第 11 篇

数字在排序数组中出现的次数

问题

统计一个数字k在排序数组中出现的次数。

思路

要充分利用数字已排序这个特点。最基本的思路是，二分查找，找到一个k，然后向前后顺序计数，找到所有k的个数。

如果要更加高效一点的话，可以变换思路，用二分查找的方式，找到首尾的两个k，首尾下标之间的距离就是k的个数。

以查找开头第一个k为例，二分查找分两半，看mid和k的大小比较，如果mid比k大，说明k应该在前一半，则递归去前一半；如果mid比k小，说明k在后一半，则递归去后一半。如果mid和k相等，则判断mid是否是第一个k，就看mid的前一个是否为k，如果不是k，则找到啦，返回mid，如果是，则乖乖去递归前一半。查找最末尾的k同上思路。

注意写的时候要留意边界条件。

代码

先皮一下，python中直接对数组 arr.count(k)就能完成这个要求了（捂脸）

正经代码：

```
# -*- coding:utf-8 -*-
class Solution:
    def GetNumberOfK(self, data, k):
        length = len(data)
        # write code here
        if data and k:
            first_k = self.get_first_k(data, length, k, 0, length - 1)
            last_k = self.get_last_k(data, length, k, 0, length - 1)
            if first_k > -1 and last_k > -1:
                return last_k - first_k + 1
        return 0

    def get_first_k(self, data, length, k, start, end):
        if start > end:
            return -1
        mid = int((start + end) / 2)
        if data[mid] < k:
```

```
            start = mid + 1
        elif data[mid] > k:
            end = mid - 1
        elif data[mid] == k:
            if (mid > 0 and data[mid-1] != k) or mid == 0:
                return mid
            else:
                end =  mid - 1
        return self.get_first_k(data, length, k, start, end)

    def get_last_k(self, data, length, k, start, end):
        if start > end:
            return -1
        mid = int((start + end) / 2)
        if data[mid] < k:
            start = mid + 1
        elif data[mid] > k:
            end = mid - 1
        elif data[mid] == k:
            if (mid < length - 1 and data[mid+1] != k) or mid == length -1:
                return mid
            else:
                start =  mid + 1
        return self.get_last_k(data, length, k, start, end)
```

二叉树的深度

问题

输入一棵二叉树，求该树的深度。从根结点到叶结点依次经过的结点（含根、叶结点）形成树的一条路径，最长路径的长度为树的深度。

思路

遍历整个树，记录下遍历的路径，从而找到一个最长路径，得到深度。这种思路需要的代码量稍大，略微复杂一点。

换个娇爽分析一下情况，如果树只有1个节点，其深度为1；如果根节点只有左子树，则深度为左子树的深度+1；如果只有右子树，则深度为右子树深度+1；如果左右都有，则深度为左右的深度的较大值+1。 按此思路，可写一个递归的方法。

代码

```
# -*- coding:utf-8 -*-
# class TreeNode:
#     def __init__(self, x):
#         self.val = x
```

```
#         self.left = None
#         self.right = None
class Solution:
    def TreeDepth(self, pRoot):
        # write code here
        if not pRoot:
            return 0
        left = self.TreeDepth(pRoot.left)
        right = self.TreeDepth(pRoot.right)
        return max(left, right) + 1
```

平衡二叉树

问题

输入一棵二叉树，判断该二叉树是否是平衡二叉树。

平衡二叉搜索树（Balanced Binary Tree）具有以下性质：它是一棵空树或它的左右两个子树的高度差的绝对值不超过1，并且左右两个子树都是一棵平衡二叉树。

思路

结合上一题，思路是对每个节点，调用 TreeDepth 来判断其左右子树的深度差是否不超过1。 虽然实现简单，但问题在于，对递归判断每个节点时，调用 TreeDepth 会遍历该节点的每个子节点，导致底层的子节点多次重复访问，效率低。

