1.概述

1.1.数组简介

数组是所有人闭着眼都知道的基本数据结构。在LeetCode中官方统计是1000多道与数组有关。不过，数组是很多高级算法的载体，例如排列组合、集合、动态规划等等，因此题目的难度跨度非常大，如果数组的题不会，不代表你不懂数组，可能是需要学习一些高阶内容。数组会贯穿我们整个算法课，包括后面的高级算法课。这里先学习一些基础的。

我们说算法要找到根，要看透本质，那本质到底是啥呢？其实就是数组的增删改查，其实任何数据结构的基本操作都是这几个，大部分题目都是基于这几个操作的拓展，因此将其搞清楚就掌握一半了。

增删改查看似简单，但是面试的时候却非常危险，经常会翻车。例如在任一位置增加元素，或者在已排序数组中插入一个元素时，可能在首位置插入错误，可能尾部插入错误，可能首位置好了，尾部又不行了等等，删除也是类似情况。涉及到数组少不了要考虑好元素存在哪里，所以在写很多高级算法问题也常常因为这些问题拿不准而出错。特别是远程面试时要，明明就要成功了，就是不对，想S的心都有。所以准确写出能适用各种场景的添加方法和删除目标元素的方法本身就是非常重要的问题。

另外，元素添加可以扩展出数组合并的系列问题和进制系列问题，元素删除可以扩展出重复项处理的系列问题，元素的查找可以引出奇偶、大小(就是排序)等的系列问题等等，而这些问题就是面试喜欢考的题目。

`## 1.2.学习目标

1.下载算法工程alg49_1，搭建本地开发环境，下载地址 https://gitee.com/zh-alg/alg49_1

2.理解数组存储的特点，理解在数组种增加和删除元素如何移动元素

3.理解双指针的作用和基本策略

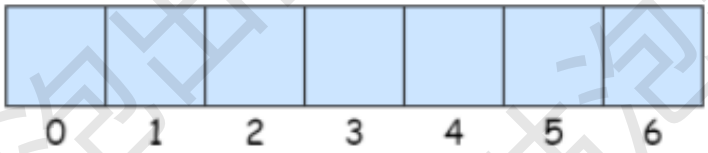
4.掌握基础过关部分的题目

2.数组基础

2.1 数组的概念

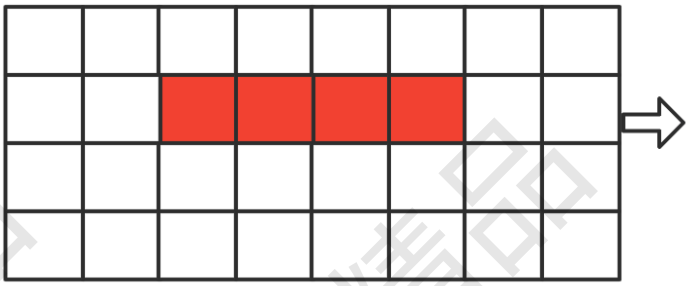
数组是线性表最基本的结构，特点是元素是一个紧密在一起的序列，相互之间不需要记录彼此的关系就能访问，例如月份、星座等

数组会用一些名为 索引 的数字来标识每项数据在数组中的位置，且在大多数编程语言中，索引是从 0 算起的。我们可以根据数组中的索引，快速访问数组中的元素。



数组有两个需要注意的点，一个是从0开始记录，也就是第一个存元素的位置是a[0]，最后一个是a[length-1]。

其次，数组中的元素在内存中是连续存储的，且每个元素占用相同大小的内存。



数组	“g”	“u”	“p”	“a”	“o”
索引	0	1	2	3	4
内存地址	2000	2001	2002	2003	2004

另外一个是数组空间不一定是满的，100的空间可能只用了10个位置，所以要注意数据长度size和数组长度length是不一样的。在写代码的时候必须要判断并处理size和length之间的关系，否则你的代码就是不合格的。

2.2 数组基本操作

在面试中，数组大部分情况下都是int类型的，所以我们就用int类型来实现这些基本功能。

(1)数组创建和初始化

代码位置 unit1_array.ch1.basic.BasicCreateArray

创建一维数组的方法如下：

```
int[] arr = new int[10];
```

初始化数组最基本的方式是循环赋值：

```
for(int i = 0 ; i < arr.length ; i ++)
   arr[i] = i;
```

但是这种方式在面试题中一般不行，因为很多题目会给定若干测试数组让你都能测试通过，例如给你两个数组 [0,1,2,3,5,6,8] 和 [1,4,5,6,7,9,10] 。那这时候该如何初始化呢？此时显然不能用循环赋值了，更多的是采用下面这种方式：

```
int[] arr = new int[]{0,1,2,3,5,6,8};
//这么写也行：
int[] nums = {2, 5, 0, 4, 6, -10};
```

如果要测试第二组数据，直接将其复制替换就行了。这种初始化方式很简单，在面试时特别实用，但是务必记住写法，否则面试时可能慌了或者忘了，老写不对，这会让你无比着急。因此我们练习算法的一个目标就是熟悉这些基本问题，要避免阴沟里翻船。

另外要注意上面在创建数组时大小就是元素的数量，是无法再插入元素的，如果遇到相关场景就考虑如何调整了。

作业1：

将上面这种赋值方法背下来。是的！这种内容要背下来，没什么好说的。

这里下面代码还有两个问题要说明。

```
int[] arr = new int[10]
```

1.数组声明之后默认值都是0，对于上面代码，如果你什么都不做，10个位置的默认值都是0。这里有些人会想我该怎么知道哪些位置是有效的数据，哪些是无效的呢？有的会想是否可以先全部初始化为-1，然后有实际值的位置再给出有效数字呢？问题是你怎么知道-1是无效的？对于大部分算法题，一般是通过一个变量来指定的，也就是下面这条。

2.一般通过其他方法处理数组的时候需要指定其有效元素的数量或者起止位置，例如上面虽然申请了10个空间的数组，但实际可能只存了3个数，这是就要通过额外的变量来指定有几个元素了，例如用size=3，还有些材料用capacity，这个都可以。而有效元素size的数量一定要小于等于arr.length。所以在写算法的时候，如果遇到size一定先判断是否小于arr.length，然后再写算法代码。

还有一种方式是指定有效元素的起止位置，例如 search(arr,2,5),也就是从第2到第5个位置的元素是有效的，这种方式在二分查找、树存储到数组等场景有大量的应用。

(2) 查找一个元素

代码位置：unit1_array.ch1.basic.BasicSearch

为什么数组的题目特别多呢，因为很多题目本质就是查找问题，而数组是查找的最佳载体。很多复杂的算法都是为了提高查找效率的，例如二分查找、二叉树、红黑树、B+树、Hash和堆等等。另一方面很多算法问题本质上都是查找问题，例如滑动窗口问题、回溯问题、动态规划问题等等等都是在寻找那个目标结果。

查找的实现策略也很多，例如先排序再查找等等。这里只看两个简单的情况：根据索引查找和根据元素线性查找。

根据索引位置查找的实现为：

```
/**
* @param arr
* @param index 要查找的位置
* @return
  */
  public static int findByIndex(int[] arr,int size, int index) {
  if (index < 0 || index > size-1)
     throw new IllegalArgumentException("Add failed. Require index >= 0 and index <
size.");
  return arr[index];
  }
```

更多的时候是需要查找目标元素所在的位置：

```
/**
* @param size 已经存放的元素容量
* @param key  待查找的元素
  */
  public static int findByElement(int[] arr, int size, int key) {
  for (int i = 0; i < size; i++) {
       if (arr[i] == key)
          return i;
  }
   return -1;
  }
```

(3) 增加一个元素

增加和删除元素是数组最基本的操作，看别人的代码非常容易，但是自己手写的时候经常bug满天飞。能准确处理游标和边界等情况是数组算法题最基础重要的问题，没有之一！

这个问题没有特别好的套路，不同场景需要灵活设置，所以务必自己亲手能写一个在有序数组种增加元素和删除元素的方法。这也是我们这一章最重要的任务。

该题目不要感觉挺简单就不写，其中遇到的问题在所有与数组有关的算法题中都会遇到。

面试冒汗时别怪没提醒！！！！

增加一个元素也有两种常见的情况：在指定位置添加；或者给出一个有序数组和一个元素让你在正确的位置添加。

如果是给出索引的，由于在数组中添加元素大部分情况下需要移动元素，所以干脆就从后向前一边遍历，一边移动，找到对应位置之后直接赋值就行了。另外一定要注意检测是否越界了，这个是数组的基本功，如果考虑不周就直接玩完。

①在指定位置添加元素

代码位置：unit1_array.ch1.basic.BasicAddByIndex

这里需要知道数组的引用，数组中已存放元素数量，插入位置和待插入的元素四个参数。有些材料会自定义一个对象来将多个入参封装在一起，但是为了不给面试制造障碍，我们就用最基本的方式全部手写。

```
    /**
     * @param arr   数组
     * @param size  已经存放元素的数量
     * @param index 索引值，从0开始
     * @param key   插入的元素
     */
    public static void addByIndex(int[] arr, int size, int index, int key) {
        if (size >= arr.length - 1) {
            throw new IllegalArgumentException(" array is full.");
        }
        if (index < 0 || index >= arr.length - 1)
```

```
            throw new IllegalArgumentException("Add failed.");
        for (int i = size - 1; i >= index; i--)
            arr[i + 1] = arr[i];
        arr[index] = key;
        size++;
    }
```

② 将给定的元素插入到有序数组的对应位置中

代码位置：unit1_array.ch1.basic.BasicAddByElement

由于数组的特性，我们少不了查找和移动元素，这就有两种处理方式:一种是先查找位置，再移动数据并插入。另一种是从后向前一边移动一边对比查找，找到位置直接插入。从效率上看后者是要好一些，但是前一种面试时更容易想到，所以我们这里手写前一种，后一种作为练习请读者自行实现。

在中间位置插入非常容易，貌似一个for循环找一下就搞定了，但是这个题如果面试直接让运行调试，我相信大部分还是会挂，为什么呢？因为必须考虑数组头和数组尾部插入时的情况，下面的实现请读者先不看，自己先自行实现一下addByElementSequence()方法试试。

