第3章

算法基本工具和优化技巧

算法设计的基础工作是把人脑思维出的解决问题的方法、步骤，规范化地描述成“机械化的操作”。这就好像自动化生产，其实质就是把人类生产中，较规范的大量重复工作交给了机器去完成。所以有人称计算机带来了“机械”思维的时代。

从已学过的计算机知识中可以了解到，计算机或者说程序设计语言为算法提供的“机械化的操作”其实是很少的。主要有计算、输入和输出操作，流程控制操作（即选择、循环和递归），以及提供了这些操作对象的不同存储模式：变量、数组、结构体（记录）和文件。而软件功能之丰富，使用之方便，却是越来越让人感叹，这都是人类设计出来的算法的功劳。相信读者能够体会到学习算法设计的必要性。

本章就是要讲解怎么样充分利用这些基本的“机械化的操作”设计高质量的算法，在程序设计与算法设计之间起承上启下的作用。

3.1 循环与递归

事实上，在一般情况下只有处理大量的数据才借助于计算机，所以算法设计中很重要的工作就是把对数据的处理归结成较规范的可重复的“机械化的操作”交给计算机去完成。即将重复处理大量数据的步骤抽象成“循环”或“递归”的模式，设计出可以针对不同规模解决问题的算法。

不同于机器生产产品的“机械化的重复操作”，计算机进行数据处理不可能是完全相同操作的重复。所以必须要设计出表现形式不变，但能实现动态处理数据的“机械化的重复操作”。也就是说，在重复操作中，“循环条件”、“循环体”都必须是“不变式”，而数据处理对象却是变化的，算法是在渐进地完成处理数据的操作。

循环模式算法设计中，一个重要的工作就是从已建立好的数学模型中，构造出“不变式”的“循环条件”、“循环体”。“不变式”主要是依靠变量或数组元素表示的，因为变量名或数组元素是“不变”的，而变量或数组元素中的数据是不断变化的，从而数据处理是动态的、渐进的。

本节通过实例介绍循环不变式和递归不变式的构造过程，简单地说明循环、递归设计的基本方法及应该考虑的因素。

【提示】 请读者回忆高级语言程序设计和数据结构课程中，设计“循环条件不变式”、“循环体不变式”的方法和技巧。

3.1.1 循环设计要点

1. 设计中要注意算法的效率

累加、累乘是学习程序设计语言中接触最多的程序，它就是通过数学模型 $S_n=S_{n-1}+A_n$，$T_n=T_{n-1}\times A_n$，构造出“循环不变”的累加式 $S=S+A$ 和累乘式 $T=T\times A$。下面看一个累加、累乘算法的设计过程。

【例 1】 求 $1/1!-1/3!+1/5!-1/7!+\cdots+(-1)^{n+1}/(2n-1)!$。

问题分析：此问题中既有累加又有累乘，准确地说累加的对象是累乘的结果。

数学模型 1：$S_n=S_{n-1}+(-1)^{n+1}/(2n-1)!$

算法设计 1：多数初学者会直接利用题目中累加项通式，构造出循环体不变式为 $S=S+(-1)^{n+1}/(2n-1)!$ 需要用二重循环来完成算法。

算法 1 如下：

```
main( )
{int i,n,j,sign=1;
float s,t=1;
  input(n);
  s=1;
for(i=2; i<=n; i=i+1)
  {t=1;                              // 求阶乘
  for(j=1; j<=2*i-1; j=j+1)
      t=t*j;
  sign=1;                            // 求(-1)^(n+1)
  for(j=1; j<=i+1; j=j+1)
      sign=sign *(-1);
  s=s+sign/t;
  }
 print("Sum=",s);
}
```

算法分析 1：以上算法是完全正确的，但算法的效率却太低是 $O(n^2)$。其原因是，当前一次循环已求出 7!，当这次要想求 9! 时，没必要再从 1 去累乘到 9，只须充分利用前一次的结果，用 $7!\times 8\times 9$ 即可得到 9!，模型为 $A_n=A_{n-1}\times 1/((2\times n-2)\times(2\times n-1))$。另外运算 sign=sign *(-1)；总共也要进行 $n\times(n-1)/2$ 次乘法，这也是没有必要的。下面就进行改进。

数学模型 2：$S_n=S_{n-1}+(-1)^{n+1}A_n$；$A_n=A_{n-1}\times 1/((2\times n-2)\times(2\times n-1))$。

