Broadview
www.broadview.com.cn

全网阅读量近1000万次的算法故事
一群可爱的小仓鼠，带你轻松入门算法和数据结构！

漫画算法 Python篇

小灰的算法之旅

魏梦舒（@程序员小灰）著

中国工信出版集团
电子工业出版社
http://www.phei.com.cn

漫画算法：小灰的算法之旅（Python篇）

第1章　算法概述

1.1　算法和数据结构

1.1.1　小灰和大黄

在大四临近毕业时，计算机专业的同学大都收到了满意的offer，可是小灰却还在着急上火。虽然他这几天面试了很多家IT公司，可每次都被面试官“虐”得很惨很惨。

就在心灰意冷之际，小灰忽然想到，他们系里有一位学霸名叫大黄，大黄不但技术很强，而且很乐于帮助同学。于是，小灰赶紧去找大黄，希望能够得到一些指点。

1.1.2　什么是算法

算法，对应的英文单词是algorithm，这是一个很古老的概念，最早来自数学领域。

有一个关于算法的小故事，估计大家都有耳闻。

在很久很久以前，曾经有一个顽皮又聪明的“熊孩子”，天天在课堂上调皮捣蛋。

终于有一天，老师忍无可忍，对“熊孩子”说：

 臭小子，你又调皮啊！今天罚你算加法，算出1+2+3+4+5+6+7+…，一直加到10000的结果，算不完不许回家！

 嘿嘿，我算就是了。

老师以为，“熊孩子”会按部就班地一步一步计算，就像下面这样。

1+2=3

3+3=6

6+4=10

10+5=15

……

这还不得算到明天天亮？够这小子受的！老师心里幸灾乐祸地想着。

谁知仅仅几分钟后……

 老师，我算完了！结果是50005000，对不对？

 这，这，这……你小子怎么算得这么快？我读书多，你骗不了我的！

看着老师惊讶的表情，“熊孩子”微微一笑，讲出了他的计算方法。

首先把从1到10000这10000个数字两两分组相加，如下：

1+10000=10001

2+9999=10001

3+9998=10001

4+9997=10001

……

一共有多少组这样结果相同的和呢？有10000÷2即5000组。

所以1到10000相加的总和可以这样来计算：

(1+10000)×10000÷2=50005000

这个“熊孩子”就是后来著名的犹太数学家约翰·卡尔·弗里德里希·高斯，而他所采用的这种等差数列求和的方法，被称为高斯算法。（上文的故事情节与史实略有出入。）

这是数学领域中算法的一个简单示例。在数学领域里，算法是用于解决某一类问题的公式和思想。

而本书所涉及的算法，是计算机科学领域的算法，它的本质是一系列程序指令，用于解决特定的运算和逻辑问题。

从宏观上来看，数学领域的算法和计算机领域的算法有很多相通之处。

算法有简单的，也有复杂的。

简单的算法，诸如给出一组整数，找出其中最大的数。

复杂的算法，诸如在多种物品里选择装入背包的物品，使背包里的物品总价值最大，或找出从一个城市到另一个城市的最短路线。

算法有高效的，也有拙劣的。

刚才所讲的从1加到10000的故事中，高斯所用的算法显然是更加高效的算法，它利用等差数列的规律，四两拨千斤，省时省力地求出了最终结果。

而老师心中所想的算法，按部就班地一个数一个数进行累加，则是一种低效、笨拙的算法。虽然这种算法也能得到最终结果，但是其计算过程要低效得多。

在计算机领域，我们同样会遇到各种高效和拙劣的算法。衡量算法好坏的重要标准有两个：

- 时间复杂度
- 空间复杂度

具体的概念会在本章进行详细讲解。

算法的应用领域多种多样。

算法可以应用在很多不同的领域中，其应用场景更是多种多样，例如下面这些。

1.运算

有人或许会觉得，不就是数学运算嘛？这还不简单？

其实还真不简单。

例如，求出两个数的最大公约数，要做到效率的极致，的确需要动一番脑筋。

再如计算两个超大整数的和，按照正常方式来计算肯定会导致变量溢出。这又该如何求解呢？

2.查找

当你使用谷歌、百度搜索某一个关键词，或在数据库中执行某一条SQL语句时，你有没有思考过数据和信息是如何被查出来的呢？

3.排序

排序算法是实现诸多复杂程序的基石。例如，当浏览电商网站时，我们期望商品可以按价格从低到高进行排序；当浏览学生管理网站时，我们期望学生的资料可以按照学号的大小进行排序。

排序算法有很多种，它们的性能和优缺点各不相同，这里面的学问可大着呢！

4.最优决策

有些算法可以帮助我们找到最优的决策。

例如在游戏中，可以让AI角色找到走出迷宫的最佳路线，这涉及A星寻路算法。

再如，对于一个容量有限的背包来说，如何决策才可以使放入的物品总价值最高，这涉及动态规划算法。

5.面试（如果这条也算的话）

凡是已走上工作岗位的程序员，在面试过程中多多少少都经历过算法问题的考查。

为什么面试官那么喜欢考查算法呢？

考查算法问题，一方面可以检验程序员对计算机底层知识的了解，另一方面也可以衡量程序员的逻辑思维能力。

1.1.3　什么是数据结构

img算法的概念我大致明白了，那数据结构又是什么呢？

img数据结构是算法的基石。如果把算法比喻成美丽灵动的舞者，那么数据结构就是舞者脚下广阔而坚实的舞台。

数据结构，对应的英文单词是data structure，是数据的组织、管理和存储格式，其使用目的是高效地访问和修改数据。

数据结构都有哪些组成方式呢？

1.线性结构

线性结构是最简单的数据结构，包括数组、链表，以及由它们衍生出来的栈、队列、哈希表。

2.树

树是相对复杂的数据结构，其中比较有代表性的是二叉树，由它又衍生出了二叉堆之类的数据结构。

3.图

图是更为复杂的数据结构，因为在图中会呈现出多对多的关联关系。

4.其他数据结构

除上述所列的几种基本数据结构以外，还有一些其他的千奇百怪的数据结构。它们由基本数据结构变形而来，用于解决某些特定问题，如跳表、哈希链表、位图等。

有了数据结构这个舞台，算法才可以尽情舞蹈。在解决问题时，不同的算法会选用不同的数据结构。例如排序算法中的堆排序，利用的就是二叉堆这样一种数据结构；再如缓存淘汰算法LRU（Least Recently Used，最近最少使用），利用的就是特殊数据结构哈希链表。

关于算法在不同数据结构上的操作过程，在后续的章节中我们会一一进行学习。

 想不到算法和数据结构包括这么多丰富多彩的内容，大黄，我以后要好好跟你混！

 嘿嘿，我所掌握的也只是广阔的算法海洋中的一个小水洼，让我们一步一步来体验算法的无穷魅力吧！

1.2 时间复杂度

1.2.1 算法的好与坏

时间复杂度和空间复杂度究竟是什么呢？首先，让我们来想象一个场景。

某一天，小灰和大黄同时加入了同一家公司。

一天后，小灰和大黄交付了各自的代码，两人的代码实现的功能差不多。

大黄的代码运行一次要花100ms，占用内存5MB。

小灰的代码运行一次要花100s，占用内存500MB。

于是……

在上述场景中，小灰虽然也按照老板的要求实现了功能，但他的代码存在两个很严重的问题。

1.运行时间长

运行别人的代码只要100ms，而运行小灰的代码则要100s，使用者肯定是无法忍受的。

2.占用空间大

别人的代码只消耗5MB的内存，而小灰的代码却要消耗500MB的内存，这会给使用者造成很多麻烦。

由此可见，运行时间的长短和占用内存空间的大小，是衡量程序好坏的重要因素。

 可是，如果代码都还没有运行，我怎么能预知代码运行所花的时间呢？

 由于受运行环境和输入规模的影响，代码的绝对执行时间是无法预估的。但我们可以预估代码的基本操作执行次数。

1.2.2　基本操作执行次数

关于代码的基本操作执行次数，下面用生活中的4个场景来进行说明。

场景1　给小灰1个长度为10cm的面包，小灰每3分钟吃掉1cm，那么吃掉整个面包需要多久？

答案自然是3×10即30分钟。

如果面包的长度是ncm呢？

此时吃掉整个面包，需要3乘以n即3n分钟。

如果用一个函数来表达吃掉整个面包所需要的时间，可以记作T（n）=3n，n为面包的长度。

场景2　给小灰1个长度为16cm的面包，小灰每5分钟吃掉面包剩余长度的一半，即第5分钟吃掉8cm，第10分钟吃掉4cm，第15分钟吃掉2cm……那么小灰把面包吃得只剩1cm，需要多久呢？

这个问题用数学方式表达就是，数字16不断地除以2，那么除几次以后的结果等于1？这里涉及数学中的对数，即以2为底16的对数log216。（注：本书下文中，对数函数的底数全部省略。）

因此，把面包吃得只剩下1cm，需要5×log16即20分钟。

如果面包的长度是ncm呢？

此时，需要5乘以logn即5logn分钟，记作T（n）=5logn。

场景3　给小灰1个长度为10cm的面包和1个鸡腿，小灰每2分钟吃掉1个鸡腿。那么小灰吃掉整个鸡腿需要多久呢？

答案自然是2分钟。因为这里只要求吃掉鸡腿，和10cm的面包没有关系。

如果面包的长度是ncm呢？

无论面包多长，吃掉鸡腿的时间都是2分钟，记作T（n）=2。

场景4　给小灰1个长度为10cm的面包，小灰吃掉第1个1cm需要1分钟，吃掉第2个1cm需要2分钟，吃掉第3个1cm需要3分钟……每吃1cm所花的时间就比吃上一个1cm多用1分钟。那么小灰吃掉整个面包需要多久呢？

答案是从1累加到10的总和，也就是55分钟。

如果面包的长度是ncm呢？

根据高斯算法，此时吃掉整个面包需要1+2+3+…+（n-1）+n即（1+n）×n/2分钟，

也就是0.5n2+0.5n分钟，记作T（n）=0.5n2+0.5n。

 怎么除了吃还是吃啊？这还不得撑死？

上面所讲的是吃东西所花费的时间，这一思想同样适用于对程序基本操作执行次数的统计。设T（n）为程序基本操作执行次数的函数（也可以认为是程序的相对执行时间函数），n为输入规模，刚才的4个场景分别对应了程序中最常见的4种执行方式。

场景1　T（n）=3n，执行次数是线性的。

场景2　T（n）=5logn，执行次数是用对数计算的。

场景3　T（n）=2，执行次数是常量。

场景4　T（n）=0.5n2+0.5n，执行次数是用多项式计算的。

1.2.3　渐进时间复杂度

有了基本操作执行次数的函数T（n），是否就可以分析和比较代码的运行时间了呢？还是有一定困难的。

例如，算法A的执行次数是T（n）=100n，算法B的执行次数是T（n）=5n2，这两个到底谁的运行时间更长一些呢？这就要看n的取值了。

因此，为了解决时间分析的难题，有了渐进时间复杂度（asymptotic time complexity）的概念，其官方定义如下：

若存在函数f（n），使得当n趋近于无穷大时，T（n）/f（n）的极限值为不等于零的常数，则称f（n）是T（n）的同数量级函数。记作T（n）=O（f（n）），称为O（f（n）），O为算法的渐进时间复杂度，简称为时间复杂度。

因为渐进时间复杂度用大写O来表示，所以也被称为大O表示法。

img这个定义好晦涩呀，看不明白。

img直白地讲，时间复杂度就是把程序的相对执行时间函数T（n）简化为一个数量级，这个数量级可以是n、n2、n3等。

如何推导出时间复杂度呢？有如下几个原则：

- 如果运行时间是常数量级，则用常数1表示。
- 只保留时间函数中的最高阶项。
- 如果最高阶项存在，则省去最高阶项前面的系数。

让我们回头看看刚才的4个场景。

场景1

T(n)=3n，

最高阶项为3n，省去系数3，则转化的时间复杂度为：

T(n)= O(n)。

img

场景2

T(n)=5logn，

最高阶项为5logn，省去系数5，则转化的时间复杂度为：

T(n)=O(logn)。

场景3

T(n)=2，

只有常数量级，则转化的时间复杂度为：

T(n)=O(1)。

场景4

T(n)=0.5n2+0.5n，

最高阶项为0.5n2，省去系数0.5，则转化的时间复杂度为：

T(n)=O(n2)。

这4种时间复杂度究竟谁的程序执行用时更长，谁更节省时间呢？当n的取值足够大时，不难得出下面的结论：

O(1)<O(logn)<O(n)<O(n2)

在编程的世界中有各种各样的算法，除了上述4个场景，还有许多不同形式的时间复杂度，例如：

O(nlogn)、O(n3)、O(mn)、O(2n)、O(n!)

今后当我们遨游在代码的海洋中时，会陆续遇到上述时间复杂度的算法。

1.2.4　时间复杂度的巨大差异

 大黄，我还有一个问题，现在计算机硬件的性能越来越强了，我们为什么还这么重视时间复杂度呢？

 问得很好，让我们用两个算法来做一个对比，看一看高效算法和低效算法有多大的差距。

举例如下：

算法A的执行次数是T（n）=100n，时间复杂度是O（n）。

算法B的执行次数是T（n）=5n2，时间复杂度是O（n2）。

算法A运行在小灰家里的老旧电脑上，算法B运行在某台超级计算机上，超级计算机的运行速度是老旧电脑的100倍。

那么，随着输入规模n的增长，两种算法谁运行速度更快呢？

从上面的表格可以看出，当n的值很小时，算法A的运行用时要远大于算法B；当n的值在1000左右时，算法A和算法B的运行时间已经比较接近；随着n的值越来越大，甚至达到十万、百万时，算法A的优势开始显现出来，算法B的运行速度则越来越慢，差距越来越明显。

这就是不同时间复杂度带来的差距。

 要想学好算法，就必须理解时间复杂度这个重要的基础概念。有关时间复杂度的知识就介绍到这里，我们下一节再见！

1.3 空间复杂度

1.3.1 什么是空间复杂度

在运行一段程序时，我们不仅要执行各种运算指令，同时也会根据需要，存储一些临时的中间数据，以便后续指令可以更方便地继续执行。

在什么情况下需要这些中间数据呢？让我们来看看下面的例子。

给出下图所示的n个整数，其中有两个整数是重复的，要求找出这两个重复的整数。

对于这个简单的需求，可以用很多种思路来解决，其中最朴素的方法就是双重循环，具体如下：

遍历整个数列，每遍历到一个新的整数就开始回顾之前遍历过的所有整数，看看这些整数里有没有与之数值相同的。

第1步，遍历整数3，前面没有数字，所以无须回顾比较。

第2步，遍历整数1，回顾前面的数字3，没有发现重复数字。

第3步，遍历整数2，回顾前面的数字3、1，没有发现重复数字。

后续步骤类似，一直遍历到最后的整数2，发现和前面的整数2重复。

双重循环虽然可以得到最终结果，但它显然不是一个好的算法。

它的时间复杂度是多少呢？

根据上一节所学的方法，我们不难得出结论，这个算法的时间复杂度是O（n2）。

 那么，怎样才能提高算法的效率呢？

 在这种情况下，我们就有必要利用一些中间数据了。

如何利用中间数据呢？

当遍历整个数列时，每遍历一个整数，就把该整数存储起来，就像放到字典中一样。当遍历下一个整数时，不必再

慢慢向前回溯比较，而直接去“字典”中查找，看看有没有对应的整数即可。

假如已经遍历了数列的前7个整数，那么字典里存储的信息如下：

“字典”左侧的Key代表整数的值，“字典”右侧的Value代表该整数出现的次数（也可以只记录Key）。

接下来，当遍历到最后一个整数2时，从“字典”中可以轻松找到2曾经出现过，问题也就迎刃而解了。

由于读写“字典”本身的时间复杂度是O（1），所以整个算法的时间复杂度是O（n），和最初的双重循环相比，运行效率大大提高了。

而这个所谓的“字典”，是一种特殊的数据结构，叫作哈希表，也称为散列表。这个数据结构需要开辟一定的内存空间来存储有用的数据信息。

但是，内存空间是有限的，在时间复杂度相同的情况下，算法占用的内存空间自然是越小越好。如何描述一个算法占用的内存空间的大小呢？这就用到了算法的另一个重要指标——空间复杂度（space complexity）。

和时间复杂度类似，空间复杂度是对一个算法在运行过程中临时占用存储空间大小的量度，它同样使用了大O表示法。

程序占用空间大小的计算公式记作S（n）=O（f（n））， 其中n为问题的规模，f（n）为算法所占存储空间的函数。

1.3.2　空间复杂度的计算

 基本的概念已经明白了，那么，我们如何来计算空间复杂度呢？

img具体情况要具体分析。和时间复杂度类似，空间复杂度也有几种不同的增长趋势。

常见的空间复杂度有下面几种情形。

1.常量空间

当算法的存储空间大小固定，和输入规模没有直接的关系时，空间复杂度记作O（1）。例如下面这段程序：

2.线性空间

当算法分配的空间是一个线性的集合（如列表），并且集合大小和输入规模n成正比时，空间复杂度记作O（n）。例如下面这段程序：

3.二维空间

当算法分配的空间是一个二维列表集合，并且集合的长度和宽度都与输入规模n成正比时，空间复杂度记作O（n2）。例如下面这段程序：

4.递归空间

递归是一个比较特殊的场景。虽然递归代码中并没有显式地声明变量或集合，但是计算机在执行程序时，会专门分配一块内存，用来存储“函数调用栈”。

“函数调用栈”包括进栈和出栈两个行为。

当进入一个新函数时，执行入栈操作，把调用的函数和参数信息压入栈中。

当函数返回时，执行出栈操作，把调用的函数和参数信息从栈中弹出。

下面这段程序是一个标准的递归程序：

假如初始传入的参数值n=5，那么函数fun4（参数n=5）的调用信息先入栈。

接下来递归调用相同的方法，函数fun4（参数n=4）的调用信息入栈。

以此类推，递归越来越深，入栈的元素就越来越多。

当n=1时，达到递归结束条件，函数出栈。

最终，“函数调用栈”的全部元素会一一出栈。

由上面“函数调用栈”的出入栈过程可以看出，执行递归操作所需要的内存空间和递归的深度成正比。纯粹的递归操作的空间复杂度也是线性的，如果递归的深度是n，那么空间复杂度就是O（n）。

1.3.3 时间与空间的取舍

人们之所以花大力气去评估算法的时间复杂度和空间复杂度，其根本原因是计算机的运算速度和空间资源是有限的。

就如一个大财主，基本不必为日常花销伤脑筋；而一个没多少积蓄的普通人，则不得不为日常花销精打细算。

对于计算机系统来说也是如此。虽然目前计算机的CPU处理速度不断飙升，内存和硬盘空间也越来越大，但是面对庞大而复杂的数据和业务，我们仍然要精打细算，选择最有效的使用方式。

但是，正所谓鱼和熊掌不可兼得。很多时候，我们不得不在时间复杂度和空间复杂度之间进行取舍。

在1.3.1节寻找重复整数的例子中，双重循环的时间复杂度是O（n2），空间复杂度是O（1），这属于牺牲时间来换取空间的情况。

相反，字典法的空间复杂度是O（n），时间复杂度是O（n），这属于牺牲空间来换取时间的情况。

在绝大多数时候，时间复杂度更为重要一些，我们宁可多分配一些内存空间，也要提升程序的执行速度。

此外，说起空间复杂度就离不开数据结构。在本章中，我们提及了哈希表、列表、二维列表这些集合。如果大家对这些数据结构不是很了解，可以仔细看看本书第2章关于基本数据结构的内容，里面有详细的介绍。

img关于空间复杂度的知识，我们就介绍到这里。时间复杂度和空间复杂度都是学好算法的重要前提，一定要牢牢掌握哦！

1.4 小结

- 什么是算法

在计算机领域里，算法是一系列程序指令，用于处理特定的运算和逻辑问题。

衡量算法优劣的主要标准是时间复杂度和空间复杂度。

- 什么是数据结构

数据结构是数据的组织、管理和存储格式，其使用目的是高效地访问和修改数据。

数据结构包含数组、链表这样的线性数据结构，也包含树、图这样的复杂数据结构。

- 什么是时间复杂度

时间复杂度是对一个算法运行时间长短的量度，用大O表示，记作T（n）=O（f（n））。

常见的时间复杂度按照从低到高的顺序，包括O（1）、O（logn）、O（n）、O（nlogn）、O（n2）等。

- 什么是空间复杂度

空间复杂度是对一个算法在运行过程中临时占用存储空间大小的量度，用大O表示，记作S（n）=O（f（n））。

常见的空间复杂度按照从低到高的顺序，包括O（1）、O（n）、O（n2）等。其中递归算法的空间复杂度和递归深度成正比。

第2章　数据结构基础

2.1　什么是数组

2.1.1　初识数组

这些特点是如何体现的呢？

参加过军训的读者，一定都记得这样的场景。

在军队里，每一个士兵都有自己固定的位置、固定的编号。众多士兵紧密团结在一起，高效地执行着一个个命令。

 大黄，咱们为什么要说这么多关于军队的事情呢？

 因为有一个数据结构就像军队一样整齐、有序，这个数据结构叫作数组。

什么是数组？

数组对应的英文是array，是有限个相同类型的变量所组成的有序集合，数组中的每一个变量称为元素。数组是最简单、最常用的数据结构。

数组的存储形式如下图所示：

正如军队里的士兵存在编号一样，数组中的每一个元素也都有着自己的下标，只不过这个下标从0开始，一直到数组长度-1。

数组的另一个特点是，在内存中顺序存储，因此可以很好地实现逻辑上的顺序表。

数组在内存中的顺序存储，具体是什么样子呢？

内存是由一个个连续的内存单元组成的，每一个内存单元都有自己的地址。在这些内存单元中，有些被其他数据占用了，有些是空闲的。

数组中的每一个元素，都存储在小小的内存单元中，并且元素之间紧密排列，既不能打乱元素的存储顺序，也不能跳过某个存储单元进行存储。

在上图中，橙色的格子代表空闲的存储单元，灰色的格子代表已占用的存储单元，而红色的连续格子代表数组在内存中的位置。对于不同的编程语言，数组在内存中的分配方式并不完全相同，本图只是一个简单的示意图。

在Python语言中，并没有直接使用数组这个概念，而是使用列表（list）和元组（tuple）这两种集合，它们本质上都是对数组的封装。其中，列表是一个动态可扩展的数组，支持任意地添加、删除、修改元素；而元组是一个不可变集合，一旦创建就不再支持修改。

元组并不是本书的重点讨论对象，后文中我们提到的数组概念，对应的是Python语言中的列表。

 那么，我们怎样来使用一个数组呢？

 数据结构的操作无非是增、删、改、查4种情况，下面让我们来看一看数组的基本操作。

2.1.2　数组的基本操作

1.读取元素

对于数组来说，读取元素是最简单的操作。由于数组在内存中顺序存储，所以只要给出一个数组下标，就可以快速读取到对应的数组元素。

首先让我们来创建一个数组，也就是Python中的列表，代码非常简单，使用方括号的形式即可：

现在我们已经拥有了一个名为my_list的数组，要读取数组下标为3的元素，就写作my_list[3]；要读取数组下标为5的元素，就写作my_list[5]。需要注意的是，输入的下标必须在数组的长度范围之内，否则会出现数组越界的问题。

