LeetCode 69. x 的平方根

69. x 的平方根

难度 简单 319

实现 `int sqrt(int x)` 函数。

计算并返回 *x* 的平方根，其中 *x* 是非负整数。

由于返回类型是整数，结果只保留整数的部分，小数部分将被舍去。

示例 1:

```
输入：4
输出：2
```

示例 2:

```
输入：8
输出：2
说明：8 的平方根是 2.82842...,
     由于返回类型是整数，小数部分将被舍去。
```

```cpp
/*
二分流程
1.确定二分边界
2.编写二分代码框架
3.设计一个check函数(性质)
4.判断一下区间如何更新
5.如果更新方式是l=mid，r=mid-1，那么就在算mid的时候加1
*/
class Solution {
public:
    int mySqrt(int x) {
        int l = 0,r = x;
        while(l < r)
        {
            int mid = l + r + 1ll >> 1;//1ll表示1ll是long long类型的1
            if(mid <= x / mid) l = mid;
            else r = mid - 1;
        }
        return r;
    }
};
```

LeetCode 35. 搜索插入位置

35. 搜索插入位置

难度 简单 446

给定一个排序数组和一个目标值，在数组中找到目标值，并返回其索引。如果目标值不存在于数组中，返回它将会被按顺序插入的位置。

你可以假设数组中无重复元素。

示例 1:

```
输入: [1,3,5,6], 5
输出: 2
```

示例 2:

```
输入: [1,3,5,6], 2
输出: 1
```

示例 3:

```
输入: [1,3,5,6], 7
输出: 4
```

示例 4:

```
输入: [1,3,5,6], 0
输出: 0
```

```cpp
class Solution {
public:
    int searchInsert(vector<int>& nums, int target) {
        int n = nums.size();
        if (n == 0)
            return 0;
        int l = 0, r = n - 1;
        while (l < r) {
            int mid = (l + r) >> 1;
            if (nums[mid] < target)
                l = mid + 1;
            else
                r = mid;
        }

        if (nums[l] >= target)
            return l;

        return n;
    }
};
**********************************************************************
class Solution {
public:
    int searchInsert(vector<int>& nums, int target) {
```

```
        int l = 0,r = nums.size();
        while(l < r)
        {
            int mid = l + r >> 1;
            if(nums[mid] >= target) r = mid;
            else l = mid + 1;
        }
        return l;
    }
};
```

LeetCode 34. 在排序数组中查找元素的第一个和最后一个位置

34. 在排序数组中查找元素的第一个和最后一个位置

难度 中等 337

给定一个按照升序排列的整数数组 `nums`，和一个目标值 `target`。找出给定目标值在数组中的开始位置和结束位置。

你的算法时间复杂度必须是 *O*(log *n*) 级别。

如果数组中不存在目标值，返回 `[-1, -1]`。

示例 1:

```
输入: nums = [5,7,7,8,8,10], target = 8
输出: [3,4]
```

示例 2:

```
输入: nums = [5,7,7,8,8,10], target = 6
输出: [-1,-1]
```

```
class Solution {
public:
    vector<int> searchRange(vector<int>& nums, int target) {
        if(nums.empty()) return {-1,-1};

        int l = 0,r = nums.size() - 1;
        while(l < r)
        {
            int mid = l + r >> 1;
            if(nums[mid] >= target) r = mid;
            else l = mid + 1;
        }

        if(nums[r] != target) return {-1,-1};
        int strat = r;
        l = 0,r = nums.size() - 1;
        while(l  < r)
        {
```

```
            int mid = l + r + 1 >> 1;
            if(nums[mid] <= target) l = mid;
            else r = mid - 1;
        }
        int end = l;
        return {strat,end};
    }
};
```

LeetCode 74. 搜索二维矩阵

74. 搜索二维矩阵

难度 中等 140

编写一个高效的算法来判断 *m* x *n* 矩阵中，是否存在一个目标值。该矩阵具有如下特性：

- 每行中的整数从左到右按升序排列。
- 每行的第一个整数大于前一行的最后一个整数。

示例 1:

```
输入：
matrix = [
  [1,   3,  5,  7],
  [10, 11, 16, 20],
  [23, 30, 34, 50]
]
target = 3
输出: true
```

