第5章 问题求解和算法设计

CHAPTER 5

本章首先讨论解决一般问题的思路以及计算机解决问题的方法，然后讨论算法。算法对计算机专业的学生来说非常重要，计算机科学有时被定义为“对算法及其在计算机上如何有效实现的研究”。本章讲述算法在解决问题中的作用、开发算法的策略、跟踪和测试算法的技术。本章还将介绍伪代码，这是一种为表示算法而设计的人工语言。最后讨论经典的搜索算法和排序算法。

本章学习目标如下。

- 描述波利亚提出的解决问题的步骤。
- 结合波利亚提出的如何解决问题的列表，描述计算机问题求解的步骤。
- 理解用于表示算法的伪代码。
- 使用伪代码表示算法。
- 开发算法来解决问题。
- 理解与问题求解相关的几点思想——信息隐蔽、抽象、事物命名和测试。

5.1 问题求解

所谓“问题求解”，是指找到解决难题的方案。现实世界中的大量问题不适合计算机处理。例如，计算机不能直接帮助农民收粮食，也不能帮忙把超额的人塞进电梯，更不能帮助平息战事以及宗教和领土冲突。涉及物理行为和情感的难题，计算机都不能解决。

此外，如果不告诉计算机要做什么，它什么也做不了。计算机是没有智能的，它不能分析问题并产生问题的解决方案。人（程序员）必须分析这些问题，为解决问题开发程序，然后让计算机执行这些程序。

计算机的作用是只要你为它编写好了解决方案，它能够快速、一致地反复执行这个方案，将人们从枯燥重复的任务中解放出来。

5.1.1 如何解决问题

1945年，著名数学家和教育学家波利亚（George Polya）写了一本书，名为《怎样解题》（*How to Solve It*）。尽管这本书写于几十年前，当时计算机还处于试验阶段，但是其中关于问题求解过程的描述非常经典，适用于各种类型的问题。

1. 解决问题的步骤及相应的策略

1）步骤 1：必须理解问题

策略：刨根问底。

当遇到一个问题或任务后，应该多提问，直到完全明白该问题或任务要做什么。下面是一些典型的提问。

- 我对这个问题了解多少？
- 要找到解决方案，我必须处理哪些信息？
- 解决方案是什么样的？
- 存在什么特例？
- 我如何知道已经找到了解决方案？

2）步骤 2：设计方案

这一步要找到已知信息和解决方案之间的联系，如果找不到直接的联系，则可能需要考虑辅助问题，目的是最终得到解决方案。

（1）策略 1：寻找与问题相关的熟悉的情况

以前见过这个问题吗？或者见过形式稍有不同的同类问题吗？

如果以前曾经解决过相同或相似的问题，只需要再次使用那种成功的解决方案即可。我们通常不会有意识地思考“我以前见过这个问题，我知道该如何处理”，而只是下意识地去做。人类是擅长识别相似的情况的，我们根本不必重复学习如何去商店买牛奶，如何去商店买鸡蛋，如何去商店买糖果。我们知道，去商店购物这件事都是一样的，只是买的东西不同罢了。

识别相似的情况在计算领域内是非常有用的。在计算领域，你会看到某种问题不断地以不同的形式出现。一个好的程序员看到以前解决过的任务或任务的一部分时，会直接选用已有的解决方案。例如，找出温度列表中的最高温和最低温与找出成绩列表中的最高分和最低分是完全相同的任务，都是找出一组数字中的最大值和最小值。

（2）策略 2：分治法

通常，我们会把一个不好处理的大问题分解成几个能直接处理的小问题。打扫一栋房子的任务看起来很繁重，而打扫餐厅、厨房、卧室和浴室的独立任务就容易多了。分治法尤其适用于计算领域，即把大的问题分割成能够单独解决的小问题。

可以把一项任务划分成若干个子任务，而子任务还可以继续划分为子任务。可以反复利用分治法，直到每个子任务都是可以实现的为止。

3）步骤 3：执行方案

执行解决方案，观察每个步骤是否都正确，看看它是否解决了问题。

4）步骤 4：分析解决方案

回顾、分析解决方案，思考一下该解决方案能不能解决其他问题，研究解决方案的未来适用性。

5.1.2 应用示例

下面，让我们应用刚才的步骤及策略解决一个特定的问题，该问题是下星期六在小明家举行聚会，大家如何到达。

需要思考：小明家在哪里？我们从哪里出发？天气如何？走路可能吗？如果开车，有停

车的地方吗？乘坐公共汽车能到达吗？这些问题都得到解答后，就可以开始设计解决方案了。

如果那天下雨，自己的车还在维修，公共汽车停运了，那么最后的解决方案可能是叫出租车，告诉司机小明家的地址。

如果自己开车，查阅地图后知道小明家在自己公司大厦的西边，相隔 6 个街区，那么解决方案的第一部分可能是重复每天早晨都会做的事情——上班（假设从家出发）；接下来是下班后从公司出发，驾车走 6 个街区。如果记不清过了几个街区，那么需要带一支铅笔，每当经过一个街区，就在纸上做一个记号。虽然重复地做记号显得有些麻烦，但在计算机解决方案中这是很常用的。如果要重复一个操作 10 次，就需要编写指令，在每次操作结束时计数，并且检查次数是否达到了 10。在计算领域，这种处理方法叫作重复或循环。

有些人从 A 地出发，其他人从 B 地出发，如果需要给所有人写到达指南，就必须编写两套说明，第一个问题都是“你从哪里出发”，如果从 A 地出发，则采用第一套说明，否则采用第二套说明。在计算领域，这种处理方法叫作条件处理。

根据一步一步的过程解决特定的问题并非不会发生任何变化。事实上，通常需要多次尝试和改进。我们将检验每种尝试，看它是否能真正地解决问题。如果它确实能解决问题，就不需要进一步的尝试，否则需要其他的尝试并验证。

5.2　计算机问题求解

前面讲的解决问题的第二步是设计解决方案。在计算领域，这种解决方案被称为算法。算法是在有限的时间内用有限的数据解决问题的一套明确的指令。在计算领域中，必须明确地描述人类解决方案中暗含的某些条件。

5.2.1　计算机问题求解过程

计算机问题求解的过程包括 4 个阶段：分析和说明阶段、算法开发阶段、实现阶段和维护阶段，如图 5-1 所示。第一阶段的输出是清楚的问题描述；第二阶段的输出是第一阶段定义的问题的一般性解决方案；第三阶段的输出是实现该算法的能在计算机上运行的程

