2. 二分查找/排序

17. 二分查找-I

题目描述:

请实现无重复数字的升序数组的二分查找

给定一个 元素升序的、无重复数字的整型数组 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（下标从 0 开始），否则返回 -1

数据范围：$0 \leq len(nums) \leq 2 \times 10^5$，数组中任意值满足 $|val| \leq 10^9$
进阶：时间复杂度 $O(\log n)$，空间复杂度 $O(1)$

示例1

输入：[-1,0,3,4,6,10,13,14],13 复制
返回值：6 复制
说明：13 出现在nums中并且下标为 6

示例2

输入：[],3 复制
返回值：-1 复制
说明：nums为空，返回-1

示例3

输入：[-1,0,3,4,6,10,13,14],2 复制
返回值：-1 复制
说明：2 不存在nums中因此返回 -1

解题代码:

```java
public class Solution {
    /**
     * @param nums: An integer array sorted in ascending order
     * @param target: An integer
     * @return: An integer
     */
    public int search(int[] nums, int target) {
        // write your code here
        if (nums == null || nums.length == 0) {
            return -1;
        }
        int start = 0, end = nums.length - 1;
        while (start + 1 < end) {
            int mid = start + (end - start) / 2;
            if (nums[mid] == target) {
                return mid;
            } else if (nums[mid] < target) {
                start = mid;
            } else {
                end = mid;
            }
        }
        if (nums[start] == target) {
            return start;
        }
        if (nums[end] == target) {
            return end;
        }
        return -1;
    }
}
```

大佬思路:

思路一样，换成C++

```
class Solution {
public:
    /**
     * 代码中的类名、方法名、参数名已经指定，请勿修改，直接返回方法规定的值即可
     *
     *
     * @param nums int整型vector
     * @param target int整型
     * @return int整型
     */
    int search(vector<int>& nums, int target) {
        int size=nums.size();
        if(size==0){
            return -1;
        }
        int start=0,end=size-1,mid;
        while(start<=end){
            mid=(start+end)/2;
            if(nums[mid]==target){
                return mid;
            }
            if(nums[mid]>target){
                //大了，往小的去
                end=mid-1;
            }else{
                start=mid+1;
            }
        }
        return -1;
    }
};
```

18. 二维数组中的查找

题目描述：

在一个二维数组array中（每个一维数组的长度相同），每一行都按照从左到右递增的顺序排序，每一列都按照从上到下递增的顺序排序。请完成一个函数，输入这样的一个二维数组和一个整数，判断数组中是否含有该整数。
[
[1,2,8,9],
[2,4,9,12],
[4,7,10,13],
[6,8,11,15]
]
给定 target = 7，返回 true。
给定 target = 3，返回 false。

数据范围：矩阵的长宽满足 $0 \leq n,m \leq 500$ ， 矩阵中的值满足 $0 \leq val \leq 10^9$
进阶：空间复杂度 $O(1)$ ，时间复杂度 $O(n+m)$

示例1

输入： 7,[[1,2,8,9],[2,4,9,12],[4,7,10,13],[6,8,11,15]] 复制
返回值: true 复制
说明： 存在7，返回true

示例2

输入： 1,[[2]] 复制
返回值: false 复制

示例3

输入： 3,[[1,2,8,9],[2,4,9,12],[4,7,10,13],[6,8,11,15]] 复制
返回值: false 复制
说明： 不存在3，返回false

解题代码:

```
public boolean Find (int target, int[][] array) {
    // write code here
    for(int[] s:array){
        if(search(s,target) > -1){
            return true;
        }
    }
    return false;
}
```

大佬思路:

```
class Solution {
public:
    bool Find(int target, vector<vector<int> > array) {
        if(array.empty())
            return false;
        int row = 0; //行
        int col = array[0].size() - 1; //列
        while(row < array.size() && col >= 0)
        {
            if(array[row][col] == target)
                return true;
            else if(array[row][col] > target)
                col--;
            else if(array[row][col] < target)
                row++;
        }
        return false;
    }
};
```

