调和级数

早在14世纪，[尼克尔·奥里斯姆](http://zh.wikipedia.org/wiki/%E5%B0%BC%E5%85%8B%E7%88%BE%C2%B7%E5%A5%A7%E9%87%8C%E6%96%AF%E5%A7%86" \o "尼克尔·奥里斯姆)(Nicole Oresme)已经证明调和级数发散，但这项成就并未为世人所知。17世纪时，[皮耶特罗·曼戈里](http://zh.wikipedia.org/w/index.php?title=%E7%9A%AE%E8%80%B6%E7%89%B9%E7%BD%97%C2%B7%E6%9B%BC%E6%88%88%E9%87%8C&action=edit&redlink=1" \o "皮耶特罗·曼戈里)(Pietro Mengoli)、[约翰·伯努利](http://zh.wikipedia.org/wiki/%E7%BA%A6%E7%BF%B0%C2%B7%E4%BC%AF%E5%8A%AA%E5%88%A9" \o "约翰·伯努利)和[雅各布·伯努利](http://zh.wikipedia.org/wiki/%E9%9B%85%E5%90%84%E5%B8%83%C2%B7%E4%BC%AF%E5%8A%AA%E5%88%A9)完成了全部证明工作。

调和序列历来很受建筑师重视；这一点在[巴洛克](http://zh.wikipedia.org/wiki/%E5%B7%B4%E6%B4%9B%E5%85%8B" \o "巴洛克)时期尤其明显。当时建筑师在建造教堂和宫殿时，运用调和序列为楼面布置和建筑物高度建立比例，并使室内外的建筑细节间呈现和谐的联系。

对刚接触这个级数的人而言，调和级数是违反直觉的——尽管随着n不断增大，1/n无限接近0，但它却是一个[发散级数](http://zh.wikipedia.org/wiki/%E5%8F%91%E6%95%A3%E7%BA%A7%E6%95%B0" \o "发散级数)。调和级数也因此成为一些[佯谬](http://zh.wikipedia.org/wiki/%E4%BD%AF%E8%AC%AC)的原型。“橡皮筋上的蠕虫”就是其中一个例子。[[2]](http://zh.wikipedia.org/wiki/%E8%B0%83%E5%92%8C%E7%BA%A7%E6%95%B0#cite_note-autogenerated258-1)假设一条蠕虫沿着一条1米长的橡皮筋爬行，而橡皮筋每分钟之后均匀伸展1米。如果相对于其所在的橡皮筋，蠕虫的爬行速度是每分钟1厘米，那么它最终会到达橡皮筋的另一头吗？与直觉相反，答案是肯定的：n分钟之后，蠕虫爬行过的距离与橡皮筋总长度的比值为：

\frac{1}{100}\sum_{k=1}^n\frac{1}{k}.

由于调和级数发散（证明见本条目“[发散性](http://zh.wikipedia.org/wiki/%E8%B0%83%E5%92%8C%E7%BA%A7%E6%95%B0#.E5.8F.91.E6.95.A3.E6.80.A7)”一节），即n趋于无穷大时级数也趋于无穷大，所以这个比值也必定在某个时刻超过1；也就是说，蠕虫最终一定会到达橡皮筋另一头。然而，在这个时刻的n的值极其之大，约为*e*100，超过1040（1后面有40个零）。这也说明了，尽管调和级数确确实实是发散的，但它发散的速度非常慢。

另一个例子：假设你有一堆完全相同的骨牌，可以肯定的是，你可以把它们叠在一起，并使得每个骨牌都突出其下方骨牌外一定长度，最终使得最上层的骨牌完全在最底层骨牌以外甚至更远。违反直觉的是，只要你的骨牌足够多，你就可以使最上层的骨牌可以离最底层骨牌无穷远。[[2]](http://zh.wikipedia.org/wiki/%E8%B0%83%E5%92%8C%E7%BA%A7%E6%95%B0" \l "cite_note-autogenerated258-1)[[3]](http://zh.wikipedia.org/wiki/%E8%B0%83%E5%92%8C%E7%BA%A7%E6%95%B0#cite_note-2)一个较简单的证明如下：