阶段 1：分析和说明阶段

分析	理解（定义）问题
说明	说明程序要解决的问题

阶段 2：算法开发阶段

开发算法	开发用于解决问题的逻辑步骤序列
测试算法	按照列出的步骤操作，看它们是否真正地解决了问题

阶段 3：实现阶段

编码	把算法用程序设计语言实现
测试	让计算机执行程序，检查结果，修改程序，直到得到正确的答案

阶段 4：维护阶段

使用	使用程序
维护	修改程序，使它满足改变了的需求，或者纠正错误

图 5-1　计算机问题求解的过程

序；第四阶段没有输出，除非检测到错误或需要进行更改，如果是这样，这些错误或更改将被以适当的方式发送回第一、第二或第三阶段。

图 5-2 展示了计算机问题求解过程中各个阶段的交互。粗线标明了各阶段间的一般信息流，细线表示在发生问题时可以退回前面阶段的路径。例如，在算法开发阶段，可能会发现问题说明中的错误或矛盾，这样就必须修改分析和说明。同样，实现阶段（程序）中的错误可能表明必须修改算法。

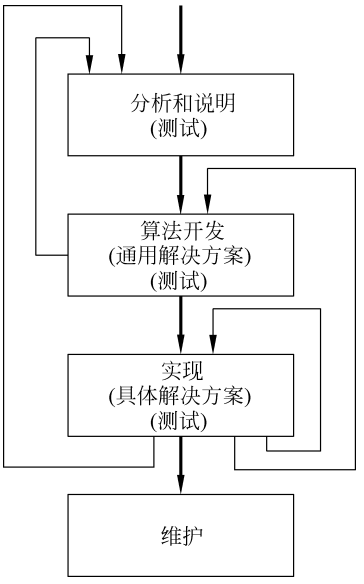


图 5-2　计算机问题求解过程中各个阶段的交互

5.1 节中的波利亚问题求解的所有步骤都包含在使用计算机解决问题的 4 个阶段中。第一步始终是了解问题，你不能为不理解的问题编写计算机解决方案。下一步是为解决方案设计一个计划（一种算法），并用伪代码表示它，这个阶段是本章的重点。再下一步是实现该计划，以使计算机能够执行并测试结果。在波利亚的解决问题步骤中，人类执行计划并评估结果；在计算机解决方案中，人类编写计算机可以执行的程序来表达计划，计算机程序运行后产生结果，人类再检查确认结果是否正确。

在本章中，解决问题的算法用伪代码描述，不涉及具体实现的高级语言，高级语言将在第 6 章介绍。有了正确的算法，用任何高级语言，都能方便地将其编写为能在计算机上运行的程序。

5.2.2　计算机问题求解要点

※读者可观看本书配套视频 7：计算机问题求解要点。

1. 算法开发的要点

设计算法可以分解为以下 4 个主要步骤。

1）分析问题

了解问题，列出必须要使用的信息，此信息可能包含问题中的数据；明确解决方案的呈现方式，如果是报告，请指定格式；列出对问题或信息所做的任何假设；思考该如何手动解决这个问题，制定总体的算法或解决方案。

2）列出主要任务

主要任务列表称为主模块。使用汉语或伪代码重述主模块中的问题，使用任务名称将问题划分为功能区域。如果主模块太长，那就是在这一层级上包含了太多的详细信息，此时需要引入控制结构，如果需要，在逻辑上重新对子部分排序，将细节推给下一级模块。

如果你不知道如何解决一个任务，不要担心，假装你有一个“聪明的朋友”，他知道答案，先将问题细节推后思考。在主模块中要做的就是给出解决某些任务的下级模块的名称，名称须使用有意义的标识符。

3）编写下一级模块

模块没有固定数量的层级，一个级别的模块可以在较低级别划分为更多模块。每个模块都必须是完整的，即使它引用了没写的模块。通过每个模块进行连续的细化，直到每个语

句都是具体的步骤。

4）必要时重新排序和修改

计划不如变换快，解决问题可能需要进行几次尝试和改进，尽量保持清晰、简单、直接的表达。按模块划分解决问题的策略称为自上而下的设计，它的结构是层次结构。

2. 测试算法

计算机解决问题的目标是创建正确的处理过程。实现此过程的算法可以反复使用不同的数据，因此必须对过程本身进行测试或验证。

测试算法通常涉及在计算机上运行对算法进行编码的程序，并检查结果。但是这种类型的测试只能在程序完成或至少部分完成时进行，这已经太晚了，我们不能仅依靠这种测试，因为发现问题越早，解决问题的代价就越小。显然，我们需要在开发过程的早期阶段就执行测试。具体来说，必须先对算法进行测试，然后再用具体的语言（程序）实现它们。

算法的测试过程叫作桌面检查。大多数计算机程序都是由一些程序员（构成一个小组）开发出来的，小组可以采用走查的方法进行算法验证。所谓走查，就是由小组成员采用实例数据手动模拟算法的运行。另一种面向小组的方法是审查。使用这种方法，要预先把设计分发给大家，由一人（非设计者）逐行读出设计，其他人负责指出其中的错误。这种方法是在无胁迫的情况下执行的，目的不是为了批评设计方案或设计者，而是去除产品中的缺陷。有时，要消除这个过程中人的自负感确实有困难，不过好的团队能想办法尽量克服这一点。

5.3　伪代码

在计算机中，解决方案称为算法，伪代码是一种让我们以更清晰的形式表达算法的语言。伪代码不能直接在计算机上运行，最终必须转换为可在计算机上运行的程序。如果算法设计好了，这种转换是很容易实现的。

5.3.1　伪代码的功能

伪代码不是一种计算机语言，更像一种人们用来说明操作的便捷语言。虽然伪代码并没有特定的语法规则，但必须能表示变量、赋值、输入/输出、选择结构、循环结构等概念。

1. 变量

伪代码算法中出现的变量用来指代内存中存储的一个值。变量应该能反映其表示的内容在算法中的角色。

2. 赋值

有了变量，就要有把确定值存入变量的办法。可以采用下面的语句：

```
Set sum to 1
```

这个语句把 1 存放到变量 sum 中。赋值的另一种表示方法是使用反向箭头（←），例如：

```
sum←1
```

用赋值语句把值赋给变量之后，如何访问它们呢？可以用下面的语句访问 sum 和 num。

```
Set sum to sum + num
```

或

```
sum←sum + num
```

上述语句的意思是将存放在 sum 中的值与存放在 num 中的值相加，结果放到 sum 中。因此，当变量放在 to 或←右边时，就能访问它存储的值；当变量用在 Set 的后面或←的左边时，就会向该变量存入一个值。

存入变量的值可以是单个值的形式，也可以是由变量或操作符构成的表达式（如 sum＋num）的形式。

3. 输入/输出

大多数计算机程序需要从外部输入数据，还要能把结果输出到屏幕上。可以使用 Write 语句进行输出，使用 Read 语句从键盘输入。

```
Write "Enter the number of values to read and sum "
Read num
```

双引号之间的字符叫作字符串，可以是汉语、英语等。它告诉用户输出什么内容。输出语句也可以采用 Display 或 Print，这无关紧要，它们都等价于 Write。输入也可以使用 Get 或 Input，都与 Read 同义。记住，伪代码算法是写给人看的，以便之后可以把它转换成程序。对于你自己和要理解你所写的算法的其他人来说，在项目中保持使用单词的一致性是一种好习惯。

