## 计数初步

*人类最大的缺点就是不理解指数函数的意义。*

——Albert A. Bartlett博士，世界人口平衡委员会顾问

从前，当著名数学家Carl Friedrich Gauss[[1]](#footnote-1)还在上小学时，他的老师曾要求学生们进行一次从1到100的整数累加运算（至少故事最常见的版本是这样的）。毫无疑问，这位老师原本以为这足以打发学生一阵子时间了，但Gauss似乎立刻就算出了结果。看上去好像这需要拥有闪电般的心算速度才行，但实际上这类计算非常简单，关键在于我们能不能真正理解问题。

或许在经历了上一章的长篇累牍之后，您已经对这类问题已经有些厌烦了。您会说：“显然答案又是Θ(1)！”好吧，的确如此……但我们现在所讨论的是从1到*n*的整数求和问题。而且在后续各节所涉及的一些重要问题中，这样的求和式还会一次又一次地出现在我们的算法分析中。尽管本章的内容会给我们带来一些挑战，但其所提出观点也较为关键，也值得我们去研究。因为它们可以让后面的章节变得更浅显易懂。首先，我们会对求和式的概念做一个简要说明，并介绍一些相关的基本操作方式。然后在接下来的两个主要小节中，我们将分别介绍两种基本的求和问题（或者，您也可以按照自己的习惯称其为组合问题）以及所谓的“递归式”。后者会被用于一些递归算法的分析。另外，我们还会在这两节内容之间插入一个小节，谈论子集和排列组合问题。

**提示：**本章将会涉及相当多的数学内容。如果您当下对这些东西不感兴趣，也可以先暂时略过它们，继续读本书其余部分的内容，待需要时再回头来看这些内容。（尽管，本章所提出的若干思维能让本书其余部分的内容变得更浅显易懂。）

### 求和式的含义

我们在第2章中曾经说过，当两个循环嵌套，内层循环的复杂度在每轮外层迭代下各不相同时，这里所开始形成的就是求和式。事实上，求和式在算法中是无处不在的，所以我们很有必要学会习惯于运用这种运算思维。下面就让我们先从一些基本记法开始吧。

#### 更多希腊字母

在Python中，求和式通常是这样写的：

x\*sum(S) == sum(x\*y for y in S)

而如果将其转换成数学记法的话，就得这样：

（公式）

（能明白为什么这个公式能成立吗？）如果以前没有用过的话，我们可能会觉得这个大写的Σ[[2]](#footnote-2)看起来有些吓人。但其实它并不比Python中的sum函数可怕多少，只是语法上稍有些不同而已。Σ本身所代表的就是我们通常所谓的求和式。其上、下方以及右边都是一些与该求和式相关的各种信息。该符号右边（即例子中*y*或*xy*所在之处）代表的是要求和的值，而Σ底部描述的则是在运算过程中将会被遍历到的各参与项。

与单纯的set（或其它集合类型）对象遍历操作不同的是，求和式是可以直接指定运算边界的，类似于range（区别是，求和式包括双边界）。也就是说，通常我们要表达“sum *f*(*i*) for *i* = *m* to *n*”的话，就应该要这样写：

（公式）

而其在Python中的等效表示则为：

sum(f(i) for i in range(m, n+1))

或许，许多程序员更容易将这些求和式的数学形式写成循环，例如：

s = 0

for i in range(m, n+1):

s += f(i)

很显然，数学记号更为紧凑，它的优势在于能帮助我们更直观地了解它所要做的事。

#### 求和式的运用

在上一节中，我们曾列过一个等式，它的*x*因子被直接移到了求和式中。很明显，这里使用了一种“乘法规则”，这意味着我们可以在求和式中使用这些规则。这其中最重要的规则主要有两条（就我们的目标而言）：

（公式）

*乘数因子可以被直接移入或移出求和式*。这也就是我们在上一节中最早描述的那个例子。它与我们在进行简单求和运算时使用的分配律是一样的，例如：*c*(*f*(*m*) + … + *f*(*n*)) = *cf*(*m*) + … + *cf*(*n*)。

（公式）

*除了将两个求和式相加之外，你也可以将两式的内容部分相加。*这意味着我们在对一堆东西执行求和运算时，将不会受到具体细节的影响，也就是说：。

sum(f(i) for i in seq) + sum(g(i) for i in seq)

 与sum(f(i) + g(i) for i in seq)的结果是完全相同的。这只是该规则*相关*实例中的一种。当两个求和式相减时，我们也可以对其使用同一技巧。（如果愿意的话，您可以认为自己是将一个-1因子移到了第二个求和式中。）

### 两种赛制的故事

您可能已经发现了，我们在工作中总是会接触到大量的求和式，而有一份良好的数学参考也许能帮助我们找出应对这些求和式的最佳解决方案。其中有两种求和式，或者说两种组合问题将会涉及到这本书半数以上的案例——或者换而言之，它们是我们工作中最为基础的算法。

这些年来，我一直在许多不同的实例或比喻场景中反复诠释这两种思想，但其中最令人印象深刻（我希望同时也是最能被理解的）的体现方式还是下面这两种*竞赛*形式。

**请注意：**事实上，图论中也有一个叫做竞赛（tournament）的技术名词（这是一种完全图，其每一条边都被赋予了一个方向），但那并不是我们现在要谈的东西（尽管它们在概念上有些相关性）。

尽管竞赛的类型很多，但我们现在只考虑既常见又引人注意的那两种形式，它们分别是*循环赛*和*淘汰赛*。

在循环赛中（或者说得更确切一点，在*单轮*循环赛中），每一个选手都会被安排轮流与其他人比上一回。然后问题就产生了，例如在一个具有*n*个骑士的格斗游戏中，我们究竟需要安排多少配对或赛程呢？（只要您愿意，也可以将其换成任何一种您喜欢的竞技活动。）而在淘汰赛中，参赛者则通常是被安排成对比赛的，只有每对当中的赢者才能进入下一轮比赛。这里面能产生的问题就更多了，例如对于*n*个骑士来说，我们究竟需要多少轮比赛？总共要进行多少场比赛？

#### 握手问题

循环赛问题完全等价于另一个著名难题；即如果在一个有n个算法专家参加的会议中，您需要他们之间所有人互相握手，那么您会得到多少次握手呢？或者该问题又等同于：一个含n个节点的完全图（见图3-1）中究竟应该有多少条边？这里所涉及的都是对于任何一种“整体对整体”的计数问题。例如，如果我们想在某地图上n个位置中找出彼此距离最短的两个点，最简单的（暴力）方法就是比较所有点之间的距离。想要找出该算法的运行时间，我们就必须要解决循环赛问题（对于*最近点问题（closest pair）*，我们在第6章中还会介绍一种更为有效的解决方案）。

（图）

**图3-1.** 一个用于诠释循环赛或握手问题的完全图

您很可能已经在猜测，比赛次数可能会是平方级。毕竟“整体对整体”听上去有点像“整体乘整体”，也就是*n*2。尽管从结果来说，它也确实是平方级的，但要将这里的比赛次数精确到*n*2的话，就未必完全正确了。您要想想看，只有想死的骑士才会跟他/她自己交手吧？而且，Galahad爵士跟Lancelot爵士交锋之后，Lancelot爵士就没有必要再回头来一次，因为他们确定已经彼此交过手了，所以我们所要安排的都是单场比赛。而简单的“*n*乘*n*”的解决方案会忽略掉这些因素，它设定每位骑士都会被单独安排跟*这些骑士中的每一个人*交手（包括跟自己）。修改方法很简单：就是让每位其实都得到一次跟其他所有人交锋的机会，也就是*n*(*n*–1)。然后，由于他们每个人都被安排交锋了两次（因为一次只能为一位骑士安排），所以我们还要对其除以2，于是其最终答案为*n*(*n*–1)/2，它确实是Θ(*n*2)。

现在，我们已经用一种相对简单易懂的方式合计出了比赛次数（或者说握手次数、地图上各点距离的比较数）——并且答案也似乎是显而易见的。嗯，尽管不能完全等同于火箭技术，但在所有的这一切当中，至少有一点是可以放心的……即对于我们现在所用来合计这些东西的*不同*方法来说，结果都必须是相同的。

这里所谓的其它合计方法是这样的：第一位骑士的对手数应该是*n*-1，接下来，第二位骑士的对手数就是n-2了。这样一直推下去，一直到与最后那位骑士（而他只剩下0场比赛和0个骑士对手了）的最后那场格斗比赛。这样，我们就可以得出它们的求和式：*n*–1 + *n*–2 + … + 1 + 0，或者sum(i for i in range(n))。我们之前是一次性合计了每一场比赛的次数，所以该求和式的结果必然与之前相同：

