长河乱语 一

OTTF 2023 3 11 起写

2023 7 31 终写

第0章 引子

一条大河奔腾着，何乐乌拿着一块板子，领着沃柔德来到了石桌石凳边。她笑着指指那河，坐了下来。沃柔德看着那条河，她的眼睛像一片玻璃一样闪烁了一会儿，然后也坐下。沃柔德向何乐乌问道：“很澎湃的地方！我爱这里！那个——你说想对我说一些话，能说了吗？”

何乐乌转过头来说道：“啊，亲爱的挚友。你知道的，我原是大学的打杂人员，同时依靠在算法竞赛网站上获取比赛奖金度日。而你是一名自由的画家，说真的，我羡慕你的职业——艺术与美，美与艺术——扯远了，嗯哼。但由于机缘巧合，我将要成为一名教授计算机科学的老师——太奇妙了！所以我打算尝试拿你做样例，我想试一试你能不能理解我说的话。”

沃柔德忍俊不禁。她又说道：“好的，那么——你的主题是什么呢？”

何乐乌说道：“嗯……就从我比较熟悉的、基础的东西开始吧，然后——还是在我熟悉的领域尝试，我不想讨论关于电线和内存什么的问题——我不喜欢它们！嗯哼。我爱算法，所以，聊聊算法吧。”

沃柔德看着何乐乌，何乐乌眨眨眼，拿着笔开了口。

第1章 时空复杂度

> 滚滚长江东逝水，
>
> 九曲黄河万里沙。
>
> 大风起兮云飞扬，
>
> 满城尽带黄金甲。
>
> ——自杨慎《临江仙·滚滚长江东逝水》、刘禹锡《浪淘沙其一》、刘邦《大风歌》、黄巢《不第后赋菊》这四首古诗拼凑而来。

我们说的是算法，显然的，是“算数的方法”。你可以认为它是对一些数据进行奇奇怪怪的操作，从而求解奇奇怪怪的问题的流程。生活之中，我们有很多流程，比如烧一壶水的流程，走路上学的流程，还有堆放和焚烧垃圾的流程——啊呀！我太爱关于垃圾的流程了，不如我们聊聊它。

当你手上有一袋垃圾的时候，你应该如何处理它呢？一般来说，你会把它扔到垃圾桶里。而当你手上有一百一十四袋垃圾的时候，你应该如何处理他们呢？假设垃圾桶足够大，而你只有一双手，你应该一个接一个，把这些垃圾全部扔进垃圾桶里。

那么，变换一下问题，你是垃圾焚烧厂的员工，现在成千上万的垃圾在焚烧坑里，而你的旁边是点火按钮，只要按下按钮，火焰会瞬间吞噬垃圾们，你应该如何处理垃圾呢？你应该按下按钮。

发现了吗？当你需要一袋一袋投掷垃圾时，你的工作量因垃圾数量的增多而增多。而当你只需按下按钮就能感叹火焰的强大时，你的工作量永远只有按下按钮。所以，你的工作量会根据所需要解决的问题的不同而有所不同。

那么，如果所需要处理的事情相同，工作量可能会不同吗？是的，确实可能。如果你选择用打火机一袋一袋烧掉垃圾，那你的工作量就随着垃圾的增多而增多了。所以，你的工作量是由你的流程而决定的。对于相同的问题，可能会有不同的解决流程。显然，我们更倾向于选择工作量少的解决流程。

这跟算法又有什么关系呢？让我们举一些与算数有关的例子。如果你有一排数，共n个，而你现在需要计算这些数的和，你会怎么做呢？你应该看一遍这n个数，把他们加起来。我们在编程中会让计算机做这种事。

而如果你需要得到最后一个，也就是第n个数的值，那么直接查看第n个数就好了。不过，你也可以看一遍所有数，得到所有数的值，然后记下最后那一个的。你一般会采用

第一种方法，而我们一般也会让计算机采用第一种方法。

诶？所以我们为什么要在计算机编程中考虑工作量呢？计算机可是不会累的啊？嗯哼。计算机不会累，但人类会不耐烦。虽然计算机很快，但随着工作量的增加，它工作的时间也会增加。而当计算机完成一个任务需要的时间较长时，人类就不高兴了。比如说，在很多算法竞赛中，一个程序运行超过一定时间就算作超时，这个“一定时间”一般是一秒。总而言之，所以我们需要使用更快的算法。

而为了比较各个算法，我们需要尝试表示算法带来的工作量，人们使用“时间复杂度”来比较算法的工作量。我们现在认为n代表一个“数据规模”，比如对于一排数，它的个数就是“数据规模”。假设算法的问题规模为n，那么它所需要的“基本操作”的数量便用$f(n)$来表示。“基本操作”可以是一次查看、一次比较、一次四则运算，还有等等等等。

