示的范围。但是 l 和 h 都为正数，因此 h - l 不会出现加法溢出问题。所以，最好使用第二种计算法方法。

**未成功查找的返回值**

循环退出时如果仍然没有查找到 key，那么表示查找失败。可以有两种返回值：

* -1：以一个错误码表示没有查找到 key
* l：将 key 插入到 nums 中的正确位置

**变种**

二分查找可以有很多变种，实现变种要注意边界值的判断。例如在一个有重复元素的数组中查找 key 的最左位置的实现如下：

public int binarySearch(int[] nums, int key) {  
 int l = 0, h = nums.length;  
 while (l < h) {  
 int m = l + (h - l) / 2;  
 if (nums[m] >= key) {  
 h = m;  
 } else {  
 l = m + 1;  
 }  
 }  
 return l;  
}

该实现和正常实现有以下不同：

* h 的赋值表达式为 h = m
* 循环条件为 l < h
* 最后返回 l 而不是 -1

在 nums[m] >= key 的情况下，可以推导出最左 key 位于 [l, m] 区间中，这是一个闭区间。h 的赋值表达式为 h = m，因为 m 位置也可能是解。

在 h 的赋值表达式为 h = m 的情况下，如果循环条件为 l <= h，那么会出现循环无法退出的情况，因此循环条件只能是 l < h。以下演示了循环条件为 l <= h 时循环无法退出的情况：

nums = {0, 1, 2}, key = 1  
l m h  
0 1 2 nums[m] >= key  
0 0 1 nums[m] < key  
1 1 1 nums[m] >= key  
1 1 1 nums[m] >= key  
...

当循环体退出时，不表示没有查找到 key，因此最后返回的结果不应该为 -1。为了验证有没有查找到，需要在调用端判断一下返回位置上的值和 key 是否相等。

## 1. 求开方

69. Sqrt(x) (Easy)

[Leetcode](https://leetcode.com/problems/sqrtx/description/) / [力扣](https://leetcode-cn.com/problems/sqrtx/description/)

Input: 4  
Output: 2  
  
Input: 8  
Output: 2  
Explanation: The square root of 8 is 2.82842..., and since we want to return an integer, the decimal part will be truncated.

一个数 x 的开方 sqrt 一定在 0 ~ x 之间，并且满足 sqrt == x / sqrt。可以利用二分查找在 0 ~ x 之间查找 sqrt。

对于 x = 8，它的开方是 2.82842...，最后应该返回 2 而不是 3。在循环条件为 l <= h 并且循环退出时，h 总是比 l 小 1，也就是说 h = 2，l = 3，因此最后的返回值应该为 h 而不是 l。

public int mySqrt(int x) {  
 if (x <= 1) {  
 return x;  
 }  
 int l = 1, h = x;  
 while (l <= h) {  
 int mid = l + (h - l) / 2;  
 int sqrt = x / mid;  
 if (sqrt == mid) {  
 return mid;  
 } else if (mid > sqrt) {  
 h = mid - 1;  
 } else {  
 l = mid + 1;  
 }  
 }  
 return h;  
}

## 2. 大于给定元素的最小元素

744. Find Smallest Letter Greater Than Target (Easy)

[Leetcode](https://leetcode.com/problems/find-smallest-letter-greater-than-target/description/) / [力扣](https://leetcode-cn.com/problems/find-smallest-letter-greater-than-target/description/)

Input:  
letters = ["c", "f", "j"]  
target = "d"  
Output: "f"  
  
Input:  
letters = ["c", "f", "j"]  
target = "k"  
Output: "c"

题目描述：给定一个有序的字符数组 letters 和一个字符 target，要求找出 letters 中大于 target 的最小字符，如果找不到就返回第 1 个字符。

public char nextGreatestLetter(char[] letters, char target) {  
 int n = letters.length;  
 int l = 0, h = n - 1;  
 while (l <= h) {  
 int m = l + (h - l) / 2;  
 if (letters[m] <= target) {  
 l = m + 1;  
 } else {  
 h = m - 1;  
 }  
 }  
 return l < n ? letters[l] : letters[0];  
}

## 3. 有序数组的 Single Element

540. Single Element in a Sorted Array (Medium)

[Leetcode](https://leetcode.com/problems/single-element-in-a-sorted-array/description/) / [力扣](https://leetcode-cn.com/problems/single-element-in-a-sorted-array/description/)

Input: [1, 1, 2, 3, 3, 4, 4, 8, 8]  
Output: 2

题目描述：一个有序数组只有一个数不出现两次，找出这个数。

要求以 O(logN) 时间复杂度进行求解，因此不能遍历数组并进行异或操作来求解，这么做的时间复杂度为 O(N)。

令 index 为 Single Element 在数组中的位置。在 index 之后，数组中原来存在的成对状态被改变。如果 m 为偶数，并且 m + 1 < index，那么 nums[m] == nums[m + 1]；m + 1 >= index，那么 nums[m] != nums[m + 1]。

