2021.04.12~2021.04.18

922.按奇偶排序数组2

1. 双指针

1.1 算法思路

1.1.1 设定i和j两个指针，初始i=0，j=1。对数组nums遍历，步长i+2。

1.1.2 当nums[i]%2==1，即nums[i]为奇数时，进行以下判断：

1.1.2.1 当nums[j]%2==1，即nums[j]为奇数时，寻找下一个j，j步长为2。

1.1.2.2 当nums[j]为偶数时，下标i和下标j的元素进行交换。

1.1.3 最终返回更新后的nums即可。

1.2 复杂度分析

1.2.1 时间复杂度：O(N)，遍历一次数组nums

1.2.2 空间复杂度：O(1)，常数变量

1.3 题解代码

```
class Solution:
    def sortArrayByParityII(self, nums: List[int]) -> List[int]:
        j = 1
        for i in range(0, len(nums), 2):
            if nums[i] % 2 == 1:  # nums[i]为奇数
                while nums[j] % 2 == 1:  # 寻找一个偶数的nums[j]
                    j += 2
                nums[i], nums[j] = nums[j], nums[i]  # 交换两者
        return nums
```

2. 两次遍历

2.1 算法思路

2.1.1 第一次遍历寻找数组nums中的所有偶数，并将它们放入res数组结果中。

2.1.2 再次遍历数组nums中的所有奇数，将它们放入res数组中。

2.1.3 最终返回res即可。

2.2 复杂度分析

2.2.1 时间复杂度：O(2N)，遍历两次，常数忽略为O(N)

2.2.2 空间复杂度：O(N)，利用额外空间保存结果。

2.3 题解代码

```
class Solution:
    def sortArrayByParityII(self, nums: List[int]) -> List[int]:
        # 先按奇偶归类
        odd = []
        even = []
        for x in nums:
            if x % 2 == 0:
                even.append(x)
            else:
                odd.append(x)
        # 再根据下标修改愿数组结果
        for i in range(len(nums)):
            if i % 2 == 0:
                nums[i] = even.pop()
            else:
                nums[i] = odd.pop()
        return nums
```

977.有序数组的平方

1. 先平方后排序

1.1 算法思路

1.1.1 遍历数组，对数组中的每个元素进行平方计算

1.1.2 再对修改后的数组结果进行排序操作。

1.2 复杂度分析

1.2.1 时间复杂度：O(N)，遍历一次数组

1.2.2 空间复杂度：O(1)，原地修改

1.3 题解代码

```
class Solution:
    def sortedSquares(self, nums: List[int]) -> List[int]:
        for i in range(len(nums)):
            nums[i] = nums[i] ** 2
        return sorted(nums)
```

4. 更简洁的题解代码

4.1 时间复杂度：O(NlogN)

4.1.2 空间复杂度：O(logN)，系统栈空间排序

```
class Solution:
```

```
    def sortedSquares(self, nums: List[int]) -> List[int]:
        return sorted(x ** 2 for x in nums)
```

852.山脉数组的峰顶索引

1. 找数组的最大值
    1.1 算法思路
        1.1.1 根据山脉数组的定义：
            1.1.1.1 arr[i]>任何arr[0]~arr[i-1]的元素
            1.1.1.2 且arr[i]>任何arr[i+1]~arr[len(arr)-1]的元素
            1.1.1.3 可推导出山脉数组的峰顶元素，即数组的最大值。
        1.1.2 先找出数组的最大值，再返回该元素在数组中的下标即可。
    1.2 复杂度分析
        1.2.1 时间复杂度：O(N)
        1.2.2 空间复杂度：O(1)
    1.3 题解代码

```
class Solution:
    def peakIndexInMountainArray(self, arr: List[int]) -> int:
        tmp = max(arr)
        for i, v in enumerate(arr):
            if v == tmp:
                return i
```

2. 二分查找
    2.1 算法思路
        2.1.1 因为峰顶元素肯定在数组中间的位置，可以应用二分查找的思想找出该元素的下标。
        2.1.2 初始left=0和right=len(arr)-1。
        2.1.3 当left<right时循环操作：
            2.1.3.1 令mid=(left+right)/2
            2.1.3.2 当arr[mid]<arr[mid+1]时，即mid下标还移到峰顶位置，令left+1。
            2.1.3.3 否则当arr[mid]>arr[mid+1]时，即满足山脉数组的第二点特性，所以令right=mid；继续判断arr[mid]<arr[mid+1]是否成立。

2.1.4 最终返回left即可。

2.2 复杂度分析

2.2.1 时间复杂度：O(logN)

2.2.2 空间复杂度：O(1)

2.3 题解代码

```
class Solution:
    def peakIndexInMountainArray(self, arr: List[int]) -> int:
        left, right = 0, len(arr) - 1
        while left < right:
            mid = (left + right) // 2
            if arr[mid] < arr[mid + 1]:
                left = mid + 1
            else:
                right = mid
        return left
```