下面两个输出语句需要重点说明一下：

```
Write "Err"
Write sum
```

第一条语句把双引号之间的字符串 Err 输出到屏幕上；第二条语句把变量 sum 中的值输出到屏幕上，sum 中的值并不改变。

4. 选择结构

用选择结构可以选择执行或不执行某项操作，也可以在两项操作中选择执行其中一项操作，选择结构括号中的条件决定了执行哪项操作。例如，下面的伪代码或者输出 sum 的值，或者输出一个错误信息。

```
//print error message or sum
IF (sum < 0)
  Print "error message"
ELSE
  Print sum
```

上述伪代码使用缩进对代码段进行分组（在这个例子中只有一组）。符号//用于注释，其主要作用是为了让人更容易理解代码，它并不是算法的一部分。

上面的选择结构称为“IF-THEN-ELSE”，因为是在两个操作之间进行选择。还有一种“IF-THEN”选择结构，则是执行或跳过某操作的情况。如果我们想在任何情况下都打印sum，可以用下面的方式表示算法。

```
IF (sum < 0)
   Print "error message"
Print sum
```

5. 循环结构

使用循环结构可以重复执行指令。下面算法的功能是求和。先输入要加的数字的总次数 limit，然后设置加的次数 counter 为 0，累加和 sum 为 0；再用 WHILE 循环结构检查 counter 是否已到总次数 limit，如果次数没到，就输入数值 num，累加到 sum 中，次数 counter 加 1，然后再回到 WHILE 那里检查加的次数是否已到。与选择结构一样，括号里的表达式是一个条件，如果条件为真，则执行缩进的代码；如果条件为假，则跳到下一个非缩进语句执行，本例中输出累加和 sum 的值。

```
Read limit
Set counter to 0
Set sum to 0
WHILE (counter < limit)
   Read num
   Set sum to sum + num
   Set counter to counter + 1
Print sum
```

WHILE 和 IF 语句括号中的表达式是布尔表达式，其结果要么为真，要么为假。在 IF 语句中，如果表达式为真，则执行缩进语句；如果表达式为假，则跳过缩进语句，如果有 ELSE，则执行 ELSE 下面的语句。对于 WHILE 语句，如果表达式为真，则执行缩进语句，然后再检查表达式的值；如果表达式为假，则跳到下一个非缩进语句执行。WHILE、IF 和 ELSE 都用大写字母，因为它们是专用词语。

这 4 种基本操作（赋值、输入/输出、选择及循环）构成计算机程序的基础。

5.3.2　伪代码示例

本节的示例既包含算法的开发过程，又综合运用了伪代码的 4 种基本操作。

1. 算法开发策略

下面通过一个小规模的算法的开发过程来演示开发策略。

该算法要求读入一些正数数对，然后按序输出数对。如果数对多于一对，就必须使用循环结构。下面是该算法的第一版。

```
WHILE (not done)
    Write "Enter two valuers separated by a blank; press return "
    Read numberl
    Read number2
    Print them in order
```

如何知道循环何时停止呢？也就是说，如何满足 WHILE 语句的 not done 条件，结束循环呢？解决办法是可以要求用户告诉程序要输入多少个数对。下面是算法的第二版。

```
Write "How many pairs of values are to be entered? "
Read numberOfPairs
Set numberRead to 0
WHILE (numberRead < numberOfPairs)
    Write "Enter two valuers separated by a blank; press return "
    Read number1
    Read numter2
    Print them in order
```

最后要按序输出数对，但如何判断数对的大小呢？可以用条件结构比较它们的值；如果 number1 小于 number2，则先输出 number1，再输出 number2；否则，就先输出 number2，再输出 number1。在完成算法前，我们注意到 numberRead 的值从未改变过，因此必须增加 numberRead 的值。下面是该算法的第三版。

```
Write "How many pairs of values are to be entered? "
Read numberOfPairs
Set numberRead to 0
WHILE (numberRead < numberOfPairs)
    Write "Enter two valuers separated by a blank; press return "
    Read number1
    Read numter2
    IF (number1 < number2)
       Print number1 + " " + number2
    ELSE
       Print number2 + " " + number1
    Set numberRead to numberRead + 1
```

在编写这个算法的过程中，我们使用了前面介绍过的两个策略——分析问题及延迟细节。分析问题是我们大多数人都熟悉的策略，延迟细节意味着最初只给任务一个名称，以后再写出完成该任务的详细过程。本示例中，前面的版本先在循环条件中用“not done”和“print them in order”这种抽象的说明描述任务，后续版本中，填写了完成这些任务的详细过程，这个策略被称为分治法。

2. 算法测试

算法在测试完成之前不算真正完成。我们可以选择一组数据，通过纸和笔将代码走一遍。图 5-3 展示了数对算法的测试过程。该算法有 4 个变量：numberOfPairs、numberRead、number1 和 number2，我们必须跟踪它们的值的变化。假设用户在看到提示后输入以下数据：

```
3
10 20
20 10
10 10
```

图 5-3(a)显示了循环开始时的变量值。numberRead 小于 numberOfPairs，因此进入

循环，输出提示，并读取两个数字。这时 number1 为 10，number2 为 20，所以 IF 语句走 THEN 分支，输出 number1，后跟 number2，然后 numberRead 加 1。

图 5-3(b)显示了第一次循环结束时的变量值。numberRead 仍然小于 numberOfPairs，因此代码重复执行，输出提示并读取数字。这时 number1 为 20，number2 为 10，所以选择结构走 ELSE 分支，输出 number2，后跟 number1，然后 numberRead 加 1。

第二次循环结束时变量的状态如图 5-3(c)所示。numberRead 小于 numberOfPairs，代码重复执行，输出提示并读取数字。这时 number1 为 10，number2 为 10，因为 number1 不小于 number2，所以执行 ELSE 分支，输出 number2，后跟 number1。因为两个值是相同的，所以输出顺序无关紧要。

第三次循环结束时变量的状态如图 5-3(d)所示。numberRead 加 1，现在 numberRead 不小于 numberOfPairs，因此代码不再重复。

