7.23（**海龟图**）Logo语言是在小学生中普及的一门语言，正是它使海龟图（turtle graphics）的概念广为人知。设想有一只机械海龟在一个C++程序的控制下在房间中移动。这只海龟拿着一只笔，笔尖或者朝上，或者朝下。当笔尖朝下时，海龟在移动时就会画出形状；当笔尖朝上时，海龟自由地走来走去，而不会画出任何东西。在这个问题中，你要模拟海龟的动作，并创建一个计算机化的画板。

使用一个初始化为0的20 × 20的数组floor。从一个含有命令的数组中读取命令。随时跟踪海龟的当前位置、当前笔尖的朝向是上还是下。假设海龟总是在floor的(0,0)位置出现，并且笔尖朝上。你的程序所要处理的海龟命令集如图7.33所示。

| 命令 | 含义 |
|---|---|
| 1 | 笔尖朝上 |
| 2 | 笔尖朝下 |
| 3 | 向右转 |
| 4 | 向左转 |
| 5,10 | 向前移动10个格（或者不是10格） |
| 6 | 打印这个20 × 20的数组 |
| 9 | 数据结束（标记值） |

图7.33 海龟图命令

假设海龟在靠近平面中心的某个位置。下面的“程序”将绘制并打印一个12 × 12的正方形，结束时笔尖朝上：

```
2
5,12
3
5,12
3
5,12
3
5,12
1
6
9
```

当海龟朝下握着笔移动时，把相应的floor数组元素设为1。当发出命令6（打印）时，在数组中元素为1的位置显示一个星号或者自己选择的字符。在元素为0的位置显示一个空格。编写一个程序，实现这里所讨论的海龟图功能。编写几个海龟图“程序”，画一些有趣的图形。还可以添加其他一些命令，以增强你的海龟图语言的功能。

7.24（**骑士周游**）骑士周游（knight's tour）问题对国际象棋爱好者来说是较有意思的难题之一。这个问题是：称为骑士的棋子在一个空的棋盘上行进，能否在64个方格棋盘上的每个方格都走一次且只走一次？我们在这道习题中仔细分析一下这个有趣的问题。

在国际象棋中，骑士的移动线路是L形的（在一个方向上走两格，在垂直方向上走一格）。因此，在一个空棋盘中间的方格上，骑士可以有8种不同的移动方式（从0到7编号），如图7.34所示。

a) 在一张纸上画一个8 × 8的棋盘，手工尝试一下骑士的周游。在移进的第一个空格中放1，第二个空格中放2，第三个中放3，依次类推。开始周游之前，估计一下可以走多远，记着一个完整的周游由64步移动组成。走了多远？和你所估计的接近吗？

b) 现在，让我们开发一个程序，将在棋盘上移动骑士。用一个8 × 8的二维数组board表示一个棋盘。每个方格都被初始化为0。根据移动的水平和竖直分量来描述8种可能的移动路线。例如，图7.34中所示的0类移动是沿水平方向右移两格，垂直方向上移动一格。而2类移动则是水平方向左移一格，垂直方向上移动两格。水平向左的移动和竖直向上的移动都用负数来表示。这8种可能的移动方式可以用两个一维数组horizontal和vertical表示，如下所示：

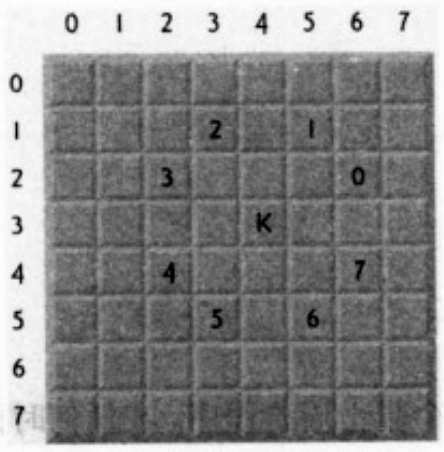


图 7.34　骑士的 8 种可能的移动情况

```
horizontal[ 0 ] = 2
horizontal[ 1 ] = 1
horizontal[ 2 ] = -1
horizontal[ 3 ] = -2
horizontal[ 4 ] = -2
horizontal[ 5 ] = -1
horizontal[ 6 ] = 1
horizontal[ 7 ] = 2

vertical[ 0 ] = -1
vertical[ 1 ] = -2
vertical[ 2 ] = -2
vertical[ 3 ] = -1
vertical[ 4 ] = 1
vertical[ 5 ] = 2
vertical[ 6 ] = 2
vertical[ 7 ] = 1
```