“$f(n)$”是一个“函数”。函数是个好用的东西，你可以认为函数的意义是：我们在一个或多个东西的外面加上括号，然后在括号前面添上一个名字，表示我们对这个东西做了奇奇怪怪的事情，并得到最终的结果。比如说，对于a与b这两个东西，我们想得到他们中比较大的一个，于是我们就可以写“$\text{比较大}(a,b)$”。如果$a=1$而$b=0$，那么显然$\text{比较大}(a,b)=\text{比较大}(1,0)=1$。

随着数据规模n的增大，算法执行时间的增长率和$f(n)$的增长率相同，这称为算法的时间复杂度，记为$O(f(n))$。你可以认为这里的字母“O”是一个奇奇怪怪的代号。比如说，计算一排n个数的和的复杂度是$O(n)$，又称“线性复杂度”。因为我们需要查看n次，做n次加法。

稍等，那么不是有$2n$次“基本操作”吗？好吧，在时间复杂度的计算中，我们忽略$f(n)$的常数。实际上如果$f(n)$是个多项式，我们一般还会忽略小的几项。比如我们一般不写$O(n+1)$，我们只写$O(n)$。

还有更多时间复杂度的例子吗？有的。比如查看一列数的最后一个数的时间复杂度是$O(1)$，我们用$O(1)$表示这个算法所需的时间不因数据规模的变化而变化。而对于一列数，得知每个数与所有其他数的大小情况的时间复杂度是$O(n^2)$，因为我们需要把n个数与其他$(n-1)$个数进行比较。

除了时间复杂度，还有更多复杂度吗？嗯哼，确实有。空间复杂度。在一个算法中，临时占用的空间的增长率与$f(n)$的增长率相同，所以类比时间复杂度，我们也用$O(f(n))$代表空间复杂度。比如$O(1)$啊，$O(n)$啊。我喜欢把时间复杂度和空间复杂度合称“时空复杂度”，这显得我很睿智、聪颖。

第2章 排序与伪代码

> 这是哪个没有语言中枢的蠢蛋写的代码？！
>
> 哦，是两周前的我。
>
> ——改编自互联网上流传已久的笑话。

嗯哼。那么，从现在开始，让我们开始讨论具体的算法。我打算先从“排序”这个问题入手，同时旁生许多枝茎。

“排序”，就是要排列很多东西的顺序。比如说，我们将方阵中的人从低到高排列，将学生考试的分数从高到低排列。显然，我们会处理很多这样的问题，于是我们需要算法来用计算机解决这些问题。

一般而言，我们要排列的东西都跟数值有关，比如从低到高排列人，本质上就是把每个人的身高排列。而每个人的身高都是一个数字。对于考试分数的例子，也是这个道理。所以，我们不妨把要解决的任务定为：对于一列数，把他们从小到大进行排序。当然实际上从小到大也可以，我们只是举例。

说到这里，让我们来完成之前就提到的“一列数”的定义。“一列数”实际上是“一列元素”的特殊版本。一列元素类似数学意义上的数列，指的是在众多编程语言中的概念“数组”。在一列元素中，存放着一个个相同的元素类型。元素是单个的一个东西，比如一个数，这类似某些编程语言中的概念“变量”。每个元素都有一个“序号”，对于有n个元素的一列元素，我们——为了方便下面的讲述，违背我的信仰，不从0到$n-1$而是——从1到n进行编号。根据序号查看对应的元素是$O(1)$的，在末尾增加或删除元素也是$O(1)$的，而在不是末尾的地方删除或插入元素就需要让后面的元素一个个前进或后退，是$O(n)$的。

我们应该如何对一列数排序呢？想一想，在排完序的一列数中，1号是最小的，2号是次小的，3号是次次小的……好吧，我没有考虑两个数相等的情况，但至少每个数都不比后面的数大。既然这样，我有一个大胆的想法：

在刚开始排序时，找出一个最小的数，把它放到最前面，不再管它。然后对于剩下的部分，找出最小的数，放到剩下的部分的最前面。一直循环下去，直到没有“剩下的部分”。我们怎么把一个数“放到最前面”呢？只需要交换“最前面”与这个数就好了，这是$O(1)$的。而我们怎么找出最小的数呢？在范围内看一遍就好了。

我们可以用介于编程语言和人类语言之间的“伪代码”描述这一操作：

```
朴素排序（一列数nums，整数n为其长度）：
    i从1到n循环：
        id为0
        num为INF
        j从i+1到n循环：
            如果nums[j]小于num：
```

```
            id为j
            num为nums的第j项
        交换（nums[i], nums[num]）
```

很奇怪，对吗？我点了一下我的平板，它就显示了串奇怪的文字——像人话又不像人话。那么，我来从头到尾解说一边：

第一行的“朴素排序”是我们的算法的名字，我们喜欢给算法起名。由于我们的方法十分朴素，我们把它称为“朴素排序”。名字后面是一对括号，里面包裹着算法所需要的“参数”。当我们使用一个算法时，我们需要把一些参数传入这个算法，比如在我们的排序例子中，我们传入一列数，同时为了方便，传入这列数有多少个数。