测试方法：

```
    private static void addTest() {
        //通过元素有序插入
        int[] arr = new int[20];
        arr[0] = 3;
        arr[1] = 4;
        arr[2] = 7;
        arr[3] = 8;
//中间位置插入
        addByElementSequence(arr, 4, 6);
        printList("通过元素顺序插入", arr, 5);
//尾部插入
        addByElementSequence(arr, 5, 9);
        printList("通过元素顺序，尾部插入", arr, 6);

//      测试元素有序并且在表头插入
        addByElementSequence(arr, 6, 0);
        printList("通过元素顺序，尾部插入", arr, 7);
    }
```

其中为了便于遍历输出数组，我们先定义一个方法：

```
    public static void printList(String msg, int[] arr, int size) {
        System.out.println("\n通过" + msg + "打印");
        for (int i = 0; i < size; i++) {
            System.out.print(arr[i] + " ");
        }
    }
```

如果你的插入方法正确执行，打印的结果应该 为：

```
通过通过元素顺序插入打印
3 4 6 7 8
通过通过元素顺序，尾部插入打印
3 4 6 7 8 9
通过通过元素顺序，首部插入打印
0 3 4 6 7 8 9
```

这个例子看上去很简单是不？你一定要亲自写一个试试，因为在写的过程中你会发现前中后都能成功执行还是要费工夫的。

```
/**
     * @param arr
     * @param size    数组已经存储的元素数量
     * @param element 待插入的元素
     * @return
     */
    public static int addByElementSequence(int[] arr, int size, int element) {
        if (size >= arr.length)
            throw new IllegalArgumentException("Add failed. Array is full.");
        int index = size;
        //找到新元素的插入位置
        for (int i = 0; i < size; i++) {
            if (element < arr[i]) {
                index = i;
                break;
            }
        }
        //元素后移
        for (int j = size; j > index; j--) {
            arr[j] = arr[j - 1]; //index下标开始的元素后移一个位置
        }
        arr[index] = element;//插入数据
        return index;
    }
```

作业

自己实现一个插入的方法，能够在数组的前中后分别插入。

该题目不要感觉挺简单就不写，其中遇到的问题在所有与数组有关的算法题中都会遇到。

面试冒汗时别怪没提醒！！！！

(4) 删除一个元素

删除同样存在根据索引位置直接删或者先查找元素位置再删除两种情况。

①根据索引位置

```
/**
```

```
* @param arr
* @param size 数组元素数量
* @param index 删除位置
* @return
*/
public static int removeByIndex(int[] arr, int size, int index) {
 if (index < 0 || index >= size)
    throw new IllegalArgumentException("Remove failed. Index is illegal.");
 int ret = arr[index];
 for (int i = index + 1; i < size; i++)
     arr[i - 1] = arr[i];
     size--;
     return ret;
}
```

② 先查找元素再删除元素。

这里需要注意的是不能一边从后向前移动一边查找，因为元素可能不存在。所以要分为两个步骤，先查是否存在元素，存在再删除。可以直接复用前面的代码：

```
/**
* 从数组中删除元素e
* @param arr
* @param size 数组
* @param key
*/
public static void removeElement(int[] arr, int size, int key) {
int index = findByElement(arr, size, key);
if (index != -1)
 remove(arr, size, index);
}
```

但是考虑到我们面试时一般只会直接写一个方法，所以我们合并在一起可以这么写：

```
    /**
    * 从数组中删除元素e
    * @param arr
    * @param size 数组
    * @param key
    */
   public static void removeByElement(int[] arr, int size, int key) {
       int index = -1;
       for (int i = 0; i < size; i++) {
           if (arr[i] == key) {
               index = i;
               break;
           }
       }
       if (index != -1) {
```

```
            for (int i = index + 1; i < size; i++)
                arr[i - 1] = arr[i];
            size--;
        }
    }
```

最后提供一个测试类:

```
    private static void deleteTest() {
        int[] arr = new int[]{2, 3, 4, 9, 10, 11, 12};
        //根据索引位置删除
        removeByIndex(arr, 7, 2);
        printList("根据索引删除", arr, 6);
        removeByElement(arr, 6, 2);
        printList("根据索引删除", arr, 5);
    }
```

作业

同样要自己实现一个在元素有序的数组种删除特定元素的方法，能够在数组有和没有元素的时候都要正确输出。

第三遍了：

该题目不要感觉挺简单就不写，其中遇到的问题在所有与数组有关的算法题中都会遇到。

面试冒汗时别怪没提醒！！！！

3 java中的数组

你平时写代码的时候是否注意过，jdk里竟然有三个数组相关的类：Array、Arrays和ArrayList。这三个看上去都是数组结构的线性表，但是有啥区别呢?

我们关注这个高级方法，主要是因为其为我们提供了reverse()，sort()等方法，在解决复杂问题的时候可以直接用，不必自己已经解决了大部分问题，但是reverse等基础问题还要写，最后代码又多又乱。简洁的代码也能让面试官喜欢。

3.1 Array与Arrays类的区别

在java中的这两个类有点奇怪，它们不是在一个包里的，Arrays在java.util包下，这个包是我们经常使用的各类基础工具。而Array不是其单数形式，而且也不在java.util包下，而是在java.lang.reflect.包下（在java.sql包下也有一个，先不管了）。这两个到底啥关系呢?

既然是在reflect包下，那一定是和反射有关系了，反射要获得什么呢？动态创建和访问的方法，所以Array就是为了外部能够动态创建和访问 Java 数组的方法而提供的一个类，其主要方法如newInstance，getByte等都是为了方便反射操作而提供的。而Arrays的路径是 java.util.Arrays，有自定义的public 方法来操作数组（比如排序和搜索等）。此类还包含一个允许将数组作为列表来查看的静态工厂。除非特别注明，否则如果指定数组引用为 null，则此类中的方法都会抛出 NullPointerException。

结论就是两者其实没有半毛钱关系，我们用到数组的时候放心的使用Arrays就行了。

而Arrays类里有几个特别重要的方法：sort()和binarySearch()，在比较复杂的题目中我们可以直接用的，关键时刻甚至能救你一命。

3.2 Arrays与ArrayList类的区别

ArrayList 也在java.util下，是 java 集合框架中比较常用的数据结构。我们先看其定义：

```
public class ArrayList<E> extends AbstractList<E>
        implements List<E>, RandomAccess, Cloneable, java.io.Serializable
       }
```

通过这个定义我们就看到，其功能远比Arrays强大。它继承自 AbstractList，实现了 List 接口。底层基于数组实现容量大小动态变化,允许 null 的存在。同时还实现了 RandomAccess、Cloneable、Serializable 接口，所以 ArrayList 是支持快速访问、复制、序列化的。因为其功能强大，特别是支持动态扩容等特性，这个类在我们的业务代码里用的更多。

ArrayList的get方法是基于迭代器来做的，其设计也非常精妙，这个我们在后面梳理递归和迭代器的时候专门分析这个问题。

现在先轻松一下，看看Arrays的使用方法吧。

3.3 Arrays的使用

Arrays类提供了几个非常有用的方法。如果熟练掌握的话，对我们写出简洁优美的算法大有裨益。

Arrays提供的方法都是静态的，比较重要的有：

① void Arrays.sort()

void Array.sort(Object[] array)

功能是对数组按照升序排序，在处理复杂算法的时候可以直接使用，这就减少了很多麻烦。

```
int[] nums = {2,5,0,4,6,-10};
Arrays.sort(nums);
```

输出结果：-10 0 2 4 5 6
这个排序还能对数组元素进行指定范围排序：

Arrays.sort(Object[] array, int from, int to)

功能：对数组元素指定范围进行排序（排序范围是从元素下标为from,到下标为to-1的元素进行排序）

```
int[] nums = {2,5,0,4,1,-10};
//对前四位元素进行排序
Arrays.sort(nums, 0, 4);
```

输出结果：0 2 4 5 1 -10

②.Arrays.fill(Object[] array,Object object)

功能：可以为数组元素填充相同的值

```
int[] nums = {2,5,0,4,1,-10};
Arrays.fill(nums, 1);
for(int i :nums)
    System.out.print(i+" ");
```

执行结果：

1 1 1 1 1 1
这里也可以对数组的部分元素填充一个值,从起始位置到结束位置，取头不取尾

Arrays.fill(Object[] array,int from,int to,Object object)

```
int[] nums = {2,5,0,4,1,-10};
//对数组元素下标2到4的元素赋值为3
Arrays.fill(nums,2,5,3);
```

执行结果：2 5 3 3 3 -10
③ Arrays.toString(Object[] array)

功能：返回数组的字符串形式

```
int[] nums = {2,5,0,4,1,-10};
System.out.println(Arrays.toString(nums));
```

输出结果：[2, 5, 0, 4, 1, -10]
④Arrays.deepToString(Object arrays)

功能：返回多维数组的字符串形式

```
int[][] nums = {{1,2},{3,4}};
System.out.println(Arrays.deepToString(nums));
```

输出结果：[[1, 2], [3, 4]]

拓展

你有没有考虑过Arrays自带的查找和排序，以及ArrayList的几个关键方法在jdk里是如何实现的呢？感兴趣的话可以研究一下，面试的时候可以作为技术问题与面试官聊的。

4.基础过关题

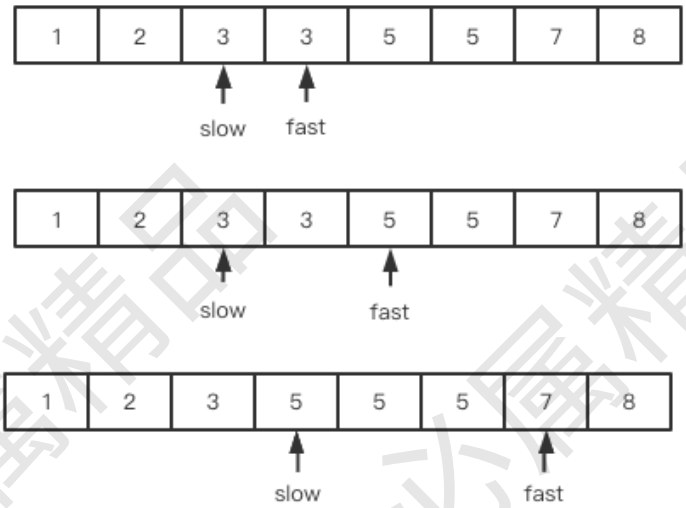
本章代码路径：unit1_array.ch1.ch3

4.1 数组问题常用思想

4.1.1.双指针思想

我们前面说过数组里的元素是紧紧靠在一起的，不能有空隙，假如有空隙，后面的元素就要整体向前移动，同样如果在中间位置插入元素，那么其后的元素都要整体向后移动。很多算法问题需要多次反复移动的，比如说连续删除多个元素，这就导致会频繁大量移动元素，进而效率低下、执行超时。所以如何尽量减少大量元素移动的次数就是数组相关算法要解决的第一个问题。这里介绍一种非常简单，但是非常有效的方式——双指针。