示例 2:

```
输入：
matrix = [
  [1,   3,  5,  7],
  [10, 11, 16, 20],
  [23, 30, 34, 50]
]
target = 13
输出: false
```

```
/*
!!!!!!!!!!!!!!!!!!!!!!!!!!!!
二维坐标(x,y)，方阵n
一维t
二维数组下标变为一维数组下标----------t = x * n + y;
一维数组下标变为二维数组下标----------x = t / n;y = t % n;
*/
class Solution {
public:
    bool searchMatrix(vector<vector<int>>& matrix, int target) {
```

```
        if(matrix.empty() || matrix[0].empty()) return false;

        int n = matrix.size(),m = matrix[0].size();
        int l = 0,r = n * m - 1;
        while(l < r)
        {
            int mid = l + r >> 1;
            if(matrix[mid / m][mid % m] >= target) r = mid;//一维数组下标变为二维
            else l = mid + 1;
        }
        if(matrix[l/ m][l % m] != target) return false;
        return true;
    }
};
```

LeetCode 153. 寻找旋转排序数组中的最小值

153. 寻找旋转排序数组中的最小值

难度 中等 148

假设按照升序排序的数组在预先未知的某个点上进行了旋转。

(例如，数组 `[0,1,2,4,5,6,7]` 可能变为 `[4,5,6,7,0,1,2]`)。

请找出其中最小的元素。

你可以假设数组中不存在重复元素。

示例 1:

```
输入：[3,4,5,1,2]
输出：1
```

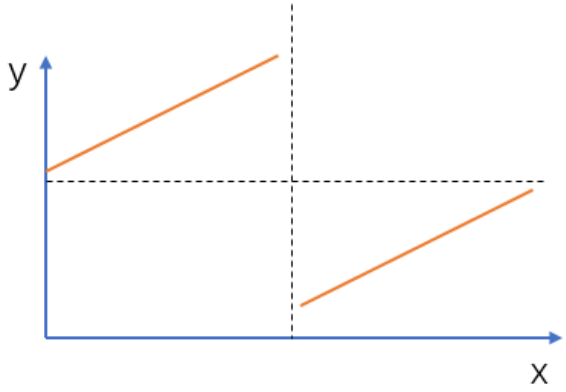
示例 2:

```
输入：[4,5,6,7,0,1,2]
输出：0
```

(二分) $O(logn)$

处理这种问题有个常用技巧：如果不想处理边界情况，比如当数组只有两三个数的时候，代码会出问题。我们可以在数组长度太短(这道题中我们判断数组长度小于5)时，直接暴力循环做；数组有一定长度时再用二分做。
这样做并不会影响算法的时间复杂度，但会缩短写代码的时间。

为了便于理解，我们将数组中的数画在二维坐标系中，横坐标表示数组下标，纵坐标表示数值，如下所示：



我们会发现数组中最小值前面的数 $nums[i]$ 都满足：$nums[i] \geq nums[0]$，其中 $nums[n-1]$ 是数组最后一个元素；而数组中最小值后面的数（包括最小值）都不满足这个条件。
所以我们可以二分出最小值的位置。

另外，不要忘记处理数组完全单调的特殊情况。

时间复杂度分析：二分查找，所以时间复杂度是 $O(logn)$。

```
/*
找出最小值
check(t)
    return nums[t] <= nums.back()
*/
class Solution {
public:
    int findMin(vector<int>& nums) {
        if (nums.back() > nums[0]) return nums[0];
        int l = 0, r = nums.size() - 1;
        while (l < r)
        {
            int mid = l + r >> 1;
            if (nums[mid] >= nums[0]) l = mid + 1;
            else r = mid;
        }
        return nums[l];
    }
};
*********************************************************************************
class Solution {
public:
    int findMin(vector<int>& nums) {
        int l = 0,r = nums.size() - 1;
        while(l < r)
        {
            int mid = l + r >> 1;
            if(nums[mid] <= nums.back()) r = mid;
            else l = mid + 1;
        }
```

```
        return nums[r];
    }
};
```