（公式）

当然在这里，我可以直接提供这样一个方程式，但是，我希望通过前面这种解释能帮助人们更好地理解其中的含义。当然了，我们也可以随时用其它方式来诠释该方程（或者其它贯穿本书的概念）。例如在本章开头的那个故事中，如果具有Gauss那样的洞察力，我们就会先将从1加到100这种求和运算的“字面形式”，转换成1加100、2加99（以此类推）等这样值为101的对组，然后再求这50个对组之和。如果我们继续将这种求和方式推广到0到*n*-1的情况，您机会得到一个与之前完全相同的公式[[3]](#footnote-3)。（另外，您能看出这些东西与邻接矩阵左下半部分，也就是对角线以下部分之间的关联性吗？）

**提示：**等差级数（arithmetic series）本身也是一种求和式，在求和式中，任何两个相邻数字项之间的差距都是一个常数。假设该常数为正的话，这种求和式的复杂度将始终是平方级的。事实上，数列i k（i = 1…n，且k > 0）的求和复杂度应始终为Θ(n k+1)[[4]](#footnote-4)。握手问题的求和式只是其中的一个特例。

#### 龟兔赛跑

假设我们的骑士数量为100，并且对于去年所经历的循环赛制，选手和工作人员都觉得有点太累了。（这其实很好理解，毕竟那意味着4950场比赛。）于是他们这次决定引入（更有效率的）淘汰赛制，并且重新计算出他们所需要进行的比赛场次。想要找出这样的解决方案可能会有点困难……或者也可以说它非常显而易见，这取决于您对这个问题的具体考虑。下面，就让我们先从较为困难的角度来考虑吧。在首轮比赛中，由于所有骑士都要参与配对，所以应该会有*n*/2场比赛。而进入第二轮比赛的人数只有原来的一半，因而这一轮的比赛场次应该为*n*/4。如果以此类推下去，一直到最后一轮比赛，比赛的总场次应该合计为*n*/2 + *n*/4 + *n*/8 + … + 1，或者也可以等效写成1 + 2 + 4 + … + *n*/2。稍后我们会看到这种求和式的多种应用，但它的答案是什么呢？

现在我们来看看显而易见的那部分吧：由于每一场比赛都会有一名骑士被淘汰。所以最终除冠军以外的所有人都会被淘汰（并且他们都是一次性淘汰），所以我们需要*n*-1场比赛来决定只留下哪一个男人（或女人）。（该比赛结构如图3-2所示，这是一颗带根的树结构，其每一片叶子都代表了一场比赛）。换而言之，也就是：

（公式）

如您所见，该比赛轮次的上限应为*h*-1（或者由于该二叉树的高度为*h*，所以2*h* = *n*）。尽管基于这样的环境设定，结果看起来并没有那么诡异，但其实还是真有些异样的。因为毕竟从某种程度上来说，整件事的基础是一个有很多人是会死而复生的神话。但即便这个神话是错的，其实它也并没有那么离谱啦！毕竟人口增长大致上都是指数级的，而且每50年就会翻一倍。假设我们有一段固定的、人口成倍增长的历史时期（虽然未必真的有，但您在这就当是玩个游戏吧），或者再简化一点，我们可以设想成在人口数达到最多之前[[5]](#footnote-5)，人口是每一代人增长一倍。如此一来，假如当代人口结构是由n个人组成的，之前各代人口（也就是我们已经看到）的数量总和就只是*n*-1（当然，这些人里面有一些还活着）。

（图）

**图3-2.** 这是一个（完全平衡）带根节点的二叉树结构，有n个叶节点及n-1个内部节点（根节点被突出显示）。虽然该树本身可以没有方向性，但如您所见，通常我们会（隐式地）认为其边线是向下延伸的。

|  |
| --- |
| **二进制的工作原理** |
| **在我们刚刚所看到的求和式中，参与项都是2的指数，而其结果总是等于2的*下一个*幂减1。譬如1 + 2 + 4 = 8 – 1、1 + 2 + 4 + 8 = 16 – 1等等以此类推。这正好（从某种角度）说明了为什么我们可以用二进制形式来进行计数。通常一个二进制数就是一个由0和1组成的字符串，它的每一位都代表着求和式中给出的二次方数（从最右边的20 = 1开始）。因此，像11010所代表的就是2 + 8 + 16 = 26。因此，对h个二次方数进行求和，与一个像1111（共有h位）这样的数字是等效的。虽然这是我们根据h个数位得出的结果，但很幸运，如果我们所求的是到*n*-1的和值，那么它下一个加数的幂正好就是*n*，例如1111等于15，10000就等于16。（在练习题3-3中，我们将会要求您凸显出该属性，用二进制的形式来表示任意正整数。）** |

上面是关于实现倍增式操作的第一课，其操作结构就像是一颗拥有n-1个内部节点，且完全平衡的二叉树（即这是一颗带根节点的树，其所有内部节点都有两个子节点，并且其所有叶子节点都具有相同的深度）。然而在这方面的问题上，我们还有更多的课要上。例如我们到现在还没有涉及到本节标题所暗示的内容：龟兔赛跑问题。

在这里，兔子和乌龟分别代表了树结构的高度和宽度。当然由于这幅图像中存在着若干问题，所以我们不必把它看得太严肃，但其中的思路是：这两个比较起来（事实上，可以把它们各自看作是对方的一个*函数*），其中一个增长得较慢一些，而另一个则快得多。我们之前已经将其状态关系表示为*n* = 2*h* ，但我们可能更容易用到其逆关系，该关系可以被定义为一个二进制对数*h* = lg *n*（如图3-3所示）。

（图）

**图3-3.** 一颗完全平衡二叉树的高度与宽度（即叶节点数）

然而，想要彻底弄清楚这两者之间究竟有多大的差距也是一件很麻烦的事。一种策略就是直接认为它们之间存在着巨大差距（即要么就是超理想的对数级算法，要么就是完全不行的指数级算法），然后经我们所能地拿出能反映这些差距的具体实例。下面，我们就来举一些这方面的入门实例。首先来玩一个被叫做“猜粒子”的游戏。具体就是我想好了已知宇宙范围内的某一种粒子，然后由您来猜它是哪一种，而我就只回答您yes或者no，OK？成交！

虽然看上去，玩这种游戏纯粹是有点神经错乱，但我向您保证，它跟可行性（例如跟踪哪种粒子已经被排除过了）的关系要比它跟选择数量的关系大得多。当然我们也可以将问题简化得更实际一些，将其简化成“猜数字”游戏。当然由于我们原本所谈论的是粒子，所以预计数量一定会很多，但1090（也就是1后面跟90个0）也应该是相当宽裕了吧。我们可以自己在Python环境中试试看：

>>> from random import randrange

>>> n = 10\*\*90

>>> p = randrange(10\*\*90)

现在，我们手里已经有了一种未知粒子（粒子编号为p）。下面，您可以开始研究应该问怎样的yes/no问题了（不许偷看！）。例如您可能会问下面这种毫无建设性的问题：

>>> p == 52561927548332435090282755894003484804019842420331

False

之前有玩过“二十个问题（twenty questions）”[[6]](#footnote-6)这类游戏的人肯定明白，这样做的问题在于：我们得不到足够多的“回报”，所以我们最好要问一个能让数字可选范围减半的yes/no问题，例如：

>>> p < n/2

True

现在我们终于有了一些进展！事实上，如果您牌玩得不错（抱歉，比喻有点乱——或者说游戏玩得不错）的话，可以这样持续成倍地缩小其可选范围，大致在300个问题之内就能找到答案。您可以像下面这样自己算算看：

>>> from math import log

>>> log(n, 2) # base-two logarithm

298.97352853986263

如果觉得这有点平淡无奇，那就请您再仔细想一想。我们只凭着一些yes/no的问题，就能*在五分钟以内猜出宇宙中任何一种可见粒子！*这是所谓超理想对数算法的一个典型示例（现在，试着快速说“logarithmic algorithm”十遍。）

**请注意：这其实**是二分法或二分搜索法的一个实例，同时也是最重要、最知名的一种对数级算法。我们将会在第6章介绍bisect模块的专栏中进一步讨论这个问题。

现在我们将注意力转向对数级算法的伪对立面，来探讨一下同样怪异的指数级算法。任何一种示例都可自动成为另一方面的示例——如果我们从单一粒子开始，并让其持续翻倍，很快您就会填满所有我们所能观测到的宇宙空间。（我们已经看到，这大约只需要进行299次的翻倍。）其实这只是*小麦与棋盘[[7]](#footnote-7)*这个老问题的一个更为极端的版本。即如果您在棋盘的第一个方格放一颗小麦，第二个方格中放两颗，第三个方格中放四颗，以此类推下去，最后会有多少小麦呢？[[8]](#footnote-8)棋盘上最后一个方格中的小麦数应该是263（我们是从20 = 1开始算的）。如果按图3-2所示的求和式来计算，这就意味着棋盘上应该一共有264–1 = 18446744073709551615颗小麦，或者一共有5 · 1014 kg的小麦。这可是一个很大的量——相当于世界粮食年产量总和的几百倍！现在再来设想一下，如果我们要处理不是粮食而是*时间*。即对于一个*n*规模的问题而言，我们程序需要的时间为2*n*毫秒。那么当n = 64时 ，该程序将需要运行584542046*年*！即要让它今天完成的话，该程序不得不在很久之前就开始运行，恐怕我们写该代码的时候周围还没有任何脊椎动物吧。指数级增长是非常可怕的。

