里，我们又可写出一个递推公式并将其解出。简单的直觉避免了盲目的强行处理。

图 2-11 中的代码实现了这个想法[⊖]。有时候看一看程序能够进行多大的调整而不影响其正确性倒是很有意思的。在图 2-11 中，第 5 行到第 6 行实际上不是必需的，因为如果 N 是 1，那么第 10 行将做同样的事情。第 10 行还可以写成：

```
10 return pow( x, n - 1 ) * x;
```

而不影响程序的正确性。事实上，程序仍将以 $O(\log n)$运行，因为乘法的序列同以前一样。不过，下面所有对第 8 行的修改都是不可取的，虽然它们看起来似乎都正确：

```
8a          return pow( pow( x, 2 ), n / 2 );
8b          return pow( pow( x, n / 2 ), 2 );
8c          return pow( x, n / 2 ) * pow( x, n / 2 );
```

```
1       public static long pow( long x, int n )
2       {
3           if( n == 0 )
4               return 1;
5           if( n == 1 )
6               return x;
7           if( isEven( n ) )
8               return pow( x * x, n / 2 );
9           else
10              return pow( x * x, n / 2 ) * x;
11      }
```

图 2-11 高效率的幂运算

8a 和 8b 两行都是不正确的，因为当 N 是 2 时递归调用 pow 中有一个是以 2 作为第 2 个参数。这样，程序产生一个无限循环，将不能往下进行(最终导致程序非正常终止)。

使用 8c 行会影响程序的效率，因为此时有两个大小为 $N/2$ 的递归调用而不是一个。分析指出，其运行时间不再是 $O(\log N)$。我们把它作为练习留给读者去确定这个新的运行时间。

2.4.5 检验你的分析

一旦分析进行过后，则需要看一看答案是否正确，是否尽可能地好。一种实现方法是编程并比较实际观察到的运行时间与通过分析所描述的运行时间是否一致。当 N 扩大一倍，则线性程序的运行时间乘以因子 2，二次程序的运行时间乘以因子 4，而三次程序则乘以因子 8。以对数时间运行的程序，当 N 增加一倍时其运行时间只是多加一个常数，而以 $O(N \log N)$运行的程序则花费比在相同环境下运行时间的两倍稍多一些的时间。如果低阶项的系数相对较大，并且 N 又不是足够地大，那么运行时间的这种增加量很难观察清楚。例如，对于最大子序列和问题，当从 $N = 10$ 增加到 $N = 100$ 时其各种实现方法运行时间的变化就是一个例子。单纯凭实践区分线性程序还是 $O(N \log N)$程序是非常困难的。

验证一个程序是否是 $O(f(N))$的另一个常用的技巧是对 N 的某个范围(通常用 2 的倍数隔开)计算比值 $T(N)/f(N)$，其中 $T(N)$是凭经验观察到的运行时间。如果 $f(N)$是运行时间的理想近似，那么所算出的值收敛于一个正常数。如果 $f(N)$估计过大，则算出的值收敛于 0。如果 $f(N)$估计过低从而 $O(f(N))$是错的，那么计算出的值发散。

⊖ Java 提供一个 BigInteger 类，这个类可以用来处理任意大的整数。很容易把图 2-11 改写成使用 BigInteger 而不用 long。