LeetCode 33. 搜索旋转排序数组

33. 搜索旋转排序数组

难度 中等 553

假设按照升序排序的数组在预先未知的某个点上进行了旋转。

(例如，数组 `[0,1,2,4,5,6,7]` 可能变为 `[4,5,6,7,0,1,2]`)。

搜索一个给定的目标值，如果数组中存在这个目标值，则返回它的索引，否则返回 `-1` 。

你可以假设数组中不存在重复的元素。

你的算法时间复杂度必须是 *O*(log *n*) 级别。

示例 1:

```
输入: nums = [4,5,6,7,0,1,2], target = 0
输出: 4
```

示例 2:

```
输入: nums = [4,5,6,7,0,1,2], target = 3
输出: -1
```

(三次二分检索) $O(logn)$

1. 数组长度为 `0` 和 `1` 时的特判处理。
2. 首先二分出是以哪个元素分割数组两部分的。
3. 具体为：每次二分时，如果 `nums[mid] >= nums[l] && nums[mid] >= nums[r]` ，则 `l = mid + 1` ；如果 `nums[mid] <= nums[l] && nums[mid] <= nums[r]` ，则 `r = mid` ；否则 `break` 。最终数组分为 `[0, l - 1]` 和 `[l, n - 1]` 两段区间。
4. 然后再在两段区间分别二分找 `target` 即可。

```
/*
找出目标值
*/
class Solution {
public:
    int search(vector<int>& nums, int target) {
        int n = nums.size();
        if (n == 0)
            return -1;
        int l = 0, r = n - 1;
        while (l < r) {
            int mid = (l + r) >> 1;
            if (nums[mid] >= nums[0]) { // mid 在数组前半部分。
                if (target > nums[mid])
```

```
                    // 可以推出 target 的值一定大于 nums[0]，target 只可能在 [mid +
1, r] 中。
                    l = mid + 1;
                if (target < nums[0])
                    // 可以推出 target 的值一定小于 nums[mid]，target只可能在 [mid +
1, r] 中。
                    l = mid + 1;
                if (target <= nums[mid] && target >= nums[0])
                    // 此时 target 的值处于 nums[0] 和 nums[mid] 中，故可能在 [l,
mid] 中。
                    r = mid;
            }
            else { // mid在数组后半部分
                if (target >= nums[0])
                    // 可以推出 target 的值一定大于 nums[mid]，target只可能在 [l,
mid] 中。
                    r = mid;
                if (target <= nums[mid])
                    // 可以推出 target 的值一定小于 nums[0]，target只可能在 [l, mid]
中。
                    r = mid;
                if (target > nums[mid] && target < nums[0])
                    // 此时 target 的值处于 nums[0] 和 nums[mid] 中，故可能在 [mid +
1, r] 中。
                    l = mid + 1;
            }
        }
        return nums[l] == target ? l : -1;
    }
};
*****************************************************************************
class Solution {
public:
    int search(vector<int>& nums, int target) {
        if(nums.empty()) return -1;

        int l = 0,r = nums.size() - 1;
        while(l < r)
        {
            int mid = l + r >> 1;
            if(nums[mid] <= nums.back()) r = mid;
            else l = mid + 1;
        }

        if(target <= nums.back()) r = nums.size() - 1;
        else l = 0,r --;

        while(l < r)
        {
            int mid = l + r >> 1;
            if(nums[mid] >= target) r = mid;
            else l = mid + 1;
        }

        if(nums[l] == target) return l;
        return -1;
    }
};
```

LeetCode 278.第一个错误的版本

278. 第一个错误的版本

难度 简单 143

你是产品经理，目前正在带领一个团队开发新的产品。不幸的是，你的产品的最新版本没有通过质量检测。由于每个版本都是基于之前的版本开发的，所以错误的版本之后的所有版本都是错的。