在这里，这列数的名字为nums，这是英文单词“number”的复数形式“numbers”的缩写。给除了算法外的东西起名时，我依照数学上的习惯，使用键盘上能直接打出的英文字母。nums的长度为n，这表示它有n个数，n在这里是一个元素，即变量。

你会发现这一行的末尾有一个冒号，而下一行的开头莫名其妙多出了四个空格，我这么做从而表示这个算法所包含的内容。事实上，这一行往下的每一行的开头都有四个空格，这表示这部分内容都是这个算法中的。对于所有需要表示代码包含关系的地方，我都采用这种方法。

下一行，开头的i也是一个变量，而从1到n循环则代表着一个“循环”。有时候我们需要重复做一些工作很多次，没准儿还要随时知道现在做到第几次了，这便是循环的作用。在此处，我们用这个句子表示：i的值一开始被设为1，然后执行这个循环包含的内容，执行完后，把i的值加1。如果在一次执行加1后，执行包含的内容前，我们发现i超过了n，也就是达到了$n+1$时，结束这个循环。

接下来的两行，每行开头有八个空格。每行中，四个表示它们在这个函数中，另外四个表示它们在循环中。后面几行的空格数量也就不用再提了。在这两行中，我用“为”字设立id与num两个变量。“为”表示将后面的东西赋给前面的东西，在我们的伪代码，如果前面这个东西从未出现过，那么直接设立它。id是“序号”的意思，初始值为0。而num的初始值为INF，我用INF表示“无穷大”，没有什么比它大。

以j为变量的那行循环开头不必再说，直接来看如果nums[j]小于num吧。在这种句式下，如果“如果”后的事情是真的，我们就执行包含着的语句，反之，则不执行，这叫做“选择”。nums[j]是“nums这列数的第j个”的意思。左右方括号是我在伪代码中使用的唯二的非中文标点符号。

最终的交换（nums[i], nums[num]）使用了一个名为“交换”的函数，它的作用是把两个参数的值互换，我不想赘述它的原理。嗯哼。

我们可以用一行`朴素排序（heights，len）`来调用这个排序方法，`heights`代表需要排序的一列代表身高的数，`len`代表`heights`中数的个数。当然，这只是一个例子，只要你有一个想要排序的数列，并且有它的长度，你就可以这样调用。

我刚刚介绍的“伪代码”只是我使用的“伪代码”，实际上每个人都可能有自己风格的“伪代码”，你可以……嗯……先习惯我这种风格的！

第3章 朴素、冒泡、插入

> 小的时候，游戏的种类很多，其中我最爱玩的是吹肥皂泡。
>
> ——冰心《肥皂泡》

好吧，我或许说了太多的话。让我们重新审视刚刚的伪代码：

```
朴素排序（一列数nums，整数n为其长度）：
    i从1到n循环：
        id为0
        num为INF
        j从i+1到n循环：
            如果nums[j]小于num：
                id为j
                num为nums的第j项
        交换（nums[i]，nums[num]）
```

可以看见，我循环了n次，每一次从尚未排序的一个元素开始，往后查找，找出最小的一个，最后把它通过排序的方法放到尚未排序的元素的最前面。这个算法的时间复杂度是$O(n^2)$。

为了找出最小的元素，我们使用了`id`来记录最小元素是第几个，还用了`num`记录最小值的值。细细想，我们好像不需要这两个变量，我们可以每找到一个比尚未排序的第一个元素小的元素，就把尚未排序的第一个元素与找到的元素交换，即这样：

```
朴素排序（一列数nums，整数n为其长度）：
    i从1到n循环：
        j从i+1到n循环：
            如果nums[j]小于nums[i]：
                交换（nums[i]，nums[j]）
```

代码一下子就变得清爽了许多。

而我们还有别的排序方法吗？当然有的，这里就还有一种$O(n^2)$的排序方法，叫做冒泡排序。我打算直接在这里贴上它的伪代码。

```
冒泡排序（一列数nums，整数n为其长度）：
    i从1到n循环：
        j从1到n-i循环：
            如果nums[j]大于nums[j+1]：
                交换（nums[j]，nums[j+1]）
```

这肯定给你带来了巨大的冲击！让我们好好看看这段代码：算法的名字叫“冒泡排序”，好怪异的名字，也不知道跟泡泡有什么关系。第一层循环的开头跟我们的朴素排序没有什么不同，看向下一行——嗯……`j`在朴素排序中是从`i+1`增加到`n`，而在冒泡排序中则是从`1`增加到`n-i`。

作个差，你会发现这两个循环的循环次数是相同的。既然`j`是`nums`的元素的一个序号的话，我们可以认为`j`到了某个值并开始一次执行就是“遍历”到了`nums`的第某个元素。在朴素排序中，我们的`j`没有遍历到`num`的前`i`个元素，而在冒泡排序中，`j`没有遍历到`num`的后`i`个元素。