到了这一步，我希望您对指数级算法与对数级算法之间所谓的互逆关系已经有了一个初步的认识。但在本节结束之前，我还想谈谈另一对我们在处理龟兔赛跑问题时经常会遇到出现的对称关系：当然在这里，从1到*n*的翻倍数与从*n*到1的减半数是相同的。虽然事情非常明显，但当我们将来处理到一点递归式时，还是有必要回顾一下这一点，这对理清思维是很有帮助的。让我们再来看看图3-4，该树结构显示了从1（即根节点）到n（n个叶节点）的整个倍增过程，但我在其中添加了一些能够在整个树结构中传递的标识（注意这些节点下面的标签）。

比方说，*某个家庭的祖父母*（即根节点）手中*n*个冰淇淋甜筒（正如练习题2-10中所暗示的那样）。他或她有两个孩子，每个孩子可以分到一半的冰淇淋。当然，这些孩子自己也是父母，他们会继续将手里的冰淇淋对半分给每个孩子。整个过程会一直持续到该家庭中每一个孩子（即叶节点）那里，他们每个人都会收到一个冰淇淋甜筒（这里有*n*个页节点及*n*个甜筒/标记）。其初始条件（即根节点拥有*n*个标记）及终结条件（即*n*个叶节点各自拥有1个标志）都非常容易掌握。这个故事中有一点值得注意，即在整个过程（即我们将要处理的递归式）中，分布在各个步骤（即该树结构的每一层）中标志数量是相同的。也就是说，第1层有两个节点，它们各自拥有*n*/2个标志；而第2层有4个节点，它们各自拥有*n*/4个标志等等，以此类推。总之这里任何一行的总和（如下图所示）都是*n*。这里还有一件事值得注意，即每个节点所拥有的标志数是随着节点数的翻倍而减半的……或者说得更直接点，*n*个冰淇淋甜筒是被一代一代传递下去的。

（图）

图 3-4. n个标志在二叉树各层中传递的过程

**提示：**任何一个几何级（或指数级）数列都是k i（其中i = 0…n，k为常数）的一个求和式。如果k大于1，该求和式复杂度始终为Θ(k n+1)。倍增式序列的求和式只是一个特例。

### 子集与排列组合

正如您在上一节中所读到的，*k*长度的二进制数串计算起来是非常容易的。例如，我们可以将那些数串看成一棵完美平衡二叉树从根节点到叶节点的遍历走向。那么该数串的长度k就应该等于树的高度，而所有可能的数串的数量则应该等于其叶节点数2*k*。除此之外，我们还有一种更为直接的方法，就是考虑各步骤中所可能需要的不同可能性的数量：第一个比特位可能是0或1，对这两个中的每个值，第二个比特位*也*只有这两种可能性，如此类推的话，它就像是一个k层嵌套的for循环，每层循环有两个迭代操作。操作总数依然是2*k*。

|  |
| --- |
| **伪多项式（时间）算法[[9]](#footnote-9)** |
| **这个词很棒，对吧？其实这是一组特定算法的名称。这些算法的运行时间都是指数级的，但往往“看上去像”是一个多项式时间的算法，甚至于其在具体实践中的行为也像是多项式级算法。比如当我们要检查或回答“某个数是不是质数”这类问题时，通常会认为其解决方案应该是多项式级的，但其实问题远没有那么想当然……而且事实上，这种想当然的方式所产生的将是一个非多项式级的解决方案。**  **下面就让我们来看一个相对直接的解决方案：**  def is\_prime(n):  for i in range(2,n):  if n % i == 0: return False  return True  **该算法其实就是从2开始，逐步遍历所有小于n的正整数，并检查它们是否能整除n。只要它们有一个能整除，n就不是质数，否则就是质数。看上去这的确像是一个多项式级算法，运行时间应该为Θ(n)。但关键是，这里的n根本不是该问题合法的规模值！**  **当然，将其运行时间描述成n的线性关系，甚至说成是n的……一个多项式也是有用的。但这并不等于说这就是一个……多项式级算法。其原因在于，通常我们所谓的运行时间应该是一个空间函数，它的参数是我们对某个问题实例进行编码时所需要的空间大小，如果我们不对它加以特别说明，那就表示这一种运行时间函数是“多项式级”的。虽说一个问题的规模值包含n并不等于就是n，但即便我们针对n规模来编码，其也是需要一定量的比特数的。例如，如果n等于2的某次方，那么其需要的比特数大约等于lg n + 1。（对于一个任意的正整数，实际则需要floor(log(n,2))+1个比特位。）**  **假设我们将问题规模（即比特数）设为k，那么（大致上）n = 2k-1。如此一来，我们宝贵的运行时间Θ(n)就在其实际问题规模函数重写下变成了Θ(2k)，这显然是一个指数级算法。[[10]](#footnote-10)像这样的其它算法还有不少，它们的运行时间可以被解释为其某个输入数值的某个多项式函数。（第8章中将要讨论的*背包问题*解决方案就是其中一个例子。）它们被统称为*伪多项式级算法*。** |

子集关系非常直接：如果每个比特所表示的是一个对象是否存在于某个k大小的集合中的话，那么每个比特串所代表的就是该集合的2k个可能子集中的某一个。或许这里最重要的是说明了每一个需要检查其输入对象中每个子集的算法，时间复杂度都必然是指数级的。

虽然对于算法设计者来说，子集问题属于基础内容，但其排列组合问题或许显得多少有些像是边缘概念。尽管如此（并且没有它的话，本节就不能叫“计数初步”了），您也依然有可能会碰上它们，所以我们觉得还是有必要对这些问题做一个简短介绍。

*排列问题*的本质就是顺序问题。例如对于n个人排队买电影票的问题，我们可以拿出多少种可能的队伍呢？这里的每一种队伍都将是这些排队者的一种排列方式。正如我们在第2章中所说的那样，n个项的排列数等于n的阶乘，即n!（该表达式包括感叹号，读作“n的阶乘”）。我们可以通过n（第一个位置可能的人数）乘以n-1（第二个位置可能的人数）再乘以n-2（第三个位置……），一直乘到1的方式计算出n!的值：

（公式）

运行时间涉及到n!级复杂度的算法并不多（尽管我们在第6章讨论有限排序问题时还会再次涉及到这一类累计运算）。其中倒是有一例名叫bogosort的无聊排序算法，其预计运行时间为Θ(*n* · *n*!)，该算法的主要工作就是对随机输入的序列进行多次地洗牌，然后检查其结果否已经被排序。

而*组合问题*与排列问题和子集问题都有紧密的联系，它其实就是n的集合中k个元素一种组合。我们有时将其写作*C*(*n*,*k*)，有时候也会用以下数学括弧来表示：

（公式）

我们通常称它为*二项式系数*（*binomial coefficient*，有时候也叫做*选择函数*，*choose function*），并读作“*n*选*k*”。虽然认识其背后的阶乘公式相对简单一些，但在二项式系数计算方面就没有那么明显。[[11]](#footnote-11)

请（再次）设想一下，如果眼下有*n*个人正要排队看电影，但电影院里却只有*k*个座位。那么这种*k*大小的子集究竟有多少种可能性呢？答案当然就是*C*(*n*,*k*)种。而且在这里，这种比喻形式还可以帮助我们完成一些工作：显然，我们已经知道整个集合的可能顺序有*n*!种。但如果只是对这些可能性进行计数，那先前那个k要怎么办呢？所以只针对这个问题而言，我们在其子集计数上花的次数太多了。毕竟，这里要计算的群组是由排在众多排队者前面的头*k*位朋友所组成的。那么事实上，排在所有人前面的这些朋友存在的排列可能性应该有*k*!种，而剩下的这些人在没有别人加入的情况下，其排列可能性也应该有(n-k)!种。这一来我们就有了答案：