## 三种排序算法

**快速排序**是由[东尼·霍尔](http://zh.wikipedia.org/wiki/%E6%9D%B1%E5%B0%BC%C2%B7%E9%9C%8D%E7%88%BE)所发展的一种[排序算法](http://zh.wikipedia.org/wiki/%E6%8E%92%E5%BA%8F%E7%AE%97%E6%B3%95)。在平均状况下，排序 *n* 个项目要**[Ο](http://zh.wikipedia.org/wiki/%E5%A4%A7O%E7%AC%A6%E5%8F%B7" \o "大O符号)**(*n* log *n*)次比较。在最坏状况下则需要**Ο**(*n*2)次比较，但这种状况并不常见。事实上，快速排序通常明显比其他**Ο**(*n* log *n*) 算法更快，因为它的内部循环（inner loop）可以在大部分的架构上很有效率地被实作出来，且在大部分真实世界的资料，可以决定设计的选择，减少所需时间的二次方项之可能性。

步骤为：

1. 从数列中挑出一个元素，称为 "基准"（pivot），
2. 重新排序数列，所有元素比基准值小的摆放在基准前面，所有元素比基准值大的摆在基准的后面（相同的数可以到任一边）。在这个分割结束之后，该基准就处于数列的中间位置。这个称为**分割（partition）**操作。
3. [递归](http://zh.wikipedia.org/wiki/%E9%80%92%E5%BD%92)地（recursive）把小于基准值元素的子数列和大于基准值元素的子数列排序。

递回的最底部情形，是数列的大小是零或一，也就是永远都已经被排序好了。虽然一直递回下去，但是这个算法总会结束，因为在每次的迭代（iteration）中，它至少会把一个元素摆到它最后的位置去。

**快速排序的最直接竞争者是[堆排序](http://zh.wikipedia.org/wiki/%E5%A0%86%E6%8E%92%E5%BA%8F" \o "堆排序)（Heapsort）**。堆排序通常比快速排序稍微慢，但是最坏情况的执行时间总是[O](http://zh.wikipedia.org/wiki/%E5%A4%A7O%E7%AC%A6%E5%8F%B7" \o "大O符号)(*n* log *n*)。快速排序是经常比较快，除了introsort变化版本外，仍然有最坏情况效能的机会。如果事先知道堆排序将会是需要使用的，那么直接地使用堆排序比等待 introsort 再切换到它还要快。堆排序也拥有重要的特点，仅使用固定额外的空间（堆排序是原地排序），而即使是最佳的快速排序变化版本也需要Θ(log *n*)的空间。然而，堆排序需要有效率的随机存取才能变成可行。

**快速排序也与**[**归并排序**](http://zh.wikipedia.org/wiki/%E5%BD%92%E5%B9%B6%E6%8E%92%E5%BA%8F)**（Mergesort）竞争**，这是另外一种递回排序算法，但有坏情况O(*n* log *n*)执行时间的优势。不像快速排序或堆排序，归并排序是一个[稳定排序](http://zh.wikipedia.org/wiki/%E7%A9%A9%E5%AE%9A%E6%8E%92%E5%BA%8F)，且可以轻易地被采用在[链表](http://zh.wikipedia.org/wiki/%E9%93%BE%E8%A1%A8)（linked list）和储存在慢速存取媒体上像是[磁盘储存](http://zh.wikipedia.org/w/index.php?title=%E7%A3%81%E7%A2%9F%E5%84%B2%E5%AD%98&action=edit&redlink=1)或[网络连接储存](http://zh.wikipedia.org/wiki/%E7%B6%B2%E8%B7%AF%E9%80%A3%E6%8E%A5%E5%84%B2%E5%AD%98)的非常巨大数列。尽管快速排序可以被重新改写使用在炼串行上，但是它通常会因为无法随机存取而导致差的基准选择。归并排序的主要缺点，是在最佳情况下需要Ω(n)额外的空间。