```
public static long factorial( int n )
{
    if( n <= 1 )
        return 1;
    else
        return n * factorial( n - 1 );
}
```

实际上这个例子对递归的使用并不好。当递归被正常使用时，将其转换成一个循环结构是相当困难的。在这种情况下，分析将涉及求解一个递推关系。为了观察到这种可能发生的情形，考虑下列程序，实际上它对递归使用的效率低得令人惊诧。

```
            public static long fib( int n )
            {
1               if( n <= 1 )
2                   return 1;
                else
3                   return fib( n - 1 ) + fib( n - 2 );
            }
```

初看起来，该程序似乎对递归的使用非常聪明。可是，如果将程序编码并在 N 值为 40 左右时运行，那么这个程序让人感到效率低得吓人。分析是十分简单的。令 $T(N)$为调用函数 fib(n)的运行时间。如果 $N=0$ 或 $N=1$，则运行时间是某个常数值，即第 1 行上做判断以及返回所用的时间。因为常数并不重要，所以我们可以说 $T(0)=T(1)=1$。对于 N 的其他值的运行时间则相对于基准情形的运行时间来度量。若 $N>2$，则执行该方法的时间是第 1 行上的常数工作加上第 3 行上的工作。第 3 行由一次加法和两次方法调用组成。由于方法调用不是简单的运算，因此必须用它们自己来分析它们。第一次方法调用是 fib(n－1)，从而按照 T 的定义它需要 $T(N-1)$ 个时间单元。类似的论证指出，第二次方法调用需要 $T(N-2)$个时间单元。此时总的时间需求为 $T(N-1)+T(N-2)+2$，其中 2 指的是第 1 行上的工作加上第 3 行上的加法。于是对于 $N \geqslant 2$，有下列关于 fib(n)的运行时间公式：

$$T(N)=T(N-1)+T(N-2)+2$$

但是 $fib(N)=fib(N-1)+fib(N-2)$，因此由归纳法容易证明 $T(N) \geqslant fib(N)$。在 1.2.5 节我们证明过 $fib(N)<(5/3)^N$，类似的计算可以证明(对于 $N>4$) $fib(N) \geqslant (3/2)^N$，从而这个程序的运行时间以指数的速度增长。这大致是最坏的情况。通过保留一个简单的数组并使用一个 for 循环，运行时间可以显著降低。

这个程序之所以运行缓慢，是因为存在大量多余的工作要做，违反了在 1.3 节中叙述的递归的第四条主要法则(合成效益法则)。注意，在第 3 行上的第一次调用即 fib(n－1)实际上在某处计算 fib(n－2)。这个信息被抛弃而在第 3 行上的第二次调用时又重新计算了一遍。抛弃的信息量递归地合成起来并导致巨大的运行时间。这或许是格言“计算任何事情不要超过一次”的最好的实例，但它不应使你被吓得远离递归而不敢使用。本书中将随处看到递归的杰出使用。

2.4.3 最大子序列和问题的求解

现在我们将要叙述四个算法来求解早先提出的最大子序列和问题。第一个算法如图 2-5 所示，它只是穷举式地尝试所有的可能。for 循环中的循环变量反映了 Java 中数组从 0 开始而不是从 1 开始这样一个事实。还有，本算法并不计算实际的子序列；实际的计算还要添加一些额外的代码。

该算法肯定会正确运行(这用不着花太多的时间去证明)。运行时间为 $O(N^3)$，这完全取决于第 13 行和第 14 行，它们由一个含于三重嵌套 for 循环中的 $O(1)$语句组成。第 8 行上的循环