算法设计 2：利用以上数学模型容易构造累加、累乘不变式，对 $(-1)^{n+1}$ 可以用一个变量 sign 记录其值，每循环一次执行“sign=-sign；”就可以模拟符号的变化过程。这样，只须一重循环就能解决问题。

算法 2 如下：

```
main( )
```

```
{int i,n,sign;
 float s,t=1;
 input(n);
  s=1;
  sign=1;
 for(i=2; i<=n; i=i+1)                    或      for(i=1; i<=n-1; i=i+1)
   {sign=-sign;                                    { sign=-sign;
   t=t*(2*i-2)*(2*i-1)};                           t=t*2*i*(2*i+1)};
   s=s+sign/t; }                                   s=s+sign/t;
 print("Sum=",s);
}
```

算法说明 2：构造循环不变式时，一定要注意循环变量的意义，如当 i 不是项数序号时（右边的循环中）有关 t 的累乘式与 i 是项数序号时就不能相同。

算法分析 2：这个算法的时间复杂性为 $O(n)$。

由此例，构造循环不变式时，一定要考虑算法的效率。

2.“自顶向下”的设计方法

对于简单的算法，可以像例 1 一样直接进行算法设计，对于比较难一些的算法，则可以用“自顶向下”的设计方法，特别是有嵌套循环的情况。

自顶向下的方法是从全局走向局部、从概略走向详尽的设计方法。自上而下是系统分解和细化的过程。

【例 2】 一个数如果恰好等于它的因子之和（包括 1，但不包括这个数本身），这个数就称为“完数”。

例如，28 的因子为 1，2，4，7，14，而 28＝1＋2＋4＋7＋14。因此 28 是“完数”。编写算法找出 1000 之内的所有完数，并按下面格式输出其因子：28 it's factors are 1，2，4，7，14。

问题分析：这个问题中不是要质因数，所以找到因数后就无须将其从数据中“除掉”。每个因数只记一次，如 8 的因数为 1，2，4 而不是 1，2，2，2，4（注：本题限定因数不包括这个数本身）。

算法设计：“自顶向下”的算法设计方法，就是先概要地设计算法的第一层（即顶层），然后步步深入，逐层细分，逐步求精，直到整个问题可用程序设计语言明确地描述出来为止。

自顶向下设计的步骤：首先对问题进行仔细分析，写出程序运行的主要过程和任务；然后从大的功能方面把一个问题的解决过程分为几个问题，每个子问题形成一个模块。这样，可以使设计过程中的每一时刻都只须考虑很少的问题。

本题的设计过程如下。

1）顶层算法

```
for(i=2; i<=n; i=i+1)
  {判断 i 是否"完数";
   是"完数"则按格式输出; }
```

2）判断 i 是否“完数”的算法

```
for(j=2; j<i; j=j+1)
    找 i 的因子,并累加;
如果累加值等于 i,i 是"完数"则输出;
```

3）进一步细化——判断 i 是否“完数”算法

```
s=1
for(j=2; j<i; j=j+1)
    if (i mod j=0) (j 是 i 的因素)     s=s+j;
if  (s=i)     i 是"完数";
```

4）考虑输出格式——判断 i 是否“完数”算法

考虑到要按格式输出结果，应该开辟数组存储数据 i 的所有因子，并记录其因子的个数，因此算法细化如下：

```
定义数组 a,变量 s=1,k=0;
for(j=2; j<i; j=j+1)
    if (i mod j=0) (j 是 i 的因素)
        {s=s+j; a[k]=j; k=k+1; }
if     (s=i)
    {按格式输出结果}
```

综合以上逐层设计结果，得到以下算法：

```
main( )
{int i,k,j,s,a[20];
for(i=1; i<=1000; i=i+1)
  {s=1;
   k=0;
  for(j=2; j<i; j=j+1)
   if (i mod j)=0)
     {s=s+j;
      a[k]=j;
      k=k+1; }
  if(i=s)
      {print(s,"it's  factors  are: ",1);
       for(j=0; i<k; j=j+1)
           print(",",a[k]);
      }
  }
}
```