（公式）

该公式就是拿全体排队的可能性总数（*n*!）去除我们累计出来各种“获胜者子集”数。

**请注意**：我们在第8章讨论动态规划时，还会从另一个不同的角度来介绍二项式系数的计算。

值得一提的是，我们在这所选取的都是大小为k的子集，这意味着该选取操作应该是*不可替换的*。如果我们只靠抽k次签来决定人选的话，就有可能会多次抽到一些相同的人（这会有效地“顶替”掉候选池中的其他人）。这样一来，其可能的结果就直接变成*nk*了。实际上，*C*(*n*,*k*)所累计的是k规模子集的可能数，而2n所累计的是总体排列的可能数。由此我们得出了以下这个漂亮的等式：

（公式）

对于这些组合性对象来说，差不多也是如此。是时候展开一些不可思议的展望了：用它们自己来解决其等式问题吧！

**提示：**对于大多数学操作来说，交互式Python解释器都是一个非常方便的计算器。其math模块中包含了许多有用的数学函数。但它们对于我们在本章中所涉及到的那些数学符号来说，帮助倒不是很大。不过，我们还有其它用Python表示特定数学符号的操作工具，例如Sage（可去<http://sagemath.org>处下载）。而如果您只是需要一个便捷工具来求解某个特别棘手的求和式或递归式（详见下节内容）的话，也许可以到Wolfram Alpha（<http://wolframalpha.com>）上面去查查。您只需要输入相关的求和式（或一些其它的数学问题），就会得到相应的答案。

### 递归与递归式

在这里，我们会假设您对递归问题已经有了*一些*起码的经验。虽然我们在本节中还是会对其做一个简短介绍，并且到了第4章还会有更为详细的讨论，但如果它对您来说是一个完全陌生的概念。或许去找一本基础性的编程参考书来看看也是一个不错的主意。

所谓递归，其实就是指某一函数——直接或间接——调用自己的操作。下面我们来简单演示一下如何用递归方式来求解某一序列的求和式：

def S(seq, i=0):

if i == len(seq): return 0

return S(seq, i+1) + seq[i]

下面，我们来了解一下该函数是如何工作的，并计算出这两个关联任务所需的运行时间。函数本身非常简单：该求和式从参数i开始，当其值超出目标序列的尾端时（这叫做*基本情况，base case*，主要用于防止无限递归），函数就直接返回0。否则就将i的位置加1，继续求剩下序列的和。在这种情况下，我们每次执行rec\_sum的工作量是恒定的，这里不包括递归调用。并且由于其执行次数是与序列中各项是一一对应的，所以很明显它的运行时间应该是线性级的。不过，我们还是来具体验证一下吧：

def T(seq, i=0):

if i == len(seq): return 1

return T(seq, i+1) + 1

这个新函数T的结构与S基本相同，但其操作值稍有些不同。即它的返回操作与S相同，但其内容却是从*一个子问题的解决方案*，换成了*该解决方案的开销*。具体到眼下的情况，就是要对if语句的执行次数进行计数。（在那些更为数学化的设定中，我们也可以通过相关的操作符来完成计数操作，例如用Θ(1)来代替1。）下面，我们就来试试这两个函数吧：

>>> seq = range(1,101)

>>> S(seq)

5050

现在我们知道了，Gauss是对的！下面我们来看看它的运行时间：

>>> T(seq)

101

一切正常。由于*n*本身大小为100，所以这儿是*n*+1。但似乎我们一般应该是这样做的：

>>> for n in range(100):

... seq = range(n)

... assert T(seq) == n+1

没有错误，所以上面的假设似乎有一定的可信度。

现在，我们要做的工作是找出T这种函数的非递归版本，以便我们来定义递归算法的运行时间复杂度。

#### 手动推导

为了从数学角度来描述递归算法的运行时间，就得要用到递推等式，我们通常称之为*递归关系*。如果我们的递归算法是像上一节中S那样的函数，那么其递归关系的定义就有点类似于后面的函数T。由于我们的工作是要得到一个渐进式的答案，所以通常都不会在意其常数部分，并且默认*T*(*k*) = Θ(1)（其中k为常数）。这意味着我们可以在设置相关等式时忽略掉其基本情况（除非它*不是*一个常数级操作），对于函数S而言，它的T可以定义如下：

（公式）

也就是说，计算S(seq, i)所需要的时间（即*T*(*n*)）等于递归调用(seq, i+1)所需的时间加上访问seq[i]所需的时间（这是个常数或Θ(1)级操作。换句话说，我们可以先在常数时间内将目标问题缩减成其自身的较小版本（譬如规模由*n*变成*n*-1），然后再来解决这个较小的子问题。而这两个操作的时间之和就等于解决该问题所需的总时间。

**请注意**：如您所见，我们在这里是用1而不是Θ(1)来表示递归以外的额外操作的，当然这里也可以用Θ来表示。只要我们描述的是一个渐进式的结果，这也确实没多大关系。但在这种情况下使用Θ(1)也存在着一点风险，因为我们要在这里构建一个求和式（1 + 1 + 1 …），而如果我们在其中使用渐进符号的话（即Θ(1) + Θ(1) + Θ(1) …），这个求和式反而容易被错误地简化为常数。

现在，我们应该如何*解决*这样的等式问题呢？线索就在我们将*T*实现成一个执行函数的过程中。由于其不需要在Python中运行，所以我们可以自行模拟递归调用的相关情况。对于这方面来说，下面这个等式是关键所在：

（公式）

显然被我们用方框框起来的那两个子公式是相等的，这就是关键所在。我们声明这两个方框内容相同的基础正来自我们原有的递归式，即如果：

（公式）

那么：

（公式）

在这里，我们将原等式中的*n*直接替换成了*n*-1（自然，*T*((*n*–1)–1) = *T*(*n*–2)），这证明了上面方框中的内容的确是相等的。然而，我们这里所定义的都是带有更小参数的T，其本质上就是递归调用执行时将要发生的情况。所以，从*T*(*n*–1)（第一个方框）扩展到*T*(*n*–2) + 1（第二个方框）本质上也只是模拟或“解开”了一层递归而已。我们仍然需要继续面对*T*(*n*–2) 的问题，但其处理方式是一样的！

（公式）

显然，这里的*T*(*n*–2) = *T*(*n*–3) + 1（即上面被框住的两个表达式）也是从原先的递归关系中推导出来的。从此我们得出了一个模式：即其参数每减小一次，其展开的工作量（或时间）之和就会*增加*一次。如果*T*(*n*)的递归展开次数为i，那么我们就会得到以下关系：

（公式）

这*才是*我们要找的表达式——一种能把递归展开的层数用一个变量i来描述的表达式。因为所有这些递归展开表达式都是相等的（我们已经推导了其每一步展开的等式），所以我们可以根据自己需要自由地给 *i*设置任何值，只要不越过其基本情况（例如*T*(1)）即可，因为递归关系到了这里就该中止了。我们要做的就是使其*持续地向*基本情况演变，即试着将*T*(*n*–*i*) 逐步变成*T*(1)，因为我们已经知道（或者默认）*T*(1)的时间复杂度为Θ(1)，这就意味着我们解决了整件事情。并且我们还可以通过设定*i* = *n*–1，很轻易地得出以下关系：

（公式）

或许，通过比预期更为复杂的分析，我们现在更有把握说S是一个线性级运行时间的操作了。在下一节中，我们将会通过一组不那么直观的递归式，来为您具体演示这种方法的使用。

**提醒：**这种被称之为*重复代入法*（repeated substitutions，有时候也叫*迭代法*，*iteration method*）的方法完全有效，当然前提是您要小心仔细。否则，它是很容易产生一两个不完全正确的假设的，*尤其*在那些较为复杂的递归式中。这意味着我们得先自己提出一个*假设性*结果，然后再用本章稍后在“猜测与检验”一节中所描述的那些技术来检验我们的答案。

#### 几个重要例子

通常情况下，我们所遇到的都是递归式的一般形式：*T*(*n*) = *a·T*(*g*(*n*)) + *f*(*n*)，其中a指的是递归调用的数量，*g*(*n*)则是递归过程中所要解决的子问题大小，而*f*(*n*)则代表了函数中的额外操作，也就是递归调用以外的操作。

**提示：**当然也存在着一些特殊的递归算法，例如当其子问题大小不相同时，它可能就无法符合上述形式了。尽管这类情况不在本书的讨论范围，但在本章最后的“如果您感兴趣……”这一节中，我们还是针对某些关键点提供了进一步的信息。

在表3-1中，我们对一些重要的递归式（都是问题规模为*n*-1或*n*/2上的一到二次递归调用，其每次调用的额外操作时间都是常数级或线性级的）进行了汇总。其中递归式1的情况我们在上一节中已经见过了。而在接下来的内容中，我们将会向您演示如何运用之前介绍的重复代入法来解决后四个递归式，而其余三个递归式（即2到4）将会成为练习题3-7到3-9中的内容，留给您自己解决。

