学习目标

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

2.理解数组的特点，理解其数据存储的特征。

3.自己实现在数组中增加和删除元素任意位置的元素。

4.理解如何避免频繁的大量进行元素移动，理解双指针思想。

5.掌握数组有关的典型题目。

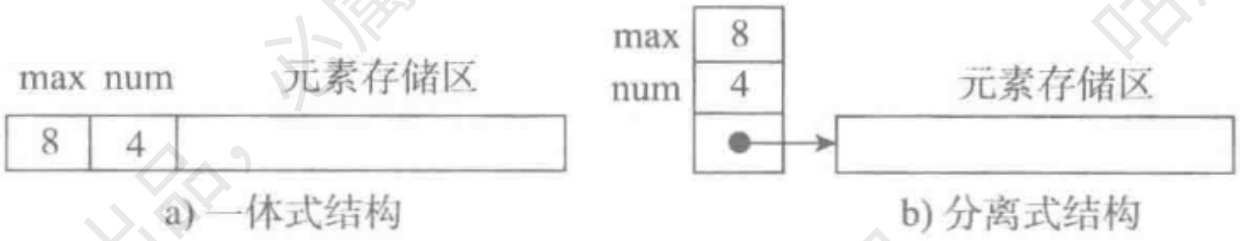
1.线性表基础

1.1 线性表

我们先搞清楚几个基本概念，在很多地方会看到线性结构、线性表这样的表述，那什么是线性结构？与数组、链表等有什么关系？常见的线性结构又有哪些呢？

所谓线性表就是具有相同特征数据元素的一个有限序列，其中所含元素的个数称为线性表的长度，从不同的角度看，线性表可以有不同的分类，例如：

从语言实现的角度顺序表有两种基本实现方式，一体式和分离式，如下：



图a为一体式结构，存储表信息的单元与元素存储区以连续的方式安排在一块存储区里，两部分数据的整体形成一个完整的顺序表对象。这种结构整体性强，易于管理。但是由于数据元素存储区域是表对象的一部分，顺序表创建后，元素存储区就固定了。C和C++都是一体式的结构。

图b为分离式结构，表对象里只保存与整个表有关的信息（即容量和元素个数），实际数据元素存放在另一个独立的元素存储区里，通过链接与基本表对象关联。Java和python是分离式结构。

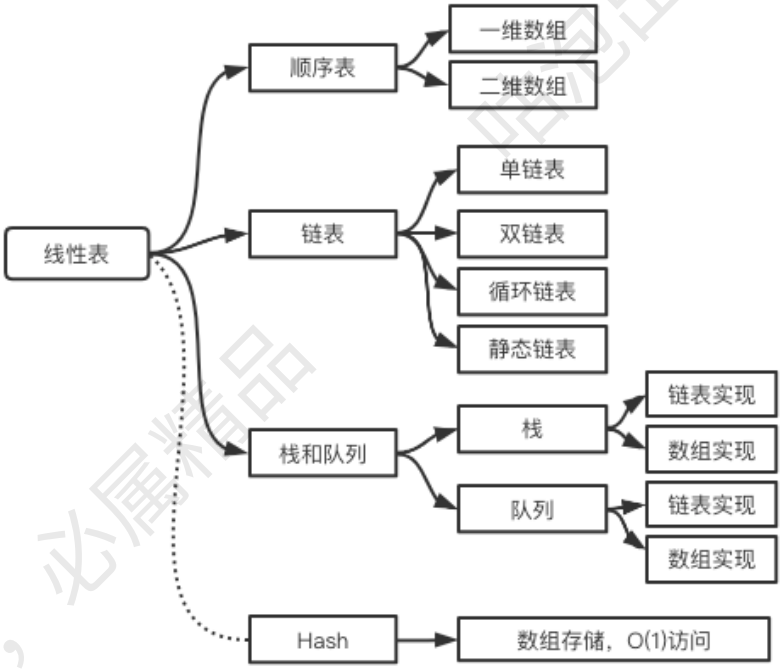
从存储的角度

一体式结构由于顺序表信息区与数据区连续存储在一起，所以若想更换数据区，则只能整体搬迁，即整个顺序表对象（指存储顺序表的结构信息的区域）改变了。

分离式结构若想更换数据区，只需将表信息区中的数据区链接地址更新即可，而该顺序表对象不变。顺序表主要是数组，而链表的实现又有单链表、循环链表、双向链表等等等。

从访问限制的角度

栈和队列又称为访问受限的线性表，插入和删除受到了限制，只能在固定的位置进行。而Hash比较特殊，其内部真正存储数据一般是数组，但是访问是通过映射来实现的，因此大部分材料里并不将Hash归结到线性表中，这里为了学习更紧凑，我们将其与队栈一起学习。线性表的知识框架如下：



线性表的常见操作有初始化、求表长、增删改查等，事实上每种数据结构都至少要有这几种操作，大部分的基础算法题都是基于此扩展的。

还有一种比较特殊的“静态链表”，就是是顺序表和链表的升级；静态链表的数据全部存储在数组中(顺序表)，但存储的位置是随机的，数据直接的一对一关系是通过一个整型变量(称为“游标”，类似指针的功能)维持，这样可以兼顾顺序表和链表的优点。不管工程里还是算法里都很少用到，所以就不再细谈，感兴趣的同学可以自己了解一下。

从扩容的角度

采用分离式结构的顺序表，若将数据区更换为存储空间更大的区域，则可以在不改变表对象的前提下对其数据存储区进行了扩充，所有使用这个表的地方都不必修改。只要程序的运行环境（计算机系统）还有空闲存储，这种表结构就不会因为满了而导致操作无法进行。人们把采用这种技术实现的顺序表称为动态顺序表，因为其容量可以在使用中动态变化。