由例题可以看出，自顶向下设计的特点：先整体后局部，先抽象后具体。

下面再看一个例子。

【例 3】 求一个矩阵的鞍点，即在行上最小而在列上最大的点。

算法设计：针对 $n\times n$ 矩阵进行设计，操作逐行进行，行列下标起始为 0。“自顶向下”的

设计如下。

1）顶层算法

```
for(i=0; i<n; i=i+1)
  {找第 i 行上最小的元素 t 及所在列 minj;
   检验 t 是否是第 minj 列的最大值,是,则输出这个鞍点; }
```

2）找第 i 行上最小的元素 t 及所在列 minj

```
t=a[i][0]; minj=0;
for(j=1; j<n; j=j+1)
  if(a[i][j]<t)
      {t=a[i][j];
       minj=j; }
```

3）检验 t 是否为第 minj 列的最大值,是,则输出这个鞍点

```
for(k=0; k<n; k=k+1)
   if(a[k][minj]>t) break;
if(k<n)      continue;
print("the result is a[",i,"][",minj,"]=",t);
```

综合以上设计结果,得到以下算法：

```
  readmtr(int a[][10],int n)
      {int   i,j;
        print("input n * n matric: ");
        for(i=0; i<n; i=i+1)
         for(j=0; j<n; j=j+1)
           input(a[i][j]);
     }
    printmtr(int a[][10],int n)
     {int i,j
      for(i=0; i<n; i=i+1)
       {for(j=0; j<n; j=j+1)
            print(a[i][j]);
        print("换行符")
        }
      }
main( )
        {int a[10][10];
        int i,j,k,minj,t,n=10,kz=0;
        readmtr(a,n);
        printmtr(a,n);
        for(i=0; i<n; i=i+1)
          {t=a[i][0];
           minj=0;
           for(j=1; j<n; j=j+1)
              if(a[i][j]<t)
```

```
            {t=a[i][j];
            minj=j; }
        for(k=0; k<n; k=k+1)
            if(a[k][minj]>t)  break;
        if(k<n) continue;
        print("the result is a[",i,"][",minj,"]=",t);
        kz=1;
        break;
        }
    if(kz=0) print("Non solution! ");
    }
```

算法说明：

(1) 算法中 minj 代表当前行中最小值的列下标，循环变量 i,j 分别代表行、列下标。循环变量 k 也代表行下标，在循环 for(k=1; k<=n; k=k+1)中只针对 minj 列进行比较。

(2) 考虑到会有无解的情况，设置标志量 kz，kz=0 代表无解，找到一个解后，kz 被赋值为 1，就不再继续找鞍点的工作。请读者考虑是否有多解的可能性吗？若有，请改写算法，找出矩阵中所有的鞍点。

3. 由具体到抽象设计循环结构

对于不太熟悉的问题，其数学模型或“机械化操作步骤”不易抽象，下面看一个由具体到抽象设计循环细节的例题。

【例 4】 编写算法：打印具有下面规律的图形。

```
 1
 5  2
 8  6  3
10  9  7  4
```

问题分析：无论从题意理解，还是从算法的通用性来考虑，算法设计不能只针对图中的 4×4 二维数组进行。下面以任意阶的二维数组 $n\times n$ 讨论。为分析方便，数组的起始下标定为 1。

算法设计：对二维表的操作一般是按行或列进行的，但此图形中数据的排列方法却不是按行、列进行的。二维表是否只能按行、列顺序进行操作呢？回答是否定的，下面根据数据排列的特点，按“层”对数据进行处理。

容易发现图形中自然数在矩阵中排列的规律，题目中 1,2,3,4 所在位置称为第 1 层(主对角线)，例图中 5,6,7 所在位置称为第二层，……。一般地，第一层有 n 个元素，第二层有 $n-1$ 个元素……

基于以上数据变化规律，以层号作为外层循环，循环变量为 i(范围为 $1\sim n$)；以层内元素从左上到右下的序号作为内循环，循环变量为 j(范围为 $1\sim n+1-i$)。这样循环的执行过程正好与“摆放”自然数的顺序相同。用一个变量 k 模拟要“摆放”的数据，下面的问题就是怎么样将数据存储到对应的数组元素。