**表3-1.** 一些基本递归式的解决方案及其应用实例

|  |  |  |  |
| --- | --- | --- | --- |
|  | 递归式 | 解决方案 | 应用实例 |
| *1* | *T*(*n*) = *T*(*n*–1) + 1 | Θ(*n*) | 序列化处理问题，例如归简操作。 |
| *2* | *T*(*n*) = *T*(*n*–1) + *n* | Θ(*n*2) | 握手问题 |
| *3* | *T*(*n*) = 2*T*(*n*–1) + 1 | Θ(2*n*) | 汉诺塔问题 |
| *4* | *T*(*n*) = 2*T*(*n*–1) + *n* | Θ(2*n*) |  |
| *5* | *T*(*n*) = *T*(*n*/2) + 1 | Θ(lg *n*) | 二分搜索问题（详见第6章中的bisect专栏） |
| *6* | *T*(*n*) = *T*(*n*/2) + *n* | Θ(*n*) | 随机选择问题、平均情况问题（详见第6章） |
| *7* | *T*(*n*) = 2*T*(*n*/2) + 1 | Θ(*n*) | 树的遍历问题（详见第5章） |
| *8* | *T*(*n*) = 2*T*(*n*/2) + *n* | Θ(*n* lg *n*) | 利用*分治法*进行排序问题 （详见第6章） |

在我们开始运用后四种递归式进行操作之前（它们全都属于*分治*递归式，我们将会在本章后面的内容以及第6章对此做更详细地说明），我们可能需要用图3-5来重新整理一下自己的记忆。该图汇总了我们到目前为止所有有关二叉树的讨论结果。其实这已经足够了，我们实际上已经向您提供了全部所需的工具了。

**请注意：**根据之前所作的假设，我们通常将基本情况视为一个常数级操作（即T(k) = t0，k ≤ n0，这里的t0与n0均为某些常数）。但在那些T的参数为n/b（这里的b为某些常数）的递归式中，我们可能会意外遇上另一个技术问题：即其参数必须要是一个整数。当然该问题可以通过舍入法（即向下取整或向上取整）来解决，但通常的做法是直接忽略掉相关的细节信息（直接假定n等于b的某次方）。如果不想那么草率的话，我们就需要用到本章稍后在“猜测与检验”一节中所介绍的方法，并用它来检验我们的答案。

（图）

图3-5. 完美平衡二叉树的一些属性概要

下面我们来看看递归式5。该递归式在每一半问题上都只有一次递归调用，外加一个额外的常数级操作。如果我们将整个递归过程当作一棵树（即*递归树*）的话，那么其额外操作（*f*(*n*)）应该就是树的各个节点上的操作，而其递归调用结构则用该树的各条边来表示。其总工作量（*T*(*n*)）就等于所有节点（或者说调用）上*f*(*n*)的操作总和。在这种情况下，由于各节点上的工作量都是常数，所以我们只需计算节点数即可。另外，由于我们每次执行*一次*递归调用，所以其整体操作量应该就等于根节点到某个叶节点的路径，所以*T*(*n*)明显应该是对数级的，但我们还是得看一看，如果我们逐步展开该递归式的话，情况究竟是怎样的；

（公式）

在这里，花括号部分与其上一行中的递归调用（*T*(…)）是等价的。然而，这种逐步展开（或重复代入）操作也只是我们解决方法中的第一步。它一般分为以下几个步骤：

1. 逐步展开递归式，一直到我们发现其中的模式为止。
2. 将该模式表示出来（通常会涉及到一个求和式），并用变量*i*来表示其行号。.
3. 根据“*i*层递归将会达到基本情况”这一条件，选择*i*的值（并解决该求和式）

现在第一步我们已经完成了。下面来看第二步：

（公式）

我希望您能同意，这一通用形式捕获了上述展开过程中递归式内在的模式：即我们每展开一层（或者说每向下推导一行），问题的规模就减少了一半（即除以2），并增加一个额外操作单元（加1）。虽然，您可能会觉得该递归式尾端的那个求和式有点傻。毕竟我们都清楚*i*个1之和等于*i*。但为了表现递归式的通用模式，我特意将其写成一个求和式。

要想到达递归式的基本情况，我们就必须要想方设法将*T*(*n*/2*i* )演变成*T*(1)。也就是说，我们要通过对半分的方式将n变成1，这一过程相信我们应该都很熟悉：其递归高度就是一个对数，即*i* = lg *n*。如果我们将其插入到之前的模式中去的话，我们的*T*(*n*)就可以直接用Θ(lg *n*)来表示了。

递归式6的展开过程也基本如此，只不过其中的求和式会更有趣一些：

（公式）

如果您还是无法看明白我们到达其通用模式的过程，可以停下来再仔细思考几分钟。（基本上，我们在这里只是使用了Σ符号来表示求和式*n* + *n*/2 + … + *n*/(2*i* – 1)，而后者早就出现在我们早先的展开步骤中了。）不过，在操心如何解决该求和式之前，我们需要再一次设置*i* = lg *n*。这样一来，假设*T*(1) = 1的话，我们就会得出以下关系：

（公式）

最后一步，因为正好*n*/2lg *n* = 1，所以我们可以将其作为单独的1加入到求和式中。

现在，您是否觉得这个求和式有些眼熟？让我们再次回到图3-5上来：如果*k*代表了树的某一高度，那么*n*/2*k* 就应该等于该高度上的节点数（从叶节点到根节点，我们所采用的都是减半策略）。这意味着该和值应该就等于其节点数：Θ(*n*)。

但在递归式7和8中，我们又将迎来新的一个小问题：多重递归调用。递归式7的情况与递归式5非常类似，只不过之前是*同一条*路径（从根节点到叶节点）上的节点计数问题，而现在我们各节点的各条子节点边都要跟踪，所以其实就等同于节点计数问题，或者说Θ(*n*)级问题。（其实您看出了吗？递归式6与7实际上只是在用两种不同的方式来对同一堆节点进行计数。）下面我们来看对递归式8使用的解决方案，该过程与递归式7非常类似（但值得检验一下）：

（公式）

如您所见，推导式前端的2在不停地堆积，以至于最终产生了一个2*i*因子。虽然括号部分看上去有点乱，但所幸这里的减半操作与倍增操作搭配得很完美：第一个括号由于有了*n*/2，于是得乘以2；而同样*n*/4得乘以4，以此类推，其通用形式就变成了*n*/2*i*乘2*i*，而这也意味着我们就剩下*i*个*n*的那部分了，也可简写成*n*·*i*。然后同样地，为了令其趋于基本情况，我们需将*i* = lg *n*代入其中：

（公式）

换句话说，其运行时间应该为Θ(*n* lg *n*)。您能出图3-5中看出这一结果吗？敢赌吗？首先递归树根节点上的操作时间为*n*，而在后面的两次递归调用（即子节点）中，各自所执行的都是减半操作。也就是说，其各节点上的操作时间就等于图3-5中的标签值。我们知道其每一行的和都等于*n*，并且一共有lg *n* +1行节点。由此我们可以得出其总和为*n* lg *n* + *n*，也就是Θ(*n* lg *n*)。

#### 猜测与检验

对于递归与数学归纳法这两者之间的概念关系，我们将会在第4章中做深入讨论。而我们在这里要提出的一个主要论点是它们彼此都像是对方的镜像：有一种观点认为归纳法能告诉我们递归操作为什么是正确的。在本节，我们将会局限于讨论如何证明一个递归式解决方案的正确性（而不会去讨论递归算法本身），但它应该依然带您一窥这些概念之间的彼此联接。

正如本章稍早前所提到的，在展开递归式并“发现”模式的过程中，我们通常会依赖于一些没有根据的假设性前提。例如我们通常会假设n是2的一个整数次方，这样的话，正好递归深度为lg *n*的操作能得以顺利实现。当然通常在大多数情况下，这些假设都是没有问题的，但为了确保一个解决方案的正确性，我们应该对它进行检验。能对解决方案进行检验的好处在于，这使得我们可以先根据自己猜测或直觉提出解决方案，然后再来证明其正确性。

请注意：为了简单起见，我们将会在后续内容中坚持使用大O记法，以表示相关操作的上线。当然，您也可以用类似方法来显示相关操作的下限（即用Ω或Θ记法）。

下面我们再来研究一下我们的第1个递归式，即*T*(*n*) = *T*(*n*–1) + 1。我们想要检验的是*T*(*n*)等同于Θ(*n*)的正确性。和（第1章中所讨论的）实验一样，我们其实是不能在这里使用渐进记法的；我们需要弄得更具体一些，往里插入一些常量，于是下面我们就来试图验证一下*T*(*n*) ≤ *cn*，其中*c*为任意常数且*c* ≥ 1。根据我们的标准假设，我们将设*T*(1) = 1。这目前看来是一切正常，但当*n*取值越来越大时情况又会如何呢？

