Arrays

Two Pointers:

15. 3Sum

Two Pointers

解法一：先确定Two Sum，用Two Pointers解决Two Sum

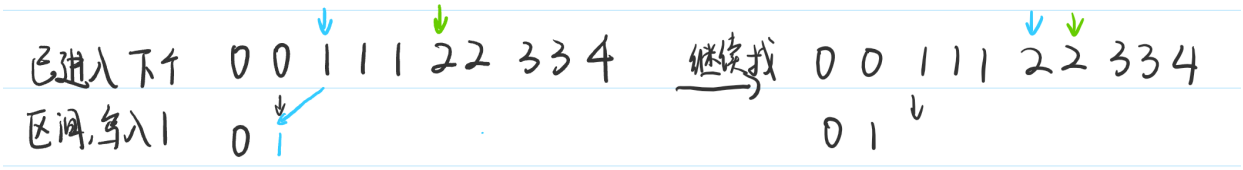
先把数组排序，接下来一重循环分别确定每个点为其中一个数，这样我们知道了剩下两个数的和。在剩下的数字中（避免重复），我们可以用两个Pointer分贝从高处和低处开始进行配对，每次如果结果小了我们就移动低处，如果数字大了我们就移动高处，直到两者相遇。我们同时要避免重复的数字，所以如果发现一个配对，我们就要跳过那些一样的数字。

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。

```java
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--];
    }
}
```

264. Ugly Number II

Three Pointers

解法一：Three Pointers

丑数只包含Prime Factor是2, 3, 5的数字，所以如果我们有之前的所有丑数，尝试将那些数字乘2, 3 或5我们就可以得到之后的丑数。接下来就是排序的问题，这个问题可以被转化成一个类似合并三个有序列表的问题。

我们先考虑对一个之前的丑数 k，2k，3k，和 5k 都可以是之后的丑数，首先可以确定的是这三个数自己是有序的，但是可能有别的数字m，有 3m 比 2k 小。但是我们可以确定 2k 一定比 2(k - 1)要大，所以我们只要建立三个Pointer，分别进行乘2，3和5的操作，每次找最小的数，然后一旦这个数经过了这个pointer的处理，我们就可以把pointer前移，直到我们的数字符合要求为止。

解法二：Three Pointers 简化比大小过程

参考

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。

287. Find the Duplicate Number

Two Pointers

解法一：快慢Two Pointers

参考

这可以转化成一个Linked List问题，这真的完全想不到。因为题目限制了数组中的数只可能是1-n之间的数，所以每个数字既是数值又可以是index。因此链表每个数字的下一个数字就是这个数字作为index所代表的数。这样数组中的任何重复的数字就会让链表出现环路，因为重复的数字会让Pointer走向重复的道路。

这之后就可以转化成142. Linked List Cycle II用快慢两个指针来寻找环路的开头。

解法二：二分查找

参考

感觉这个解法也很妙，二分查找一开始也没想出来能确定到底要找的数字在左边还是在右边。但是可以这样，每次中位数往左边数，如果有超过一半的数字比当前数字小（或等于当前数字），说明那个重复的数字一定在这半边，否则就在另一半边。

