704.二分查找（简单）

1. 题目描述

   给定一个 `n` 个元素有序的（升序）整型数组 `nums` 和一个目标值 `target` ，写一个函数搜索 `nums` 中的 `target`，如果目标值存在返回下标，否则返回 `-1`。 **示例 1:**

   ```
   输入: nums = [-1,0,3,5,9,12], target = 9
   输出: 4
   解释: 9 出现在 nums 中并且下标为 4
   ```

   示例 2:

   ```
   输入: nums = [-1,0,3,5,9,12], target = 2
   输出: -1
   解释: 2 不存在 nums 中因此返回 -1
   ```

   提示:

   1. 你可以假设 `nums` 中的所有元素是不重复的。
   2. `n` 将在 `[1, 10000]` 之间。
   3. `nums` 的每个元素都将在 `[-9999, 9999]` 之间。

2. 简单实现

```cpp
class Solution {
public:
    int search(vector<int>& nums, int target) {
        int l = 0;
        int r = nums.size() - 1;
        while(l <= r){
            int mid = l + (r - l) / 2;//(l+r)/2可能溢出！！！
            if(nums[mid] == target)
                return mid;
            else if(nums[mid] < target)
                l = mid + 1;
            else
                r = mid - 1;
        }
        return -1;
    }
};
```

识别和模板简介

如何识别二分查找?

如前所述，二分查找是一种在每次比较之后*将查找空间一分为二*的算法。每次需要查找集合中的索引或元素时，都应该考虑二分查找。如果集合是无序的，我们可以总是在应用二分查找之前先对其进行排序。

成功的二分查找的 3 个部分

二分查找一般由三个主要部分组成：

1. ***预处理*** —— 如果集合未排序，则进行排序。
2. **二分查找** —— 使用循环或递归在每次比较后将查找空间划分为两半。
3. **后处理** —— 在剩余空间中确定可行的候选者。

3 个二分查找模板

当我们第一次学会二分查找时，我们可能会挣扎。我们可能会在网上研究数百个二分查找问题，每次我们查看开发人员的代码时，它的实现似乎都略有不同。尽管每个实现在每个步骤中都会将问题空间划分为原来的 1/2，但其中有许多问题：

- 为什么执行方式略有不同？
- 开发人员在想什么？
- 哪种方法更容易？
- 哪种方法更好？

经过许多次失败的尝试并拉扯掉大量的头发后，我们找到了三个主要的二分查找模板。为了防止脱发，并使新的开发人员更容易学习和理解，我们在接下来的章节中提供了它们。

二分查找模板 I

模板 #1：

```
int binarySearch(vector<int>& nums, int target){
  if(nums.size() == 0)
    return -1;

  int left = 0, right = nums.size() - 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;
}
```

模板 #1 是二分查找的最基础和最基本的形式。这是一个标准的二分查找模板，大多数高中或大学会在他们第一次教学生计算机科学时使用。模板 #1 用于查找可以通过*访问数组中的单个索引*来确定的元素或条件。

关键属性

- 二分查找的最基础和最基本的形式。
- 查找条件可以在不与元素的两侧进行比较的情况下确定（或使用它周围的特定元素）。
- 不需要后处理，因为每一步中，你都在检查是否找到了元素。如果到达末尾，则知道未找到该元素。

区分语法

- 初始条件：`left = 0, right = length-1`
- 终止：`left > right`
- 向左查找：`right = mid-1`
- 向右查找：`left = mid+1`

69.x 的平方根（简单）

---

1. 题目描述

   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。
   ```

2. 简单实现

   注意两数相乘可能溢出，要使用long long

   ```
   class Solution {
   public:
       int mySqrt(int x) {
           int l = 0;
           int r = x;
           while(l <= r){
               long long mid = l + (r - l) / 2;
               long long temp = mid*mid;
               if(temp <= x && temp+2*mid+1 > x)
                   return mid;
               else if(temp < x)
                   l = mid + 1;
               else
                   r = mid - 1;
   ```

```
        }
        return -1;
    }
};
```

374.猜数字大小（简单）

1. 题目描述

   我们正在玩一个猜数字游戏。 游戏规则如下： 我从 **1** 到 ***n*** 选择一个数字。 你需要猜我选择了哪个数字。 每次你猜错了，我会告诉你这个数字是大了还是小了。 你调用一个预先定义好的接口 `guess(int num)`，它会返回 3 个可能的结果（`-1`，`1` 或 `0`）：

   ```
   -1 : 我的数字比较小
    1 : 我的数字比较大
    0 : 恭喜！你猜对了！
   ```

   示例：

   ```
   输入: n = 10, pick = 6
   输出: 6
   ```

2. 简单实现