数组元素的存取，只能是按行、列号操作的。所以下面用由具体到抽象设计循环的“归

纳法”，找出数组元素的行号、列号与层号 i 及层内序号 j 的关系。

(1) 每层内元素的列号都与其所在层内的序号 j 是相同的。因为每层的序号是从第一列开始向右下进行。

(2) 元素的行与其所在的层号及在层内的序号均有关系，具体如下：

第一层行号 1～n，行号与 j 同；

第二层行号 2～n，行号比 j 大 1；

第三层行号 3～n，行号比 j 大 2；

⋮

行号起点随层号 i 增加而增加，层内其他各行的行号又随层内序号 j 增加而增加，由于编号起始为 1，i 层第 j 个数据的列下标为 $i-1+j$。

综合以上分析，i 层第 j 个数据对应的数组元素是 $a[i-1+j][j]$。

算法如下：

```
main( )
{int i,j,a[100][100],n,k;
 input(n);
 k=1;
for(i=1; i<=n; i=i+1)
  for(j=1; j<=n+1-i; j=j+1)
   {a[i-1+j][j]=k;
    k=k+1; }
for(i=1; i<=n; i=i+1)
  {print("换行符");
   for(j=1; j<=i; j=j+1)
      print(a[i][j]);
  }
}
```

算法说明：仅就层内元素个数而言，内层循环变量 j 的变化过程也可以为 i～n，但这样不利于与列下标对应。

由此例注意，要以问题的特点为依据进行算法设计，循环变量的意义不是固定不变的。其范围及引用等细节，由具体的实例进行归纳，可以比较容易地得到抽象的表达式。

【提示】 由此例可以看出为什么数学上会有很多“猜想”，要证明一个猜想很难，但通过实例总结归纳规律的能力大家都是很强的，这是算法设计基本方法之一。

3.1.2　递归设计要点

上一节中，为了处理重复性的操作，采用的办法是构造循环。本小节则介绍另一种方法，采用递归的办法来实现重复性的操作。在程序设计语言中，已经学习了递归的概念，递归(recursion)是一个过程或函数在其定义或说明中又直接或间接调用自身的一种方法。

递归算法设计，就是把一个大型复杂的问题层层转化为一个与原问题相似的规模较小的问题，在逐步求解小问题后，再返回(回溯)得到大问题的解。递归算法只须少量的步骤就

可描述出解题过程所需要的多次重复计算,大大地减少了算法的代码量。

递归算法设计的关键在于找出递归关系(方程)和递归终止(边界)条件。递归关系就是使问题向边界条件转化的规则。递归关系必须能使问题越来越简单,规模越来越小。递归边界条件就是所描述问题最简单的、可解的情况,它本身不再使用递归的定义。

因此,用递归算法解题,通常有 3 个步骤。

(1) 分析问题、寻找递归关系:找出大规模问题与小规模问题的关系,这样通过递归使问题的规模逐渐变小。

(2) 设置边界、控制递归:找出停止条件,即算法可解的最小规模问题。

(3) 设计函数、确定参数:和其他算法模块一样设计函数体中的操作及相关参数。

下面是一个经典的递归例题。

【例 5】 汉诺塔问题:古代有一个梵塔,塔内有 3 个基座 A,B,C,开始时 A 基座上有 64 个盘子,盘子大小不等,大的在下,小的在上。有一个老和尚想把这 64 个盘子从 A 座移到 B 座,但每次只允许移动一个盘子,且在移动过程中,3 个基座上的盘子都始终保持大盘在下,小盘在上。移动过程中可以利用 C 基座做辅助。请编程打印出移动过程。

问题分析:此问题又称为"世界末日问题",因为以最高效的移动(无不必要的移动)方法,以每秒移动一次的速度 64 个盘子也需要近 5800 亿年的时间。当然不必真的去解 64 阶汉诺塔问题,一般地对任意 n 阶的汉诺塔问题进行讨论。

算法设计:用人类的大脑直接去解 3,4 或 5 阶的汉诺塔问题(当然是以最高效的移动方法)还可以,但更高阶的问题就难以完成了,更不用说是把问题的解法抽象成循环的机械操作了。所以此问题多用递归算法来解,即使有非递归算法,也是模仿递归算法的执行过程而得到的。下面用递归法解此题,约定盘子自上而下盘子的编号为 1,2,3,…,n。

首先,看一下 2 阶汉诺塔问题的解,不难理解以下移动过程(括号中是基座现有盘子的号):

初始状态为A(1,2)　　B( )　　C( )

第一步后　A(2)　　B( )　　C(1)

第二步后　A( )　　B(2)　　C(1)

第三步后　A( )　　B(1,2)　　C( )