思考是否有只需遍历一次的解法？ 上面的思路中，我们是先判断根节点的树是否平衡，再判断子节点是否平衡，会导致判断根节点时访问过子节点了，判断子节点时又重复访问子节点。这实际上是先序遍历！所以，如果变为后序遍历呢？先访问左右子节点判断是否平衡，再判断根节点是否平衡，在访问子节点后记录子节点的深度，根节点时就可以直接获取此深度值，避免了对子节点的重复访问。

代码

```
# -*- coding:utf-8 -*-
# class TreeNode:
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None
class Solution:
    def IsBalanced_Solution(self, pRoot):
        # write code here
        if not pRoot:
            return True
        # 判断左子树是否平衡，不平衡则直接终止判断，返回false
        is_left = self.IsBalanced_Solution(pRoot.left)
        if not is_left:
            return False
```

```
        # 判断右子树是否平衡，不平衡则直接终止判断，返回false
        is_right = self.IsBalanced_Solution(pRoot.right)
        if not is_right:
            return False
        # 左右均平衡，则计算深度，判断当前为根的树是否平衡
        left_depth = pRoot.left.depth if pRoot.left else 0
        right_depth = pRoot.right.depth if pRoot.right else 0
        pRoot.depth = max(left_depth, right_depth) + 1
        return abs(left_depth - right_depth) <= 1
```

数组中只出现一次的数字

问题

一个整型数组里除了两个数字之外，其他的数字都出现了偶数次。请写程序找出这两个只出现一次的数字。要求：时间复杂度 O(n)，空间复杂度 O(1)

思路

容易想到的方法，用list或set来记住两个数字，把数组读一遍，遇到当前在list里数字就把它去掉，最后剩下的就是两个只出现一次的数字。但这样导致空间复杂度超出限制了。

原书中的思路是，利用了二进制位运算xor的特性。同一个数字的二进制和它自己进行xor，最终一定会相互抵消（=0），所以说，假如这题是只有一个只出现了一次的数字，则可以从头对每个数累计进行xor，由于出现了两次的数都抵消了，最后剩下的就是那个只出现了一次的数字了。

那现在有两个只出现了一次的数字，该如何操作？由于这两个数字肯定不相同，所以对全部数字进行一遍xor之后，结果的值必不为0，而且结果是这两个数字进行xor之后的结果。所以，在结果二进制中，找到为1的一位，说明这两个数字的二进制，在这一位上必定不相同。因此，就根据这一位是1或是0，把整个数组划分为两个小数组，能够保证这两个数字分别出现在两个数组中（其他出现两次的数字，两个相同数字必定会被划到同一个子数组中，也就能抵消），然后两个小数组再各进行一次全面xor，就可以得到这两个数字了。

总共需要先把大数组xor一遍，找到哪一位二进制可用于区分，然后不必物理上把大数组划开，第二趟仍然可以把大数组过一遍，xor时根据这一位是0还是1，分别与不同的累积进行xor即可。

（感觉这个方法也挺折腾的……）

代码

偷懒就用简单的方法来写了……

```python
# -*- coding:utf-8 -*-
class Solution:
    # 返回[a,b] 其中ab是出现一次的两个数字
    def FindNumsAppearOnce(self, array):
        data = set()
        for d in array:
            if d in data:
                data.remove(d)
            else:
                data.add(d)
        return list(data)
```

和为s的两个数字

问题

输入一个递增排序的数组，和一个数字s，在数组中查找两个数，使它们的和正好是s。如果有多对和为s，输出两个数的乘积最小的。

思路

最基本的是从前往后，固定住某一位，然后判断它后面各位与它的和是否等于s，是的话就找到了一对。

但由于这个题只要求找一对即可，不用找全，所以可以有更优化的方法。两个指针分别指向首尾，两个值相加如果大于s，则让后面的指针往前走一走，如果小于s则让前面的指针往后走一走，直到找到和为s，或者指针重合也没找到。