扩充的两种策略：

- 第一种：每次扩充增加固定数目的存储位置，如每次扩充增加10个元素位置，这种策略可称为线性增长。

  特点：节省空间，但是扩充操作频繁，操作次数多。
- 第二种：每次扩充容量加倍，如每次扩充增加一倍存储空间。

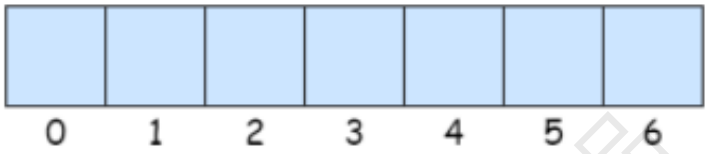
  特点：减少了扩充操作的执行次数，但可能会浪费空间资源。以空间换时间，推荐的方式。

具体到每种结构语言中的结构，实现方式千差万别。其中Java基本是扩容时加倍的方式。而在Python的官方实现中，list实现采用了如下的策略：在建立空表（或者很小的表）时，系统分配一块能容纳8个元素的存储区；在执行插入操作（insert或append）时，如果元素存储区满就换一块4倍大的存储区。但如果此时的表已经很大（目前的阀值为50000），则改变策略，采用加一倍的方法。引入这种改变策略的方式，是为了避免出现过多空闲的存储位置。

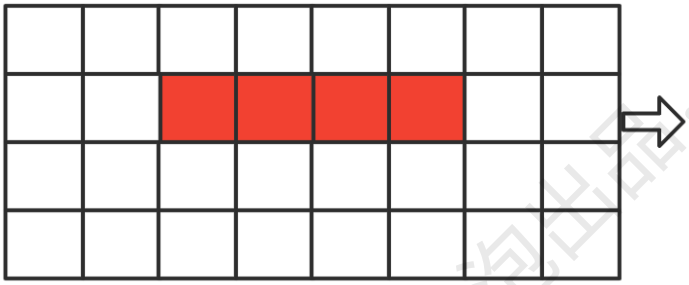
1.2 数组的概念

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。

数组用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

另外需要注意的是数组空间不一定是满的，100的空间可能只用了10个位置，所以要注意数据个数的变量size和数组长度length可能不一样，解题时必须注意。

1.3 数组存储元素的特征

在往期训练营中，发现很多学员对数组如何存储元素的并不太清楚，这里采用连环炮方式来说明。

第一炮，我创建了一个大小为10的数组 int[] arr = new int[10]，请问此时数组里面是什么?

答：不同的语言处理会不一样，在c语言里每个位置都是一个随机数。而在java里，默认会初始化为0。而python更为灵活可以直接指定是什么，例如a = [1,2,3,4]，就是数组里有四个元素，而 `a = [0 for i in range(10)]` 这样定义的数组就是[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

第二炮：是否可以只初始化一部分位置？初始化的本质是什么?

答：当然可以，你可以将前面5个位置依次，后面的空着，此时数组内容为{1,2,3,4,5,0,0,0,0,0}。

初始化的本质就是覆盖已有的值，用你需要的值覆盖原来的0，因为数组本来是{0,0,0,0,0,0,0,0,0,0}，这里只不过被你替换成了{1,2,3,4,5,0,0,0,0,0}。如果此时你想知道有效元素的个数，就必须再使用一个额外的变量，例如size来标记。

第三炮：上面已经初始化的元素之间是否可以空着，例如初始化为{1,0,0,4,5,0,2,0,3,0}。其中0位置仍然是未初始化的?

答：不可以！绝对不可以！要初始化，就必须从前向后的连续空间初始化，不可以出现空缺的情况，这是违背数组的原则的。你正在进行某种运算期间可以先给部分位置赋值，而一旦稳定了，就不可以再出现空位置的情况。

第四炮：如果我需要的数据就是在中间某一段该怎么办呢？例如{0,0,3,4,5,6,7,0,0,0}，此时该怎么拿到从3到7的元素呢?

答：你需要使用两个变量，例如left=2，right=6来表示区间[left,right]是有效的。

第五炮：我删除的时候，已经被删除的位置该是什么呢？例如原始数组为{1,2,3,4,5,6,7,8,0,0}，我删除4之后，根据数组的移动原则，从5开始向前移动，变成{1,2,3,5,6,7,8,?,0,0}，那原来8所在的位置应该是什么呢?

答：仍然是8，也就是删除4之后的结构为{1,2,3,5,6,7,8,8,0,0}，此时表示元素数量的size会减1变成7，原来8的位置仍然是8。因为我们是通过size来标记元素数量的，所以最后一个8不会被访问到。

第六炮：这个里8看起来很不爽啊，是否可以再优化一下?

答：不爽就不爽，习惯就好！不用优化，优化了也没啥用。

1.4 数组基本操作

本节参考代码位置：unit1_一维数组.part1_basic_array 包下的文件

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。

1.4.1 数组创建和初始化

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？显然不能用循环了，可以采用下面的方式：

```
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，直接将其替换就行了。这种方式很简单，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，想死的心都有! 我们练习算法的一个目标就是熟悉这些基本问题，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，如果需要增加新元素就不能这么用了。

作业：

将int[] nums = {2, 5, 0, 4, 6, -10}这种赋值方法背下来。

是的！！！背下来，没什么好说的。

1.4.2 查找一个元素

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、动态规划问题等等都是在寻找那个目标结果。

这里只写最简单的方式，根据值是否相等进行线性查找，基本实现如下：

```
/**
* @param size 已经存放的元素个数
* @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;
  }
```

作业

还有一种很常见的情况，如果数组是递增的，此时查找时如果相等或者当前位置元素比目标值更大就停下了。 你是否可以修改上面的代码来实现这个功能?

1.4.3 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之一。所以务必自己亲手能写一个才可以，**不要感觉挺简单就不写，其中涉及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！！！！

将给定的元素插入到有序数组的对应位置中，我们可以先找位置，再将其后元素整体右移，最后插入到空位置上。这里需要注意，算法必须能保证在数组的首部、尾部和中间位置插入都可以成功。该问题貌似一个for循环就搞定了，但是如果面试直接让你写并能正确运行，我相信很多人还是会折腾很久，甚至直接会挂。因为自己写的时候会发现游标写size还是size-1，判断时要不要加等于等等，这里推荐一种实现方式。