所谓的双指针其实就是两个变量，不一定真的是指针，这里只是一个统称。看一个例子，从下面序列中删除重复元素[1,2,3,3,5,5,7,8]，删除之后的结果应该为[1,2,3,5,7,8]。如果按照普通遍历的方式，删除一个移动一次，那你要在删除3和5时都要将其后的移动一次，而使用双指针可以方便的解决这个问题。



首先我们定义两个指针slow、fast。slow表示当前以及之前的位置都是单一的，fast则一直向后找与slow位置不一样的，找到之后将其复制到slow的后面，然后slow再向前，而fast则会继续向后找。找完之后slow以及之前的元素都是单一的了。这样就可以用最少的移动次数来解决问题。这种就是双指针思想。

上面这种也称为快慢指针，实际在用的时候，还要有双向指针，就是从两端向中间走，还可能是从后向前的快慢指针等等。

双指针的思想在处理数组、字符串等场景下有大量的应用，简单好用。

4.1.2 排序思想

在工程中很多问题排序一下就能轻松搞定，但是在算法中只有比较难的问题才允许你排序，不然的话就没思维含量了。我们再层次遍历和广度优先等稍微复杂的算法问题中也会使用排序来降低实现难度，而且此时可以使用Arrays.sort()方法直接来做。

具体例子，我们遇到了再说吧。

4.1.3 集合和Hash

在处理重复等情况时，Hash是一个非常好用的方法，同样，只有比较难的问题才允许你使用，另外就是要注意将谁作为Key放入Hash中，一般我们习惯将index作为key，但是有些问题是可以将key作为hash值，而将index作为结果的，这种做法会比较难理解，实现却非常容易。

4.1.4 带入尝试法

这是最low但是最有效的一种方式，数组最大的痛苦就是边界和判断条件很难判断，例如大于的时候要不要等于，小于的时候要不要等于，变量直接返回还是要返回变量-1。这些问题没有明确的规律，最有效的方式就是写几个开始和结束位置的元素来试一下。这个方法在解决二维数组问题时更加有效，因为二维数组里要处理的边界和条件有多个，只要一个有问题就废了。

我们接下来就具体看一些题目。

4.2. 单调数组问题

代码位置 Monotonic

我们在写算法的时候，数组是否有序是一个非常重要的前提，有或者没有可能会采用完全不同的策略。而且有些地方会说序列是单调序列，这意味着序列中可能会有重复元素，者都会直接影响我们怎么写算法。

先来思考一下什么是数组是否有序，或者是单调的。如果对于所有 i <= j，A[i] <= A[j]，那么数组 A 是单调递增的。 如果对于所有 i <= j，A[i]> = A[j]，那么数组 A 是单调递减的。所以遍历数组执行这个判定条件就行了，由于有递增和递减两种情况。于是我们执行两次循环就可以了，代码如下：

```
public static boolean isMonotonic(int[] nums) {
        return isSorted(nums, true) || isSorted(nums, false);
    }

    public static boolean isSorted(int[] nums, boolean increasing) {
        int n = nums.length;
        if (increasing) {
            for (int i = 0; i < n - 1; ++i) {
                if (nums[i] > nums[i + 1]) {
                    return false;
                }
            }
        } else {
            for (int i = 0; i < n - 1; ++i) {
                if (nums[i] < nums[i + 1]) {
                    return false;
                }
            }
        }
        return true;
    }
```

这样虽然实现功能了，貌似有点繁琐，还要遍历两次，面试官可能会感觉不太够，能否一次遍历实现呢？当然可以，假如我们在i和i+1位置出现了a[i]>a[i+1],而在另外一个地方j和j+1出现了a[i]<a[i+1]，那是不是说明就不是单调了呢？这样我们就可以使用两个变量标记一下就行了，代码如下：

```
class Solution {
    public boolean isMonotonic(int[] nums) {
        boolean inc = true, dec = true;
```

```
        int n = nums.length;
        for (int i = 0; i < n - 1; ++i) {
            if (nums[i] > nums[i + 1]) {
                inc = false;
            }
            if (nums[i] < nums[i + 1]) {
                dec = false;
            }
        }
        return inc || dec;
    }
}
```

我们判断整体单调性不是白干的，很多时候需要将特定元素插入到有序序列中，并保证插入后的序列仍然有序，例如这个问题：

```
给定一个排序数组和一个目标值，在数组中找到目标值，并返回其索引。如果目标值不存在于数组中，返回它将会被按顺序插入的位置。
示例1:
输入: nums = [1,3,5,6], target = 5
存在5，并且在索引为2的位置，所以输出: 2
示例2:
输入: nums = [1,3,5,6], target = 2
不存在2，2插入之后在索引为1的位置，所以输出: 1
```

这个问题没有让你将新元素插入到原始序列中，还是比较简单的，只要遍历一下就找到了。如果面试官再问你，该如何更快的找到目标元素呢？那他其实是想考你二分查找。凡是提到在单调序列中查找的情况，我们应该马上就要想到是否能用二分来提高查找效率。二分的问题我们后面专门讨论，这里只看一下实现代码：

```
class Solution {
    public int searchInsert(int[] nums, int target) {
        int n = nums.length;
        int left = 0, right = n - 1, ans = n;
        while (left <= right) {
            int mid = ((right - left) >> 1) + left;
            if (target <= nums[mid]) {
                ans = mid;
                right = mid - 1;
            } else {
                left = mid + 1;
            }
        }
        return ans;
    }
}
```

这个题想告诉大家的就是数组问题可以很简单，但是也可能非常难，难就难在如果涉及高级问题你要清楚是在考什么，如果看不出来就只能黯然离场了。

4.3.数组合并专题

数组合并就是将两个或者多个有序数组合并成一个新的。这个问题的本身不算难，但是要写的够出彩才可以。还有就是在排序中，归并排序本身就是多个小数组的合并，所以研究该问题也是为了后面打下基础。

问题1 合并两个有序数组

先来看如何合并两个有序数组，完整要求：

```
给你两个按非递减顺序 排列的整数数组 nums1 和 nums2，另有两个整数 m 和 n ，分别表示 nums1 和 nums2 中的元素数目。
请你合并 nums2 到 nums1 中，使合并后的数组同样按 非递减顺序 排列。
注意：最终，合并后数组不应由函数返回，而是存储在数组 nums1 中。为了应对这种情况，nums1 的初始长度为 m + n，其中前 m 个元素表示应合并的元素，后 n 个元素为 0 ，应忽略。nums2 的长度为 n 。

例子1：
输入：nums1 = [1,2,3,0,0,0], m = 3, nums2 = [2,5,6], n = 3
输出：[1,2,2,3,5,6]
解释：需要合并 [1,2,3] 和 [2,5,6] 。
合并结果是 [1,2,2,3,5,6] ，其中斜体加粗标注的为 nums1 中的元素。
```

对于有序数组的合并，一种简单的方法，先不考虑顺序问题，将B合并到A上，然后再对A进行排序。而排序我们可以直接使用Arrays提供的排序方法，也就是这样:

```
class Solution {
    public void merge(int[] nums1, int m, int[] nums2, int n) {
        for (int i = 0; i < n; ++i) {
            nums1[m + i] = nums2[i];
        }
        Arrays.sort(nums1);
    }
}
```

很明显，这么写只是为了开拓思路，而面试官会不喜欢，因为就是要你自己实现，即使排序也要自己实现。而且这样的代码在实现的时候可能会存在一个漏洞，你知道是什么吗?

这个问题的关键是将B合并到A的仍然要保证有序。因为A是数组不能强行插入，如果从前向后插入，数组A后面的元素会多次移动，代价比较高。为此我们可以借助一个新数组C来做，先将选择好的放入到C中，最后再返回。这样虽然解决问题了，但是面试官可能会问你能否再优化一下，或者不申请新数组就能做呢？更专业一点的提问是：上面算法的空间复杂度为O(n)，能否有O(1)的方法。

比较好的方式是从后向前插入，A和B的元素数量是固定的，所以排序后最远位置一定是A和B元素都最大的那个，依次类推，这一点比较绕，但是画画图能很容易想明白的。代码如下：

```
class Solution {
  public void merge(int[] nums1, int nums1_len, int[] nums2, int nums2_len) {
        int i = nums1_len + nums2_len - 1;
        int len1 = nums1_len - 1, len2 = nums2_len - 1;
```

```
        while (len1 >= 0 && len2 >= 0) {
            if (nums1[len1] <= nums2[len2])
                nums1[i--] = nums2[len2--];
            else if (nums1[len1] > nums2[len2])
                nums1[i--] = nums1[len1--];
        }
        //假如A或者B数组还有剩余
        while (len2 != -1) nums1[i--] = nums2[len2--];
        while (len1 != -1) nums1[i--] = nums1[len1--];
    }
}
```

对于数组，最大的问题是很多处理非常灵活，稍微变一下就需要调整其他相关的条件，其实上面再分析，我们发现如果B已经遍历完了，其实剩余的A就不必再处理，因此可以继续精简一下，例如下面的实现也可以：

```
public class Solution {
    public void merge(int A[], int m, int B[], int n) {
        int indexA=m-1;
        int indexB=n-1;
        int index=m+n-1;
        while (indexA>=0&&indexB>=0){
            A[index--]=A[indexA]>=B[indexB]?A[indexA--]:B[indexB--];
        }
        //假如A或者B数组还有剩余
        while (indexB>=0){
            A[index--]=B[indexB--];
        }
  }
}
```

问题2 合并n个有序数组

如果顺利将上面的代码写出来了就基本过关了，但是面试官可能会忍不住给加餐：如果是n个数组合并成一个该真么办呢？不要怕，能解决说明你能力强，加薪过程到这里才开始。

这个问题题有三种基本的思路，一种是先将数组全部拼接到一个数组中，然后再排序。第二种方式是不断进行两两要有序合并，第三种方式是使用堆排序。

先合并再排序最简单了，新建一个N*L的数组，将原始数组拼接存放在这个大数组中，再调用Arrays.sort()进行排序，这种方式比较容易实现，如下。

```
import java.util.Arrays;
class Solution {
   public static int[] MergeArrays(int[][] array) {
        int N = array.length, L;
        if (N == 0) {
            return new int[0];
        }
```

```
        //数组内容校验
        L = array[0].length;
        for (int i = 1; i < N; i++)
            if (L != array[i].length)
                return new int[0];
        //开辟空间
        int[] result = new int[N * L];
        //将各个数组依次合并到一起
        for (int i = 0; i < N; i++)
            for (int j = 0; j < L; j++)
                result[i * L + j] = array[i][j];
        //排序一下完事
        Arrays.sort(result);
        return result;
    }
}
```