在这个叫作桌面检查的过程中，我们用笔和纸把设计的算法模拟运行了一遍。这项技术比较简单，但对于测试算法是否正确效果极好。

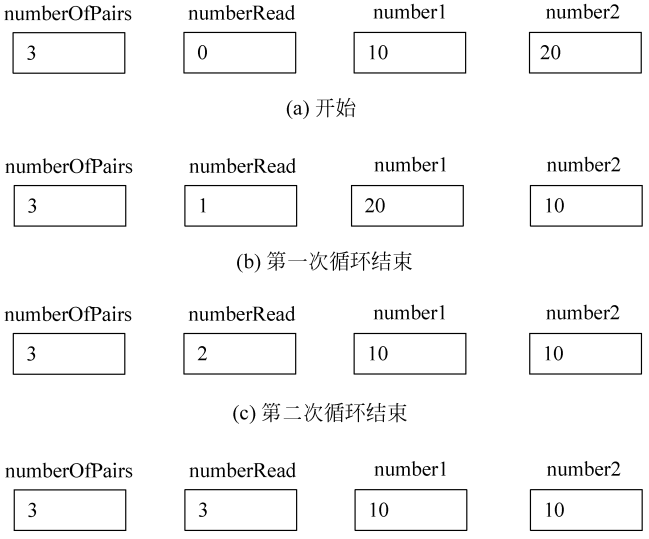


图 5-3　数对算法的测试过程

5.4　算法基础

算法是个很大的话题，本书只能做一点基本的介绍。

简单(原子)变量是那些不能分割的变量，它们是存储在内存某个位置的值。上一节介绍伪代码时举的例子都使用了简单变量，本节前面部分也只使用简单变量。与简单变量相对应的是复合变量，本节后面会介绍。

5.4.1　使用选择结构

※读者可观看本书配套视频 8：选择结构。

5.3 节介绍了基本的选择结构，本节应用较复杂的选择结构描述算法。

假设编写一个算法，该算法能在已知室外温度的前提下，提示适合穿什么衣服。我们是这样想的：如果天气热，穿短裤；如果不冷不热，穿短袖；如果有点凉，穿夹克；如果天气冷，穿厚大衣；如果温度低于冰点，就待在屋里。

开始设计这个算法时，顶级(主)模块只是表示任务。

```
Write "enter the temperature"
Read temperature
Determine dress
```

前两条语句不需要进一步分解，然而，确定穿什么衣服需要和温度联系起来。让我们定义高于 32℃为热，22～32℃为不冷不热，12～22℃为有点凉，0～12℃为冷。现在就可以编写算法来确定穿什么衣服了。

```
IF (temperature > 32)
  Write "Hot weather: wear short"
ELSE IF (temperature > 22)
  Write "Ideal weather: short sleeves fine"
ELSE IF (temperature > 12)
  Write "A little chilly: wear a light jacket"
ELSE IF (temperature > 0)
  Write "Cold weather: wear short"
ELSE
  Write "Stay inside"
```

如果第一个表达式不为真，则到达第二个 IF 语句，如果温度为 22～32℃，那么第二个表达式为真。如果第一个和第二个表达式都不为真，第三个为真，则温度为 12～22℃。同样的推论可以得出，冷的天气温度为 0～12℃，如果温度小于或等于 0℃，则输出“Stay inside”。

5.4.2 使用循环结构

※读者可观看本书配套视频 9：循环结构。

有两种基本的循环类型：计数控制循环和事件控制循环。

1. 计数控制循环

计数控制循环根据设定的次数重复某个过程，循环机制需要在每次重复该过程时进行计数，并在再次开始之前测试该过程是否已完成。计数控制循环在 5.3 节已经使用过了。

计数控制循环由 3 个不同的部分组成，使用了一个称为循环控制变量的特殊变量。第一部分是初始化，循环控制变量初始化为某一起始值；第二部分是测试，检查循环控制变量是否达到预定值；第三部分是递增，循环控制变量加 1。

下面的算法重复一个过程 limit 次(在这段代码之前，limit 已经设定了一个值)。

```
Set count to 0                  //初始化 count 为 0
WHILE(count < limit)            //测试
   …                            //循环体,可以是任何想要的过程
```

```
   Set count to count + 1   //递增
…                           //循环执行完后的语句,注意这一行的缩进与上面一行不同
```

循环控制变量（count）在循环外设置为0。然后检测表达式count＜limit是否为真，只要表达式为真，就执行循环。循环中的最后一个语句递增循环控制变量count。循环执行多少次呢？当count为0，1，2，…，limit－1时，循环执行，因此，循环执行limit次。循环控制变量的初始值和布尔表达式中使用的关系运算符确定了循环执行的次数。

WHILE循环称为预先检测循环，因为在执行循环之前先检测。如果第一次的检测结果为假，则不进入循环。如果省略了递增语句，会发生什么情况呢？情况是布尔表达式永远不会更改，如果开始时该布尔表达式为真，因为表达式不会更改，因此循环将一直执行下去。永不终止的循环称为无限循环，或称为死循环。

2. 事件控制循环

重复次数由循环体内发生的事件控制的循环称为事件控制循环。使用WHILE语句实现事件控制循环时，其过程同样由3个部分组成：初始化事件、测试事件、更新事件。

计数控制循环比较简单，重复次数是指定的，而事件控制循环中3个组成部分可能不是那么明显。下面看一个例子，它完成的功能是循环读取数字，如果为非负数就求和，直到读到负数结束。这里，初始化事件就是读取第一个数字，然后检测该值，如果大于或等于0就进入循环。如何更新事件呢？就是读取下一个数字。下面是该算法的描述。

```
Set sum to 0
Read value                      //初始化事件
WHILE (value > = 0)             //测试事件
   Set sum to sum + value       //value的值加到sum中
   Read value                   //更新事件
…                               //循环结束后的语句
```

现在，再写一个与刚才有点区别的算法，这个算法也是读数字并求正数的和，但设置为计算10个正数的和。这意味着每次读到一个正数时必须对其进行计数，这个计数变量命名为posCount。算法先把posCount设置为0，在WHILE里检测当posCount到10时退出循环。每次读取到正数时，我们都会让posCount加1。

```
Set sum to 0
Set posCount to 0               //初始化事件
WHILE (posCount < 10)           //测试事件
   Read value
   IF (value > 0)
     Set posCount to posCount + 1 //更新事件
     Set sum to sum + value     //value的值加到sum中
…                               //循环结束后的语句
```

这不是一个计数控制的循环，因为我们不是读取10个数字，而是读取多个数字，直到读入了10个正数。请注意嵌入在循环中的选择控制结构。要在任何控制结构中执行或跳过的语句可以是一条语句或多条语句（缩进语句块），这些语句没有什么限制。因此，要跳过或

重复执行的语句可以包含控制结构。选择结构可以嵌套在循环结构中，循环结构也可以嵌套在选择结构中。一个控制结构嵌入到另一个控制结构中的结构称为嵌套结构。

5.4.3 复合变量及用法

之前描述的简单变量存储值的地方本质上都是不可分割的，也就是说，每个地方只能存储一个数据，不能再被分割为更小的部分。本节将介绍两种把数据集合起来的机制，以及单独访问集合中的数据项和整体访问集合的方法。

