Arrays

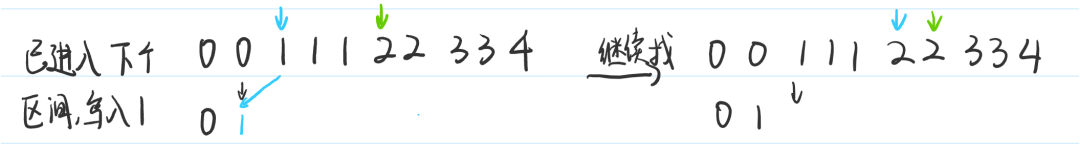
Two Pointers:

26. Remove Duplicates From Sorted Array

Two Pointers; In-place Modification

解法一：Three Pointers

一个pointer从 0 ~ nums.length 进行写入，两个pointer在前方不断寻找一个一个数字区间，当找到新区间或走到终点时让第一个pointer将旧区间的值写入。



解法二：Two Pointers

可将此解法化简为两个pointers，**一个读入一个写入**。因为负责写入的pointer已经存储了数字，读入的pointer可以通过这个存储确定是否已经进入了新的数字区间，从而让第一个pointer继续写入。

---

27. Remove Element

Two Pointers; In-place Modification

解法一：死算

用循环进行数字移位

解法二：Two Pointers

时间复杂度：O(n)； 空间复杂度：O(1)

读入pointer碰到val则跳过，否则让写入pointer写入该值。

88. Merge Sorted Array

Two Pointers

> Given two sorted integer arrays nums1 and nums2, merge nums2 into nums1 as one sorted array.
>
> Note: The number of elements initialized in nums1 and nums2 are m and n respectively. You may assume that nums1 has enough space (size that is greater or equal to m + n) to hold additional elements from nums2.

解法一：死算

用循环进行数字移位

解法二：Three Pointers

参考

2个pointer分别在两个数组的前方读取，一个pointer在num1里进行写入，每次写入时判断，先写小的那个。

```
class Solution {
    public void merge(int[] nums1, int m, int[] nums2, int n)
{
        int cur1 = m - 1;
        int cur2 = n - 1;
        int curf = m + n - 1;
        while (cur1 >= 0 && cur2 >= 0)
        {
            if (nums1[cur1] < nums2[cur2])
            {
                nums1[curf] = nums2[cur2];
                curf--;
                cur2--;
            }
            else
            {
                nums1[curf] = nums1[cur1];
                curf--;
                cur1--;
            }
        }
        while (cur1 >= 0)
            nums1[curf--] = nums1[cur1--];
        while (cur2 >= 0)
```

```
                nums1[curf--] = nums2[cur2--];
        }
    }
```

---

283. Move Zeros

Two Pointers

解法一：Two Pointers

两个Pointers分别进行读和写，当读到非零数则让写入Pointer写入。最后将后面所有数字填补为0即可。

---

905. Sort Array by Parity

Two Pointers; Sort

解法一：Two Pointers & Swap

两个指针，一个读奇数一个读偶数。奇数指针尽可能地保持在偶数指针之前。奇数指针找到最近的奇数，偶数指针找到最近的在奇数指针之后的偶数。如果偶数指针已经超过了数组边界，则说明不需要再对数组做任何改动，返回当前数组即可。否则将奇数和偶数指针所指的两个数交换，并继续重复以上步骤。