令变量 currentRow 和 currentColumn 表示骑士当前位置的行和列。为了执行一次 moveNumber 类型的移动（其中 moveNumber 在 0 和 7 之间），程序应该使用如下语句：

```
currentRow += vertical[ moveNumber ];
currentColumn += horizontal[ moveNumber ];
```

定义一个从 1～64 变化的计数器。记录骑士在每一方格中移动的最近计数。记住，要检测每种可能的移动，以确定骑士是否已经访问过该方格，当然，也要检测每种可能的移动以保证骑士不会跑到棋盘外面。现在编写一个程序，在棋盘上移动骑士。运行这个程序，看看骑士移动了几次？

c) 在尝试编写和运行了一个骑士周游程序之后，你或许已经发现一些有用的想法了。我们将利用这些智慧，为骑士的移动开发一种试探法（或策略）。试探法不一定保证成功，但是一个精心研制的试探法会极大地增加成功的机会。你可能已经注意到外部的方格比靠近棋盘中心的方格更麻烦。事实上，最麻烦的或难以接近的方格就是四个角落。

最直观的想法是应该先把骑士移动到最难到达的方格，将最容易到达的空出来，这样，在接近周游末期棋盘变得拥挤时，成功的机会就更大。

我们可以开发一个“可达性试探法”，根据每个方格的可到达程度将它们分类，然后总是把骑士移动到最难到达的那个方格（当然，要符合骑士的 L 形移动规则）。我们给一个二维数组 accessibility 填上数，这些数表示每个方格周围有多少个可到达的方格。在一个空棋盘上，每个中心方格定为 8，每个角落方格定为 2，其他的方格为 3、4 或 6，如下所示：

```
2 3 4 4 4 4 3 2
3 4 6 6 6 6 4 3
4 6 8 8 8 8 6 4
4 6 8 8 8 8 6 4
4 6 8 8 8 8 6 4
4 6 8 8 8 8 6 4
3 4 6 6 6 6 4 3
2 3 4 4 4 4 3 2
```

现在，利用可达性试探法编写一个骑士周游程序。在任何时候，骑士都应该移动到具有最低可达数的方格。如果满足此条件的方格不止一个，骑士可以移动到其中的任何一个方格。因此，骑士周游可以从任何一个角落开始。[注意：随着骑士在棋盘上的移动，越来越多的方格被占用，因此你的程序应该随之减少可达数。这样，在周游的任何时刻，每个有效方格的可达数与该方格可到达的确切方格数保持相等。] 运行你的这版程序。能得到一个完整的周游吗？现在修改程序，从棋盘的每个方格开始一个周游，运行64个周游。能得到多少个完整的周游？

d) 编写一个骑士周游程序，当遇到两个或更多的方格有相同的可达数时，通过预测从这些方格可以达到哪些方格来决定选择哪一个方格。在你的程序中，骑士移进的方格，在下一次移动时应该到达一个有最低可达数的方格。

7.25 （**骑士周游：蛮力方法**）在习题7.24中，我们开发了骑士周游问题的一种解决方法。该法使用所谓的“可达性试探法”，可以产生很多解，并且执行效率较高。