```
/**
     * @param arr
     * @param size    数组已经存储的元素数量，从1开始编号
     * @param element 待插入的元素
     * @return
     */
    public static int addByElementSequence(int[] arr, int size, int element) {
```

```
        //问题①：是否应该是size>arr.length
        if (size >= arr.length)
             throw new IllegalArgumentException("Add failed. Array is full.");

         //问题②想想这里是否是index=0或者size-1?
          int index = size;
         //找到新元素的插入位置，问题③ 这里是否应该是size-1?
         for (int i = 0; i < size; i++) {
             if (element < arr[i]) {
                 index = i;
                 break;
             }
         }
         //元素后移，问题④想想这里为什么不是size-1
         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;
    }
```

上面的代码在往期课程里被提出疑问特别多，主要是标记编号的几个位置，这几个全都是边界问题。这里回答几个：

问题①处，注意这里的size是从1开始编号的，表示的就是实际元素的个数。而arr.length也是从1开始的，当空间满的时候就是size=arr.length，此时就不能再插入元素了。

问题② 处只能令index=size， 0或者size-1都不对。例如已有序列为{3,4,7,8}，如果插入的元素比8大，例如9，假如index=0，则最后结果是{9,3,4,7,8}。假如index=size-1，最后结果就是{3,4,7,9,8}。

问题③和④处，这个就不用解释了吧，请读者自己思考。

作业

除了上面的方式，还可以一开始就从后向前一边移动一边对比查找，找到位置直接插入。从效率上看这样更好一些，因为只遍历了一次，你是否可以实现一下呢?

1.4.4 删除一个元素

对于删除，不能一边从后向前移动一边查找了，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。

这个方法和增加元素一样，必须自己亲自写才有作用，该方法同样要求删除序列最前、中间、最后和不存在的元素都能有效，下面给一个参考实现：

```
    /**
    * 从数组中删除元素key
    * @param arr 数组
    * @param size 数组中的元素个数，从1开始
    * @param key   删除的目标值
    */
```

```
public  int 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--;
        }
        return size;
    }
```

1.5 java中的数组

java对数组做了一定的封装，封装后的数组提供了reverse()，sort()等方法，这在解决复杂问题的时候可以直接用。你是否注意到，jdk里竟然有三个数组相关的类：Array、Arrays和ArrayList，这三个有啥区别呢?

【1】Array与Arrays类的区别

在java中的这两个类不是在一个包里的，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，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）。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。

【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功能更为强大，这个类在我们业务里用得更多。

【3】Arrays的使用

Arrays提供的方法都是静态的，比较重要的有：

① void Arrays.sort()

void Arrays.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。该方法还能对数组元素进行指定范围排序：

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

功能：返回数组的字符串形式，这个方法在我们平时写代码检查结果时特点有用。

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

输出结果：[2, 5, 0, 4, 1, -10]

③Arrays.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。

1.6 本课的代码风格

我们在课程里会大量使用LeetCode中的题目，所有的代码都要求能通过，因此本讲义的代码也按照LeetCode 的进行。

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

