递归是一种强有力的技巧，但和其他技巧一样，它也可能被误用。

一般需要递归解决的问题有两个特点：

* 存在限制条件，当符合这个条件时递归便不再继续；
* 每次递归调用之后越来越接近这个限制条件。

递归使用最常见的一个例子就是求阶乘，具体描述和代码请看这里：C语言递归和迭代法求阶乘

但是，递归函数调用将涉及一些运行时开销——参数必须压到堆栈中，为局部变量分配内存空间（所有递归均如此，并非特指求阶乘这个例子），寄存器的值必须保存等。当递归函数的每次调用返回时，上述这些操作必须还原，恢复成原来的样子。所以， 基于这些开销，对于递归求阶乘而言，它并没有简化问题的解决方案。

迭代求阶乘使用简单循环的程序，看上去不甚符合前面阶乘的数学定义，但它却能更为有效地计算出结果。如果你仔细观察递归函数，你会发现递归调用是函数所执行的最后一项任务。这个函数是尾部递归(tail recursion)的一个例子。由于函数在递归调用返回之后不再执行任何任务，所以尾部递归可以很方便地转换成一个简单循环，完成相同的任务。

提示：许多问题是以递归的形式进行解释的，这只是因为它比非递归形式更为清晰。但是，这些问题的迭代实现往往比递归实现效率更高，虽然代码的可读性可能稍差一些，当一个问题相当复杂，难以用迭代形式实现时，此时递归实现的简洁性便可以补偿它所带来的运行时开销。

这里有一个更为极端的例子，菲波那契数就是一个数列，数列中每个数的值就是它前面两个数的和。 这种关系常常用递归的形式进行描述：



同样，这种递归形式的定义容易诱导人们使用递归形式来解决问题。这里有一个陷牌：它使用递归步骤计算Fibonacci(n-1)和Fibonacci(n-2)。但是，在计算Fibonacci(n-1)时也将计算Fibonacci(n-2)。这个额外的计算代价有多大呢？

答案是，它的代价远远不止一个冗余计算：每个递归调用都触发另外两个递归调用，而这两个调用的任何一个还将触发两个递归调用，再接下去的调用也是如此。这样，冗余计算的数量增长得非常快。例如，在递归计算Fibonacci(10)时，Fibonacci(3)的值被计算了21次。但是，在递归计算 Fibonacci(30)时，Fibonacci(3)的值被计算了317811次。当然，这317811次计算所产生的结果是完全一样的，除了其中之一外，其余的纯属浪费。这个额外的开销真是相当恐怖！

如果使用一个简单循环来代替递归，这个循环的形式肯定不如递归形式符合前面菲波那契数的抽象定义，但它的效率提高了几十万倍！

当你使用递归方式实现一个函数之前，先问问你自己使用递归带来的好处是否抵得上它的代价。 而且你必须小心：这个代价可能比初看上去要大得多。

不信请看下面的代码，分别用递归和迭代计算斐波那契数，效率差距真是大的惊人。

1. #include <stdio.h>
2. #include <time.h>
3. #include <windows.h>
4. // 递归计算斐波那契数
5. long fibonacci\_recursion( int n )
6. {
7. **if**( n <= 2 )
8. **return** 1;
9. **return** fibonacci\_recursion(n-1) + fibonacci\_recursion(n-2);
10. }
11. // 迭代计算斐波那契数
12. long fibonacci\_iteration( int n )
13. {
14. long result;
15. long previous\_result;
16. long next\_older\_result;
17. result = previous\_result = 1;
18. **while**( n > 2 ){
19. n -= 1;
20. next\_older\_result = previous\_result;
21. previous\_result = result;
22. result = previous\_result + next\_older\_result;
23. }
24. **return** result;
25. }
26. int main(){
27. int N = 45;
28. // 递归消耗的时间
29. clock\_t recursion\_start\_time = clock();
30. long result\_recursion = fibonacci\_recursion(N);
31. clock\_t recursion\_end\_time = clock();
32. // 迭代消耗的时间
33. clock\_t iteration\_start\_time = clock();
34. long result\_iteration = fibonacci\_iteration(N);
35. clock\_t iteration\_end\_time = clock();
36. // 输出递归消耗的时间
37. printf("Result of recursion: %ld \nTime: %fseconds",
38. fibonacci\_recursion(N),
39. (double)(recursion\_end\_time-recursion\_start\_time)/CLOCKS\_PER\_SEC
40. );
41. printf("\n-----------------------\n");
42. // 输出迭代消耗的时间
43. printf("Result of iteration: %ld \nTime: %fseconds",
44. fibonacci\_iteration(N),
45. (double)(iteration\_end\_time-iteration\_start\_time)/CLOCKS\_PER\_SEC
46. );
47. **return** 0;
48. }

运行结果：

Result of recursion: 1134903170

Time: 7.494000 seconds

---------------------------------------

Result of iteration: 1134903170

Time: 0.000000 seconds