随着计算机技术发展的突飞猛进，我们可以凭借其强大的计算能力，利用相对简单的算法来解决更多的问题。这就是解决问题的“蛮力”方法。

a) 用随机数生成来使骑士在棋盘上随意地走动（当然，符合它的L形移动规则）。你的程序运行一次周游，并打印出最终的棋盘。骑士可以走多远？

b) 多数情况下，前面的程序会产生一个相对较短的周游。现在修改这个程序，尝试1000次周游。用一个一维数组跟踪每个长度的周游个数。当程序完成1000次周游尝试后，以表格形式打印这些信息。最好的结果是多少？

c) 多数情况下，前面的程序会得到一些“质量不错”的周游，但不是完整周游。现在，“把次数限制去掉”，只是让你的程序一直运行，直到它产生一个完整周游为止。[注意：这个程序可能会在一台强大的计算机上运行数小时。] 同样，保存每个长度的周游次数表格，当找到第一个完整周游时以表格形式打印出来。你的程序在产生一个完整周游之前尝试了几次？它花费了多少时间？

d) 比较骑士周游问题的蛮力方法和可达性试探法。哪一种需要我们更仔细地研究问题？哪一种算法的开发更难？哪一种需要更强的计算能力？用可达性试探法，我们能预先确定可以得到一个完整周游吗？用蛮力方法，我们能预先确定可以得到一个完整周游吗？讨论一般情况下用蛮力方法解决问题的优点和缺点。

7.26 （**八皇后问题**）另一个关于国际象棋的问题是八皇后问题。简单地说，就是是否可能在一个空的棋盘上放8个皇后，并使这8个皇后之间不会相互“攻击”，即没有两个皇后在同一行、同一列或同一对角线上。利用习题7.24中的思路设计解决八皇后问题的试探法。运行你的程序。[提示：可以给棋盘的每个方格赋予一个值，表明放一个皇后在该方格时，空棋盘上有多少方格可以被“排除”。棋盘四个角落的值都是22，如图7.35所示。] 一旦在64个方格中都放入“排除数”后，一个适当的试探法可以是：把下一个皇后放在具有最小排除数的方格中。为什么会凭直觉选用这个策略？

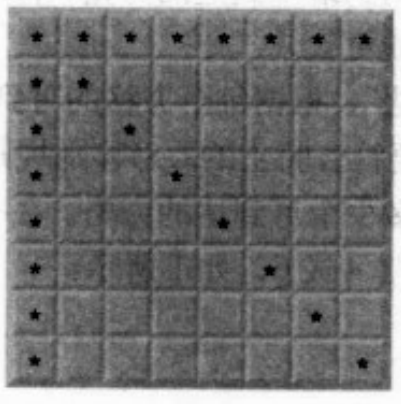

图7.35 放一个皇后在左上角后，要排除的22个方格

7.27（**八皇后问题：蛮力方法**）在这道习题中，你将开发几个蛮力方法来解决习题7.26所介绍的八皇后问题。

a) 利用习题 7.25 中介绍的随机蛮力方法，解决八皇后问题。

b) 使用穷举法，即尝试八个皇后在棋盘上的所有可能组合。

c) 为什么穷举法不适宜解决骑士周游问题?

d) 全面比较随机蛮力方法和穷举蛮力方法。

7.28（**骑士周游问题：封闭周游测试**）在骑士周游问题中，当骑士移动64次并经过每个方格一次且只有一次时，才算是一个完整的周游。封闭周游是指骑士的第65次移动回到了出发点的周游。修改你在习题7.24中编写的骑士周游程序，当产生一个完整周游时，测试它是否是一个封闭周游。

7.29（**Eratosthenes筛选法**）质数是只能被1及其本身整除的数。Eratosthenes筛选法是一种寻找质数的方法。它的操作如下：

a) 创建一个数组，将它的所有元素都初始化为1（真）。下标为质数的数组元素将保持是1，其他元素最后都会被设置为0。在这道习题中，可以不考虑元素0和元素1。

b) 从数组下标2开始，每次找到一个值为1的数组元素时，对数组剩余部分循环，并将下标为该元素下标倍数的元素设置为0。对于数组下标2，数组中2之后的下标为2倍数的元素（下标4、6、8、10等）都被设置为0；对于数组下标3，数组中3之后的下标为3倍数的元素（下标6、9、12、15等）都被设置为0；依次类推。