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

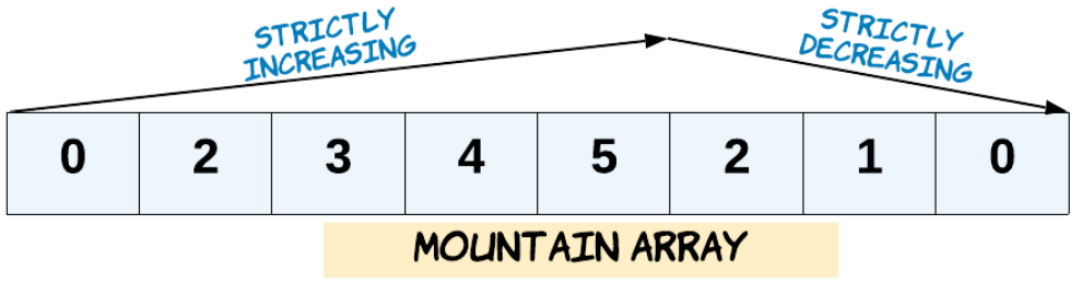
参考

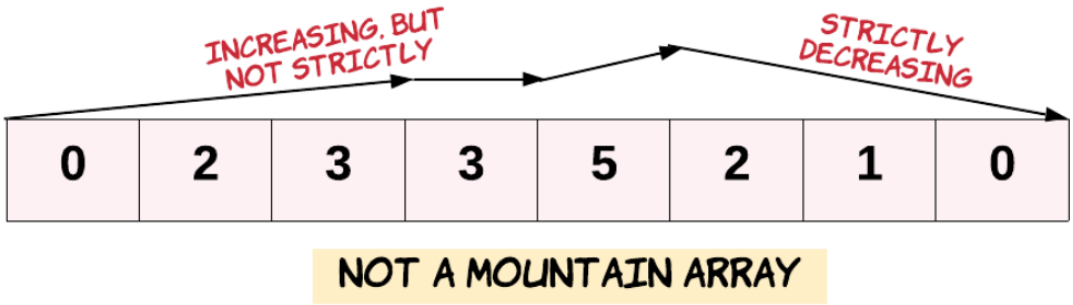
建立Integer数组，使用Java自带Customized Sort完成排序，最后赋值回A，返回A。

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

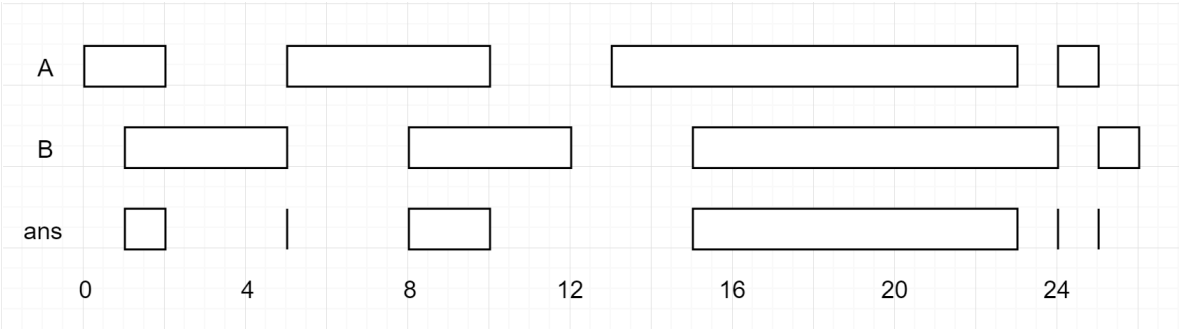
设置两个pointer，一个**从后往前**，到下一位数开始变小时停止，一个**从前往后**，到下一位数开始变小时停止。如果两个指针相遇则说明数组符合条件

```
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;
        }
    }
};
```

1471. The k Strongest Values in an Array

Two Pointers; Sort

解法一：排序 + Two Pointers处理

首先将数组排序，找到中位数（注意题目定义的中位数和正常的定义不同），然后从两头开始的数字一定是和中位数差值最大的，让两个Pointers从两头分别开始，像Merge Sorted Arrays一样进行比较，取得前k个差值最大的数字。当差值一样的时候，因为数组已经经过排序，所以肯定是使用从尾端开始的Pointer指向的数字。