```
// Forward declaration of guess API.
// @param num, your guess
// @return -1 if my number is lower, 1 if my number is higher, otherwise return 0
int guess(int num);
class Solution {
public:
    int guessNumber(int n) {
        int l = 1;
        int r = n;
        while(l <= r){
            int mid = l + (r - l) / 2;
            int re = guess(mid);
            if(re == 0)
                return mid;
            else if(re > 0)
                l = mid + 1;
            else
                r = mid - 1;
        }
         return -1;
    }
};
```

33.搜索旋转排序数组（中等）

1. 题目描述

   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
   ```

2. 简单实现

   分情况讨论，主要是把情况枚举全，判断target/mid分别在左右半区的情况

   ```
   
   ```

二分查找模板 II

模板 #2:

```
int binarySearch(vector<int>& nums, int target){
  if(nums.size() == 0)
    return -1;

  int left = 0, right = nums.size();
  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.size() && nums[left] == target) return left;
  return -1;
}
```

模板 #2 是二分查找的高级模板。它用于查找需要*访问数组中当前索引及其直接右邻居索引*的元素或条件。

关键属性

- 一种实现二分查找的高级方法。
- 查找条件需要访问元素的直接右邻居。
- 使用元素的右邻居来确定是否满足条件，并决定是向左还是向右。
- 保证查找空间在每一步中至少有 2 个元素。
- 需要进行后处理。 当你剩下 1 个元素时，循环 / 递归结束。 需要评估剩余元素是否符合条件。

区分语法

- 初始条件：`left = 0, right = length`
- 终止：`left == right`
- 向左查找：`right = mid`
- 向右查找：`left = mid+1`

278.第一个错误的版本（简单）

---

1. 题目描述

   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。
   ```

2. 简单实现

不难看出，这道题可以用经典的二分查找算法求解。我们通过一个例子，来说明二分查找如何在每次操作中减少一半的搜索空间，以此减少时间复杂度。

```
场景一：isBadVersion(mid) => false

 1 2 3 4 5 6 7 8 9
 G G G G G G B B B       G = 正确版本，B = 错误版本
 |       |       |
left    mid    right
```

场景一中，`isBadVersion(mid)` 返回 false，因此我们知道 mid 左侧（包括自身）的所有版本都是正确的。所以我们令 $\text{left} = \text{mid} + 1$，把下一次的搜索空间变为 $[\text{mid} + 1, \text{right}]$。

```
场景二：isBadVersion(mid) => true

 1 2 3 4 5 6 7 8 9
 G G G B B B B B B       G = 正确版本，B = 错误版本
 |       |       |
left    mid    right
```

场景二中，`isBadVersion(mid)` 返回 true，因此我们知道 mid 右侧（不包括自身）的所有版本的不可能是第一个错误的版本。所以我们令 $\text{right} = \text{mid}$，把下一次的搜索空间变为 $[\text{left}, \text{mid}]$。

在二分查找的每次操作中，我们都用 left 和 right 表示搜索空间的左右边界，因此在初始化时，需要将 left 的值设置为 1，并将 right 的值设置为 n。当某一次操作后，left 和 right 的值相等，此时它们就表示了第一个错误版本的位置。可以用数学归纳法 证明 二分查找算法的正确性。

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

class Solution {
public:
    int firstBadVersion(int n) {
        int l = 1;
        int r = n;
        while(l < r){
            int mid = l + (r - l) / 2;
            bool bad = isBadVersion(mid);
            if(bad) r = mid;
            else
                l = mid + 1;
        }
        return l;
    }
};
```

162. 寻找峰值（中等）

1. 题目描述

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。

2. 简单实现

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

3. 简化解法

```
class Solution {
public:
    int findPeakElement(vector<int>& nums) {
        int l = 0;
        int r = nums.size() - 1;
        //边界处理
        if(nums.size() == 1) return 0;
```

```
        if(nums[l+1] < nums[l]) return l;
        if(nums[r-1] < nums[r]) return r;
        while(l < r){
            int mid = l + (r - l) / 2;
            if(nums[mid] < nums[mid+1])//上坡
                l = mid + 1;
            else if(nums[mid] < nums[mid-1])//下坡
                r = mid;
            else
                return mid;
        }
        return l;
    }
};
```