这两个函数的循环次数相同，而且都能对数组排序，那么它们的基本逻辑是不是类似的？是不是呢？嗯？

是的！朴素排序每进行一边遍历都确定了一个值：第一次，它把最小的放在第一位，确定了第一个值，后面如是。而冒泡排序每次也确定了一个值，但正好相反，它在第一次把最大的放在了最后一位，确定了最后一个值。比如说，对以下的数列，我们第一次遍历它时：

```
3 1 5 2 4
交换第一个与第二个
1 3 5 2 4
不交换第二个与第三个
1 3 5 2 4
交换第三个与第四个
1 3 2 5 4
交换第四个与第五个
1 3 2 4 5
```

我们将最大的数放到了最后面。然后我们继续一次次遍历，比较大的数被放到后面，这在童心未泯的计算机科学家们看来，就像肥皂泡在冒出来一样！于是这种排序方法被称为“冒泡排序”。

而“朴素排序”这个名字是谁起的？是我随便起的。如果你想的话，你也可以称它为“粉红长毛兔跳跃排序”，因为它就像是让比较小的兔子跳到前面一样。

还有一种类似冒泡排序的方法：

```
插入排序（一列数nums，整数n为其长度）：
    i从2到n循环：
        j从i到2循环：
            如果nums[j-1]大于nums[j]：
                交换（nums[j-1]，nums[j]）
            否则：
                退出循环
```

哦我的天哪，i竟然从2而不是1开始！j竟然从i到2！这种排序方法与整理扑克牌的方式类似：

你的手上握着几张牌，首先看向第一张——不动它，把它放在那里。在我们的伪代码中，这体现为i的循环不从1开始。然后第二张牌，如果它比第一张要大，那么也不用动。如果它比第一张小，那么把它放在第一张牌前面。然后第三张牌，我们要把它放在一个地方，使得它前面那张牌——如果有的话——比它小，后面那张牌——如果有的话——比它大。

推而广之，在处理每一张牌时，前面的牌是我们刚刚排过的，所以我们可以扫一眼，发现一个比要处理的牌小的牌，就把要处理的牌插到它后面。在伪代码中，我们发现它比上一张牌小，就交换它与上一张牌。那否则又是什么呢？上面有一个如果，否则指的便是“如果如果那儿的条件不成立，就按我这里搞”。而在这里，如果nums[j-1]并未大于nums[j]，而是比它小，就退出循环。退出循环指的是最深层的那个循环，在这里，指j的那个。为什么要退出呢？因为我们已经把这个数放好了。

我们发现，j既然是从i到2，便是会慢慢减小的，这是可行的。顺便补充一句，我们在很多编程语言中，也可以写出类似这样的循环：

```
i从0到20循环，每次增加3：
    ……
```

在这时，i一开始是0，然后是3，再然后是6……一切都按照每次增加3的指令而井井有条。最后，当它是18的时候，如果它增加3，那它就会超过20了，只是超过了1啊！可恶啊！是的，我们的i在这种情况下，便不到20就结束循环了——因为它永远也不能比它的终点大，最好的情况也只是刚好到达。

这种每次增加的设定，有时被称为“步长”。

刚刚讲述的排序算法，被称为“插入排序”。不得不说，计算机科学家先辈们真的特别擅长起名字——至少给这个算法起名的先辈们擅长。插入排序与冒泡排序都是经常使用的排序算法，但插入排序的应用貌似更广泛，因为它更直观。而且由于有个“退出机制”——就是那个退出循环，插入排序有时更快。但我更喜欢冒泡排序，所以就先讲了它——这美丽的泡泡！朴素排序是我瞎想的，忘了它吧。

第4章 斐波那契

> 斐波那契霜满天，
>
> 月落乌啼来算钱。
>
> 两只兔子生一对，
>
> 马上占满大草原。
>
> ——点击洛谷P1720了解更多。

排序的方法有很多，但是我决定暂且按下不表。让我来讲一个老套的故事：

一片草原，气候温和湿润，有丰富的多汁牧草，适合发展畜牧业。于是为了将自然资源转变为经济资源，我决定在这里饲养兔子。

第一年，我带着一对小兔定居在这里。这对小兔一公一母，生性活泼。我让他们生活在一起，培养它们的感情。在这一年，我有一对兔子。

第二年，这对小兔变成了大兔。在情意绵绵之中，母兔怀孕了。是的，只有大兔才能怀孕。在这一年，我有一对兔子。

第三年，这对大兔生下了一对小兔。对于刚生下来的这对小兔，我还是准备培养它们的感情，把它们放在一起——虽然近亲繁殖并不好，但……这都是为了科学！在这一年，我有两对兔子。

对于一对兔子来说，在刚出生的那年，它们是小兔。在下一年，它们会怀孕，具备生殖能力。从第三年开始的每一年，它们都会生下一对小兔，并且还是一公一母。以此推算，第四年，我有三对兔子，分别是第三年就有的两对和大兔新生的一对。第五年，我有五对兔子，分别是第四年就有的三对和大兔新生的两对。我们假设虽然近亲繁殖不太好，但兔子都是“完美的兔子”，永远不会死亡。