```
public  boolean isSorted(int[] nums, boolean increasing) {
          int n = nums.length;
           for (int i = 0; i < n - 1; ++i) {
               if(increasing){
                  if (nums[i] > nums[i + 1]) {
                    return false;
                  }
                }else{
                    if (nums[i] < nums[i + 1]) {
                    return false;
                  }
                }
          }
       return true;
  }
      public static void main(String[] args) {
        int a[] = {1, 2, 2, 3};
        int testMethod = 1;
//        通过两次遍历来实现
        System.out.println(isMonotonic(a));
    }
```

2.高频面试题

数组是很多高级算法的载体， 整个LeetCode中有接近一半的题目都与数组有关。一直到动态规划，数组将伴随我们整个算法课程，因此一维数组题不会，可能是你高阶知识不会，这里我们先学习一些比较基础的问题。

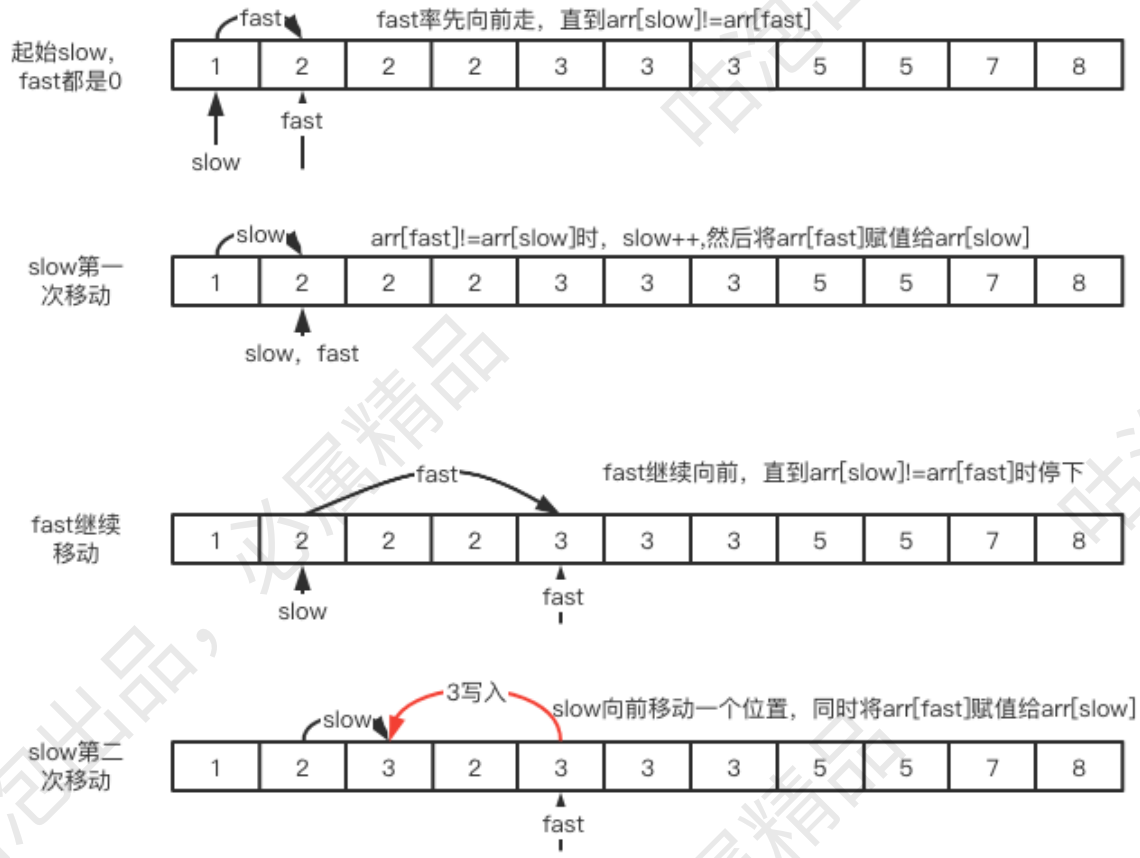
2.1 数组问题常用思想

我们首先介绍几个常用的思想，然后再看题目。

2.1.1 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，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。在后面可以看到很多算法题都需要多轮、大量移动元素，这就导致执行效率低下。如何解决该问题是数组算法的一个重要问题。

这里介绍一种简单但非常有效的方式——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。双指针思想简单好用，在处理数组、字符串等场景下很常见。看个例子，从下面序列中删除重复元素[1,2,2,2,3,3,3,5,5,7,8]，重复元素只保留一个。删除之后的结果应该为[1,2,3,5,7,8]。我们可以在删除第一个2时将将其后面的元素整体向前移动一次，删除第二个2时再将其后的元素整体向前移动一次，处理后面的3和5都一样的情况，这就导致我们需要执行5次大量移动才能完成，效率太低。如果使用双指针可以方便的解决这个问题，如图：



首先我们定义两个指针slow、fast。slow表示当前位置之前的元素都是不重复的，而fast则一直向后找，直到找到与slow位置不一样的，找到之后就将slow向后移动一个位置，并将arr[fast]复制给arr[slow]，之后fast继续向后找，循环执行。找完之后slow以及之前的元素就都是单一的了。这样就可以只用一轮移动解决问题。

上面这种一个在前一个在后的方式也称为快慢指针，有些场景需要从两端向中间走，这种就称为对撞型指针或者相向指针，很多题目也会用到，我们接下来会看到很多相关的算法题。

还有一种比较少见的背向型，就是从中间向两边走。这三种类型其实非常简单，看的只是两个指针是一起向前走（相亲相爱一起走），还是从两头向中间走（冲破千难万险来爱你），还是从中间向两头走（缘分已尽，就此拜拜）。

2.1.2 排序思想和集合思想

在工程中很多问题排序一下就会变得比较简单，但由于排序的代价比较高，所以要慎用。除了排序，在处理重复等情况时，Hash也是一个非常好用的方法，但是Hash只是个备胎，因为很多问题一用Hash就太简单了，没有思维含量了，因此只要有其他方式一般就不允许使用了。
那具体什么时候才可以用呢？这个没有定论，个人感觉如果用了之后就没什么技术含量了就不用。如果面试时拿不准，可以直接问面试官。

2.1.3 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。这些问题比较灵活，具体实现更为灵活。因此如果对边界等拿不准了，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。
该方法在解决二维数组问题也同样非常有效，因为二维数组里要处理四周多个位置的边界条件。

2.2. 单调数组问题

先看个热身问题，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。
LeetCode 896.判断一个给定的数组是否为单调数组。

分析：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  boolean isMonotonic(int[] nums) {
        return isSorted(nums, true) || isSorted(nums, false);
    }

public  boolean isSorted(int[] nums, boolean increasing) {
          int n = nums.length;
           for (int i = 0; i < n - 1; ++i) {
               if(increasing){
                 if (nums[i] > nums[i + 1]) {
                   return false;
                 }
               }else{
                   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，而且还要遍历两次，能否优化一下呢？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[j]<a[j+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leetcode35：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。以后凡是提到在单调序列中查找的情况，我们应该马上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;
    }
}
```

2.3.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。还有后面要学的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。

2.3.1 合并两个有序数组

先来看如何合并两个有序数组，LeetCode88：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: 合并 [1,2,3] 和 [2,5,6] 的结果是 [1,2,2,3,5,6]
```

对于有序数组的合并，一种简单的方法是先将B直接合并到A的后面，然后再对A排序，也就是这样:

```
public  void merge1(int[] nums1, int nums1_len, int[] nums2, int nums2_len) {
        for (int i = 0; i < nums2_len; ++i) {
            nums1[nums1_len + i] = nums2[i];
        }
        Arrays.sort(nums1);
    }
```

但是这么写只是为了开拓思路，面试官会不喜欢，太没技术含量了。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。此时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？更专业的问法是：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，每次都找最大的那个从后向前填就可以了，代码如下:

```
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--];
    }
```

分析上述代码，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，代码如下:

```
public  void merge3(int[] nums1, int nums1_len, int[] nums2, int nums2_len) {
        int indexA=nums1_len-1;
        int indexB=nums2_len-1;
        int index=nums1_len+nums2_len-1;
        while (indexA>=0&&indexB>=0){
            nums1[index--]=nums1[indexA]>=nums2[indexB]?nums1[indexA--]:nums2[indexB--
];
        }
        //假如A或者B数组还有剩余
        while (indexB>=0){
            nums1[index--]=nums2[indexB--];
        }
    }
```

2.3.2 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你加餐：如果是n个长度一样的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。第二种方式不断两两合并就是归并的思想，我们到学习归并排序时再看这个问题。第三种我们在学习完堆之后再详细解释。这里只看第一种，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，实现如下。

