致谢

2015 地空数算课程教材 4.5 - 5 翻译小组

组长

庞磊

组员

于润泽、蒋明轩、彭玉恒、武化雨、贾博、李昆鹏、陈哲萌、赵琰喆、温景充、余晓辉、龚世泽（排名不分先后）

本文为教材 4.5 - 5

4.5.复杂递归问题

在之前的章节中我们考虑了一些相对容易和灵活有趣的问题来帮助大家理解递归。在这个部分我们将考虑一些用迭代法解决很困难但用递归却很简单的问题。最后我们会以一个看似能用递归解决但实际却不能的问题来作为结尾。

4.5.1.河内塔问题

河内塔问题是法国数学家于 1883 年发现的。他受到一个出现在印度教寺庙年轻僧侣中的问题的启发。在开始的时候，有三个杆串在杆上的 64 个金圆盘，每个圆盘比在它下方的盘小一点。僧侣们的任务是将 64 个圆盘从一根杆上转移到另一根杆上，但有两项重要的限制，一是他们一次只能移动一个圆盘，另一个是不能将大圆盘放在小圆盘之上。僧侣们高效地工作着，日日夜夜地移动圆盘。有传言说当他们完成这项工作的时候这个寺庙将崩塌， 这个世界将终结。

虽然这个传言很有趣，但你不用担心这个世界会很快地结束。完成这项工作需要移动的次数是$2^{64}-1$=18,446,774,073,709,551,615。如果每秒移动一次则需要 584,942,417,335 年！显然实际上会更多。

图 1 展示了一个例子，将圆盘从第一根杆移动到第三根杆。我们注意到，根据规则，圆盘在每根杆上都是从大到小堆放的以保持小盘永远在大盘上面。如果你之前没有度过解决这个问题，现在你可以试一下。你不需要实际的圆盘，一堆书或者纸就可以了。

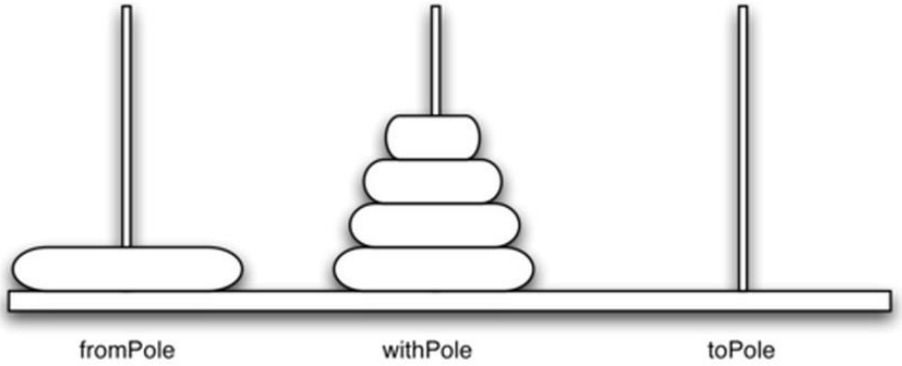


图 1：一个河内塔问题的例子

我们如何用递归地解决这个问题呢？我们又如何完全解决这个问题？基础方案又是什么？让我们从递归调用的最底部开始思考这个问题。假设你有一个有五个圆盘小塔，初始在一号杆上。如果你已经知道如何将有四个圆盘的小塔从一号杆移动到二号杆，你就可以很容易地将第五个圆盘移动到三号杆，最后又将有四个圆盘的小塔从二号杆移动到三号杆。但是如果你不知道如何将有四个圆盘的小塔从一根杆移动到另一根呢？这时又假设你知道如何将有三个圆盘的小塔从一根杆移动到另一根杆，然后你就可以将第四个圆盘移动到目标杆，然后再将有三个圆盘的小塔移动到目标杆。但是如果你不知道如何移动有三个圆盘的小塔又怎么办呢？那考虑先将有两个圆盘的小塔移动到二号杆，再将第三个圆盘移动到三号杆，最后将两个圆盘的小塔移动到三号杆会如何呢？但如果你连这个也不会该怎么办呢？那你必须承认知道如何将单个圆盘移动到三号杆。这听起来像是这个问题中的一个基本情况。

这里有一个高度概要，描述如何将一个小塔从起始杆，经过一个中间杆，移动到目标杆：

- 把 height-1 层数的小塔经过目标杆移动到中间杆
- 把剩下的圆盘移动到目标杆
- 把 height-1 层数的小塔从中间杆，经过起始杆移动到目标杆

只要我们一直遵循大的圆盘保持在底层的规则，我们就能用以上的三步来递归，很容易地处理任意多圆盘的问题。上述概要唯一缺少的是对基本情况的识别。最简单的河内塔问题是只有一个圆盘时的情况，在这种情况下，我们只需要将单个圆盘移动到它的目标杆。所以只有一个圆盘的情况是我们的基本情况。此外，上面的步骤可以通过减少小塔的高度来使我们将问题向基本情况靠近。列表 1 展示了解决河内塔问题的 Python 代码。

列表 1

```
def moveTower(height,fromPole, toPole, withPole):
    if height >= 1:
        moveTower(height-1,fromPole,withPole,toPole)
        moveDisk(fromPole,toPole)
        moveTower(height-1,withPole,toPole,fromPole)
```

我们注意到列表 1 中的代码描述与语言描述非常相近。算法简化的关键在于我们用了两个不同的递归调用，一个在第三行，另一个在第五行。在第三行我们把起始杆上的除了最下面的圆盘全部移动到中间杆，下一行将原来在最底层的最大圆盘移动到目标杆，然后在第五行，我们将位于中间杆的圆盘移动到最大圆盘的上面。当小塔高度为 0 时就是最简情况，在这种情况下，函数没什么可做，直接返回了。重要的事情是记住能这样处理基本情况正是因为直接返回了。

列表 2 中的 moveDisk 函数非常简单，它所做的事情就是打印 出把一个圆盘从一根杆移动到另一根杆。如果你输入然后运行 moveTower 程序，你会发现它能非常有效的解决河内塔问题。

列表 2

```
def moveDisk(fp,tp):
    print("moving disk from",fp,"to",tp)
```

下面的程序为只有三个圆盘的情况提供的一个完整的解决方案。

```
def moveTower(height,fromPole, toPole, withPole):
    if height >= 1:
        moveTower(height-1,fromPole,withPole,toPole)
        moveDisk(fromPole,toPole)
        moveTower(height-1,withPole,toPole,fromPole)

def moveDisk(fp,tp):
    print("moving disk from",fp,"to",tp)

moveTower(3,"A","B","C")
```

现在你重新审视下 moveTower 和 moveDisk 函数，你可能会好奇为什么我们没有一个数据结构用来精确地记录哪一个圆盘在哪一个杆上。这里有一个提示：如果你要精确地记录圆盘的移动，你可以用三个 Stack 对象，每一个对应一根杆。Python 提供了栈，我们只需要调用就行了。

4.6. 探索迷宫

在本节中，我们将着眼一个关于机器人世界遍历的问题：你如何找到你的出路？如果你有一个的 Roomba 机器人吸尘器，以便打扫你的宿舍房间，你会希望根据你在本章所学知识进行重新编程了。我们要解决的问题是帮助我们的海龟在虚拟的迷宫寻找出路。迷宫问题可以追溯到希腊神话中忒修斯被送入迷宫杀牛头的故事。忒修斯使用一端捆绑着球的线，帮助他能够找到出口，最后他结束了野兽。在我们的问题，我们将假设我们的海龟落入迷宫的中央，它必须找到自己的出路。请看图 2 中帮海龟想想出去的办法。

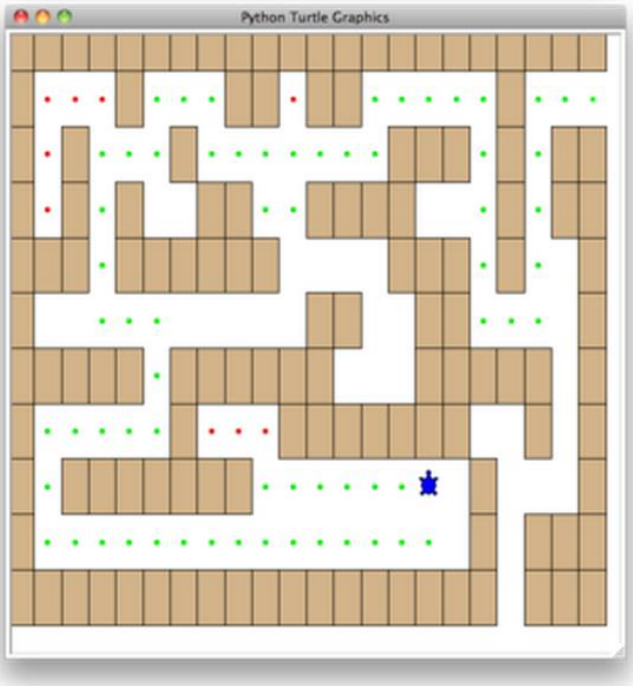


图 2：探索迷宫

为了简化问题，我们假定这个迷宫被分成正方形了。每块小方格是开放或者被不可通过的墙壁占据。海龟只能通过迷宫中开放的部分，如果海龟撞到墙上就必须尝试不同方向。因此，我们需要编写一个系统程序，以找到正确的方式走出迷宫，下面是具体的步骤：

- 从我们初始位置，我们将尝试向北走一步，然后递归我们的程序这这里开始。
- 如果我们不能成功通过尝试北面路径作为第一步，我们将采取向南走一步然后递归重复我们的程序。
- 如果我们不能向南走，那我们将尝试将西面作为我们的第一步并递归使用我们的程序。3
- 如果北面、南面、西面都不能成功地迈出第一步，那么我们将从东门迈出一步并递归我们的应用程序。
- 如果上述几个方向都不能让我们迈出第一步，那么我们将被困在迷宫中出不去了

现在，这听上去很简单，但是还是有几个细节需要说说。第一，假设我们第一步迈向北边，按照我们的程序来看下一步也将是向北。但是如果北边是墙壁，我们就必须着眼于程序中的下一步并尝试迈向南方。不幸的是迈向南边后，我们回到了原来的起点。如果我们的递归过程像这样就会形成一个无限循环。因此，我们必须有一个策略去记住我们走过的路径。假设我们有一个装有面包屑的背包以便于我们可以在行进过程中撒下面包屑，如果我们在某个方向准备迈出一步时发现这上面已经有了面包屑，我们一个个立刻尝试过程中的下一个方向。这样我们将看到这个算法的代码，对于递归调用来说，就是某方向的方格上发现“面包屑”，就立即从递归调用返回上一级。

因为我们做所有的递归算法都是让我们回溯到基础情况。它们中的一些基于前一段的描述，你可能已经猜到了。在这个算法中，有四个基础情况需要考虑：

- 海龟碰到了“墙壁”方格，递归调用结束，返回失败；
- 海龟碰到了“面包屑”方格，表示此方格已访问过，递归调用结束，返回失败；
- 海龟碰到了“出口”方格，即“位于边缘的通道”方格，递归调用结束，返回成功！
- 海龟在四个方向上探索都失败，递归调用结束，返回失败

对于我们的程序运行，我们需要用一种方式来表示迷宫。为了使这更有趣，我们将用海龟模块来绘制和探索迷宫。为了让海龟在迷宫图里跑起来，我们给迷宫数据结构 Maze Class 添加一些成员和方法：

- t：一个作图的海龟，设置其 shape 为海龟的样子（缺省是一个箭头）
- drawMaze()：绘制出迷宫的图形，墙壁用实心方格绘制
- updatePosition(row, col, val)：更新海龟的位置，并做标注
- isExit(row, col)：判断是否“出口”

让我们来看一下被我们成为 searchFrom 搜索功能的代码。如下图所示，注意这个函数有三个参数，一个迷宫，起始行，起始列。这是很重要的，因为作为一个递归函数的搜索逻辑每次递归调用都要重新开始。

列表 3：

```
def searchFrom(maze, startRow, startColumn):
    maze.updatePosition(startRow, startColumn)
   #  Check for base cases:
   #  1. We have run into an obstacle, return false
   if maze[startRow][startColumn] == OBSTACLE :
        return False
    #  2. We have found a square that has already been explored
    if maze[startRow][startColumn] == TRIED:
        return False
    # 3. Success, an outside edge not occupied by an obstacle
    if maze.isExit(startRow,startColumn):
        maze.updatePosition(startRow, startColumn, PART_OF_PATH)
        return True
    maze.updatePosition(startRow, startColumn, TRIED)

    # Otherwise, use logical short circuiting to try each
    # direction in turn (if needed)
    found = searchFrom(maze, startRow-1, startColumn) or \
            searchFrom(maze, startRow+1, startColumn) or \
            searchFrom(maze, startRow, startColumn-1) or \
            searchFrom(maze, startRow, startColumn+1)
```

```
    if found:
        maze.updatePosition(startRow, startColumn, PART_OF_PATH)
    else:
        maze.updatePosition(startRow, startColumn, DEAD_END)
    return found
```

通过算法，你会看到代码的过程（第二行）的第一件事就是调用 updatePosition。这是为了帮助您直观地理解算法，这样您就可以观看海龟究竟是如何探索并通过迷宫的。接下来的算法检查前三的四种基本情况：有海龟碰壁（第五行）？海龟回到标记过的地方（第八行）？海龟找到了出口（第十一行）？如果这些条件不为真，那么我们递归继续搜索。

你会发现，在递归过程中有四个递归调用 searchFrom。它是很难预测这些递归被调用多少次，因为它们都是相互联结的。如果 searchFrom 第一次调用返回 true，则不需要调用后三个。您可以将此理解为迈一步向（上一格）（或北）是更好的走出迷宫的路径。如果向北走不是一个很好的路径走出迷宫那么接下来将进行递归调用尝试，这一次向南。如果南边失败，那就再试试向西，最后是向东。如果所有四个递归调用都返回 false，那么我们已经进入了死胡同。你需要下载或键入整个程序和调整它通过改变这些调用的顺序。

对于迷宫类的代码如 list 4 所示，list 5 和 list 6 中__init__方法接受一个文件作为其唯一的参数的名称。此文件是通过使用“+”字符作为墙壁，包围成的空心正方形空间，并用字母“S”来表示起始位置的表示一个迷宫的文本文件。图 3 是一个迷宫数据文件的例子。迷宫的内部表示是一个 2*2 列表。所述 mazelist 实例变量的每一行也是列表。该 2*2 列表用上述的字符表示每平方一个方格。对于数据文件 3 的内部表示如下所示：

```
[ ['+','+','+','+',...,'+','+','+','+','+','+','+','+'],
  ['+',' ',' ',' ',' ',....,' ',' ',' ',' ','+',' ',' ',' ',' ','+'],
  ['+',' ',' ','+',' ',...,'+','+',' ',' ','+',' ',' ','+','+'],
  ['+',' ',' ','+',' ',....,' ',' ',' ',' ','+',' ',' ','+','+'],
  ['+',' ','+','+',' ',...,'+','+',' ',' ','+',' ',' ',' ','+'],
  ['+',' ',' ',' ',' ',' ',....,'+','+',' ',' ',' ',' ',' ',' ','+'],
  ['+','+','+','+',...,'+','+','+','+','+','+',' ',' ','+'],
```

```
['+',' ',' ',' ',...,'+','+',' ',' ',' ','+',' ',' ','+'],
['+',' ',' ','+','+',...,' ',' ',' ','+',' ',' ',' ',' ','+'],
['+',' ',' ',' ',' ',...,' ',' ',' ','+',' ',' ','+','+','+'],
['+','+','+','+','+',...,'+','+','+',' ',' ','+','+','+']]
```