像这种根据下标读取元素的方式叫作随机读取。

代码示例如下：

2.更新元素

要把数组中某一个元素的值替换为一个新值，也是非常简单的操作。直接利用数组下标，就可以把新值赋给该元素。

简单的代码示例如下：

 小灰，咱们刚刚讲过时间复杂度的概念，你说说数组读取元素和更新元素的时间复杂度分别是多少？

 嘿嘿，这难不倒我。数组读取元素和更新元素的时间复杂度都是O（1）。

3.插入元素

插入数组元素的操作存在3种情况：

- 尾部插入
- 中间插入
- 超范围插入

尾部插入，是最简单的情况，直接把插入的元素放在数组尾部的空闲位置即可，等同于更新元素的操作。

中间插入，稍微复杂一些。由于数组的每一个元素都有其固定下标，所以不得不首先把插入位置及后面的元素向后移动，腾出地方，再把要插入的元素放到对应的数组位置上。

上面所描述的插入过程，在Python底层已经为我们做了很好的实现，调用起来非常简单：

但是，为了更好地理解数组的工作方式，让我们来自己实现一段插入操作的代码：

代码中的成员变量size是数组中实际元素的数量。如果插入元素在数组尾部，传入的下标参数index等于size；如果插入元素在数组中间或头部，则index小于size。

如果传入的下标参数index大于size或小于0，则认为是非法输入，会直接抛出异常。

 可是，如果往数组中不断插入新的元素，元素数量超过了数组的最大长度，数组岂不是要“撑爆”了？

 问得很好，这就是接下来要讲的情况——超范围插入。

超范围插入，又是什么意思呢？

假如现在有一个长度为6的数组，已经装满了元素，这时还想插入一个新元素。

这就涉及数组的扩容了。可是数组的长度在创建时就已经确定了，无法像孙悟空的金箍棒那样随意变长或变短。这该如何是好呢？

此时可以创建一个新数组，长度是旧数组的2倍，再把旧数组中的元素统统复制过去，这样就实现了数组的扩容。

如此一来，我们的插入元素方法也需要改写了，改写后的代码如下：

4.删除元素

数组的删除操作和插入操作的过程相反，如果删除的元素位于数组中间，其后的元素都需要向前挪动1位。

同样的，对于删除操作，虽然Python底层已经做了很好的实现，但我们仍然要尝试自己来实现一下。由于不涉及扩容问题，所以删除操作的代码实现比插入操作要简单：

 小灰，我再考考你，数组的插入和删除操作，时间复杂度分别是多少？

 先说说插入操作，数组扩容的时间复杂度是O（n），插入并移动元素的时间复杂度也是O（n），综合起来插入操作的时间复杂度是O（n）。至于删除操作，只涉及元素的移动，时间复杂度也是O（n）。

 说的没错。对于删除操作，其实还存在一种取巧的方式，前提是数组元素没有顺序要求。

例如下图所示，需要删除的是数组中的元素2，可以把最后一个元素复制到元素2所在的位置，然后再删除最后一个元素：

这样一来，无须进行大量的元素移动，时间复杂度降低为O（1）。当然，这种方式只作为参考，并不是删除元素时主流的操作方式。

2.1.3 数组的优势和劣势

 数组的基本知识我懂了，那么，使用数组这种数据结构有什么优势和劣势呢？

 数组拥有非常高效的随机访问能力，只要给出下标，就可以用常量时间找到对应元素。有一种高效查找元素的算法叫作二分查找，就是利用了数组的这个优势。

 至于数组的劣势，体现在插入和删除元素方面。由于数组元素连续紧密地存储在内存中，插入、删除元素都会导致大量元素被迫移动，影响效率。

 总的来说，数组所适合的是读操作多、写操作少的场景，下一节我们要讲解的链表则恰恰相反。好了，让我们下一节再会！

2.2 什么是链表

2.2.1 “正规军”和“地下党”

地下党是一些什么样的人物呢？

在影视作品中，我们可能都见到过地下工作者的经典话语：

“上级的姓名、住址，我知道，下级的姓名、住址，我也知道，但是这些都是秘密，不能告诉你们！”

地下党借助这种单线联络的方式，灵活隐秘地传递着各种重要信息。

在计算机科学领域里，有一种数据结构也恰恰具备这样的特征，这种数据结构就是链表。

链表是什么样子的？为什么说它像地下党呢？

让我们来看一看单向链表的结构。

链表（linked list）是一种在物理上非连续、非顺序的数据结构，由若干节点（node）所组成。

单向链表的每一个节点又包含两部分，一部分是存放数据的变量data，另一部分是指向下一个节点的指针next。

链表的第1个节点被称为头节点，最后1个节点被称为尾节点，尾节点的next指针指向空。

与数组按照下标来随机寻找元素不同，对于链表的其中一个节点A，我们只能根据节点A的next指针来找到该节点的下一个节点B，再根据节点B的next指针找到下一个节点C……

 那么，通过链表的一个节点，如何能快速找到它的前一个节点呢？

 要想让每个节点都能回溯到它的前置节点，我们可以使用双向链表。

什么是双向链表？

双向链表比单向链表稍微复杂一些，它的每一个节点除了拥有data和next指针，还拥有指向前置节点的prev指针。

接下来我们看一看链表的存储方式。

如果说数组在内存中的存储方式是顺序存储，那么链表在内存中的存储方式则是随机存储。

什么叫随机存储呢？

上一节我们讲解了数组的内存分配方式，数组在内存中占用了连续完整的存储空间。而链表则采用了见缝插针的方式，链表的每一个节点分布在内存中的不同位置，依靠

next指针进行关联。这样可以灵活有效地利用零散的碎片空间。

让我们看一看下面两张图，对比一下数组和链表在内存中分配方式的不同：

数组的内存分配方式

链表的内存分配方式

图中的箭头代表链表节点的next指针。

 那么，我们怎样来使用一个链表呢？

 上一节刚刚讲过数组的增、删、改、查，这一次来看看链表的相关操作。

2.2.2　链表的基本操作

1.查找节点

在查找元素时，链表不像数组那样可以通过下标快速进行定位，只能从头节点开始向后一个一个节点逐一查找。

例如给出一个链表，需要查找从头节点开始的第3个节点：

第1步，将查找的指针定位到头节点。

第2步，根据头节点的next指针，定位到第2个节点。

第3步，根据第2个节点的next指针，定位到第3个节点，查找完毕。

 小灰，你说说查找链表节点的时间复杂度是多少？

img链表中的数据只能按顺序进行访问，最坏的时间复杂度是O（n）。

2.更新节点

如果不考虑查找节点的过程，链表的更新过程会像数组那样简单，直接把旧数据替换成新数据即可。

3.插入节点

与数组类似，在链表中插入节点时，同样分为3种情况：

- 尾部插入
- 头部插入
- 中间插入

尾部插入，是最简单的情况，把最后一个节点的next指针指向新插入的节点即可：

头部插入，可以分成两个步骤：

第1步，把新节点的next指针指向原先的头节点。

第2步，把新节点变为链表的头节点。

中间插入，同样分为两个步骤：

第1步，新节点的next指针指向插入位置的节点。

第2步，插入位置前置节点的next指针，指向新节点。

只要内存空间允许，能够插入链表的元素是无穷无尽的，不需要像数组那样考虑扩容的问题。

4.删除元素

链表的删除操作同样分为3种情况：

- 尾部删除
- 头部删除
- 中间删除

尾部删除，是最简单的情况，把倒数第2个节点的next指针指向空即可：

头部删除，也很简单，把链表的头节点设为原先头节点的next指针所指向的节点即可：

中间删除，同样很简单，把要删除节点的前置节点的next指针，指向要删除元素的下一个节点即可：

这里需要注意的是，许多高级语言，如Java、Python，拥有自动化的垃圾回收机制，所以我们不用刻意去释放被删除的节点，只要没有外部引用指向它们，被删除的节点会被自动回收。

 小灰，我再考考你，链表的插入和删除操作，时间复杂度分别是多少？

 如果不考虑插入、删除操作之前查找元素的过程，只考虑纯粹的插入和删除操作，时间复杂度都是O（1）。

 很好，接下来看一看实现链表的完整代码。

以上是对单链表相关操作的代码实现。为了尾部插入的方便，代码中额外增加了指向链表尾节点的指针last。

2.2.3 数组VS链表

img链表的基本知识我懂了。数组和链表都属于线性的数据结构，用哪一个更好呢？

img数据结构没有绝对的好与坏，数组和链表各有千秋。下面我总结了数组和链表相关操作的性能，我们来对比一下。

img从表格可以看出，数组的优势在于能够快速定位元素，对于读操作多、写操作少的场景来说，用数组更合适一些。

img相反地，链表的优势在于能够灵活地进行插入和删除操作，如果需要频繁插入、删除元素，用链表更合适一些。

img关于链表的知识我们就介绍到这里，咱们下一节再见！

2.3 栈和队列

2.3.1 物理结构和逻辑结构

什么是数据存储的物理结构呢？

如果把数据结构比作活生生的人，那么物理结构就是人的血肉和骨骼，看得见，摸得着，实实在在。例如，我们刚刚学过的数组和链表，都是内存中实实在在的存储结构。

而在物质的人体之上，还存在着人的思想和精神，它们看不见、摸不着。看过电影《阿凡达》吗？男主角的思想意识从一个瘦弱残疾的人类身上被移植到一个高大威猛的蓝皮肤外星人身上，虽然承载思想意识的肉身改变了，但是人格却是唯一的。

如果把物质层面的人体比作数据存储的物理结构，那么精神层面的人格则是数据存储的逻辑结构。逻辑结构是抽象

的概念，它依赖于物理结构而存在。

下面我们来讲解两个常用的数据结构：栈和队列。这两者都属于逻辑结构，它们的物理实现既可以利用数组，也可以利用链表来完成。

在后面的章节中，我们会学习到二叉树，这也是一种逻辑结构。同样地，二叉树也可以依托于物理上的数组或链表来实现。

2.3.2 什么是栈

要弄明白什么是栈，我们需要先举一个生活中的例子。

假如有一个又细又长的圆筒，圆筒一端封闭，另一端开口。往圆筒里放入乒乓球，先放入的靠近圆筒底部，后放入的靠近圆筒入口。

那么，要想取出这些乒乓球，则只能按照和放入顺序相反的顺序来取，先取出后放入的，再取出先放入的，而不可能把最里面最先放入的乒乓球优先取出。

栈（stack）是一种线性数据结构，它就像一个上图所示的放乒乓球的圆筒容器，栈中的元素只能先入后出（First In Last Out，简称FILO）。最早进入的元素存放的位置叫作栈底（bottom），最后进入的元素存放的位置叫作栈顶（top）。

栈这种数据结构既可以用数组来实现，也可以用链表来实现。

栈的数组实现如下：

栈的链表实现如下：

 那么，栈可以进行哪些操作呢？

 栈的最基本操作是入栈和出栈，下面让我们来看一看。

2.3.3 栈的基本操作

1.入栈

入栈操作（push）就是把新元素放入栈中，只允许从栈顶一侧放入元素，新元素的位置将会成为新的栈顶。

这里我们以数组实现为例：

2.出栈

出栈操作（pop）就是把元素从栈中弹出，只有栈顶元素才允许出栈，出栈元素的前一个元素将会成为新的栈顶。

这里我们以数组实现为例：

在Python语言中，列表很好地实现了栈的功能，append方法相当于入栈，pop方法相当于出栈。由于栈操作的代码实现相对简单，这里就不展示代码了，有兴趣的读者可以自己写写看。

 小灰，你说说，入栈和出栈操作，时间复杂度分别是多少？

 入栈和出栈只会影响最后一个元素，不涉及其他元素的整体移动，所以无论是以数组还是以链表实现，入栈、出栈的时间复杂度都是O（1）。

2.3.4 什么是队列

要弄明白什么是队列，我们同样可以用一个生活中的例子来说明。

假如公路上有一条单行隧道，所有通过隧道的车辆只允许从隧道入口驶入，从隧道出口驶出，不允许逆行。

因此，要想让车辆驶出隧道，只能按照它们驶入隧道的顺序，先驶入的车辆先驶出，后驶入的车辆后驶出，任何车辆都无法跳过它前面的车辆提前驶出。

队列（queue）是一种线性数据结构，它的特征和行驶车辆的单行隧道很相似。不同于栈的先入后出，队列中的元素只能先入先出（First In First Out，简称FIFO）。队列的出口端叫作队头（front），队列的入口端叫作队尾（rear）。

与栈类似，队列这种数据结构既可以用数组来实现，也可以用链表来实现。

用数组实现时，为了入队操作的方便，把队尾位置规定为最后入队元素的下一个位置。

队列的数组实现如下：

队列的链表实现如下：

 那么，队列可以进行哪些操作呢？

 和栈操作相对应，队列的最基本操作是入队和出队。

2.3.5　队列的基本操作

对于链表实现方式，队列的入队、出队操作和栈是大同小异的。但对于数组实现方式来说，队列的入队和出队操作有了一些有趣的变化。怎么有趣呢？我们后面会看到。

1.入队

入队（enqueue）就是把新元素放入队列中，只允许在队尾的位置放入元素，新元素的下一个位置将会成为新的队尾。

2.出队

出队操作（dequeue）就是把元素移出队列，只允许在队头一侧移出元素，出队元素的后一个元素将会成为新的队头。

 如果像这样不断出队，队头左边的空间失去作用，那队列的容量岂不是越来越小了？例如像下面这样。

 问得很好，这正是我后面要讲的。用数组实现的队列可以采用循环队列的方式来维持队列容量的恒定。

循环队列是什么意思呢？让我们看看下面的例子。

假设一个队列经过反复的入队和出队操作，还剩下2个元素，在“物理”上分布于数组的末尾位置。这时又有一个新元素将要入队。

在数组不做扩容的前提下，如何让新元素入队并确定新的队尾位置呢？我们可以利用已出队元素留下的空间，让队尾指针重新指回数组的首位。

这样一来，整个队列的元素就“循环”起来了。在物理存储上，队尾的位置也可以在队头之前。当再有元素入队时，将其放入数组的首位，队尾指针继续后移即可。

一直到（队尾下标+1）%数组长度=队头下标时，代表此队列真的已经满了。需要注意的是，队尾指针指向的位置

永远空出1位，所以队列最大容量比数组长度小1。

以上就是循环队列的基本原理。在Python语言中提供了多种队列工具，比如collections.deque，queue.Queue等。但是为了加深大家对队列原理的理解，我们仍然自己尝试实现一个队列：

 循环队列不但充分利用了数组的空间，还避免了数组元素整体移动的麻烦，还真是有点意思呢！至于入队和出队的时间复杂度，也同样是O（1）吧？

 说得完全正确！下面我们来看一看栈和队列都可以应用在哪些地方。

2.3.6　栈和队列的应用

1.栈的应用

栈的输出顺序和输入顺序相反，所以栈通常用于对“历史”的回溯，也就是逆流而上追溯“历史”。

例如实现递归的逻辑，就可以用栈来代替，因为栈可以回溯方法的调用链。

栈还有一个著名的应用场景是面包屑导航，使用户在浏览页面时可以轻松地回溯到上一级或更上一级页面。

2.队列的应用

队列的输出顺序和输入顺序相同，所以队列通常用于对“历史”的回放，也就是按照“历史”顺序，把“历史”重演一遍。

例如在多线程中，争夺公平锁的等待队列，就是按照访问顺序来决定线程在队列中的次序的。

再如网络爬虫实现网站抓取时，也是把待抓取的网站URL存入队列中，再按照存入队列的顺序来依次抓取和解析的。

3.双端队列

 那么，有没有办法把栈和队列的特点结合起来，既可以先入先出，也可以先入后出呢？

 还真有，这种数据结构叫作双端队列（deque）。

双端队列这种数据结构，可以说综合了栈和队列的优点，对双端队列来说，从队头一端可以入队或出队，从队尾一端也可以入队或出队。

有关双端队列的细节，感兴趣的读者可以查阅资料做更多的了解。

4.优先队列

还有一种队列，它遵循的不是先入先出，而是谁的优先级最高，谁先出队。

 这种队列叫作优先队列。

优先队列已经不属于线性数据结构的范畴了，它是基于二叉堆来实现的。关于优先队列的原理和使用情况，我们会在下一章进行详细介绍。

 好了，关于栈和队列的知识我们就介绍到这里，下一节再见！

2.4　神奇的哈希表

2.4.1 为什么需要哈希表

说起学习英语，小灰上学时可没有那么丰富的学习资源和工具。当时有一款很流行的电子词典，小伙伴们遇到不会的单词，只要输入小小的电子词典里，就可以查出它的中文含义。

当时的英语老师强烈反对使用这样的工具，因为电子词典查出来的中文资料太有限，而传统的纸质词典可以查到单词的多种含义、词性、例句等。

但是，同学们还是倾向于使用电子词典。因为电子词典实在太方便了，只要输入要查的单词，瞬间就可以得到结果，而不需要像纸质词典那样烦琐地进行人工查找。

在我们的程序世界里，往往也需要在内存中存放这样一个“词典”，方便我们进行高效的查询和统计。

例如开发一个学生管理系统，需要有通过输入学号快速查出对应学生的姓名的功能。这里不必每次都去查询数据库，而可以在内存中建立一个缓存表，这样做可以提高查询效率。

再如我们想统计一本英文书里某些单词出现的频率，就需要遍历整本书的内容，把这些单词出现的次数记录在内存中。

因为这些需求，一个重要的数据结构诞生了，这个数据结构叫作哈希表。

哈希表（hash table）也叫作散列表，这种数据结构提供了键（Key）和值（Value）的映射关系。只要给出一个

Key，就可以高效查找到它所匹配的Value，时间复杂度接近于O（1）。

 那么，哈希表是如何根据Key来快速找到它所匹配的Value的呢？

 这就是我下面要讲的哈希表的基本原理。

2.4.2 哈希函数

 小灰，在咱们之前学过的几个数据结构中，谁的查询效率最高？

 当然是数组喽，数组可以根据下标进行元素的随机访问。

 说得没错，哈希表在本质上也是一个数组。

 可是数组只能根据下标，像a[0]、a[1]、a[2]、a[3]、a[4]这样来访问，而哈希表的Key则是以字符串类型为主的。

 例如以学生的学号作为Key，输入002123，查询到李四；或者以单词为Key，输入by，查询到数字46……

 所以我们需要一个“中转站”，通过某种方式，把Key和数组下标进行转换。这个中转站就叫作哈希函数。

这个所谓的哈希函数是怎么实现的呢？

在不同的语言中，哈希函数的实现方式是不一样的，在Python语言中，哈希表对应的集合叫作字典（dict）。下面我们以Python为例，来看看哈希函数是如何实现的。

在Python及大多数面向对象的语言中，每一个对象都有属于自己的hash值，这个hash值是区分不同对象的重要标识。无论对象自身的类型是什么，它们的hash值都是一个整型变量。

既然都是整型变量，想要转化成数组的下标也就不难实现了。最简单的转化方式是什么呢？是按照数组长度进行取模运算。

index=hash (key)% size

实际上，Python中的哈希函数并没有直接采用取模运算，而是利用了位运算的方式来优化性能。不过在这里我们可以姑且把它简单理解成取模操作。

通过哈希函数，可以把字符串或其他类型的Key，转化成数组的下标index。

如给出一个长度为8的数组，则当

key=001121时，

index= hash (“001121”)% size=1420036703%8=7

而当key=“this”时，

index= hash (“this”)% size=3559070%8=6

2.4.3 哈希表的读写操作

有了哈希函数，就可以在哈希表中进行读写操作了。

1.写操作（put）

写操作就是在哈希表中插入新的键值对（也被称为Entry）。

如调用 dict[“002931”]=“王五”，意思是插入一组Key为002931、Value为王五的键值对。

具体该怎么做呢？

第1步，通过哈希函数，把Key转化成数组下标5。

第2步，如果数组下标5对应的位置没有元素，就把这个Entry填充到数组下标5的位置。

但是，由于数组的长度是有限的，当插入的Entry越来越多时，不同的Key通过哈希函数获得的下标有可能是相同的。例如，002936这个Key对应的数组下标是2；002947这个 Key对应的数组下标也是2。

这种情况，就叫作哈希冲突。

 哎呀，哈希函数“撞衫”了，这该怎么办呢？

 哈希冲突是无法避免的，既然不能避免，我们就要想办法来解决。解决哈希冲突的方法主要有两种，一种是开放寻址法，一种是链表法。

开放寻址法的原理很简单，当一个Key通过哈希函数获得对应的数组下标已被占用时，我们可以“另谋高就”，寻找下一个空当位置。

以上面的情况为例，Entry6通过哈希函数得到下标2，该下标在数组中已经有了其他元素，那么就向后移动1位，看看数组下标3的位置是否有空。

很不巧，下标3也已经被占用，那么就再向后移动1位，看看数组下标4的位置是否有空。

幸运的是，数组下标4的位置还没有被占用，因此把Entry6存入数组下标4的位置。

这就是开放寻址法的基本思路。当然，在遇到哈希冲突时，寻址方式有很多种，并不一定只是简单地寻找当前元素的后一个元素，这里只是举一个简单的示例而已。

那么链表法又是怎样的呢？

哈希表数组的每一个元素不仅是一个Entry对象，还是一个链表的头节点。每一个 Entry对象通过next指针指向它的下

一个Entry节点。当新来的Entry映射到与之冲突的数组位置时，只需要插入对应的链表中即可。

2.读操作（get）

讲完了写操作，我们再来讲一讲读操作。读操作就是通过给定的Key，在哈希表中查找对应的Value。

例如调用 dict[“002936”]，意思是查找Key为002936的Entry在哈希表中所对应的值。

具体该怎么做呢？下面以链表法为例来讲一下。

第1步，通过哈希函数，把Key转化成数组下标2。

第2步，找到数组下标2所对应的元素，如果这个元素的Key是002936，那么就找到了；如果这个Key不是002936也没关系，由于数组的每个元素都与一个链表对应，我们可以顺着链表慢慢往下找，看看能否找到与Key相匹配的节点。

在上图中，首先查到的节点Entry6的Key是002947，和待查找的Key002936不符。接着定位到链表的下一个节点Entry1，发现Entry1的Key002936正是我们要寻找的，所以返回 Entry1的Value即可。

在众多编程语言中，有些语言的哈希表采用了链表法，最具代表性的就是Java中的HashMap；有些编程语言采用的是开放寻址法，比如Python中的dict。

以上就是哈希表各种基本操作的原理。有兴趣的读者可以看一下Python官方解释器（CPython）中，关于PyDictObject的C语言源码实现。

 我基本明白了，哈希表还真是一个神奇的数据结构！

img哈希表可以说是数组和链表的结合，它在算法中的应用很普遍，是一种非常重要的数据结构，大家一定要认真掌握哦。

img关于哈希表就讲到这里，咱们下一章再见。

2.5 小结

- 什么是数组

数组是由有限个相同类型的变量所组成的有序集合，它的物理存储方式是顺序存储，访问方式是随机访问。利用下标查找数组元素的时间复杂度是O（1），中间插入、删除数组元素的时间复杂度是O（n）。