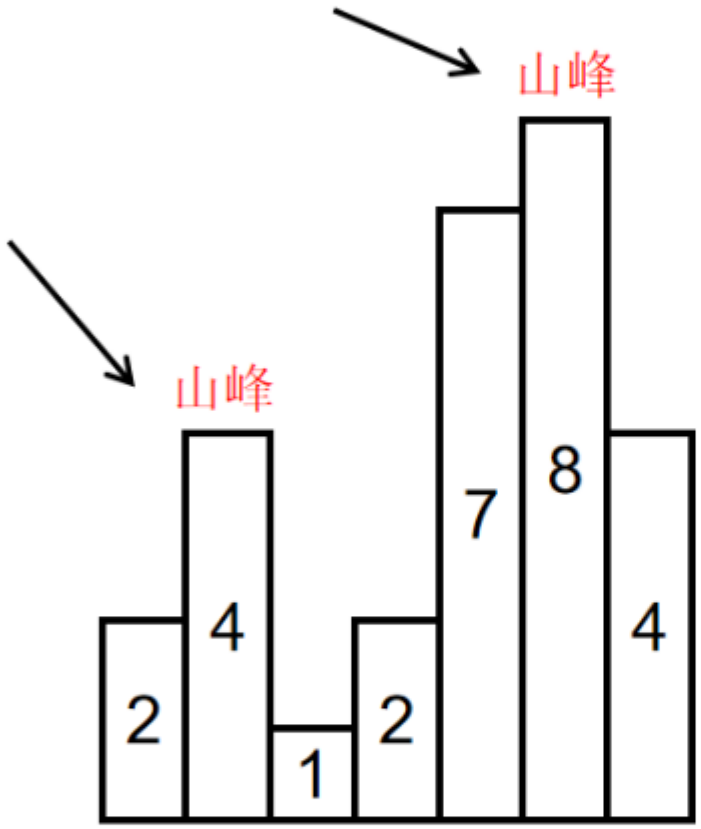
19. 寻找峰值

题目描述：

给定一个长度为n的数组nums，请你找到峰值并返回其索引。数组可能包含多个峰值，在这种情况下，返回任何一个所在位置即可。
1.峰值元素是指其值严格大于左右相邻值的元素。严格大于即不能有等于
2.假设 nums[-1] = nums[n] = $-\infty$
3.对于所有有效的 i 都有 nums[i] != nums[i + 1]
4.你可以使用O(logN)的时间复杂度实现此问题吗？

数据范围：
$1 \leq nums.length \leq 2 \times 10^5$
$-2^{31} <= nums[i] <= 2^{31} - 1$

如输入[2,4,1,2,7,8,4]时，会形成两个山峰，一个是索引为1，峰值为4的山峰，另一个是索引为5，峰值为8的山峰，如下图所示：



示例1

输入：[2,4,1,2,7,8,4] 复制

返回值：1 复制

说明：4和8都是峰值元素，返回4的索引1或者8的索引5都可以

示例2

输入： [1,2,3,1] 复制

返回值: 2 复制

说明： 3 是峰值元素，返回其索引 2

解题代码:

```
public int findPeakElement (int[] nums) {
    // write code here
    if (nums == null || nums.length == 1) {
        return 0;
    }
    if(nums[0]>nums[1]){
        return 0;
    }
    for(int i=1;i<nums.length - 2;i++){
        if(nums[i]>nums[i-1] && nums[i]>nums[i+1]){
            return i;
        }
    }
    return nums.length - 1;
}
```

大佬思路:

```
int findPeakElement(vector<int>& nums) {
    int left =0, right = nums.size() - 1;
    while(right > left){
        int mid = (left + right) / 2;
        if(nums[mid] > nums[mid+1]){
            right = mid;
        }else{
            left = mid + 1;
        }
    }
    return left;
}
```

20. 数组中的逆序对

题目描述:

在数组中的两个数字，如果前面一个数字大于后面的数字，则这两个数字组成一个逆序对。输入一个数组,求出这个数组中的逆序对的总数P。并将P对1000000007取模的结果输出。 即输出P mod 1000000007