该 drawMaze 方法使用这样的内部表示，以在屏幕上绘制迷宫的初始视图。

```
++++++++++++++++++++
+   +   ++ ++    +
+ +   +      +++ + ++
+ + +  ++  ++++   + ++
+++ ++++++    +++ +  +
+         ++  ++    +
+++++ ++++++   +++++ +
+     +   +++++++  + +
+ +++++++     S +   +
+               + +++
++++++++++++++++ +++
```

图 3: 一个迷宫数据文件的例子

updatePosition 方法，如 list 5 中使用的显示，观察海龟是否已经碰壁。它还更新内部表示用“。”或“ - ”来表示乌龟已经访问特定的路径或者路径是死胡同的一部分。此外，该 updatePosition 方法使用两个辅助方法，moveTurtle 和 dropBreadCrumb，用以更新屏幕上的图。

最后，该 isExit 方法使用海龟的当前位置来测试退出条件。退出条件是每当龟已导航到迷宫的边缘，无论是 0 行或 0 列，或最右列或最后行。

列表 4

```
class Maze:
    def __init__(self,mazeFileName):
        rowsInMaze = 0
        columnsInMaze = 0
        self.mazelist = []
        mazeFile = open(mazeFileName,'r')
        rowsInMaze = 0
        for line in mazeFile:
            rowList = []
```

```
            col = 0
            for ch in line[:-1]:
                rowList.append(ch)
                if ch == 'S':
                    self.startRow = rowsInMaze
                    self.startCol = col
                col = col + 1
            rowsInMaze = rowsInMaze + 1
            self.mazelist.append(rowList)
            columnsInMaze = len(rowList)

        self.rowsInMaze = rowsInMaze
        self.columnsInMaze = columnsInMaze
        self.xTranslate = -columnsInMaze/2
        self.yTranslate = rowsInMaze/2
        self.t = Turtle(shape='turtle')
        setup(width=600,height=600)
        setworldcoordinates(-(columnsInMaze-1)/2-.5,
                            -(rowsInMaze-1)/2-.5,
                            (columnsInMaze-1)/2+.5,
                            (rowsInMaze-1)/2+.5)
```

列表 5

```
def drawMaze(self):
    for y in range(self.rowsInMaze):
        for x in range(self.columnsInMaze):
            if self.mazelist[y][x] == OBSTACLE:
                self.drawCenteredBox(x+self.xTranslate,
                                     -y+self.yTranslate,
                                     'tan')
    self.t.color('black','blue')

def drawCenteredBox(self,x,y,color):
    tracer(0)
    self.t.up()
    self.t.goto(x-.5,y-.5)
    self.t.color('black',color)
    self.t.setheading(90)
    self.t.down()
    self.t.begin_fill()
    for i in range(4):
        self.t.forward(1)
        self.t.right(90)
```

```
    self.t.end_fill()
    update()
    tracer(1)

def moveTurtle(self,x,y):
    self.t.up()
    self.t.setheading(self.t.towards(x+self.xTranslate,
                                     -y+self.yTranslate))
    self.t.goto(x+self.xTranslate,-y+self.yTranslate)

def dropBreadcrumb(self,color):
    self.t.dot(color)

def updatePosition(self,row,col,val=None):
    if val:
        self.mazelist[row][col] = val
    self.moveTurtle(col,row)

    if val == PART_OF_PATH:
        color = 'green'
    elif val == OBSTACLE:
        color = 'red'
    elif val == TRIED:
        color = 'black'
    elif val == DEAD_END:
        color = 'red'
    else:
        color = None

    if color:
        self.dropBreadcrumb(color)
```

列表 6

```
def isExit(self,row,col):
     return (row == 0 or
             row == self.rowsInMaze-1 or
             col == 0 or
             col == self.columnsInMaze-1 )

def __getitem__(self,idx):
     return self.mazelist[idx]
```

完整的程序示于 ActiveCode1.本程序使用以下所示的数据文件 maze2.txt。

注意，它是一个更简单的例子文件中，该出口是非常接近的龟的开始位置。

```
++++++++++++++++++++
+  +  ++ ++       +
     +    ++++++++++
+ +   ++  ++++ +++ ++
+ +  + + ++   +++  +
+        ++  ++  + +
+++++ + +     ++  + +
+++++ +++  + +  ++   +
+        + + S+ +  +
+++++ +  + + +    + +
++++++++++++++++++++
```

```
import turtle

PART_OF_PATH = 'O'
TRIED = '.'
OBSTACLE = '+'
DEAD_END = '-'

class Maze:
    def __init__(self,mazeFileName):
        rowsInMaze = 0
        columnsInMaze = 0
        self.mazelist = []
        mazeFile = open(mazeFileName,'r')
        rowsInMaze = 0
        for line in mazeFile:
            rowList = []
            col = 0
            for ch in line[:-1]:
                rowList.append(ch)
                if ch == 'S':
                    self.startRow = rowsInMaze
                    self.startCol = col
                col = col + 1
            rowsInMaze = rowsInMaze + 1
            self.mazelist.append(rowList)
            columnsInMaze = len(rowList)

        self.rowsInMaze = rowsInMaze
        self.columnsInMaze = columnsInMaze
        self.xTranslate = -columnsInMaze/2
```

```
        self.yTranslate = rowsInMaze/2
        self.t = turtle.Turtle()
        self.t.shape('turtle')
```

完整迷宫解决(completemaze)

自我检测

修改迷宫搜索程序，使得调用 searchFrom 是按照不同的顺序。观看程序运行。你能解释为什么行为是不同的？你能预测一下路径龟将遵循何种给定的变化？

4.7. 动态规划

在计算机科学中，许多程序是写来用于使一些问题得到最优解；例如，找到两点间的最短路径，找到最匹配一组点的线，或找到满足某些条件的最小对象集。计算机学家有许多策略来解决这些问题。这本书的目的之一就是为你揭示几个不同的解题策略。动态规划是这类求最优解问题的解决策略之一。

动态规划的一个典型例子就是用最少的硬币来找零。假设你是一家自动售货机制造商的程序员。你的公司正设法在每一笔交易找零时都能提供最少的零钱以便工作能更加简单。假设一个顾客投了 1 美元来购买 37 美分的物品。你能用来找零的最少硬币数量是多少？答案是 6 枚硬币：2 个 25 美分，1 个 10 美分，3 个 1 美分。我们是怎么得到 6 个硬币这个答案的呢？首先我们要使用面值中最大的硬币（25 美分），并且尽可能多的使用它，接着我们再使用下一个可使用的最大面值的硬币，也是尽可能多的使用。这种方法被称为贪心法，因为我们试图尽可能快的解决一个问题。

当我们使用美国硬币时，贪心算法工作的很好，但假设你的公司决定在 Lower Elbonia 也部署自动售货机，那个地方除了有 1，5，10 和 25 美分的硬币外，还有 21 美分的硬币。在这种情况下，贪心算法就不能找到 63 美分找零问题的最优解了。多了 21 面值的美分，贪心算法的答案仍是 6 个硬币，然而问题的

最优解是 3 个 21 美分。

让我们来看看一个算法，这个算法肯定能让我们找到问题的最优解。既然这章是关于递归的，你可能已经猜到我们将使用递归的解决方法。首先我们要弄清楚基本结束条件。如果我们要找的零钱的价值和某一个硬币的价值一样，那么答案很简单，只要一个硬币。

如果价值不匹配，我们就有几种选择。我们需要的是 1 个 1 美分加上给原始价值减去 1 美分找零所需硬币数量的最小值，或者 1 个 5 美分加上给原始价值减去 5 美分找零所需硬币数量的最小值，或者 1 个 10 美分加上给原始价值减去 10 美分找零所需硬币数量的最小值，等等。所以，给原始总数找零的硬币数量可以根据下面的方法计算：

$$numCoins = min \begin{cases} 1 + numCoins(originalamount - 1) \\ 1 + numCoins(originalamount - 5) \\ 1 + numCoins(originalamount - 10) \\ 1 + numCoins(originalamount - 25) \end{cases}$$

完成我们刚才的描述的问题的算法在 Listing 7 展示出来了。在第三行，我们检查基本结束条件；也就是说，需要兑换的找零数等于我们硬币的某个面值。如果我们没有等于找零数目的硬币面额，那么我们就对每个小于我们找零总数的不同的硬币值调用递归。第六行展示了我们应该怎样通过使用一个硬币面值的列表，以帮助我们筛选出比当前找零价值小的硬币的列表。通过选定硬币的值，递归调用减小了我们需要找零的总数。递归调用在第七行展示了。注意，在同一行，我们要给硬币总数加 1，这是因为我们使用了一枚硬币。只需加 1 就相当于一满足基本结束条件时我们就做一次递归调用。

列表 7

```
def recMC(coinValueList,change):
   minCoins = change
   if change in coinValueList:
     return 1
```

```
    else:
        for i in [c for c in coinValueList if c <= change]:
            numCoins = 1 + recMC(coinValueList,change-i)
            if numCoins < minCoins:
                minCoins = numCoins
    return minCoins

print(recMC([1,5,10,25],63))
```

Listing 7 这种算法的问题就是它太低效了。事实上，它需要 67716925 次递归调用才能得出 63 美分问题的最优解是 4 枚硬币！为了理解我们的算法中的致命缺陷，观察图 3，它列出了我们为了找到兑换 26 美分的最优解所需的 377 次函数调用中的一小部分。

图中每一个节点对应一次函数调用 recMC。节点上的数字显示了我们要计算的需要找零的硬币总量。箭头上的数字则显示了我们使用的硬币的面额。顺着图形，我们可以看到图中任何点的硬币的组合。主要问题就是我们做了大量的重复计算。例如，该图显示，这种算法会重复计算为 15 美分找零的最优解至少三次。每一次这种计算都要调用 52 次函数。显然我们浪费了大量的时间和精力来重复计算旧的结果。

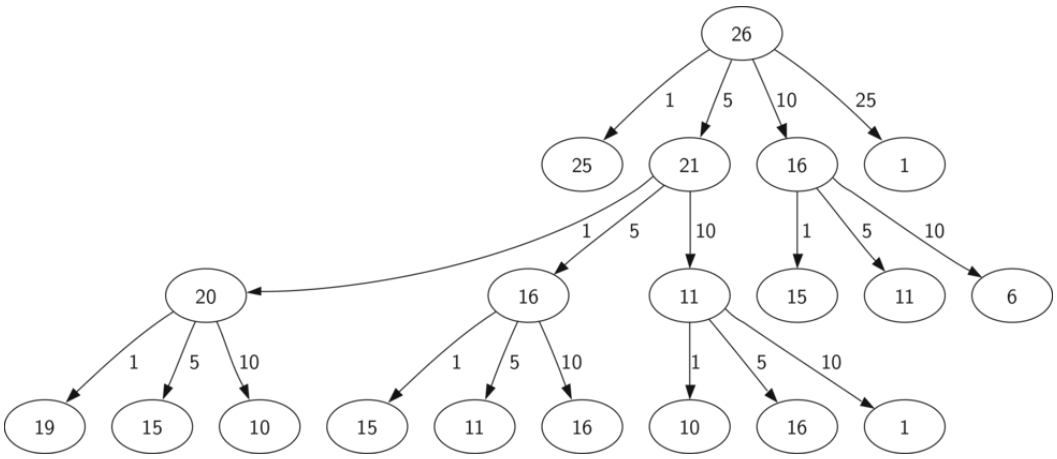


图 3：Listing 7 的函数调用树

减少我们的工作量的关键在于记住一些出现过的结果，这样就能避免重复计算我们已经知道的结果。一个简单的解决方案就是当我们一找到给硬币找零的最

小数目时，我们就将这个结果存储在一个表中。然后在我们计算一个新的最小值之前，可以先查表看这个结果是否已知。如果表中已经有了这个结果，我们就可以从表中引用这个值而不是重复计算。ActiveCode1 展示了包含了查表法的改善算法。

```
def recDC(coinValueList,change,knownResults):
   minCoins = change
   if change in coinValueList:
      knownResults[change] = 1
      return 1
   elif knownResults[change] > 0:
      return knownResults[change]
   else:
       for i in [c for c in coinValueList if c <= change]:
         numCoins = 1 + recDC(coinValueList, change-i,
                              knownResults)
         if numCoins < minCoins:
            minCoins = numCoins
            knownResults[change] = minCoins
   return minCoins

print(recDC([1,5,10,25],63,[0]*64))
```

ActiveCode 1：查表法递归硬币计数(lst_change2)

注意，在第六行我们添加了一个测试来检查表中是否包含了为某个特定数目找零的硬币的最小值。如果没有，我们就调用递归来计算这个最小值并把它存储在表中。使用这个改进后的算法减少了我们得到这 4 个硬币的答案所需的递归调用次数，63 美分的问题只需要 221 此调用！

虽然 AcitveCode 3 的算法是正确的，但看起来感觉好像被黑客黑过一样。此外，如果我们观察 knowResults 列表我们会发现，表中还有不少空洞。事实上，我们目前所做的还不是动态规划，我们只是使用了一种叫做“memoization（函数值缓存）”，或者一般称为“caching（缓存）”的方法改善了程序的性能。

真正的动态规划会采用更系统化的方法来解决问题。动态规划的解决方法是

从为 1 分钱找零的最优解开始，逐步递加上去，直到我们需要的找零钱数。这就保证了在算法的每一步过程中，我们已经知道了为这个数值的硬币兑换任何更小数值的零钱的最小值。

让我们来看看我们如何在为 11 美分找零时用最小数目的硬币使用量来填表。图 4 列出了过程。我们先从一美分开始。唯一可行的解决方案就是就是一个硬币（一美分）。下一行显示了为 1 美分和 2 美分找零的最小值。同样，唯一的答案是 2 个 1 美分硬币。在第 5 行事情开始变得有趣了。现在我们考虑两种情况，5 个 1 美分硬币或 1 个 5 美分硬币。我们如何决定那个才是做好的选择呢？通过查表我们可以看到，为 4 美分找零的数量是 4 个，再加一个 1 美分变成了 5 美分，相当于有 5 个硬币。或者我们可以考虑 0 个硬币加上 1 个 5 美分硬币等于 5 美分，只有一个硬币。由于 1 比 5 小，我们把 1 存储在列表中。很快到了表的结尾，该考虑 11 美分了。图 5 显示了我们必须考虑的三种情况：