第二种方式不断两两合并就是归并的思想，实现稍微麻烦一些，我们到学习归并排序再看这个问题。第三种是使用优先级队列排序实现，这个我们在学习完堆之后再详细解释。

4.4.字符串替换空格问题

这个题是出现频率很高的题目用的就是双指针思想，要求是：

> 请实现一个函数，将一个字符串中的每个空格替换成“%20”。例如，当字符串为We Are Happy.则经过替换之后的字符串为We%20Are%20Happy。

基本思路：这个题处理到时候首先要考虑用什么来存储这个字符串，如果是长度不可变的char数组，那么必须新申请一个更大的空间。如果使用长度可变的空间来管理原始数组，或者原始数组申请得足够大，这时候就可能要求你不能申请O(n)大小的空间来解决问题。这两种方式的处理方法和实现逻辑会不一样，我们一个个看。

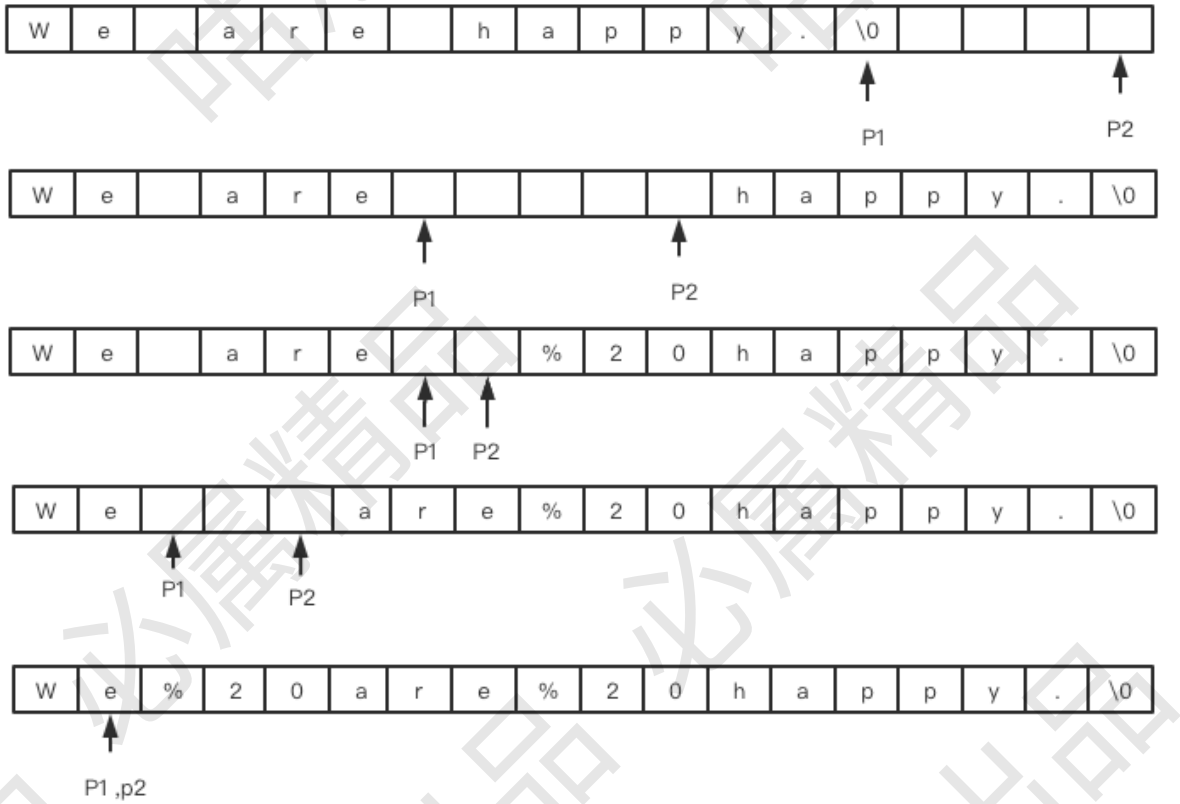
首先是如果长度不可变，我们必须新申请一个更大的空间，然后将原始数组中出现空格的位置直接替换成%20即可，代码如下：

```
    public String replaceSpace(StringBuffer str) {
    String res="";
      for(int i=0;i<str.length();i++){
          char c=str.charAt(i);
          if(c==' ')
              res += "%20";
          else
              res += c;
      }
      return res;
   }
```

对于第二种情况，我们首先想到的是从头到尾遍历整个字符串，遇到空格的时候就将其后面的元素向后移动2个位置，但是这样的问题在前面说过会导致后面的元素大量移动，时间复杂度为O(n^2)，执行的时候非常容易超时。

比较好的方式是可以先遍历一次字符串，这样可以统计出字符串中空格的总数，由此计算出替换之后字符串的长度，每替换一个空格，长度增加2，即替换之后的字符串长度为原来的长度+2*空格数目。接下来从字符串的尾部开始复制和替换，用两个指针P1和P2分别指向原始字符串和新字符串的末尾，然后向前移动P1，若指向的不是空格，则将其复制到P2位置，P2向前一步；若P1指向的是空格，则P1向前一步，P2之前插入%20，P2向前三步。这样，便可以完成替换，时间复杂度为O(n)。

详细过程如下：




```
public String replaceSpace(StringBuffer str) {
  if(str==null)
      return null;
  int numOfblank = 0;//空格数量
  int len=str.length();
  for(int i=0;i<len;i++){  //计算空格数量
      if(str.charAt(i)==' ')
          numOfblank++;
  }
  str.setLength(len+2*numOfblank); //设置长度
  int oldIndex=len-1;  //两个指针
  int newIndex=(len+2*numOfblank)-1;

  while(oldIndex>=0 && newIndex>oldIndex){
      char c=str.charAt(oldIndex);
      if(c==' '){
          oldIndex--;
          str.setCharAt(newIndex--,'0');
          str.setCharAt(newIndex--,'2');
          str.setCharAt(newIndex--,'%');
```

```
            }else{
                str.setCharAt(newIndex,c);
                oldIndex--;
                newIndex--;
            }
        }
        return str.toString();
    }
```

测试方法如下：

```
public static void main(String[] args) {
    StringBuffer  sb =new StringBuffer("We are happy.") ;
    System.out.println(replaceSpace(sb));
}
```

4.5.删除数组元素专题

所谓算法，其实就是将一个问题改改条件多折腾，上面专题就是添加的变形，再来看几个删除的变形问题。

问题1：原地移除所有数值等于 val 的元素。

```
给你一个数组 nums 和一个值 val，你需要原地移除所有数值等于 val 的元素，并返回移除后数组的新长度。
要求：不要使用额外的数组空间，你必须仅使用 O(1) 额外空间并原地修改输入数组。元素的顺序可以改变。你不需要考虑数组中超出新长度后面的元素。

例如：
输入：nums = [3,2,2,3], val = 3
输出：2, nums = [2,2]
例子2：
输入：nums = [0,1,2,2,3,0,4,2], val = 2
输出：5, nums = [0,1,4,0,3]
```

在删除的时候，从删除位置开始的所有元素都要向前移动下，所以这题的关键就是如果很多值为val的元素，如何避免反复向前移动。 有两种处理方式，为了开拓思路，我们都看一下：

（1）第一种方法：快慢双指针移动

将数组分成前后两段，定义两个指针i和j，初始值都是0。

i之前的位置都是有效部分，j表示当前要访问的元素。

这样遍历的时候，j不断向后移动：

如果array[j]的值不为val，则将其移动到array[++i]处。

如果array[j]的值为val，则j继续向前移动

这样，前半部分是有效部分，后半部分是无效部分。

其实只可以借助for的语法进一步简化实现：

```
    public int removeElement(int[] nums, int val) {
        int ans = 0;
        for(int num: nums) {
            if(num != val) {
                nums[ans] = num;
                ans++;
            }
        }
        //最后剩余元素的数量
        return ans;
    }
```

这里虽然只有一个变量ans，但是for循环的写法帮助我们减少了一个变量的定义。

（2）第二种方式：双指针交换移除

这里还有一种解法，有的地方叫做交换移除，思路就是我们还是从两端开始向中间遍历，left遇到num[i]=val的时候停下来，右侧继续。当右侧遇到num[j]!=val的位置的时候，将num[j]交换或者直接覆盖num[i]。之后i继续向右走。

```
    public int removeElement(int[] nums, int val) {
        int ans = nums.length;
        for (int i = 0; i < ans;) {
            if (nums[i] == val) {
                nums[i] = nums[ans - 1];
                ans--;
            } else {
                i++;
            }
        }
        return ans;
    }
```

这个思路其实和快速排序中执行一轮的过程非常类似。快速排序就是标定一个元素，比它小的全在其左侧，比它大的全在右侧。

这么做的坏处是元素的顺序和原来的可能不一样了。

问题2：删除有序数组中的重复项

我们再看一个问题：

给你一个有序数组 nums ，请你原地删除重复出现的元素，使每个元素只出现一次 ，返回删除后数组的新长度。
不要使用额外的数组空间，你必须在原地修改输入数组 并在使用 O(1) 额外空间的条件下完成。

例如
输入：nums = [1,1,2]
输出：2, nums = [1,2]
解释：函数应该返回新的长度 2 ，并且原数组 nums 的前两个元素被修改为 1, 2 。不需要考虑数组中超出新长度后面的元素。

例子2：
输入：nums = [0,0,1,1,1,2,2,3,3,4]
输出：5, nums = [0,1,2,3,4]
解释：函数应该返回新的长度 5 ， 并且原数组 nums 的前五个元素被修改为 0, 1, 2, 3, 4 。不需要考虑数组中超出新长度后面的元素。

使用双指针最方便，一个指针负责责数组遍历，一个指向有效数组的最后一个位置。当两个指针的值不一样时，才将指向有效位的向下移动。所以代码就可以这样写：

```
public static int removeDuplicates(int[] nums) {
    int n = nums.length;
    //j用来标记有效位
    int j = 1;
    for (int i = 0; i < n; i++) {
        if (nums[i] != nums[j - 1]) {
            nums[j] = nums[i];
            j++;
        }
    }
    return j;
}
```