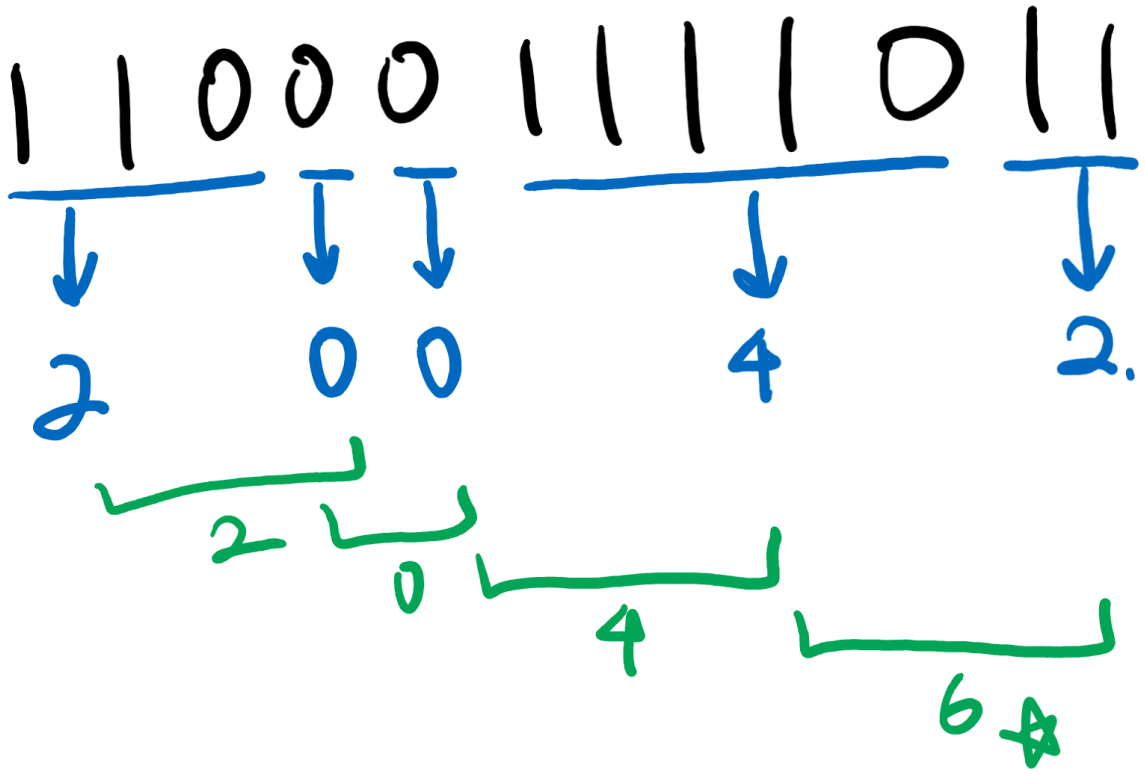
1493. Longest Subarray of 1's After Deleting One Element

Sliding Window

解法一：压缩所有的1

因为我们需要连续的1的长度，我们可以先把已经连续的1压缩，然后再看最大的值能到多少。每次取一个sum来对1进行加和，一旦我们遇到一个0就把这个数字放进ArrayList中，并把sum重设成0，注意如果这个时候sum本来就是0，0也会被存放进ArrayList中。结束遍历的时候最后的sum也要放进ArrayList中。

最后我们遍历ArrayList，设置一个长度为2的窗口，比较窗口中数字的和，找到最大的和。这其实就代表了把两组1中的0去掉能达到的最大长度。而且因为我们只能去掉一个0，所以有多个连续的0的时候就不能跳过这些0。



解法二：Sliding Window

参考

怎么说呢，也许Sliding Window也能勉强算Two Pointers吧（大雾）。

其实既然都用到Sliding Window了就可以整个题目都用Sliding Window，而且这种Sliding Window的Window长度可以变化，因为我们只需要保证窗口内的0的数量最多不超过1。我们只需要加入一个计数器记录窗口中0的数量，如果超过了我们就需要缩减窗口的尾部直到0的数量降低为止。

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)

66. Plus One

解法一：直接加，最后进位分开考虑

参考

最后的额外进位只有当所有数字都是9的时候才会发生，所以肯定是1开头后面全是0，建立一个新数组即可。

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)

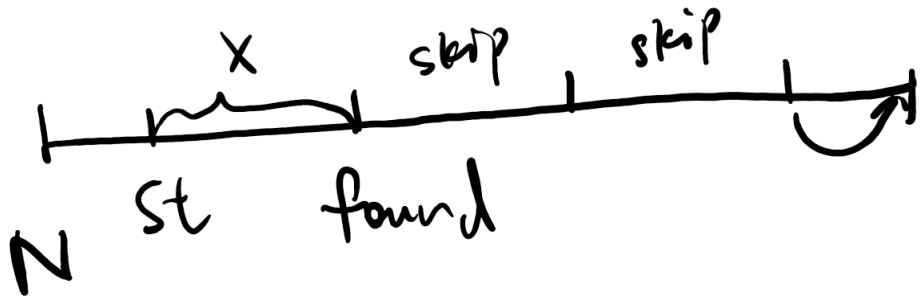
遇到一个数就把数组对应这个数 - 1的为下表的数变成负数来做标记，最后遍历，那些没有变成负数的下标 + 1 即为没有出现过的数。