```
import java.util.Arrays;
   public static int[] mergeArrays(int[][] array) {
        int arrNums = array.length, arrLen;
        if (arrNums == 0) {
            return new int[0];
        }

        //数组长度校验
        arrLen = array[0].length;
        for (int i = 1; i < arrNums; i++) {
            if (arrLen != array[i].length) {
                return new int[0];
            }
        }

        //开辟新空间
        int[] result = new int[arrNums * arrLen];
        //将各个数组依次合并到一起
        for (int i = 0; i < arrNums; i++) {
            for (int j = 0; j < arrLen; j++) {
                result[i * arrLen + j] = array[i][j];
            }
        }
```

```
        //排序一下
        Arrays.sort(result);
        return result;
    }
```

2.4.字符串替换空格问题

这是剑指offer中的题目，出现频率也很高：

> 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。

首先要考虑用什么来存储字符串，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，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。

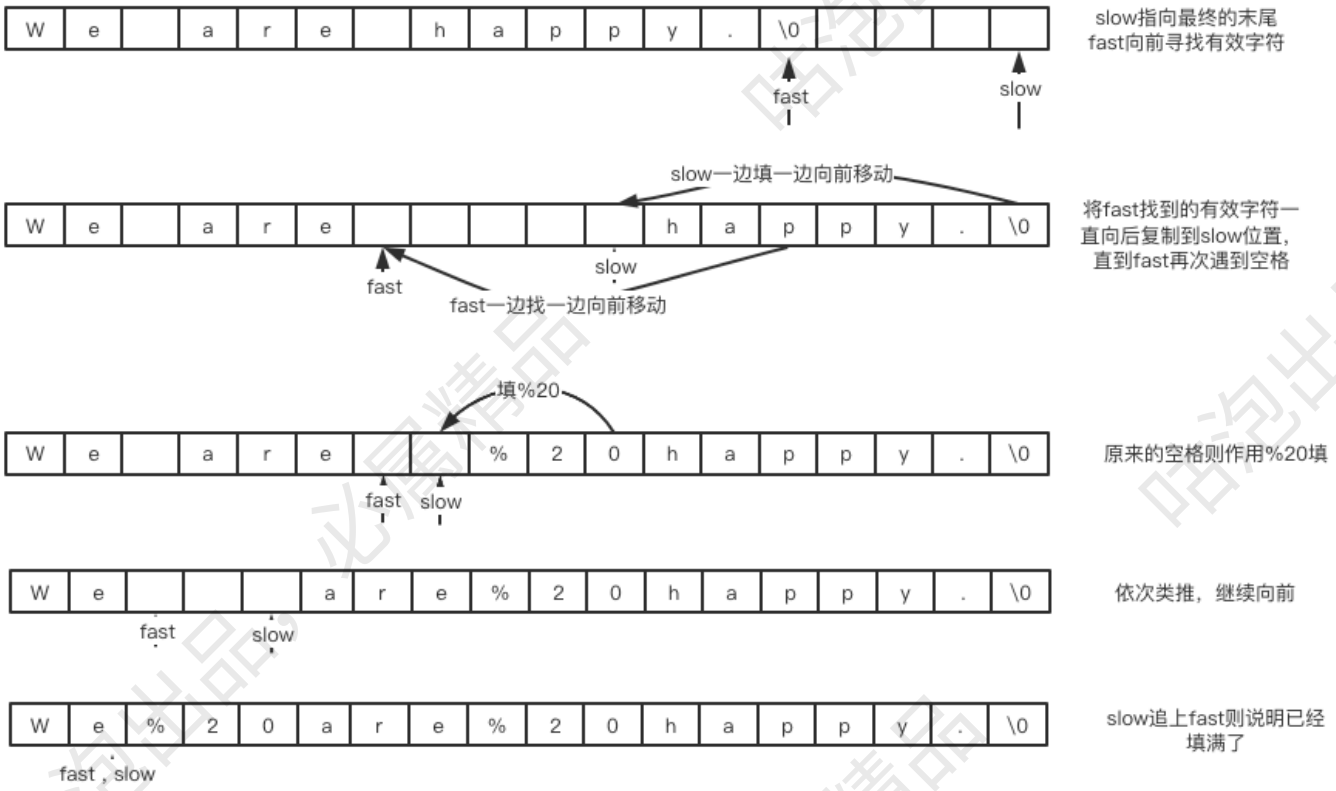
比较好的方式是可以先遍历一次字符串，这样可以统计出字符串中空格的总数，由此计算出替换之后字符串的长度，每替换一个空格，长度增加2，即替换之后的字符串长度为：

`新串的长度=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fast和slow分别指向原始字符串和新字符串的末尾，然后：slow不动，向前移动fast：

- 若指向的不是空格，则将其复制到slow位置，然后fast和slow同时向前一步；
- 若fast指向的是空格，则在slow位置插入一个%20，fast则只移动一步。

循环执行上面两步，便可以完成替换。详细过程如下：



实现代码如下：

```
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 fast = len - 1;  //两个指针
     int slow = (len + 2 * numOfblank) - 1;

     while (fast >= 0 && slow > fast) {
         char c = str.charAt(fast);
         if (c == ' ') {
             fast--;
             str.setCharAt(slow--, '0');
             str.setCharAt(slow--, '2');
             str.setCharAt(slow--, '%');
         } else {
             str.setCharAt(slow, c);
             fast--;
             slow--;
         }
     }
```

```
        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));
}
```

2.5.删除元素专题

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。

2.5.1 原地移除所有数值等于 val 的元素

LeetCode27.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。

```
例子1：
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，所以这题的关键是如果有很多值为val的元素的时候，如何避免反复向前移动呢?