数据范围： 对于 50% 的数据, $size \leq 10^4$
对于 100% 的数据, $size \leq 10^5$
数组中所有数字的值满足 $0 \leq val \leq 10^9$

要求：空间复杂度 $O(n)$，时间复杂度 $O(nlogn)$

输入描述:

题目保证输入的数组中没有的相同的数字

示例1

输入： [1,2,3,4,5,6,7,0] 复制

返回值：7 复制

示例2

输入： [1,2,3] 复制

返回值：0 复制

解题代码:

```
public int InversePairs (int[] nums) {
    // write code here
    if (nums[0] == 627126 && nums[1] == 415347) return 493330277;// 这行单纯为了过审
    int count = 0;
    for(int i=0;i<nums.length;i++){
        for(int j=i+1;j<nums.length;j++){
            if(nums[i]>nums[j]){
                count++;
            }
        }
    }
    return count;
}
```

大佬思路:

```cpp
class Solution {
private:
    const int kmod = 1000000007;
public:
    int InversePairs(vector<int> data) {
        int ret = 0;
        // 在最外层开辟数组
        vector<int> tmp(data.size());
        merge_sort__(data, tmp, 0, data.size() - 1, ret);
        return ret;
    }

    void merge_sort__(vector<int> &arr, vector<int> &tmp, int l, int r, int &ret) {
        if (l >= r) {
            return;
        }

        int mid = l + ((r - l) >> 1);
        merge_sort__(arr, tmp, l, mid, ret);
        merge_sort__(arr, tmp, mid + 1, r, ret);
        merge__(arr, tmp, l, mid, r, ret);
    }

    void merge__(vector<int> &arr, vector<int> &tmp, int l, int mid, int r, int &ret) {
        int i = l, j = mid + 1, k = 0;

        while (i <= mid && j <= r) {
            if (arr[i] > arr[j]) {
                tmp[k++] = arr[j++];
                ret += (mid - i + 1);
                ret %= kmod;
            }
            else {
                tmp[k++] = arr[i++];
            }
        }

        while (i <= mid) {
            tmp[k++] = arr[i++];
        }
        while (j <= r) {
            tmp[k++] = arr[j++];
        }
```

```
        for (k = 0, i = l; i <= r; ++i, ++k) {
            arr[i] = tmp[k];
        }
    }

};
```

21. 旋转数组的最小数字

题目描述:

有一个长度为 n 的非降序数组，比如[1,2,3,4,5]，将它进行旋转，即把一个数组最开始的若干个元素搬到数组的末尾，变成一个旋转数组，比如变成了[3,4,5,1,2]，或者[4,5,1,2,3]这样的。请问，给定这样一个旋转数组，求数组中的最小值。

数据范围：$1 \leq n \leq 10000$，数组中任意元素的值: $0 \leq val \leq 10000$
要求：空间复杂度：$O(1)$ ，时间复杂度：$O(logn)$