- 什么是链表

链表是一种链式数据结构，由若干节点组成，每个节点包含指向下一节点的指针。链表的物理存储方式是随机存储，访问方式是顺序访问。查找链表节点的时间复杂度是O（n），中间插入、删除节点的时间复杂度是O（1）。

- 什么是栈

栈是一种线性逻辑结构，可以用数组实现，也可以用链表实现。栈包含入栈和出栈操作，遵循先入后出的原则（FILO）。

- 什么是队列

队列也是一种线性逻辑结构，可以用数组实现，也可以用链表实现。队列包含入队和出队操作，遵循先入先出的原则（FIFO）。

- 什么是哈希表

哈希表也叫散列表，是存储Key-Value映射的集合。对于某一个Key，哈希表可以在接近O（1）的时间内进行读写操作。哈希表通过哈希函数实现Key和数组下标的转换，通过开放寻址法和链表法来解决哈希冲突。

第3章　树

3.1　树和二叉树

3.1.1　什么是树

小灰的“家谱”是这样子的：

 所以说，有许多逻辑关系并不是简单的线性关系，在实际场景中，常常存在一对多，甚至多对多的情况。

 其中树和图就是典型的非线性数据结构，我们首先讲一讲树的知识。

什么是树呢？在现实生活中有很多体现树的逻辑的例子。

例如前面提到的小灰的“家谱”，就是一个“树”。

再如企业里的职级关系，也是一个“树”。

除了人与人之间的关系之外，许多抽象的东西也可以成为一个“树”，如一本书的目录。

以上这些例子有什么共同点呢？为什么可以称它们为“树”呢？

因为它们都像自然界中的树一样，从同一个“根”衍生出许多“枝干”，再从每一个“枝干”衍生出许多更小的“枝干”，最后衍生出更多的“叶子”。

在数据结构中，树的定义如下：

树（tree）是n（n≥0）个节点的有限集。当n=0时，称为空树。在任意一个非空树中，有如下特点。

1.有且仅有一个特定的称为根的节点。

2.当n>1时，其余节点可分为m（m>0）个互不相交的有限集，每一个集合本身又是一个树，并称为根的子树。

下面这张图，就是一个标准的树结构。

在上图中，节点1是根节点（root）；节点5、6、7、8、9是树的末端，没有“孩子”，被称为叶子节点（leaf）。图中的虚线部分，是根节点1的其中一个子树。

同时，树的结构从根节点到叶子节点，分为不同的层级。从一个节点的角度来看，它的上下级和同级节点关系如下：

在上图中，节点4的上一级节点，是节点4的父节点（parent）；从节点4衍生出来的节点，是节点4的孩子节点（child）；和节点4同级，由同一个父节点衍生出来的节点，是节点4的兄弟节点（sibling）。

树的最大层级数，称为树的高度或深度。显然，上图这个树的高度是4。

 哎呀，这么多的概念还真是不好记。

 这些都是树的基本术语，多看几次就记住啦。下面我们来介绍一种典型的树——二叉树。

3.1.2　什么是二叉树

二叉树（binary tree）是树的一种特殊形式。二叉，顾名思义，这种树的每个节点最多有2个孩子节点。注意，这里是最多有2个，也可能只有1个，或者没有孩子节点。

二叉树的结构如下图所示：

二叉树节点的两个孩子节点，一个被称为左孩子（left child），一个被称为右孩子（right child）。这两个孩子节点的顺序是固定的，就像人的左手就是左手，右手就是右手，不能够颠倒或混淆。

此外，二叉树还有两种特殊形式，一个叫作满二叉树，另一个叫作完全二叉树。

什么是满二叉树呢？

一个二叉树的所有非叶子节点都存在左孩子和右孩子，并且所有叶子节点都在同一层级上，那么这个树就是满二叉树。

简单点说，满二叉树的每一个分支都是满的。

什么又是完全二叉树呢？完全二叉树的定义很有意思。

对一个有n个节点的二叉树，按层级顺序编号，则所有节点的编号为从1到n。如果这个树所有节点和同样深度的满二叉树的编号为从1到n的节点位置相同，则这个二叉树为完全二叉树。

这个定义还真绕，看看下图就很容易理解了。

在上图中，二叉树编号从1到12的12个节点，和前面满二叉树编号从1到12的节点位置完全对应。因此这个树是完全二叉树。

完全二叉树的条件没有满二叉树那么苛刻：满二叉树要求所有分支都是满的；而完全二叉树只需保证最后一个节点之前的节点都齐全即可。

 那么，二叉树在内存中是怎样存储的呢？

img上一章咱们讲过，数据结构可以划分为物理结构和逻辑结构。二叉树属于逻辑结构，它可以通过多种物理结构来表达。

二叉树可以用哪些物理存储结构来表达呢？

1.链式存储结构。

2.数组。

让我们分别看看二叉树如何使用这两种结构进行存储吧。

首先来看一看链式存储结构。

链式存储是二叉树最直观的存储方式。

上一章讲过链表，链表是一对一的存储方式，每一个链表节点拥有data变量和一个指向下一节点的next指针。

而二叉树稍微复杂一些，一个节点最多可以指向左右两个孩子节点，所以二叉树的每一个节点包含3部分。

- 存储数据的data变量
- 指向左孩子的left指针
- 指向右孩子的right指针

再来看看用数组是如何存储的。

使用数组存储时，会按照层级顺序把二叉树的节点放到数组中对应的位置上。如果某一个节点的左孩子或右孩子空缺，则数组的相应位置也空出来。

为什么这样设计呢？因为这样可以更方便地在数组中定位二叉树的孩子节点和父节点。

假设一个父节点的下标是parent，那么它的左孩子节点的下标就是2×parent +1；右孩子节点的下标就是2×parent +2。

反过来，假设一个左孩子节点的下标是leftChild，那么它的父节点下标就是（leftChild-1）/2。

假如节点4在数组中的下标是3，节点4是节点2的左孩子，节点2的下标可以直接通过计算得出：

节点2的下标= （3-1）/2=1

显然，对于一个稀疏的二叉树来说，用数组表示法是非常浪费空间的。

什么样的二叉树最适合用数组表示呢？

我们后面即将学到的二叉堆，一种特殊的完全二叉树，就是用数组来存储的。

3.1.3 二叉树的应用

 咱们讲了这么多理论，二叉树究竟有什么用处呢？

 二叉树的用处有很多，让我们来具体看一看。

二叉树包含许多特殊的形式，每一种形式都有自己的作用，但是其最主要的应用还在于进行查找操作和维持相对顺序这两个方面。

1.查找

二叉树的树形结构使它很适合扮演索引的角色。

这里我们介绍一种特殊的二叉树：二叉查找树（binary search tree）。仅看名字就可以知道，这种二叉树的主要作用就是进行查找操作。

二叉查找树在二叉树的基础上增加了以下几个条件：

- 如果左子树不为空，则左子树上所有节点的值均小于根节点的值。
- 如果右子树不为空，则右子树上所有节点的值均大于根节点的值。
- 左子树、右子树也都是二叉查找树。

下图就是一个标准的二叉查找树。

二叉查找树的这些条件有什么用呢？当然是为了查找方便。

例如查找值为4的节点，步骤如下。

1.访问根节点6，发现4<6。

2.访问节点6的左孩子节点3，发现4>3。

3.访问节点3的右孩子节点4，发现4=4，这正是要查找的节点。

对于一个节点分布相对均衡的二叉查找树来说，如果节点总数是n，那么搜索节点的时间复杂度就是O（logn），和树的深度是一样的。

这种依靠比较大小来逐步查找的方式，和二分查找算法非常相似。

2.维持相对顺序

这一点仍然要从二叉查找树说起。二叉查找树要求左子树节点的值小于父节点的值，右子树节点的值大于父节点的值，正是这样保证了二叉树的有序性。

因此二叉查找树还有另一个名字——二叉排序树（binary sort tree）。

新插入的节点，同样要遵循二叉排序树的原则。例如插入新元素5，由于5<6、5>3、5>4，所以5最终会插到节点4的右孩子位置。

再如插入新元素10，由于10>6、10>8、10>9，所以10最终会插到节点9的右孩子位置。

这一切看起来很顺利，然而却隐藏着一个致命的问题。什么问题呢？下面请试着在二叉查找树中依次插入9、8、7、6、5、4，看看会出现什么结果。

 哈哈，好好的一个二叉树，变成“跛脚”啦！

 不只是外观看起来变得怪异了，查询节点的时间复杂度也退化成了O（n）。

怎么解决这个问题呢？这就涉及二叉树的自平衡了。二叉树自平衡的方式有多种，如红黑树、AVL树、树堆等。由于篇幅有限，本书就不一一详细讲解了，感兴趣的读者可以查一查相关资料。

除二叉查找树以外，二叉堆也维持着相对的顺序。不过二叉堆的条件要宽松一些，只要求父节点的值比它的左右孩子的值都大，这一点在后面的章节中我们会详细讲解。

 好了，有关树和二叉树的基本知识，我们就讲到这里。

 本节所讲的内容偏于理论方面，没有涉及代码。但是下一节讲解二叉树的遍历时，会涉及大量代码，大家要做好准备哦！

3.2　二叉树的遍历

3.2.1　为什么要研究遍历

当我们介绍数组、链表时，为什么没有着重研究它们的遍历过程呢？

二叉树的遍历又有什么特殊之处呢？

在计算机程序中，遍历本身是一个线性操作。所以遍历同样具有线性结构的数组或链表，是一件轻而易举的事情。

反观二叉树，是典型的非线性数据结构，遍历时需要把非线性关联的节点转化成一个线性的序列。以不同的方式来遍历，遍历出的序列顺序也不同。

那么，二叉树都有哪些遍历方式呢？

从节点之间位置关系的角度来看，二叉树的遍历分为4种。

1.前序遍历。

2.中序遍历。

3.后序遍历。

4.层序遍历。

从更宏观的角度来看，二叉树的遍历归结为两大类。

1.深度优先遍历（前序遍历、中序遍历、后序遍历）。

2.广度优先遍历（层序遍历）。

下面就来具体看一看这些不同的遍历方式。

3.2.2　深度优先遍历

深度优先和广度优先这两个概念不止局限于二叉树，它们更是一种抽象的算法思想，决定了访问某些复杂数据结构的顺序。在访问树、图，或其他一些复杂数据结构时，这两个概念常常被使用到。

所谓深度优先，顾名思义，就是偏向于纵深，“一头扎到底”的访问方式。可能这种说法有些抽象，下面就通过二

叉树的前序遍历、中序遍历、后序遍历，来看一看深度优先是怎么回事吧。

1.前序遍历

二叉树的前序遍历，输出顺序是根节点、左子树、右子树。

上图就是一个二叉树的前序遍历，每个节点左侧的序号代表该节点的输出顺序，详细步骤如下。

1.首先输出的是根节点1。

2.由于根节点1存在左孩子，输出左孩子节点2。

3.由于节点2也存在左孩子，输出左孩子节点4。

4.节点4既没有左孩子，也没有右孩子，那么回到节点2，输出节点2的右孩子节点5。

5.节点5既没有左孩子，也没有右孩子，那么回到节点1，输出节点1的右孩子节点3。

6.节点3没有左孩子，但是有右孩子，因此输出节点3的右孩子节点6。

到此为止，所有的节点遍历输出完毕。

2.中序遍历

二叉树的中序遍历，输出顺序是左子树、根节点、右子树。

上图就是一个二叉树的中序遍历，每个节点左侧的序号代表该节点的输出顺序，详细步骤如下。

1.首先访问根节点的左孩子，如果这个左孩子还拥有左孩子，则继续深入访问下去，一直找到不再有左孩子的节点，并输出该节点。显然，第一个没有左孩子的节点是节点4。

2.依照中序遍历的次序，接下来输出节点4的父节点2。

3.再输出节点2的右孩子节点5。

4.以节点2为根的左子树已经输出完毕，这时再输出整个二叉树的根节点1。

5.由于节点3没有左孩子，所以直接输出根节点1的右孩子节点3。

6.最后输出节点3的右孩子节点6。

到此为止，所有的节点遍历输出完毕。

3.后序遍历

二叉树的后序遍历，输出顺序是左子树、右子树、根节点。

上图就是一个二叉树的后序遍历，每个节点左侧的序号代表该节点的输出顺序。

由于二叉树的后序遍历和前序、中序遍历的思想大致相同，相信聪明的读者已经可以推测出分解步骤，这里就不

再列举细节了。

img那么，二叉树的前序、中序、后序遍历的代码怎么写呢？

img二叉树的这3种遍历方式，用递归的思路可以非常简单地实现出来，让我们看一看代码。

img

img

img

二叉树用递归方式来实现前序、中序、后序遍历，是最为自然的方式，因此代码也非常简单。

这3种遍历方式的区别，仅仅是输出的执行位置不同：前序遍历的输出在前，中序遍历的输出在中间，后序遍历的输出在最后。

代码中值得注意的一点是二叉树的构建。二叉树的构建方法有很多，这里把一个线性的链表转化成非线性的二叉树，链表节点的顺序恰恰是二叉树前序遍历的顺序。链表中的空值，代表二叉树节点的左孩子或右孩子为空的情况。

在代码中，通过{3，2，9，None，None，10，None，None，8，None，4}这样一个线性序列，构建成的二叉树如下：

img

img除使用递归以外，二叉树的深度优先遍历还能通过其他方式实现吗？

img当然也可以用非递归的方式来实现，不过要稍微复杂一些。

绝大多数可以用递归解决的问题，其实都可以用另一种数据结构来解决，这种数据结构就是栈。因为递归和栈都有回溯的特性。

如何借助栈来实现二叉树的非递归遍历呢？下面以二叉树的前序遍历为例，看一看具体过程。

1.首先遍历二叉树的根节点1，放入栈中。

2.遍历根节点1的左孩子节点2，放入栈中。

3.遍历节点2的左孩子节点4，放入栈中。

4.节点4既没有左孩子，也没有右孩子，我们需要回溯到上一个节点2。可是现在并不是做递归操作，怎么回溯呢？

别担心，栈已经存储了刚才遍历的路径。让旧的栈顶元素4出栈，就可以重新访问节点2，得到节点2的右孩子节点5。

此时节点2已经没有利用价值（已经访问过左孩子和右孩子），节点2出栈，节点5入栈。

5.节点5既没有左孩子，也没有右孩子，我们需要再次回溯，一直回溯到节点1。所以让节点5出栈。

根节点1的右孩子是节点3，节点1出栈，节点3入栈。

6.节点3的右孩子是节点6，节点3出栈，节点6入栈。

7.节点6既没有左孩子，也没有右孩子，所以节点6出栈。此时栈为空，遍历结束。

img二叉树非递归前序遍历的代码已经写好了，让我们来看一看。

至于二叉树的中序、后序遍历的非递归实现，思路和前序遍历差不多，都是利用栈来进行回溯。各位读者要是有兴趣的话，可以自己尝试用代码实现一下。

3.2.3 广度优先遍历

如果说深度优先遍历是在一个方向上“一头扎到底”，那么广度优先遍历则恰恰相反：先在各个方向上各走出1步，再在各个方向上走出第2步、第3步……一直到各个方向全部走完。听起来有些抽象，下面让我们通过二叉树的层序遍历，来看一看广度优先是怎么回事。

层序遍历，顾名思义，就是二叉树按照从根节点到叶子节点的层次关系，一层一层横向遍历各个节点。

上图就是一个二叉树的层序遍历，每个节点左侧的序号代表该节点的输出顺序。

可是，二叉树同一层次的节点之间是没有直接关联的，如何实现这种层序遍历呢？这里同样需要借助一个数据结构来辅助工作，这个数据结构就是队列。

详细遍历步骤如下。

1.根节点1进入队列。

2.节点1出队，输出节点1，并得到节点1的左孩子节点2、右孩子节点3。让节点2和节点3入队。

3.节点2出队，输出节点2，并得到节点2的左孩子节点4、右孩子节点5。让节点4和节点5入队。

4.节点3出队，输出节点3，并得到节点3的右孩子节点6。让节点6入队。

5.节点4出队，输出节点4，由于节点4没有孩子节点，所以没有新节点入队。

6.节点5出队，输出节点5，由于节点5同样没有孩子节点，所以没有新节点入队。

7.节点6出队，输出节点6，节点6没有孩子节点，没有新节点入队。

到此为止，所有的节点遍历输出完毕。

 这个层序遍历看起来有点意思，代码怎么写呢？

 代码不难写，让我们来看一看。

 我基本上明白了，最后想问问，二叉树的层序遍历可以用递归方式来实现吗？

 可以，不过在思路上有一点绕。我们把这个作为思考题，聪明的读者如果有兴趣，可以想一想层序遍历的递归实现方法哦！

 好了，有关二叉树的遍历问题就讲到这里，咱们下一节再见！

3.3 什么是二叉堆

3.3.1 初识二叉堆

什么是二叉堆呢？

二叉堆本质上是一种完全二叉树，它分为两种类型：

1.最大堆。

2.最小堆。

什么是最大堆呢？最大堆的任何一个父节点的值，都大于或等于它左孩子或右孩子节点的值。

什么是最小堆呢？最小堆的任何一个父节点的值，都小于或等于它左孩子或右孩子节点的值。

二叉堆的根节点叫作堆顶。

最大堆和最小堆的特点决定了：最大堆的堆顶是整个堆中的最大元素；最小堆的堆顶是整个堆中的最小元素。

 那么，我们如何构建一个堆呢？

 这就需要依靠二叉堆的自我调整了。

3.3.2　二叉堆的自我调整

对于二叉堆，有如下几种操作：

1.插入节点。

2.删除节点。

3.构建二叉堆。

这几种操作都基于堆的自我调整。所谓堆的自我调整，就是把一个不符合堆性质的完全二叉树，调整成一个堆。下面让我们以最小堆为例，看一看二叉堆是如何进行自我调整的。

1.插入节点

当在二叉堆中插入节点时，插入位置是完全二叉树的最后一个位置。例如插入一个新节点，值是0。

这时，新节点的父节点5比0大，显然不符合最小堆的性质。于是让新节点“上浮”，和父节点交换位置。

继续用节点0和父节点3做比较，因为0小于3，则让新节点继续“上浮”。

继续比较，最终新节点0“上浮”到了堆顶位置。

2.删除节点

从二叉堆删除节点的过程和插入节点的过程正好相反，所删除的是处于堆顶的节点。例如，删除最小堆的堆顶节点1。

这时，为了继续维持完全二叉树的结构，我们把堆的最后一个节点10临时补到原本堆顶的位置。

接下来，让暂处堆顶位置的节点10和它的左孩子、右孩子进行比较，如果左孩子、右孩子节点中最小的一个（显然是节点2）比节点10小，那么让节点10“下沉”。

继续让节点10和它的左孩子、右孩子做比较，左孩子、右孩子中最小的是节点7，由于10大于7，让节点10继续“下沉”。

这样一来，二叉堆重新得到了调整。

3.构建二叉堆

构建二叉堆，也就是把一个无序的完全二叉树调整为二叉堆，本质就是让所有非叶子节点依次“下沉”。

下面举一个无序完全二叉树的例子，如下图所示：

首先，从最后一个非叶子节点开始，也就是从节点10开始。如果节点10大于它的左孩子、右孩子节点中最小的一个，则节点10“下沉”。

接下来轮到节点3，如果节点3大于它的左孩子、右孩子节点中最小的一个，则节点3“下沉”。

然后轮到节点1，如果节点1大于它的左孩子、右孩子节点中最小的一个，则节点1“下沉”。事实上，节点1小于它的左孩子、右孩子，所以不用改变。

接下来轮到节点7，如果节点7大于它的左孩子、右孩子节点中最小的一个，则节点7“下沉”。

继续比较节点7，继续“下沉”。

经过上述几轮比较和“下沉”操作，最终每一节点都小于它的左孩子、右孩子节点，一个无序的完全二叉树就被构建成了一个最小堆。

 小灰，你来思考一下，堆的插入、删除、构建操作的时间复杂度各是多少？

 堆的插入操作是单一节点的“上浮”，堆的删除操作是单一节点的“下沉”，这两个操作的平均交换次数都是堆高度的一半，所以时间复杂度是O（logn）。至于堆的构

建，需要所有非叶子节点依次“下沉”，所以我觉得时间复杂度应该是O（nlogn）吧？

img关于堆的插入和删除操作，你说的没错，时间复杂度确实是O（logn）。但构建堆的时间复杂度却不是O（nlogn），而是O（n）。这涉及数学推导过程，有兴趣的话，你可以自己琢磨一下哦。

img这二叉堆还真有点意思，那么怎么用代码来实现呢？

3.3.3 二叉堆的代码实现

在展示代码之前，我们还需要明确一点：二叉堆虽然是一个完全二叉树，但它的存储方式并不是链式存储，而是顺序存储。换句话说，二叉堆的所有节点都存储在数组中。

在数组中没有左指针和右指针的情况下，如何定位一个父节点的左孩子和右孩子呢？

像上图那样，可以依靠数组下标来计算。

假设父节点的下标是parent，那么它的左孩子的下标就是2×parent+1；右孩子的下标就是2×parent+2。

例如在上面的例子中，节点6包含9和10两个孩子节点，节点6在数组中的下标是3，节点9在数组中的下标是7，节点10在数组中的下标是8。

那么，

7=3×2+1，

8=3×2+2，

刚好符合规律。

有了这个前提，下面的代码就更好理解了。

代码中有一个优化的点，就是在父节点和孩子节点做连续交换时，并不一定要真的交换，只需先把交换一方的值存入temp变量，做单向覆盖，循环结束后，再把temp的值存入交换后的最终位置即可。

img咱们讲了这么多关于二叉堆的知识，二叉堆究竟有什么用处呢？

img二叉堆是实现堆排序及优先队列的基础。关于这两者，我们会在后续的章节中详细介绍。

3.4 什么是优先队列

3.4.1 优先队列的特点

img

img

队列的特点是什么？

在之前的章节中已经讲过，队列的特点是先进先出（FIFO）。

入队列，将新元素置于队尾：

img

出队列，队头元素最先被移出：

img

那么，优先队列又是什么样子的呢？

优先队列不再遵循先入先出的原则，而是分为两种情况：

- 最大优先队列，无论入队顺序如何，都是当前最大的元素优先出队。
- 最小优先队列，无论入队顺序如何，都是当前最小的元素优先出队。

例如，有一个最大优先队列，其中的最大元素是8，那么虽然8并不是队头元素，但出队时仍然让元素8首先出队。

要实现以上需求，利用线性数据结构并非不能实现，但是时间复杂度较高。

 哎呀，那该怎么办呢？

 别担心，这时候我们的二叉堆就派上用场了。

3.4.2　优先队列的实现

先来回顾一下二叉堆的特性：

1.最大堆的堆顶是整个堆中的最大元素。

2.最小堆的堆顶是整个堆中的最小元素。

因此，可以用最大堆来实现最大优先队列，这样的话，每一次入队操作就是堆的插入操作，每一次出队操作就是删除堆顶节点。

入队操作具体步骤如下。

1.插入新节点5。

2.新节点5“上浮”到合适位置。

出队操作具体步骤如下。

1.让原堆顶节点10出队。

2.把最后一个节点1替换到堆顶位置。

3.节点1“下沉”，节点9成为新堆顶。

 小灰，你说说这个优先队列的入队和出队操作的时间复杂度分别是多少？

img二叉堆节点“上浮”和“下沉”的时间复杂度都是O（logn），所以优先队列入队和出队的时间复杂度也是O（logn）！

img说的没错，下面让我们来看一看代码实现。

img

img

上述代码采用数组来存储二叉堆的元素，因此当元素数量超过数组长度时，需要进行扩容来扩大数组长度。

img好了，关于优先队列我们就介绍到这里，下一章再见！

3.5 小结

- 什么是树

树是n个节点的有限集，有且仅有一个特定的称为根的节点。当n>1时，其余节点可分为m个互不相交的有限集，每一个集合本身又是一个树，称为根的子树。

- 什么是二叉树

二叉树是树的一种特殊形式，每一个节点最多有两个孩子节点。二叉树包含完全二叉树和满二叉树两种特殊形式。

- 二叉树的遍历方式有几种

根据节点之间的位置关系，可以分为前序遍历、中序遍历、后序遍历、层序遍历这4种方式；从更宏观的角度划分，可以划分为深度优先遍历和广度优先遍历两大类。

- 什么是二叉堆

二叉堆是一种特殊的完全二叉树，分为最大堆和最小堆。

在最大堆中，任何一个父节点的值，都大于或等于它的左孩子、右孩子节点的值。

在最小堆中，任何一个父节点的值，都小于或等于它的左孩子、右孩子节点的值。

- 什么是优先队列

优先队列分为最大优先队列和最小优先队列。

在最大优先队列中，无论入队顺序如何，当前最大的元素都会优先出队，这是基于最大堆实现的。

在最小优先队列中，无论入队顺序如何，当前最小的元素都会优先出队，这是基于最小堆实现的。

第4章　排序算法

4.1　引言

在生活中，我们离不开排序。例如上体育课时，同学们会按照身高顺序进行排队；又如每一场考试后，老师会按照考试成绩排名次。

在编程的世界中，应用到排序的场景也比比皆是。例如当开发一个学生管理系统时，需要按照学号从小到大进行排序；当开发一个电商平台时，需要把同类商品按价格从低到高进行排序；当开发一款游戏时，需要按照游戏得分从多到少对玩家进行排序，排名第一的玩家就是本场比赛的MVP，等等。