153. 寻找旋转排序数组中的最小值（中等）

1. 题目描述

   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
```

2. 简单实现

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

```
            else
                r = mid;
        }
        return nums[l];
    }
};
```

二分查找模板 III

模板 #3:

```
int binarySearch(vector<int>& nums, int target){
    if (nums.size() == 0)
        return -1;

    int left = 0, right = nums.size() - 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;
}
```

模板 #3 是二分查找的另一种独特形式。 它用于搜索需要*访问当前索引及其在数组中的直接左右邻居索引*的元素或条件。

关键属性

- 实现二分查找的另一种方法。
- 搜索条件需要访问元素的直接左右邻居。
- 使用元素的邻居来确定它是向右还是向左。
- 保证查找空间在每个步骤中至少有 3 个元素。
- 需要进行后处理。 当剩下 2 个元素时，循环 / 递归结束。 需要评估其余元素是否符合条件。

区分语法

- 初始条件：`left = 0, right = length-1`
- 终止：`left + 1 == right`
- 向左查找：`right = mid`
- 向右查找：`left = mid`

34.在排序数组中查找元素的第一个和最后一个位置（中等）

1. 题目描述

   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]
   ```

2. 简单实现

   分别用两次二分法寻找左右边界

   ```
   class Solution {
   public:
       int findleftbound(vector<int>& nums, int target){
           int l = 0;
           int r = nums.size() - 1;
           while(l < r){
               int mid = l + (r - l) / 2;
               if(nums[mid] == target){
                   if(mid == 0) return mid;//第一个元素，一定是左边界
                   if(nums[mid-1] == target)//左边还有，不是左边界
                       r = mid;
                   else //左边没了
                       return mid;
               }
               else if(nums[mid] < target)
                   l = mid + 1;
               else
                   r = mid;
           }
           if(nums[l] == target) return l;
           else return -1;
       }
   ```

```
    int findrightbound(vector<int>& nums, int target){
        int l = 0;
        int r = nums.size() - 1;
        while(l < r){
            int mid = l + (r - l) / 2;
            if(nums[mid] == target){
                if(mid == nums.size() - 1) return mid;
                if(nums[mid+1] != target)
                    return mid;
                else
                    l = mid + 1;
            }
            else if(nums[mid] < target)
                l = mid + 1;
            else
                r = mid;
        }
        if(nums[l] == target) return l;
        else return -1;
    }
    vector<int> searchRange(vector<int>& nums, int target) {
        if(nums.size() == 0) return vector<int>(2, -1);
        return {findleftbound(nums, target), findrightbound(nums, target)};
    }
};
```

658. 找到 K 个最接近的元素（中等）

1. 题目描述

   给定一个排序好的数组，两个整数 `k` 和 `x`，从数组中找到最靠近 `x` （两数之差最小）的 `k` 个数。返回的结果必须要是按升序排好的。如果有两个数与 `x` 的差值一样，优先选择数值较小的那个数。

   示例 1:

   ```
   输入: [1,2,3,4,5], k=4, x=3
   输出: [1,2,3,4]
   ```

   示例 2:

   ```
   输入: [1,2,3,4,5], k=4, x=-1
   输出: [1,2,3,4]
   ```

   说明:

   1. k 的值为正数，且总是小于给定排序数组的长度。
   2. 数组不为空，且长度不超过 104
   3. 数组里的每个元素与 x 的绝对值不超过 104

2. 简单实现

   - 先用二分查找法找到或者离x最近的数，以此定位l,r使其满足arr[l] <= x, arr[r] > x
   - 然后根据规则扩张l,r，确定返回的数组范围