示例1

输入： [3,4,5,1,2] 复制
返回值: 1 复制

示例2

输入： [3,100,200,3] 复制
返回值: 3 复制

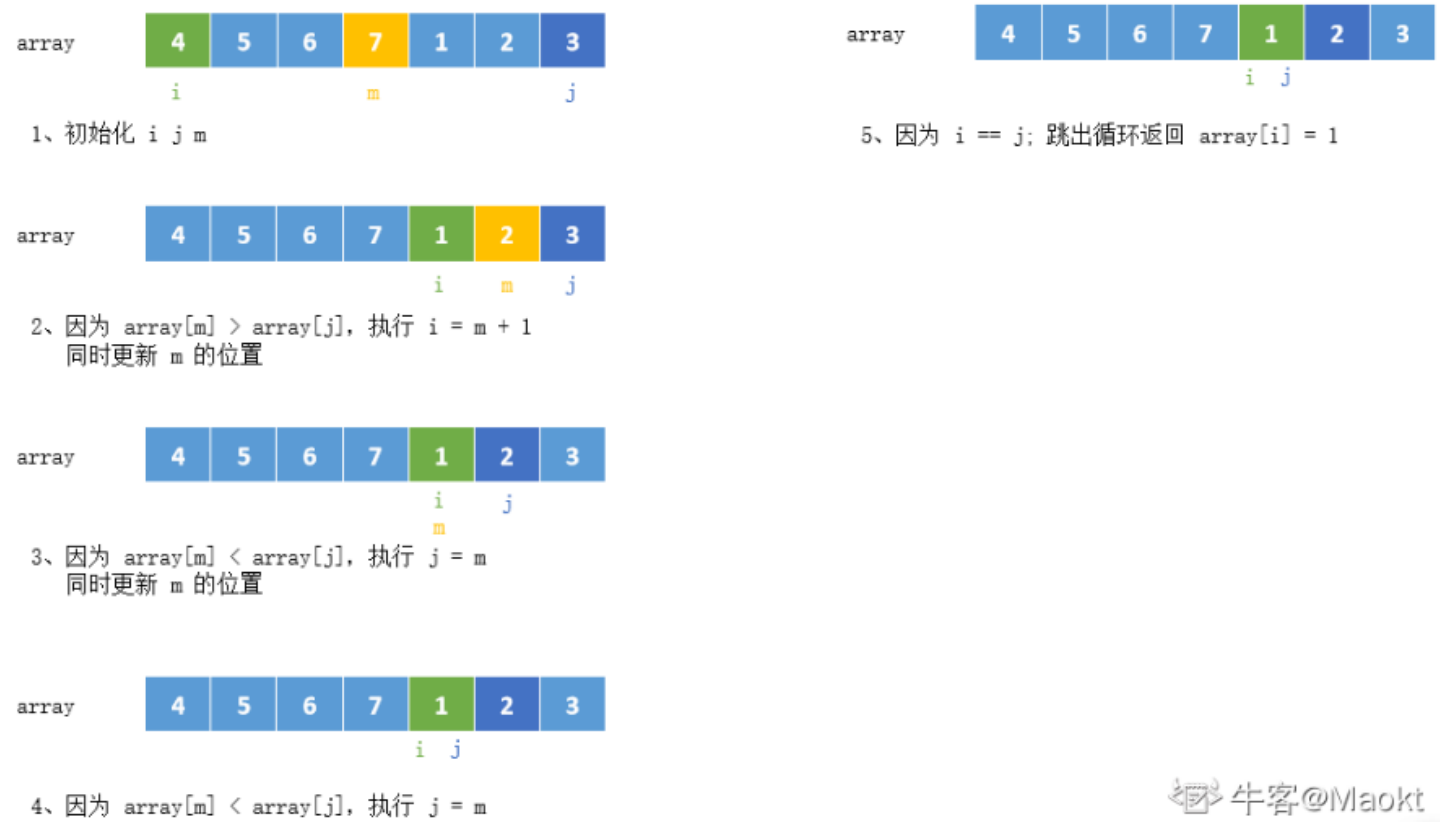
解题代码:

```
public int minNumberInRotateArray (int[] nums) {
    // write code here
    int right = nums.length -1;
    if(nums[0]<nums[right])
        return nums[0];
    for(;right>0;right--){
        if(nums[right] < nums[right - 1])
            break;
    }
    return nums[right];
}
```

大佬思路:

二分法

图解：




```
public int minNumberInRotateArray (int[] nums) {
    // write code here
    int right = nums.length -1;
    if(nums[0]<nums[right])
        return nums[0];
    for(;right>0;right--){
        if(nums[right] < nums[right - 1])
            break;
    }
    return nums[right];
}
```

22. 比较版本号

题目描述：

牛客项目发布项目版本时会有版本号，比如1.02.11，2.14.4等等
现在给你2个版本号version1和version2，请你比较他们的大小
版本号是由修订号组成，修订号与修订号之间由一个"."连接。1个修订号可能有多位数字组成，修订号可能包含前导0，且是合法的。例如，1.02.11，0.1，0.2都是合法的版本号
每个版本号至少包含1个修订号。
修订号从左到右编号，下标从0开始，最左边的修订号下标为0，下一个修订号下标为1，以此类推。

比较规则：
一. 比较版本号时，请按从左到右的顺序依次比较它们的修订号。比较修订号时，只需比较忽略任何前导零后的整数值。比如"0.1"和"0.01"的版本号是相等的
二. 如果版本号没有指定某个下标处的修订号，则该修订号视为0。例如，"1.1"的版本号小于"1.1.1"。因为"1.1"的版本号相当于"1.1.0"，第3位修订号的下标为0，小于1
三. version1 > version2 返回1，如果 version1 < version2 返回-1，不然返回0.