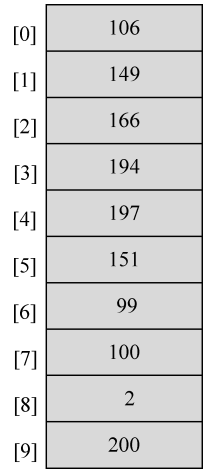


图 5-4 含有 10 个元素的数组

1. 数组

数组是同类数据项的集合，可以通过单个数据项在集合中的位置来访问它们。数据项在集合中的位置称为索引。虽然日常生活中，人们习惯从 1 开始计数，但多数编程语言都是从 0 开始的，因此本书也采用该方式。图 5-4 显示了一个索引从 0 到 9，有 10 个元素的数组。

如果数组叫 numbers，则通过表达式 numbers[position]来访问数组中的每个值，其中 position 就是索引，是一个从 0 到 9 的整数。

下面是将值输入到数组的算法。

```
integer numbers[10]              //声明一个存储 10 个整数值的数组 numbers
Write "Enter 10 integer numbers, one per line"
Set position to 0                //初始化变量 position 为 0
WHILE (position < 10)
    Read numbers[position]
    Set position to position + 1
```

我们通过 integer 后跟数组名称 numbers，并在名称后的括号中写明整数值来指明 numbers 是一个数组，能存储整数值。在前面的算法中，我们没有先列出一个变量，而是假设使用一个变量名时这个变量是存在的，现在我们使用的是复合结构，需要说明想要的是哪一种结构。

与数组有关的算法分为 3 类：搜索（查找）、排序和处理。搜索就是遍历数组中的每个项目，查找特定的值。排序是将数组中的元素按顺序排好，如果元素是数字，将按数字大小排好；如果元素是字符或字符串，则按字母顺序排好。处理是一个统称，包含了对数组中的元素所做的所有其他计算。

2. 记录

记录是不同类型数据项的集合，其中的元素通过名称来进行访问。记录中的元素不必相同，可以包含整数、实数、字符串或其他类型的数据。记录是将与同一对象相关的数据项捆绑在一起的一种很好的选择。例如，我们要读入一个人的姓名、年龄和薪水，可以把这 3 个项集合在一个记录 Employee 中，这个记录由 name、age 和 wage 3 个字段组成，如图 5-5 所示。

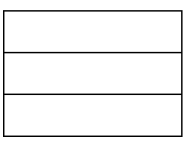


图 5-5 Employee 记录

如果我们声明一个 Employee 类型的变量 employee，则记录的每个字段可以通过记录变量加点加字段名来访问。例如，employee. name 指记录变量 employee 中的 name 字段。记录没有专门的算法，因为它只是相关数据集合在一起的一种方式，但也是引用一组相关数据的便捷方法。

下面的算法是将值存入记录字段中。

```
Employee employee              //声明一个 Employee 类型的变量
Set employee.name to"Frank Jones"
Set employee.age to 32
Set employee.wage to 27.50
```

第 3 种复合数据结构是面向对象编程中的类，我们将在第 6 章讨论这种结构。

5.4.4　搜索算法

1. 顺序搜索

依次搜索每一个元素并与要找的元素进行比较，如果匹配，则找到了该元素，否则继续搜索下一个元素。搜索什么时候停止呢？当找到了这个元素或查找完所有元素都没有找到匹配项时就停止，这听起来像是有两个结束条件的循环。下面使用数组 numbers 编写算法，注意，numbers 有 10 个元素。

```
Set position to 0
Set found to FALSE
WHILE (position < 10 AND found is FALSE)
      IF (numbers[position] equals searchItem)
          Set found to TRUE
      ELSE
          Set position to position + 1
```

在 WHILE 表达式中有复合条件，用到了 AND 这个布尔操作符。布尔操作符包括 AND、OR 和 NOT 等，AND 和 OR 连接两个布尔表达式，NOT 对一个布尔表达式取反。当两个表达式都为真时，AND 操作返回 TRUE(真)，否则返回 FALSE(假)。当两个表达式都为假时，OR 操作返回 FALSE，否则返回 TRUE。这些操作符和第 4 章中描述的门的功能一致。在第 4 章中，门的输入是 1 或 0，在本章中，“真”相当于 1，“假”相当于 0，逻辑是一致的。

我们可以使用 NOT 操作符简化第二个布尔表达式 (found is FALSE)。当 found 为假时，NOT found 就为真，所以也可以写成 WHILE(position < 10 AND NOT found)。只要索引小于 10 且还没有找到匹配项时，循环会一直重复。

2. 有序数组的顺序搜索

如果知道数组中的元素是有序的，那么在查找时，当通过了该元素本应在数组中的位置时，就可以停止查找了。我们来看看这个算法具体是怎么做的。算法使用变量 length 来确定数组中有多少个有效数据项，而不是数组定义时方括号内写的总数据项数，length 可能比数组的总数据项数要小。当有数据读入时，计数器就会更新，因此我们能知道数组中存储了多少个有效数据项。如果数组名为 data，其中的数据就是从 data[0] 到 data[length－1]。

图 5-6 显示了一个从小到大排好序的有序数组。

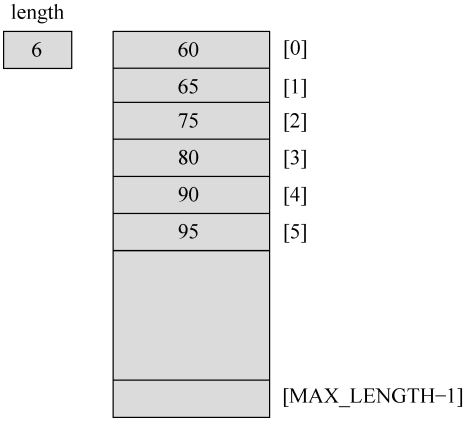


图 5-6 有序数组

在有序数组中，如果我们要查找 76，只要检查到 data[3]就能知道它不在数组中，因为 data[3]的值 80 已经比 76 大了。下面是嵌在一个完整程序中的有序数组搜索算法。

```
Read in array of values
//这里只抽象描述了向数组输入值的功能,为了表明这是抽象描述,用了下画线
Write "Enter value for which to search"
Read searchItem
Set found to TRUE if searchItem is there
//这里只抽象描述了查找算法,具体操作在后面
IF (found)
     Write "Item is found"
ELSE
     Write "Item is not found"
```

下面是 Read in array of values 的具体实现。

```
Write "How many values? "
Read length
Set index to 0
WHILE (index < length)
      Read data[index]
      Set index to index + 1
```

下面是 Set found to TRUE if searchItem is there 的具体实现。

```
Set index to 0
Set found to FALSE
WHILE (index < length AND NOT found)
      IF (data[index] equals searchItem)
           Set found to TRUE
      ELSE IF (data[index] > searchItem)
```

```
        Set index to length
    ELSE
        Set index to index + 1
```

上述代码中，第一个 IF 条件如果为真，就找到了想要的数据，下面一句将 found 置为 TRUE。第二个 IF 条件如果为真，则当前数据项大于要查找的数据，下面一句将 index 置为 length，随后 WHILE 测试不满足 index < length 的条件，就会退出循环。