由此可见，排序无处不在。

排序看似简单，它的背后却隐藏着多种多样的算法和思想。那么常用的排序算法都有哪些呢？

根据时间复杂度的不同，主流的排序算法可以分为3大类。

1.时间复杂度为O（n2）的排序算法

- 冒泡排序
- 选择排序
- 插入排序
- 希尔排序（希尔排序比较特殊，它的性能略优于O（n2），但又比不上O（nlogn），姑且把它归入本类）

2.时间复杂度为O（nlogn）的排序算法

- 快速排序

- 归并排序
- 堆排序

3.时间复杂度为线性的排序算法

- 计数排序
- 桶排序
- 基数排序

当然，以上列举的只是主流的排序算法，在算法界还存在着更多五花八门的排序算法，它们有些基于传统排序算法变形而来；有些则是脑洞大开，如鸡尾酒排序、猴子排序、睡眠排序等。

此外，排序算法还可以根据其稳定性，划分为稳定排序和不稳定排序。即如果值相同的元素在排序后仍然保持着排序前的顺序，则这样的排序算法是稳定排序；如果值相同的元素在排序后打乱了排序前的顺序，则这样的排序算法是不稳定排序。例如下面的例子。

在大多数场景中，值相同的元素谁先谁后是无所谓的。但是在某些场景下，值相同的元素必须保持原有的顺序。

由于篇幅所限，我们无法把所有的排序算法都一一详细讲述。在本章中，将只讲述几个具有代表性的排序算法：冒泡排序、快速排序、堆排序、计数排序、桶排序。

下面就要带领大家进入有趣的排序世界了，请“坐稳扶好”！

4.2 什么是冒泡排序

4.2.1 初识冒泡排序

什么是冒泡排序？

冒泡排序的英文是bubble sort，它是一种基础的交换排序。

大家一定都喝过汽水，汽水中常常有许多小小的气泡哗啦哗啦浮到上面来。这是因为组成小气泡的二氧化碳比水轻，所以小气泡可以一点一点地向上浮动。

而冒泡排序之所以叫冒泡排序，正是因为这种排序算法的每一个元素都可以像小气泡一样，根据自身大小，一点一点地向着数组的一侧移动。

具体如何移动呢？让我们先来看一个例子。

有8个数字组成一个无序数列{5，8，6，3，9，2，1，7}，希望按照从小到大的顺序对其进行排序。

按照冒泡排序的思想，我们要把相邻的元素两两比较，当一个元素大于右侧相邻元素时，交换它们的位置；当一个元素小于或等于右侧相邻元素时，位置不变。详细过程如下：

这样一来，元素9作为数列中最大的元素，就像是汽水里的小气泡一样，“漂”到了最右侧。

这时，冒泡排序的第1轮就结束了。数列最右侧元素9的位置可以认为是一个有序区域，有序区目前只有1个元素：

下面，让我们来进行第2轮排序：

第2轮排序结束后，数列右侧的有序区有了2个元素，顺序如下：

后续的交换细节，这里就不详细描述了，第3轮到第7轮的状态如下：

到此为止，所有元素都是有序的了，这就是冒泡排序的整体思路。

冒泡排序是一种稳定排序，值相等的元素并不会打乱原本的顺序。由于该排序算法的每一轮都要遍历所有元素，总共遍历（元素数量-1）轮，所以平均时间复杂度是O（n2）。

 OK，冒泡排序的思路我大概明白了，那么，怎么用代码来实现呢？

 原始的冒泡排序算法的代码我写了一下，你来看一看。

冒泡排序算法第1版代码示例如下：

代码非常简单，使用双循环进行排序。外部循环控制所有的回合，内部循环实现每一轮的冒泡处理，先进行元素比较，再进行元素交换。

 原来如此，冒泡排序的代码并不难理解呀。

 这只是冒泡排序的原始实现，还存在很大的优化空间呢。

4.2.2 冒泡排序的优化

原始的冒泡排序有哪些可以优化的点呢？

让我们回顾一下刚才描述的排序细节，仍然以{5，8，6，3，9，2，1，7}这个数列为例，当排序算法分别执行到第6轮、第7轮时，数列状态如下：

很明显可以看出，经过第6轮排序后，整个数列已经是有序的了，可是排序算法仍然兢兢业业地继续执行了第7轮排序。

在这种情况下，如果能判断出数列已经有序，并做出标记，那么剩下的几轮排序就不必执行了，可以提前结束工作。

冒泡排序算法第2版代码示例如下：

与第1版代码相比，第2版代码做了小小的改动，利用布尔变量is_sorted作为标记。如果在本轮排序中，元素有交换，则说明数列无序；如果没有元素交换，则说明数列已经有序，然后直接跳出大循环。

 不错呀，原来冒泡排序算法还可以这样优化。

 这只是冒泡排序算法优化的第一步，我们还可以进一步来提升它的性能。

为了说明问题，这次以一个新的数列为例：

这个数列的特点是，前半部分的元素（3、4、2、1）无序，后半部分的元素（5、6、7、8）按升序排列，并且后半部分元素中的最小值也大于前半部分元素的最大值。

下面按照冒泡排序算法的思路来进行排序，看一看具体效果。

第1轮

元素4和5比较，发现4小于5，所以位置不变。

元素5和6比较，发现5小于6，所以位置不变。

元素6和7比较，发现6小于7，所以位置不变。

元素7和8比较，发现7小于8，所以位置不变。

第1轮结束，数列有序区包含1个元素。

第2轮

元素3和2比较，发现3大于2，所以3和2交换。

元素3和4比较，发现3小于4，所以位置不变。

元素4和5比较，发现4小于5，所以位置不变。

元素5和6比较，发现5小于6，所位位置不变。

元素6和7比较，发现6小于7，所以位置不变。

元素7和8比较，发现7小于8，所以位置不变。

第2轮结束，数列有序区包含2个元素。

 小灰，你发现其中的问题了吗？

 其实右面的许多元素已经是有序的了，可是每一轮还是白白地比较了许多次。

 没错，这正是冒泡排序算法中另一个需要优化的点。

这个问题的关键点在于对数列有序区的界定。

按照现有的逻辑，有序区的长度和排序的轮数是相等的。例如第1轮排序过后的有序区长度是1，第2轮排序过后的有序区长度是2……

实际上，数列真正的有序区可能会大于这个长度，如上述例子中在第2轮排序时，后面的5个元素实际上都已经属于有序区了。因此后面的多次元素比较是没有意义的。

那么，该如何避免这种情况呢？我们可以在每一轮排序后，记录下最后一次元素交换的位置，该位置即为无序数列的边界，再往后就是有序区了。

冒泡排序算法第3版代码示例如下：

在第3版代码中，sort_border就是无序数列的边界。在每一轮排序过程中，处于sort_border之后的元素就不需要再进行比较了，肯定是有序的。

 真是学到了很多知识，想不到冒泡排序可以玩出这么多花样！

 其实这仍然不是最优的，还有一种排序算法叫作鸡尾酒排序，是基于冒泡排序的一种升级排序法。

4.2.3 鸡尾酒排序

冒泡排序的每一个元素都可以像小气泡一样，根据自身大小，一点一点地向着数组的一侧移动。算法的每一轮都是从左到右来比较元素，并进行单向的位置交换的。

那么鸡尾酒排序做了怎样的优化呢？

鸡尾酒排序的元素比较和交换过程是双向的。

下面举一个例子。

由8个数字组成一个无序数列{2，3，4，5，6，7，8，1}，希望对其进行从小到大的排序。

如果按照冒泡排序的思想，排序过程如下：

 元素2、3、4、5、6、7、8已经是有序的了，只有元素1的位置不对，却还要进行7轮排序，这也太“憋屈”了吧！

 没错，鸡尾酒排序正是要解决这个问题的。

那么鸡尾酒排序是什么样子的呢？让我们来看一看详细过程。

第1轮（和冒泡排序一样，8和1交换）

第2轮

此时开始不一样了，我们反过来从右往左比较并进行交换。

第3轮（虽然实际上已经有序，但是流程并没有结束）

在鸡尾酒排序的第3轮，需要重新从左向右比较并进行交换。

1和2比较，位置不变；2和3比较，位置不变；3和4比较，位置不变……6和7比较，位置不变。

没有元素位置进行交换，证明已经有序，排序结束。

这就是鸡尾酒排序的思路。排序过程就像钟摆一样，第1轮从左到右，第2轮从右到左，第3轮再从左到右……

 哇，本来要用7轮排序的场景，用3轮就解决了，鸡尾酒排序可真是巧妙的算法！

 确实挺巧妙的，让我们来看一下它的代码实现吧。

这段代码是鸡尾酒排序的原始实现。代码外层的大循环控制着所有排序回合，大循环内包含2个小循环，第1个小循环从左向右比较并交换元素，第2个小循环从右向左比较并交换元素。

 代码大致看明白了。之前讲冒泡排序时，有一种针对有序区的优化，鸡尾酒排序是不是也能用到呢？

 当然喽！鸡尾酒排序也可以和之前所学的优化方法结合使用，只不过代码实现会稍微复杂一些，这里就不再展开讲解了，有兴趣的话，可以自己写一下代码哦。

imgOK，最后我想问问，鸡尾酒排序的优点和缺点是什么？适用于什么样的场景？

img鸡尾酒排序的优点是能够在特定条件下，减少排序的回合数；而缺点也很明显，就是代码量几乎增加了1倍。

img至于它能发挥出优势的场景，是大部分元素已经有序的情况。好了，关于冒泡排序和鸡尾酒排序，我们就介绍到这里。下一节再见！

4.3 什么是快速排序

4.3.1 初识快速排序

同冒泡排序一样，快速排序也属于交换排序，通过元素之间的比较和交换位置来达到排序的目的。

不同的是，冒泡排序在每一轮中只把1个元素冒泡到数列的一端，而快速排序则在每一轮挑选一个基准元素，并让其他比它大的元素移动到数列一边，比它小的元素移动到数列的另一边，从而把数列拆解成两个部分。

这种思路就叫作分治法。

每次把数列分成两部分，究竟有什么好处呢？

假如给出一个8个元素的数列，一般情况下，使用冒泡排序需要比较7轮，每一轮把1个元素移动到数列的一端，时间复杂度是O（n2）。

而快速排序的流程是什么样子的呢？

如上图所示，在分治法的思想下，原数列在每一轮都被拆分成两部分，每一部分在下一轮又分别被拆分成两部分，直到不可再分为止。

每一轮的比较和交换，需要把数组中的全部元素都遍历一遍，时间复杂度是O（n）。这样的遍历一共需要多少轮呢？假如元素个数是n，那么平均情况下需要logn轮，因此快速排序算法总体的平均时间复杂度是O（nlogn）。

 分治法果然神奇！那么基准元素是如何选的呢？又如何把其他元素移动到基准元素的两端呢？

 基准元素的选择，以及元素的交换，都是快速排序的核心问题。让我们先来看看如何选择基准元素。

4.3.2　基准元素的选择

基准元素，英文是pivot，在分治过程中，以基准元素为中心，把其他元素移动到它的左右两边。

那么如何选择基准元素呢？

最简单的方式是选择数列的第1个元素。

这种选择在绝大多数情况下是没有问题的。但是，假如有一个原本逆序的数列，期望排序成顺序数列，那么会出现什么情况呢？

 哎呀，整个数列并没有被分成两半，每一轮都只确定了基准元素的位置。

 是呀，在这种情况下，数列的第1个元素要么是最小值，要么是最大值，根本无法发挥分治法的优势。

 在这种极端情况下，快速排序需要进行n轮，时间复杂度退化成了O（n2）。

那么，该怎么避免这种情况发生呢？

其实很简单，我们可以随机选择一个元素作为基准元素，并且让基准元素和数列首元素交换位置。

这样一来，即使在数列完全逆序的情况下，也可以有效地将数列分成两部分。

当然，即使是随机选择基准元素，也会有极小的概率选到数列的最大值或最小值，同样会影响分治的效果。

所以，虽然快速排序的平均时间复杂度是O（nlogn），但最坏情况下的时间复杂度是O（n2）。

在后文中，为了简化步骤，省去了随机选择基准元素的过程，直接把首元素作为基准元素。

4.3.3 元素的交换

选定了基准元素，我们要做的就是把其他元素中小于基准元素的都交换到基准元素的一边，大于基准元素的都交换到基准元素的另一边。

具体如何实现呢？有两种方法：

1.双边循环法。

2.单边循环法。

何谓双边循环法？下面来看一看详细过程。

给出原始数列如下，要求对其从小到大进行排序。

首先，选定基准元素pivot，并且设置两个指针left和right，指向数列的最左和最右两个元素。

接下来进行第1次循环，从right指针开始，让指针所指向的元素和基准元素做比较。如果大于或等于pivot，则指针向左移动；如果小于pivot，则right指针停止移动，切换到left指针。

在当前数列中，1<4，所以right直接停止移动，换到left指针，进行下一步行动。

轮到left指针行动，让指针所指向的元素和基准元素做比较。如果小于或等于pivot，则指针向右移动；如果大于pivot，则left指针停止移动。

由于left开始指向的是基准元素，判断肯定相等，所以left右移1位。

由于7>4，left指针在元素7的位置停下。这时，让left指针和right指针所指向的元素进行交换。

接下来，进入第2次循环，重新切换到right指针，向左移动。right指针先移动到8，8>4，继续左移。由于2<4，停止在2的位置。

按照这个思路，后续步骤如下图所示：

 大致明白了，那么快速排序怎样用代码来实现呢？

 我们来看一下用双边循环法实现的快速排序，代码使用了递归的方式。

在上述代码中，quick_sort方法通过递归的方式，实现了分而治之的思想。

partition_v1方法则实现了元素的交换，让数列中的元素依据自身大小，分别交换到基准元素的左右两边。在这里，我们使用的交换方式是双边循环法。

 partition_v1的代码实现好复杂呀，在一个大循环里还嵌套着两个子循环……让我仔细消化消化。

img双边循环法的代码确实有些烦琐。除了这种方式，要实现元素的交换也可以利用单边循环法，下一节我们来仔细讲一讲。

4.3.4 单边循环法

双边循环法从数组的两边交替遍历元素，虽然更加直观，但是代码实现相对烦琐。而单边循环法则简单得多，只从数组的一边对元素进行遍历和交换。我们来看一看详细过程。

给出原始数列如下，要求对其从小到大进行排序。

开始和双边循环法相似，首先选定基准元素pivot。同时，设置一个mark指针指向数列起始位置，这个mark指针代表小于基准元素的区域边界。

接下来，从基准元素的下一个位置开始遍历数组。

如果遍历到的元素大于基准元素，就继续往后遍历。

如果遍历到的元素小于基准元素，则需要做两件事：第一，把mark指针右移1位，因为小于pivot的区域边界增大了1；第二，让最新遍历到的元素和mark指针所在位置的元素交换位置，因为最新遍历的元素归属于小于pivot的区域。

首先遍历到元素7，7>4，所以继续遍历。

接下来遍历到的元素是3，3<4，所以mark指针右移1位。

随后，让元素3和mark指针所在位置的元素交换，因为元素3归属于小于pivot的区域。

按照这个思路，继续遍历，后续步骤如下图所示：

 明白了，这个方法只需要单边循环，确实简单了许多呢！怎么用代码来实现呢？

 双边循环法和单边循环法的区别在于partition_v2函数的实现，让我们来看一下代码。

可以很明显地看出，partition_v2方法只要一个大循环就搞定了，的确比双边循环法简单多了。

 以上所讲的快速排序实现方法，都是以递归为基础的。其实快速排序也可以基于非递归的方式来实现。

4.3.5　非递归实现

 怎样用非递归的方式来实现呢？

 绝大多数的递归逻辑，都可以用栈的方式来代替。

为什么这样说呢？

在第1章介绍空间复杂度时我们曾经提到过，代码中一层一层的方法调用，本身就使用了一个方法调用栈。每次进入一个新方法，就相当于入栈；每次有方法返回，就相当于出栈。

所以，可以把原本的递归实现转化成一个栈的实现，在栈中存储每一次方法调用的参数。

下面来看一下具体的代码：

和刚才的递归实现相比，非递归方式代码的变动只发生在quick_sort方法中。该方法引入了一个栈，栈中的字典元素用于存储每一次交换时的起始下标和结束下标。

每一次循环，都会让栈顶元素出栈，通过partition方法进行分治，并且按照基准元素的位置分成左右两部分，左右两部分再分别入栈。当栈为空时，说明排序已经完毕，退出循环。

img居然真的实现了非递归方法，好棒！

img嘿嘿，快速排序是很重要的算法，与傅里叶变换等算法并称为二十世纪十大算法。

img有关快速排序的知识我们就介绍到这里，希望大家把这个算法吃透，未来会受益无穷！

4.4 什么是堆排序

4.4.1 传说中的堆排序

img

img

还记得二叉堆的特性是什么吗？

1.最大堆的堆顶是整个堆中的最大元素。

2.最小堆的堆顶是整个堆中的最小元素。

以最大堆为例，如果删除一个最大堆的堆顶（并不是完全删除，而是跟末尾的节点交换位置），经过自我调整，第2大的元素就会被交换上来，成为最大堆的新堆顶。

img

正如上图所示，在删除值为10的堆顶节点后，经过调整，值为9的新节点就会顶替上来；在删除值为9的堆顶节点后，经过调整，值为8的新节点就会顶替上来……

由于二叉堆的这个特性，每一次删除旧堆顶，调整后的新堆顶都是大小仅次于旧堆顶的节点。那么只要反复删除堆顶，反复调整二叉堆，所得到的集合就会成为一个有序集合，过程如下。

删除节点9，节点8成为新堆顶。

删除节点8，节点7成为新堆顶。

删除节点7，节点6成为新堆顶。

删除节点6，节点5成为新堆顶。

删除节点5，节点4成为新堆顶。

删除节点4，节点3成为新堆顶。

删除节点3，节点2成为新堆顶。

到此为止，原本的最大二叉堆已经变成了一个从小到大的有序集合。之前说过，二叉堆实际存储在数组中，数组中的元素排列如下：

由此，可以归纳出堆排序算法的步骤：

1.把无序数组构建成二叉堆。需要从小到大排序，则构建成最大堆；需要从大到小排序，则构建成最小堆。

2.循环删除堆顶元素，替换到二叉堆的末尾，调整堆产生新的堆顶。

 大体思路明白了，那么该如何用代码来实现呢？

img讲二叉堆时，我们写了二叉堆操作的相关代码。现在只要在原代码的基础上稍微改动一点，就可以实现堆排序了。

4.4.2　堆排序的代码实现

img原来如此，我现在明白了！那么堆排序的时间复杂度和空间复杂度各是多少呢？

img毫无疑问，空间复杂度是O（1），因为并没有开辟额外的集合空间。至于时间复杂度，我们来分析一下。

二叉堆的节点“下沉”调整（down_adjust 方法）是堆排序算法的基础，这个调节操作本身的时间复杂度在上一章讲过，是O（logn）。

我们再来回顾一下堆排序算法的步骤：

1.把无序数组构建成二叉堆。

2.循环删除堆顶元素，并将该元素移到集合尾部，调整堆产生新的堆顶。

第1步，把无序数组构建成二叉堆，这一步的时间复杂度是O（n）。

第2步，需要进行n-1次循环。每次循环调用一次down_adjust方法，所以第2步的计算规模是（n-1）×logn，时间复杂度为O（nlogn）。

两个步骤是并列关系，所以整体的时间复杂度是O（nlogn）。

img最后一个问题，从宏观上看，堆排序和快速排序相比，有什么区别和联系呢？

img先说说相同点，堆排序和快速排序的平均时间复杂度都是O（nlogn），并且都是不稳定排序。至于不同点，

快速排序的最坏时间复杂度是O（n2），而堆排序的最坏时间复杂度稳定在O（nlogn）。

img此外，快速排序递归和非递归方法的平均空间复杂度都是O（logn），而堆排序的空间复杂度是O（1）。

img好了，关于堆排序算法，我们就介绍到这里。

4.5 计数排序和桶排序

4.5.1 线性时间的排序

img

img哇，什么样的排序算法可以这么厉害？

img让我们先来回顾一下以前所学的排序算法，无论是冒泡排序，还是快速排序，都是基于元素之间的比较来进行排序的。

例如冒泡排序。

如下图所示，因为8>3，所以8和3的位置交换：

img

例如堆排序。

如下图所示，因为10>7，所以10和7的位置交换：

img

img排序当然要先比较呀，难道还有不需要比较的排序算法？

img有一些特殊的排序并不基于元素比较，如计数排序、桶排序、基数排序。

img以计数排序来说，这种排序算法是利用数组下标来确定元素的正确位置的。

4.5.2 初识计数排序

img还是不明白，元素下标怎么能用来帮助排序呢？

img那让我们来看一个例子。

假设数组中有20个随机整数，取值范围为0～10，要求用最快的速度把这20个整数从小到大进行排序。

如何给这些无序的随机整数进行排序呢？

考虑到这些整数只能够在0、1、2、3、4、5、6、7、8、9、10这11个数中取值，取值范围有限。所以，可以根据这有限的范围，建立一个长度为11的数组。数组下标从0到10，元素初始值全为0：