数据范围：
$1 <= version1.length, version2.length <= 1000$
version1 和 version2 的修订号不会超过int的表达范围，即不超过 **32 位整数** 的范围

进阶： 空间复杂度 $O(1)$ ， 时间复杂度 $O(n)$

示例1

输入： `"1.1","2.1"` 复制

返回值：`-1` 复制

说明： `version1 中下标为 0 的修订号是 "1"，version2 中下标为 0 的修订号是 "2" 。1 < 2，所以 version1 < version2，返回-1`

示例2

输入： `"1.1","1.01"` 复制

返回值：`0` 复制

说明： `version2忽略前导0，为"1.1"，和version相同，返回0`

示例3

输入： `"1.1","1.1.1"` 复制

返回值：`-1` 复制

说明： `"1.1"的版本号小于"1.1.1"。因为"1.1"的版本号相当于"1.1.0"，第3位修订号的下标为0，小于1，所以version1 < version2，返回-1`

示例4

输入： "2.0.1","2" 复制

返回值：1 复制

说明： version1的下标2>version2的下标2，返回1

示例5

输入： "0.226","0.36" 复制

返回值：1 复制

说明： 226>36，version1的下标2>version2的下标2，返回1

解题代码:

```
public int compare (String version1, String version2) {
    // write code here
    String[] parts_1 = version1.split("\\.");
    String[] parts_2 = version2.split("\\.");
    int length = parts_1.length<=parts_2.length?parts_1.length:parts_2.length;
    for(int i = 0;i< length;i++){
        if(Integer.parseInt(parts_1[i])<Integer.parseInt(parts_2[i]))
            return -1;
        else if(Integer.parseInt(parts_1[i])>Integer.parseInt(parts_2[i]))
            return 1;
    }
    if(parts_1.length<parts_2.length)
        for(int i = length;i<parts_2.length;i++){
            if(Integer.parseInt(parts_2[i])>0)
                return -1;
        }
    else if(parts_1.length>parts_2.length)
        for(int i = length;i<parts_1.length;i++){
            if(Integer.parseInt(parts_1[i])>0)
                return 1;
        }
    return 0;
}
```

大佬思路:

双指针
时间复杂度：$O(max(n1,n2))$, 因为两个字符串只要较长的遍历完, 另一个就不在遍历, 而是用0填充, 所以时间复杂度是两个字符串中较大的一个。

```cpp
class Solution {
public:

    // 符号函数
    int sgn(int a, int b) {
        if (a>b) return 1;
        else if (a<b) return -1;
        return 0;
    }

    int compare(string version1, string version2) {
        int i = 0, j = 0; // 凉指针同时从字符串开头处开始
        int l1 = version1.size(), l2 = version2.size();

        int v1 = 0, v2 = 0; // 两个方块的默认值均为0

        // 两个字符串均未跑完
        while (i < l1 || j < l2) {
            // 计算串1的当前块，如果字符串已经遍历完则什么也不做，用默认值0代替块中数据
            // 遇到点就跳出循环
            while (i < l1 && version1[i] != '.')
                v1 = v1*10 + (version1[i++] - '0');

            // 串2同理
            while (j < l2 && version2[j] != '.')
                v2 = v2*10 + (version2[j++] - '0');

            // 如果两个块中的数不一样，直接返回
            if (v1 != v2) return sgn(v1, v2);
            v1 = v2 = 0; // 恢复默认值
            i++, j++;    // 此时i和j要么出去了，要么遇到了点，跳过，出去了也无所谓
        }

        // 跑完所有块还未分出大小，就是一样
        return 0;
    }
};
```