3. 二分搜索

如何在字典中查找一个单词？希望你不是从第一页开始顺序查找。数组的顺序搜索从数组的开头开始，直到找到匹配项，或者整个数组都搜索过了也没有匹配项。

二分搜索使用与顺序搜索不同的策略进行查找，它采用了分治法。这个方法与你在字典中查找单词是类似的，都是从中间开始，确定要查找的单词是在前部还是后部，然后重复这个方法。

二分搜索要求要查找的数组是有序的，如图 5-7 所示。由于该数组的元素是字符串，其大小已按字典顺序排序好了。搜索从数组的中间开始，如果要搜索的数据项小于数组中间项，那么就可以确定搜索项一定不会在数组的后半部分，因此只需要搜索数组前半部分即可。如果要找的数据项大于中间项，那么继续在后半部分搜索。如果搜索项等于中间项，搜索终止。搜索以这种方式反复进行，每次比较都会将搜索范围缩小一半。当找到匹配项或可能存在匹配项的数组为空时，搜索停止。

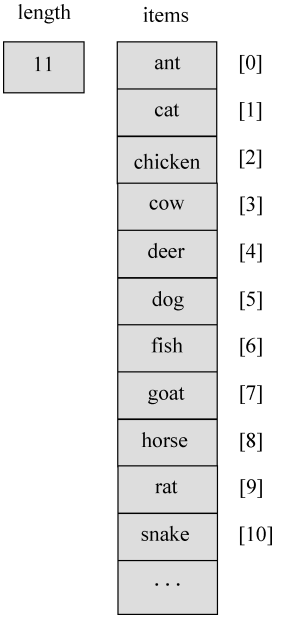


图 5-7　用于二分搜索的有序数组

下面是具体的二分搜索算法。

```
Set first to 0
Set last to length - 1
Set found to FALSE
WHILE (first < last AND NOT found)
    Set middle to (first + last)/ 2
    IF (item equals data[middle])
        Set found to TRUE
    ELSE
        IF (item < data[middle])
            Set last to middle - 1
        ELSE
            Set first to middle + 1
Return found
```

让我们用走查的办法测试一下这个算法，分别搜索 cat、fish 和 zebra。图 5-8 展示了搜索的过程。

二分搜索真的比顺序搜索快吗？表 5-1 展示了顺序搜索和二分搜索的平均比较次数。如果二分搜索这么快，为什么我们不一直使用它呢？原因是，首先二分搜索要求数组必须有

搜寻cat

first	last	middle	比较
0	10	5	cat<dog
0	4	2	cat<chicken
0	1	0	cat>ant
1	1	1	cat=cat Return:TRUE

搜寻fish

first	last	middle	比较
0	10	5	fish>dog
6	10	8	fish<horse
6	7	6	fish=fish Return:TRUE

搜寻zebra

first	last	middle	比较
0	10	5	zebra>dog
6	10	8	zebra>horse
9	10	9	zebra>rat
10	10	10	zebra>snake
11	10		first>last Return:FALSE

图 5-8 二分搜索过程

序；其次，由于需要计算出中间索引，二分搜索的每次比较操作都需要更多的计算。当然如果数组有序且数据项超过 20 个，那么二分搜索是更好的选择。

表 5-1 平均比较次数

数据项个数	顺序搜索	二分搜索
10	5.5	2.9
100	50.5	5.8
1000	500.5	9.0
10000	5000.5	12.0

5.4.5 排序算法

排序就是使事物有序排列。在计算机中，把无序数组转为有序数组是很常见且有用的操作。排序算法有很多种，有不少专门介绍排序的书，这里只介绍几种常见的排序算法。由于对大量元素进行排序非常耗时，所以一个好的排序算法是非常有用的。有时，程序员为了算法运算速度更快，甚至愿意牺牲算法的可理解性。

1. 选择排序

如果给你一组写有学号的答题卡，要求按照学号从小到大进行排序，你可能会浏览一遍答题卡，找到学号最小的，然后把这张答题卡排到新的一组（有序组）的第一个位置。你是怎么确定哪张答题卡的学号最小呢？你可能会把第一张答题卡拿出来放到一边，然后往后，如

果发现有一张答题卡上的学号比刚才拿出来的那张小，就把它拿出来，把原来拿出来那张放回去，再往后翻，重复这个过程，当你检查完所有的卡片后，拿出来的这张就是学号最小的。把这一张放到有序组中，继续重复这个过程直到所有的答题卡都被放到有序组里。

```
WHILE more cards in first deck
    Find smallest left in first deck
    Move to new deck
```

选择排序算法也许是最简单的，因为它与我们手动排序十分相似。未排序的答题卡就像是一个初始数组，排好序的答题卡就是有序数组。

这个算法虽然简单，但也有缺陷，它需要两个完整数组的空间，这显然很浪费。不过，稍微调整一下这种方法就不会重复浪费空间了。当找到最小值并移出时，它所占的位置就空出了，因此不必把最小值写入到第二个数组中，而是将它与它应该在的位置（排好序所处的位置）处的当前值进行交换即可。

图 5-9 是一个选择排序的示例。

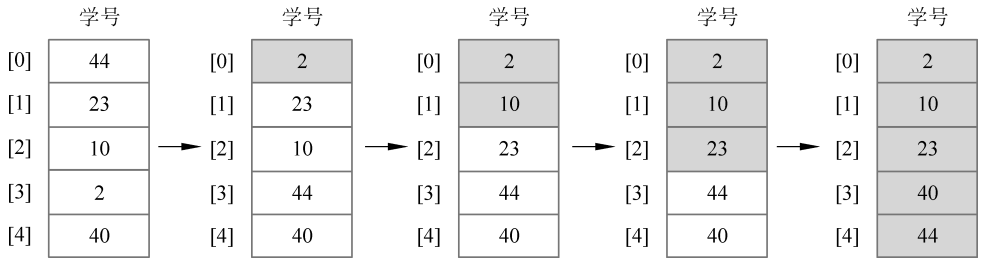


图 5-9　选择排序示例（阴影部分为有序元素）

设想这个数组由两部分组成，即无序部分（非阴影部分）和有序部分（阴影部分）。每当我们把一个元素放到正确的位置，无序部分缩小，有序部分扩展。排序开始时，所有的元素都处于无序部分；排序结束后，所有的元素都处于有序部分。下面是这个算法的描述。

```
//Selection sort
Set firstUnsorted to 0
WHILE (not sorted yet)
    Find smallest unsorted item
    Swap firstUnsorted item with the smallest
    Set firstUnsorted to firstUnsorted + 1
```