本题可以使用双指针方式，而且还有三种，我们都看一下：

第一种：快慢双指针

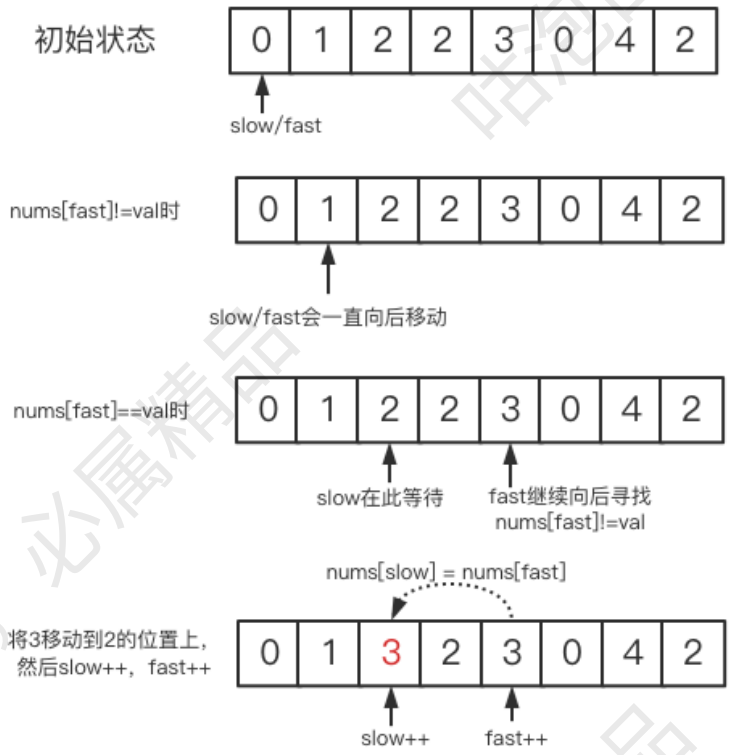
整体思想就是2.1.1中介绍的双指针的图示的方式，定义两个指针slow和fast，初始值都是0。

Slow之前的位置都是有效部分，fast表示当前要访问的元素。

这样遍历的时候，fast不断向后移动：

```
*  如果nums[fast]的值不为val，则将其移动到nums[slow++]处。
*  如果nums[fast]的值为val，则fast继续向前移动，slow先等待。
```

图示如下：

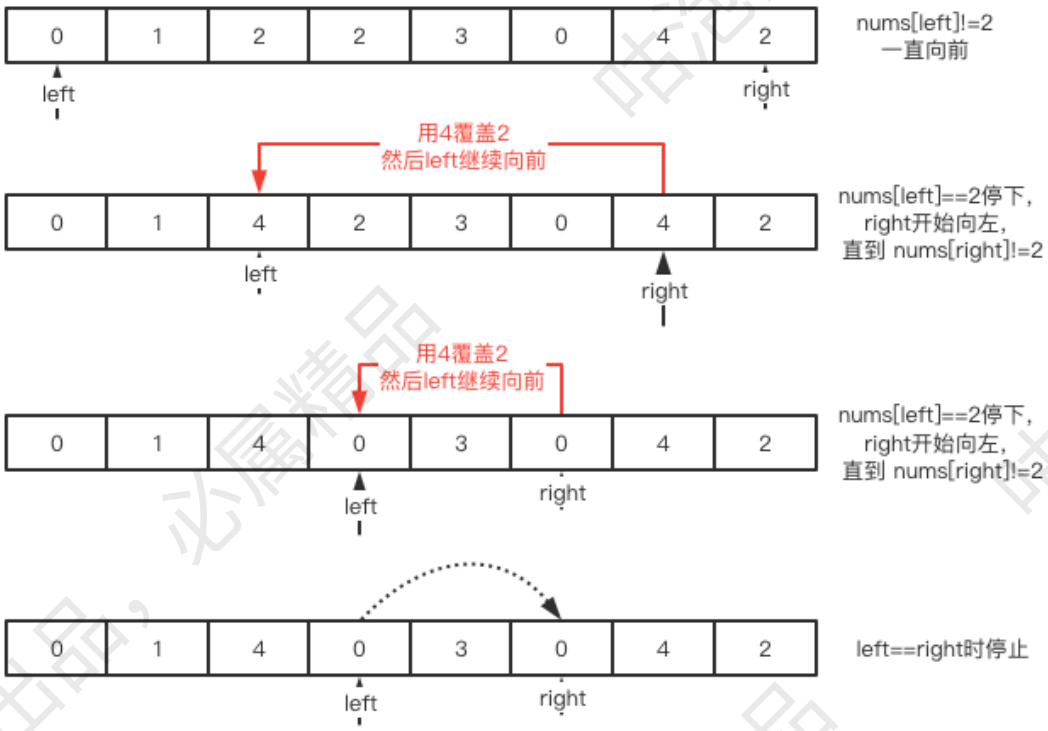


这样，前半部分是有效部分，后半部分是无效部分。

```
public static int removeElement(int[] nums, int val) {
        int slow = 0;
        //fast充当了快指针的角色
        for (int fast = 0; fast < nums.length; fast++) {
            if (nums[fast] != val) {
                nums[slow] = nums[fast];
                slow++;
            }
        }
        //最后剩余元素的数量
        return slow;
    }
```

第二种：对撞双指针

对撞指针，有的地方叫做交换移除，核心思想是从右侧找到不是val的值来顶替左侧是val的值。什么意思呢？我们看图，我们以nums = [0,1,2,2,3,0,4,2], val = 2为例：



上图完整描述了执行的思路，当left==right的时候，left以及左侧的就是删除掉2的所有元素了。实现代码：

```
public int removeElement(int[] nums, int val) {
        int right = nums.length - 1;
       int left = 0;

       for (left = 0; left <= right; ) {
           if ((nums[left] == val) && (nums[right] != val)) {
               int tmp = nums[left];
               nums[left] = nums[right];
               nums[right] = tmp;
           }
           if (nums[left] != val) left++;
           if (nums[right] == val) right--;
       }
       return left ;
   }
```