这就是所谓的归纳法。这种方法的思路很简单：即先从*T*(1)开始入手，在这里我们的解决方案显然是正确的，然后我们再来证明其同样也适用于*T*(2)、*T*(3)等等。接着，我们以通用的形式来证明，也就是提供一个所谓*归纳步骤（induction step）*，证明如果我们的解决方案在*T*(*n*-1)上是正确的，那么其在*T*(*n*)上也应该是成立的（其中*n* > 1）。该步骤允许我们从*T*(1)推导出*T*(2)，然后再由*T*(2)推出*T*(3)，一直类推下去，而这正是我们想要的。

证明该归纳步骤的关键在于我们需要先假设（在这种情况下）其在*T*(*n*−1)上是正确的。这是我们推导*T*(*n*)的依据所在，我们称之为*归纳前提（inductive hypothesis）*。拿眼下这个例子来说，我们设置的归纳前提是 *T*(*n*–1) ≤ *c*(*n*–1)（其中c为某些常数），而我们想要证明的是该假设也适用于*T*(*n*)。

|  |  |  |
| --- | --- | --- |
|  | （公式）  请注意注释行对齐 | 我们假设*T*(*n*−1) ≤ *c*(*n*−1)  我们知道*c*≥ 1，所以−*c*+1 ≤ 0 |

在这里，我们特地用方框标出归纳前提的使用之处：即我们用*c*(*n*–1)替换掉了*T*(*n*–1)，（根据归纳前提）我们知道这是一个更大（或一样大）的值。这样做可以使相关的替换操作更安全，只要我们将第一、二行之间的等号换成“小于等于”即可。在经历一些基本代数运算后，从*T*(*n*–1) ≤ *c*(*n*–1)的假设中推导出*T*(*n*) ≤ *cn*的全过程就呈现在我们眼前了，并且其还（因此）可以继续推导出*T*(*n*+1) ≤ *c*(*n*+1)等。现在，我们由基本情况*T*(1)开始着手，证明了通用模式中的*T*(*n*)就等于Θ(*n*)。

那么基于分治法的递归式会更难吗？下面就让我们来看看（表3-1中的）递归式8的情况。这一次，我们要使用一种被称之为*强归纳*（*strong induction*）的方法。在之前的例子中，我们所做的假设都是关于前一个值（例如*n*-1，这种方法叫做*弱归纳，weak induction*）的，现在我们的归纳前提将是“所有比它小的数字”。更具体地说，我们假设*T*(*k*) ≤ *ck* lg *k*（k为正整数且*k* < *n*），然后证明由该假设可以推导出*T*(*n*) ≤ *cn* lg *n*。其基本思路依旧是相同的——我们的解决方案依然会经历从*T*(1)到*T*(2)这一路的“擦除式”推导过程——只是我们的工作量稍微多了一些。（尤其我们现在的前提假设将会是一些与`*T*(*n*/2)有关的内容，这就不会像*T*(*n*-1)那么简单了。）下面让我们开始吧：

|  |  |  |
| --- | --- | --- |
|  | （公式）  请注意注释行对齐 | 假设*T*(*k*) ≤ *c*(*k*lg*k*)，其中*k*=*n*/2 < *n*  lg(*n*/2) = lg*n*−lg2  lg2 = 1  只要令*c*= 2 |

与之前一样，通过假设我们的结果对于较小的参数是成立的，我们就成功地证明了*T*(*n*)也适用于该假设。

**提醒：**请小心递归式中，尤其是其递归部分的渐进记法。我们可以来想一想下面这个（假）“论证”：如果*T*(*n*) = 2*T*(*n*/2) + *n*的意思是*T*(*n*)等同于*O*(*n*)，那么我们就可以直接在我们的归纳前提中使用大O记法：  
（公式）

这里存在着许多错误，但或许其中最突出的问题是：归纳前提所需要的是各种具体的参数值（如k = 1, 2…），但渐进记法显然与整个函数有点格格不入。

|  |
| --- |
| **跳进兔子洞[[12]](#footnote-12)（或者说更改我们的变量）** |
| **先提醒一句，这篇专栏材料可能会带有一点挑战性。如果现在您的脑子里已经塞满了各种递归式概念，那么或许等晚些时候再来重新面对它是一个不错的想法。**  **在某些情况下（尽管可能并不多见），我们可能会遇到一些类似于这样的递归式：**  **（公式）**  **换句话说，该子问题的大小是原先规模的*b*次方根。那*现在*您要怎么做呢？事实上，我们可以穿越“另一个世界”里去，在那里递归式也许会变得非常简单。当然，那个世界必须是真实世界的某种映射。只有如此，当我们回归现实时才能真正得到处理原递归式的解决方案。**  **我们的“兔子洞”其实就是一种被称之为*更改变量法*的东西。它实际上是一个协调变化过程，即我们用替换T（比如换成S）和n（换成m）的方式，让我们的递归式真正变回之前的样子——其实我们只是换了一种不同的书写方式而已。那么，到底我们从*T*(*n*1/*b* )到*S*(*m*/*b*)的过程中究竟要改变什么，才能使操作简化呢？下面我们就拿平方根做一个例子：**  **（公式）**  **那么如何才能让*T*(*n*1/2) = *S*(*m*/2)呢？直觉告诉我们想要把乘方运算变成乘法运算，我们需要用到对数运算。具体技巧是先设置*m* = lg *n*，接着我们就可以将递归式中的*n*全都替换成2*m*：**  **（公式）**  **然后再设置*S*(*m*) = *T*(2*m* )，将相关指数都隐藏起来，瞧！我们这就进入仙境了：**  **（公式）**  **现在问题解决起来就简单了：*T*(*n*) = *S*(*m*) ，它是 Θ(*m* lg *m*) = Θ(lg *n* · lg lg *n*)。**  **在本专栏最先出现的那个递归式中，常数*a*和*b*当然还是可以有其它值的（而且*f*可能会更不协调），我们会得到*S*(*m*) = *aS*(*m*/*b*) + *g*(*m*)（其中*g*(*m*) = *f* (2*m* )）。我们既可以用重复代入法来对付它，也可以使用下一节将会介绍的一刀切式（cookie-cutter）的解决方案，因为两者都非常适合用来解决这一类递归式。** |

#### 主定理：一刀切式的解决方案

递归式与所谓的分治法（后者详见第6章）之间存在着许多对应关系，其一般形式如下（其中*a* ≥ 1且*b* > 1）：

**（公式）**

其主要想法是：假设我们有*a*重递归调用，每重调用处理掉一定比例的数据（数据集的1/b）。除了这些递归调用外，算法中还有一个额外的*f*(*n*)操作单元。我们可以用图3-6来诠释一下该算法。在之前的树结构中，数字2是唯一的重要常数，但现在我们的重要常量有*两个*：*a*和*b*。即对我们所下到各个树层来说，分配到其每一个节点上的问题规模都还必须得除以*b*，这就意味着我们如果想让（叶节点上的）问题规模为1的话，该树的高度就必须等于log*b* *n*。（记住，为了获得*n*的值，我们必须要让*b*的指数提高到这个值。）

但该树每个内部节点下面都会有*a*个子节点，因此逐层增加之后，其节点数未必就（一定）能抵消掉问题规模上的递减。也就是说，其叶节点数未必就（一定）等于*n*。相反，如果直接按各节点上的倍增因子*a*，及树高log*b* *n*来算，其树宽值应该等于*a* log*b n*。但由于对数运算规则允许*a*和*n*之间相互交换，所以我们可以得出其叶节点数为*n*log*b**a*（在习题3-10中，我们会让您来证明其正确性）。

本节的最终目的是要建立三个一刀切式的解决方案，并统一成为所谓的*主定理（master theorem）*[[13]](#footnote-13)。这些解决方案分别对应了三种可能的情况：有可能其大部分操作都运行在（也就是大部分运行时间消耗在）*根节点*上的；也有可能是在*叶节点*上，或者有可能*均匀分布*在该递归树的各行之间。下面就让我们逐个来分析这三种情况：

（图）

**图3-6.** 用一颗完美平衡的多路（a路）树来诠释分治递归式。

在第一种情况中，其大部分操作是在根节点上执行的，这里“大部分”的意思是说它在整体操作时间的渐进表示中是占主导地位的，其既定总运行时间为Θ(*f*(*n*))。但如何才能确定根节点的占主导地位呢？如果树上的操作量是随某个常数因子而逐层递减的，并且根节点上的操作（即渐进时间）始终多于叶节点的话，基本上就是属于这种情况了。更正式地说：