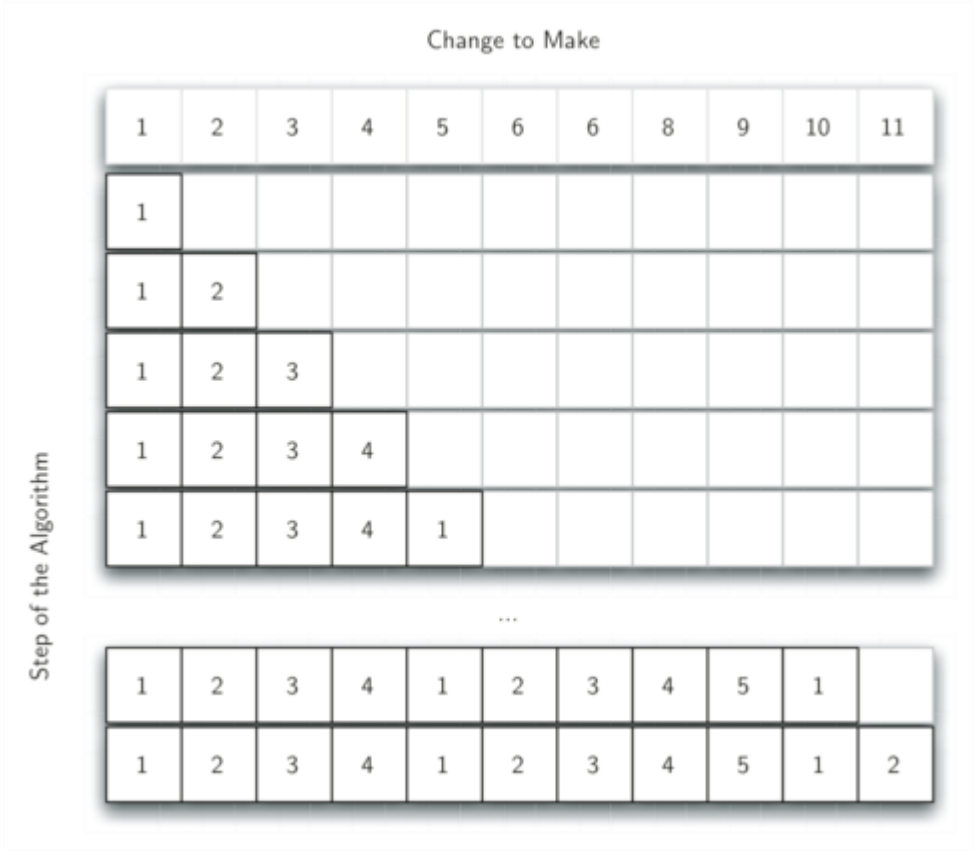
1．一个 1 美分加上为 11-1=10 美分找零的最小值（1）

2．一个 5 美分加上为 11-5=6 美分找零的最小值（2）

3．一个 10 美分加上为 11-10=1 美分找零的最小值（1）

情况 1 和 3 都给出了为 11 美分找零的最小值是 2 个硬币的答案。

图 4：需要找零的硬币数量的最小值



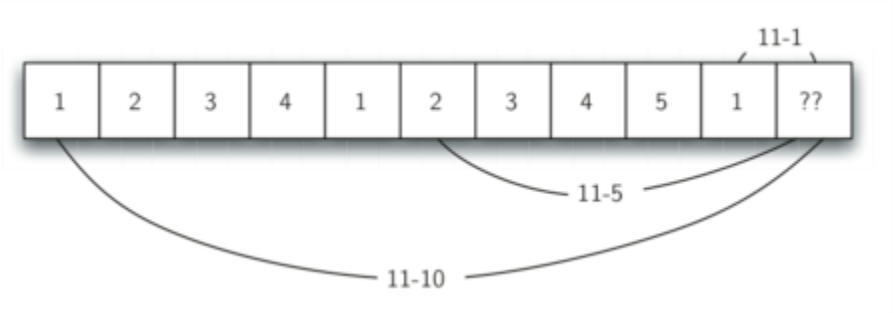


图 5:考虑为 11 美分找零的最小值的 3 种方案

Listing 8 是一个为解决找零问题的动态规划算法。dpMakeChnge 有三个参数：一个有效硬币面值的列表、我们想要兑换硬币的数值、一个包含所有部分找零最优解的列表。当函数运行完，minCoins 会包含从 0 到所需兑换数值的每一种情况对应的最优解。

列表 8

```
def dpMakeChange(coinValueList,change,minCoins):
   for cents in range(change+1):
      coinCount = cents
      for j in [c for c in coinValueList if c <= cents]:
            if minCoins[cents-j] + 1 < coinCount:
               coinCount = minCoins[cents-j]+1
      minCoins[cents] = coinCount
   return minCoins[change]
```

注意，dpMakeChange 不是一个递归函数，即使我们开始使用了来递归解决这个问题。必须要认识到的是，你可以写一个递归算法来解决问题，但这并不意味着它就是最好和最有效的解决方案。这个函数的大部分内容是做循环，循环从第 4 行开始。在这个循环中，我们要考虑使用所有可能的硬币面值来为指定的数值兑换硬币。像我们在上面举的给 11 分钱兑换硬币的例子，我们把部分找零的最优解记录下来并保存在 minCoins 列表中。

尽管我们的找零算法在找出所需硬币数量的最小值上做得很好，但是它并不能真的帮助我们兑换硬币，因为我们没有跟踪记录我们使用的硬币。我们可以很

容易地扩展 dpMakeChange 来跟踪记录我们所使用的硬币，只要简单的记录我们为 minCoins 的每一项添加的最后一个硬币就可以了。如果我们知道了最后一个添加的硬币，就可以简单的减去这个硬币的币值来找到最优解列表中之前的一项进行找零。我们可以一直倒退访问列表直到回到列表的最开始。

ActiveCode 2 显示了改善了性能的 dpMakeChnge 算法跟踪了硬币使用的路径，同时有一个 PrintCoins 的功能，通过重访列表打印出每个使用过的硬币的值。这表明了这个算法解决了我们在 Lower Elbonia 的朋友的问题。main 的前两行设置了要转换的总量并且创建了可使用的硬币面值的列表。接下来的两行创建了我们存储结果的列表。coinUsed 是一个我们用来找零的硬币的列表，coinCount 是为表中相应的位置的量找零所需的硬币最小值。

注意我们打印出的硬币值直接来自 coinUsed 阵列。第一次调用函数时，我们从阵列的 63 位置开始并打印出 21。然后我们得到 63-21=42 并观察列表的第 42 级。我们又一次发现存储了 21。最后，列表的 21 级也包含了 21，最后我们得到了 3 个 21 分硬币。

```
def dpMakeChange(coinValueList,change,minCoins,coinsUsed):
   for cents in range(change+1):
      coinCount = cents
      newCoin = 1
      for j in [c for c in coinValueList if c <= cents]:
            if minCoins[cents-j] + 1 < coinCount:
               coinCount = minCoins[cents-j]+1
               newCoin = j
      minCoins[cents] = coinCount
      coinsUsed[cents] = newCoin
   return minCoins[change]

def printCoins(coinsUsed,change):
   coin = change
   while coin > 0:
      thisCoin = coinsUsed[coin]
      print(thisCoin)
      coin = coin - thisCoin
```

```
def main():
    amnt = 63
    clist = [1,5,10,21,25]
    coinsUsed = [0]*(amnt+1)
    coinCount = [0]*(amnt+1)

    print("Making change for",amnt,"requires")
    print(dpMakeChange(clist,amnt,coinCount,coinsUsed),"coins")
    print("They are:")
    printCoins(coinsUsed,amnt)
    print("The used list is as follows:")
    print(coinsUsed)

main()
```

ActiveCode 2：圆满的解决了找零问题(lst_dpremember)

4.8. 小结

在本章我们看了几种递归算法的例子。这些算法向你展示了几种不同的可以用递归解决的问题，表明递归是一种解决问题的有效技术。本章的要点有以下内容：

- 所有递归算法必须具备基本结束条件。
- 递归算法必须要减小规模，改变状态，向基本结束条件演进。
- 递归算法必须调用自身（递归地）。
- 某些情况下，递归可以代替迭代循环。
- 递归算法通常能够跟问题的表达很自然地契合。
- 递归不总是最合适的算法，有时候递归算法可能会引发巨量的重复计算。

4.9. 关键词

基本结束条件	解密	动态规划
递归	递归调用	栈框架

4.10. 问题讨论

1. 为汉诺塔问题设计一个调用栈。假设开始时有三个盘片。

2. 运用上文描述的递归规则，用纸和笔画一个谢尔宾斯基三角形。

3. 运用找零的动态规划算法，找出为 33 美分找零所需硬币的最少数量。除了通常的硬币面值以外，假设你还有面值为 8 美分的硬币。

4.11. 词汇表

基本结束条件

在一个递归函数中，不会引起进一步的递归调用的条件语句的一个分支。

数据结构

一种数据的组织方案，可以使数据使用更加方便。

异常

程序运行时出现的错误。

异常处理

通过将一部分代码放入一个 try-except 结构来防止异常终止程序。

不可变数据类型

一种不可以改变的数据类型。元素的操作或不可变类型的切片操作会导致运行错误。

无限递归

一个函数不停地递归调用自己，无法达到基本结束条件。最终，一个无限递归导致运行出现错误。

可变数据类型

一种可以修改的数据类型。所有的可变类型都是复合类型。列表和字典（见下一章）是可变数据类型；字符串和元组不是。

raise

使用 raise 语句引发一个异常。

递归

调用已经执行的函数的过程。

递归调用

调用已经执行的函数的语句。递归也可以不是直接的——函数 f 可以调用 g，g 调用 h，而 h 又调用回 f。

递归定义

一个定义自己本身的定义。为了发挥作用，递归定义必须包含不是递归的基本结束条件，使它不同于循环定义。递归的定义通常可以提供一种优雅的方式来表达复杂的数据结构。

元组

包含一个由任何类型的元素组成的序列的数据类型，像一个列表，但是是不可变的。元组可以用于任何一个必须是不可变类型的情况，比如字典中的 key（见下一章）。

元组赋值

使用一个赋值语句就可以给元组中的所有元素赋值。赋值是平行发生的，而不是顺序发生的，这对于两个元组之间数值的交换十分有用。

4.12. 编程练习

1. 写一个递归函数来计算一个数的阶乘。

2. 写一个递归函数来反转一个列表。

3. 修改递归树的程序，增加如下功能:

   - 树枝的粗细可以变化，随着树枝缩短，也相应变细。
   - 树枝的颜色可以变化，当树枝非常短的时候，使之看起来像树叶的颜色。
   - 让树枝倾斜角度在一定范围内随机变化，如 15~45 度之间，运行一下看看怎样比较好看。

- 树枝的长短可以变化，使之不用一直保持同样的数值，而是在一定范围内随机变化。

如果你实现了以上所有功能，你将会获得一棵非常逼真的树。

4. 找出或发明一种算法绘制分形山。提示：其中一种方法再次利用了三角形。

5. 写一个递归函数来计算斐波那契数列。递归函数的性能和利用迭代计算的方法来比较，效果如何？

6. 采取一种方法解决汉诺塔问题，利用三个栈来跟踪盘子的轨迹。

7. 使用海龟绘图模块，编写递归程序画出希尔伯特曲线。

8. 使用海龟绘图模块，编写递归程序画出科赫雪花。

9. 写一个程序来解决以下问题：你有两个水壶，一个 1 加仑的水壶和一个 3 加仑的水壶，两个水壶上都没有标记，有一个泵可以用来往水壶中加水。如何做能在 4 加仑的水壶中恰好得到两加仑的水？

10. 总结上题，使得你的解决方案的参数包括每个水壶的大小，和最后大壶中所剩的水量。

11. 写一个程序，解决以下问题：三个传教士和三个食人族来到河边，河边只有一艘船，每次可以载两个人，每个人都必须要过河来继续旅程。然而，如果河岸上的食人族人数超过传教士人数，传教士将被吃掉。找出每次的过河方案使每个人都安全地到达河的另一边。

12. 用海龟绘图模块，将汉诺塔解决过程做成动画。提示：你可以改变海龟的形状，可以将海龟 shape 改为方块的盘子。

13. Pascal 三角形是一个由数字组成的三角形，这些数字以如下方式在每一行交错排列：

$$a_{nr} = \frac{n!}{r!(n-r)!}$$

这个方程是一个二项式系数的方程。可以通过添加数字来建立 Pascal 三角形，每行的数字都是由上一行的对角线数字相加而得。Pascal 三角形举例如下：

```
        1
      1   1
    1   2   1
  1   3   3   1
1   4   6   4   1
```

写一个程序输出 Pascal 三角形。程序里应接受一个参数来定义三角形的行数。

14. 假设你是一个计算机科学家或是一个艺术小偷，闯入了一个艺术画廊。你身上只有一个背包可以用来偷出宝物，这个背包只能装 W 英镑的艺术品，但你知道每一件艺术品的价值和它的重量。运用动态规划写一个函数，来帮助你获得最多价值的宝物。你可以利用下面的例子来编写程序：假设你的背包可以容纳的总重量为 20，你有如下 5 件宝物：

item	weight	value
1	2	3
2	3	4
3	4	8
4	5	8
5	9	10

15. 这个问题叫做单词最小编辑距离问题，在很多领域的研究中起到了很大作用。假设你想把单词“algorithm”变为“alligator”。对于每一个字母，你有三种变换方式：从源单词复制一个字母到目标单词，计 5 分；从源单词删除

一个字母，计 20 分；在目标单词插入一个字母，计 20 分。最后将一个单词转换为另一个的分数可以被拼写检查系统使用，用来给彼此相似的单词提供建议。使用动态规划技术，写一种算法得到任何两个单词之间的最小编辑距离。

5. 排序与搜索

5.1. 目标

- 了解和实现顺序搜索和二分法搜索；
- 了解和实现选择排序、冒泡排序、归并排序、快速排序、插入排序和希尔排序；
- 了解用散列 Hashing 实现搜索的技术；
- 了解抽象数据类型：映射 Map；
- 采用散列实现抽象数据类型 Map。

5.2. 搜索

5.2.1. 顺序搜索

当数据项被存储在集合,如一个列表,我们说,他们有一个线性或顺序的关系。每一个数据项存储在一个与其他数据项相对的位置。在 Python 列表,这些相对位置是单个项的索引值。由于这些索引值是有一定次序的,可以依次访问它们。这一过程产生第一个搜索方法,顺序搜索。

图 1 显示了这个搜索是如何工作的。从列表中的第一项开始,我们只是从一项移到另一项,按照索引值,直到我们发现正在寻找的数据项或者遍历所有数据项。如果我们遍历所有数据项,我们会发现,我们正在寻找的数据项是不存在。

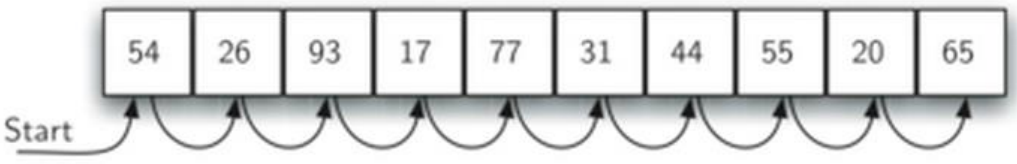


图 1：整数列表的顺序搜索

这个算法的 Python 实现如 CodeLens 1 所示。函数的参数为一个列表和一个我们正在寻找的数据项，并且返回一个布尔值,判断它是否存在。布尔变量初始化为 False，如果我们发现列表中的我们所要找的数据项，就将布尔变量赋为 True。

```
1  def sequentialSearch(alist, item):
2      pos = 0
3      found = False
4
5      while pos < len(alist) and not found:
6          if alist[pos] == item:
7              found = True
8          else:
9              pos = pos+1
10
11     return found
12
13 testlist = [1, 2, 32, 8, 17, 19, 42, 13, 0]
14 print(sequentialSearch(testlist, 3))
15 print(sequentialSearch(testlist, 13))
```

一个无序表的顺序排序

5.2.1.1. 排序算法的分析

分析搜索算法,我们需要确定一个基本单元的计算。回想一下,为了解决这一问题,这些简单的步骤必须重复。对于搜索搜索,计算比对执行次数是有必要的。每次对比都有可能发现或不发现我们正在寻找的数据项。此外,我们再作一个猜想。数据列表不是有序的。数据项是被随机放置到列表中。换句话说,我们寻找的目标项可能在任意位置，对列表的每一个位置，我们找到它的概率是相等的。