解法二：Customized Sort

参考

建立Integer数组，使用Java自带Customized Sort完成排序，最后赋值回A，返回A。

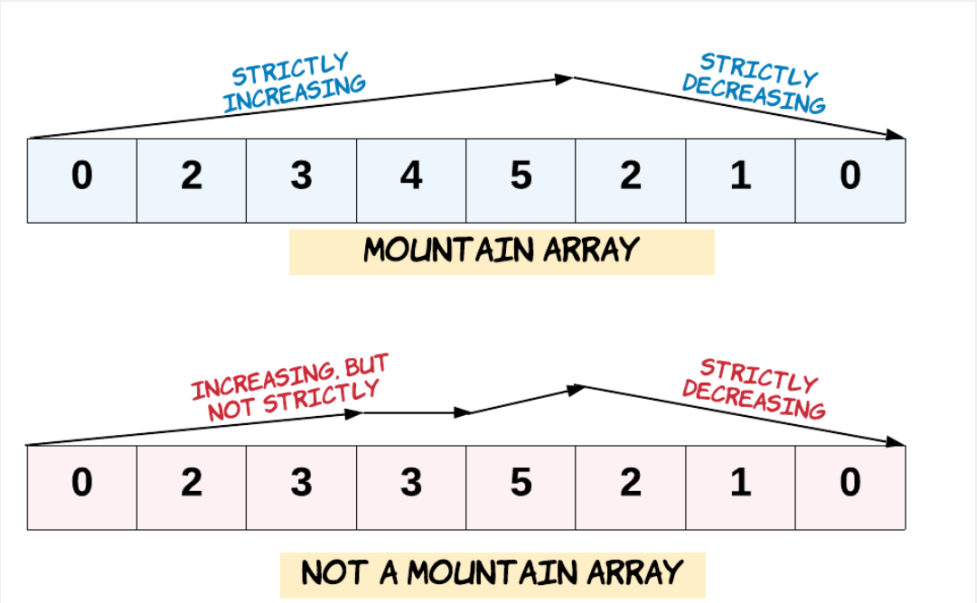
---

941. Valid Mountain Array

Two Pointers

Given an array A of integers, return true if and only if it is a valid mountain array. Recall that A is a mountain array if and only if:

1. A.length >= 3
2. There exists some i with 0 < i < A.length - 1 such that: A[0] < A[1] < ... A[i-1] < A[i] A[i] > A[i+1] > ... > A[A.length - 1]



解法一：循环遍历

直接循环遍历整个数组，记录上升或下降的改变。当且仅当数组先升再降且改变一次升降情况时数组符合条件。比较简单，要判断的情况有点多。

解法二：Two Pointers

参考

设置两个pointer，一个从后往前，到下一位数开始变小时停止，一个从前往后，到下一位数开始变小时停止。如果两个指针相遇则说明数组符合条件

```
class Solution
{
    public boolean validMountainArray(int A[])
    {
        int j = A.length - 1;
        int i = 0;
        while (j - 1 >= 0 && A[j - 1] > A[j])
            j--;
        while (i + 1 < A.length && A[i + 1] > A[i])
            i++;
        if (i == j && i != 0 && i != A.length - 1)
            return true;
```

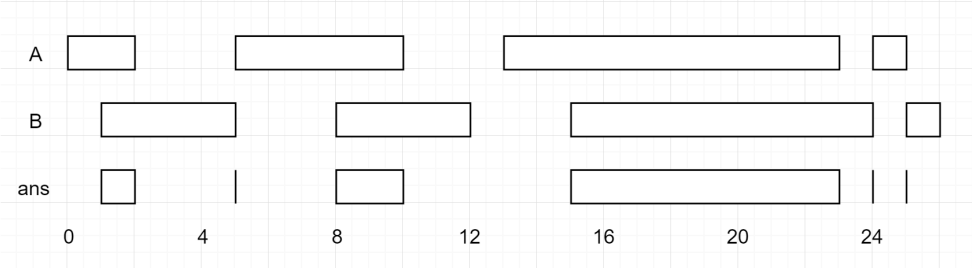
```
            return false;
        }
    }
```

---

986. Interval List Intersections

Two Pointers

> Given two lists of closed intervals, each list of intervals is pairwise disjoint and in sorted order.
>
> Return the intersection of these two interval lists.