假设20个随机整数的值如下所示：

9，3，5，4，9，1，2，7，8，1，3，6，5，3，4，0，10，9，7，9

下面就开始遍历这个无序的随机数列，每一个整数按照其值对号入座，同时，对应数组下标的元素进行加1操作。

例如，第1个整数是9，那么数组下标为9的元素加1：

第2个整数是3，那么数组下标为3的元素加1：

继续遍历数列并修改数组……

最终，当数列遍历完毕时，数组的状态如下：

该数组中每一个下标位置的值代表数列中对应整数出现的次数。

有了这个统计结果，排序就很简单了。直接遍历数组，输出数组元素的下标值，元素的值是几，就输出几次：

0，1，1，2，3，3，3，4，4，5，5，6，7，7，8，9，9，9，9，10

显然，现在输出的数列已经是有序的了。

img这就是计数排序的基本过程，它适用于一定范围内的整数排序。在取值范围不是很大的情况下，它的性能甚至快过那些时间复杂度为O（nlogn）的排序。

img明白了，计数排序还真是个神奇的算法！那么，用代码怎么实现呢？

img我写了一个计数排序的初步实现代码，我们来看一下。

img

img

这段代码在开头有一个步骤，就是求数列的最大整数值max_value。后面创建的统计数组count_array，长度是max_value+1，以此来保证数组的最后一个下标是max_value。

4.5.3　计数排序的优化

img从实现功能的角度来看，这段代码可以实现整数的排序。但是这段代码也存在一些问题，你发现了吗？

img哦，让我想想……

img对了！我们只以数列的最大值来决定统计数组的长度，其实并不严谨。例如下面的数列：

95，94，91，98，99，90，99，93，91，92

img这个数列的最大值是99，但最小的整数是90。如果创建长度为100的数组，那么前面从0到89的空间位置就都浪费了！

怎么解决这个问题呢？

很简单，只要不再以输入数列的最大值+1作为统计数组的长度，而是以数列最大值-最小值+1作为统计数组的长度即可。

同时，数列的最小值作为一个偏移量，用于计算整数在统计数组中的下标。

以刚才的数列为例，统计出数组的长度为99-90+1=10，偏移量等于数列的最小值90。

对于第1个整数95，对应的统计数组下标是95-90=5，如下图所示：

 是的，这确实对计数排序进行了优化。此外，朴素版的计数排序只是简单地按照统计数组的下标输出元素值，并没有真正给原始数列进行排序。

 如果只是单纯地给整数排序，这样做并没有问题。但如果在现实业务里，例如给学生的考试分数进行排序，遇到相同的分数就会分不清谁是谁。

什么意思呢？让我们看看下面的例子。

给出一个学生成绩表，要求按成绩从低到高进行排序，如果成绩相同，则遵循原表固有顺序。

那么，当我们填充统计数组以后，只知道有两个成绩并列为95分的同学，却不知道哪一个是小红，哪一个是小绿。

 明白你的例子了，但为什么我的成绩最低呀……那么，这种分数相同的情况要怎么解决？

 在这种情况下，需要稍微改变之前的逻辑，在填充完统计数组以后，对统计数组做一下变形。

仍然以刚才的学生成绩表为例，将之前的统计数组变形成下面的样子：

这是如何变形的呢？其实就是从统计数组的第2个元素开始，每一个元素都加上前面所有元素之和。

为什么要相加呢？初次接触的读者可能会觉得莫名其妙。

这样相加的目的，是让统计数组存储的元素值，等于相应整数的最终排序位置的序号。例如，下标是9的元素值为5，代表原始数列的整数9，最终的排序在第5位。

接下来，创建输出数组sorted_array，长度和输入数列一致。然后从后向前遍历输入数列。

第1步，遍历成绩表最后一行的小绿同学的成绩。

小绿的成绩是95分，找到count_array下标是5的元素，值是4，代表小绿的成绩排名位置在第4位。

同时，给count_array下标是5的元素值减1，从4变成3，代表下次再遇到95分的成绩时，最终排名是第3。

第2步，遍历成绩表倒数第2行的小白同学的成绩。

小白的成绩是94分，找到count_array下标是4的元素，值是2，代表小白的成绩排名位置在第2位。

同时，给count_array下标是4的元素值减1，从2变成1，代表下次再遇到94分的成绩时（实际上已经遇不到了），最终排名是第1。

第3步，遍历成绩表倒数第3行的小红同学的成绩。

小红的成绩是95分，找到count_array下标是5的元素，值是3（最初是4，减1变成了3），代表小红的成绩排名位置在第3位。

同时，给count_array下标是5的元素值减1，从3变成2，代表下次再遇到95分的成绩时（实际上已经遇不到了），最终排名是第2。

这样一来，同样是95分的小红和小绿就能够清楚地排出顺序了，也正因为这样，优化版本的计数排序属于稳定排

序。

后面的遍历过程以此类推，这里就不详细描述了。

img还真是够绕的，不过大体上明白了。那么，优化之后的计数排序如何用代码实现呢？

img说起来复杂，其实代码很简洁，让我们来看一看。

img小灰，如果原始数列的规模是n，最大整数和最小整数的差值是m，你说说计数排序的时间复杂度和空间复杂度是多少？

img代码第1、2、4步都涉及遍历原始数列，运算量都是n，第3步遍历统计数列，运算量是m，所以总体运算量是3n+m，去掉系数，时间复杂度是O（n+m）。

img至于空间复杂度，如果不考虑结果数组，只考虑统计数组大小的话，空间复杂度是O（m）。

img不错哦，回答得很赞！

img不过我有一点不太明白，既然计数排序这么强大，为什么很少被大家使用呢？

img因为计数排序有它的局限性，主要表现为如下两点。

1.当数列最大和最小值差距过大时，并不适合用计数排序。

例如，给出20个随机整数，范围在0到1亿之间，这时如果使用计数排序，需要创建长度为1亿的数组。不但严重浪费空间，而且时间复杂度也会随之升高。

2.当数列元素不是整数时，也不适合用计数排序。

如果数列中的元素都是小数，如25.213或0.00000001这样的数字，则无法创建对应的统计数组。这样显然无法进行计数排序。

img对于这些局限性，另一种线性时间排序算法做出了弥补，这种排序算法叫作桶排序。

4.5.4 什么是桶排序

img桶排序又是什么？

img桶排序同样是一种线性时间的排序算法，它类似于计数排序所创建的统计数组，桶排序需要创建若干个桶来协助排序。

那么，桶排序中所谓的“桶”，又是什么呢？

每一个桶（bucket）代表一个区间范围，里面可以承载一个或多个元素。

假设有一个非整数数列，如下：

4.5，0.84，3.25，2.18，0.5

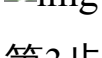
让我们来看看桶排序的工作原理。

桶排序的第1步，就是创建这些桶，并确定每一个桶的区间范围。

具体需要建立多少个桶，如何确定桶的区间范围，有很多种不同的方式。我们这里创建的桶数量等于原始数列的元素数量，除最后一个桶只包含数列最大值外，前面各个桶的区间按照比例来确定。

区间跨度=（最大值-最小值）/（桶的数量-1）

第2步，遍历原始数列，把元素对号入座放入各个桶中。



第3步，对每个桶内部的元素分别进行排序（显然，只有第1个桶需要排序）。

第4步，遍历所有的桶，输出所有元素。

0.5，0.84，2.18，3.25，4.5

到此为止，排序结束。

img我大体明白了，那么，代码怎么写呢？

img我们来看一看桶排序的代码实现。

img

img

在上述代码中，所有的桶都保存在bucket_list集合中，每个桶都是一个列表。

同时，上述代码使用了sort方法对桶内元素进行排序。sort方法底层采用的是Timsort排序算法，各位读者可以简单地把它们当作一种时间复杂度为O（nlogn）的排序。

img那么，桶排序的时间复杂度是多少呢？

img桶排序的时间复杂度有些复杂，让我们来计算一下。

假设原始数列有n个元素，分成n个桶。

下面逐步来分析一下算法复杂度。

第1步，求数列最大值、最小值，运算量为n。

第2步，创建空桶，运算量为n。

第3步，把原始数列的元素分配到各个桶中，运算量为n。

第4步，在每个桶内部做排序，在元素分布相对均匀的情况下，所有桶的运算量之和为n。

第5步，输出排序数列，运算量为n。

因此，桶排序的总体时间复杂度为O（n）。

至于空间复杂度就很容易得到了，同样是O（n）。

img桶排序的性能并非绝对稳定。如果元素的分布极不均衡，在极端情况下，第一个桶中有n-1个元素，最后一

个桶中有1个元素。此时的时间复杂度将退化为O（nlogn），而且还白白创建了许多空桶。

 由此可见，并没有绝对好的算法，也没有绝对不好的算法，关键要看具体的应用场景。

 关于计数排序和桶排序的知识，我们就介绍到这里，下一章再见！

4.6 小结

本章我们学习了一些具有代表性的排序算法。下面根据算法的时间复杂度、空间复杂度、是否稳定等维度来做一个归纳。

第5章　面试中的算法

5.1　踌躇满志的小灰

这一章，我们开始讲解形形色色的算法面试题，其中有许多是面试过程中常常遇到的经典题目。小灰究竟能不能面试成功呢？让我们为他加油吧！

5.2　如何判断链表有环

5.2.1　一场与链表相关的面试

 下面我来考查你一道算法题。

题目

有一个单向链表，链表中有可能出现“环”，就像下图这样。

那么，如何用程序来判断该链表是否为有环链表呢？

 哦，让我想想啊……

 有了！我可以从头节点开始遍历整个单链表……

方法1：

首先从头节点开始，依次遍历单链表中的每一个节点。每遍历一个新节点，就从头检查新节点之前的所有节点，用新节点和此节点之前所有节点依次做比较。如果发现新节点和之前的某个节点相同，则说明该节点被遍历过两次，

链表有环；如果之前的所有节点中不存在与新节点相同的节点，就继续遍历下一个新节点，继续重复刚才的操作。

就像图中这样，当遍历链表节点7时，从头访问节点5和节点3，发现已遍历的节点中并不存在节点7，则继续往下遍历。

当第2次遍历到节点2时，从头访问曾经遍历过的节点，发现已经遍历过节点2，说明链表有环。

假设链表的节点数量为n，则该解法的时间复杂度为O（n2）。由于并没有创建额外的存储空间，所以空间复杂度为O（1）。

 OK，这姑且算是一种方法，有没有效率更高的解法？

 哦，让我想想啊……

 或者，我创建一个哈希表，然后……

方法2：

首先创建一个以节点ID为Key的set集合，用来存储曾经遍历过的节点。然后同样从头节点开始，依次遍历单链表中的每一个节点。每遍历一个新节点，都用新节点和set集合中存储的节点进行比较，如果发现set中存在与之相同的节点ID，则说明链表有环，如果set中不存在与新节点相同的节点ID，就把这个新节点ID存入 set中，之后进入下一节点，继续重复刚才的操作。

遍历过5、3。

遍历过5、3、7、2、6、8、1。

当再一次遍历节点2时，查找set，发现节点已存在。

由此可知，链表有环。

这个方法在流程上和方法1类似，本质的区别是使用了set作为额外的缓存。

假设链表的节点数量为n，则该解法的时间复杂度是O（n）。由于使用了额外的存储空间，所以算法的空间复杂度同样是O（n）。

imgOK，这种方法在时间上已经最优了。有没有可能在空间上也得到优化？

img哦，让我想想啊……

img想不出来啊，怎么能让时间复杂度不变，同时让空间复杂度降低呢？

img呵呵，没关系，今天就到这里，你回家等通知吧。

img

5.2.2　解题思路

img小灰，你刚刚去面试了？结果怎么样？

img唉……

img大黄，你给我讲讲呗，怎么能够更高效地判断一个链表是否有环呀？

img哈哈，小灰，有环链表的判断问题是很基础的算法题，许多面试官都喜欢考查，你必须要掌握哦！

img对于这道题，有一个很巧妙的方法，这个方法利用了两个指针。

方法3：

首先创建两个指针p1和p2（在Python里就是两个对象引用），让它们同时指向这个链表的头节点。然后开始一个大循环，在循环体中，让指针p1每次向后移动1个节点，让指针p2每次向后移动2个节点，然后比较两个指针指向

的节点是否相同。如果相同，则可以判断出链表有环，如果不同，则继续下一次循环。

第1步，p1和p2都指向节点5。

第2步，p1指向节点3，p2指向节点7。

第3步，p1指向节点7，p2指向节点6。

第4步，p1指向节点2，p2指向节点1。

第5步，p1指向节点6，p2也指向节点6，p1和p2所指相同，说明链表有环。

学过小学奥数的读者，一定听说过数学上的追及问题。此方法就类似一个追及问题。

在一个环形跑道上，两个运动员从同一地点起跑，一个运动员速度快，另一个运动员速度慢。当两人跑了一段时间后，速度快的运动员必然会再次追上并超过速度慢的运动员，原因很简单，因为跑道是环形的。

假设链表的节点数量为n，则该算法的时间复杂度为O（n）。除两个指针外，没有使用任何额外的存储空间，所以空间复杂度是O（1）。

 那么，这个算法用代码怎么实现呢？

 代码实现很简单，让我们来看一下。

 明白了，这真是个好方法！

5.2.3　问题扩展

img这道题其实还可以扩展出许多有意思的问题，例如下面这些。

扩展问题1：

如果链表有环，如何求出环的长度？

扩展问题2：

如果链表有环，如何求出入环节点？

 哎呀，这两个问题怎么解呢？

 第1个问题求环长，非常简单，解法如下。

当两个指针首次相遇，证明链表有环的时候，让两个指针从相遇点继续循环前进，并统计前进的循环次数，直到两个指针第2次相遇。此时，统计出来的前进次数就是环长。

因为指针p1每次走1步，指针p2每次走2步，两者的速度差是1步。当两个指针再次相遇时，p2比p1多走了整整1圈。

因此，环长= 每一次速度差 × 前进次数= 前进次数。

 第2个问题是求入环点，有些难度，我们可以做一个抽象的推断。

上图是对有环链表所做的一个抽象示意图。假设从链表头节点到入环点的距离是D，从入环点到两个指针首次相遇点的距离是S1，从首次相遇点回到入环点的距离是S2。

那么，当两个指针首次相遇时，各自所走的距离是多少呢？

指针p1一次只走1步，所走的距离是D+S1。

指针p2一次走2步，多走了n（n>=1）整圈，所走的距离是D+S1+n（S1+S2）。

由于p2的速度是p1的2倍，所以所走距离也是p1的2倍，因此：

2(D+S1)=D+S1+n(S1+S2)

等式经过整理得出：

D= (n-1)(S1+S2)+S2

也就是说，从链表头节点到入环点的距离，等于从首次相遇点绕环n-1圈再回到入环点的距离。

这样一来，只要把其中一个指针放回到头节点位置，另一个指针保持在首次相遇点，两个指针都是每次向前走1步。那么，它们最终相遇的节点，就是入环节点。

 哇，居然这么神奇？

 我们不妨用原题中链表的例子来演示一下。

首先，让指针p1回到链表头节点，指针p2保持在首次相遇点。

指针p1和p2各自前进1步。

指针p1和p2第2次前进。

指针p1和p2第3次前进，指向了同一个节点2，节点2正是有环链表的入环点。

 果真在入环点相遇了呢，这下明白了！

 好了，关于判断链表是否有环及其扩展的题目，我们就介绍到这里。咱们下一节再见！

5.3 最小栈的实现

5.3.1 一场关于栈的面试

 下面我来考查你一道算法题。

题目

实现一个栈，该栈带有出栈（pop）、入栈（push）、取最小元素（get_min）3个方法。要保证这3个方法的时间复杂度都是O（1）。

 哦，让我想想……

 我想到啦！可以把栈中的最小元素下标暂存起来……

小灰的思路如下。

1.创建一个整型变量min，用来存储栈中的最小元素。当第1个元素进栈时，把进栈元素赋值给min，即把栈中唯一的元素当作最小值。

2.之后每当一个新元素进栈时，就让新元素和min比较大小。如果新元素小于min，则min等于新进栈的元素；如果新元素大于或等于min，则不做改变。

3.当调用get_min方法时，直接返回min的值即可。

 小灰，你有没有觉得这个思路存在什么问题？

 没有问题呀？这个解法杠杠的！

 呵呵，今天面试就先到这里，回家等通知去吧！

5.3.2　解题思路

img小灰，你刚刚去面试了？结果怎么样？

img唉……

img大黄，怎么才能实现一个最小栈呀？我采用临时变量暂存栈的最小值，究竟存在什么问题呢？

img小灰，你想得太简单啦！你只考虑了进栈场景，却没有考虑出栈场景。

img哦？出栈场景有什么问题吗？

img让我来给你演示一下。

原本，栈中最小的元素是3，min变量记录的值也是3。

这时，栈顶元素出栈了。

此时的min变量应该等于几呢？

虽然此时的最小元素是4，但是程序并不知道。

img哎呀，还真是……

img所以说，只暂存一个最小值是不够的，我们需要存储栈中曾经的最小值，作为“备胎”。

详细的解法步骤如下。

1.设原有的栈叫作栈A，此时创建一个额外的“备胎”栈B，用于辅助栈A。

2.当第1个元素进入栈A时，让新元素也进入栈B。这个唯一的元素是栈A的当前最小值。

3.之后，每当新元素进入栈A时，比较新元素和栈A当前最小值的大小，如果小于栈A当前最小值，则让新元素进入栈B，此时栈B的栈顶元素就是栈A当前最小值。

4.每当栈A有元素出栈时，如果出栈元素是栈A当前最小值，则让栈B的栈顶元素也出栈。此时栈B余下的栈顶元素所指向的，是栈A中原本第2小的元素，代替刚才的出栈元素成为栈A的当前最小值。（备胎转正。）

5.当调用get_min方法时，返回栈B的栈顶所存储的值，这也是栈A的最小值。

显然，这个解法中进栈、出栈、取最小值的时间复杂度都是O（1），最坏情况空间复杂度是O（n）。

 这下我明白了！那么代码怎么来实现呢？

 代码不难实现，让我们来看一看。

这段代码的第一行输出的是3，因为当时的最小值是3；第二行输出的是4，因为元素3出栈后，最小值是4。

 好了，关于最小栈题目的解法就介绍到这里，咱们下一节再见！

5.4　如何求出最大公约数

5.4.1　一场求最大公约数的面试

 下面我来考查你一道算法题，数学里面的最大公约数，知道吧？

 这个我知道，小学就学过。

 那么，看看下面这道算法题。

题目

写一段代码，求出两个整数的最大公约数，要尽量优化算法的性能。

 哦，让我试试……

 写出来啦！你看看。

小灰的代码如下：

小灰的思路十分简单。他使用“暴力枚举”的方法，从较小整数的一半开始，试图找到一个合适的整数i，看看这个整数能否被a和b同时整除。

 你的这个方法虽然实现了所要求的功能，但是效率不行啊。想想看，如果我传入的整数是10000和10001，用你的方法就需要循环10000/2-1=4999次！

 哎呀，这倒是个问题。

 想不出更好的方法了……

 呵呵，没关系，回家等通知去吧！

5.4.2　解题思路

 小灰，你刚刚去面试了？结果怎么样？

 唉……

 大黄，怎么才能更高效地求出两个整数的最大公约数呀？

 小灰，你听说过辗转相除法吗？

 辗……什么除法？

 是辗转相除法！又叫作欧几里得算法。

辗转相除法，又名欧几里得算法（Euclidean algorithm），该算法的目的是求出两个正整数的最大公约数。它是已知最古老的算法，其产生时间可追溯至公元前300年。

这条算法基于一个定理：两个正整数a和b（a>b），它们的最大公约数等于a除以b的余数c和b之间的最大公约数。例如，10和25，25除以10商2余5，那么10和25的最大公约数，等于10和5的最大公约数。

有了这条定理，求最大公约数就变得简单了。我们可以使用递归的方法把问题逐步简化。

首先，计算出a除以b的余数c，把问题转化成求b和c的最大公约数；然后计算出b除以c的余数d，把问题转化成求c和d的最大公约数；再计算出c除以d的余数e，把问题转化成求d和e的最大公约数……

以此类推，逐渐把两个较大整数之间的运算简化成两个较小整数之间的运算，直到两个数可以整除，或者其中一个数减小到1为止。

 说了这么多理论不如直接写代码，小灰，你按照辗转相除法的思路改改你的代码吧。

 好的，让我试试！

辗转相除法的实现代码如下：

 没错，这确实是辗转相除法的思路。不过有一个问题，当两个整数较大时，做a%b取模运算的性能会比较差。

 这我也明白，可是不取模的话，还能怎么办呢？

 说到这里，另一个算法就要登场了，它叫作更相减损术。

更相减损术，出自中国古代的《九章算术》，也是一种求最大公约数的算法。古希腊人很聪明，可是我们炎黄子孙也不差。

它的原理更加简单：两个正整数a和b（a>b），它们的最大公约数等于a-b的差值c和较小数b的最大公约数。例如，10和25，25减10的差是15，那么10和25的最大公约数，等同于10和15的最大公约数。

由此，我们同样可以通过递归来简化问题。首先，计算出a和b的差值c（假设a>b），把问题转化成求b和c的最大公约数；然后计算出c和b的差值d（假设c>b），把问题转化成求b和d的最大公约数；再计算出b和d的差值e（假设b>d），把问题转化成求d和e的最大公约数……