如果目标项不在列表内，唯一的办法就是比对现有的每一个数据项。如果这里有 n 个数据项，那么顺序搜索就要求 n 次比对去发现目标项不在列表里面。在列表包含目标项的情况下，算法分析就不是这么的直接。这儿通常有三种不同的情况发生。最好的情况就是我们要找的数据项就在我们第一次比对的位置(列表的开头)。我们仅仅需要一次比对。最坏的情况是直到最后一次比对（第 n 次比对），我们才能发现目标项。

平均情况怎么样呢？在平均情况下，我们将在半路上发现目标项，就是说，我们需要比对 n/2 的数据项。回忆,然而,随着 n 较大,系数不管它们是多么大,成为无关紧要的近似,所以顺序搜索的复杂性是 O(n)。表 1 总结了这些结果。

表 1：无序表顺序搜索的比对

Case	Best Case	Worst Case	Average Case
item is present	1	n	n/2
item is not present	n	n	n

我们早先假定我们集合里面的数据项都是被打乱放入的，以便数据之间没有相对关系。如果在某些情况下，数据项是有序的，顺序搜索又会发生什么？我们能让我们的搜索技术更高效吗？

假定列表是按照一个递增的顺序构建的(从小到大)。如果我们要找的数据项在列表里面，它在 n 个位置的机会如以前一样是相等的。我们将依然经过相同的比较次序去发现目标项。然而，如果数据项不在列表里面，将会有一个明显的优势。图 2 就展示了搜索 50 这个数据项的算法流程。注意在到 54 之前一直是执行比对操作的。然而，在这一点上，我们知道了一些另外的东西。因为这个列表是被排好序的，所以在 54 之前找不到，那么比 54 大的数据项肯定也不能满足条件。在这种情况下，算法并没有要求继续去遍历所有数据项来表明目标项没有被找到。它可以立刻停止。CodeLens 2 展示了顺序搜索函数的变量。

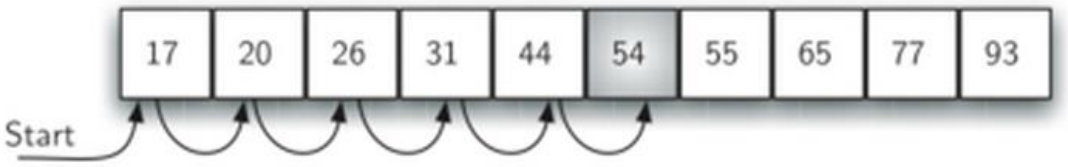


图 2：一个整数有序表的顺序搜索

```
1  def orderedSequentialSearch(alist, item):
2      pos = 0
3      found = False
4      stop = False
5      while pos < len(alist) and not found and not stop:
6          if alist[pos] == item:
7              found = True
8          else:
9              if alist[pos] > item:
10                 stop = True
11             else:
12                 pos = pos+1
13
14     return found
15
16 testlist = [0, 1, 2, 8, 13, 17, 19, 32, 42,]
17 print(orderedSequentialSearch(testlist, 3))
```

```
18  print(orderedSequentialSearch(testlist, 13))
```

一个有序列表的顺序搜索

表 2 总结了这些结果。注意我们可以发现项不在列表的最好结果只需要看一个数据项。平均一下，通过遍历 n/2 的数据项，我们将知道目标项是否在列表中。
表 2：有序列表顺序搜索的比较

Case	Best Case	Worst Case	Average Case
item is present	1	n	n/2
item not present	1	n	n/2

自我检测
Q-29：假定你在做列表[15, 18, 2, 19, 18, 0, 8, 14, 19, 14]的一个顺序搜索，为了找到 18，你需要做多少次比对？
a) 5
b) 10
c) 4
d) 2

Q-30:假定你在做有序列表[3, 5, 6, 8, 11, 12, 14, 15, 17, 18]的一个顺序搜索，为了找到 13，你需要做多少次比对？

a) 10
b) 5
c) 7
d) 6

5.2.2.二分法搜索

更好地利用有序表的优势是可能的，如果我们的比较方法更聪明一些。在顺序搜索中，当我们和第一项相比较时，如果第一个数据项不是我们要找的项，最多还有 n-1 项待比对。二分搜索将从中间项开始检测，而不是按顺序搜索列表。如果查找项与我们刚搜索到的项匹配，则搜索结束。如果不匹配，我们可以利用列表的有序性来排除掉一半的剩余项。如果查找项比中间项大，我们知道列表中较小的那一半全部和中间项可以从接下来的考察中排除了。如果查找项在列表中，那它一定在较大的那一半。

接下来，我们可以在较大的一半中重复这个过程。从中间项开始，拿它和查找项作比较。再来一次，我们或者找到查找项或者从中间分割列表，并因此排除掉另一大部分我们的可能搜索区域。图解 3 展示了这种算法如何能快速找到值为 54 的项。完整的过程在代码信息指示器 3 中被展示。

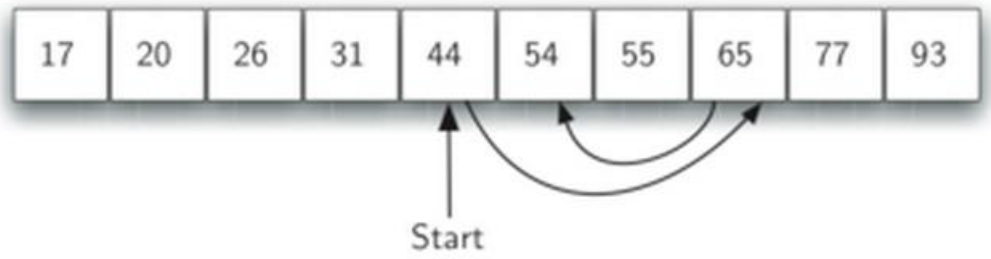


图 3：二分搜索一个整数有序表

```
1   def binarySearch(alist, item):
2       first = 0
3       last = len(alist)-1
4       found = False
5
6       while first<=last and not found:
7           midpoint = (first + last)//2
8           if alist[midpoint] == item:
9               found = True
10          else:
11              if item < alist[midpoint]:
12                  last = midpoint-1
13              else:
14                  first = midpoint+1
15
16      return found
17
18  testlist = [0, 1, 2, 8, 13, 17, 19, 32, 42,]
19  print(binarySearch(testlist, 3))
20  print(binarySearch(testlist, 13))
```

二分搜索一个有序表（搜索 3）

在我们继续分析之前，我们应该指出这个算法是分而治之策略的一个很好的例子。分而治之意味着我们把一个问题分成更小的规模，用一些方法解决这些更小规模问题，然后重组整个问题来得到结果。当我们对一个列表执行二分搜索时，我们首先选择中间项。如果搜索项比中间项小，我们可以在原来列表的左半部分执行二分搜索，同样地，如果搜索项更大，我们可以在右半部分执行二分搜索。无论怎样，这是一个对较小的列表二分搜索实现的递归调用。代码指示器 4 展示了这种递归版本。

```
1   def binarySearch(alist, item):
2       if len(alist) == 0:
```

```
3          return False
4      else:
5          midpoint = len(alist)//2
6          if alist[midpoint]==item:
7            return True
8          else:
9            if item<alist[midpoint]:
10             return binarySearch(alist[:midpoint],item)
11           else:
12             return binarySearch(alist[midpoint+1:],item)
13
14 testlist = [0, 1, 2, 8, 13, 17, 19, 32, 42,]
15 print(binarySearch(testlist, 3))
16 print(binarySearch(testlist, 13))
```

二分搜索——递归版本（搜索 4）

5.2.2.1. 二分搜索分析

为了分析二分搜索算法，我们需要回顾每一个从考察中对比、排除约一半的待搜索项。为了检查整个列表，这种算法所需最多的比对次数是多少呢？如果我们开始有 n 项，大约有 n/2 项将在第一次比对之后被遗弃。在第二次比对之后，将会有约 n/4，然后是 n/8,n/16,等等。我们能分割列表多少次呢？表 3 帮我们看到答案。

表 3：二分搜索的表格式分析

比对	剩余项的大概数量
1	n/2
2	n/4
3	n/8
…	…
i	n/2^i

当我们分割足够次时，我们结束于一个只有一项的列表，无论它是不是我们要找的那一项，总之，我们完成了。必需比对次数通过解方程 n/2^i=1 得到。解得 i=log（n）。最大比对次数是关于列表中项数的对数。因此，二分搜索是 O（log（n））。

一个附加的分析消耗需要被处理。在上面的递归处理演示中，递归调用

```
binarySearch(alist[:midpoint],item)
```

使用了切片操作符来创建左半部分列表，然后传递到下一个调用（右半部分

也一样）。我们上面所做的分析假定切片操作消耗的是常数时间。然而，我们知道 Python 的切片操作实际上是 O（k）。这意味着二分搜索使用切片将不会运行严格的对数时间。幸运的是这可以通过列表，以及开始和结束索引补救。这个索引可以向我们在列表 3 中精心设计的那样。我们将这个实现留作练习。

即使二分搜索通常比顺序搜索要好，值得注意的是对于较小的 n 值，排序所附加的消耗可能是不值得的。事实上，我们应当一直考虑进行额外的排序工作来得到搜索优势是否是有效开销。如果我们可以排序一次然后搜索许多次，排序开销并不那么显著。然而，对大列表，甚至一次排序消耗也可如此巨大，从一开始简单执行顺序搜索也许是最好的选择。

自我检测

Q-19：假如你有以下已排好序的列表[3,5,6,8,11,12,14,15,17,18]而且用二分搜索算法。哪组数正确地展示了用来找到关键数 8 的比对次序？

a) 11,5,6,8
b) 12,6,11,8
c) 3,5,6,8
d) 18,12,6,8

Q-20：假如你有以下已排好序的列表[3,5,6,8,11,12,14,15,17,18]而且用二分搜索算法。哪组数正确地展示了用来搜索关键数 16 的比对次序？

a) 11,14,17
b) 18,17,15
c) 14,17,15
d) 12,17,15

5.2.3. 散列

在前面的部分我们利用数据之间的相互关系来实现数据集中，从而提高我们搜索算法的效率。例如，给定一个有序列表，我们就可以用二分法进行搜索，在一个对数级别的时间复杂度上。在这一章节我们进一步去构造一个新的数据结构，能够实现搜索时间复杂度为 O(1)，这个概念叫做散列。

为了实现这个，在进行寻找的时候我们甚至要知道数据项所处的位置。如果我们事先知道数据项应该出现在哪个位置，那我们就可以到那个位置去看看数据项是不是存在。但是，通常并不是这么简单的。

散列表是一种数据集，存储数据的方式有利于后续的查找。散列表的每一个位置叫做槽，能够存放一个数据项，并且从 0 开始命名。例如，第一个槽为 0，下一个为 1，下一个为 2，以此类推。初始条件下，散列表中没有数据项，所以槽是空的。我们可以初始化散列表的每一个元素都是 None。图 4 展示了一个长

度为 m=11 的散列表，也就是有 m 个槽位，命名从 0 到 10。

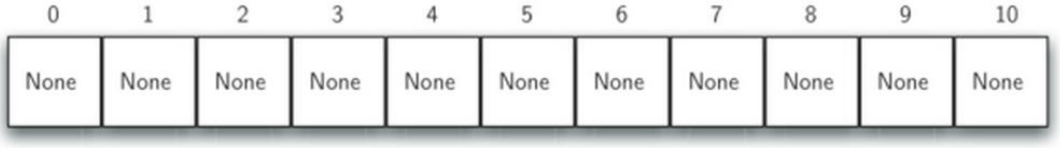


图 4：有 11 个空槽的散列表

实现从数据项到存储槽的方法叫做散列函数。散列函数可以将任何一个数据项放到存储槽中并且返回一个槽的名字的整数，从 0 到 m-1。假设我们有包含 54、26、93、17、77 和 31 的整数集合，第一个散列函数，有时被称作为求余，将数据项与散列表的大小相除，返回余数作为散列值。表 4 给出了所有的数据项的散列值。记录所有的余数，并且将结果当做槽的名称。

表 4：求余散列函数

Item	Hash Value
54	10
26	4
93	5
17	6
77	0
31	9

一旦散列值求了出来，我们就可以将每一个数据项插入到散列表中相应的槽位，例如图 5。11 个槽位中占据 6 个。槽位被数据项占据的比例叫做负载因子 λ=数据项数目/表大小，这里的 λ=6/11。

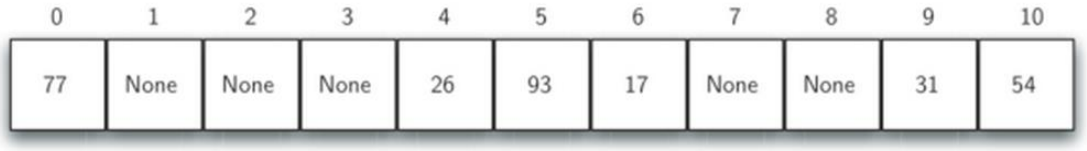


图 5：6 个数据项的散列表

现在，当我们想要寻找一个数据项，我们只需要使用散列函数去计算槽的名称并且检查散列表中是不是存在。这种寻找操作的时间复杂度为 O(1)，实现了时间复杂度为常数，并且找到了数据项的位置。

但是，你可能也看出了该方法的问题所在，就是每一个数据是占据特有的槽位。例如，我们在散列表中插入 44，它的散列值为 0（44%11=0）。但是 77 的散列值也是 0，那么问题就来了。根据这种求余的散列函数，两个或更多的数据就需要在同一个槽位，这就产生了冲突。显然，冲突造成了散列函数的问题，我们在后面的章节进行讨论。

5.2.3.1. 散列函数

给定一组数据项，如果散列函数可以将每一个数据项都映射到不同的槽位中，这样就叫做完美散列函数。如果数据项是固定的，我们就可以创造一个完美散列函数。但是，如果给定的是任意的一组数据项，这就没有一种固定的方法去创建一个完美散列函数。幸运的是，我们不需要一个散列函数既完美又高效的收集数据。

一种实现完美散列函数的方法就是扩大散列表的尺寸，以至于每一个数据项都可以占据不同的槽位。但是这种方法对数据过大而数据量过少的数据就不适合。例如，如果数据是九位数的社会安全码，这种方法就需要差不多 10 亿个槽位，如果我们只想保存班里 25 个同学的数据，我们就会浪费大量的存储空间。

我们的目标是创建一个散列函数去将冲突的数量降到最小，方便去计算并且最终将数据项分配到散列表中。这里有几种普通的方法去扩展求余方法，我们将在这里考虑几个的可能性。

折叠法创建散列函数的基本步骤是，首先将数据分成相同长短的片段（最后一个可能不一样），这些片段再相加，求余得到其散列值。例如，如果我们有一串电话号码 436-555-4601，我们可以两个一组分为 5 段（43 65 55 46 01），然后相加 43+65+55+46+01，我们得到 210,。如果我们假设散列表有 11 个槽位，我们需要将和进行求余。210%11 得到 1，所以电话号码 436-555-4601 就散列到槽位 1。有的折叠法还包含一个隔数反转再相加。比如，我们得到 43+56+55+64+01=219，求余得到 10。

另一种创建散列函数的数值方法叫做平方取中法。我们首先将数据取平方，然后取平方数的某一部分。例如，数据项是 44，我们首先计算 44^2=1936，然后取出中间的两位数 93，然后再进行求余计算，我们得到 5(93%11)。表 5 展示了数据项在求余法和平方取中法的结果。你应该却分并明白它们是如何计算出来的。