测试代码如下，注意这里更换测试序列的方法，在线性表之一介绍过，这是面试时特别实用的方法，一定要记住哟。

```
public static void main(String[] args) {
    int[] arr = new int[]{0, 0, 1, 1, 1, 2, 2, 3, 3, 4};
    int last = removeDuplicates(arr);
    for (int i = 0; i < last; i++) {
        System.out.print(arr[i] + "  ");
    }
}
```

输出结果为：0 1 2 3 4

上面这题既然重复元素可以保留一个，那我是否可以最多保留2个，3个或者K个呢？甚至一个都不要呢？有一定的难度，我们在大厂冲刺部分再讨论这个问题。

4.6.元素奇偶移动专题

移动元素也是数组部分非常重要的问题，典型的是数组的奇偶调整、排序等。

问题1：奇偶数分离

> 输入一个整数数组，通过一个函数来调整该数组中数字的顺序，使得所有的奇数位于数组的前半部分，所有的偶数位于数组的后半部分。
> 例如：
> 输入：`[3,1,2,4]`
> 输出：`[2,4,3,1]`
> 输出 `[4,2,3,1]`，`[2,4,1,3]` 和 `[4,2,1,3]` 也会被接受。

这个题最直接的方式是使用一个临时数组，第一遍查找并将所有的偶数复制给新数组，第二遍查找并复制所有的奇数给数组。这种方式实现比较简单，会面临面试官的灵魂之问："是否有空间复杂度为O(1)的"方法。我们采用双指针的方法，维护两个指针 i 和 j，循环保证每刻小于 i 的变量都是偶数（也就是 A[k] % 2 == 0 当 k < i），所有大于 j 的都是奇数。

所以， 4 种情况针对 (A[i] % 2, A[j] % 2)：

如果是 (0, 1)，那么万事大吉 i++ 并且 j--。
如果是 (1, 0)，那么交换两个元素，然后继续。
如果是 (0, 0)，那么说明 i 位置是正确的，只能 i++。
如果是 (1, 1)，那么说明 j 位置是正确的，只能 j--。

通过这 4 种情况，循环不变量得以维护，并且 `j-i` 不断变小。最终就可以得到奇偶有序的数组。

```
class Solution {
    public int[] sortArrayByParity(int[] A) {
        int i = 0, j = A.length - 1;
        while (i < j) {
            if (A[i] % 2 > A[j] % 2) {
                int tmp = A[i];
                A[i] = A[j];
                A[j] = tmp;
            }

            if (A[i] % 2 == 0) i++;
            if (A[j] % 2 == 1) j--;
        }

        return A;
    }
}
```

拓展：上面的代码适合面试，比较容易想出来。如果比较熟了可以采用下面这种更紧凑的方式：

```
public int[] reOrderArray (int[] array)
        int left=0, right=array.length-1;
         while(left<right){
             while(array[left]%2!=0 &&left<right) left++;
             while(array[right]%2==0 &&left<right ) right--;
             int temp=array[left];
             array[left]=array[right];
             array[right]=temp;
         }
        return array;
```

问题2.调整后的顺序仍与原始数组的顺序一致

对于第二题，这里对顺序也有要求，上面的方式就不行了，为此考虑冒泡的方式交换奇偶，就是比较的时候如果左边是偶数右边是奇数，就进行交换，否则就继续扫描。而冒泡排序比较的是大小，因此两者的原理是一致的。

```
public int[] reOrderArray (int[] array) {
       if (array == null || array.length == 0)
          return new int[0];
       int n = array.length;
       for (int i=0; i<n; i++) {
            for (int j=0; j<n-1-i; j++) {
                // 左边是偶数，右边是奇数的情况
                if ((array[j] & 1) == 0 && (array[j+1] & 1) == 1) {
                    int tmp = array[j];
                    array[j] = array[j+1];
                    array[j+1] = tmp;
                }
            }
        }
        return array;
    }
```

这里需要注意的是，j的循环终止条件为什么是j<n-1-i，而不是其他的呢？比如n-i呢？这里其实就可以用我们说的将两端带入的方式，第一次循环i=0，我们要让j走到底，所以应该是n-1的位置，所以此时就是n-i-1了。

4.7 数组轮转问题

先看题目要求：

```
给你一个数组，将数组中的元素向右轮转 k 个位置，其中 k 是非负数。
例如：
输入：nums = [1,2,3,4,5,6,7], k = 3
输出：[5,6,7,1,2,3,4]
解释：
向右轮转 1 步：[7,1,2,3,4,5,6]
向右轮转 2 步：[6,7,1,2,3,4,5]
向右轮转 3 步：[5,6,7,1,2,3,4]
```

这个题怎么做呢?你是否想到可以逐个移动来实现？理论上可以，但是实现的时候会发现困难重重，很多条件不好处理。这里直接介绍一种很简单，但是很奇妙的方法：两轮翻转。

思路如下：

1. 首先对整个数组实行翻转，这样子原数组中需要翻转的子数组，就会跑到数组最前面。
2. 这时候，从 k 处分隔数组，左右两数组，各自进行翻转即可。

例如 [1,2,3,4,5,6,7] 我们先将其整体翻转成[7,6,5,4,3,2,1],

然后再根据k将其分成两组 [7,6,5] 和[4,3,2,1],

最后将两个再次翻转：[5,6,7] 和[1,2,3,4],这时候就得到了最终结果[5,6,7,1,2,3,4]

代码如下：

```
class Solution {
    public void rotate(int[] nums, int k) {
        k %= nums.length;
        reverse(nums, 0, nums.length - 1);
        reverse(nums, 0, k - 1);
        reverse(nums, k, nums.length - 1);
    }
    public void reverse(int[] nums, int start, int end) {
        while (start < end) {
            int temp = nums[start];
            nums[start] = nums[end];
            nums[end] = temp;
            start += 1;
            end -= 1;
        }
    }
}
```

5.大厂冲刺题

如果前面的问题搞清楚了，恭喜你，一维数组已经迈开一大步了。如果你志存高远，想超越自己，练习更多有挑战的题目，那我们继续看吧。

5.1 数组加法专题

数字加法，小学生都会的问题，但是如果让你用数组来表示一个数，如何实现加法呢？你可能会感觉这个简单，仍然从数组末尾向前挨着计算就行了，但是实现的时候会发现有很多问题，例如算到A[0]位置时发现还要进位该怎么办呢？

再拓展，假如一个是数组，一个是普通的整数，要一边遍历数组一边处理整数，你能轻松搞定吗？

再拓展，如果两个整数是用字符串表示的呢？如果要按照二进制加法的规则来呢？

再拓展，如果整数是用链表表示的呢？该问题我们到下一章链表再谈。这里先看看前面这几种情况。

问题1：数组实现整数加法

先看一个用数组实现逐个加一的问题。具体要求是由整数组成的非空数组所表示的非负整数，在其基础上加一。这里最高位数字存放在数组的首位， 数组中每个元素只存储**单个**数字。并且假设除了整数 0 之外，这个整数不会以零开头。例如：

```
输入：digits = [1,2,3]
输出：[1,2,4]
解释：输入数组表示数字 123。
```

这个看似很简单是不？从后向前依次加就行了，如果有进位就标记一下，但是如果到头了要进位怎么办呢？

例如在示例2中，从后向前加的时候，到了A[0]的位置计算为0，需要再次进位但是数组却不能保存了，这该怎么办呢？如果你在网上找相关的实现，会发现很多实现特别麻烦，看的欲望都没有，这里我们可以利用java的特定大大简化操作。

这里的关键是A[0]处什么时候出现再次进位的情况，我们知道此时一定是9，99，999，这样的结构才会出现加1之后再次进位，而进位之后的结果一定是10，100，1000这样的结构，由于java中数组默认初始化为0，所以我们此时只要申请一个空间比A[]大一个的数组B[]，然后将B[0]设置为1就行了。

这样代码就会变得非常简洁：

```
public static int[] plusOne(int[] digits) {
    int len = digits.length;
    for (int i = len - 1; i >= 0; i--) {
        digits[i]++;
        digits[i] %= 10;
        if (digits[i] != 0)
            return digits;
    }
    // 比较巧妙的设计
    digits = new int[len + 1];
    digits[0] = 1;
    return digits;
}
```

这里使用数组默认初始化为0的特性来大大简化了我们的处理复杂度。如果使用的是C等默认值不是0的语言，我们只要在申请的时候先将所有的元素初始化为0就行了。

再看一个实现两个数相加的问题，具体要求：

对于非负整数 X 而言，X 的数组形式是每位数字按从左到右的顺序形成的数组。例如，如果 X = 1231，那么其数组形式为 [1,2,3,1]。给定非负整数 X 的数组形式 A，返回整数 X+K 的数组形式。

示例1：
输入：A = [1,2,0,0], K = 34
输出：[1,2,3,4]
解释：1200 + 34 = 1234

示例2：
输入：A = [9,9,9,9,9,9,9,9,9,9], K = 1
输出：[1,0,0,0,0,0,0,0,0,0,0]
解释：9999999999 + 1 = 10000000000

这个问题不需要太复杂，根据加法的原理逐个位置相加就行，但是这里需要先处理K，采用取余和除法来逐步获得所有低位元素，再将其与数组数字加在一起就行了。例如计算 123+912，我们从低位到高位依次计算 3+23+2、2+12+1 和 1+91+9。任何时候，若加法的结果大于等于 1010，把进位的 11 加入到下一位的计算中，所以最终结果为 10351035。

```
class Solution {
    public List<Integer> addToArrayForm(int[] num, int k) {
        List<Integer> res = new ArrayList<Integer>();
        int n = num.length;
        for (int i = n - 1; i >= 0; --i) {
            int sum = num[i] + k % 10;
            k /= 10;
            if (sum >= 10) {
                k++;
                sum -= 10;
            }
            res.add(sum);
        }
        for (; k > 0; k /= 10) {
            res.add(k % 10);
        }
        Collections.reverse(res);
        return res;
    }
}
```

问题2：字符串加法与二进制加法

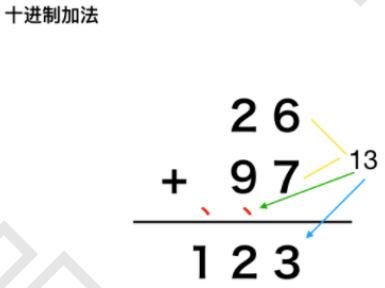
上面这个问题如果都使用两个数组A[]和B[]来存就毫无难度了，加数使用K就是为了提高实现的难度，假如这里将整数换成字符串或者二进制会怎么样呢？我们继续看：

