上面的讨论导致递归的前两个基本法则：

1．基准情形(base case)。必须总要有某些基准的情形，它们不用递归就能求解。

2．不断推进(making progress)。对于那些要递归求解的情形，递归调用必须总能够朝着一个基准情形推进。

在本书中我们将用递归解决一些问题。作为非数学应用的一个例子，考虑一本大词典。词典中的词都是用其他的词定义的。当查一个单词的时候，我们不是总能理解对该词的解释，于是我们不得不再查找解释中的一些词。同样，对这些词中的某些地方我们又不理解，因此还要继续这种查找。因为词典是有限的，所以实际上或者我们最终要查到一处，明白了此处解释中所有的单词(从而理解这里的解释，并按照查找的路径回查其余的解释)或者我们发现这些解释形成一个循环，无法理解其最终含义，或者在解释中需要我们理解的某个单词不在这本词典里。

我们理解这些单词的递归策略如下：如果知道一个单词的含义，那么就算我们成功；否则，就在词典里查找这个单词。如果我们理解对该词解释中的所有的单词，那么又算我们成功；否则，通过递归查找一些我们不认识的单词来“算出”对该单词解释的含义。如果词典编纂得完美无暇，那么这个过程就能够终止；如果其中一个单词没有查到或是循环定义(解释)，那么这个过程则循环不定。

打印输出整数

设有一个正整数 n 并希望把它打印出来。我们的例程的名字为 printOut(n)。假设仅有的现成 I/O 例程将只处理单个数字并将其输出到终端。我们为这种例程命名为 printDigit；例如，printDigit(4)将输出 4 到终端。

递归将为该问题提供一个非常漂亮的解。要打印 76234，我们首先需要打印出 7623，然后再打印出 4。第二步用语句 printDigit(n % 10)很容易完成，但是第一步却不比原问题简单多少。它实际上是同一个问题，因此可以用语句 printOut(n/10)递归地解决它。

这告诉我们如何去解决一般的问题，不过我们仍然需要确认程序不是循环不定的。由于我们尚未定义一个基准情况，因此很清楚，我们仍然还有些事情要做。如果 $0 \leqslant n < 10$，那么基准情形就是 printDigit(n)。现在，printOut(n)已对每一个从 0 到 9 的正整数定义，而更大的正整数则用较小的正整数定义。因此，不存在循环的问题。整个方法在图 1-4 中指出。

```
1       public static void printOut( int n )  /* Print nonnegative n */
2       {
3           if( n >= 10 )
4               printOut( n / 10 );
5           printDigit( n % 10 );
6       }
```

图 1-4　打印整数的递归例程

我们没有努力去高效地做这件事。我们本可以避免使用 mod 例程(它是非常耗时的)，因为 $n\%10 = n - \lfloor n/10 \rfloor * 10$。[⊖]

递归和归纳

让我们多少严格一些地证明上述递归的整数打印程序是可行的。为此，我们将使用归纳法证明。

⊖ $\lfloor x \rfloor$是小于或等于 x 的最大整数。