表 5：求余法和平方取中法的比较

Item	Remainder	Mid-Square
54	10	3
26	4	7
93	5	9
17	6	8
77	0	4
31	9	6

我们也可以为非数字的数据项例如字符串创建散列表。‘cat’可以看做一个连续的 ASCII 数值。

```
>>> ord('c')
99
>>> ord('a')
97
```

```
>>> ord('t')
116
```

然后我们可以将三个数值加起来，接着用求余法得到一个散列值（如图 6）。表 1 展示了一个散列函数，用来产生一个字符串和一个表的大小并且返回一个 0 到表大小-1 的散列值。

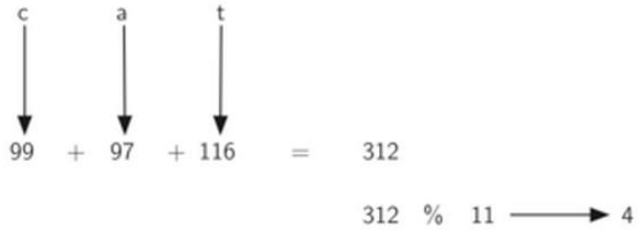


图 6：用 ASCII 数值散列一个字符串

表 1

```
def hash(astring, tablesize):
    sum = 0
    for pos in range(len(astring)):
        sum = sum + ord(astring[pos])

    return sum%tablesize
```

当我们用散列函数记录散列值的时候，点到字母会得到相同的散列值。为了纠正这个，我们可以将字母的位置作为权重因子。图 7 展示了一种可能利用权重因子的方式。调整后的散列函数在左侧作为练习。

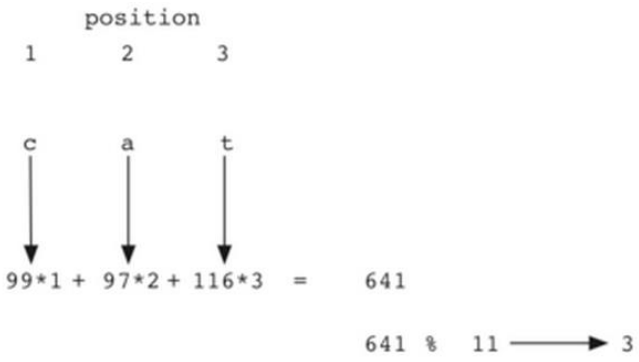


图 7：用权重因子散列字符串

你们还可以想出一些其他的方法去计算数据的散列值，但是最重要的事情就是散列函数必须高效以至于它不能成为存储空间和搜索进程的主要部分。如果散列函数过于复杂，去花费大量的工作去计算槽位的名称，可能还不如进行简单的顺序搜索或者二分法搜索，这就失去了散列的意义。

5.2.3.2. 冲突解决方法

我们现在回到冲突的问题。当两个数据散列到相同的槽位，我们必须用一种系统的方法将第二个数据放到散列表里。这个过程叫做冲突解决。正如我们前面提到的，如果散列函数是完美的，冲突不会出现。但是，这一般是不可能的，冲突解决成为散列很重要的一部分。

一种解决冲突的方法就是扫描散列表并且寻找另一个空的槽位来存放这个有冲突的数据。一种简单的方法就是我们从发生冲突的位置开始寻找，顺序向下寻找知道我们找到第一个空的槽位。我们可能需呀回到第一个槽位以实现覆盖整个散列表。这种寻找空槽的方法叫做开放地址，系统地向后寻找每一个槽位，我们将这种开放地址的技术叫做线性探测。

图 8 展示了一个扩展的数据集合通过简单的求余散列函数得到的结果（54、26、93、17、77、31、44、55、20）。上面的表 4 展示了原始数据的散列值。图 5 展示了原始内容。当我们尝试把 44 放到槽位 0 中的时候，冲突出现了。通过线性探测，我们一个一个槽位的检查，指导我们找到一个空槽为。因此，我们找到了槽位 1,。

同样的,55 本应该在槽位 0 当时必须放到槽位 2,因为它是接下来的空槽位。最后一个是 20,应该散列到槽位 9,但是槽位 9 是满的,我们开始进行线性探测，我们查看槽位 10,0,1 和 2，并且最后我们找到了空的槽位 3。

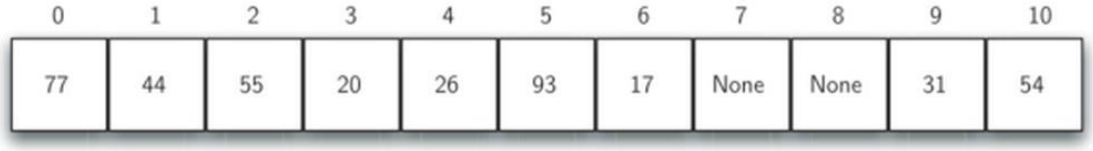


图 8：线性探测解决冲突

一旦我们用开放地址和线性探测建立一个散列表，很重要的就是我们要利用相同的方法寻找数据。假设我们寻找数据 93，当我们计算其散列值，我们得到 5，查看槽位 5，是 93，所以我们可以返回真。那如果我们寻找 20 呢？现在散列值是 9，但是槽位 9 现在存放的是 31.我们不能简单的返回假，因为我们知道这可能存在冲突。现在我们需要紧接着进行搜索，从位置 10 开始，直到我们找到数据 20 或者找到一个空的槽位。

线性探测法的一个缺点是聚类的趋势：数据会在表中聚集。这意味着如果对于同一散列值产生了许多冲突时，周边的一系列槽将会被填充。这将会对正在被新填入的数据产生影响，就像我们看到的，将数字 20 加入表中一样。一簇值散列到 0 时必须跳过原槽直到找到新的空槽。如图 9 所示。

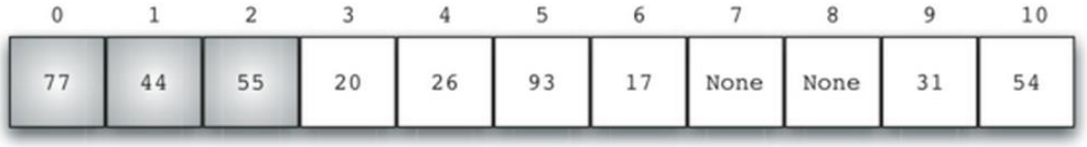


图 9： 一簇对应槽 0 的数据

一种处理方法是扩展线性探测的技术，不再是按顺序一个一个寻找新槽，我们跳过一些槽，这样能更加平均地分配出现冲突的数据。这样做可以减少聚类的发生。图 10 显示了同样的数字如何通过遇到冲突时“+3”的方法被填入表中的。这表示一旦一个冲突出现，我们将每次跳过两个槽寻找下一个新的空槽。

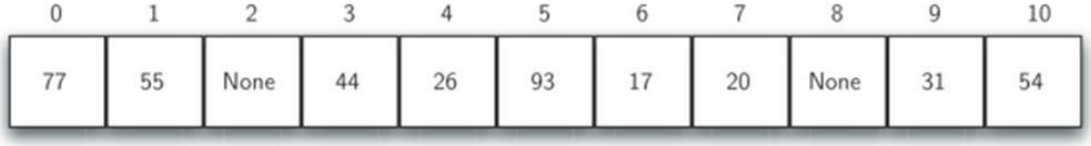


图 10：用“+3”发解决冲突

这种在冲突时为数据寻找新槽的进程被称作再散列。有着简单的线性探测的功能的再散列函数语言是 newhashvalue = rehash(oldhashvalue) where rehash(pos) = (pos+1)%sizeoftable。“+3”rehash 法则写作 rehash(pos)=(pos+3)%sizeoftable。归纳起来的语言是 rehash(pos)=(pos+skip) %sizeoftable。重要的一点是，选择跳过槽的数值必须能保证所有的槽都能被访问过。否则，有些槽将会被闲置。为了保证这一点，一个经常的建议是将槽的数量设置为质数，这也是我们在例子里设置 11 个槽的原因。

另一种线性探测方法叫做二次探测法。与其每次在冲突时选择跳过一个常量的槽，我们每次跳过的槽的数量将会依次增加 1,3,5,7,9,等等。这意味着，如果原槽为第 h 个，那么再散列时访问的槽为第 h+1, h+4, h+9, h+16 个，等等。换句话说，二次探测法使用一个连续的平方数组成的跳跃值。图 11 显示了我们的例子在运用二次探测法时的填充结果。

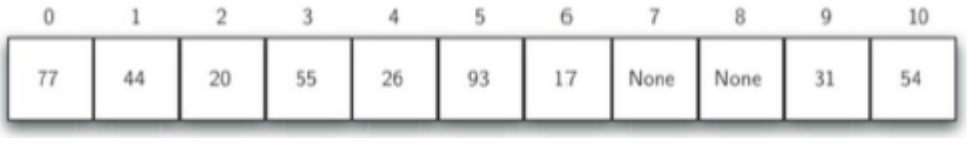


图 11：用二次探测法解决冲突

另一个替代方法是允许每一个槽都能填充一串而不是一个数据（称作链）。链能允许多个数据填在散列表中的同一个位置上。当冲突发生时，数据还是填在应有的槽中。随着一个槽中填入的数据的增多，搜索的难度也随之增加。图 12 显示了数据在用链法时填入散列表的结果。

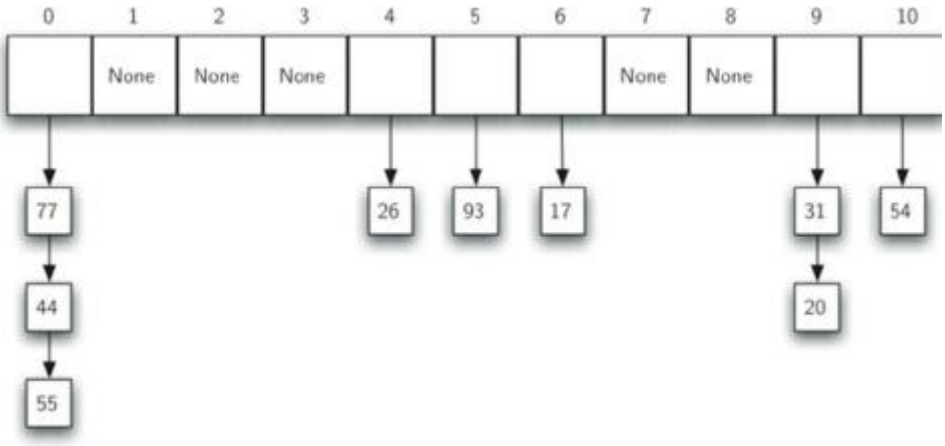


图 12：数据链法解决冲突

当我们希望搜索一个数据时，我们用散列函数去找出数据所在的槽。因为每个槽都有一个集合，我们运用搜索技术去确定数据是否在槽中。这种方法的优势是平均来说每个槽中的数据都相对很少，因此搜索会非常便捷。我们将会在章末讨论对于散列法的分析。

自我检测

Q-17:在一个散列表中有 13 个槽，27 和 130 将会被填入第几个槽中？

a) 1, 10
b) 13, 0
c) 1, 0
d) 2, 3

Q-18:假设你将下列 11 个数根据线性探测方法填入散列表中，哪一个选项最好地表达了填入之后散列表的状况？113，117，97，100，114，108，116，105，99

a) 100, __, __, 113, 114, 105, 116, 117, 97, 108, 99
b) 99, 100, __, 113, 114, __, 116, 117, 105, 97, 108
c) 100, 113, 117, 97, 14, 108, 116, 105, 99, __, __
d) 117, 114, 108, 116, 105, 99, __, __, 97, 100, 113

5.2.3.3. 实现图的抽象数据类型

一个最有用的 python 数据结构是字典。字典是一个关联的数据类型，可以存储密钥－数据对。密钥用来关联数据值。我们通常把这个想法比作图。

图的抽象数据类型定义如下。结构是单个密钥和数据值的关联的无序集合。图中的密钥都是独特的，以保证和数据值的一一对应。相关操作如下。

- Map() 产生一个新的空图，返回一个空的图集合。
- Put(key,val) 往图中添加一个新的密钥-数据值对，如果密钥以及存在，那么将旧的数据值置换为新的。
- get(key) 给定一个 key 值，返回关联的数据值或者 None。
- del 从图中删除密钥-数据值对，声明形式为 del map[key]
- len() 返回图中的某个密钥-数据值对的长度
- in 返回 True，当声明的 key 在图中，否则返回 False

字典的一个巨大的好处是给定关键码时，我们能迅速找到数据值。为了使得这种快速寻找方法得以实现，我们需要一种有效搜索的方法。我们可以使用列表顺序或二进制搜索但是更好的方法是利用散列表，因为后者可以实现 O(1)复杂度。

在表 2 我们运用 2 个列表来创造一个 HashTable 类来实现图的数据结构类型。其中一个，称为 slots，用来储关键码，另一个列表称为 data，用来储存数据值。当我们查找一个关键码时，对应的数据值列表中的位置保存着对应的数据值。我们将密钥列表当做一个散列表，正如前面所言。注意散列表的槽数为 11。虽然对于槽数的选择是任意的，但是将其确定为一个质数还是很有意义的，这样能够使得解决冲突的程序发挥最大威力。

表 2

```
class HashTable:
    def __init__(self):
        self.size = 11
        self.slots = [None] * self.size
        self.data = [None] * self.size
```

散列函数实现了其他的简单方法。这里的冲突解决技术是运用“+1”的线性探测法。put 函数（见表 3）假设最终一定能找到一个能让新的关键码填入的槽。它计算原散列值，当发现对应的槽不为空时，启动再散列函数直到空槽找到为止。如果一个非空的槽已经含有该密钥，那么就替换其数据值为当前数据值

表 3

```
def put(self,key,data):
  hashvalue = self.hashfunction(key,len(self.slots))

  if self.slots[hashvalue] == None:
    self.slots[hashvalue] = key
    self.data[hashvalue] = data
  else:
    if self.slots[hashvalue] == key:
      self.data[hashvalue] = data  #replace
    else:
      nextslot = self.rehash(hashvalue,len(self.slots))
```

```
     while self.slots[nextslot] != None and \
                    self.slots[nextslot] != key:
       nextslot = self.rehash(nextslot,len(self.slots))

     if self.slots[nextslot] == None:
       self.slots[nextslot]=key
       self.data[nextslot]=data
     else:
       self.data[nextslot] = data #replace

def hashfunction(self,key,size):
     return key%size

def rehash(self,oldhash,size):
    return (oldhash+1)%size
```

相似的，get 函数（见表 4）也是以计算原散列值开始。如果结果不在对应的槽中，再散列就会被用来定位到下一个可能的位置。注意第 15 行确保我们没有再次回到原槽，保证了搜索操作不会陷入死循环。如果这种情况发生了，那么我们会穷尽所有的可能的槽同时依然不会搜索到所需搜索的数据。

散列表类的最后的一些操作提供了额外的字典类功能。我们超载__getitem__和__setitem__方法允许访问使用“[]”。这表示一旦一个散列表被建立，熟悉的的索引操作符将是可用的。我们把剩下的方法留作练习。

表 4

```
def get(self,key):
  startslot = self.hashfunction(key,len(self.slots))

  data = None
  stop = False
  found = False
  position = startslot
  while self.slots[position] != None and  \
                      not found and not stop:
     if self.slots[position] == key:
       found = True
       data = self.data[position]
     else:
       position=self.rehash(position,len(self.slots))
       if position == startslot:
           stop = True
  return data
```

```
def __getitem__(self,key):
    return self.get(key)

def __setitem__(self,key,data):
    self.put(key,data)
```