当这个过程完成之后，仍然为1的数组元素，表明其下标就是一个质数。然后可以把这些下标打印出来。编写一个程序，利用一个1000个元素的数组，判断并打印2到999之间的质数。忽略数组的元素0。

7.30（**桶排序**）桶排序（bucket sort）从一个一维的待排序的正整数数组和一个二维整数数组开始，其中二维数组的行下标是从0到9，列下标是从0到 $n-1$，$n$ 是一维数组中待排序值的个数。这个二维数组的每一行都称为一个桶。编写一个函数bucketSort，它采用一个整数数组和该数组的大小作为参数，并执行以下操作：

a) 对于一维数组的每个值，根据值的个位数，将其放到桶数组的各行中。例如，97放在第7行，3放在第3行，100放在第0行。这称为“分布过程”。

b) 在桶数组中逐行循环编历，并把值复制回原始数组。这称为“收集过程”。上述值在一维数组中的新次序是100、3和97。

c) 对随后的每个数位（十位、百位、千位等）重复这个过程。

在第二遍排序时，100放在第0行，3放在第0行（因为3没有十位），97放在第9行。收集过程之后，一维数组中值的顺序为100、3和97。在第三遍排序时，100放在第1行，3放在第0行，97放在第0行（在3之后）。在最后一次收集过程后，原始数组就是有序的了。

请注意，二维桶数组的大小是被排序的整数数组大小的10倍。这种排序方法的性能比插入排序好，但是需要非常多的内存空间。插入排序只需要额外的一个数据元素的空间。这是一个时空权衡的范例：桶排序使用的内存空间比插入排序多，但是性能较好。这个版本的桶排序需要在每一遍排序时把所有数据复制回原始数组。另外一种方法是创建第二个二维桶数组，并在这两个桶数组间重复交换数据。

## 递归习题

7.31（**选择排序**）选择排序查找数组的最小元素，然后，这个最小元素和数组的第一个元素交换。对从第二个元素开始的子数组，重复这一过程。每次数组排序都会导致一个元素被放到它的恰当位置。这种排序类似于插入排序：对于 $n$ 个元素的数组，必须要执行 $n-1$ 次排序；对于每个子数组，还必须进行 $n-1$ 次比较以找出最小值。当要处理的子数组只包含一个元素时，数组就排序完成了。编写一个递归函数selectionSort执行这个算法。

7.32（**回文**）回文是一种字符串，正读和反读该字符串都会得到同样的拼写。例如，“radar”、“able was i ere i saw elba”和“a man a plan a canal panama”都是回文（如果忽略空格）。编写一个递归函数testPalindrome，如果数组中存储的字符串是回文，则返回true；否则，返回false。这个函数应当忽略字符串中的空格和标点符号。

7.33（**线性查找**）修改图7.19中的程序，用递归函数linearSearch执行对数组的线性查找。该函数应当接收一个整数数组和该数组的大小作为实参。如果找到查找关键值，则返回它的数组下标；否则，返回 -1。

7.34（**八皇后问题**）修改你在习题7.26中创建的八皇后问题程序，用递归方法解决这个问题。

7.35（**打印数组**）编写一个递归函数printArray，它以一个数组、一个开始下标和一个结束下标作为参数，且不返回任何值。当开始下标和结束下标相等时，函数应该停止处理并返回。

7.36（**逆向打印字符串**）编写一个递归函数stringReverse，该函数取一个包含字符串的字符数组和一个开始下标作为实参，逆向打印字符串且不返回任何值。当遇到终止的空字符时，函数应当停止处理并返回。

7.37（**找出数组中的最小值**）编写一个递归函数recursiveMinimum，该函数以一个整数数组、一个开始下标和一个结束下标作为参数，且返回数组的最小元素。当开始下标和结束下标相等时，函数应当停止处理并返回。

## vector 习题

7.38 使用一个整数vector，解决习题7.10中所描述的问题。

7.39 修改你在习题7.17中创建的掷骰子程序，使用一个vector存储两个骰子每种可能的和（sum）出现的次数。

7.40（**查找vector中的最小值**）修改你对习题7.37的解答，在一个vector中查找最小值，而不是在一个数组中。