这样就是一个中规中矩的的双指针解决方法。

拓展本题还可以进一步融合上面两种方式创造出：“对撞双指针+覆盖”法。当nums[left]等于val的时候，我们就将nums[right]位置的元素覆盖nums[left]，继续循环，如果nums[left]等于val就继续覆盖，否则才让left++，这也是双指针方法的方法，实现代码：

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

对撞型双指针的过程与后面要学习的快速排序是一个思路，快速排序要比较很多轮，而这里只执行了一轮，理解本题将非常有利于后面理解快速排序算法。

另外，我们可以发现快慢型双指针留下的元素顺序与原始序列中的是一致的，而在对撞型中元素的顺序和原来的可能不一样了。

2.5.2 删除有序数组中的重复项

LeetCode26 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。

```
示例1：
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。
```

本题使用双指针最方便，思想与2.1.1的一样，一个指针负责数组遍历，一个指向有效数组的最后一个位置。为了减少不必要的操作，我们做适当的调整，例如令slow=1，并且比较的对象换做nums[slow - 1]，代码如下：

```
    public static int removeDuplicates(int[] nums) {
        //slow表示可以放入新元素的位置，索引为0的元素不用管
        int slow = 1;
        //循环起到了快指针的作用
        for (int fast = 0; fast < nums.length; fast++) {
            if (nums[fast] != nums[slow - 1]) {
                nums[slow] = nums[fast];
                slow++;
            }
        }
        return slow;
    }
```

你能参考2.1.1的图示将上面代码的图示画一下吗?

作业

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？感兴趣的同学可以继续研究一下LeetCode80一题。

2.6.元素奇偶移动专题

根据某些条件移动元素也是一类常见的题目，例如排序本身就是在移动元素，这里看一个奇偶移动的问题。

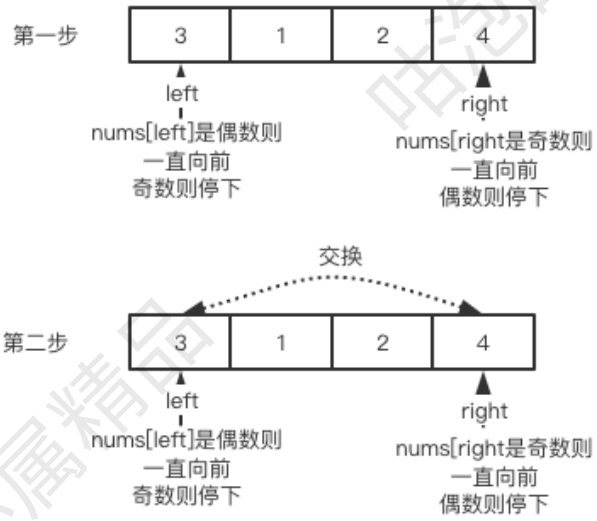
2.6.1 奇偶数分离

LeetCode905，按奇偶排序数组。给定一个非负整数数组 `A`，返回一个数组，在该数组中， `A` 的所有偶数元素之后跟着所有奇数元素。你可以返回满足此条件的任何数组作为答案。

```
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。
```

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到新数组的前部分，第二遍查找并复制所有的奇数到数组后部分。这种方式实现比较简单，但是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。

我们可以采用对撞型双指针的方法，图示与2.5.2中的对撞型基本一致，只不过比较的对象是奇数还是偶数。如下图所示：



维护两个指针 left=0 和 right=arr.length-1，left从0开始逐个检查每个位置是否为偶数，如果是则跳过，如果是奇数则停下来。然后right从右向左检查，如果是奇数则跳过偶数则停下来。然后交换array[left]和array[right]。之后再继续巡循环，直到left>=right。

```
public static int[] sortArrayByParity(int[] A) {
        int left = 0, right = A.length - 1;
        while (left < right) {
            if (A[left] % 2 > A[right] % 2) {
                int tmp = A[left];
                A[left] = A[right];
                A[right] = tmp;
            }

            if (A[left] % 2 == 0) left++;
            if (A[right] % 2 == 1) right--;
        }

        return A;
    }
```

你有没有发现这种解法与2.5.2里第二种解法对撞指针几乎一样，只是处理条件换了一下？因为这就是对撞型双指针的解题模板！

2.6.2 调整后的顺序仍与原始数组的顺序一致

如果这里要求偶元素和偶元素、奇元素和奇元素之间的相对顺序一致该怎么做呢？例如在上面示例[3,1,2,4]中，奇数3本来是在1前面的，偶数2是在4前面的。如果要求你移动之后要保证这个相对关系不变，该怎么做呢？

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，右边是奇数时就交换两个数
                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;
    }
```

2.7 数组轮转问题

先看题目要求，LeetCode189.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?你是否想到可以逐次移动来实现？理论上可以，但是实现的时候会发现要处理的情况非常多，比较难搞。这里介绍一种简单的方法：两轮翻转。

方法如下：

1. 首先对整个数组实行翻转，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]。
2. 从 k 处分隔成左右两个部分，这里就是根据k将其分成两组 [7,6,5] 和[4,3,2,1]。
3. 最后将两个再次翻转就得到[5,6,7] 和[1,2,3,4]，最终结果就是[5,6,7,1,2,3,4]

代码如下：

```
    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;
        }
    }
```

2.8 数组的区间专题

数组中表示的数据可能是连续的，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：

LeetCode228.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"
```

本题容易让人眼高手低，一眼就看出来结果，但是编程实现则很麻烦。这个题使用双指针也可以非常方便的处理，慢指针指向每个区间的起始位置，快指针从慢指针位置开始向后遍历直到不满足连续递增（或快指针达到数组边界），则当前区间结束；
然后将 slow指向更新为 fast + 1，作为下一个区间的开始位置，fast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。

```
    public static List<String> summaryRanges(int[] nums) {
        List<String> res = new ArrayList<>();
        // slow 初始指向第 1 个区间的起始位置
        int slow = 0;
        for (int fast = 0; fast < nums.length; fast++) {
            // fast 向后遍历，直到不满足连续递增(即 nums[fast] + 1 != nums[fast + 1])
            // 或者 fast 达到数组边界，则当前连续递增区间 [slow, fast] 遍历完毕，将其写入结果列
表。
```

```
            if (fast + 1 == nums.length || nums[fast] + 1 != nums[fast + 1]) {
                // 将当前区间 [slow, fast] 写入结果列表
                StringBuilder sb = new StringBuilder();
                sb.append(nums[slow]);
                if (slow != fast) {
                    sb.append("->").append(nums[fast]);
                }
                res.add(sb.toString());
                // 将 slow 指向更新为 fast + 1，作为下一个区间的起始位置
                slow = fast + 1;
            }
        }
        return res;
    }
```