下面的部分显示了散列表类的操作。首先我们建立一个散列表并保存一些包含整数关键码和字符串数据值的数据。

```
>>> H=HashTable()
>>> H[54]="cat"
>>> H[26]="dog"
>>> H[93]="lion"
>>> H[17]="tiger"
>>> H[77]="bird"
>>> H[31]="cow"
>>> H[44]="goat"
>>> H[55]="pig"
>>> H[20]="chicken"
>>> H.slots
[77, 44, 55, 20, 26, 93, 17, None, None, 31, 54]
>>> H.data
['bird', 'goat', 'pig', 'chicken', 'dog', 'lion',
      'tiger', None, None, 'cow', 'cat']
```

接下来我们将访问并修改一些数据。注意关键码 20 对应的数据值

```
>>> H[20]
'chicken'
>>> H[17]
'tiger'
>>> H[20]='duck'
>>> H[20]
'duck'
>>> H.data
['bird', 'goat', 'pig', 'duck', 'dog', 'lion',
      'tiger', None, None, 'cow', 'cat']
>> print(H[99])
None
```

完整的散列表的例子见 ActiveCode 1。

完成散列表的例子（hashtablecomplete）

5.2.3.4.分析散列

我们前面提到的,在最好的情况下散列表将提供一个 O(1)的时间复杂度,确定时间的搜索技术。然而,由于散列冲突,比较的次数通常没有这么简单。尽管散列的完整分析超出了本文的范围,我们可以提出一些已经被很好知道结果---为了搜索一个数据而大约必须的比对次数。

我们需要分析散列表使用情况最重要的一条信息是负载因子λ。从概念上讲,如果小λ,然后是冲突的几率较低,这意味着数据项更有可能在被填充。如果λ很大,这意味着表被填满了,然后越来越多的冲突。这意味着碰撞解决更加困难,需要更多的比对找到一个空槽。通过数据项链,增加的碰撞机会意味着在每个链上增加数据项的数目。

和之前一样,我们将会有一个成功的和不成功的搜索结果。对于一个成功的搜索使用开放地质寻址与线性探测,比对的平均数量级大约是 12(1 + 1/1-λ)和一个不成功的搜索是 12(1 +(1/1-λ)^ 2)如果我们使用数据项链，成功的例子比对的平均数量级是 1 +λ/ 2，如果比对失败就是λ量级的。

5.3.排序

5.3.1.冒泡排序

冒泡排序要对一个列表多次重复遍历。它要比较相邻的两项，交换顺序排错的项。每对列表实行一次遍历，就有一个最大项排在了正确的位置。大体上讲，每一个数据项都在它该在的位置“冒泡”。

图 1 展示了冒泡排序对列表的第一次遍历。这个过程中要比较阴影的两个数据项以确定它们的顺序是否正确。如果列表有 n 项，第一次遍历就要比较 n-1 对数据。需要注意，一旦列表中最大（按照规定的原则定义大小）的数据是所比较的数据对中的一个，它就会沿着列表一直后移，直到这次遍历结束。

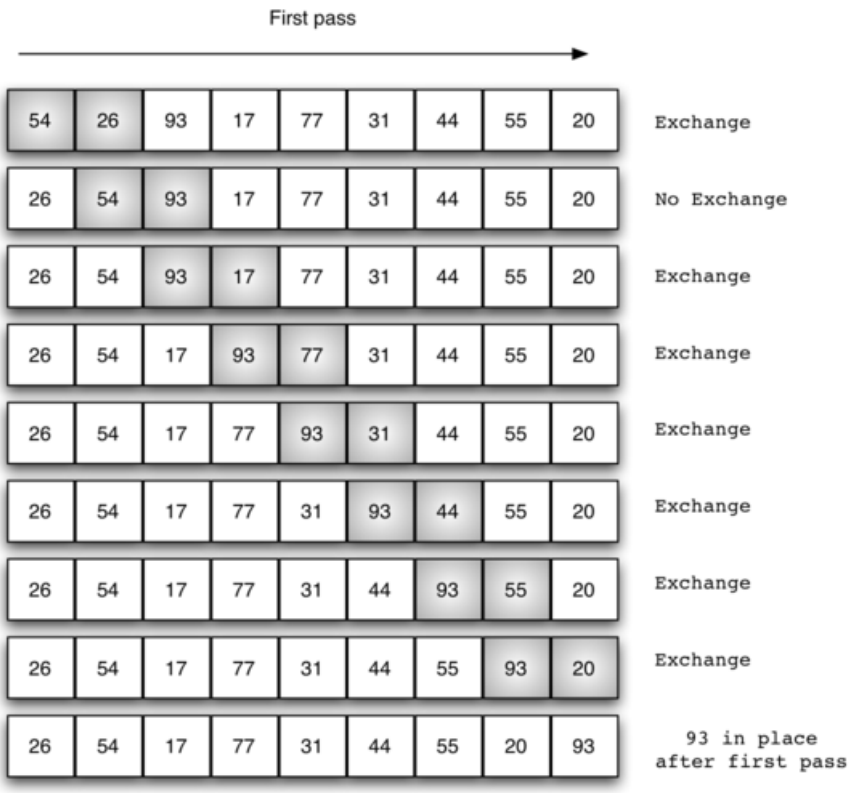


图 1 冒泡排序：第一次遍历

第二次遍历开始时，最大的数据项已经归位。现在还剩 n-1 个待排数据，即有 n-2 个要比较的数据对。由于每一次遍历都会使下一个最大项归位，所需要遍历的总次数就是 n-1。完成 n-1 次遍历之后，最小的数据项一定已经归位，不需要再执行其他步骤。代码 1 是完成冒泡排序的函数。它以待排列表为参数，通过交换必要的数据项实现对列表的修改，完成排序。

```
temp = alist[i]
alist[i] = alist[j]
alist[j] = temp
```

Python 中的翻译操作与其他许多编程语言不同。在典型的操作方法中，交换列表中的两项需要一个暂存位置（一个附加的储存空间）。以上这段代码可以交换列表第 i 项和第 j 项。如果没有暂存位置，其中一个值就会被覆盖。

但是在 Python 里，可以“同时赋值”。通过“a,b=b,a”的语句就可以让两个赋值语句同时进行（如图 2）。用同时赋值的方法可以用一条语句实现交换操作。

代码 1 中 5-7 行交换第 i 项和第 i+1 项用了前述的三行代码，其实我们也可以用同时赋值来交换数据。

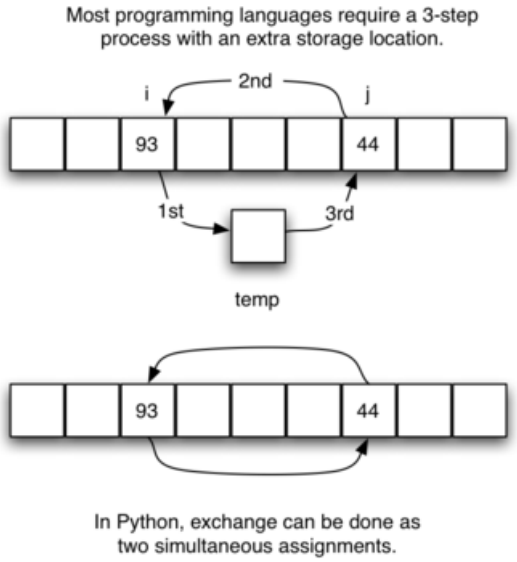


图 2 Python 中的值交换

下面的代码展示了实现冒泡排序的完整函数。

```
def bubbleSort(alist):
    for passnum in range(len(alist)-1,0,-1):
        for i in range(passnum):
            if alist[i]>alist[i+1]:
                temp = alist[i]
                alist[i] = alist[i+1]
                alist[i+1] = temp

alist = [54,26,93,17,77,31,44,55,20]
bubbleSort(alist)
print(alist)
```

代码 1 冒泡排序

下面我们来分析冒泡排序。注意到，不考虑初始列表中的数据如何排列，排一个长度为 n 的列表就要 n-1 次遍历。表 1 显示了每次遍历需要比较的次数。总的比较次数是前 n-1 个正整数的和，即 $1/2(n^2-n)$。比较复杂度为 $O(n^2)$。在最好的情况下，如果列表已经排好序，就不需交换，但是在最坏的情况下，每一次比较都要进行一次交换，平均下来，我们交换操作次数是比较次数的一半。

表格 1 冒泡排序每次遍历的比较次数

遍历次数	比较次数
1	n-1
2	n-2
3	n-3
...	...

n-1	1

因为冒泡排序必须要在最终位置找到之前不断交换数据项，所以它经常被认为是最低效的排序方法。这些“浪费式”的交换操作消耗了许多时间。但是，由于冒泡排序要遍历整个未排好的部分，它可以做一些大多数排序方法做不到的事。尤其是如果在整个排序过程中没有交换，我们就可断定列表已经排好，因此可改良冒泡排序，使其在已知列表排好的情况下提前结束。这就是说，如果一个列表只需要几次遍历就可排好，冒泡排序就占有优势：它可以在发现列表已排好时结束。代码 2 就是改良版冒泡排序。它通常被称作“短路冒泡排序”

```
def shortBubbleSort(alist):
    exchanges = True
    passnum = len(alist)-1
    while passnum > 0 and exchanges:
       exchanges = False
       for i in range(passnum):
           if alist[i]>alist[i+1]:
               exchanges = True
               temp = alist[i]
               alist[i] = alist[i+1]
               alist[i+1] = temp
       passnum = passnum-1

alist=[20,30,40,90,50,60,70,80,100,110]
shortBubbleSort(alist)
print(alist)
```

代码 2 短路冒泡排序

自我检测

Q-21: 给定排序列表[19, 1, 9, 7, 3, 10, 13, 15, 8, 12] ，如下哪个列表是冒泡排序第三次遍历之后的列表？

a) [1, 9, 19, 7, 3, 10, 13, 15, 8, 12]
b) [1, 3, 7, 9, 10, 8, 12, 13, 15, 19]
c) [1, 7, 3, 9, 10, 13, 8, 12, 15, 19]
d) [1, 9, 19, 7, 3, 10, 13, 15, 8, 12]

5.3.2. 选择排序

选择排序提高了冒泡排序的性能，它每遍历一次列表只交换一次数据，即进行一次遍历时找到最大的项，完成遍历后，再把它换到正确的位置。和冒泡排序

一样，第一次遍历后，最大的数据项就已归位，第二次遍历使次大项归位。这个过程持续进行，一共需要 n-1 次遍历来排好 n 个数据，因为最后一个数据必须在第 n-1 次遍历之后才能归位。

图 3 展示了选择排序的整个过程。每一次遍历，最大的数据项被选中，随后排到正确位置。第一次遍历排好了 93，第二次排好了 77，第三次排好了 55，以此类推。代码 1 展示了实现选择排序的函数。

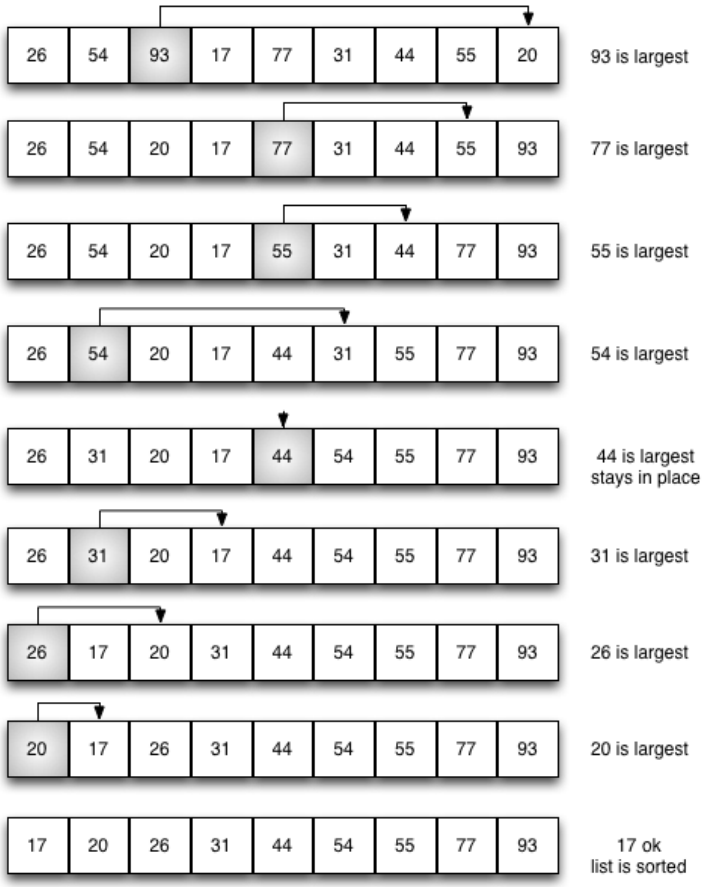


图 3 选择排序

```
def selectionSort(alist):
    for fillslot in range(len(alist)-1,0,-1):
        positionOfMax=0
        for location in range(1,fillslot+1):
            if alist[location]>alist[positionOfMax]:
                positionOfMax = location

        temp = alist[fillslot]
        alist[fillslot] = alist[positionOfMax]
        alist[positionOfMax] = temp
```

```
alist = [54,26,93,17,77,31,44,55,20]
selectionSort(alist)
print(alist)
```

代码 1 选择排序(lst_selectionsortcode)

你可能看出来了，选择排序和冒泡排序做了相同次数的比较，比较复杂度都是 $O(n^2)$，但是，由于减少了交换的次数，在代码的基准测试中选择排序更快。事实上，对我们上述的列表排序，冒泡排序用了 20 次交换，选择排序只有 8 次。

自我检测

Q-28：给定排序列表[11，7，12，14，19，1，6，18，8，20]，如下哪个列表是选择排序第三次遍历之后的列表？

a) [7，11，12，1，6，14，8，18，19，20]
b) [7，11，12，14，19，1，6，18，8，20]
c) [11，7，12，14，1，6，8，18，19，20]
d) [11，7，12，14，8，1，6，18，19，20]

5.3.3. 插入排序

插入排序的算法复杂度仍然是 $O(n^2)$，但其工作原理稍有不同。它总是保持一个位置靠前的已排好的子表，然后每一个新的数据项被“插入”到前边的子表里，排好的子表增加一项。图 4 展示了插入排序过程。阴影数据代表了程序每运行一步后排好的子表。