以此类推，逐渐把两个较大整数之间的运算简化成两个较小整数之间的运算，直到两个数相等为止，最大公约数就是最终相等的这两个数的值。

imgOK，这就是更相减损术的思路，你按照这个思路再写一段代码看看。

img好的，让我试试！

更相减损术的实现代码如下：

img

img很好，更相减损术的过程就是这样的。我们避免了大整数取模可能出现的性能问题，已经越来越接近最优解决方案了。

img但是，更相减损术依靠两数求差的方式来递归，运算次数肯定远大于辗转相除法的取模方式吧？

img能发现这个问题，看来你进步了。更相减损术是不稳定的算法，当两数相差悬殊时，如计算10000和1的最大公约数，就要递归9999次！

img有什么办法可以既避免大整数取模，又能尽可能地减少运算次数呢？

img下面就是我要说的最优方法：把辗转相除法和更相减损术的优势结合起来，在更相减损术的基础上使用移位运算。

众所周知，移位运算的性能非常好。对于给出的正整数a和b，不难得到如下的结论：

（从下文开始，获得最大公约数的方法get_greatest_common_divisor简写为gcd。）

当a和b均为偶数时，gcd（a，b）=2×gcd（a/2，b/2）=2×gcd（a>>1，b>>1）。

当a为偶数，b为奇数时，gcd（a，b）= gcd（a/2，b）= gcd（a>>1，b）。

当a为奇数，b为偶数时，gcd（a，b）= gcd（a，b/2）= gcd（a，b>>1）。

当a和b均为奇数时，先利用更相减损术运算一次，gcd（a，b）= gcd（b，a-b），此时a-b必然是偶数，然后又可以继续进行移位运算。

例如，计算10和25的最大公约数的步骤如下：

1.整数10通过移位，可以转换成求5和25的最大公约数。

2.利用更相减损术，计算出25-5=20，转换成求5和20的最大公约数。

3.整数20通过移位，可以转换成求5和10的最大公约数。

4.整数10通过移位，可以转换成求5和5的最大公约数。

5.利用更相减损术，因为两数相等，所以最大公约数是5。

这种方式在两数都比较小时，可能看不出计算次数的优势；当两数越大时，计算次数的减少就会越明显。

 说了这么多，来看看代码吧，这是最终版本的代码。

在上述代码中，判断整数奇偶性的方式是让整数和1进行与运算，如果（a&1）==0，则说明整数a是偶数；如果（a&1）！=0，则说明整数a是奇数。

img真不容易呀，终于得到了最优解！

img嘿嘿，作为程序员，就是需要反复推敲，追求代码的极致！

img我还有最后一个问题，咱们使用的这些方法，时间复杂度分别是多少呢？

img让我们来总结一下上述解法的时间复杂度。

1.暴力枚举法：时间复杂度是O（min（a，b））。

2.辗转相除法：时间复杂度不太好计算，可以近似为O（log（max（a，b））），但是取模运算性能较差。

3.更相减损术：避免了取模运算，但是算法性能不稳定，最坏时间复杂度为O（max（a，b））。

4.更相减损术与移位相结合：不但避免了取模运算，而且算法性能稳定，时间复杂度为O（log（max（a，b））））。

img好了，有关最大公约数的求解，我们就介绍到这里。咱们下一节再会！

5.5　如何判断一个数是否为2的整数次幂

5.5.1　一场很“2”的面试

img

img下面我来考查你一道算法题，给你一个正整数，如何判断它是不是2的整数次幂？

题目

实现一个方法，来判断一个正整数是否是2的整数次幂（如16是2的4次方，返回true；18不是2的整数次幂，则返回false），要求性能尽可能高。

img哦，让我想想……

img我想到了！利用一个整型变量，让它从1开始不断乘以2，将每一次乘2的结果和目标整数进行比较。

小灰的具体想法如下：

创建一个中间变量temp，初始值是1。然后进入一个循环，每次循环都让temp乘以2，并和目标整数相比较，如果相等，则说明目标整数是2的整数次幂；如果不相等，则让temp增大1倍，继续循环并进行比较。当temp的值大于目标整数时，说明目标整数不是2的整数次幂。

举一个例子。

给出一个整数19，则

1×2=2，

2×2=4，

4×2=8，

8×2=16，

16×2=32，

由于32>19，所以19不是2的整数次幂。

如果目标整数的大小是n，则此方法的时间复杂度是O（logn）。

img代码已经写好了，快来看看！

img

imgOK，这样写实现了所要求的功能，你思考一下该怎么来提高其性能呢？

img哦，让我想想……

img我想到了，可以把之前乘以2的操作改成向左移位，移位的性能比乘法高得多。来看看改变之后的代码吧。

img

OK，这样确实有一定优化。但目前算法的时间复杂度仍然是O（logn），本质上没有变。

imgimg如何才能在性能上有质的飞跃呢？

imgimg哦，让我想想……

imgimg想不出来啦，时间复杂度为O（logn）已经很快了，难道还能有O（1）的方法？

imgimg呵呵，没关系，今天面试就到这儿，回家等通知去吧。

imgimg

5.5.2 解题思路

imgimg小灰，你刚刚去面试了？结果怎么样？

imgimg唉……

imgimg大黄，怎么才能更高效地判断一个整数是否是2的整数次幂呢？难道存在时间复杂度只有O（1）的方法？

imgimg小灰呀，这个题目还真有时间复杂度为O（1）的解法。

imgimgReally？怎么做到呢？

imgimg你先想一想，如果把2的整数次幂转换成二进制数，会有什么样的共同点？

imgimg让我想想，十进制数的2转换成二进制数是10B，4转换成二进制数是100B，8转化成二进制数是1000B……

imgimg

imgimg我知道了！如果一个整数是2的整数次幂，那么当它转化成二进制数时，只有最高位是1，其他位都是0！

imgimg没错，是这样的。接下来如果把这些2的整数次幂各自减1，再转化成二进制数，会有什么样的特点呢？

imgimg都减1？让我试试啊！

imgimg

img我发现了，2的整数次幂一旦减1，它的二进制数的数字就全部变成了1！

img很好，这时候如果用原数值（2的整数次幂）和它减1的结果进行按位与运算，也就是n&（n-1），会是什么结果呢？

img

img0和1按位与运算的结果是0，所以凡是2的整数次幂和它本身减1的结果进行与运算，结果都必定是0。反之，如果一个整数不是2的整数次幂，结果一定不是0！

img那么，解决这个问题的方法已经很明显了，你说说怎样来判断一个整数是否是2的整数次幂？

img很简单，对于一个整数n，只需要计算n&（n-1）的结果是不是0。这个方法的时间复杂度只有O（1）。

img代码我已经写好了，除方法声明外，只有1行哦！

img

img非常好，这就是位运算的妙用。关于这道题目我们就说到这里，下一节再会！

5.6　无序数组排序后的最大相邻差

5.6.1　一道奇葩的面试题

img

img下面我来考查你一道算法题，有一个无序整型数组……

题目

有一个无序整型数组，如何求出该数组排序后的任意两个相邻元素的最大差值？要求时间复杂度和空间复杂度尽可能低。

可能题目有点绕，让我们来看一个例子。


哦，让我想想……


嗨，这还不简单吗？先使用时间复杂度为O（nlogn）的排序算法给原来的数组排序，然后遍历数组，对每两个相邻元素求差，最大差值不就求出来了吗？

解法1：

使用任意一种时间复杂度为O（nlogn）的排序算法（如快速排序）给原数组排序，然后遍历排好序的数组，并对每两个相邻元素求差，最终得到最大差值。

该解法的时间复杂度是O（nlogn），在不改变原数组的情况下，空间复杂度是O（n）。


唉，我出这样的题目，显然不是为了让你来排序的。你再想想，有没有更快的解法？


没有了呀。不排序的话还能怎么做呢？


呵呵，那你回家等通知去吧！

5.6.2　解题思路


小灰，你刚刚去面试了？结果怎么样？


唉……


大黄，我今天遇见一道怪题，怎样才能计算出无序数组排序后的最大相邻差值？


嗯……这道题确实很有意思。虽然对数组排序以后肯定能得到正确的结果，但我们没有必要真的去进行排序。


不排序的话，该怎么办呢？


小灰，你记不记得，有哪些排序算法的时间复杂度是线性的？

img好像有计数排序、桶排序，还有个什么基数排序……可你刚才不是说不用排序吗？

img别着急，我们仅仅是借助一下这些排序的思想而已。小灰，你想一下，这道题能不能像计数排序一样，利用数组下标来解决？

img像计数排序一样？让我想想啊……

img有了！我可以使用计数排序的思想，先找出原数组中最大值和最小值的差……

解法2：

1.利用计数排序的思想，先求出原数组中的最大值max与最小值min的区间长度k（k=max-min+1），以及偏移量d=min。

2.创建一个长度为k的新数组Array。

3.遍历原数组，每遍历一个元素，就把新数组Array对应下标的值+1。例如原数组元素的值为n，则将Array[n-min]的值加1。遍历结束后，Array的一部分元素的值变成了1或更高的数值，一部分元素的值仍然是0。

4.遍历新数组Array，统计出Array中最大连续出现0值的次数+1，即为相邻元素最大差值。

例如给定一个无序数组 {2，6，3，4，5，10，9}，处理过程如下。

第1步，确定k（数组长度）和d（偏移量）。

img

第2步，创建数组。

img

第3步，遍历原数组，对号入座。

img

第4步，判断0值最多连续出现的次数，计算出最大相邻差。

 很好，我们已经进步了很多。这个思路在数组元素差值不是很悬殊的时候，确实效率很高。

 可是设想一下，如果原数组只有3个元素：1、2、1000000，那就要创建长度是1000000的数组！想一想还能如何优化？

 让我想想啊……

 对了！桶排序的思想正好解决了这个问题！

解法3：

1.利用桶排序的思想，根据原数组的长度n，创建出n个桶，每一个桶代表一个区间范围。其中第1个桶从原数组的最小值min_value开始，区间跨度是（max_value-min_value）/（n-1）。

2.遍历原数组，把原数组中的每一个元素插入对应的桶中，记录每一个桶的最大值和最小值。

3.遍历所有的桶，统计出每一个桶的最大值，和这个桶右侧非空桶的最小值的差，数值最大的差即为原数组排序后的相邻最大差值。

例如，给出一个无序数组 {2，6，3，4，5，10，9}，处理过程如下。

第1步，根据原数组，创建桶，确定每个桶的区间范围。

第2步，遍历原数组，确定每个桶内的最大和最小值。

第3步，遍历所有的桶，找出最大相邻差。

img这个方法不需要像标准桶排序那样在每一个桶内部进行排序，只需要记录桶内的最大值和最小值即可，所以时间复杂度稳定在O（n）。

img很好，让我们来写一下代码吧。

img好的，我试试。

img

img

代码的前几步都比较直观，唯独第4步稍微有些不好理解：使用临时变量left_max，在每一轮迭代时存储当前左侧桶的最大值。而两个桶之间的差值，则是buckets[i].min-left_max。

img没错，这就是这道题目的最优解决方法。关于无序数组排序后最大差值的问题就介绍到这里，咱们下一节再见！

5.7　如何用栈实现队列

5.7.1　又是一道关于栈的面试题

img

img那么下面我来考查你一道算法题，怎样用栈来实现一个队列？

题目

用栈来模拟一个队列，要求实现队列的两个基本操作：入队、出队。

img哦……栈是先入后出，队列是先入先出，用栈没办法实现队列吧？

img提示你一下，用一个栈肯定是没办法实现队列的，但如果我们有两个栈呢？

img让我想想啊……

img没想出来，就算给我8个栈，我也不知道怎么实现队列。

img呵呵，没事，回家等通知去吧！

5.7.2　解题思路

img小灰，你刚刚去面试了？结果怎么样？

img唉……

img大黄，你能不能给我讲讲，怎样可以用两个栈来实现一个队列呀？

img要解决这个问题，我们先来回顾一下栈和队列的不同特点。

栈的特点是先入后出，出入元素都是在同一端（栈顶）。

入栈：

出栈：

队列的特点是先入先出，出入元素是在两端（队头和队尾）。

入队：

出队：

既然我们拥有两个栈，那么可以让其中一个栈作为队列的入口，负责插入新元素；另一个栈作为队列的出口，负责移除老元素。

img可是，两个栈是各自独立的，怎么能把它们有效地关联起来呢？

img别着急，让我来具体演示一下。

队列的主要操作无非有两个：入队和出队。

在模拟入队操作时，每一个新元素都被压入栈A当中。

让元素1入队。

让元素2入队。

让元素3入队。

这时，我们希望最先入队的元素1出队，需要怎么做呢？

让栈A中的所有元素按顺序出栈，再按照出栈顺序压入栈B。这样一来，元素从栈A弹出并压入栈B的顺序是3、2、1，和当初进入栈A的顺序1、2、3是相反的。

此时让元素1出队，也就是让元素1从栈B中弹出。

让元素2出队。

img如果这个时候又想做入队操作了呢？

img很简单，当有新元素入队时，重新把新元素压入栈A。

让元素4入队。

此时出队操作仍然从栈B中弹出元素。

让元素3出队。

 现在栈B已经空了，如果再想出队该怎么办呢？

 也不难，只要栈A中还有元素，就像刚才一样，把栈A中的元素弹出并压入栈B即可。

让元素4出队。

 怎么样，这回你绕明白了吗？

 哦，基本上明白了，那么代码怎么来实现呢？

 代码很好写，让我们来看一看。

 小灰，你说说，这个队列的入队和出队操作，时间复杂度分别是多少？

 入队操作的时间复杂度显然是O（1）。至于出队操作，如果涉及栈A和栈B的元素迁移，那么一次出队的时间复杂度是O（n）；如果不用迁移，时间复杂度是O（1）。咦，在这种情况下，出队的时间复杂度究竟应该是多少呢？

 这里涉及一个新的概念，叫作均摊时间复杂度。需要元素迁移的出队操作只是少数情况，并且不可能连续出现，其后的大多数出队操作都不需要元素迁移。

 所以把时间均摊到每一次出队操作上面，其时间复杂度是O（1）。这个概念并不常用，稍做了解即可。

img好了，用栈实现队列的题目，我们就介绍到这里，咱们下一节再见！

5.8 寻找全排列的下一个数

5.8.1 一道关于数字的题目

img

img下面我来考查你一道算法题，假设给出一个正整数，请找出这个正整数所有数字全排列的下一个数。

题目

给出一个正整数，找出这个正整数所有数字全排列的下一个数。

说通俗点就是，在一个整数所包含数字的全部组合中，找到一个大于且仅大于原数的新整数。让我们举几个例子。

如果输入12345，则返回12354。

如果输入12354，则返回12435。

如果输入12435，则返回12453。

img让我想一想啊……

img我发现了，这里面有一个规律！让我来解释一下。

小灰发现的“规律”如下：

输入12345，返回12354，那么

12354-12345=9，

刚好相差9的一次方。

输入12354，返回12435，那么

12435-12354=81，

刚好相差9的二次方。

所以，每次计算最近的换位数，只需要加上9的n次方即可。

img怎么样，我是不是很机智？

img这算哪门子规律？12453-12435=18，24135-23541=594，也并不都是9的整数次幂啊！

img啊，尴尬了……

img呵呵，今天就到这里，回家等通知去吧！

img

5.8.2　解题思路

img小灰，你刚刚去面试了？结果怎么样？

img唉……

img大黄，你能不能给我讲讲，怎样寻找一个整数所有数字全排列的下一个数？

img好啊，在给出具体解法之前，小灰你先思考一个问题：由固定几个数字组成的整数，怎样排列最大？怎样排列最小？

img让我想一想啊……

img知道了，如果是固定的几个数字，应该是在逆序排列的情况下最大，在顺序排列的情况下最小。

举一个例子。

给出1、2、3、4、5这几个数字。

最大的组合：54321。

最小的组合：12345。

img没错，数字的顺序和逆序，是全排列中的两种极端情况。那么普遍情况下，一个数和它最近的全排列数存在什么关联呢？

例如，给出整数12354，它包含的数字是1、2、3、4、5，如何找到这些数字全排列之后仅大于原数的新整数呢？

为了和原数接近，我们需要尽量保持高位不变，低位在最小的范围内变换顺序。

至于变换顺序的范围大小，则取决于当前整数的逆序区域。

如上图所示，12354的逆序区域是最后两位，仅看这两位已经是当前的最大组合。若想最接近原数，又比原数更大，必须从倒数第3位开始改变。

怎样改变呢？12354的倒数第3位是3，我们需要从后面的逆序区域中找到大于3的最小的数字，让其和3的位置进行互换。

互换后的临时结果是12453，倒数第3位已经确定，这个时候最后两位仍然是逆序状态。我们需要把最后两位转变为顺序状态，以此保证在倒数第3位数值为4的情况下，后 两位尽可能小。

这样一来，就得到了想要的结果12435。

 有些明白了，不过还真是复杂呀！

 看起来复杂，其实只要3个步骤。

获得全排列下一个数的3个步骤。

1.从后向前查看逆序区域，找到逆序区域的前一位，也就是数字置换的边界。

2.让逆序区域的前一位和逆序区域中大于它的最小的数字交换位置。

3.把原来的逆序区域转为顺序状态。

img最后让我们用代码来实现一下。这里为了方便数字位置的交换，入参和返回值的类型都采用了整型数组。

img

img

这种解法拥有一个“高大上”的名字：字典序算法。

img小灰，你说说这个解法的时间复杂度是多少？

img该算法3个步骤每一步的时间复杂度都是O（n），所以整体时间复杂度也是O（n）！

img完全正确。关于这道算法题的解答就介绍到这里，咱们下一节再会！

5.9 删去k个数字后的最小值

5.9.1 又是一道关于数字的题目

img

img好吧，下面考你一道算法题：给出一个整数，从该整数中去掉k个数字，要求剩下的数字形成的新整数尽可能小。

题目

给出一个整数，从该整数中去掉k个数字，要求剩下的数字形成的新整数尽可能小。应该如何选取被去掉的数字？

其中整数的长度大于或等于k，给出的整数的大小可以超越Python语言中整型类型的最大值。

什么意思呢？让我们举几个例子。

假设给出一个整数1593212，删去3个数字，新整数最小的情况是1212。

img

假设给出一个整数30200，删去1个数字，新整数最小的情况是200。

假设给出一个整数10，删去2个数字（注意，这里要求删去的不是1个数字，而是2个），新整数的最小情况是0。

 这道题听起来还挺有意思，让我想想……

 你可以先说说你的第一感觉，为了让新整数尽可能小，什么样的数字应该优先删除？

 我知道了！肯定要优先删除最大的数字！如先删除9，再删除8，再删除7……

 那可不一定，如整数3549，删除1个数字的话，是应该删除数字9吗？

 哎呀，还真是！让我再想想……

 呵呵，不用想了，回家等通知去吧！

5.9.2　解题思路

 小灰，你刚刚去面试了？结果怎么样？

 唉……

 大黄，你能不能给我讲讲，怎样寻找删去k个数字后的最小值呀？

 这道题目要求我们删去k个数字，但我们不妨把问题简化一下：如果只删除1个数字，如何让新整数的值最小？

 我的第一感觉是优先删除最大的数字，可是这个策略似乎不对……

img数字的大小固然重要，而数字的位置更加重要。你想想，一个整数的最高位哪怕只减少1，对数值的影响也是非常大的。

我们来举一个例子。

给出一个整数541270936，要求删去1个数字，让剩下的整数尽可能小。

此时，无论删除哪一个数字，最后的结果都是从9位整数变成8位整数。既然同样是8位整数，显然应该优先把高位的数字降低，这样对新整数的值影响最大。

如何把高位的数字降低呢？很简单，把原整数的所有数字从左到右进行比较，如果发现某一位数字大于它右面的数字，那么在删除该数字后，必然会使该数位的值降低，因为右面比它小的数字顶替了它的位置。

在上面这个例子中，数字5右侧的数字4小于5，所以删除数字5，最高位数字降低成4。

img对于整数541270936，删除一个数字所能得到的最小值是41270936。那么对于41270936，删除一个数字后的最小值，你说说是多少。

img我知道了，是删除数字4！因为从左向右遍历，数字4是第1个比右侧数字大的数（4>1）。

img很好，那么接下来呢？从刚才的结果1270936中再删除一个数字，能得到的最小值是多少？

img这一次的情况略微复杂，因为1<2、2<7、7>0，所以被删除的数字应该是7！

img不错，这里每一步都要求得到删除一个数字后的最小值，经历3次，相当于求出了删除k（k=3）个数字后的最小值。

img像这样依次求得局部最优解，最终得到全局最优解的思想，叫作贪心算法。

img小灰，按照这个思路，你尝试用代码来实现一下吧。

img好的，我来写一写试试吧。

img

img

小灰的代码使用了两层循环，外层循环次数就是要删除的数字个数k，内层循环从左到右遍历所有数字。当遍历到需要删除的数字时，利用Python的字符串截取方法把对应的数字删除，并重新拼接字符串。

显然，这段代码的时间复杂度是O（kn）。

imgOK，这段代码在功能实现上是没有问题的，但是性能却不怎么好。主要问题在于以下两个方面。

1.每一次内层循环都需要从头开始遍历所有数字。

例如，给出的整数是11111111111114132，在第1轮循环中，需要遍历大部分数字，一直遍历到数字4，发现4>1，从而删除4。

以目前的代码逻辑，下一轮循环时，还要从头开始遍历，再次重复遍历大部分数字，一直遍历到数字3，发现3>2，从而删除3。

事实上，我们应该停留在上一次删除的位置继续进行比较，而不是再次从头开始遍历。

2.字符串截取方法本身性能不高。

字符串截取方法的底层实现，涉及新字符串的创建，以及逐个字符的复制。这个方法自身的时间复杂度是O（n）。

因此，我们应该避免在每删除一个数字后就进行字符串的截取和拼接。

img哎呀，那应该怎么来优化呢？

img以k作为外循环，遍历数字作为内循环，需要额外考虑的东西非常多。

img所以我们换一个思路，以遍历数字作为外循环，以k作为内循环，这样可以写出非常简洁的代码，让我们来看一看。

上述代码非常巧妙地运用了栈的特性，在遍历原整数的数字时，让所有数字一个一个入栈，当某个数字需要被删除时，让该数字出栈。最后，程序把栈中的元素转化为字符串类型的结果。