这五年来，我拥有的兔子对数分别是1、1、2、3、5。看起来很平常的五个数，不是吗？但细细一看，你会发现，除了前两个数以外，每个数都等于它前面两个数的和。比

如5就是3与2的和。

这是巧合吗？好像是。但当发现这样的事时，我们应该思考一下——这是一种规律吗？在第六年，除了第五年就有了的五对兔子之外，我还得到了三对大兔所生的三对小兔。也就是说，在第六年，我有八对兔子——符合我们发现的规律！

这是为什么呢？你估计在我不厌其烦的重复中发现了，如果真的，那么恭喜你！嗯嗯。这是因为对于从第三年开始的每一年，这一年我拥有的兔子可以分为两个部分：一个部分是上一年就有的，另一个部分是大兔新生的。而去年的大兔数量等于前年的兔子总量，因为千年的所有兔子在去年都一定是大兔。所以，将两个部分相加，今年的兔子总量就等于去年的加前年的。

而兔子数量形成的数列，从第三项开始，每一项也就等于前两项之和。这个数列被称为“斐波那契数列”，因为这个故事是斐波那契讲出来的。

斐波那契数列有很多性质，比如跟“黄金比例”的关系什么的，这里不再赘述。我们考虑一个现实的问题——现在，把这个数列的第n项称为“第n个斐波那契数”，如何用计算机算出第n个斐波那契数呢？

我们可以使用一种叫做“递归”的有趣思想：

```
斐波那契（一个整数n）：
    如果n小于3：
        返回 1
    否则：
        返回 斐波那契（n-1）+ 斐波那契（n-2）
```

首先，在介绍递归之前，先来看看“返回”这个词。之前说过，所谓“函数”是我们对某东西做某种事并得到结果的记号，那么“得到结果”这个行为，就需要“返回”一个东西，从而让我们可以使用它。比如之前的比较大(a,b)，如果用我们的伪代码书写，可能会是这样：

```
比较大（一个数a，一个数b）：
    如果a大于b：
        返回 a
    否则：
        返回 b
```

如果一个函数会返回某种东西，我们就可以把它写到式子里，比如：

```
一个数a为114，一个数b为514
一个数c为比较大（a，b）
```

所以显然的，第1个与第2个斐波那契数都是1，所以如果n小于3，就可以直接返回1。那么返回 斐波那契（n-1）+ 斐波那契（n-2）这句话又是什么意思呢？

“返回”意味着我们要返回后面的那个值，而这个值是……斐波那契（n-1）与斐波那契（n-2)的和……而这段话是在斐波那契这个函数中编写的。所以说，我们在一个函数中调用了这个函数，这是可行的吗？

还真可行！这类似俄罗斯套娃的操作被称为“递归”！真是个好听的时髦词！运用递归，我们可以完成许多任务，虽然方法乍一看略显怪异，但细细思考则醍醐灌顶。在斐波那契的这个例子里，如果我们要计算不是第1或第2个斐波那契数，我们则转而去计算它的前两项，然后转转转转转，总能转到第1或第2项的。由于我们特殊处理了第1和第2项，所以我们总能得到结果。

但这种方式有个问题——太慢。想象一下，如果我们要计算第100个斐波那契数，我们会先去计算第99个和第98个。对于第99个，我们会去计算第98和第97个……看见了吗？这样下去，我们会经历很多很多迭代，从而用去大量的时间。

问题在哪里呢？问题在于，我们会重复计算同样的东西。比如说，在上面的例子里，即使我们已经计算了第98个斐波那契数，当我们再次用到它时，我们还是需要进行计算。所以，我们可以用一个东西记录下每一个斐波那契数——就用数组吧。

```
一列数fibo，一开始都为0
斐波那契（一个整数n）：
    如果fibo[n]等于0：
        如果n小于3：
            fibo[n] 为 1
        否则：
            fibo[n] 为 斐波那契（n-1）+ 斐波那契（n-2）
    返回 fibo[n]
```

我们使用了fibo这列数，在一开始，其中每一个数都是0。然后，当我们计算某一个斐波那契数时，若我们计算过它，fibo的这一项将不是0，我们可以直接用，这会大大减少我们所需的时间。若我们没有计算过它，则我们先计算一下，然后将得到的值赋给fibo的这一项。最终，我们会返回fibo的这一项。

在这个例子里，我们用一个数组节省了大量的时间。也就是说，我们通过占用内存减少了时间复杂度，这用“黑话”来说，是“以空间换时间”。

刚才使用一个fibo数组来记住结果的做法，被称为是“记忆化”。“记忆化”通常可以得到进一步的修改，从而直接不用进行递归：

既然我们想要计算一个斐波那契数，就会用fibo，记下它以及它之前所有的斐波那契数，那我们或许可以从最小的斐波那契数开始处理，直到我们需要的那个。

```
一列数fibo，一开始前2个为1，其他都为0
斐波那契（一个整数n）：
    对于i从3到n：
        如果fibo[i]等于0：
            fibo[i] 为 fibo[i-1] + fibo[i-2]
    返回 fibo[n]
```