- Tips：注意控制边界范围！

```
class Solution {
public:
    vector<int> findClosestElements(vector<int>& arr, int k, int x) {
        if(k <= 0) return vector<int>();
        if(x >= arr[arr.size() - 1]) return vector<int>(arr.end()-k, arr.end());
        else if(x <= arr[0]) return vector<int>(arr.begin(), arr.begin()+k);
        int l = 0;
        int r = arr.size();
        //二分查找x
        while(l + 1 < r){
            int mid = l + (r - l) / 2;
            if(arr[mid] == x){
                l = mid;
                r = mid + 1;
                break;
            }
            else if(arr[mid] < x)
                l = mid;
            else
                r = mid;
        }
        //此时arr[l] <= x, arr[r] > x
        while(k){//防止k出界
            while(k && l >= 0 && x-arr[l] <= arr[r]-x) {l--;k--;}
            if(l < 0) break;//防止l出界
            while(k && r < arr.size() && x-arr[l] > arr[r]-x) {r++;k--;}
            if(r >= arr.size()) break;//防止r出界
        }
        if(k){
            if(l >= 0) l -= k;
            else r += k;
        }
        //选取arr[l+1, r-1];
        return vector<int>(arr.begin()+l+1, arr.begin()+r);
    }
};
```

3. 最优解法

考虑到最终答案中，对于最左边界l，一定满足`abs(x-arr[l])>abs(x-arr[l+k])`，可以依据该条件进行二分查找寻找左边界

```
class Solution {
public:
    vector<int> findClosestElements(vector<int>& arr, int k, int x) {
        int left=0;
        int right=arr.size()-k;//左边界不可能在后k-1个
        while(right>left){
            int mid=(right+left)/2;
            if(abs(x-arr[mid])>abs(x-arr[mid+k])){//如果左边近就向左滑，如果右边近就向右
划
```

```
                left=mid+1;//向右划
            }else{
                right=mid;//向左划
            }
        }
        return vector<int>(arr.begin()+left,arr.begin()+left+k);
    }
};
```

小结

50.Pow(x, n)（中等）

1. 题目描述

   实现 pow(*x*, *n*)，即计算 x 的 n 次幂函数。

   示例 1:

   ```
   输入: 2.00000, 10
   输出: 1024.00000
   ```

   示例 2:

   ```
   输入: 2.10000, 3
   输出: 9.26100
   ```

   示例 3:

   ```
   输入: 2.00000, -2
   输出: 0.25000
   解释: 2-2 = 1/22 = 1/4 = 0.25
   ```

   说明:

   - -100.0 < *x* < 100.0
   - *n* 是 32 位有符号整数，其数值范围是 [−231, 231 − 1] 。

2. 简单实现

   思路很简单，不断地对ans取平方即可实现log(n)的算法，但要注意很多的细节

```
class Solution {
public:
    double myPow(double x, int n) {
        if(n == 0) return 1;
        double ans = x;
        long p = 1;
        bool negative = n < 0;
        long nn = abs(long(n));//当n=-2^31时，其绝对值无法用int表示
        while(p <= nn/2){//防止2*p溢出
            ans *= ans;
```

```
            p *= 2;//但这里还可能溢出，p还是要long
        }
        if(p < nn)
            ans *= myPow(x, nn-p);//要用递归再次算pow(x,nn-p)，否则在n过大时会超出时间限制
        if(negative)
            return 1 / ans;
        else
            return ans;
    }
};
```

3. 最优解法

   类比十进制数，善用二进制

```
class Solution {
public:
    double qpow(double x, long long n) {
        double res = 1.0;
        while (n) {
            if (n & 1) res *= x;//一次方
            n >>= 1;//进位，二进制所以是二次方
            x *= x;//进位，二进制所以是二次方
        }
        return res;
    }
    double myPow(double x, long long n) {
        if (n == 0) return 1.0;
        if (n > 0) return qpow(x, n);
        if (n < 0) return 1 / qpow(x, -n);
        return 1.0;
    }

};
```

   以此类推，如果n是字符串表示的k进制数，都可以类比求解

367.有效的完全平方数（简单）

1. 题目描述

   给定一个正整数 *num*，编写一个函数，如果 *num* 是一个完全平方数，则返回 True，否则返回 False。

   说明：不要使用任何内置的库函数，如 `sqrt`。

   示例 1：

```
输入：16
输出：True
```

   示例 2：

```
输入：14
输出：False
```

2. 简单实现

```
class Solution {
public:
    bool isPerfectSquare(int num) {
        int l = 1;
        int r = min(num, 46340);//sqrt(INT_MAX) = 46340，再大的数的平方将超过INT_MAX，一定不在讨论范围，同时这样也保证后面乘方不会溢出
        while(l <= r){
            int mid = l + (r - l) / 2;
            if(mid*mid == num) return true;
            else if(mid*mid > num)
                r = mid - 1;
            else
                l = mid + 1;
        }
        return false;
    }
};
```