如何找出大规模问题与小规模问题的关系,从而设计出递归算法呢?在已经会做两个盘子的汉诺塔问题后,这个关系就不难找到了。把 n 个盘子抽象地看作"两个盘子",上面"一个"由 1～n－1 号组成,下面一个就是 n 号盘子。移动过程如下。

第一步:先把上面"一个"盘子以 A 基座为起点借助 B 基座移到 C 基座。

第二步:把下面一个盘子从 A 基座移到 B 基座。

第三步:再把 C 基座上的"一个"盘子借助 A 基座移到 B 基座。

把 n 阶汉诺塔问题记作 hanoi(n,a,b,c),注意这里 a,b,c 并不总代表 A,B,C 3 个基座,其意义为:第一个参数 a 代表每一次移动的起始基座,第二个参数 b 代表每一次移动的终点基座,第三个参数 c 代表每一次移动的辅助基座。由上述约定,n 阶的汉诺塔问题记作 hanoi(n,a,b,c),a,b,c 初值为"A","B","C",以后的操作等价于以下 3 步。

第一步:hanoi(n－1,a,c,b);

第二步：把下面“一个”盘子从 A 基座移到 B 基座；

第三步：hanoi($n-1$,c,b,a)。

至此找出了大规模问题与小规模问题的递归关系。操作过程如下：

(1) 将 A 杆上面的 $n-1$ 个盘子，借助 B 杆，移到 C 杆上，如图 3-1(a)所示；

(2) 将 A 杆上剩余的一个 n 号盘子移到 B 杆上，如图 3-1(b)所示；

(3) 将 C 杆上的 $n-1$ 个盘子，借助 A 杆，移到 B 杆上，如图 3-1(c)所示。

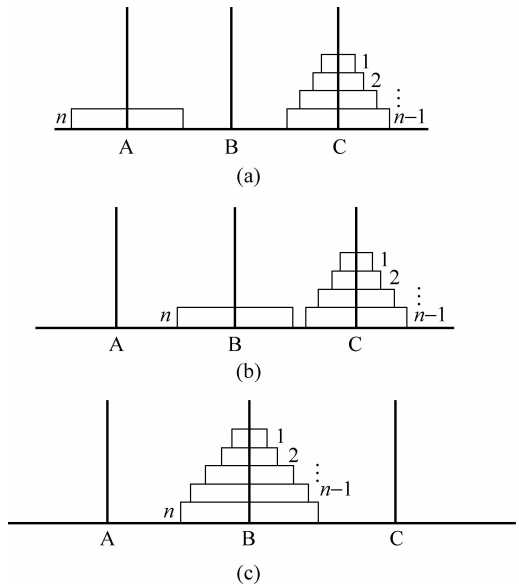

图 3-1　汉诺塔问题求解算法图示

有读者可能想到把 2 阶或 1 阶的汉诺塔问题，当作停止条件，即问题可解的最小规模。其实没有必要，只要用 0 阶的汉诺塔问题当作停止条件即可，这时什么都不需要做。

算法如下：

```
main( )
{int n;
 input(n);
 hanoi (n,"A","B","C");
}
   hanoi (int n,char a,char b,char c)
   { if(n>0)
       {hanoi(n-1,a,c,b);
       输出" Move dish",n."from pile",a," to"b);
       hanoi(n-1,c,b,a); }
   }
```

递归算法执行中有递归调用的过程和回溯的过程(当然二者是不可分的)，递归法就是通过递归调用把问题从大规模归结到小规模，当最小规模得到解决后又把小规模的结果回溯，从而推出大规模的解。

【提示】 如果再思考一下用循环实现此问题,应该发现递归设计要比循环设计简单得多。反过来请尝试将前面一些用循环机制实现的问题用递归机制实现。

下面再看一个比较复杂的递归算法设计。

【例 6】 整数的分划问题。

对于一个正整数 n 的分划,就是把 n 表示成一系列正整数之和的表达式。注意,分划与顺序无关,例如 6=5+1 和 6=1+5 被认为是同一种分划。另外,这个整数 n 本身也算是一种分划。

例如,对于正整数 $n=6$,它可以分划为:

6
5+1
4+2　　4+1+1
3+3　　3+2+1　　3+1+1+1
2+2+2　　2+2+1+1　　2+1+1+1+1
1+1+1+1+1+1

现在的问题是,对于给定的正整数 n,要求编算法计算出其分划的数目 $P(n)$。

模型建立:这里的目标是要建立递归分划数目的递归公式。

从上面 $n=6$ 的实际例子可以看出,很难找到大规模问题 $P(n)$ 与小规模问题 $P(n-d)$ $(d=1,2,3,\cdots)$ 的关系。根据 $n=6$ 的实例发现"第一行及以后的数据不超过 6,第二行及以后的数据不超过 5,……,第六行的数据不超过 1"。因此,定义一个函数 $Q(n,m)$,表示整数 n 的"任何加数都不超过 m"的分划的数目,n 的所有分划数目 $P(n)$ 就应该表示为 $Q(n,n)$。

一般地 $Q(n,m)$ 有以下递归关系:

(1) $Q(n,n)=1+Q(n,n-1)$

等式右边的"1"表示 n 只包含一个被加数等于 n 本身的分划;则其余的分划表示 n 的所有其他分划,即最大加数 $m<=n-1$ 的划分。

(2) $Q(n,m)=Q(n,m-1)+Q(n-m,m)$　　$(m<n)$

等式右边的第一部分表示被加数中不包含 m 的分划的数目;第二部分表示被加数中包含(注意不是小于)m 的分划的数目,因为如果确定了一个分划的被加数中包含 m,则剩下的部分就是对 $n-m$ 进行不超过 m 的划分。

到此找到了大规模问题与小规模问题的递归关系,下面是递归的停止条件:

(1) $Q(n,1)=1$,表示当最大的被加数是 1 时,该整数 n 只有一种分划,即 n 个 1 相加;

(2) $Q(1,m)=1$,表示整数 $n=1$ 只有一个分划,不管最大被加数的上限 m 是多大。

算法设计:由以上模型不难写出算法。考虑算法的稳健性,如果 $n<m$,则 $Q(n,m)$ 是无意义的,因为 n 的分划不可能包含大于 n 的被加数 m,此时令 $Q(n,m)=Q(n,n)$;同样当 $n<1$ 或 $m<1$ 时,$Q(n,m)$ 也是无意义的。算法如下:

```
main( )
{int n;
 input(n);
 Divinteger(n,n);
}
```

```
Divinteger(int n,int m)
{if(n<1 or m<1)
    Error("输入参数错误");
else if (n=1 or m=1)
      return (1);
     else if(n<m)
            return Divinteger(n,n)
         else if(n=m)
              return (1+Divinteger(n,n-1))
            else
              return (Divinteger(n,m-1)+Divinteger(n-m,m));
}
```

算法说明：由于算法中，多次进行递归调用，正整数划分的数目随着 n 的增加增长的非常快，大约是以指数级增长，所以此算法不适合对较大的整数进行划分。感兴趣的读者可以在学习完回溯算法后，完成解决此问题的高效算法。

由以上例子可以看出，虽然递归算法与循环设计的思想不同，但由具体实例从“具体到抽象”归纳算法设计的方法是一样的。

【提示】 与求阶层的递归程序比较，这个例题的递归解法是否可以称为“二维递归”？

3.1.3　递归与循环的比较

由上一节递归算法设计的例子，不难理解递归也是一种实现“重复操作”的机制。它把“较复杂”操作依次地归结为“较简单”操作，一直归结到“最简单”操作，能方便完成操作为止。在实际运用中，有很多问题的数学模型本来就是递归的，用递归来描述它们不仅非常自然而且证明算法的正确性也相应地比非递归形式容易得多。可以证明：每个迭代算法原则上总可以转换成与它等价的递归算法；反之不然，即并不是每个递归算法都可以转换成与它等价的循环结构算法，例如上一小节的例 5。

下面通过几个具体的例子来说明循环和递归的差异和优劣。

【例 7】 任给十进制的正整数，请从低位到高位逐位输出各位数字。

循环算法设计：从题目中并不能获知正整数的位数，再看题目的要求，算法应该从低位到高位逐位求出各位数字并输出，详细设计如下。

(1) 求个位数字的算式为 n mod 10。

(2) 为了保证循环体为“不变式”，求十位数字的算式仍旧为 n mod 10，这就要通过算式 $n=n\backslash 10$，将 n 的十位数变成个位数。

循环算法如下：

```
main( )
{int n;
 input(n);
  while(n>=10)
   { print(n mod 10);
     n=n\10; }
```