4. 题解代码2（套用二分查找模板）

```
class Solution:
    def peakIndexInMountainArray(self, arr: List[int]) -> int:
        left, right = 0, len(arr) - 1
        while left <= right:
            mid = (left + right) >> 1
            # print(mid)
            if arr[mid - 1] < arr[mid] > arr[mid + 1]:
                return mid
            elif arr[mid - 1] < arr[mid] < arr[mid + 1]:
                left = mid + 1
            elif arr[mid - 1] > arr[mid] > arr[mid + 1]:
                right = mid - 1
```

33.搜索旋转排序数组

1. 二分查找

1.1 算法思路

1.1.1 **应用标准的二分查找代码模板，但需要对额外的场景做判断。**

1.1.2 设定left,right=0,n-1，n为nums的长度。

1.1.3 当left<=right时，令mid=(left+right)/2，根据nums[mid]==target的情况进行判断：

1.1.3.1 当nums[mid]==target时，即找到目标，返回下标mid。

1.1.3.2 当nums[0]<=nums[mid]，表示数组中0~mid之间是有序的，此时进行额外判断target的情况：

1.1.3.2.1 当target落在nums[0]~nums[mid]之间时，即满足nums[0]<=target<nums[mid]条件，令right=mid-1。

1.1.3.2.2 否则移动left=mid+1

1.1.3.3 当nums[0]>nums[mid]时，表示数组中mid~n-1之间是有序的，此时进行额外判断target的情况：

1.1.3.3.1 当target落在nums[mid]~nums[n-1]之间时，即满足nums[mid]<target<=nums[n-1]条件，令left=mid+1。

1.1.3.3.2 否则移动right=mid-1

1.1.4 当left<=right迭代完成，以上都不满足时，返回-1。

1.1.5 考虑，当数组nums为空时，直接返回-1。

1.2 复杂度分析

1.2.1 时间复杂度：O(logN)，二分查找的时间复杂度

1.2.2 空间复杂度：O(1)，常数变量

1.3 题解代码

```
class Solution:
    def search(self, nums: List[int], target: int) -> int:
        # nums为空
        if not nums:
            return -1
        # 二分查找
        n = len(nums)
        left, right = 0, n - 1
        while left <= right:
            mid = (left + right) // 2
            if nums[mid] == target:
                return mid
            elif nums[left] <= nums[mid]:
                if nums[left] <= target < nums[mid]:
                    right = mid - 1
                else:
                    left = mid + 1
            else:  # nums[left]>nums[mid]
                if nums[mid] < target <= nums[right]:
                    left = mid + 1
                else:
                    right = mid - 1
        return -1
```

213.打家劫舍2

1. 动态规划

1.1 算法思路

1.1.1 环状排列的房屋，即首尾相接，那么只能在第一个或最后一个中偷窃其中一个。因此将环状排列约化为两个单排排列的问题：

1.1.1.1 偷第一个房子，不偷最后一个，为**nums[:-1]**，最大金额p1

1.1.1.2 偷最后一个房子，不偷第一个，为**nums[1:]**，最大金额p2

1.1.1.3 综合偷窃的最大金额，**Max(p1,p2)**

1.1.2 转移方程：

1.1.2.1 假设n间房子，前n间房子最高偷窃金额为**dp[n]**，前n-1间房子最高偷窃金额为**dp[n-1]**，此时再加一间房子，金额为**num**，则有：

1.1.2.1.1 不抢第n+1间房子，则**dp[n+1]=dp[n]**

1.1.2.1.2 抢n+1间房子，不抢第n间房子，则**dp[n+1]=dp[n-1]+num**

1.1.2.1.3 关于第n间房子是否被偷，存在两种可能：

1.1.2.1.3.1 假设第n间房子没有被偷，则dp[n]=dp[n-1]，那么**dp[n+1]=dp[n]+num=dp[n-1]+num**

1.1.2.1.3.2 假设第n间房子被偷，则**dp[n+1]=dp[n-1]+num**

1.1.2.2 最终方程为**dp[n+1]=max(dp[n],dp[n-1]+num)**

1.1.3 简化空间复杂度：因为dp[n]只与dp[n-1]和dp[n-2]有关系，所以通过变量cur和pre互相交替记录，即可把空间复杂度降为O(1)

1.2 复杂度分析

1.2.1 时间复杂度：O(n)，n为数组长度，需要遍历两次

1.2.2 空间复杂度：O(1)

1.3 题解代码

```python
class Solution:
    def rob(self, nums: List[int]) -> int:
        # 递归打家劫舍1的方法
        def rob1(nums):
            cur = pre = 0
            for num in nums:
                cur, pre = max(pre + num, cur), cur  # dp[n-1]+num和dp[n]的较大值
            return cur

        # 将环形拆分成两个链式
        if not nums:
            return 0
        n = len(nums)
        if n == 1:
            return nums[0]
```

```
        # nums[:-1] 和 nums[1:]
        return max(rob1(nums[:-1]), rob1(nums[1:]))
```

