有必要对算法3的程序进行一些说明。递归过程调用的一般形式是传递输入的数组以及左边界和右边界，它们界定了数组要被处理的部分。单行驱动程序通过传递数组以及边界0和$N-1$而将该过程启动。

第8行至第12行处理基准情况。如果left==right，那么只有一个元素，并且当该元素非负时它就是最大子序列。left>right的情况是不可能出现的，除非N是负数(不过，程序中小的扰动有可能致使这种混乱产生)。第15行和第16行执行两个递归调用。我们可以看到，递归调用总是对小于原问题的问题进行，不过程序中的小扰动有可能破坏这个特性。第18行至第24行以及第26行至第32行计算达到中间分界处的两个最大和的和数。这两个值的和为扩展到左右两部分的最大和。例程max3(未给出)返回这三个可能的最大和中的最大者。

显然，算法3需要比前面两种算法更多的编程努力。然而，程序短并不总意味着程序好。正如我们在前面显示算法运行时间的表中已经看到的，除最小的输入量外，该算法比前两个算法明显要快。

对运行时间的分析方法与在分析计算斐波那契数程序时的方法类似。令$T(N)$是求解大小为N的最大子序列和问题所花费的时间。如果$N=1$，则算法3执行程序第8行到第12行花费某个常数时间量，我们称之为一个时间单位。于是，$T(1)=1$。否则，程序必须运行两个递归调用，即在第19行和第32行之间的两个for循环，以及某个小的簿记量，如第14行和第18行。这两个for循环总共接触到从A_0到A_{N-1}的每一个元素，而在循环内部的工作量是常量，因此，在第19到32行花费的时间为$O(N)$。在第8行到第14行，第18、26和34行上的程序的工作量都是常量，从而与$O(N)$相比可以忽略。其余就是第15、16行上运行的工作。这两行求解大小为$N/2$的子序列问题(假设N是偶数)。因此，这两行每行花费$T(N/2)$个时间单元，共花费$2T(N/2)$个时间单元。算法3花费的总的时间为$2T(N/2)+O(N)$。我们得到方程组

$$T(1)=1$$

$$T(N)=2T(N/2)+O(N)$$

为了简化计算，我们可以用N代替上面方程中的$O(N)$项；由于$T(N)$最终还是要用大O来表示，因此这么做并不影响答案。在第7章，我们将会看到如何严格地求解这个方程。至于现在，如果$T(N)=2T(N/2)+N$，且$T(1)=1$，那么$T(2)=4=2*2$，$T(4)=12=4*3$，$T(8)=32=8*4$，以及$T(16)=80=16*5$。其形式是显然的并且可以得到，即若$N=2^k$，则$T(N)=N*(k+1)=N\log N+N=O(N\log N)$。

这个分析假设N是偶数，否则$N/2$就不确定了。通过该分析的递归性质可知，实际上只有当N是2的幂时结果才是合理的，否则我们最终要得到大小不是偶数的子问题，方程就是无效的了。当N不是2的幂时，我们多少需要更加复杂一些的分析，但是大O的结果是不变的。

在后面的章节中，我们将看到递归的几个漂亮的应用。这里，我们还是介绍求解最大子序列和的第4种方法，该算法实现起来要比递归算法简单而且更为有效。它在图2-8中给出。

不难理解为什么时间的界是正确的，但是要明白为什么算法是正确可行的却需要多加思考。为了分析原因，注意，像算法1和算法2一样，j代表当前序列的终点，而i代表当前序列的起点。碰巧的是，如果我们不需要知道具体最佳的子序列在哪里，那么i的使用可以从程序上被优化，因此在设计算法的时候假设i是需要的，而且我们想要改进算法2。一个结论是，如果a[i]是负的，那么它不可能代表最优序列的起点，因为任何包含a[i]的作为起点的子序列都可以通过用a[i+1]作起点而得到改进。类似地，任何负的子序列不可能是最优子序列的前缀(原理相同)。如果在内循环中检测到从a[i]到a[j]的子序列是负的，那么可以推进i。关键的结论是，我们不仅能够把i推进到i+1，而且实际上还可以把它一直推进到j+1。为了看清楚这一点，