这样，操作时，从最小的一直推到最大的，我们称为“递推”。

借助斐波那契，我们了解了递归与递推两种思想。递推暂且不过多叙述，而在众多排序算法中，使用递归思想的，确实有。

第5章 归并排序

> 向左转，两队变一队。
>
> ——体育课代表

在使用递归思想进行排序前，让我们再回顾一下递归而没有记忆化的斐波那契数计算方法：

```
斐波那契（一个整数n）：
    如果n小于3：
        返回 1
    否则：
        返回 斐波那契（n-1）+ 斐波那契（n-2）
```

当我们使用递归的思想处理问题时，我们笃定地认为，我们现在处理的问题可以由“更小”的问题“转化”——或称“化归”——而来。重点是“更小”的问题。对于一个问题，我们需要找到一个合适的“更小”的问题。对于求第n个斐波那契数，我们找到了求第$n-1$个斐波那契数和求第$n-2$个斐波那契数这两个问题。

那么，对于排序一列数这个问题，“更小”的问题是什么呢？在斐波那契数的例子里，我们把n变成了$n-1$和$n-2$，看起来在数字上做手脚是个好方法。那么，让我们试试……

排序“半列数”？

呃……我的意思是……把一列数分成左右两半，分别排序这两半，然后再把它们组合起来。“分成左右两半”好像有些难办，但显然，我们可以用“最左边的位置”和“最右边的位置”来表示数列的一段。所以我们可以写出这样的伪代码：

```
归并排序（一列数nums，l为我们处理的部分的最左边的位置，r为我们处理的部分的最右边的位置）：
    m为向下取整(l+r)/2
    归并排序（nums，l，m）
    归并排序（nums，m+1，r）
    ……
```

这段代码显然是没有完成的，省略号代表我们还没有进行的操作——合并已经完成排序的左右两边。现在让我们借助`314 159 265 358`这列数，来模拟一下归并排序的递归调用。

最初，`l`为`1`，`r`为`4`，因为在最开始，我们需要处理整个数组。然后，我们计算出`m`，`m=向下取整（(1+4)/2）=向下取整（2.5）=`……`向下取整`是个什么函数？好吧，`向下取整`指的是，对于一个数，得到最小的大于等于它的整数。比如`向下取整（2.5）=2`。也就是说，`m`在这里是`2`。

接下来的事情就明了了！我们先后调用`归并排序（nums，l，m）`和`归并排序（nums，m+1，r）`，这代表分别对数组的左边和右边排序。在这个例子里，分别是`314 159`这一段和`265 358`这一段。

稍等！我们递归貌似没有止境！一眼就可以看出，我们应当在处理的部分只剩一个数甚至连一个数也没有时直接结束这次处理。也就是说，我们应该：

```
归并排序（一列数nums，l为我们处理的部分的最左边的位置，r为我们处理的部分的最右边的位置）：
    如果l大于等于r：
        返回

    m为向下取整（(l+r)/2）
    归并排序（nums，l，m）
    归并排序（nums，m+1，r）
    ……
```

那么接下来，就只剩一件事了：如何把排完序的左右两边合并到一起呢？让我们想一想：排完序的左右两边都是从小到大的，而我们的最终要求是一整个从小到大的数列，所以……每次选择左右两边第一个数的最小一个，然后把它挪移到一个临时数列的末尾。如此一直进行，直到左右两边中的一个被挪完，便把剩下的数一次放到临时数组的末尾。最终让处理的部分的数字等于临时数组中的。也就是：

```
合并（一列数nums，l为我们处理的部分的最左边的位置，r为我们处理的部分的最右边的位置）：
    一列数temp，一开始什么都没有
    m为向下取整（(l+r)/2）
    i为l，j为m+1
    重复：
        如果nums[i]小于nums[j]：
            添加nums[i]到temp末尾
            i为i+1
        否则：
            添加nums[j]到temp末尾
            j为j+1

        如果i大于m：
            k从j到r循环：
                添加nums[k]到temp末尾
            退出循环
        如果j大于r：
            k从i到m循环：
                添加nums[k]到temp末尾
            退出循环

    k从1到l-r+1循环：
        nums[l+k-1]为temp[k]
```

虽然与我刚才的口述略有不同——我们还是不常用计算机搞“挪移”的——但它确实能“合并”数组的两个部分。最后一个循环涉及到一些数学运算，你可以琢磨琢磨。最终，我们只需要：

```
合并（一列数nums，l为我们处理的部分的最左边的位置，r为我们处理的部分的最右边的位置）：
    一列数temp，一开始什么都没有
    m为向下取整（(l+r)/2）
    i为l，j为m+1
    重复：
        如果nums[i]小于nums[j]：
            添加nums[i]到temp末尾
            i为i+1
        否则：
            添加nums[j]到temp末尾
            j为j+1

        如果i大于m：
            k从j到r循环：
                添加nums[k]到temp末尾
            退出循环
        如果j大于r：
            k从i到m循环：
                添加nums[k]到temp末尾
            退出循环

    k从1到l-r+1循环：
        nums[l+k-1]为temp[k]
```

```
归并排序（一列数nums，l为我们处理的部分的最左边的位置，r为我们处理的部分的最右边的位置）：
    如果l大于等于r：
        返回

    m为向下取整((l+r)/2)
    归并排序(nums, l, m)
    归并排序(nums, m+1, r)
    合并(nums, l, r)
```