（公式）

其中常数*c* < 1，且n得够大，并且：

（公式）

其中*ε*为常数。这就意味着*f*(*n*)的增长趋势将*严格快于*叶节点数的增长（这也是为什么我们要在叶节点计数公式的指数值上加*ε*）。下面我们来看一个具体示例：

（公式）

在这里*a* = 2、*b* = 3且*f*(*n*) = *n*。因而如果想计算叶节点数的话，我们只要计算出log3 2的值即可。过去在标准计算器中，我们常会用log 2/log 3这种表达式来计算，而在Python中只需调用math模块中的log函数就行了，并且您还会发现其实log(2,3)的值是略小于0.631的。换句话说，我们在这里想了解的是*f*(*n*) = *n*是否属于Ω(*n*0.631)，这一点显然是确定的，由此我们就可以得出*T*(*n*) = Θ(*f*(*n*)) = Θ(*n*)。对此，我们还有一种更快捷的方式：如果您看到*b*是大于*a*的，就立即可以确定*n*属于表达式的主导部分了（能明白这是为什么吗？）。

下面，我们可以将上面的根-叶关系翻转一下：

（公式）

现在画面上的主导换成叶节点了。那么您认为其最终的总运行时间会是多少呢？正确答案是：

（公式）

下面我们再来看一个递归式示例：

（公式）

由于这里*a* = *b*，所以*n*的值就是我们所得到的叶节点数，显然它的渐进增长趋势是快于*f*(*n*) = lg *n*的，这意味着其最终渐进运行时间就是计算叶节点数的时间Θ(*n*)。

**请注意：**为了确定根节点的主导地位，我们必须额外要求其满足af(n/b) ≤ cf(n)，且c < 1。而确立叶节点的主导地位则没有类似的条件。

最后一种情况是根节点与叶节点上的操作时间具有相同的渐进增长趋势：

（公式）

这样一来，事情就变成了求该树各层操作之和的求和式（因为从根节点到叶节点，操作量始终既没有增加也没有减少），这意味着我们可以通过乘以其高度的方式来计算其总和：

（公式）

下面我们接着来看递归式示例：

（公式）

平方根运算看着似乎很吓人，但其实只是指数*n*0.5的另一种形式。由于我们这里设*a* = 2、*b* = 4，因此可以得出log*b* *a* = log4 2 = 0.5。这样一来，我们就知道——该树根节点、叶节点以及各层节点上的操作时间应该都等于Θ(*n*0.5)，于是其产生的总运行时间应该是：

（公式）

在表3-2中，我们对主定理中的三种情况做了汇总，将它们按照习惯顺序排列：情况1是叶节点占主导地位时的状况；情况2是各层操作量相同时的“死竞争”状况；情况3是根节点占主导时的状况。

**表3-2**. 主定理中的三种情况

|  |  |  |  |
| --- | --- | --- | --- |
| 情况 | 前提条件 | 解决方案 | 相关示例 |
| 1 | *f* (*n*) ∈*O*(*n*log*ba*−*ε*) | *T*(*n*) ∈Θ(*n*log*ba*) | *T*(*n*) = 2*T* (*n*/2)+lg*n* |
| 2 | *f* (*n*) ∈Θ(*n*log*ba*) | *T*(*n*) ∈Θ(*n*log*ba* lg*n*) | *T*(*n*) = 2*T* (*n*/4)+ |
| 3 | *f* (*n*) ∈Ω(*n*log*ba*+*ε*) | *T*(*n*) ∈Θ(*f* (*n*)) | *T*(*n*) = 2*T* (*n*/3)+*n* |

### 那么，***这一切***究竟是什么呢？

OK，数学内容我们说得已经不少了，但代码到目前为止还没见着多少。接下来，我们就要来看看上面这些公式究竟有什么用。首先，请花点时间思考一下清单3-1及3-2中的Python程序。[[14]](#footnote-14)（您可以在清单6-6中找到mergesort函数的完整注释版）我们就假设这里都是一些新算法，所以也不必上网去搜索它们的名字了，您现在的任务是要判读出它们哪一个的运行时间复杂度比较好。

**清单3-1.** 排序算法之：侏儒排序法

def gnomesort(seq):

i = 0

while i < len(seq):

if i == 0 or seq[i-1] <= seq[i]:

i += 1

else:

seq[i], seq[i-1] = seq[i-1], seq[i]

i -= 1

**清单3-2.** 排序算法之：归并排序法

def mergesort(seq):

mid = len(seq)//2

lft, rgt = seq[:mid], seq[mid:]

if len(lft) > 1: lft = mergesort(lft)

if len(rgt) > 1: rgt = mergesort(rgt)

res = []

while lft and rgt:

if lft[-1] >= rgt[-1]:

res.append(lft.pop())

else:

res.append(rgt.pop())

res.reverse()

return (lft or rgt) + res

侏儒排序法（gnome sort）中只有一个简单的while循环和一个索引范围为0到len(seq)-1的索引变量。这就容易会让人认为它是一个线性时间的算法，但这个结论被最后一行中i -= 1语句给否定了。要想弄清楚该算法究竟运行了多长时间，我们就必须搞懂它工作过程中的一些细节。起初，我们会从左边开始扫描（即持续递增i），直至找到一个seq[i-1]大于seq[i]的位置i，这时两个值顺序就错了，于是else部分边开始生效。

然后else分支会持续交换seq[i]与seq[i-1]的值并递减i，直到其再次恢复之前seq[i-1] <= seq[i]的顺序（或当前位置为0）。换句话说，该算法会沿着目标序列交叉向上扫描来发现错了位（也就是太小）的元素，并将它们下移到合适位置（经反复交换）。那么其整体开销是多少呢？让我们先忽略掉其平均值，只关注最好的与最坏的情况。最好的情况当然是目标序列已经排好序了，这时gnomesort只需将整个序列扫描一遍即可，并不会发现任何错位，然后便停止了，其运行时间应为Θ(*n*)。

而最坏的情况就没那么简单了，但也没有多复杂。需要注意的是，通常每当我们发现一个元素错位时，这之前的所有元素都应该已经排好序了。所以当我们将新元素移动到合适位置时是不会发生冲突的。这也就是说我们每纠正一个错位元素，已排序元素的数量就会有所增加，并且下一个错位元素会出现在更右的位置上。所以可能的最坏情况就是查找并移动这些错位元素的开销与其位置形成了正比关系。因而最糟糕的运行时间就应该是1 + 2 + … + *n*–1，也就是Θ(*n*2)。这目前只是一个假设——我们这里只是证明了不会有比这个更糟的情况了，但事情真会如此糟糕吗？。

恐怕确实是会的：我们可以考虑一下元素降序排列时的情况（即让其顺序与我们所想的相反）。这时所有元素都错位了，我们得将每一个元素移动到起始位置，自然就需要平方级运行时间。所以通常来说，侏儒排序的运行时间应该用Ω(*n*)与*O*(*n*2)来表示，它们分别代表了最好与最坏情况的严格边界。

现在，我们来看看归并排序（见清单3-2）。它要比侏儒排序稍微复杂一些，所以我们会把一些与该排序事宜相关的解释留到到第6章。幸运的是，即便在不了解该算法工作过程的情况下，我们也是可以对其运行时间进行分析的。只要您观察一下其整体结构：其输入（即seq）规模为n，有两重递归调用，每次子问题的规模为*n*/2（尽可能让其接近整数规模）。除此之外，while循环以及res.reverse()中也包含了某些操作。我们会在习题3-11中让您证明这是一个Θ(*n*)级操作。（在习题3-12中，我们还会让您说明如果将pop()替换成pop(0)时会发生什么。）这正好就是我们著名的第8号递归式*T*(*n*) = 2*T*(*n*/2) + Θ(*n*)，这就说明了归并排序的运行时间应该为Θ(*n* lg *n*)（无论其输入情况如何）。这意味着如果面对的是几乎已经排好序的数据，我们或许会更喜欢用侏儒排序，但在一般情况下我们通常都会倾向于用更强大的合并排序来解决问题。

**请注意：**Python中的排序算法timsort，其实是归并排序的一种自然适应版。它在保留最糟的、线性对数级时间的同时，兼具实现了最佳的、线性级的运行时间。我们将会在第6章的timsort专栏中更详细地讨论这一算法。

### 本章小结

*n*个整数相加的求和式是平方级的，而lg *n*个2的指数相加的求和式则是线性级的。其中的第一个我们可以用循环赛制来诠释，它的*n*个元素都会被两两配对；而第二个则与淘汰赛有些关联，它有lg *n*轮比赛，并且除赢者外所有人都会被淘汰。另外，*n*个数的排列数等于*n*!，而从*n*个数中取*k*个的组合数（即其大小为*k*的子集数）通常被写作*C*(*n*,*k*)，等于*n*!/(*k*!·(*n*–*k*)!)。这也就是所谓的*二项式系数*。