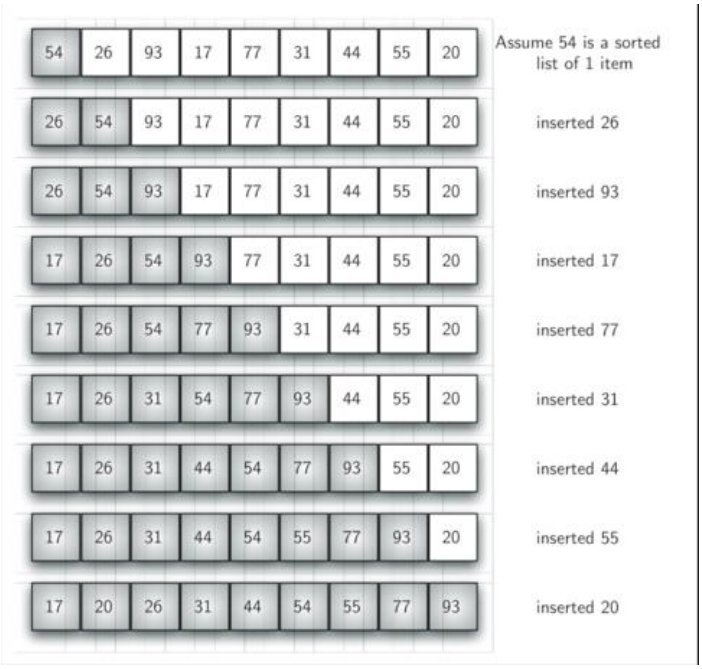


图 4 插入排序

我们认为只含有一个数据项的列表是已经排好的。每排后面一个数据（从 1 开始到 n-1），这个的数据会和已排好的子表数据比较。比较时，我们把之前已经排好的列表中比这个数据大的移到它的右边。当子表数据小于当前数据，或者当前数据已经和子表所有数据比较了时，就可以在此处插入当前数据项。

图 5 展示了第五步排序的细节。程序运行到当前位置，排好的子表包含“17,26,54,77,93”五个数据。我们想让 31 插入该子表中。第一次，31 和 93 比较，93 要移到 31 右边。77 和 54 也要同样移位。遇到 26 时，移动步骤停止，31 被插入到此处。现在我们就有了含 6 个数据项的已排好的子表。

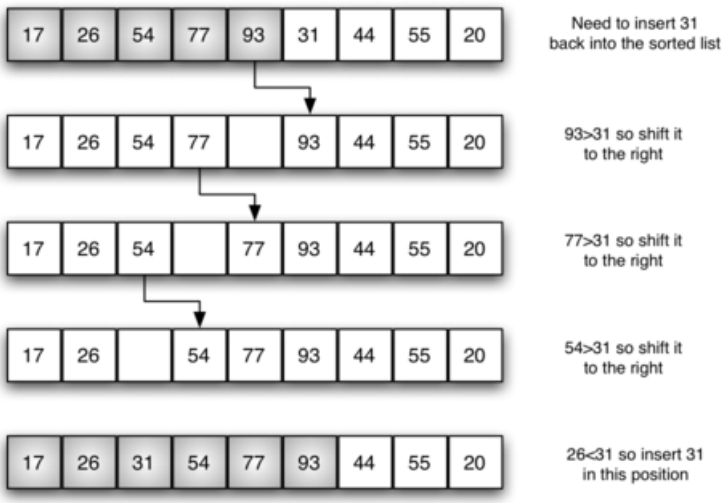


图 5 插入排序：第五步排序

插入排序（代码 1）排 n 个数据仍然需要 n-1 次遍历（n-1 步）。由于每个数据需要被插入到之前排好的子表中，故迭代操作要从只有一个位置的情况开始，在余下 n-1 个位置重复进行。程序第 8 行是转移操作，即把某个数据放到后一个位置，空出当前位置以待数据插入。但是注意，这并不是前两个排序方法里的完整的“交换”步骤。

插入排序需要进行的最多的比较次数仍是前 n-1 个正整数之和，即复杂度为 $O(n^2)$。但是，最好的情况下，每排一个数据只需要一次比较，即列表已经排好的情况。

关于“转移”与“交换”操作的考虑也很重要。通常情况下，“转移”的步骤约为“交换”步骤的 1/3，因为它只有一次赋值操作。在基准测试中，插入排序将展示非常好的性能。

```
def insertionSort(alist):
   for index in range(1,len(alist)):

     currentvalue = alist[index]
     position = index
```

```
    while position>0 and alist[position-1]>currentvalue:
        alist[position]=alist[position-1]
        position = position-1

    alist[position]=currentvalue

alist = [54,26,93,17,77,31,44,55,20]
insertionSort(alist)
print(alist)
```

代码 1 插入排序(lst_insertion)

自我检测

Q-22: 给定排序列表[15, 5, 4, 18, 12, 19, 14, 10, 8, 20] ，如下哪个列表是插入排序第三次遍历之后的列表？

a) [4, 5, 12, 15, 14, 10, 8, 18, 19, 20]
b) [15, 5, 4, 10, 12, 8, 14, 18, 19, 20]
c) [4, 5, 15, 18, 12, 19, 14, 10, 8, 20]
d) [15, 5, 4, 18, 12, 19, 14, 8, 10, 20]

5.3.4. 希尔排序

希尔排序有时又叫做“缩小间隔排序”，它以插入排序为基础，将原来要排序的列表划分为一些子列表，再对每一个子列表执行插入排序，从而实现对插入排序性能的改进。划分子列的特定方法是希尔排序的关键。我们并不是将原始列表分成含有连续元素的子列，而是确定一个划分列表的增量“i”，这个 i 更准确地说，是划分的间隔。然后把间隔为 i 的元素选出来组成子列表。

如图 6，这里有一个含九个元素的列表。如果我们以 3 为间隔来划分，就会分为三个子列表，每一个可以执行插入排序。这三次插入排序完成之后，我们得到了如图 7 所示的列表。虽然这个列表还没有完全排好序，但有趣的是，经过我们对子列的排序之后，列表中的每个元素更加靠近它最终应该处在的位置。

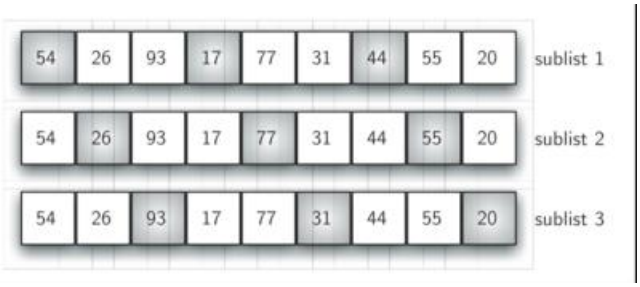


图 6：以 3 为间隔的希尔排序

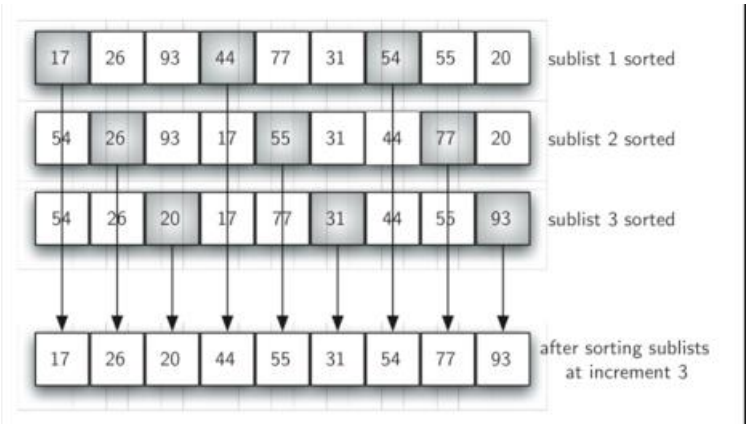


图 7：对每个子列排序后的情况

图 8 为我们展示了最终以 1 为间隔进行插入排序，即标准的插入排序的过程。注意到，在执行了对之前子列的排序之后，我们减少了将原始列表排序时需要比对和移动的次数。在这个例子中，我们仅需要再进行四次移动就可以完成排序。

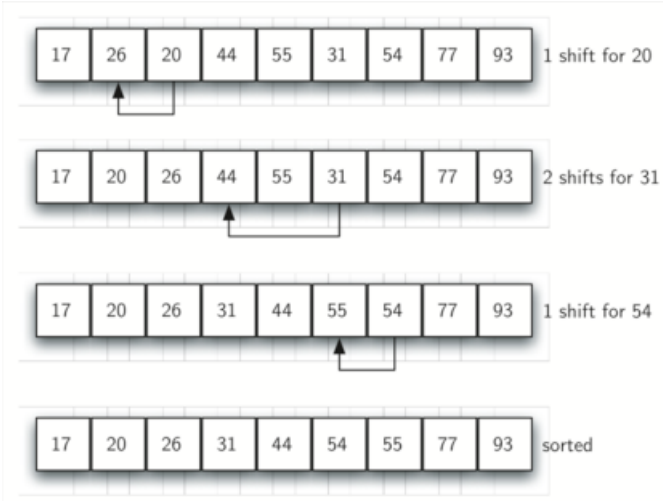


图 8：最后以 1 为间隔的插入排序

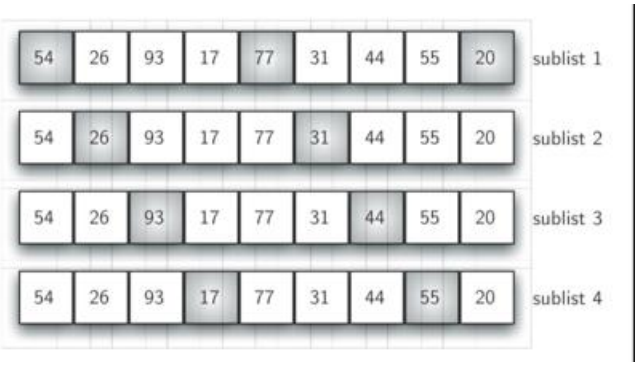


图 9：希尔排序的初始子列

之前我们提到过，特定选取划分间隔的方法是希尔排序的独特之处。代码 1

中的函数使用了一组不同的间隔。在这个例子中，我们从含 2 个元素的子列开始排序；下一步排含 4 个元素的子列。最终，整一个数列用基本的插入排序排好。图 9 显示了我们的例子中用这个间隔的第一组子列表（可能有四种情况）。

下面那个对希尔排序函数的调用会显示出以不同间隔部分排序后的列表，最后一次排序则以 1 为间隔进行插入排序。

```
def shellSort(alist):
    sublistcount = len(alist)//2
    while sublistcount > 0:

      for startposition in range(sublistcount):
        gapInsertionSort(alist,startposition,sublistcount)

      print("After increments of size",sublistcount,
                                  "The list is",alist)

      sublistcount = sublistcount // 2

def gapInsertionSort(alist,start,gap):
    for i in range(start+gap,len(alist),gap):

        currentvalue = alist[i]
        position = i

        while position>=gap and alist[position-gap]>currentvalue:
            alist[position]=alist[position-gap]
            position = position-gap

        alist[position]=currentvalue

alist = [54,26,93,17,77,31,44,55,20]
shellSort(alist)
print(alist)
```

代码 1　　希尔排序

乍一看，你可能会觉得希尔排序不会比插入排序要好，因为它最后一步执行了一次完整的插入排序。但事实上，最后的一次排序并不需要很多次的比对和移动，因为正如上面所讨论的，这个列表已经在之前的对子列的插入排序中实现了部分排序。换句话说，每个步骤使得这个列表与原来相比更趋于有序状态。这使得最后的排序非常高效。

对希尔排序的综合算法分析已经远超出本书的讨论范围，我们可以说它的时

间复杂度大致介于 O(n)和 $O(n^2)$之间。如果使用某些间隔时，它的时间复杂度为 $O(n^2)$。通过改变间隔大小，比如以 2^k-1（1,3,5,7,15,31 等等）为间隔，希尔排序的时间复杂度可以达到 $O(n^{3/2})$。

自我检测

Q-31：给定列表：[5, 16, 20, 12, 3, 8, 9, 17, 19, 7]，下面那个表示了以 3 为间隔完成所有交换后的列表情况？

a) [5, 3, 8, 7, 16, 19, 9, 17, 20, 12]
b) [3, 7, 5, 8, 9, 12, 19, 16, 20, 17]
c) [3, 5, 7, 8, 9, 12, 16, 17, 19, 20]
d) [5, 16, 20, 3, 8, 12, 9, 17, 20, 7]

5.3.5.归并排序

我们现在把注意力转移到用分而治之的策略来提高排序算法的表现。我们要学的第一种算法就是归并排序。归并排序是一种递归算法，它持续地将一个列表分成两半。如果列表是空的或者只有一个元素，那么根据定义，它就被排序好了（最基本的情况）。如果列表里的元素超过一个，我们就把列表拆分，然后分别对两个部分调用递归排序。一旦这两个部分被排序好了，那么最基本的操作——归并，就被执行了。归并是这样一个过程：把两个排序好了的列表结合在一起组合成一个单一的，有序的新列表。图 10 就展示了我们熟悉的作为例子的列表如何被归并排序算法拆分，而图 11 则展示了一个他们组合成单一的列表的过程。

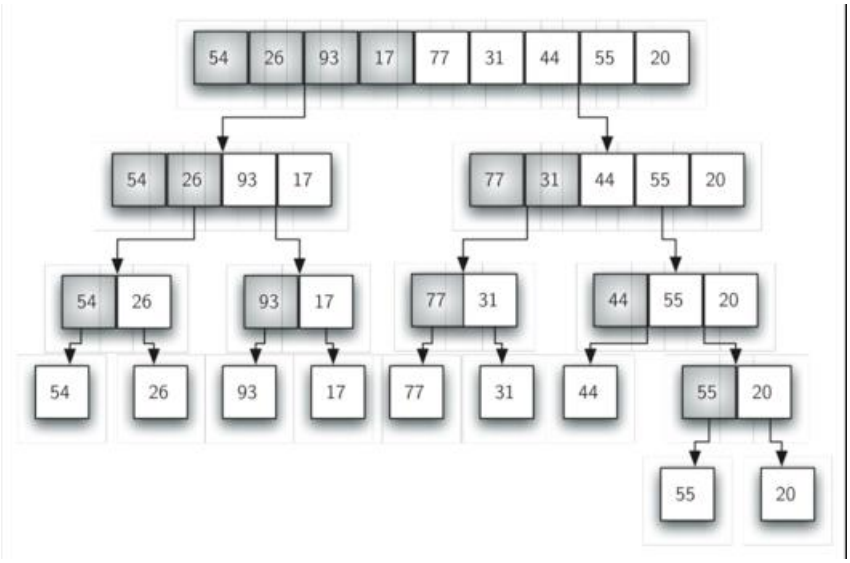


图 10：在归并排序中拆分列表

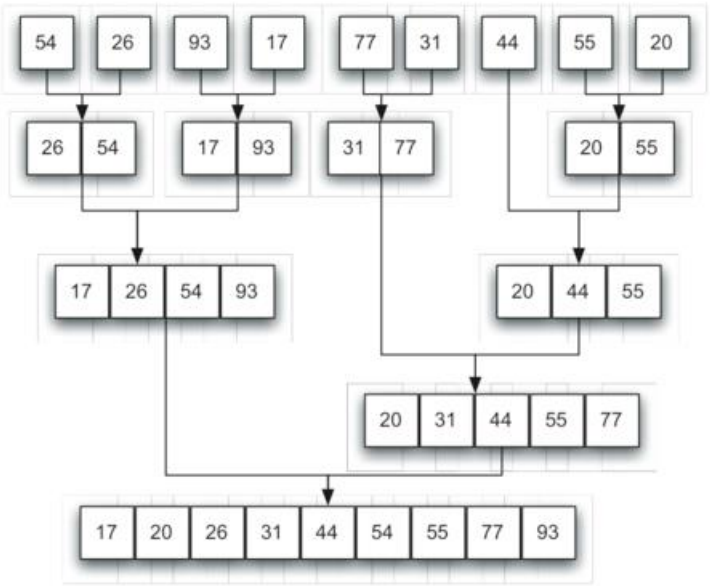


图 11：被组合中的列表

在源代码 1 中展示的归并排序函数开始于询问一个基本条件的问题。如果列表的长度小于等于一，那么我们已经有了一个排好序的列表并不需要做任何更多的过程了。如果不是，即列表长度大于一，那么我们用 Python 的切片操作将左右两部分拆开。注意到这一点是非常重要的：拆分出的两个列表的元素数目不一定相等。但是这无关紧要，因为长度的最大差别也不会超过一。