从上面的规律可以知道，如果 nums[m] == nums[m + 1]，那么 index 所在的数组位置为 [m + 2, h]，此时令 l = m + 2；如果 nums[m] != nums[m + 1]，那么 index 所在的数组位置为 [l, m]，此时令 h = m。

因为 h 的赋值表达式为 h = m，那么循环条件也就只能使用 l < h 这种形式。

public int singleNonDuplicate(int[] nums) {  
 int l = 0, h = nums.length - 1;  
 while (l < h) {  
 int m = l + (h - l) / 2;  
 if (m % 2 == 1) {  
 m--; // 保证 l/h/m 都在偶数位，使得查找区间大小一直都是奇数  
 }  
 if (nums[m] == nums[m + 1]) {  
 l = m + 2;  
 } else {  
 h = m;  
 }  
 }  
 return nums[l];  
}

## 4. 第一个错误的版本

278. First Bad Version (Easy)

[Leetcode](https://leetcode.com/problems/first-bad-version/description/) / [力扣](https://leetcode-cn.com/problems/first-bad-version/description/)

题目描述：给定一个元素 n 代表有 [1, 2, ..., n] 版本，在第 x 位置开始出现错误版本，导致后面的版本都错误。可以调用 isBadVersion(int x) 知道某个版本是否错误，要求找到第一个错误的版本。

如果第 m 个版本出错，则表示第一个错误的版本在 [l, m] 之间，令 h = m；否则第一个错误的版本在 [m + 1, h] 之间，令 l = m + 1。

因为 h 的赋值表达式为 h = m，因此循环条件为 l < h。

public int firstBadVersion(int n) {  
 int l = 1, h = n;  
 while (l < h) {  
 int mid = l + (h - l) / 2;  
 if (isBadVersion(mid)) {  
 h = mid;  
 } else {  
 l = mid + 1;  
 }  
 }  
 return l;  
}

## 5. 旋转数组的最小数字

153. Find Minimum in Rotated Sorted Array (Medium)

[Leetcode](https://leetcode.com/problems/find-minimum-in-rotated-sorted-array/description/) / [力扣](https://leetcode-cn.com/problems/find-minimum-in-rotated-sorted-array/description/)

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

public int findMin(int[] nums) {  
 int l = 0, h = nums.length - 1;  
 while (l < h) {  
 int m = l + (h - l) / 2;  
 if (nums[m] <= nums[h]) {  
 h = m;  
 } else {  
 l = m + 1;  
 }  
 }  
 return nums[l];  
}

## 6. 查找区间

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

[Leetcode](https://leetcode.com/problems/find-first-and-last-position-of-element-in-sorted-array/) / [力扣](https://leetcode-cn.com/problems/find-first-and-last-position-of-element-in-sorted-array/)

Input: nums = [5,7,7,8,8,10], target = 8  
Output: [3,4]  
  
Input: nums = [5,7,7,8,8,10], target = 6  
Output: [-1,-1]

题目描述：给定一个有序数组 nums 和一个目标 target，要求找到 target 在 nums 中的第一个位置和最后一个位置。

可以用二分查找找出第一个位置和最后一个位置，但是寻找的方法有所不同，需要实现两个二分查找。我们将寻找 target 最后一个位置，转换成寻找 target+1 第一个位置，再往前移动一个位置。这样我们只需要实现一个二分查找代码即可。

public int[] searchRange(int[] nums, int target) {  
 int first = findFirst(nums, target);  
 int last = findFirst(nums, target + 1) - 1;  
 if (first == nums.length || nums[first] != target) {  
 return new int[]{-1, -1};  
 } else {  
 return new int[]{first, Math.max(first, last)};  
 }  
}  
  
private int findFirst(int[] nums, int target) {  
 int l = 0, h = nums.length; // 注意 h 的初始值  
 while (l < h) {  
 int m = l + (h - l) / 2;  
 if (nums[m] >= target) {  
 h = m;  
 } else {  
 l = m + 1;  
 }  
 }  
 return l;  
}

在寻找第一个位置的二分查找代码中，需要注意 h 的取值为 nums.length，而不是 nums.length - 1。先看以下示例：

nums = [2,2], target = 2

如果 h 的取值为 nums.length - 1，那么 last = findFirst(nums, target + 1) - 1 = 1 - 1 = 0。这是因为 findLeft 只会返回 [0, nums.length - 1] 范围的值，对于 findFirst([2,2], 3) ，我们希望返回 3 插入 nums 中的位置，也就是数组最后一个位置再往后一个位置，即 nums.length。所以我们需要将 h 取值为 nums.length，从而使得 findFirst返回的区间更大，能够覆盖 target 大于 nums 最后一个元素的情况。