字符串加法就是使用字符串来表示数字，然后计算他们的和，具体要求如下：

给定两个字符串形式的非负整数 num1 和num2 ，计算它们的和并同样以字符串形式返回。
你不能使用任何内建的用于处理大整数的库（比如 BigInteger）， 也不能直接将输入的字符串转换为整数形式。

例如：
输入：num1 = "456", num2 = "77"
输出："533"

我们先想一下小学里如何计算两个比较大的数相加的，经典的竖式加法是这样的：



从低到高逐位相加，如果当前位和超过 1010，则向高位进一位？因此我们只要将这个过程用代码写出来即可。具体实现也不复杂，我们定义两个指针 ii 和 jj 分别指向num1和num2的末尾，即最低位，同时定义一个变量 add 维护当前是否有进位，然后从末尾到开头逐位相加即可。你可能会想两个数字位数不同怎么处理，这里我们统一在指针当前下标处于负数的时候返回 00，等价于对位数较短的数字进行了补零操作，这样就可以除去两个数字位数不同情况的处理，具体可以看下面的代码：

```
class Solution {
    public String addStrings(String num1, String num2) {
        int i = num1.length() - 1, j = num2.length() - 1, add = 0;
        StringBuffer ans = new StringBuffer();
        while (i >= 0 || j >= 0 || add != 0) {
            int x = i >= 0 ? num1.charAt(i) - '0' : 0;
            int y = j >= 0 ? num2.charAt(j) - '0' : 0;
            int result = x + y + add;
            ans.append(result % 10);
            add = result / 10;
            i--;
            j--;
        }
        // 计算完以后的答案需要翻转过来
        ans.reverse();
        return ans.toString();
    }
}
```

这个问题虽然需要在两头都使用字符串函数转换一下，但是难度不大，我们再看一下，如果这里是二进制该怎么处理呢？看一下题目要求：

给你两个二进制字符串，这个字符串是用数组保存的，返回它们的和（用二进制表示）。
输入为 非空 字符串且只包含数字 1 和 0。

示例1:
输入: a = "11", b = "1"
输出: "100"

示例2:
输入: a = "1010", b = "1011"
输出: "10101"

这个题也是用字符串来表示数据的，也要先转换为字符数组。我们熟悉的十进制，是从各位开始，逐步向高位加，达到10就进位，而对于二进制则判断相加之后是否为二进制的10，是则进位。本题解中大致思路与上述一致，但由于字符串操作原因，不确定最后的结果是否会多出一位进位，所以会有 2 种处理方式：

- 第一种，在进行计算时直接拼接字符串，会得到一个反向字符，需要最后再进行翻转
- 第二种，按照位置给结果字符赋值，最后如果有进位，则在前方进行字符串拼接添加进位

这两种方法都可以用，关键看具体题目适合哪一种，我们这里采用第二种实现。

```
class Solution {
    public String addBinary(String a, String b) {
        StringBuilder ans = new StringBuilder();
        int ca = 0;
        for(int i = a.length() - 1, j = b.length() - 1;i >= 0 || j >= 0; i--, j--) {
            int sum = ca;
            sum += i >= 0 ? a.charAt(i) - '0' : 0;
            sum += j >= 0 ? b.charAt(j) - '0' : 0;
            ans.append(sum % 2);
            ca = sum / 2;
        }
        ans.append(ca == 1 ? ca : "");
        return ans.reverse().toString();
    }
}
```

这里还有人会想，先将其转换成十进制，加完之后再转换成二进制可以吗？这么做实现非常容易，而且可以使用jdk的方法直接转换，但是还是那句话，工程里可以这么干，稳定可靠，但是算法里不行，太简单了。

5.2进制转换问题

进制转换是一个说起来简单，实现起来非常头疼的问题，主要是转换时建立映射需要很大的代码量，例如十进制转16进制时，ABCDEF 的处理，至少要写6个if else，即使使用Switch也不简单。另外还有正负等问题，那该办呢?

我们先看一个简单的问题，将一个整数转换成七进制数，并以字符串的形式输出，例如

```
示例1;
输入：num = 100
输出："202"

示例2:
输入：num = -7
输出："-10"
```

这个问题其实可以用jdk的方法一行搞定：

```
public String convertToBase7(int num) {
        return Integer.toString(num, 7);
    }
```

很明显，这适合工作中用，在这里我们还是要手写，这个题目也不需要太多技巧，按照计算规则写就可以了：

```
class Solution {
    public String convertToBase7(int num) {
        boolean flag = true;
        if(num<0){
           flag = false;
           num = (-1)*num;
        }
        if(num==0){
           return 0+"";
        }
        int a = 0;
        a=num%7;
        num=num/7;
        int i=1;
        while(num>0){
            a+=(num%7)*(int)Math.pow(10,i);
            i++;
            num=num/7;
        }
        if(flag){
          return a+"";
        }else{
            return "-"+a;
        }
    }
}
```

上面这个问题还比较简单，再看一个考察更频繁，难度也更大，技巧也更多的问题，先看题意：

```
给定一个十进制数M，以及需要转换的进制数N。将十进制数M转化为N进制数。
M是32位整数，2<=N<=16.
```

这个题目的思路不难，但是想写正确却很不容易，关键问题在于超过10进制之后如何准确映射，这会有大量的if 判断，如果一个个定义，会出现了写了一坨，结果越写越乱的情况。

另外还需要对结果进行一次转置，还需要判断负号，是非常考验编程功底的题。为此采取三个措施：
一个是定义大小为16的数组F，保存的是2到16的各个值对应的标记，这样赋值时只计算下表，必考虑每种进制的转换关系了。
第二个是 使用StringBuffer完成数组转置等功能。
第三是通过一个flag来判断正数还是负数。

```
public class Solution {
    /**
     * 进制转换
     * @param M int整型 给定整数
     * @param N int整型 转换到的进制
     * @return string字符串
     */
     // 因为要考虑到 余数 > 9 的情况，2<=N<=16.
    public static final String[] F = {"0", "1", "2", "3", "4", "5", "6", "7", "8", "9",
"A", "B", "C", "D", "E", "F"};

    public String solve (int M, int N) {
       Boolean flag=false;
        if(M<0){
            flag=true;
            M*=-1;
        }
        StringBuffer sb=new StringBuffer();
        int temp;
        while(M!=0){
            temp=M%N;
//精华之一：通过数组F[]解决了大量繁琐的不同进制之间映射的问题
            sb.append(F[temp]);
            M=M/N;
        }
//精华之二：使用StringBuffer的reverse()方法，让原本麻烦的转置瞬间美好
        sb.reverse();
        //精华之三：最后处理正负的问题，不要从一开始就揉在一起。
        return (flag? "-":"")+sb.toString();

    }
}
```

上面的写法是我找到的最简洁，最好实现的方式。

5.3 数组重排问题

先看题目要求：

找出数组中重复的数字。在一个长度为 n 的数组 nums 里的所有数字都在 0～n-1 的范围内。数组中某些数字是重复的，但不知道有几个数字重复了，也不知道每个数字重复了几次。请找出数组中任意一个重复的数字。
例如：
输入：
[2, 3, 1, 0, 2, 5, 3]
因为2和3有重复，所以可以输出：2 或 3

对于重复元素，Hash和集合是一个常用策略，我们首先想到可以用集合来记录数组的各个数字，当查找到重复数字则直接返回就行了，具体说来，
(1)初始化： 新建 HashSet ，记为 set;
(2)然后遍历数组 nums 中的每个数字 num：

- 当 num在 set 中，说明重复，直接返回 num；
- 将 num 添加至 set 中；

如果不存在则返回-1 。本题中说一定有重复数字，因此遇到了直接返回就行了。

所以代码就这样子：

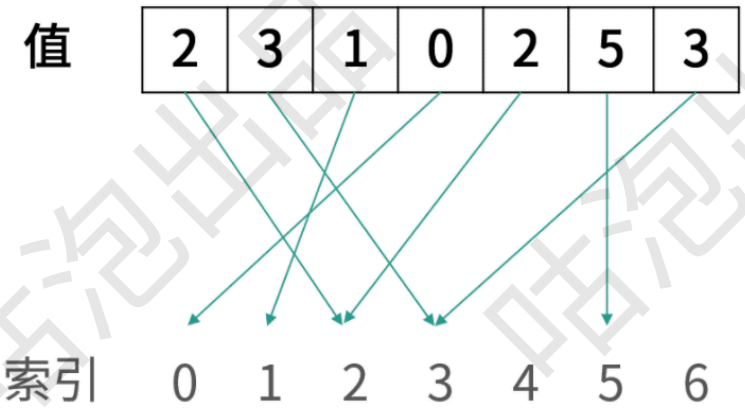
```
    public int findRepeatNumber(int[] nums) {
        Set<Integer> set = new HashSet<>();
        for(int num : nums) {
            if(set.contains(num))
                return num;
            set.add(num);
        }
        return -1;
    }
```

如果是在工程里遇到这个问题，就足够了，但是在算法里还不行，因为你需要开辟一个O(n)的空间。到这里你只能得个及格分。面试官可能会问你还有其他方法吗？然后貌似排序也可以是吧，从排序的数组中找出重复的数字只需要从头到尾扫描即可，所以先排序再查找的时间复杂度主要就取决于排序算法，一般为O(nlogn)。但是为了一个搜索就执行排序这样的重量级操作，还将原始数组给改了，还不好，继续找。

我们希望得到一种不消耗额外空间的算法，也就是本题的第三种解法：**数组重排**。由于题目中告诉我们所有的数字都在0到n-1的范围内，因此如果没有重复，那么所存储的值也正好是0到n-1这n个数字，我们把原数组重新排列为一个元素和对应下标值相同的数组。具体思路如下：

从头到尾扫描整个数组中的数字，当扫描到下标为i的数字时，首先比较这个数字（用arr[i]表示）是不是等于下标i，如果是，接着比较下一个数字；如果不是，则将其与索引位置为arr[i]的数字比较，若与其相同，则说明它就是一个重复数字，如果不同，就将其与第m个数字进行交换，也就是把它放到自己应在的位置去。重复这个过程，直到该位置上的数与下标相同为止。这种思路就像收拾房间一样，物品如果不在自己的位置上就将其拿过去，这样假如你买了两包一样的零食，骗来一个妹子几张一样的照片都可以清楚地知道了。