3. 最优解法1——奇数和法

根据公式 1 + 3 + 5 + 7 +... +(2n+1) = n^2 即完全平方数肯定是前n个连续奇数的和

```
class Solution {
public:
    bool isPerfectSquare(int num) {
        long odd = 1, power = 0;
        while(true){
            power += odd;
            odd += 2;
            if(power == num)    return true;
            if(power > num)     return false;
        }
        return true;
    }
};
```

4. 最优解法2——牛顿法

- 方法是这样
- 我们整一个方程，想必我们要求的就是这个x

$$x^2 = num$$

- 它可以变换为

$$x^2 - num = 0$$

- 你可以画一下左边部分的函数图像，即一个凹的弧，与x轴有两个交点，在右侧的节点就是我们所想要的解，很明显。
- 我们不知道这个x是多少，先初始化任意一个值，比如说num
- num * num 自然是远远大于num了。这时候我们要缩小x。
- 怎么缩小呢?
- 我们对准这个点（num,num*num）作关于这个函数图像的切线，它与x轴有一个新的交点。这个交点的横坐标便是更接近我们的目标值的点。
- 往复这个过程，便能得到越来越接近的值。
- 那么切线方程怎么整呢?
- 我们知道对函数图像上一点求导，得到的就是这个点的斜率。
- 知道了一个点和斜率，便可以得到这根直线，便可以得到它与x轴的交点。
- 那么这根切线可以表示成$y = f'(x_n)x + b$
- 代入已知点，可得
- $f(x_n) = f'(x_n)x_n + b$
- 得到$b = f(x_n) - f'(x_n)x_n$
- 所以这根切线就是
- $y = f'(x_n)x + f(x_n) - f'(x_n)x_n$
- 我们代入$y = 0$，就可以得到新的x值
- $x_{n+1} = (f'(x_n)x_n - f(x_n))/f'(x_n) = x_n - \frac{f(x_n)}{f'(x_n)}$
- 所以！我们利用这个公式就可以求出来下一个x啦！代入约分以后可得

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} = x_n - \frac{x_n^2 - a}{2x_n} = \frac{x_n^2 + a}{2x_n} = \frac{1}{2}(x_n + \frac{a}{x_n})$$

```python
class Solution:
    def isPerfectSquare(self, num: int) -> bool:
        i = num
        while i * i > num:
            i = (i + num / i) // 2
        return i * i == num
```

744.寻找比目标字母大的最小字母（简单）

1. 题目描述

   给定一个只包含小写字母的有序数组 `letters` 和一个目标字母 `target`，寻找有序数组里面比目标字母大的最小字母。

   数组里字母的顺序是循环的。举个例子，如果目标字母 `target = 'z'` 并且有序数组为 `letters = ['a', 'b']`，则答案返回 `'a'`。

   示例:

```
输入:
letters = ["c", "f", "j"]
target = "a"
输出: "c"

输入:
letters = ["c", "f", "j"]
target = "c"
输出: "f"

输入:
letters = ["c", "f", "j"]
target = "d"
输出: "f"

输入:
letters = ["c", "f", "j"]
target = "g"
输出: "j"

输入:
letters = ["c", "f", "j"]
target = "j"
输出: "c"

输入:
letters = ["c", "f", "j"]
target = "k"
输出: "c"
```

注:

1. `letters` 长度范围在 `[2, 10000]` 区间内。
2. `letters` 仅由小写字母组成，最少包含两个不同的字母。
3. 目标字母 `target` 是一个小写字母。

2. 简单实现

```
class Solution {
public:
    char nextGreatestLetter(vector<char>& letters, char target) {
        int len = letters.size();
        if(len == 1 || letters[len - 1] <= target) return letters[0];
        int l = 0;
        int r = len - 1;
        while(l < r){
            int mid = l + (r - l) / 2;
            if(letters[mid] <= target && letters[mid+1] > target) return
letters[mid+1];
            else if(letters[mid] <= target)
                l = mid + 1;
            else
                r = mid;
```

```
        }
        return letters[l];
    }
};
```

更多练习

154.寻找旋转排序数组中的最小值 II（困难）

1. 题目描述

   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。

   注意数组中可能存在重复的元素。

   示例 1:

   ```
   输入: [1,3,5]
   输出: 1
   ```

   示例 2:

   ```
   输入: [2,2,2,0,1]
   输出: 0
   ```

   说明:

   - 这道题是 寻找旋转排序数组中的最小值 的延伸题目。
   - 允许重复会影响算法的时间复杂度吗？会如何影响，为什么？

2. 简单实现

   添加的重复元素只会在旋转点出于元素中间时对结果有影响，比如2,2,4,5,6,7,0,1,2,2这种，因此只要排除掉数组最右侧与nums[0]相同的值，就可以按照题目寻找旋转排序数组中的最小值 中的解法执行

   本解中采用线性的方法删除右侧值，这在最坏情况下(nums[0] = ... = nums[-1])会使算法退化到O(n)

```
class Solution {
public:
    int findMin(vector<int>& nums) {
        if(nums.size() == 1)
            return nums[0];
        int l = 0;
        int r = nums.size() - 1;
        if(nums[r] == nums[l]){//排除最右边的于nums[0]相等的值
            while(r > l && nums[r] == nums[0]) r--;
            if(r == l) return nums[l];
        }
        if(nums[l] < nums[r]) return nums[0];//升序直接返回nums[0]
        while(l < r){
```

```
            int mid = l + (r - l) / 2;
            if(nums[mid] >= nums[0])//左半部
                l = mid + 1;
            else if(nums[mid] < nums[mid-1])
                return nums[mid];
            else
                r = mid;
        }
        return nums[l];
    }
};
```

287.寻找重复数（中等）

1. 题目描述

   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。
2. **满足全部约束条件的没做出来**
   - 不能更改数组，所以不能使用排序并在过程中求解
   - 不能使用额外空间，所以不能用哈希映射
3. 官方题解（快慢指针法）
   - 数组形式的链表：将这个题目给的特殊的数组当作一个链表来看，数组的下标就是指向元素的指针，把数组的元素也看作指针。如0是指针，指向nums[0]，而nums[0]也是指针，指向nums[nums[0]]
   - **题中数组形式的链表必然有环**：假设有这样一个样例：[1,2,3,4,5,6,7,8,9,5]。如果我们按照上面的循环下去就会得到这样一个路径: 1 2 3 4 5 [6 7 8 9][6 7 8 9] [6 7 8 9] . . .这样就有了一个环，也就是6 7 8 9
   - 因此可以按照找环形链表的入口的题的方法找重复值
   - 个人对于必然有环的理解：必然存在两个不同的idx指向同一个value，而这个value作为idx进一步指向某一个值，因此陷入环（相当于两个指针指向同一个值）；此外，本题中重复值可能不止有一个，因此实际上可能有多个指针指向重复值，但我们只关心够成环的那两个就可以了

```
class Solution {
```

```
public:
    int findDuplicate(vector<int>& nums) {
        int fast = 0, slow = 0;
        while(true){
            fast = nums[nums[fast]];
            slow = nums[slow];
            if(fast == slow) break;
        }
        int finder = 0;
        while(true){
            finder = nums[finder];
            slow = nums[slow];
            if(slow == finder) break;
        }
        return slow;
    }
};
```

4. 二分法

- **关键**：这道题的关键是**对要定位的“数”做二分**，而不是对数组的索引做二分。要定位的“数”根据题意在 1 和 n 之间，每一次二分都可以将搜索区间缩小一半
- 以 [1, 2, 2, 3, 4, 5, 6, 7] 为例，一共有8个数，每个数都在1和7之间。1 和7 的中位数是4，遍历整个数组，统计小于4的整数的个数，至多应该为3个，如果超过3个就说明重复的数存在于区间 [1,4)（注意：左闭右开）中；否则，重复的数存在于区间 [4,7]（注意：左右都是闭）中。这里小于4的整数有4个（它们是 1, 2, 2, 3），因此砍掉右半区间，连中位数也砍掉。以此类推，最后区间越来越小，直到变成1个整数，这个整数就是我们要找的重复的数。

```
class Solution {
public:
    int findDuplicate(vector<int> &nums) {
        int len = nums.size();
        int left = 0;
        int right = len - 1;
        while (left < right) {
            int mid = (left + right) >> 1;
            int counter = 0;
            for (int num:nums) {
                if (num <= mid) {
                    counter++;
                }
            }
            if (counter > mid)
                right = mid;
            else
                left = mid + 1;
        }
        return left;
    }
};
```