假设你有 `n` 个版本 `[1, 2, ..., n]`，你想找出导致之后所有版本出错的第一个错误的版本。

你可以通过调用 `bool isBadVersion(version)` 接口来判断版本号 `version` 是否在单元测试中出错。实现一个函数来查找第一个错误的版本。你应该尽量减少对调用 API 的次数。

示例:

```
给定 n = 5，并且 version = 4 是第一个错误的版本。

调用 isBadVersion(3) -> false
调用 isBadVersion(5) -> true
调用 isBadVersion(4) -> true

所以，4 是第一个错误的版本。
```

```cpp
// Forward declaration of isBadVersion API.
bool isBadVersion(int version);

class Solution {
public:
    int firstBadVersion(int n) {
        long long l = 0,r = n;
        while(l < r)
        {
            int mid = l + r + 0ll >> 1;
            if(isBadVersion(mid)) r = mid;
            else l = mid + 1;
        }
        return r;
    }
};
```

LeetCode 162. 寻找峰值

162. 寻找峰值

难度 中等 158

峰值元素是指其值大于左右相邻值的元素。

给定一个输入数组 `nums`，其中 `nums[i] ≠ nums[i+1]`，找到峰值元素并返回其索引。

数组可能包含多个峰值，在这种情况下，返回任何一个峰值所在位置即可。

你可以假设 `nums[-1] = nums[n] = -∞`。

示例 1:

```
输入: nums = [1,2,3,1]
输出: 2
解释: 3 是峰值元素，你的函数应该返回其索引 2。
```

示例 2:

```
输入: nums = [1,2,1,3,5,6,4]
输出: 1 或 5
解释: 你的函数可以返回索引 1，其峰值元素为 2；
     或者返回索引 5， 其峰值元素为 6。
```

说明:

你的解法应该是 *O*(*logN*) 时间复杂度的。

(二分) $O(logn)$

仔细分析我们会发现：

- 如果 `nums[i-1] < nums[i]`，则如果 `nums[i-1], nums[i], ... nums[n-1]` 是单调的，则 `nums[n-1]` 就是峰值；如果 `nums[i-1], nums[i], ... nums[n-1]` 不是单调的，则从 i 开始，第一个满足 `nums[i] > nums[i+1]` 的 i 就是峰值；所以 $[i, n-1]$ 中一定包含一个峰值；
- 如果 `nums[i-1] > nums[i]`，同理可得 $[0, i-1]$ 中一定包含一个峰值；

所以我们可以每次二分中点，通过判断 `nums[i-1]` 和 `nums[i]` 的大小关系，可以判断左右两边哪边一定有峰值，从而可以将检索区间缩小一半。

时间复杂度分析：二分检索，每次删掉一半元素，所以时间复杂度是 $O(logn)$。

```cpp
/*
两边都是负无穷
峰值：比两边都大
暴力O(n)
二分O(logn)
找中点，若中点比下一个点小，则峰值一定在后面，否则在前面
*/
class Solution {
public:
    int findPeakElement(vector<int>& nums) {
        int l = 0, r = nums.size() - 1;
        while (l < r)
```

```
        {
            int mid = (l + r + 1) / 2;
            if (nums[mid] > nums[mid - 1]) l = mid;
            else r = mid - 1;
        }
        return l;
    }
};
**********************************************************************
class Solution {
public:
    int findPeakElement(vector<int>& nums) {
        int l = 0,r = nums.size() - 1;
        while(l < r)
        {
            int mid = l + r >> 1;
            if(nums[mid] > nums[mid + 1]) r = mid;
            else l = mid + 1;
        }
        return r;
    }
};
```

LeetCode 287. 寻找重复数

287. 寻找重复数

难度 中等 448

给定一个包含 *n* + 1 个整数的数组 *nums*，其数字都在 1 到 *n* 之间（包括 1 和 *n*），可知至少存在一个重复的整数。假设只有一个重复的整数，找出这个重复的数。