该算法看起来是两层循环，但是每个数字最多进行两次交换就会找到属于自己的位置，因为总的时间复杂度还是O(n)，不需要额外内存也就是这样子：



{2,3,1,0,2,5,3}具体调整过程是：

- 0(索引值)和2(索引值位置的元素)不相等，并且**2**(索引值位置的元素)和1(以该索引值位置的元素**2**为索引值的位置的元素)不相等，则交换位置，数组变为：{1,3,2,0,2,5,3}；
- 0(索引值)和1(索引值位置的元素)仍然不相等，并且**1**(索引值位置的元素)和3(以该索引值位置的元素**1**为索引值的位置的元素)不相等，则交换位置，数组变为：{3,1,2,0,2,5,3}；
- 0(索引值)和3(索引值位置的元素)仍然不相等，并且**3**(索引值位置的元素)和0(以该索引值位置的元素**3**为索引值的位置的元素)不相等，则交换位置，数组变为：{0,1,2,3,2,5,3}；
- 0(索引值)和0(索引值位置的元素)相等，遍历下一个元素；
- 1(索引值)和1(索引值位置的元素)相等，遍历下一个元素；
- 2(索引值)和2(索引值位置的元素)相等，遍历下一个元素；
- 3(索引值)和3(索引值位置的元素)相等，遍历下一个元素；
- 4(索引值)和2(索引值位置的元素)不相等，但是**2**(索引值位置的元素)和2(以该索引值位置的元素**2**为索引值的位置的元素)相等，则找到了第一个重复的元素。

实现代码如下：

```
public static int duplicate(int numbers[]) {
        int length = numbers.length;
        if (numbers == null || length < 1) {
            return -1;
        }
        for (int i = 0; i < length; i++) {
          //每个元素最多被交换两次就可以找到自己的位置，依次复杂度是O（n）
            while (numbers[i] != i) {
                if (numbers[numbers[i]] == numbers[i]) {
                    return numbers[i];
                } else {
                    int temp = numbers[numbers[i]]; //交换
                  //将numbers[i]放到属于他的位置上
                    numbers[numbers[i]] = numbers[i];
                    numbers[i] = temp;
                }
            }
        }
```

```
        return -1;
    }
```

这里虽然效率更高了，但是仍然改变了原始数组的内容，有没有不改变的方法呢？还有一种思路简单，但是实现比较麻烦的方式，这种思路类似折半查找，这里说一下思路。

以长度为8的数组{2,3,5,4,3,2,6,7}为例，根据题目要求，这个长度为8的数组，所有元素都在1到7的范围内，中间的数字4将1—7分成两部分，分别为1—4和5—7，接下来统计1—4在数组中出现的次数，发现是5次，则说明这4个数字中一定有重复数字。接下来再把1—4分成1、2和3、4两部分，1和2一共出现了两次，3和4一共出现了3次，说明3和4中有一个重复，再分别统计即可得到是3重复了。这并不保证找出所有的重复数字，比如2就没有找到。

实际上，这种二分查找时间复杂度也达到了O(nlogn)，不如用哈希表空间换时间来的直观。

5.4出现次数专题

数组还有一种类型就是让你解决元素出现次数， 虽然看起来简单，但是直观想到的不一定是最优解，所以我们还是要认真看一下的。

问题1.数组中出现次数超过一半的数字

> 数组中有一个数字出现的次数超过数组长度的一半，请找出这个数字。
> 例如：输入如下所示的一个长度为9的数组{1,2,3,2,2,2,5,4,2}。由于数字2在数组中出现了5次，超过数组长度的一半，因此输出2。如果不存在则输出0。

分析

对于没有思路的问题，我们的策略都是先在脑子里快速过一遍常见的数据结构和常见的算法策略，看看谁能解决问题，所以很多问题就会自然而然的出现多种解法。

首先，想想排序行不行？这里说一定有了，那么先对数组进行排序，在一个有序数组中次数超过一半的必定是中位数，那么可以直接取出中位数，然后遍历数组，看中位数是否出现次数超过一半。OK，没问题，第一种方法就出来了，这种方法的的时间复杂父取决于排序的时间复杂度，最快为O(nlogn)。

其次，Hash行不行？遍历数组，HashMap的key是元素的值，value是已经出现的次数，这样可以从map中直接判断是否有超过一半的数字。OK，第二种方法出来了。这种算法的时间复杂度为O(n)，但是这个性能提升是用O(n)的空间复杂度换来的。代码就是：

```
    public int moreThanHalfNum(int [] array) {
        if(array==null)
            return 0;
        Map<Integer,Integer> res=new HashMap<>();
        int len = array.length;
        for(int i=0;i<array.length;i++){
            res.put(array[i],res.getOrDefault(array[i],0)+1);
            if(res.get(array[i])>len/2)
                return array[i];
        }
        return 0;
    }
```

上面的方式虽然能解决问题，但是不够好，面试的时候如果你实在想不出来更好的方式。但是将前面这两种方式实现得很漂亮，那就得了80分了。

本题的一种最优方式是：根据数组特点，数组中有一个数字出现的次数超过数组长度的一半，也就是说它出现的次数比其他所有数字出现的次数之和还要多。因此，我们可以在遍历数组的时候设置两个值：一个是数组中的数result，另一个是出现次数times。当遍历到下一个数字的时候，如果与result相同，则次数加1,不同则次数减一，当次数变为0的时候说明该数字不可能为多数元素，将result设置为下一个数字，次数设为1。这样，当遍历结束后，最后一次设置的result的值可能就是符合要求的值（如果有数字出现次数超过一半，则必为该元素，否则不存在），因此，判断该元素出现次数是否超过一半即可验证应该返回该元素还是返回0。这种思路是对数组进行了两次遍历，复杂度为O(n)。

在这里times最小为0，如果等于0了，遇到下一个元素就开始+1。看两个例子，例如 [1,2,1,3,1,4,1]和[2,1,1,3,1,4,1]两个序列。

首先看 [1,2,1,3,1,4,1]：

开始的时候result=1，times为1

然后result=2，times减一为0

然后result=1，times已经是0了，遇到新元素就加一为1

然后result=3，times减一为0

然后result=1，times已经是0了，遇到新元素就加一为1

然后result=4，times减一为0

然后result=1，times加一为1

所以最终重复次数超过一半的就是1了。

这里可能有人会有疑问，假如1不是刚开始的元素会怎样呢？例如假如是序列[2,1,1,3,1,4,1]，你按照上面的过程写一下，会发现扛到最后的还是result=1，此时times为1。

还有一种情况假如是偶数，而元素个数恰好一半会怎么样呢？例如[1,2,1,3,1,4]，很明显最后结果是0，只能说明没有存在重复次数超过一半的元素。代码如下：

```
public int moreThanHalfNum(int [] array) {
     if(array==null||array.length==0)
          return 0;
       int len = array.length;
       int result=array[0];
       int times=1;
       for(int i=1;i<len;i++){
           if(times==0){
               result=array[i];
               times=1;
               continue;
           }

           if(array[i]==result)
               times++;
           else
```

```
                times--;
        }
        //检查是否符合
        times=0;
        for(int i=0;i<len;i++){
            if(array[i]==result)
                times++;
            if(times>len/2)
                return result;
        }
        return 0;
    }
```

问题2：数组中只出现一次的数字

题目要求

给定一个非空整数数组，除了某个元素只出现一次以外，其余每个元素均出现两次。找出那个只出现了一次的元素。

```
示例1:
输入: [2,2,1] 输出: 1
示例2:
输入: [4,1,2,1,2] 输出: 4
```

分析

这个题貌似使用Set集合比较好，Set集合不存储重复值，add()方法返回值为boolean类型，这一特点可以利用。题目明确说其他元素都是出现两次，我们也可以利用这个操作，当要添加的数与集合中已存在的数重复时，不会再进行添加操作，返回false，这时再进行remove操作，将集合中已存在的那个与要添加的数相同的元素移除，这样将作为方法参数传递过来的整型数组遍历完后，到最后集合中就只剩下了那个只出现了一次的数字。

```
public static Integer findOneNum(int[] arr) {
    Set<Integer> set = new HashSet<Integer>();
    for(int i : arr) {
      if(!set.add(i))//添加不成功返回false，前加上！运算符变为true
        set.remove(i);//移除集合中与这个要添加的数重复的元素
    }
    //注意边界条件的处理
    if(set.size() == 0)
        return null;
    //如果Set集合长度为0，返回null表示没找到
    return set.toArray(new Integer[set.size()])[0];
  }
```

上面要注意，必须存在那个只出现了一次的数字，否则Set集合长度将为0，最后一句代码运行时会出错，改进一下的话，所以前面加了个空来拦截可能的错误。

这个题如果真的遇到了，面试官可能不让你用集合或者Hash，或者直接提示你要用位运算来做，该怎么办呢?

我们继续拓展！使用异或运算符^！

0与其他数字异或的结果是那个数字，相等的数字异或得0。要操作的数组中除了某个数字只出现了一次之外，其他数字都出现了两次，所以可以定义一个变量赋初始值为0，用这个变量与数组中每个数字做异或运算，并将这个变量值更新为那个运算结果，直到数组遍历完毕，最后得到的变量的值就是数组中只出现了一次的数字了。这种方法只需遍历一次数组，提高了程序运行的效率。

总结一下：这种解法要用到异或运算的几个规则：

```
0^0 = 0;
0^a = a;
a^a = 0;
a ^ b ^ a = b.
```

实现代码就是：

```
public static int findOneNum(int[] arr) {
    int flag = 0;
    for(int i : arr) {
      flag ^= i;
    }
    return flag;
  }
```

由此，也回到我们刚开始说的问题，数组的问题不会做，不是说明你数组没学好，而是要学习Hash、集合、位运算等等很多高级问题。

5.5 继续谈重复数字的问题

我们在前面删除数组元素专题部分提到了如何删除重复项的问题，假如重复的元素可以保留两个，或者K个该怎么办呢?