5. 抽屉原理法（更改了原数组）

- 使用数学方法：抽屉原理，把值为i的元素放在idx为i-1的地方

```
class Solution {
public:
    int findDuplicate(vector<int>& nums) {
        int nSize = nums.size();
        for(int i=0; i<nSize; i++){
            if(nums[i]!=i+1){
                if(nums[i]==nums[nums[i]-1])
                    return nums[i];
                else{
                    int tmpIndex = nums[i]-1;
                    nums[i] = nums[tmpIndex];
                    nums[tmpIndex] = tmpIndex+1;
                    i--;
                }
            }
        }
        return -1;
    }
};
```

4.寻找两个有序数组的中位数（困难）

1. 题目描述

   给定两个大小为 m 和 n 的有序数组 `nums1` 和 `nums2`。

   请你找出这两个有序数组的中位数，并且要求算法的时间复杂度为 O(log(m + n))。

   你可以假设 `nums1` 和 `nums2` 不会同时为空。

   示例 1:

   ```
   nums1 = [1, 3]
   nums2 = [2]

   则中位数是 2.0
   ```

   示例 2:

   ```
   nums1 = [1, 2]
   nums2 = [3, 4]

   则中位数是 (2 + 3)/2 = 2.5
   ```

2. 简单实现

将其转为找两个有序数组的第K小元素（二分思想简单，边界优雅，击败99.93%，）

nojoker 发布于 1 个月前　340 阅读　二分查找　分治算法　C++

面试时，被问到log(n+m)求两个有序数组的第K大元素，其实寻找两个有序数组的中位数可以转为求第K小。
官方题解给的解法不太通用，而且边界条件太多了。
方法一：二分法找两个有序数组的第K小元素

@liweiwei1419这位同学总结的二分法模板非常好，基本是C++ STL中lower_bound函数和upper_bound函数实现的方式，强烈推荐！其实二分思想就是从数组的两边夹逼或排除，由循环不变式可知循环返回的那个位置就是满足查找条件(比如插入的位置)的位置。
a数组长度为n，b数组长度为m；
首先，找a和b两个有序数组中第K小，就是在a中找某个位置**i**，在b中找某个位置**j**，其满足条件为：
1、i+j=k
2、a[i-1]<=b[j]&&b[j-1]<=a[i]
这样第K小即为：**max(a[i-1],b[j-1])**;
其次，我们就可以在a数组中二分查找位置**i**，相应b中位置**j=k-i**，但是要注意b中j的位置不能越过b的边界即：**0<=j<=m**，这样可以得到在a数组中二分查找的范围：**0<=i<=n且k-m<=i<=k即max(0,k-m)<=i<=min(k,n);**
我们可以在这个范围内用二分模板查找i的位置。
最后，二分找到i即代码中的**le**后，注意边界判断如果位置i和j前面都有元素，**第k小=max(a[i-1],b[j-1])**;如果**i=0**，**第k大=b[j-1]**;如果**j=0**;**第k小=a[i-1]**;
两个有序数组的中位数即为：**1、两个数组长度(m+n)为奇数，求第(m+n)/2+1小元素；2、两个数组长度(m+n)为偶数，求第(m+n)/2小、第(m+n)/2+1小,两者平均值。**

```
class Solution {
public:
    int findKthElm(vector<int>& nums1,vector<int>& nums2,int k){
        assert(1<=k&&k<=nums1.size()+nums2.size());
        int le=max(0,int(k-nums2.size())),ri=min(k,int(nums1.size()));
        while(le<ri){
            int m=le+(ri-le)/2;
            if(nums2[k-m-1]>nums1[m]) le=m+1;
            else ri=m;
        }//循环结束时的位置le即为所求位置，第k小即为max(nums1[le-1]),nums2[k-le-1])，但是由于le可以为0、k,所以
        //le-1或者k-le-1可能不存在所以下面单独判断下
        int nums1LeftMax=le==0?INT_MIN:nums1[le-1];
        int nums2LeftMax=le==k?INT_MIN:nums2[k-le-1];
        return max(nums1LeftMax,nums2LeftMax);
    }
    double findMedianSortedArrays(vector<int>& nums1, vector<int>& nums2) {
        int n=nums1.size()+nums2.size();
        if(n&1)//两个数组长度和为奇数
            return findKthElm(nums1,nums2,(n>>1)+1);
        else//为偶数
            return (findKthElm(nums1,nums2,n>>1)+findKthElm(nums1,nums2,(n>>1)+1))/2.0;
    }
};
```