这个算法有 3 个抽象步骤（下画线部分），分别是确定数组何时有序、找到最小元素以及互换两者位置。通常情况下，当把最后两项中较小的元素放到正确的位置后，最后一个元素一定也位于正确的位置了。因此只要 firstUnsorted 小于 length －1，循环就会继续。

not sorted yet 功能由下面的语句具体完成。

```
firstUnsorted < length - 1
```

当手动进行排序时，你是如何在无序卡中找到最小学号的卡片呢？你可以翻看卡片，当看到比第一张卡的学号还小的卡片时，记住这个更小的学号，然后继续往后翻卡片，寻找比刚才记住的那个学号更小的卡片。这个过程总是记住迄今为止最小的学号，当翻完全部答题卡，就找到了学号最小的。这个手动算法与这里使用的算法完全相同，只是这里的算法必须记住最小元素的索引（位置），以便与 firstUnsorted 处的元素进行交换。下面是从 firstUnsorted 到 length －1 这部分无序列表中寻找最小元素的代码。

Find smallest unsorted item 功能由下面的语句具体完成。

```
Set indexOfSmallest to firstUnsorted
Set index to firstUnsorted + 1
WHILE (index < = length - 1)
      IF (data[index] < data[indexOfSmallest])
          Set indexOfSmallest to index
      Set index to index + 1
```

交换两个杯子里的液体需要几个杯子？答案是 3 个。当你想把第二个杯子中的液体倒入第一个杯子中时，需要一个临时的杯子存放第一个杯子的液体，互换两个数据也是同样的道理。交换算法需要有被交换的两个元素的索引（位置），以及一个临时变量。

Swap firstUnsorted with smallest 功能由下面的语句具体完成。

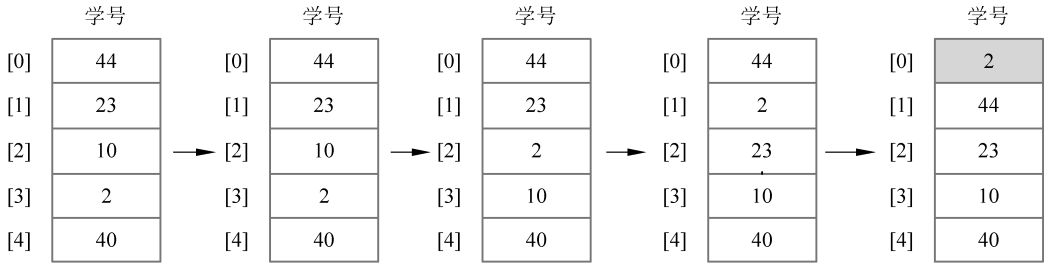
```
Set tempItem to data[firstUnsorted]
Set data[firstUnsorted] to data[indexOfSmallest]
Set data[indexOfSmallest] to tempItem
```

2. 冒泡排序

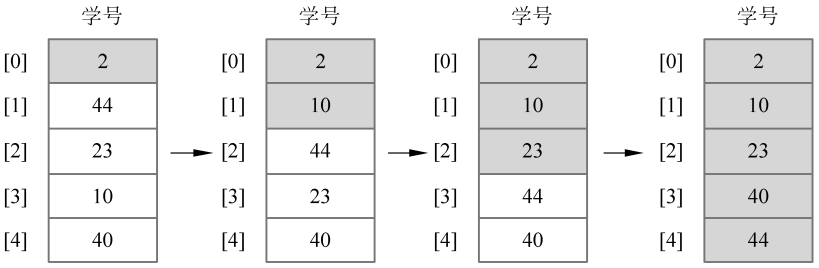
冒泡排序也是一种选择排序，只是在寻找最小值时采用了不同的方法，图 5-10 是冒泡排序的一个示例。排序时从数组最后一个元素开始，与相邻的元素进行比较，当下面的元素小于上面的元素时，就交换这两个元素，图 5-10(a)显示了第一次迭代的过程。这一次迭代完成后，最小的元素就会“冒”到数组的顶部。图 5-10(b)显示了剩余几次迭代完成后的情形，每次迭代都会把未排序的最小元素放到它正确的位置，但也会改变数组中其他元素的位置。

在写这个算法之前，必须说明一下，冒泡排序是非常慢的排序算法。排序算法通常是根据排序数组所需要的迭代次数进行比较，而冒泡排序要对除最后一个元素外的所有元素进行一次迭代。此外，冒泡排序中还有大量的交换操作。既然冒泡排序效率这么差，为什么我们要介绍它呢？因为只要对它稍加修改，就能够让它成为某些情况下的最佳选择。

让我们把冒泡排序应用到一个已经排好序的数组上，如图 5-10(b)最右一列所示。比较 44 与 40，不必交换；再比较 40 与 23，也不必交换；然后比较 23 和 10，仍然不需要交换；最后比较 10 与 2，还是不需要交换。如果在一次迭代过程中没有交换任何数据，那么这个数组就是有序的。在开始循环前，我们把一个布尔型变量设置为 FALSE，如果发生交换，就把它设置为 TRUE。如果循环体运行一遍，布尔型变量仍为 FALSE，那么这个数组就是有序的。



(a) 第一次迭代(阴影部分为有序元素)



(b) 剩余迭代(阴影部分为有序元素)

图 5-10　冒泡排序示例

```
//Bubble sort
Set firstUnsorted to 0
Set swap to TRUE
WHILE (firstUnsorted < length-1 AND swap)
      Set swap to FALSE
      "Bubble up" the smallest item in unsorted part
      Set firstUnsorted to firstUnsorted + 1
```

"Bubble up" the smallest item in unsorted part 的具体实现如下。

```
Set index to length - 1
WHILE (index > firstUnsorted)
      IF (data[index] < data[index - 1])
          Swap data[index] and data[index - 1]
          Set swap to TRUE
      Set index to index - 1
```

在一个有序数组上比较冒泡排序和选择排序的过程，可以观察到，因为选择排序算法不能确定数组是否已经有序，一定要把算法完整执行完，而冒泡排序可以发现数组已排好，提前结束排序。

3. 插入排序

※读者可观看本书配套视频 10：插入排序。

如果数组中只有一个元素，它就是有序的；再加入一个元素，与原来的元素比较，需要的话可以进行交换，现在，这两个元素是有序的；再加入第三个元素，根据原

来两个元素的值，把第三个元素插入到合适的位置，现在，这 3 个元素是有序的。将元素添加到有序部分类似于冒泡排序中的冒泡过程。图 5-11 从左到右展示了插入排序的过程，其中阴影部分是暂时排好序的部分。