这个实现的精华是"fast + 1 == nums.length || nums[fast] + 1 != nums[fast + 1]"，我们是用fast+1来进行比较的，如果使用fast比较也可以，但是实现起来会有些case一直过不了，不信你可以试一下。

拓展我们本着不嫌事大的原则，假如这里是要你在上面的情况反过来，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"]
```

这是LeetCode163题，你可以试着做一下。

2.9 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的问题， 修改条件，可以造出很多题目，有些还是比较难的，我们具体来看。

2.9.1 数组中出现次数超过一半的数字

这是剑指offer中的一道题目，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。
```

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，看看谁能帮我们解决问题，所以很多问题就会自然而然的出现多种解法。

首先，用排序行不行？这里说一定存在出现次数超过一半的数字了，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，所以可以直接取出中位数。如果不放心，可以再遍历数组，确认一下这个数字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度取决于排序算法的时间复杂度，最快为O(nlogn)。由于排序的代价比较高，所以我们继续找其他方法。

其次，用Hash行不行？我们先创建一个HashMap的key是元素的值，value是已经出现的次数，然后遍历数组来统计所有元素出现的次数。最后再次遍历Hash，找到出现次数超过一半的数字。OK，第二种方法出来了，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，如果面试时写到这种程度，那就得了80分了。

拓展本题有一种巧妙的解法。上面算法的时间复杂度为O(n)，但是这是用O(n)的空间复杂度换来的。那是否有空间复杂度为O(1)，时间复杂度为O(n)的算法呢?

下面介绍的方法普适性并不好，而且实现也挺麻烦，可以先跳过。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， [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，与上一个不一样，所以times减一为0
然后result=1，与上一个不一样，times已经是0了，遇到新元素就加一为1
然后result=3，与上一个不一样，times减一为0
然后result=1，与上一个不一样，times已经是0了，遇到新元素就加一为1
然后result=4，与上一个不一样，times减一为0
然后result=1，与上一个不一样，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。假如是奇数个，例如[1,2,1,3,1,4,1]，此时结果还是0，但是明显有超过一半的，所以我们最后还是要再遍历一遍来检查一下，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--;
        }

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

2.9.2 数组中只出现一次的数字

LeetCode136.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
```

这个题貌似使用Set集合比较好，Set集合不存重复值，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，当要添加的元素key与集合中已存在的数重复时，不再进行添加操作，而是将集合中的key一起删掉，这样整个数组遍历完后，集合中就只剩下了那个只出现了一次的数字了。

```
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，最后一行代码运行时会出错。

第二种方法：位运算这个题面试官可能还会让你用位运算来做，该怎么办呢?

异或运算的几个规则是：

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

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即可，代码如下：

```
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。

元素次数是非常重要的专题，而且难度略大的问题，而各个大厂又非常喜欢考察。因此，我们提前将常见的场景都研究一遍是非常必要的，重复问题除了我们上面提到的之外，还有大量相关的问题，感兴趣的同学可以继续研究：

- 剑指offer 题目1：找出数组中的重复数字
- 剑指offer题目2：不修改数组找出重复的数字
- LeetCode137：在一个数组中除一个数字出现一次之外，其他数字都出现了三次，请找出那个只出现一次的数字。

3.大厂算法实战

这一章我们梳理了一维数组的基本问题，本章的特点是思维难度往往不大，很多都能直接看出结果，但是想准确写出代码并不容易，如果要运行出来就更难，因此大厂面试也特别喜欢考，而且一般是要求能运行出来。

要掌握数组，我们首先要理解数组的特性，我们在1.3节采用连环炮的方式逐步阐明，希望同学认真对待，可以少走很多坑。

数组的特征是元素是排在一起的，中间不能有空隙，这就导致处理很多场景时会频繁移动元素，为此双指针思想就登场了。我们在2.4 、 2.5 、2.6和2.8小节里都介绍过相关的问题。双指针是一种非常重要的思想，我们要好好掌握。而通过双指针可以进一步拓展出滑动窗口的思想，这个我们在后面有单独的章节讲解。

其次，我们要掌握如果基于数组进行增删改查，这是所有数组算法题的根基。我们说算法要看清本质，看清内在逻辑。本质是什么呢？其实就是如何针对当前类型的数据结构高效进行增删改查。处理元素的时候要考虑数组的首部、中间、尾部分别是如何处理的。遍历的时候变量索引该如何指向、边界如何处理等等，这些都要非常熟练，否则基础一定是不扎实的，宁可少做题，也要将这些问题搞清楚。我们整个《高频面试题》的内容都是以此为基础拓展出来的。

数组是所有人都知道的简单的数据结构，但是面试的时候却经常翻车。主要有三个原因：一个是题目灵活性大，而边界和异常情况等考虑不周导致明明感觉没问题，但执行就是不对；二是这里题目一般要求最高的性能（时间和空间复杂度），而很多好的方法不经过训练往往想不到；第三是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，在LeetCode中有1000多道与数组有关，接近总题目量的一半，可见其应用之广泛，因此题目的难度跨度非常大。如果数组的题不会，不代表你不会数组，而是需要学习一些高阶内容。

本章的内容相对基础，请尽量能自己写一写试试，切勿眼高手低，否则就是简单的不会，难的学不清楚。数组会贯穿我们整个算法课，一些比较难的问题，我们会在合适的章节继续展开。