81.搜索旋转排序数组2

1. 二分查找

1.1 算法思路

1.1.1 本题基于**33.搜索旋转排序数组**产生了变化，数组中存在重复元素。

1.1.2 对于数组中重复元素的情况，二分查找时可能存在num[left]==num[mid]==num[right]，此时不能判断出[left,mid]和[mid,right]哪边是有序的，这时left和right都要移动一步。然后在新区间上继续二分查找。

1.1.3 其他要点和33题解法一致。

1.2 复杂度分析

1.2.1 时间复杂度：O(logN)，最差为O(N)

1.2.2 空间复杂度：O(1)

1.3 题解代码

```
class Solution:
    def search(self, nums: List[int], target: int) -> bool:
        # nums为空
        if not nums:
            return False
        # 二分查找
        n = len(nums)
        left, right = 0, n - 1
        while left <= right:
            mid = (left + right) // 2
            if nums[mid] == target:
                return True
            # mid和left和right的元素相等时，left,right各自移动一步
            elif nums[left] == nums[mid] == nums[right]:
                left += 1
                right -= 1
            elif nums[left] <= nums[mid]:
                if nums[left] <= target < nums[mid]:
                    right = mid - 1
                else:
                    left = mid + 1
            else:  # nums[left]>nums[mid]
                if nums[mid] < target <= nums[right]:
                    left = mid + 1
                else:
                    right = mid - 1
        return False
```

153.寻找旋转排序数组中的最小值

1. 二分查找

1.1 算法思路

1.1.1 原数组是生序数组，所以旋转后的数组中，原数组的下标0元素即旋转数组中的最小值。

1.1.2 套用二分查找模板时考虑以下几种情况。初始left=0，right=len(nums)-1，当left<=right时循环，令mid=(left+right)/2。

1.1.2.1 当nums[left]<nums[right]，表示从left到right每个元素都是生序排序的，nums[left]即是最小值。

1.1.2.2 当nums[mid]>nums[right]，表示旋转后从left到mid的元素是生序的，那么原数组的最小值在mid到right之间，所以移动left，继续在mid到right之间寻找最小值。

1.1.2.3 旋转后，如果mid=0或nums[mid-1]>nums[mid]，表示mid的元素就是最小值。

1.1.2.4 当nums[left]<nums[mid]时，表示最小值在left到mid之间，移动right。

1.2 复杂度分析

1.2.1 时间复杂度：O(logN)，二分查找的时间复杂度。

1.2.2 空间复杂度：O(1)

1.3 题解代码

```python
class Solution:
    def findMin(self, nums: List[int]) -> int:
        # 二分查找
        left, right = 0, len(nums) - 1
        while left <= right:
            mid = (left + right) // 2
            # 生序数组
            if nums[left] < nums[right]:
                return nums[left]
            # 0-mid是生序 min值在mid-right之间
            elif nums[mid] > nums[right]:
                left = mid + 1
            # mid=0是生序数组的第一个元素，或 在生序数组中mid-1的元素>mid的元素
            elif mid == 0 or nums[mid - 1] > nums[mid]:
                return nums[mid]
            else:  # mid-right是生序，min在left-mid之间
                right = mid - 1
```

154.寻找旋转排序数组中的最小值2

1. 二分查找

1.1 算法思路

1.1.1 本题在153.寻找旋转排序数组中的最小值的基础上增加了变化，数组nums中存在重复元素。

1.1.2 仍然采用二分查找的方式解题，但有一些变化，考虑以下几点：

1.1.2.1 当nums[mid]==nums[right]时，无法判断mid-right之间的元素是否为排序数组时，但又因为nums[mid]==nums[right]，此时移动right，令right-1，在新区间内继续寻找最小值。

1.1.2.2 当nums[mid]>nums[right]时，最小值的下标在mid-right之间，同时nums[mid]不会是最小值，所以令left=mid+1，在新区间内继续寻找最小值。

1.1.2.3 当nums[mid]<nums[right]时，最小值下标在left-mid之间，同时不能排除mid就是最小值下标，所以令right=mid。

1.1.3 最终当循环结束，返回nums[left]即可。

1.2 复杂度分析

1.2.1 时间复杂度：O(logN)，二分查找时间复杂度，最差O(N)

1.2.2 空间复杂度：O(1)

1.3 题解代码

```python
class Solution:
    def findMin(self, nums: List[int]) -> int:
        if len(nums) == 1:
            return nums[0]
        # 二分查找
        left, right = 0, len(nums) - 1
        while left <= right:
            mid = (left + right) // 2  # left + (right - left ) //2
            if nums[mid] == nums[right]:
                right -= 1
            elif nums[mid] > nums[right]:
                left = mid + 1
            elif nums[mid] < nums[right]:
                right = mid
        return nums[left]
```