解法一：Two Pointers分类判断

时间复杂度：O(m + n)；空间复杂度：O(m + n)

两个指针分别在A数组和B数组的某个Interval上，首先判断A指针在前还是B指针在前。其次判断两个Interval是否有交集。在前的Interval的末端需要比在后的Interval的前端靠后才能有交集。有交集之后还需要判断靠后的Interval的末端在前还是靠前的Interval末端在前。判断完成之后把对应的交集放进答案集里，然后把靠前的指针后移。

解法二：Two Pointers精简判断

时间复杂度：O(m + n)； 空间复杂度：O(m + n)

参考

判断的过程可以简化，首先取得两个Interval的前端中靠后的一个，再取得两个Interval后端之中靠前的一个，判断两者是否相交。如果相交前端结果应该小于等于后端结果，而这个前后端就是交集的前后端。

之后把靠前的指针后移。

```
class Solution {
public:
                                                vector<vector<int>>
intervalIntersection(vector<vector<int>>&                    A,
vector<vector<int>>& B) {
        int i = 0;
        int j = 0;
        vector<vector<int>> ans;
        while (i < A.size() && j < B.size())
        {
            int st = max(A[i][0], B[j][0]);
            int ed = min(A[i][1], B[j][1]);
            if (st <= ed)
                ans.push_back(vector<int> {st, ed});
            if (A[i][1] <= B[j][1])
                i++;
            else
                j++;
        }
        return ans;
    }
};
```

1089. Duplicate Zeros

Two Pointers

解法一：死算

用循环进行数字移位

解法二：Two Pointers

参考

通过遍历查找数组中0的数量来确定数字需要移动多少位。从后往前进行写入覆盖。

假定数组可以无限长，则写入pointer从数组的理论最长位置（加上复制的0之后）开始，读入pointer从真实长度位置开始，一旦遇到0则让写入pointer多写一位。

写入pointer一直在移动，但是**仅当进入数组真实长度位置后才开始正式写入。**

```
class Solution {
public:
    void duplicateZeros(vector<int>& A) {
         int n = A.size(), j = n + count(A.begin(), A.end(),
0);
        for (int i = n - 1; i >= 0; --i) {
            if (--j < n)
                A[j] = A[i];
            if (A[i] == 0 && --j < n)
                A[j] = 0;
        }
    }
};
```

简单遍历：

1. Two Sums

HashMap

解法一：二重循环遍历查找

时间复杂度：O(n^2)； 空间复杂度：O(1)

解法二：HashMap保存Complement值

时间复杂度：O(n)； 空间复杂度：O(n)

448. Find All Numbers Disappeared in an Array

HashMap

解法一：用加法加上不可能出现的数做标记

时间复杂度：O(n)； 空间复杂度：O(1)

一道很有意思的题目，对时间和空间复杂度都有要求。因为题目给定了数字的范围，因此可以用特殊的方法来做标记。 因为每个数都小于等于n，所以每次遇到一个数，就给数组对应这个数下标的数加上n。 最后再遍历一遍数组，对每个数 div n，如果得到0或者这个数是n就说明这个数没有在数组里出现过。

解法二：用负数做标记

参考 *时间复杂度：O(n)； 空间复杂度：O(1)*

遇到一个数就把数组对应这个数 - 1的为下表的数变成负数来做标记，最后遍历，那些没有变成负数的下标 + 1 即为没有出现过的数。

1229. Replace Elements with Greatest Element on Right Side

解法一：从后往前循环遍历

时间复杂度：O(n)； 空间复杂度：O(1)

一边记录遇到的最大值一边进行替换，因此仅需单重循环

Sort：

169. Majority Element

Sort; HashMap

977. Squares of a Sorted Array

Sort; Two Pointers

解法一：直接平方后重新排序

时间复杂度：O(nlogn)； 空间复杂度：O(1)

解法二：Two Pointers，存储进新数组

参考

时间复杂度：O(n)； 空间复杂度：O(n)

1051. Height Checker

Sort

其他：

9. Palindrome Number

387. First Unique Character in a String

HashMap

414. Third Manimum Number

485. Max Consecutive Ones

Find Biggest Num

1295. Find Numbers with Even Number of Digits

Easy Counting

1346. Check if N and its Duplicate Exists

HashMap