下面仍然以整数541270936，k=3为例。当遍历到数字5时，数字5入栈。

当遍历到数字4时，发现栈顶5>4，栈顶5出栈，数字4入栈。

当遍历到数字1时，发现栈顶4>1，栈顶4出栈，数字1入栈。

继续遍历数字2、数字7，并依次入栈。

最后，遍历数字0，发现栈顶7>0，栈顶7出栈，数字0入栈。

此时k的次数已经用完，无须再比较，让剩下的数字一起入栈即可。

此时栈中的元素就是最终的结果。

上面的方法只对所有数字遍历了一次，遍历的时间复杂度是O（n），把栈转化为字符串的时间复杂度也是O（n），所以最终的时间复杂度是O（n）。

同时，程序中利用栈来回溯遍历过的数字及删除数字，所以程序的空间复杂度是O（n）。

 哇，这段代码好巧妙啊！

 这段代码其实仍然有优化空间，各位读者可以思考一下。好了，关于这道题目我们就介绍到这里，感谢大家！

5.10　如何找到两个数组的中位数

5.10.1　有关中位数的问题

 好吧，下面考你一道算法题……

题目：

给定两个升序数组，如何找出这两个数组归并以后新的升序数组的中位数？

什么意思呢？让我们来看两个例子：

上图这两个给定数组A和B，一个长度是6，一个长度是5，归并之后的大数组仍然要保持升序，结果如下：

大数组的长度是奇数（11），中位数显然是位于正中的第6个元素，也就是元素5。上面的例子是奇数个元素的情况。那么偶数个元素是什么样呢？让我们来看另一个例子：

上图这两个给定数组A和B，长度都是5，归并之后的大数组如下：

大数组的长度是偶数（10），位于正中的元素有两个，分别是6和7，这时候的中位数就是两个数的平均值，也就是6.5。

 题意明白了，让我想想啊……

 嗨，这还不简单吗？我直接把两个数组进行归并操作，中位数结果不就出来了吗？

 OK，如果两个数组长度分别是m和n，你这样做的时间复杂度和空间复杂度都是O（m+n），有没有更高效的方法？

 好吧，让我再想想啊……

 不行了，想不出更好的方法了……

 呵呵，没关系，回家等通知去吧！

5.10.2　解题思路

 小灰，你刚刚去面试了？结果怎么样？

 唉……

 大黄，你能不能给我讲讲，如何找出两个升序数组归并之后的中位数呀？

 好啊，在介绍解法之前请你先想一想，一个升序数组的中位数，具备什么样的特点呢？

 特点？中位数、中位数，当然是在数组中间位置喽……

img说的没错，所以中位数把一个升序数组分成了长度相等的两部分，其中左半部分的最大值永远小于或等于右半部分的最小值。

或许这听起来有点绕，我们仍然用刚才的例子来说明：

如上图所示，对于偶数长度的数组，可以根据中位数分成长度相等的两部分，左半部分的最大元素（6），永远小于或等于右半部分的最小元素（7）。

对于奇数长度的数组，同样可以根据中位数分成两部部分：

如上图所示，对于奇数长度的数组，如果把中位数本身归入左半部分，则左半边长度= 右半边长度+1。

左半部分的最大元素（5），永远小于或等于右半部分的最小元素（6）。

img可是题目给定的是两个升序数组呀？

如果不直接进行归并，又如何找出大数组的中位数呢？

img这正是我接下来要讲的，大数组的左右两部分，分别来源于两个初始数组A和B的左右部分。

什么意思呢？大数组被中位数等分的左右两部分，每一部分根据来源又可以再划分成两部分，其中一部分来自数组A的元素，另一部分来自数组B的元素：

如上图所示，原始数组A和数组B，各自分成绿色和橙色两部分。其中数值较小的绿色元素组成了大数组的左半部分，数值较大的橙色元素组成了大数组的右半部分。

最重要的是，绿色元素和橙色元素的数量是相等的（偶数情况），而且最大的绿色元素小于或等于最小的橙色元素。

假设数组A的长度是m，绿色和橙色元素的分界点是i，数组B的长度是n，绿色和橙色元素的分界点是j，那么为了让大数组的左右两部分长度相等，则i和j需要符合如下两个条件：

i+j=（m+n+1）/2

（之所以m+n后面要再加1，是为了应对大数组长度为奇数的情况。）

Max(A[i-1],B[j-1])<= Min(A[i], B[j])

（直白地说，就是最大的绿色元素小于或等于最小的橙色元素。）

由于m+n的值是恒定的，所以我们只要确定一个合适的i，就可以确定j，从而找到大数组左半部分和右半部分的分界，也就找到了归并之后大数组的中位数。

 那么，如何找到这个合适的i值呢？

 办法很简单，我们可以利用“二分查找”的思想。

如何利用二分查找来确定i值呢？通过具体示例，让我们来演示一下：

第一步，就像二分查找那样，把i设在数组A的正中位置，也就是让i=3：

第二步，根据i的值来确定j的值，j=（m+n+1）/2-i=5：

第三步，验证i和j，分为下面三种情况：

1.B[j-1]≤A[i] && A[i-1]≤B[j]

说明i和j左侧的元素都小于或等于右侧的元素，这一组i和j是我们想要的。

2.A[i]<B[j-1]

说明i对应的元素偏小了，i应该向右侧移动。

3.A[i-1]>B[j]

说明i-1对应的元素偏大了，i应该向左侧移动。

显然，图中的例子属于情况2，A[3] < B[5]，所以i应该向右移动。

第四步，在数组A的右半部分，重新确定i的位置，就像二分查找一样：

第五步，同第二步，根据i的值来确定j的值，j=（m+n+1）/2-i=3：

第六步，同第三步，验证i和j：

由于A[5] >= B[2]且B[3]>=A[4]，所以这一组i和j是合适的！

第七步，找出中位数：

如果大数组的长度是奇数，那么：

中位数= Max（A[i-1]，B[j-1]）

（也就是大数组左半部分的最大值。）

如果大数组的长度是偶数，那么：

中位数=（Max（A[i-1]，B[j-1]）+ Min（A[i]， B[i]））/2

（也就是大数组左半部分的最大值和大数组右半部分的最小值取平均。）

在本例中，大数组的长度是奇数，所以中位数=Max（8，12）=12。

 原来如此，我大体上明白了！不过有些特殊情况该怎么处理呢？比如数组A的长度远大于数组B的长度，或者在数组A中无法找到合适的i值……

img对于这些特殊情况，处理的方法也不难。

1.数组A的长度远大于数组B

也就是m远大于n，这时候会出现什么问题呢？

当我们设定了i的初值，也就是数组A正中间的元素，再计算j的时候有可能发生数组越界。

因此，我们可以提前把数组A和数组B进行交换，较短的数组放在前面，i从较短的数组中取。

这样做还有一个好处，由于数组A是较短数组，i的搜索次数减少了。

2.无法找到合适的i值

什么情况下会无法找到合适的i值呢？有两种情况：

数组A的长度小于数组B的长度，并且数组A的所有元素都大于数组B的元素。

在这种情况下，无法通过二分查找寻找到符合B[j-1]≤A[i] && A[i-1]≤B[j]的i值，一直到i=0为止。

此时我们可以跳出二分查找的循环，所求的中位数是B[j-1]。（仅限奇数情况。）

数组A的长度小于数组B的长度，并且数组A的所有元素都小于数组B的元素。

在这种情况下，同样无法通过二分查找寻找到符合B[j-1]≤A[i] && A[i-1]≤B[j]的i值，一直到i=（数组A的长度-1）为止。

此时我们可以跳出二分查找的循环，所求的中位数是Max（A[i-1]，B[j-1]）。（仅限奇数情况。）

img好了，最后让我们看一看代码实现。

img

img

img代码很清晰，这还真是一个巧妙的方法！

img小灰，你说说这个算法的时间复杂度是多少？

img由于最初的交换，数组A的长度是m、n中的较小值，而确定i的过程类似二分查找，所以时间复杂度是O（log min（m，n））。

img说的没错。关于两个数组的中位数问题我们就介绍到这里，咱们下一节再会！

5.11 如何求解金矿问题

5.11.1 一个关于财富自由的问题

img

img下面考你一道算法题，这道算法题目和钱有关系。

题目

很久很久以前，有一位国王拥有5座金矿，每座金矿的黄金储量不同，需要参与挖掘的工人人数也不同。例如，有的金矿储量是500kg黄金，需要5个工人来挖掘；有的金矿储量是200kg黄金，需要3个工人来挖掘……

如果参与挖矿的工人的总数是10。每座金矿要么全挖，要么不挖，不能派出一半人挖取一半的金矿。要求用程序求出，要想得到尽可能多的黄金，应该选择挖取哪几座金矿？

img

img哇，要是我家也有5座金矿，我就财富自由了，也用不着来你这里面试了！

img说正经的！关于这道题你有什么思路吗？

img题目好复杂啊，让我想想……

img我想到了一个办法！我们可以按照金矿的性价比从高到低进行排序，优先选择性价比最高的金矿来挖掘，然后是性价比第2的……

按照小灰的思路，金矿按照性价比从高到低进行排序，排名结果如下：

第1名，350kg黄金/3人的金矿，人均产量约为116.6kg黄金。

第2名，500kg黄金/5人的金矿，人均产量为100kg黄金。

第3名，400kg黄金/5人的金矿，人均产量为80kg黄金。

第4名，300kg黄金/4人的金矿，人均产量为75kg黄金。

第5名，200kg黄金/3人的金矿，人均产量约为66.6kg黄金。

由于工人数量是10人，小灰优先挖掘性价比排名为第1名和第2名的金矿之后，工人

还剩下2人，不够再挖掘其他金矿了。

所以，小灰得出的最佳金矿收益是350+500即850kg黄金。

img怎么样？我这个方案妥妥的吧？

img你的解决思路是使用贪心算法。这种思路在局部情况下是最优解，但是在整体上却未必是最优的。

img给你举个例子吧，如果我放弃性价比最高的350kg黄金/3人的金矿，选择500kg黄金/5人和400kg黄金/5人的金矿，加起来收益是900kg黄金，是不是大于你得到的850kg黄金？

img啊，还真是呢！

img呵呵，没关系，回家等通知去吧！

img

5.11.2 解题思路

img小灰，你刚刚去面试了？结果怎么样？

img唉……

img大黄，你能不能给我讲讲，怎么来求解金矿问题呀？

img好啊，这是一个典型的动态规划题目，和著名的“背包问题”类似。

img动态规划？好“高大上”的概念呀！

img其实也没有那么高深啦。所谓动态规划，就是把复杂的问题简化成规模较小的子问题，再从简单的子问题自底向上一步一步递推，最终得到复杂问题的最优解。

img哦，说了半天我还是没听明白……

img没关系，让我们具体分析一下这个金矿问题，你就能明白动态规划的核心思想了。

首先，对于问题中的金矿来说，每一个金矿都存在着“挖”和“不挖”两种选择。让我们假设一下，如果最后一个金矿注定不被挖掘，那么问题会转化成什么样子呢？显然，问题简化成了10个工人在前4个金矿中做出最优选择。

img

相应地，假设最后一个金矿一定会被挖掘，那么问题又转化成什么样子呢？

由于最后一个金矿消耗了3个工人，问题简化成了7个工人在前4个金矿中做出最优选择。

img

这两种简化情况，被称为全局问题的两个最优子结构。

究竟哪一种最优子结构可以通向全局最优解呢？换句话说，最后一个金矿到底该不该挖呢？

那就要看10个工人在前4个金矿的收益，和7个工人在前4个金矿的收益+最后一个金矿的收益谁大谁小了。

同样的道理，对于前4个金矿的选择，我们还可以做进一步简化。

首先针对10个工人4个金矿这个子结构，第4个金矿（300kg黄金/4人）可以选择挖与不挖。根据第4个金矿的选择，问题又简化成了两种更小的子结构：

1.10个工人在前3个金矿中做出最优选择。

2.6（10-4=6）个工人在前3个金矿中做出最优选择。

相应地，对于7个工人4个金矿这个子结构，第4个金矿同样可以选择挖与不挖。根据第4个金矿的选择，问题也简化成了两种更小的子结构：

1.7个工人在前3个金矿中做出最优选择。

2.3（7-4=3）个工人在前3个金矿中做出最优选择。

……

就这样，问题一分为二，二分为四，一直把问题简化成在0个金矿或0个工人时的最优选择，这个收益结果显然是0，也就是问题的边界。

 这就是动态规划的要点：确定全局最优解和最优子结构之间的关系，以及问题的边界。

这个关系用数学公式来表达的话，叫作状态转移方程式。

 好像有点明白了……那这个所谓的状态转移方程式是什么样子？

 我们把金矿数量设为n，工人数量设为w，金矿的含金量设为数组g[]，金矿所需开采人数设为数组p[]，设F（n，w）为n个金矿、w个工人时的最优收益函数，那么状态转移方程式如下：

F（n，w）=0（n=0或w=0）

问题边界，金矿数为0或工人数为0的情况。

F(n,w)=F(n-1,w)(n≥1,w<p[n-1])

当所剩工人不够挖掘当前金矿时，只有一种最优子结构。

F(n,w)= max(F(n-1,w),F(n-1,w-p[n-1])+g[n-1])(n≥1,w≥p[n-1])

在常规情况下，具有两种最优子结构（挖当前金矿或不挖当前金矿）。

img小灰，既然有了状态转移方程式，你能实现代码来求出最优收益吗？

img这还不简单？用递归就可以解决！

img

imgOK，这样确实可以得到正确结果，不过你思考过这段代码的时间复杂度吗？

img让我分析一下啊……全局问题经过简化，会拆解成两个子结构；两个子结构再次简化，会拆解成4个更小的子结构……就像下图一样。

img

img我的天哪，这样算下来，如果金矿数量是n，工人数量充足，时间复杂度就是O（2n）！

img没错，现在我们的题目中只有5个金矿，问题还不算严重。如果金矿数量有50个，甚至100个，这样的时间复杂度是根本无法接受的。

img啊，那该怎么办呢？

img首先来分析一下递归之所以低效的根本原因，那就是因为递归做了许多重复的计算，看看下面的图你就明白了。

img

在上图中，标为橘色的方法调用是重复的。可以看到F（2，7）、F（1，7）、F（1，2）这几个入参相同的方法都被调用了两次。

当金矿数量为5时，重复调用的问题还不太明显。金矿数量越多，递归层次越深，重复调用也就越多，这些无谓的调用必然会降低程序的性能。

img那我们怎样避免这些重复调用呢？

img这就要说到动态规划的另一个核心要点：自底向上求解。让我们来详细演示一下这种求解过程。

在进行求解之前，先准备一张表格，用于记录选择金矿的中间数据。

img

表格最左侧代表不同的金矿选择范围，从上到下，每多增加1行，就代表多1个金矿可供选择，也就是F（n，w）函数中的n值。

表格的最上方代表工人数量，从1个工人到10个工人，也就是F（n，w）函数中的w值。

其余空白的格子，都是等待填写的，代表当给出n个金矿、w个工人时的最优收益，也就是F（n，w）的值。

举个例子，在下图中绿色的这个格子里，应该填充的是在有5个工人的情况下，在前3个金矿可供选择时，最优的黄金收益。

img

下面让我们从第1行第1列开始，尝试把空白的格子一一填满，填充的依据就是状态 转移方程式。

对于第1行的前4个格子，由于w<p[n-1]，对应的状态转移方程式如下：

F(n,w)=F(n-1,w)(n>1,w<p[n-1])

带入求解：

F(1,1)=F(1-1,1)=F(0,1)=0

F(1,2)=F(1-1,2)=F(0,2)=0

F(1,3)=F(1-1,3)=F(0,3)=0

F(1,4)=F(1-1,4)=F(0,4)=0

第1行的后6个格子怎么计算呢？此时w≥p[n-1]，对于如下公式：

F(n,w)= max(F(n-1,w),F(n-1,w-p[n-1])+g[n-1])(n>1,w≥p[n-1])

带入求解：

F(1,5)=max(F(1-1,5),F(1-1,5-5)+400)=
max(F(0,5),F(0,0)+400)= max(0,400)=400

F(1,6)=max(F(1-1,6),F(1-1,6-5)+400)=
max(F(0,6),F(0,1)+400)= max(0,400)=400

……

F(1,10)=max(F(1-1,10),F(1-1,10-5)+400)=
max(F(0,10),F(0,5)+400)= max(0,400)=400

对于第2行的前4个格子，和第1行同理，由于w<p[n-1]，对应的状态转移方程式如下：

F(n,w)=F(n-1,w)(n>1,w<p[n-1])

带入求解：

F(2,1)=F(2-1,1)=F(1,1)=0

F(2,2)=F(2-1,2)=F(1,2)=0

F(2,3)=F(2-1,3)=F(1,3)=0

F(2,4)=F(2-1,4)=F(1,4)=0

第2行的后6个格子，和第1行同理，此时w≥p[n-1]，对应的状态转移方程式如下：

F(n,w)= max(F(n-1,w),F(n-1,w-p[n-1])+g[n-1])(n>1,w≥p[n-1])

带入求解：

F(2,5)=max(F(2-1,5),F(2-1,5-5)+500)=
max(F(1,5),F(1,0)+500)= max(400,500)=500

F(2,6)=max(F(2-1,6),F(2-1,6-5)+500)=
max(F(1,6),F(1,1)+500)= max(400,500)=500

……

F(2,10)=max(F(2-1,10),F(2-1,10-5)+500)=
max(F(1,10),F(1,5)+500)= max(400,400+500)=900


img

第3行的计算方法如出一辙。


img

再接再厉，计算出第4行的答案。


img

最后，计算出第5行的结果。


img

此时，最后1行最后1个格子所填的900就是最终要求的结果，即5个金矿、10个工人的最优收益是900kg黄金。


img好了，这就是动态规划自底向上的求解过程。


img哇，这个方式还真有意思！那么，怎么用代码来实现呢？


img在程序中，可以用二维数组来代表所填写的表格，让我们看一看代码吧。


img


img小灰，你说说上述代码的时间复杂度和空间复杂度分别是怎样的？


img程序利用双循环来填充一个二维数组，所以时间复杂度和空间复杂度都是O（nw），比递归的性能好多啦！


img是的，这段代码在时间上已经没有什么可优化的了，但是在空间上还可以做一些优化。

img想一想，在表格中除第1行之外，每一行的结果都是由上一行数据推导出来的。我们以4个金矿9个工人为例。

4个金矿、9个工人的最优结果，是由它的两个最优子结构，也就是3个金矿、5个工人和3个金矿、9个工人的结果推导而来的。这两个最优子结构都位于它的上一行。

所以，在程序中并不需要保存整个表格，无论金矿有多少座，我们只保存1行的数据即可。在计算下一行时，要从右向左统计（读者可以想想为什么从右向左），把旧的数据一个一个替换掉。

优化后的代码如下：

img哇，优化后的代码真的好简洁呀！

img是呀，而且空间复杂度降低到了O（n）。好了，关于金矿问题我们就讲解到这里，咱们下一节再会！

5.12 寻找缺失的整数

5.12.1 “五行”缺一个整数

img下面考你一道算法题：在一个无序数组里有99个不重复的正整数，范围从1到100……

题目

在一个无序数组里有99个不重复的正整数，范围是1～100，唯独缺少1个1～100中的整数。如何找出这个缺失的整数？

img哦，让我想想……

 有了！创建一个哈希表，以1到100这100个整数为Key，然后遍历数组。

解法1：

创建一个哈希表，以1到100这100个整数为Key。然后遍历整个数组，每读到一个整数，就定位到哈希表中对应的Key，然后删除这个Key。

由于数组中缺少1个整数，哈希表最终一定会有99个Key被删除，从而剩下1个Key。这个剩下的Key就是那个缺失的整数。

假设数组长度是n，那么该解法的时间复杂度是O（n），空间复杂度是O（n）。

 OK，这个解法在时间上是最优的，但额外开辟了内存空间。那么，有没有办法降低空间复杂度呢？

 哦，让我想想……

 有了！首先给原数组排序，然后……

解法2：

先把数组元素从小到大进行排序，然后遍历已经有序的数组，如果发现某两个相邻元素并不连续，说明缺少的就是这两个元素之间的整数。

假设数组长度是n，如果用时间复杂度为O（nlogn）的排序算法进行排序，那么该解法的时间复杂度是O（nlogn），空间复杂度是O（1）。

 OK，这个解法没有开辟额外的空间，但是时间复杂度又太大了。有没有办法对时间复杂度和空间复杂度都进行优化呢？

 哦，让我想想……

 有了！先算出1～100的累加和，然后再依次减去数组里的所有元素，最后的差值就是所缺少的整数。这么简单的办法我竟然才想到！

解法3：

这是一个很简单也很高效的方法，先算出1+2+3+…+100的和，然后依次减去数组里的元素，最后得到的差值，就是那个缺失的整数。

假设数组长度是n，那么该解法的时间复杂度是O（n），空间复杂度是O（1）。

imgOK，对于没有重复元素的数组，这个解法在时间复杂度和空间复杂度上已经最优了。但如果把问题扩展一下……

5.12.2　问题扩展

题目第1次扩展：

一个无序数组里有若干个正整数，范围是1～100，其中99个整数都出现了偶数次，只有1个整数出现了奇数次，如何找到这个出现奇数次的整数？

img哦，让我想想……

img按照刚才的方法先求和肯定不行，因为根本不知道每个整数出现的次数……同时又要保证时间复杂度和空间复杂度的最优，怎么办呢？

img让我提示你一下吧，你知道异或运算吗？

img

img异或运算，我当然知道，在进行位运算时，相同位得0，不同位得1。可是怎么应用到这个题目上面呢？

img啊，我想到了！只要把数组里所有元素依次进行异或运算，最后得到的就是那个缺失的整数！

解法：

遍历整个数组，依次做异或运算。由于异或运算在进行位运算时，相同为0，不同为1，因此所有出现偶数次的整数

都会相互抵消变成0，只有唯一出现奇数次的整数会被留下。

让我们举一个例子：给出一个无序数组{3，1，3，2，4，1，4}。

异或运算像加法运算一样，满足交换律和结合律，所以这个数组元素的异或运算的结果如下图所示：

假设数组长度是n，那么该解法的时间复杂度是O（n），空间复杂度是O（1）。

 这个方案已经非常好了。我们把问题最后扩展一下，如果数组里有2个整数出现了奇数次，其他整数出现偶数次，该如何找出这2个整数呢？

题目第2次扩展：

假设一个无序数组里有若干个正整数，范围是1～100，其中有98个整数出现了偶数次，只有2个整数出现了奇数次，如何找到这2个出现奇数次的整数？

 啊，这次要找2个整数，刚才的方法已经不够用了。因为把数组所有元素进行异或运算，最终只会得到2个整数的异或运算结果。

 我来提示你一下吧，你知道分治法吗？

 说起分治法，我似乎想到了什么……如果把数组分成两部分，保证每一部分都包含1个出现奇数次的整数，这样就与上一题的情况一样了。

 终于想到了！首先把数组元素依次进行异或运算，得到的结果是2个出现了奇数次的整数的异或运算结果，在这个结果中至少有1个二进制位是1。

解法：

把2个出现了奇数次的整数命名为A和B。遍历整个数组，然后依次做异或运算，进行异或运算的最终结果，等同于

A和B进行异或运算的结果。在这个结果中，至少会有一个二进制位是1（如果都是0，说明A和B相等，和题目不相符）。

举个例子，给出一个无序数组{4，1，2，2，5，1，4，3}，所有元素进行异或运算的结果是00000110B。

选定该结果中值为1的某一位数字，如00000110B的倒数第2位是1，这说明A和B对应 的二进制的倒数第2位是不同的。其中必定有一个整数的倒数第2位是0，另一个整数的倒 数第2位是1。

根据这个结论，可以把原数组按照二进制位的倒数第2位的不同，分成两部分，一部分的倒数第2位是0，另一部分的倒数第2位是1。由于A和B的倒数第2位不同，所以A被分配到其中一部分，B被分配到另一部分，绝不会出现A和B在同一部分，另一部分既没有A，也没有B的情况。

这样一来就简单多了，我们的问题又回归到了上一题的情况，按照原先的异或算法，从每一部分中找出唯一的奇数次整数即可。

假设数组长度是n，那么该解法的时间复杂度是O（n）。把数组分成两部分，并不需要 借助额外的存储空间，完全可以在按二进制位分组的同时来做异或运算，所以空间复杂度仍然是O（1）。