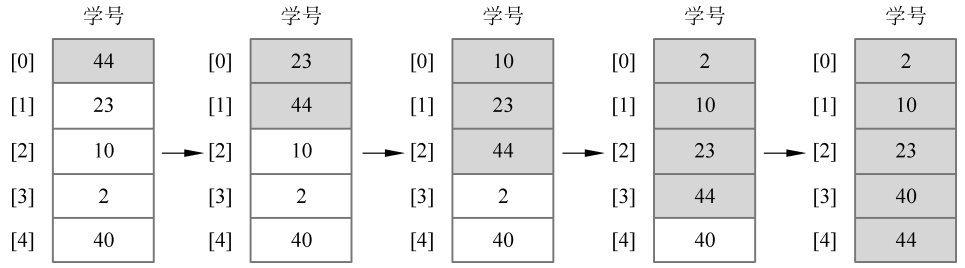


图 5-11　插入排序

```
//Insertion sort
Set current to 1                    // current 是要插入到有序部分的元素的位置
WHILE (current < length)
      Set index to current
      Set placeFound to FALSE
      WHILE ( index > 0 AND NOT placeFound)
           IF (data[ index] < data[ index - 1])
                Swap data[ index] and data[ index - 1]
                Set index to index - 1
           ELSE
                Set placeFound to TRUE
      Set current to current + 1
```

选择排序在每次迭代后，都有一个元素被放到它的永久位置上。而插入排序在每次迭代后，都有一个元素被放到相对于有序部分来说更合适的位置上。

5.5　几个重要思想

本章提到过几个主题，它们不仅对于问题求解重要，对于整个计算领域都很重要。让我们回顾一下在本章中讨论过的一些通用思想。

1. 信息隐蔽

我们已经多次使用过推迟细节的思想。在问题求解时，可以先给任务命名，而把如何实现任务的具体细节推迟到以后再考虑。在设计过程中，高层设计隐蔽细节，把具体细节延后处理具有明显的好处。对于每个特定的层次，设计者只须考虑与该层相关的细节，这种做法叫作信息隐蔽，即在进行高层设计时屏蔽低层的细节。

这种做法看起来很奇怪，为什么在高层设计时不能见到细节呢？设计者不是应该无所不知吗？当然不是。如果设计者知道一个模块的低层细节，他就更可能根据这些细节来构建这个模块的算法，但恰恰这些低层的细节容易发生变化；一旦它们改变了，那么整个算法模块就要重写。

2. 抽象

抽象和信息隐蔽就像一个硬币的两面，信息隐蔽是隐藏细节的做法，抽象则是隐藏细节后的结果。第 1 章提到过，抽象是复杂系统的一种模型，只包括对观察者来说必需的细节。我们用一条狗做个比喻。对狗的主人来说，它是一只宠物；对猎人来说，它是一只猎犬；对兽医来说，它是一只哺乳动物。狗的主人关注的是它摇尾巴的样子，以及它被关在大门外面时的尖叫声，还有随处可见的狗毛。在猎人眼里，它是一个训练有素的帮手，知道自己的工作，并且能够出色地完成。兽医看到的则是构成它身体的器官、血肉和骨头。

在计算领域，算法就是需要实现的步骤的抽象。使用包含算法的软件的一般用户，只需要知道如何运行这个程序，就像狗的主人，只看到表面即可。而在自己程序中使用别人算法的程序员就像使用训练有素的狗的猎人一样，他们有目的地使用算法。算法的开发者必须透彻地了解算法的细节，就像兽医一样，他们必须要看到内部工作原理来实现算法。

在计算领域，有多种抽象。数据抽象指的是把数据的逻辑视图和它的实现分离开。例如，你的开户银行使用的计算机内部可能采用补码表示数字，也可能采用反码表示数字，但是这种区别你无须关注，只要你账户中的钱数正确即可。过程抽象指的是把操作的逻辑视图和它的实现分离开。例如，当我们为子程序命名时，我们正在实践过程抽象。

计算领域中的第三种抽象类型叫作控制抽象，它指的是把控制结构的逻辑视图和它的实现分离开。使用控制结构可以改变算法的顺序控制流。例如，WHILE 和 IF 都是控制结构。在用于实现算法的程序设计语言中如何实现控制结构，对于算法设计来说并不重要。

抽象是人们用来处理复杂事务的最强有力的工具，这句话无论在计算领域还是在现实生活中都是适用的。

3. 事物命名

在编写算法时，我们使用速记符号表示要处理的任务和数据，也就是说，给数据和过程一个名字，这些名字叫作标识符。例如，我们在顺序搜索算法中，用 searchItem 表示要搜寻的数据。

当我们要用一种程序设计语言把算法转换成计算机能够执行的程序时，可能要修改标识符。每种语言都有自己的标识符命名规则，因此，转换过程分两个阶段。首先在算法中命名数据和动作，然后把这些名字转换成符合计算机语言规则的标识符。请注意，数据和动作的标识符都是抽象的一种形式。

4. 测试

我们已经演示了在算法阶段使用桌面走查来进行测试。在实现阶段的测试涉及使用各种数据运行程序，这些数据旨在测试程序的所有部分。后面虽然有章节讨论该阶段的测试理论，但是本章讲解的算法设计阶段的测试也适用于其他阶段。

5.6　小结

波利亚在他的经典著作《怎样解题》中列出了数学问题的求解策略，这个策略适用于所有问题，包括如何用计算机程序求解问题。这个策略的步骤是提出问题，寻找熟悉的情况，

然后用分治法解决。应用这一策略，将生成一个解决问题的方案。在计算领域，这种方案称为算法。

伪代码是一种方便的表示算法的语言。使用伪代码可以命名变量、把数值输入变量以及输出存储在变量中的值。使用伪代码还可以描述算法中重复执行的动作以及选择执行的动作。搜索与排序是算法中常见的内容，本章介绍了几种搜索与排序算法，使读者能更好地理解伪代码及算法的实现过程。

本章提出了4种计算领域常用的重要思想：信息隐蔽、抽象、事物命名和测试。信息隐蔽是隐藏子任务细节的过程，抽象是隐藏细节的结果。在解决与计算机无关的问题时，我们自己执行解决方案，因此知道问题是否解决了。如果问题求解的结果是一个计算机程序，那么就必须全面地测试每个算法，以确保执行程序后，这个程序给出的答案是正确的。

5.7 习题

1. 问答题

(1) 计算机问题求解模型有哪4个阶段？各阶段完成什么工作？

(2) 计算机问题求解模型与波利亚模型有哪些不同之处？

(3) 计算领域有哪几种抽象？

2. 应用题

(1) 用伪代码写一个算法，要求读入姓名，并输出姓名及“早上好”。

(2) 用伪代码写一个算法，要求对循环读入的数值进行累加，如果读入数值小于0，退出循环，输出累加值。

(3) 一个列表中包含以下元素，请使用折半查找算法，给出查找88的步骤，要求给出每一步中first、middle和last的值。

8　13　17　26　44　56　88　97

(4) 使用选择排序算法，手工排序下列数据并给出每次扫描所做的工作。

17　2　8　20　10　32　7

(5) 使用冒泡排序算法，手工排序下列数据并给出每次扫描所做的工作。

17　2　8　20　10　32　7

(6) 使用插入排序算法，手工排序下列数据并给出每次扫描所做的工作。

17　2　8　20　10　32　7