957. Prison Cells After N Days

解法一：寻找循环

参考

如果按照普通算法模拟就会超时，因此要记录并找到整个循环，最后跳过那些重复的步骤。可以把数组转换成字符串来存储，一旦找到一个重复的状态则跳过等倍数的状态，最后算出最终状态。



解法二：确定14天是一个循环

参考1

参考2

额，怎么说，其实我还是没太懂为什么是14天。总之就是说1号和8号从第一天开始就是一定是0的，然后2，4，6号的上一天决定3，5，7号的下一天，反之也是。接下来因为3，5，7又决定下一天的2，4，6，所以2，4，6，可以决定2天之后的自己。因此三个数字的变化决定了整个数组状态的变化。所以一共2^3 = 8种循环，每个循环是两天，所以最大的循环可能是16天。经过尝试发现14天一个循环，因此取余数即可。

1299. Replace Elements with Greatest Element on Right Side

解法一：从后往前循环遍历

参考

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。因为每次需要的信息都只包括这个数字之后的数字，且不需要这个数字之前的数字，所以可以从后往前进行处理，也不需要回来再看之前的数据。

1492. The kth Factor of n

解法一：直接遍历前半部分Factor

先遍历前半部分Factor，如果k小于这个长度则我们直接找到了第k个factor，否则我们需要看k是否在另一半里。注意如果n是完全平方数则总的factor的数量是奇数，所以需要分类讨论。但是最后我们可以找到和k对称的那个factor并将n除以那个factor得到第k个factor。

1496. Path Crossing

解法一：直接模拟记录到达的所有结点

1497. Check If Array Pairs Are Divisible by k

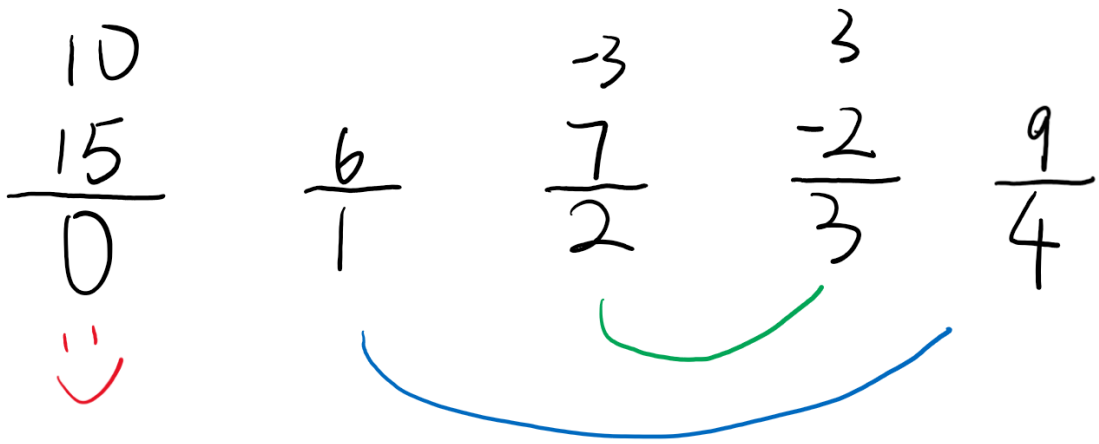
解法一：计算余数配对

当且仅当 (a + b) mod k = 0时，我们可以将两个数字配对。所以可以允许两种情况：

1. a mod k + b mod k = 0
2. a mod k + b mod k = k

因此我们只需要计算每种余数的出现频率，保证余数为0的数有偶数个，剩下其他的余数的数量能一一配对。

另外C++中对负数的余数计算和数学定义不同，因此对于C++，负数的余数计算完之后需要再加一下k，从负的加回正的。



1503. Last Moment Before All Ants Fall Out of a Plank

解法一：蚂蚁转向和没转向是一样的

题目中虽然提到了蚂蚁转向，但是其实蚂蚁转向了之后速度不变所以还是蚂蚁。并不需要考虑转向，只需要找最大最小值即可。

其他：

9. Palindrome Number

344. Reverse String

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

1464. Maximum Product of Two Elements in an Array

1470. Shuffle the Array

1480. Running Sum of 1d Array

1491. Average Salary Excluding the Minimum and Maximum Salary

Find Biggest Num