示例 1:

```
输入: [1,3,4,2,2]
输出: 2
```

示例 2:

```
输入: [3,1,3,4,2]
输出: 3
```

说明:

1. **不能**更改原数组（假设数组是只读的）。
2. 只能使用额外的 *O*(1) 的空间。
3. 时间复杂度小于 $O(n^2)$ 。
4. 数组中只有一个重复的数字，但它可能不止重复出现一次。

(双指针移动) $O(n)$

1. 因为每个数都是 1 到 n，所以此题可以当做 Linked List Cycle II 来处理，即 `i` 位置的 `val` 和 `next` 都是 `nums[i]`。
2. 首先 `first` 和 `second` 指针均指向 `0` 位置，然后 `first` 每次前进一次，`second` 每次前进两次。剩余部分请参考 Linked List Cycle II 中的算法证明。

```
class Solution {
public:
    int findDuplicate(vector<int>& nums) {
        int first = 0, second = 0;
        do {
            first = nums[first];
            second = nums[nums[second]];
        } while (first != second);

        first = 0;
        while (first != second) {
            first = nums[first];
            second = nums[second];
        }
        return first;
    }
};
********************************************************************************
***
/*
抽屉原理
二分中点
计算左右两区间的数的个数，不可能有两边同时小于苹果个数的情况
*/
class Solution {
public:
    int findDuplicate(vector<int>& nums) {
        int l = 1,r = nums.size() - 1;
        while(l < r)
        {
            int mid = l + r >> 1;
            int cnt = 0;
            for(auto x : nums)
                if(x >= l && x <= mid)
                    cnt ++;

            if(cnt > mid - l + 1) r = mid;
            else l = mid + 1;
        }
        return r;
    }
};
```

LeetCode 275. H指数 II

275. H指数 II

难度 中等 34

给定一位研究者论文被引用次数的数组（被引用次数是非负整数），数组已经按照**升序排列**。编写一个方法，计算出研究者的 *h* 指数。

h 指数的定义: "h 代表"高引用次数"（high citations），一名科研人员的 h 指数是指他（她）的 （N 篇论文中）**至多**有 h 篇论文分别被引用了**至少** h 次。（其余的 *N* - *h* 篇论文每篇被引用次数**不多于** *h* 次。）"

示例:

```
输入: citations = [0,1,3,5,6]
输出: 3
解释: 给定数组表示研究者总共有 5 篇论文，每篇论文相应的被引用了 0, 1, 3, 5, 6 次。
     由于研究者有 3 篇论文每篇至少被引用了 3 次，其余两篇论文每篇被引用不多于 3 次，所以她的 h 指数是 3。
```

(二分) $O(logn)$

由于数组是从小到大排好序的，所以我们的任务是：
在数组中找一个最大的 h，使得后 h 个数大于等于 h。

我们发现：如果 h 满足，则小于 h 的数都满足；如果 h 不满足，则大于 h 的数都不满足。所以具有二分性质。
直接二分即可。

时间复杂度分析：二分检索，只遍历 $logn$ 个元素，所以时间复杂度是 $O(logn)$。

```cpp
/*
找到一个h使得至少存在h个数大于等于h，h最大为多少。
h不一定在数组里
0 <= h <= n
具有二分性质
*/
class Solution {
public:
    int hIndex(vector<int>& citations) {
        if (citations.empty()) return 0;
        int l = 0, r = citations.size() - 1;
        while (l < r)
        {
            int mid = (l + r) / 2;
            if (citations.size() - mid <= citations[mid]) r = mid;
            else l = mid + 1;
        }
        if (citations.size() - l <= citations[l]) return citations.size() - l;
        return 0;
    }
};
```

```
*****************************************************************************************
***
class Solution {
public:
    int hIndex(vector<int>& citations) {
        if(citations.empty()) return 0;
        int l = 0,r = citations.size();
        while(l < r)
        {
            int mid = l + r + 1>> 1;
            if(citations[citations.size() - mid] >= mid) l = mid;
            else r = mid - 1;
        }
        return r;
    }
};
```