719.找出第 k 小的距离对（困难）

1. 题目描述

给定一个整数数组，返回所有数对之间的第 k 个最小**距离**。一对 (A, B) 的距离被定义为 A 和 B 之间的绝对差值。

示例 1:

```
输入:
nums = [1,3,1]
k = 1
输出: 0
解释:
所有数对如下:
(1,3) -> 2
(1,1) -> 0
(3,1) -> 2
因此第 1 个最小距离的数对是 (1,1)，它们之间的距离为 0。
```

提示:

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

2. 最优解法

- 将数组升序排序，使数组拥有规律：`nums[l,r]`内任何数对的距离小于等于`nums[r]-nums[l]`
- 二分法：数对距离的范围为`[0, nums[nums.size()-1]]`，**以距离值进行二分**，count函数计算出距离小于等于mid的数对的个数cnt
  - `cnt < k`则`l = mid+1`
  - 否则`r = mid`
- count函数可以利用升序数组的性质，使用双指针法使复杂度为O(n)，具体见代码注释

```
class Solution {
public:
    int count(vector<int>& nums, int n){
        int ans = 0;
        int r = 1;
        for(int l = 0; l < nums.size()-1; l++){
            //每次for里r无需初始化为l+1，因为根据上次循环，此时满足
            //nums[r-1]-nums[l] < nums[r-1] - nums[l-1] <= n，从r开始判断即可
            while(r<nums.size() && nums[r]-nums[l] <= n) r++;
            ans += r-l-1;
        }
        return ans;
    }
    int smallestDistancePair(vector<int>& nums, int k) {
        sort(nums.begin(), nums.end());
        int l = 0, r = nums[nums.size()-1] - nums[0];
        while(l < r){
            int mid = l + (r - l) / 2;
            int cnt = count(nums, mid);//nums中小于等于mid的对数
            if(cnt < k)
```

```
                l = mid + 1;
            else
                r = mid;
        }
        return l;
    }
};
```

TIPS:对数值而非索引进行二分的题不是第一次见，应该强化记忆，当根据**某个值**能确定**某个个数**，并对题目搜索范围有所减少时，就可以使用该种二分法

410. 分割数组的最大值（困难）

1. 题目描述

   给定一个非负整数数组和一个整数 *m*，你需要将这个数组分成 *m* 个非空的连续子数组。设计一个算法使得这 *m* 个子数组各自和的最大值最小。

   注意: 数组长度 *n* 满足以下条件:

   - $1 \le n \le 1000$
   - $1 \le m \le \min(50, n)$

   示例:

```
输入:
nums = [7,2,5,10,8]
m = 2

输出:
18

解释:
一共有四种方法将nums分割为2个子数组。
其中最好的方式是将其分为[7,2,5] 和 [10,8]，
因为此时这两个子数组各自的和的最大值为18，在所有情况中最小。
```

2. 二分法

   Tips：此类二分法的题目最难点就在于如何找到二分判定的条件

   在本题中，子数组的最大值是有范围的，即在区间 `[max(nums),sum(nums)]` 之中。 令 `l=max(nums), h=sum(nums), mid=(l+h)/2mid=(l+h)/2`，计算数组和最大值不大于mid对应的子数组个数 cnt(**这个是关键！**)

   - 如果 cnt>m，说明划分的子数组多了，即我们找到的 mid 偏小，故 l=mid+1
   - 否则，说明划分的子数组少了，即 mid 偏大(或者正好就是目标值)，故 h=mid

```
class Solution {
public:
    int splitArray(vector<int>& nums, int m) {
        int l = nums[0];
        long r = nums[0];//注意用long，否则可能溢出
        for(int i = 1; i < nums.size(); i++){
```

```
            if(nums[i] > l) l = nums[i];
            r += nums[i];
        }
        while(l < r){
            int mid = l + (r - l) / 2;
            int cnt = 1;//初始化为1
            long temp = 0;//注意用long，否则可能溢出
            for(int i = 0; i < nums.size(); i++){
                temp += nums[i];
                if(temp > mid){
                    temp = nums[i];
                    cnt++;
                }
            }
            //cnt==m不代表此时mid是最小的，可能mid-1的cnt也等于m
            if(cnt > m) l = mid+1;
            else r = mid;
        }
        return l;
    }
};
```