 没错，就是这个思路。请你按照这个思路来写一下代码。

 好的，我来试试！

 很好，我们的技术面试就到这里。请你稍等一下，我去叫HR来和你谈谈。

10分钟后……

就这样，小灰拿到了职业生涯中的第一个offer，但这并不意味着结束，小灰的程序 员之路才刚刚开始。

第6章　算法的实际应用

6.1　小灰上班的第1天

几天之后，小灰高高兴兴地去公司报到了……

就这样，小灰正式进入了职场。接下来等待他的会是什么样的挑战呢？

6.2　Bitmap的巧用

6.2.1　一个关于用户标签的需求

 为了帮助公司精准定位用户群体，咱们需要开发一个用户画像系统，实现用户信息的标签化。

 用户标签包括用户的社会属性、生活习惯、消费行为等信息，例如下面这个样子。

 通过用户标签，我们可以对多样的用户群体进行统计。例如统计用户的男女比例、统计喜欢旅游的用户数量等。

 放心吧，这个需求交给我一定会妥妥的！

为了满足用户标签的统计需求，小灰利用关系数据库设计了如下的表结构，每一个维度的标签对应着数据库表中的一列：

要想统计所有“90后”的程序员，该怎么做呢？

用一条求交集的SQL语句即可。

要想统计所有使用苹果手机或“00后”的用户总和，该怎么做呢？

用一条求并集的SQL语句即可。

 看起来很简单嘛，嘿嘿……

两个月之后……

 事情没那么简单，现在标签越来越多，例如，用户去过的城市、消费水平、爱吃的东西、喜欢的音乐……都快有上千个标签了，这要给数据库表增加多少列啊！

 筛选的标签条件过多的时候，拼出来的SQL语句像面条一样长……

 不仅如此，当对多个用户群体求并集时，需要用distinct来去掉重复数据，性能实在太差了……

6.2.2　用算法解决问题

 小灰，你怎么愁眉苦脸的呀？

 唉，还不是被一个需求折腾的！

 事情是这样子的……（小灰把工作中的难题告诉了大黄。）

 哈哈，小灰，你听说过Bitmap算法吗？在中文里叫作位图算法。

 我又不是搞计算机图形学的，研究位图算法干什么？

img这里所说的位图并不是像素图片的位图，而是内存中连续的二进制位（bit）所组成的数据结构，该算法主要用于对大量整数做去重和查询操作。

img举一个例子，假设给出一块长度为10bit的内存空间，也就是Bitmap，想要依次插入整数4、2、1、3，需要怎么做呢？

很简单，具体做法如下。

第1步，给出一块长度为10的Bitmap，其中的每一个bit位分别对应着从0到9的整型数。此时，Bitmap的所有位都是0（用紫色表示）。

第2步，把整型数4存入Bitmap，对应存储的位置就是下标为4的位置，将此bit设置为1（用黄色表示）。

第3步，把整型数2存入Bitmap，对应存储的位置就是下标为2的位置，将此bit设置为1。

第4步，把整型数1存入Bitmap，对应存储的位置就是下标为1的位置，将此bit设置为1。

第5步，把整型数3存入Bitmap，对应存储的位置就是下标为3的位置，将此bit设置为1。

如果问此时Bitmap里存储了哪些元素，显然是4、3、2、1，一目了然。

Bitmap不仅方便查询，还可以去掉重复的整数。

img看起来有点意思，可是Bitmap算法跟我的项目有什么关系呢？

img你仔细想一想，你所做的用户标签能不能用Bitmap的形式进行存储呢？

img我的每一条用户数据都对应着成百上千个标签，怎么也无法转换成Bitmap的形式啊？

img别急，我们不妨转换一下思路，为什么一定要让一个用户对应多个标签，而不是一个标签对应多个用户呢？

img一个标签对应多个用户？让我想想啊……

img我明白了！信息不一定非要以用户为中心存储，也能够以标签为中心来存储，让每一个标签存储包含此标签的所有用户ID，就像倒排索引一样！

第1步，建立用户名和用户ID的映射。

第2步，让每一个标签存储包含此标签的所有用户ID，每一个标签都是一个独立的Bitmap。

这样一来，每一个用户特征都变得一目了然。

例如，程序员和“00后”这两个群体，各自的Bitmap分别如下所示。

imgBingo！这就是Bitmap算法的运用。

img我还有一点不太明白，使用哈希表也同样能实现用户的去重和统计操作，为什么一定要使用Bitmap呢？

img傻孩子，如果使用哈希表的话，每一个用户ID都要用整型数据存储，少则占用4字节（32bit），多则占用8字节（64bit）。而一个用户ID在Bitmap中只占1bit，内存是使用哈希表所占用内存的1/32，甚至更少！

img不仅如此，Bitmap在对用户群做交集和并集运算时也有极大的便利。我们来看看下面的例子。

1.如何查找使用苹果手机的程序员用户

2.如何查找所有男性用户或“00后”用户

 这就是Bitmap算法的另一个优势——高性能的位运算。

 原来如此。我还有一个问题，如何利用Bitmap实现反向匹配呢？例如，我想查找非“90后”的用户，如果简单地做取反运算操作，会出现问题吧？

会出现什么问题呢？我们来看一看。

“90后”用户的Bitmap如下所示。

如果想得到非“90后”的用户，能够直接进行非运算吗？

显然，非“90后”用户实际上只有1个，而不是图中所得到的8个结果，所以不能直接进行非运算。

 这个问题提得很好，但是也不难解决，我们可以借助一个全量的Bitmap。

同样是刚才的例子，我们给出“90后”用户的Bitmap，再给出一个全量用户的Bitmap。最终要求出的是存在于全量用户，但又不存在于“90后”用户的部分。

如何求出这部分用户呢？我们可以使用异或运算进行操作，即相同位为0，不同位为1。

 我明白了，这真是个好方法！那么Bitmap的代码该怎么来实现呢？

imgBitmap的实现方法稍微有些难理解，让我们来看看代码。

img

img

在上述代码中，使用一个命名为words的int类型数组来存储所有的二进制位。每一个int元素控制其中的64位。（在Python3中，int既可以表示整型，也可以表示长整型。）

如果要把Bitmap的某一位设为1，需要经过两步。

1.定位到words中的对应的int元素。

2.通过与运算修改int元素的值。

如果要查看Bitmap的某一位是否为1，也需要经过两步。

1.定位到words中的对应的int元素。

2.判断int元素的对应的二进制位是否为1。

有了Bitmap的基本读写操作，该如何实现两个Bitmap的与、或、异或运算呢？感兴趣的读者可以思考一下。

img关于Bitmap，今天就介绍到这里，咱们下一节再见！

6.3 LRU算法的应用

6.3.1 一个关于用户信息的需求

img

img

img现在公司的业务越来越复杂，我们需要抽出一个用户系统，向各个业务系统提供用户的基本信息。

img

img业务方对用户信息的查询频率很高，一定要注意性能问题哦。

img放心吧，交给我，妥妥的！

用户信息当然是存放在数据库里，但是由于我们对用户系统的性能要求比较高，显然不能在每一次请求时都去查询数据库。

所以，小灰在内存中创建了一个哈希表作为缓存，每当查找一个用户时会先在哈希表中进行查询，以此来提高访问的性能。

很快，用户系统上线了，小灰美美地休息了几天。

一个多月之后……

img小灰，小灰，大事不好了！

img哦，出了什么事？

img线上服务器宕机了！

img让我看看……糟了，是内存溢出了，用户数量越来越多，当初设计的哈希表把内存给撑爆了，赶紧重启吧！

img可是以后该怎么办呢？我们能不能给服务器的硬件升级，或者加几台服务器呀？

img可是咱们公司没钱呀！

img那我能不能在内存快耗尽的时候，随机删掉一半用户缓存呢？

img唉，这样也不妥，如果删掉的用户信息，正好是被高频查询的用户，会影响系统性能的。

6.3.2　用算法解决问题

img小灰，你怎么日渐消瘦了啊？

img唉，还不是被一个需求折腾的！

img事情是这样子的……（小灰把工作中的难题告诉了大黄。）

img小灰，你听说过LRU算法吗？

img只听说过URL，没听说过LRU，那是什么？

imgLRU全称Least Recently Used，也就是最近最少使用的意思，是一种内存管理算法，该算法最早应用于Linux操作系统。

img这个算法基于一种假设：长期不被使用的数据，在未来被用到的概率也不大。因此，当数据所占内存达到一定阈值时，我们要移除最近最少被使用的数据。

img原来如此，这个算法正好对我的用户系统有帮助！可以在内存不够时，从哈希表中移除一部分很少被访问的用户。

img可是，我怎么知道哈希表中哪些Key-Value最近被访问过，哪些没被访问过？总不能给每一个Value加上时间戳，然后遍历整个哈希表吧？

img这就能展现LRU算法的精妙所在了。在LRU算法中，使用了一种有趣的数据结构，这种数据结构叫作哈希链表。

什么是哈希链表呢？

我们都知道，哈希表是由若干个Key-Value组成的。在“逻辑”上，这些Key-Value是无所谓排列顺序的，谁先谁后都一样。

img

在哈希链表中，这些Key-Value不再是彼此无关的了，而是被一个链条串了起来。每一个Key-Value都具有它的前驱Key-Value、后继Key-Value，就像双向链表中的节点一样。

img

这样一来，原本无序的哈希表就拥有了固定的排列顺序。

 可是，这哈希链表和LRU算法有什么关系呢？

 依靠哈希链表的有序性，我们可以把Key-Value按照最后的使用时间进行排序。

让我们以用户信息的需求为例，来演示一下LRU算法的基本思路。

1.假设使用哈希链表来缓存用户信息，目前缓存了4个用户，这4个用户是按照被访问的时间顺序依次从链表右端插入的。

2.如果这时业务方访问用户5，由于哈希链表中没有用户5的数据，需要从数据库中读取出来，插入缓存中。此时，链表最右端是最新被访问的用户5，最左端是最近最少被访问的用户1。

3.接下来，如果业务方访问用户2，哈希链表中已经存在用户2的数据，这时我们把用户2从它的前驱节点和后继节点之间移除，重新插入链表的最右端。此时，链表的最右端变成了最新被访问的用户2，最左端仍然是最近最少被访问的用户1。

4.接下来，如果业务方请求修改用户4的信息。同样的道理，我们会把用户4从原来 的位置移动到链表的最右侧，并把用户信息的值更新。这时，链表的最右端是最新被访问的用户4，最左端仍然是最近最少被访问的用户1。

5.后来业务方又要访问用户6，用户6在缓存里没有，需要插入哈希链表中。假设这时缓存容量已经达到上限，必须

先删除最近最少被访问的数据，那么位于哈希链表最左端的用户1就会被删除，然后再把用户6插入最右端的位置。

以上，就是LRU算法的基本思路。

 明白了，这真是个巧妙的算法！那么LRU算法怎么用代码来实现呢？

 虽然Python中的collections.OrderedDict已经对哈希链表做了很好的实现，但为了加深印象，我们还是自己写代码来简单实现一下吧。

需要注意的是，这段代码不是线程安全的代码，要想做到线程安全，需要加锁。

 小灰，对于用户系统的需求，你也可以使用缓存数据库Redis来实现，Redis底层也实现了类似LRU的回收算法。

 啊，你怎么不早说？我直接用Redis就好了，省得费这么大劲去研究LRU算法。

 千万不能这么想，底层原理和算法还是需要学习的，这样才能让我们更好地去选择技术方案，排查疑难问题。

 好了，关于LRU算法就介绍到这里，咱们下一节再会！

6.4 什么是A星寻路算法

6.4.1 一个关于迷宫寻路的需求

 公司开发了一款“迷宫寻路”的益智游戏。现在大体上开发得差不多了，但为了让游戏更加刺激，还需要加上一点新内容。

 我的天，咱们公司怎么什么都做呀？不过看起来很有意思呢！

 在这个迷宫游戏中，有一些小怪物要攻击主角，现在希望你给这些小怪物加上聪明的AI（Artificial Intelligence，人工智能），让它们可以自动绕过迷宫中的障碍物，寻找到主角的所在。

例如像下面这样。

 放心吧，交给我妥妥的！

三天之后……

 这个需求看起来简单，但是要做出聪明有效的寻路AI，绕过迷宫的所有障碍，还真不是一件容易的事情呢！

6.4.2　用算法解决问题

 小灰，你怎么最近下班这么晚啊？

 唉，还不是被一个需求折腾的！

 事情是这样子的……（小灰把工作中的难题告诉了大黄。）

 小灰，你听说过A星寻路算法吗？

 A什么算法？那是什么？

 是A星寻路算法！它的英文名字叫作A*search algorithm，是一种用于寻找有效路径的算法。

imgimg哇，有这么实用的算法？给我科普一下呗？

imgimg好吧，我用一个简单的场景来举例，咱们看一看A星寻路算法的工作过程。

imgimg

imgimg迷宫游戏的场景通常都是由小方格组成的。假设我们有一个7×5大小的迷宫，上图中绿色的格子是起点，红色的格子是终点，中间的3个蓝色格子是一堵墙。

imgimgAI角色从起点开始，每一步只能向上、下、左、右移动1格，且不能穿越墙壁。那么如何让AI角色用最少的步数到达终点呢？

imgimg哎呀，这正是我们开发的游戏所需要的效果，怎么做到呢？

imgimg在解决这个问题之前，我们先引入2个集合和1个公式。

两个集合如下：

- open_list——可到达的格子。
- close_list——已到达的格子。

一个公式如下：

- F=G+H

每一个格子都具有F、G、H这3个属性，就像下图这样。

imgimg

G：从起点走到当前格子的成本，也就是已经花费了多少步。

H：在不考虑障碍的情况下，从当前格子走到目标格子的距离，也就是离目标还有多远。

F：G和H的综合评估，也就是从起点到达当前格子，再从当前格子到达目标格子的总步数。

imgimg这些都是什么玩意儿？好复杂啊！

img其实并不复杂，我们通过实际场景来分析一下，你就明白了。

第1步，把起点放入open_list，也就是刚才所说的可到达格子的集合。

第2步，找出open_list中F值最小的方格作为当前方格。虽然我们没有直接计算起点方格的F值，但此时open_list中只有唯一的方格grid（1，2），把当前格子移出open_list，放入 close_list。代表这个格子已到达并检查过了。

第3步，找出当前方格（刚刚检查过的格子）上、下、左、右所有可到达的格子，看它们是否在open_list或close_list当中。如果不在，则将它们加入open_list，计算出相应的 G、H、F值，并把当前格子作为它们的“父节点”。

在上图中，每个格子的左下方数字是G，右下方是H，左上方是F。

img我有一点不明白，“父节点”是什么意思？为什么格子之间还有父子关系？

img一个格子的“父节点”代表它的来路，在输出最终路线时会用到。

img刚才经历的几个步骤是一次局部寻路的步骤。我们需要一次又一次重复刚才的第2步和第3步，直到找到终点为止。

下面进入A星寻路的第2轮操作。

第1步，找出open_list中F值最小的方格，即方格grid（2，2），将它作为当前方格，并把当前方格移出open_list，放入close_list。代表这个格子已到达并检查过了。

第2步，找出当前方格上、下、左、右所有可到达的格子，看它们是否在open_list或 close_list当中。如果不在，则将它们加入open_list，计算出相应的G、H、F值，并把当前格子作为它们的“父节点”。

为什么这一次open_list只增加了2个新格子呢？因为grid（3，2）是墙壁，自然不用考虑，而grid（1，2）在close_list中，说明已经检查过了，也不用考虑。

下面我们进入第3轮寻路历程。

第1步，找出open_list中F值最小的方格。由于此时有多个方格的F值相等，任意选择一个即可，如将grid（2，3）作为当前方格，并把当前方格移出open_list，放入close_list。代表这个格子已到达并检查过了。

第2步，找出当前方格上、下、左、右所有可到达的格子，看它们是否在open_list当中。如果不在，则将它们加入open_list，计算出相应的G、H、F值，并把当前格子作为它们的“父节点”。

剩下的操作就是以前面的方式继续迭代，直到open_list中出现终点方格为止。

这里我们仅仅使用图片简单描述一下，方格中的数字表示F值。

 像这样一步一步来，当终点出现在open_list中时，我们的寻路之旅就结束了。

 哈哈，还挺好玩的。可是我们怎么获得从起点到终点的最佳路径呢？

 还记得刚才方格之间的父子关系吗？我们只要顺着终点方格找到它的父亲，再找到父亲的父亲……如此依次回溯，就能找到一条最佳路径了。

 这就是A星寻路算法的基本思想。像这样以估值高低来决定搜索优先次序的方法，被称为启发式搜索。

 这种算法怎么用代码来实现呢？一定很复杂吧？

 代码确实有些复杂，但并不难懂。让我们来看一看A星寻路算法核心逻辑的代码实现吧。

 好长的代码啊，不过能勉强看明白。我要回去完善我的游戏了，嘿嘿……

6.5 如何实现红包算法

6.5.1 一个关于钱的需求

 “双11”快要到了，我们需要上线一个发放红包的功能。

这个功能类似于微信群发红包的功能。

例如，一个人在群里发了1个100元的红包，群里有10个人一起来抢红包，每人抢到的金额随机分配。

 哎呀，为什么我只抢到了2分钱呢？

 嘿嘿，只是举个例子啦。此外，我们的红包功能有一些具体规则。

红包功能需要满足哪些具体规则呢？

1.所有人抢到的金额之和要等于红包金额，不能多也不能少。

2.每个人至少抢到1分钱。

3.要保证红包拆分的金额尽可能分布均衡，不要出现两极分化太严重的情况。

这个简单，放心交给我吧！

为了避免出现高并发引起的一些问题，每个人领取红包的金额不能在领的时候才计算，必须先计算好每个红包拆出的金额，并把它们放在一个队列里，领取红包的用户要在队列中找到属于自己的那一份。

于是，小灰很快想出了一个拆分红包金额的方法。

小灰的思路是怎样的呢？具体如下所示：

每次拆分的金额= 随机区间 [1分，剩余金额 -1分]

举个例子，如果分发的红包是100元，有5个人抢，那么队列第1个位置的金额在0.01到99.99元之间取随机数。

假设第1个位置随机得到20元，队列第2个位置的金额要在0.01到79.99元之间取随机数。

假设第2个位置随机得到30元，队列第3个位置的金额要在0.01到49.99元之间取随机数。

假设第3个位置随机得到15元，队列第4个位置的金额要在0.01到34.99元之间取随机数。

假设第4个位置随机得到22元，那么第5个位置自然是35-22=13元。

小灰把做出的Demo演示给产品经理……

 哎呀，你这不行啊，这样随机的结果很不均衡！

 这不是挺好的吗？怎么不行了？

 如果以这样的方式来拆分红包的话，前面拆分的金额会很大，后面的金额会越来越小！

为什么这么说呢？让我们来分析一下。

假设红包总额为100元，有5个人来抢。

第1个人抢到金额的随机范围是[0.01，99.99]元，在正常的情况下，抢到金额的中位数是50元。

假设第1个人随机抢到了50元，那么剩余金额是50元。

第2个人抢到金额的随机范围就小多了，只有[0.01，49.99]元，在正常的情况下，抢到金额的中位数是25元。

假设第2个人随机抢到了25元，那么剩余金额是25元。

第3个人抢到金额的随机范围就更小了，只有[0，24.99]元，按中位数可以抢到12.5元。

以此类推，红包的随机范围将会越来越小，这样的结果一点也不公平，用户肯定要气得大骂了。

 说得也是啊……那如果我把随机的拆分金额打乱顺序放入队列呢？这样避免了先抢的用户占优势，后抢的用户吃亏。

 那也不行，虽然金额的顺序被打乱了，但金额的大小仍然是两极分化严重，最大的金额可能超过总额的一半，最小的金额会非常小。

6.5.2　用算法解决问题

 小灰，你怎么还不找个女朋友，工作太忙了吗？

 唉，还不是被一个需求给折腾的！

 事情是这样子的……（小灰把工作中的难题告诉了大黄。）

 小灰，关于红包拆分的问题，其实没有固定答案，稍微动动脑筋，就可以想出很多种高效又均衡的分配算法。

 有什么好的方法呢，你给举个例子呗？

 有一个最简单的思路，就是把每次随机金额的上限定为剩余人均金额的2倍。

方法1：二倍均值法

假设剩余红包金额为m元，剩余人数为n，那么有如下公式：

每次抢到的金额= 随机区间 [0.01，m/n×2-0.01]元

这个公式，保证了每次随机金额的平均值是相等的，不会因为抢红包的先后顺序而造成不公平。

下面举个例子。

假设有5个人，红包总额为100元。

100÷5×2=40，所以第1个人抢到的金额的随机范围是[0.01，39.99]元，在正常情况下，平均可以抢到20元。

假设第1个人随机抢到了20元，那么剩余金额是80元。

80÷4×2=40，所以第2个人抢到的金额的随机范围同样是[0.01，39.99]元，在正常的情况下，还是平均可以抢到20元。

假设第2个人随机抢到了20元，那么剩余金额是60元。

60÷3×2=40，所以第3个人抢到的金额的随机范围同样是[0.01，39.99]元，平均可以抢到20元。

以此类推，每一次抢到金额的随机范围的均值都是相等的。

这样做真的是均等的吗？如果第1个人运气很好，随机抢到39元，第2个人所抢金额的随机区间不就缩减到[0.01，60.99]元了吗？

img这个问题提得很好。第1次随机的金额有一半概率超过20元，使得后面的随机金额上限不足39.99元；但相应地，第1次随机的金额同样也有一半的概率小于20元，使得后面的随机金额上限超过39.99元。因此从整体来看，第2次随机的金额平均范围仍然是 [0.01，39.99]元。

img原来如此，那么代码怎么实现呢？

img代码非常简单，让我们来看一看。

img

img明白了，还真是个好办法！

img这个方法虽然公平，但也存在局限性，即除最后一次外，其他每次抢到的金额都要小于剩余人均金额的2倍，并不是完全自由地随机抢红包。

img哦，那怎样能做到既公平，又不超过总金额，又能提高随机抢红包的自由度呢？

img有另一种方法，我们姑且把它叫作线段切割法吧。

方法2：线段切割法

何谓线段切割法？我们可以把红包总金额想象成一条很长的线段，而每个人抢到的金额，是这条主线段所拆分出的若干子线段。

img

如何确定每一条子线段的长度呢？

由“切割点”来决定。当n个人一起抢红包时，就需要确定n-1个切割点。

因此，当n个人一起抢总金额为m元的红包时，我们需要做n-1次随机运算，以此确定n-1个切割点。随机的范围区间是[0.01，m-0.01]。

当所有切割点确定以后，子线段的长度也随之确定。此时红包的拆分金额，就等同于每个子线段的长度。

这就是线段切割法的思路，在这里需要注意以下两点：

1.当随机切割点出现重复时，如何处理。

2.如何尽可能降低时间复杂度和空间复杂度。

img关于线段切割法，我们就不写具体代码了，有兴趣的读者可以尝试一下。此外，实现红包拆分的算法肯定不止这两种，聪明的读者可以开动脑筋，想一想还有没有更好的选择。

img好了，关于红包算法我们就介绍到这里，祝愿大家每次抢红包时都能拥有好手气！

6.6 算法之路无止境

就这样，小灰继续在算法的世界中摸索、前进着，这个世界充满了新奇，也同样充满了挑战。

尽管小灰学到了许多东西，但小灰仍然保持着一颗求索的心。因为小灰明白，算法之路，永无止境……

欢迎关注微信公众号“程序员小灰”

在这个公众号里，你有如下福利：

- 在后台回复“漫画算法”，获得全书完整的、可运行的代码。
- 阅读作者更多生动有趣的原创漫画。
- 参加不定期抽奖、赠书、购书优惠券等活动。
- 参加作者发起的各种有趣的大赛。

读者如果在阅读过程中产生疑问或发现Bug，欢迎随时到微信公众号的后台留言。