问题1 重复项保留K次

我们在上面删除数组元素的专题中分析了删除值为特定元素的情况，也分析了如果有序数组中存在重复项，我们只保留一个的情况。既然能保留一个，那是否可以将条件改为最多保留2个，3个甚至K个呢？看一下具体要求：

给你一个有序数组 nums ，请你原地删除重复出现的元素，使每个元素最多出现两次 ，返回删除后数组的新长度。不要使用额外的数组空间，你必须在 原地修改输入数组并在使用 O(1) 额外空间的条件下完成。

输入：nums = [1,1,1,2,2,3]
输出：5, nums = [1,1,2,2,3]
解释：函数应返回新长度 length = 5，并且原数组的前五个元素被修改为 1, 1, 2, 2, 3 。 不需要考虑数组中超出新长度后面的元素。

输入：nums = [0,0,1,1,1,1,2,3,3]
输出：7, nums = [0,0,1,1,2,3,3]
解释：函数应返回新长度 length = 7，并且原数组的前五个元素被修改为 0, 0, 1, 1, 2, 3, 3 。 不需要考虑数组中超出新长度后面的元素。

看到这个题，你是否会认同文章开头说的，算法就是给基本数据结构操作换个条件，来回折腾？这里如果改成三次、四次、K次是不是就是面试可能会遇到的题?

为了让解法更有普适性，我们就将保留2个修改为保留K个。

对于此类问题，我们应该进行如下考虑：

（1）由于是保留 k 个相同数字，对于前 k 个数字，我们可以直接保留
（2）对于后面的任意数字，能够保留的前提是：与当前写入的位置前面的第 k 个元素进行比较，不相同则保留

还是看个例子吧：

[0,1,1,1,1,1,1,2,2,2,2,2,2,3]，假如K是5。

（1）我们先让前 2 位直接保留，得到 0,1
（2）对后面的每一位继续遍历，能够保留的前提是与当前位置的前面 2个元素不同（也就是上一步的0 1），因此我们会保留一个1，得到 0,1,1
（3）之后，会跳过所有的1，找到2，得到 0,1,1,2，然后保留后的一个2，得到0,1,1,2,2
（4）依次类推，得到：0,1,1,2,2,3

对于两个的情况，我们仍然可以使用快慢指针。slow之前的都已经处理完毕了，而fast则是当前访问的位置。

```
class Solution {
    public int removeDuplicates(int[] nums) {
        int n = nums.length;
        if (n <= 2) {
            return n;
        }
        //前两个元素不必管
        //slow=2表示索引为2的位置可以被替换
        int slow = 2, fast = 2;
        while (fast < n) {
            if (nums[slow - 2] != nums[fast]) {
                nums[slow] = nums[fast];
                ++slow;
            }
            ++fast;
        }
```

```
        return slow;
    }
}
```

再回到本题的要求，2个相同则保留，我们几乎什么都不动，传入的参数k设置为2就行了：

```
public int removeDuplicates(int[] nums) {
        return process(nums, 2);
    }
```

看到了吗？假如面试官给你任意的K，你还怕吗？如果我们刷到这个程度，刷题才是真正有效的，而不会刷了很多题还是一脸懵*。

我们来造题：

既然是换条件再折腾，那我们自己造一个题：在有序数组中，凡是重复的一个都不要，该怎么做呢？

这里貌似用双指针就不够了，需要借助Hash和双指针才行，第二种方法是通过三个指针来实现，请读者思考。

问题2 数组中的元素两次重复

假定有些元素出现两次而有的只出现了一次，该怎么找到所有出现两次的元素呢？先看一下完整要求：

```
给定一个整数数组 a，其中1 ≤ a[i] ≤ n （n为数组长度），其中有些元素出现两次而其他元素出现一次。
找到所有出现两次的元素。
你可以不用到任何额外空间并在O(n)时间复杂度内解决这个问题吗？

示例：
输入：
[4,3,2,7,8,2,3,1]
输出：
[2,3]
```

这个题目使用Hash是可以的，value也是出现的次数，但是申请一个O(n)空间还是不占优势，有没有更巧妙的方法呢？可以这么干：

由于1<=nums[i]<=n(数组长度)，所以（nums[i]-1）可以成为nums中的下标，记index=nums[i]-1又因为1<=nums[i]<=n，
所以可以通过nums[i]每出现过一次之后对nums[index]+=n,确保当nums[index]>2*n时，index+1(即nums[i])出现过两次。
小细节：这里通过对index取模来保证其仍在下标范围内。

代码如下：

```
public List<Integer> findDuplicates(int[] nums) {
        List<Integer> res=new ArrayList<Integer>();
        int n=nums.length;
        for(int i=0;i<nums.length;i++)
        {
            int index=(nums[i]-1)%n;
            nums[index]+=n;
            if(nums[index]>2*n)res.add(index+1);
        }
        return res;
    }
```

如果想不明白，可以看一个例子，对于序列 [4,3,2,7,8,2,3,1]

首先这里元素个数为8，所有元素的值也不超过8，所以每一个值都可以作为索引。然后执行如下操作:

首先index=(nums[0]-1)%8=3,所以将nums[3]+8=15,所以数组变成了[4,3,2,15,8,2,3,1]

继续向前，index=(nums[1]-1)%8=2，所以将nums[2]+8=10,数组变成了[4,3,10,15,8,2,3,1]

继续向前，index=(nums[2]-1))%8=1，所以将nums[1]+8=12，数组变成了[4,12,10,15,8,2,3,1]

继续向前，index=(nums[3]-1)%8=6,所以将nums[6]+8=11,数组变成[4,12,10,15,8,2,11,1]

继续向前，index=(nums[4]-1)%8=7，所以将nums[7]+8=9，数组变成[4,12,10,15,8,2,11,9]

继续向前，index=(nums[5]-1)%8=1，所以将nums[1]+8=11，数组变成了[4,20,10,15,8,2,11,9]

继续向前，index=(nums[6]-1)%8=2，所以将nums[2]+8=18,数组变成了[4,20,18,15,8,2,11,9]

最后一次，index=(nums[7]-1)%8=0，所以将nums[0]+0=12,数组最终变成了[12,20,18,15,8,2,11,9]

仔细观察这个过程你会发现，索引为1和2的位置被加了两次8，而本来就有值，所以最终结果一定是大于2*n的，因此执行完这个过程之后，凡是满足 nums[index]>2*n 的都是重复的，这里恰好对应的就是2和3。

这个方法比较难想到，而且非常巧妙，我们可以熟悉这种方式，上述过程可以自己耐心写一下看看就会恍然大悟。

5.6 数组的区间专题

数组中表示的数组可能是连续的，也可能是不连续的，如果将连续的空间标记成一个区间，那么我们可以再造几道题，先看一个例子：

```
给定一个无重复元素的有序整数数组nums。返回恰好覆盖数组中所有数字的最小有序区间范围列表。也就是说，nums 的每个元素都恰好被某个区间范围所覆盖，并且不存在属于某个范围但不属于 nums 的数字 x 。
列表中的每个区间范围 [a,b] 应该按如下格式输出：
"a->b" ，如果 a != b
"a" ，如果 a == b

示例1：
输入：nums = [0,1,2,4,5,7]
输出：["0->2","4->5","7"]
解释：区间范围是：
[0,2] --> "0->2"
```

```
[4,5] --> "4->5"
[7,7] --> "7"

示例2:
输入: nums = [0,2,3,4,6,8,9]
输出: ["0","2->4","6","8->9"]
解释: 区间范围是:
[0,0] --> "0"
[2,4] --> "2->4"
[6,6] --> "6"
[8,9] --> "8->9"
```

这个题目是典型的容易让人眼高手低，结果一眼就看出来，而实现则很麻烦，一个是里面有很多条件要处理，另一个是还要处理字符串的问题。

这个题使用双指针也可以非常方便的处理，i 指向每个区间的起始位置，j 从 i 开始向后遍历直到不满足连续递增（或 j 达到数组边界），则当前区间结束；然后将 i 指向更新为 j + 1，作为下一个区间的开始位置，j 继续向后遍历找下一个区间的结束位置，如此循环，直到输入数组遍历完毕。

```
class Solution {
    public List<String> summaryRanges(int[] nums) {
        List<String> res = new ArrayList<>();
        // i 初始指向第 1 个区间的起始位置
        int i = 0;
        for (int j = 0; j < nums.length; j++) {
            // j 向后遍历，直到不满足连续递增(即 nums[j] + 1 != nums[j + 1])
            // 或者 j 达到数组边界，则当前连续递增区间 [i, j] 遍历完毕，将其写入结果列表。
            if (j + 1 == nums.length || nums[j] + 1 != nums[j + 1]) {
                // 将当前区间 [i, j] 写入结果列表
                StringBuilder sb = new StringBuilder();
                sb.append(nums[i]);
                if (i != j) {
                    sb.append("->").append(nums[j]);
                }
                res.add(sb.toString());
                // 将 i 指向更新为 j + 1，作为下一个区间的起始位置
                i = j + 1;
            }
        }
        return res;
    }
}
```

我们本着不嫌事大的原则，假如这里是要你找缺失的区间该怎么做呢？例如：

```
示例:
输入: nums = [0, 1, 3, 50, 75], lower = 0 和 upper = 99,
输出: ["2", "4->49", "51->74", "76->99"]
```

6.总结

今天我们总结了一维数组的基本问题和一些常见面试题，万事开头难，练习好了，对于我们后面继续学习其他内容非常有好处。

我们说算法要看清本质，看清内在逻辑。本质是什么呢？其实就是如何针对其特征进行高效增删改查。而每种特征又可以扩展出大量的题目，也就是所谓的小专题了。因为每个特征都有自己的共性问题和基础问题，所以我们刷题的目标就是搞清楚这些共性问题如何处理。例如数组的基本问题就是准确控制指针的移动，而添加删除的时候要注意如何处理数据大量移动的问题，进制问题要处理如何连锁修改数据的问题等等。

只有这些问题都刷清楚了，如果遇到新的变形题，那我们只需要关注如何针对新场景进行调整就行了。这才是我们要达到的目标。

另外，我们已经初步感受到了什么是融会贯通，上面的题目里已经用上了排序的三大重点：快速排序、归并排序和冒泡排序。不信请看：

1.上面解题用到了双指针方法，很多是从数组两端向中间移动，而这个过程也是快速排序的基本原理，只不过快速排序要多次执行，更为复杂一些。

2.多个数组合并可以采用先两两合并，再将合并的再次两两合并，这就是归并排序的基本思想。

3.在奇偶移动部分为了保证调整后奇数之间以及偶数仍然与原始位置一样，我们采用了冒泡排序的思想，只要将比较大小改成比较奇偶就可以了。

所以我们后面学习排序的时候就容易很多了，很多题目也有类似情况，看到这一点，我们才会真正明白为什么题目会越刷越少。

最后，常见的关于数组的问题还有很多，有些还需要不少技巧才可以，所以看似简单的数组并不简单，我们先掌握这些，遇到新问题见招拆招。