通常，我们将一个函数的（直接或通过其它函数）自我调用操作称之为递归。而递归关系则是一种用于反映函数自身关联的等式，表现为某种递归形式（譬如*T*(*n*) = *T*(*n*/2) + 1），这些等式常会被用来描述递归算法的运行时间。并且为了解决这些等式，我们必须要对递归的基本情况做一些假设。正常情况下，我们通常会假设*T*(*k*)等于Θ(1)（其中k为常数）。本章所介绍的递归式解法主要有三种：（1）反复运用原始等式以展开*T*的递归部分，直至发现其中的模式；（2）对解决方案进行假设性猜测，然后再用归纳法证明其正确性；（3）对符合某种主定理情况的分治递归式，直接采用相应的解决方案。

### 如果您有兴趣……

本章所讨论的话题（以及前一章中的相关内容）通常应该被归类为*离散数学（discrete mathematics）*的一部分。[[15]](#footnote-15)写这一类话题的书非常多，而且其中大部分我都觉得很酷，如果您喜欢的话，可以随意到本地图书馆或线上书店去找找，我敢说这些书肯定会花去您不少时间。

有一本书我在处理计数运算以及相关证明时经常会用到（但这并不是一般意义上的离散数学），书名叫《*Proofs That Really Count*》（由Benjamin和Quinn合著），这本书还是值得看看的。而如果您是想找一本关于求和式处理、组合问题、递归式问题以及其它实质内容的、专门为计算机科学领域而编写的书，当然就要选择Graham, Knuth和Patashnik的那本经典《*Concrete Mathematics*》（Yeah，就是*那个*Knuth，地球人都知道！）。如果您只想一个用于查找求和式解决方案的地方，或许可以试试我们之前提到过的那个网站：Wolfram Alpha（<http://wolframalpha.com>），或者找一本全面的递归公式参考手册放在口袋里（可能您得再去一次书店）。

如果您想了解更多关于递归式方面的知识，可以任选一本我们在第1章中所列举的那些算法教科书，查看一下这方面的标准方法，或者您也可以自行研究出一些更先进的方法，以便我们处理更多的递归式类型，超越我们在这里所学到的内容。例如，《*Concrete Mathematics*》中就曾介绍过一种被称之为*生成函数（generating functions）[[16]](#footnote-16)*的东西及其用法。此外，如果您上网四处看看，也一定能找到一些用*零化子（annihilators）*或者*Akra-Bazzi定理来*解相关递归式的有趣资料。

在本章稍早前那个关于伪多项式（时间）算法的专栏中，我们曾举过一个质数检查的例子。许多（老式）教科书都宣称这是一个尚未解决的问题（也就是说，不知道它能不能用多项式算法来解决）。现在您知道——事情不再是这样了。2002年，Agrawal、Kayal和Saxena发表了他们那篇开创性的论文《PRIMES is in P》，里面描述了用多项式算法来解决质数检查问题的具体方法（奇怪的是，因数分解却依然是一个待解决的问题）

### 练习题

3-1. 请证明“求和式的运用”一节中所描述的那些特性是正确的。

3-2. 请用第2章中的规则证明*n*(*n*–1)/2属于Θ(*n*2)。

3-3. 2的最小的k个非负指数项相加的和等于2*k*+1 – 1，请用该特性证明所有正整数都可以用二进制来表示。

3-4. 在“龟兔赛跑”一节中，我们介绍两种查找数字的方法。现在请将这两个方法转换成猜数字算法，并用Python实现出来。

3-5. 请证明*C*(*n*,*k*) = *C*(*n*,*n*–*k*)。

3-6. 在“递归与递归式”这一节中，我们早先曾在递归函数S中假设过，如果它没有使用位置参数i，而是直接让函数返回sec[0] + S(seq[1:])的话，该函数的渐进运行时间又会是什么？

3-7. 请用重复代入法求解表3-1中的递归式2。

3-8. 请用重复代入法求解表3-1中的递归式3。

3-9. 请用重复代入法求解表3-1中的递归式4。

3-10.请证明无论对数的底是什么，都有*x* log *y* = *y* log *x*。

3-11.请证明在清单3-2的归并排序实现中，*f*(*n*)属于Θ(*n*)。

3-12.在清单3-2那一版的归并排序中，对象是从序列每一半的尾端弹出（经由pop()）的。或许选择从首端弹出（经由pop(0)）会更直观一点，以免我们要对res进行反转处理（而且也更符合我们实际的生活习惯），但pop(0)（跟insert(0)一样）是一个线性级操作，而pop()则是常数级的。那么切换这两种调用会对整体运行时间产生什么影响呢？

### 参考资料

* Agrawal, M., Kayal, N., and Saxena, N. (2004). *PRIMES is in P*. The Annals of Mathematics, 160(2):781793.
* Akra, M. and Bazzi, L. (1998). *On the solution of linear recurrence equations*. Computational Optimization and Applications, 10(2):195-210.
* Benjamin, A. T. and Quinn, J. (2003). *Proofs that Really Count: The Art of Combinatorial Proof*. The Mathematical Association of America.
* Graham, R. L., Knuth, D. E., and Patashnik, O. (1994). *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley Professional, second edition.

1. 译者注：约翰·卡尔·弗里德里希·高斯（1777年4月30日－1855年2月23日），德国著名数学家、物理学家、天文学家、大地测量学家。高斯被认为是历史上最重要的数学家之一，并有“数学王子”的美誉。 [↑](#footnote-ref-1)
2. 译者注：希腊字母表中的18个字符，常被音译为“西格玛（sigma）”。 [↑](#footnote-ref-2)
3. 译者注：该公式在数学上被叫做等差级数（arithmetic series），即一个等差数列的和等于其首项与末项的和乘以项数除以2。相传是由著名数学家Gauss小时候所发现的，但经历史学家考证，在更早的时候，古希腊甚至古埃及就已经有人在使用这种运算公式了。 [↑](#footnote-ref-3)
4. <http://prb.org/Articles/2002/HowManyPeoplehaveEverLivedonEarth.aspx> [↑](#footnote-ref-4)
5. 如果这真的成立，那该人口数中可能还包含了32代前的那对男女……，但正如我说的，就当它是个游戏吧。 [↑](#footnote-ref-5)
6. 译者注：一种猜谜游戏，游戏允许玩家提问题，但被问者只回答yes或者no，而玩家必须在二十个问题之内猜出答案。 [↑](#footnote-ref-6)
7. 译者注：这里是指国际象棋的棋盘。 [↑](#footnote-ref-7)
8. 据传闻，这本来是国际象棋的创建者所要求并被授予的奖励……但他被要求数清这里的每一粒粮食。我猜，他改主意了。 [↑](#footnote-ref-8)
9. 译者注：在计算理论领域中，若一个数值算法的时间复杂度可以表示为输入数值N的多项式，则称其时间复杂度为伪多项式时间。由于N的值是N的位数的幂，故该算法的时间复杂度实际上应视为输入数值N的位数的幂。 [↑](#footnote-ref-9)
10. 您知道原本指数中的-1到哪儿去了吗？请记住，2a+b = 2a \* 2b…… [↑](#footnote-ref-10)
11. 另外，“二项式系数”这个名称来历也没有那么明显，您可能需要了解一下，它的定义很简洁。 [↑](#footnote-ref-11)
12. 译者注：这里引用了《爱丽丝漫游仙境》中的典故，该故事在西方几乎家喻户晓，其主人公爱丽丝就是因为掉进了兔子洞而来到了一个奇幻世界。 [↑](#footnote-ref-12)
13. 译者注：在算法分析中，主定理（master theorem）提供了用渐近符号表示许多由分治法得到的递推关系式的方法。此方法经由经典算法教科书《算法导论》而为人熟知。不过，并非所有递推关系式都可应用主定理。该定理的推广形式包括Akra-Bazzi定理。 [↑](#footnote-ref-13)
14. 归并排序是一个经典算法，诞生于1945年，是由计算机科学界的传奇John von Neumann在EDVAC（译者注：Electronic Discrete Variable Automatic Computer的缩写，它被认为是世上第一批可编程电子计算机的代表）上首先实现出来的。您在第6章中还会看到更多类似的算法。而侏儒排序则是由Hamid Sarbazi-Azad在2000年提出的，也叫愚人排序法（stupid sort）。 [↑](#footnote-ref-14)
15. 如果您不太分得清楚*discrete*和*discreet*这两个词，或许应该去查一下资料。 [↑](#footnote-ref-15)
16. 译者注：该书第7章。 [↑](#footnote-ref-16)