就完成了整个“归并排序”！诶？我好像没有告诉你“归并”这个名字的由来？呀哈！我认为它是“递归”与“合并”的组合。

第6章 快速排序

> 兵贵神速。
>
> ——成语

归并排序直接将数列分成一左一右两个部分，确实可行，但你就不觉得这有点……直接吗？让我们来考虑另一种划分方法，比如说……选择数列中的一个元素，把比他小或等于它的放在它左边，而比它大或等于它的放在它右边，然后对左右两边递归下去。这样，我们就能完成整个数组的排序。

很好！实际上这种排序方法叫做“快速排序”！它十分——快速！既然这样，让我们用激动的心情来思考：如何完成我们说的“划分”呢？

首先，选择数列中的一个元素——就第一个吧，这对于我们的代码实现比较简单。然后，我们先把第一个——称它为num吧——之后的部分按两部分分开：小于num的和大于num的。最终，只要把小于num的那一部分的最后一个数与num交换，就达到我们的目的了！

所以，把num之后的部分划分成小于它和大于它的两部分是非常重要的。如此，请看这段伪代码：

```
划分（一列数nums，l为我们处理的部分的最左边的位置，r为我们处理的部分的最右边的位置）：
    num为nums[l]
    i为l+1，j为r

    重复：
        如果nums[i]小于num，重复：
            如果i等于r：
                退出循环
            i为i+1
        如果nums[j]大于num，重复：
```

```
            如果j等于l:
                退出循环
            j为j-1
        如果i>=j:
            退出循环
        交换（nums[i]，nums[j]）

    交换（nums[l]，nums[j]）
    返回j
```

虽然格式十分奇怪，但它们做的事实际十分简单：首先，把num独立出来，我们之后的大多数——注意不是全部——处理都在它之后。接着，我们让i位于num的下一个位置，也就是我们划分的区域的开头。相应的，j位于划分的区域的末尾。

接下来，我们重复一串内容。首先，让i一直增长，直到找到一个不小于num的nums[i]。然后，于此对应，我们让j一只减小，直到找到一个不大于num的nums[j]。这时，我们应该怎么做呢？我们惊奇地发现，由于nums[i]大于num，而nums[j]小于num，nums[i]也就大于nums[j]！所以，只要交换它们，我们就可以让数列更有序，同时也能继续我们的“划分”。当然，如果i已经大于等于j，那么说明我们扫完了整个数列，也就需要退出循环。

最终，别忘记num的来源——nums[l]。由于此时，j肯定小于等于i，指向的数肯定不大于num，所以我们交换nums[l]与nums[j]，就能使num左边的数不大于它，而右边的数不小于它。而最终，我们返回j，也就是num所在的位置，方便我们递归。

有了划分的函数，我们也就可以完成整个快速排序。

```
快速排序（一列数nums，l为我们处理的部分的最左边的位置，r为我们处理的部分的最右边的位置）：
    如果l大于等于r:
        返回

    cut为划分（nums，l，r）

    快速排序（nums，l，cut-1）
    快速排序（nums，cut+1，r）
```

话说……为什么快速排序叫做“快速排序”呢？计算机的专家们为什么如此褒奖这个算法呢？因为它确实很快！快速排序的时间复杂度是$O(n\log n)$。我们之前介绍的归并排序的时间复杂度也是$O(n\log n)$，但实际使用中并没有快速排序要快。你可能会注意到我没有写底数。在计算机科学中，由于2这个数字十分重要，log一般指log_2。但是，既然我们在讨论时间复杂度，情况也就有所不同——时间复杂度忽略常数，而由于数学上的“换底公式”，对数运算的底数是可以随便转换的，只需要乘一个常数罢了。所以在时间复杂度中，对数运算不用关心底数。

当待排序数组很小时，快速排序会慢一点，因为它会进行很多很多划分。所以，在此时，我们可以使用其他排序方法。比如说，插入排序之类的。实际上，很多计算机语言都需要实现排序的功能，它们一般使用快速排序，不过可能会有修改，比如刚才我们提到的，必要时转换排序方法。

你会发现，归并排序和快速排序每一次都把数列分成两部分。这种“将大问题分成小问题，然后解决小问题从而解决大问题”的操作称为“分治”，是“分而治之”的意思。

第7章 计数排序

> 明明想在学校中请一些同学一起做一项问卷调查。
>
> ——NOIP2006 普及组 明明的随机数

刚才，我们完成了两个时间复杂度为$O(n\log n)$的排序方法。那么，还有更快的排序方法吗？接下来，听一个故事：

我有很多皮球，它们堆在我的房间里，十分杂乱。于是我决定收纳它们。思来想去，我觉得按照直径收纳它们可能是个好主意。也就是说，我把直径为10厘米的球放在红色箱子里，把直径为11厘米的球放在绿色箱子里，把直径为12厘米的球放在蓝色箱子里。