为什么这样可行呢？如果值小于s，为什么不让后面的指针往后走，和也能变大呢？这是因为，后面的指针如果能够往后走（说明它曾经往前走过），是因为它加上数组最小的值之后还是大于s了，所以这时如果让后面指针往后走，结果必定大于s。如果值大于s，为什么不让前面的指针往前走呢？这是因为，如果前面的指针能往前走（说明它曾经往后走过），是因为它加上数组最大的值之后还是小于s了，这时往前走了，结果也必定小于s，没有必要。所以，上述方法虽然看起来略让人怀疑，但是可行的。

代码

```python
# -*- coding:utf-8 -*-
class Solution:
    def FindNumbersWithSum(self, array, tsum):
        if not array or not tsum:
            return []
        front = 0
        rear = len(array) - 1
        while front < rear:
            added = array[front] + array[rear]
            if added == tsum:
                return [array[front], array[rear]]
```

```
            elif added < tsum:
                front += 1
            else:
                rear -= 1
        return []
```

补充

为什么从两端往中间找，找到的第一组满足条件的，就是乘积最小的呢？ 试想

```
xy - (x-a)*(y+b) = xy - xy + bx - ay - ab = -b(a-x) - ay
```

肯定小于0，也就是说xy比xy向里逼近一点的乘积要更小。

如果要找乘积最大的一对，用变量存下当前找到的一对，继续往里直到front和rear重合，每找到新的就覆盖变量，就能找到最里面的满足条件的一对。

和为s的连续正数序列

问题

有了前面的问题，再变得难一点: 不再局限为两个数字的和了，也不局限在给定数组了，而是要在1，2，3.....的正数序列中，找到连续的子序列（至少包含两个数），使得其中的数之和为s；而且不光要找到一对，要找到所有的和为s的连续正数序列。

思路

原始的容易想到的思路就是，从头开始，对每一个数字，从它开始往后找序列，直到找到或者序列的和大于s了，就把前面的指针往后挪一个，继续找。直到指针指向的值大于s的一半了，说明再往后不会有和为s的了，就停止找下去。

这样的思路比较简单，但可能会有一些多余的计算。

仍然延续前面的思路，用两个指针，一个small，一个big。开始时，small指向1，big指向2。我们以s=9为例，一开始{1，2}，和为3，小于s，则让big往后移动，{1，2，3}，再继续{1，2，3，4}，此时和为10，大于s，则让small往后移动，{2，3，4}，等于9，找到一个。然后继续增加big，{2，3，4，5}大于s，让small往后，{3，4，5}大于s，让small往后，{4，5}等于9，找到一个。再让big往后，{4，5，6}大于s，让small往后，{5，6}，此时small已经大于9的一半了，不再继续寻找。

这样不会遗漏吗？比如，当和大于s时，为什么不让big往前走？是因为big之所以到这里，是由于它之前的和小于s了，再让big往前，和还是小于s；当小于s时为什么不让small往前走？是因为small之所以到这里，是由于它之前的和大于s了，再让small往前，和还是大于s。

> 更多可了解 https://blog.csdn.net/qq_41822235/article/details/82109081 （p.s.拿python刷题真的是写代码一时爽，内存和耗时伤不起啊，给耗时几毫秒的c++跪了）

代码

```
# -*- coding:utf-8 -*-
```

```python
class Solution:
    def FindContinuousSequence(self, tsum):
        if not tsum:
            return []
        result = []
        small = 1
        big = 2
        current_sum = small + big
        while small < (tsum + 1) / 2:
            while current_sum < tsum:
                big += 1
                current_sum += big
            while sum(range(small, big + 1)) > tsum:
                current_sum -= small
                small += 1
            if current_sum == tsum and small != big:
                result.append(list(range(small, big + 1)))
                big += 1
                current_sum += big
        return result
```