这时，我突然想要把这些球的直径排个序，于是我看到：红色箱子里有2个球，绿色箱子里有3个球，红色箱子里有4个球。于是，我的排序结果是`10 10 11 11 11 12 12 12 12`。

哇哦！这貌似是一种新的排序方法。我们抽象一下上面的操作：把球放在盒子里相当于对数列中的数做一些事，嗯……由于我们需要“箱子”，所以我们可以设立一个“计数数列”。所以，对于数列中的每一个数，将计数数组的相应一项加1。最终，计数数组的第x项表示数列中x的个数。

然后呢？我们发现，只是一些简单的累加操作，好像就完成了整个排序？因为只要我们按照数列中x的个数在新数组中产生x就可以了。由于这种排序方法用到了计数数列，它便被叫做“计数排序”

```
计数排序（一列数nums，其长度为n）：
    一列数howMany，长度为最大的数（nums）
    对于nums中的每一个num：
        howMany[num]为howMany[num]+1

    一列数new，一开始什么也没有
    i从1到最大的数（nums）循环：
        j从1到howMany[i]循环：
```

```
            添加i到new末尾

    nums为new
```

所谓对于nums中的每一个num也是一种循环，只是我懒得在这种时候写i这种东西了。你会发现，这个排序方法十分简便，但完成了与前面那些排序方法一样的事。它的时间复杂度是……嗯……$O(n)$！因为它先对nums中的每个数计数，然后根据计数产生新数组new，再让nums变成new。所以，我们为什么不广泛使用它呢？

注意这句话，一列数howMany，长度为最大的数（nums）。所谓最大的数，自然是一个会求出一个数列中最大的数的函数。也就是说，如果nums中最大的数max等于2147483647，那么howMany的长度也就是2147483647。对于大多数计算机来说，这不可接受。

还记得我说过的“空间复杂度”吗？这种计数排序的空间复杂度是$O(n+max)$。如果max很大，这种排序方法就玩完了。还有，我们的数组从1开始——我背叛我的精神——是的，所以，如果nums中有非负数，我们也玩完了。

呀哈，聪明的你肯定想到了，我们可以再得到最小的数（nums），然后做些许改动。

```
计数排序（一列数nums，其长度为n）：
    max为最大的数（nums），min为最小的数（nums）
    一列数howMany，长度为max-min+1
    对于nums中的每一个num：
        howMany[num-min+1]为howMany[num-min+1]+1

    一列数new，一开始什么也没有
    i从1到max-min+1循环：
        j从1到howMany[i]循环：
            添加i+min-1到new末尾

    nums为new
```

我们东搞搞，西搞搞，使得hwoMany的长度变成max-min+1，空间复杂度也就变成$O(n+max-min)$。但当然，如果数列中有从−2147483647到2147483647的所有数，我们的空间还是不够用。

所以说，这种排序方法只适用于部分情况，比如当我们只有1到100的数时。

第-1章 结语

“所以，感觉怎么样？”何乐乌用指尖转着笔，歪歪头，眨眨眼，向沃柔德问道。

何乐乌张着亮晶晶的眼睛，头一仰一仰吸着空气，说道：“你在使用一种新颖的教学方式，我不好对你独特的方式进行评论……但我确实从你这里听到了很多神奇的东西……你真的要这样在大学讲课吗？”

“会做一点改动——只有你是最不幸的小白鼠。”

“嗯，你真有趣！”沃柔德笑趴在桌子上，“你真是我有趣的朋友！”

“好，有趣的朋友！话说，你对我讲述的内容——‘神奇的东西’——持什么看法呢？你想了解更多吗？”何乐乌说道。

“像你一样有趣，或者说……你与它们一样神奇……啊！你会接着讲下去，是吗？”

“是的！”何乐乌凑近过来，两人鼻尖几乎相接，然后它们又不约而同往后顿过去，“我确实会讲下去，但——不是现在。看呐！已经到午夜了，人总是要睡觉的，而我还需要养精蓄锐。所以，等一段时间，我们再来这里。”

“好。嗯呢。”

“我还启发了你对语气词的使用啊！”何乐乌笑笑，“如果你想干一些与算法相关的事的话，我有几条建议：第一，在网上查询我讲过的算法，毕竟我的讲解可能不太深入，甚至可能有错；第二，查询‘希尔排序’与‘基数排序’这两种算法……”

“为什么？”

“因为我懒得讲——啊不是，因为你需要……呃……‘自主学习’！第三，难度比较大的一条——学习一种编程语言，实现以上我讲过的和你查的算法！”

“很好，”沃柔德说道，“你就像——啊不是——确实是个老师。我想，我们各自回去吧”

两个人笑着起身。