Run Save Load Show in Codelens

```
def mergeSort(alist):
    print("Splitting ",alist)
    if len(alist)>1:
        mid = len(alist)//2
        lefthalf = alist[:mid]
        righthalf = alist[mid:]

        mergeSort(lefthalf)
        mergeSort(righthalf)

        i=0
        j=0
        k=0
        while i<len(lefthalf) and j<len(righthalf):
            if lefthalf[i]<righthalf[j]:
                alist[k]=lefthalf[i]
                i=i+1
```

```
            else:
                alist[k]=righthalf[j]
                j=j+1
            k=k+1

        while i<len(lefthalf):
            alist[k]=lefthalf[i]
            i=i+1
            k=k+1

        while j<len(righthalf):
            alist[k]=righthalf[j]
            j=j+1
            k=k+1
    print("Merging ",alist)

alist = [54,26,93,17,77,31,44,55,20]
mergeSort(alist)
print(alist)
```

Merge Sort (lst_merge)

一旦归并排序函数被调用处理左右两半边（8-9 行），它假定它们已经被排序完了。而剩余部分的函数（11-31 行）是用来把两个小的有序列表合成一个大的有序列表的。注意归并操作是重复地把两个小列表中的最小元素一一放回原列表的。

归并排序函数被增加了一个打印语句（第二行）用来显示每次被调用之前被排序的列表的内容。同时也有一条打印语句（32 行）用来显示归并的过程。记录显示了函数在我们的示例列表里的执行结果。显示了有 44,55 和 20 的列表并没有被平均地拆分，44 被分给了第一个列表而 55 和 20 被分给了第二个。我们可以很容易地看到拆分过程是如何最终产生一个能迅速被另一些有序列表合并的列表的。

Initialize Run Stop
Beginning Step Forward Step Backward End

为了分析归并算法，我们需要考虑它实施的两个不同步骤。第一步，列表被拆分，我们已经（在二分查找中）计算过，我们能通过 logn 的数量级的计算将长度为 n 的列表拆分。而第二个过程是合并。每个列表中的元素最终将被处理并被放置在排序好的列表中，所以合并操作对一个长度为 n 的列表需要 n 的数量级的操作。因此分析结果就是，拆分需要 logn 数量级的操作而每次拆分需要 n 数量级的操作因此最终操作的复杂度为 nlogn。归并排序是一种 O(nlogn) 的算法。

回忆起，切片操作对于一个长度为 K 的切片需要 K 的复杂度，为了保证归并排序的复杂度为 O(nlogn)，我们需要移开切片操作，这是可能的，只要我们简

单地在递归的时候传入列表的开始和结束的参数就可以了。我们将这个步骤作为一个课后练习。

很重要的是要意识到在拆分列表时归并排序函数需要额外的空间来存放被拆分出来的两个部分。如果列表很大的话，这额外空间将是一个很重要的因素，可能使得这种排序被运用在大数据集合时出现错误。

自我检测

Q-23: 给定如下数字列表 [21, 1, 26, 45, 29, 28, 2, 9, 16, 49, 39, 27, 43, 34, 46, 40]哪个答案展示了归并排序被三次调用后的结果？

a) [16, 49, 39, 27, 43, 34, 46, 40]
b) [21,1]
c) [21, 1, 26, 45]
d) [21]

Q-24: 给定如下数字列表[21, 1, 26, 45, 29, 28, 2, 9, 16, 49, 39, 27, 43, 34, 46, 40] 哪个答案展示了最先被合并的两个列表？

a) [21, 1] 和[26, 45]
b) [[1, 2, 9, 21, 26, 28, 29, 45] 和[16, 27, 34, 39, 40, 43, 46, 49]
c) [21] 和 [1]
d) [9] 和[16]

5.3.6. 快速排序

快速排序用了和归并排序一样分而治之的方法来获得同样的优势，但同时不需要使用额外的存储空间。经过权衡之后，我们发现列表不分离成两半是可能的，当这发生的时候，我们可以看到，操作减少了。

快速排序首先选择一个中值。虽然有很多不同的方法来选择这个数值，我们将会简单地选择列表里的第一项。中值的角色在于辅助拆分这个列表。中值在最后排序好的列表里的实际位置，我们称之为分割点的，是用来把列表变成两个部分来随后分别调用快速排序函数的。

图 12 展示了 54 将被用来作为我们的第一个中值。由于我们已经看过这儿例子很多遍了，我们知道 54 最终将占据 31 的位置。随后，分区过程将要开始。它将找到分割点的位置并且同时将比中值小的数放在左边，大的放在右边。

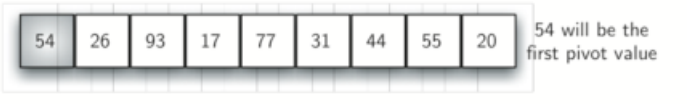


图 12 快速排序的第一个中值

分区过程由设置两个位置标记开始——让我们叫它们左标记和右标记——在列表的第一项和最后一项（图 13 的位置 1 和位置 8）。分区过程的目标是把相

对于中值在错误的一边的数据放到正确的一边，同时找到分割点。图 13 展示了这个过程，同时我们把 54 放到了正确的位置。

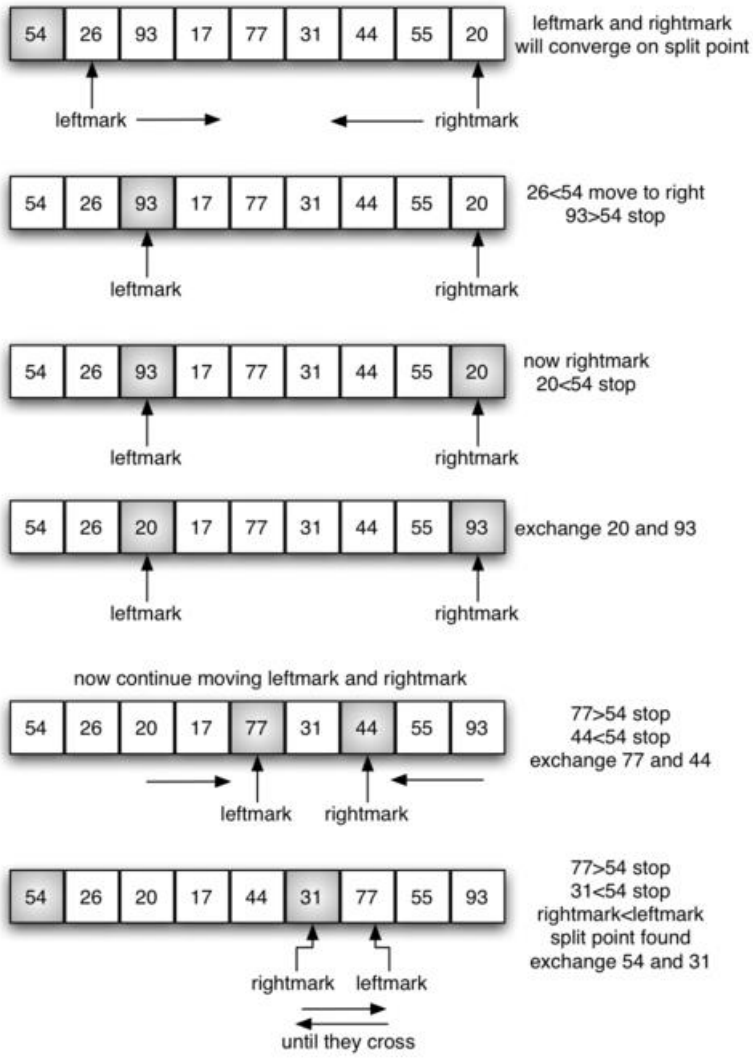


图 13 为 54 找到分割点

我们是这样开始的：我们不断把左标记向右移动直到它指向了一个比中值更大的数字。我们然后把右标记向左移动直到我们找到一个比中值更小的数字。在这个时候我们就找到了两个相对于最终的分割点在错误的位置的元素。在我们的例子中，就是 93 和 20。现在我们可以交换这两个元素，然后重复这个步骤了。

在右标记变得比左标记小的时候，我们停止，此时右标记在的位置就是分割点在的位置。而中值就可以和分割点的内容互换位置而被放置在正确的位置上了。另外，每个分割点左边的元素都比中值小，每个右边的都比它大了。这个列表就可以在分割点被分成两半然后快速排序可以在这两个部分被分别调用了。

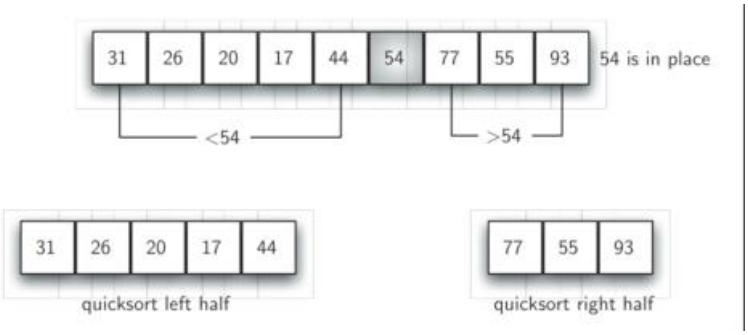


图 14：完成分裂过程找到 54 应在的分割点

在源代码 1 中所示的快速排序函数中，调用了递归函数，quickSortHelper。quickSortHelper 从与归并排序相同的最基本的情况开始。如果列表长度小于或者等于 1，那么它已经排过序。如果列表的长度更大，那么列表可以被分割并进行递归排序。分裂函数的实现过程在前面已经描述。

Run Save Load Show in Codelens

```
def quickSort(alist):
   quickSortHelper(alist,0,len(alist)-1)

def quickSortHelper(alist,first,last):
   if first<last:

       splitpoint = partition(alist,first,last)

       quickSortHelper(alist,first,splitpoint-1)
       quickSortHelper(alist,splitpoint+1,last)


def partition(alist,first,last):
   pivotvalue = alist[first]

   leftmark = first+1
   rightmark = last

   done = False
   while not done:

       while leftmark <= rightmark and \
               alist[leftmark] <= pivotvalue:
           leftmark = leftmark + 1

       while alist[rightmark] >= pivotvalue and \
```

```
                rightmark >= leftmark:
            rightmark = rightmark -1

        if rightmark < leftmark:
            done = True
        else:
            temp = alist[leftmark]
            alist[leftmark] = alist[rightmark]
            alist[rightmark] = temp

    temp = alist[first]
    alist[first] = alist[rightmark]
    alist[rightmark] = temp


    return rightmark

alist = [54,26,93,17,77,31,44,55,20]
quickSort(alist)
print(alist)

Quick Sort (lst_quick)
```

Initialize Run Stop
Beginning Step Forward Step Backward End

分析快速排序函数，注意一个长度为 n 的列表，如果每次分裂都发生在列表的中央，那么将会重复 logn 次分裂。 为了找到分割点，n 个项目中的每一个都需要和中值进行对比。那么，综合起来是 nlogn。此外，快速排序不需要像在归并排序时所需的额外内存。

不幸的是,在最坏的情况下,分割点可能不在中间,可以是非常偏左或偏右，留下一个很不均匀的分裂。在最坏的情况下，给一个长度为 n 的列表排序，分成了给一个长度为 0 的列表排序和给一个长度为 n-1 的列表排序。然后，给一个长度为 n-1 的列表排序，分成了给一个长度为 0 的列表排序和给一个长度为 n-2 的列表排序,以此类推。最终,用以上所述的递归过程,成了时间复杂度为 O(n^2)的排序。

我们之前提到过，有多种不同的方法用来选择中值。特别的是，我们可以通过使用一种名为“三点取样”的方法，来尝试着降低不均匀分割的可能性。为了选择中值，我们要考虑列表中第一个、中间一个以及最后一个三个元素。在我们的例子中，它们分别是 54,77 和 20。现在选取中间值（在我们的例子中是 54），并且使用它作为中值（当然，这和我们最初使用的中值相同）。这种方法在于当列表的第一项不趋于处于列表中间位置时,三个值的中间值将会是一个更好的中

值选择。当最初的列表已经经过一定程度的排序时，这种方法就显得尤其有用。我们将这种中值的选择方法留下作为一个练习。

自我检测

Q-25：给定列表[14, 17, 13, 15, 19, 10, 3, 16, 9,12]。依据给出的快速排序算法，下面哪个选项表示的是上述列表在快速排序时，在第2次分裂后的列表内容：

a) [9, 3, 10, 13, 12]
b) [9, 3, 10, 13, 12, 14]
c) [9, 3, 10, 13, 12, 14, 17, 16, 15, 19]
d) [9, 3, 10, 13, 12, 14, 19, 16, 15, 17]

Q-26：给定列表 [1, 20, 11, 5, 2, 9, 16, 14, 13,19]。使用“三点取样”，下面哪个选项表示的是上述列表在快速排序时，得到的第1个“中值”：

a) 1 b) 9 c) 16 d) 19

Q-27：下面哪些算法，即使在最坏情况下，复杂度还保证是O(nlog n)

a)谢尔排序
b)快速排序
c)归并排序
d)插入排序

5.4.小结

- 在无序表或者有序表上的顺序搜索，其时间复杂度为O(n)；
- 在有序表上进行二分查找，在最差情况下，复杂度为O(log n)；
- 散列表可以实现常数级时间的搜索；
- 冒泡排序、选择排序和插入排序是O(n^2)的算法；
- 谢尔排序在插入排序的基础上进行了改进，采用对递增子表排序的方法，其时间复杂度可以在O(n)和O(n^2)之间；
- 归并排序的时间复杂度是O(nlog n)，但归并的过程需要额外存储空间；
- 快速排序的时间复杂度是O(nlog n)，但如果分割点偏离列表中心的话，最坏情况下会退化到O(n^2)。快速排序不需要额外的存储空间。

5.5.关键词

二分搜索	冒泡排序	数据项链
聚集	冲突	冲突解决
折叠法	间隔	散列函数
散列表	散列过程	插入排序

线性探测	负载系数	映射
三点取样	归并	归并排序
平方取中	开放定址	分裂
完美散列函数	中值	二次探查
快速排序	再散列	选择排序
顺序查找	谢尔排序	改进冒泡
槽	分裂点	

5.6. 问题讨论

1. 使用在本章中给出的散列表的性能公式，计算平均情况下必要的比较次数，当散列表中已占槽数与总槽数之比为：

- 10%
- 25%
- 50%
- 75%
- 90%
- 99%

在什么情况下，你认为散列表太小了？请解释。

2. 使用将在字符串中所在的位置作为权重因子的方法，修改散列函数。

3. 我们使用一个将在字符串中所在的位置作为权重因子的散列函数。设计一个可选择的加权方案。这些函数的偏向有何不同？

4. 研究完美散列函数。使用一个姓名的列表（同学，家人……），用完美散列函数生成对应的散列值。

5. 产生一个随机的整数列表，并表示该列表是怎样被下列的算法排序的:

- 冒泡排序
- 选择排序
- 插入排序
- 谢尔排序（增量由你决定）
- 归并排序
- 快速排序（取中值由你决定）

6. 考虑下面的整数列表：[1,2,3,4,5,6,7,8,9,10]，表示该列表是怎样被下列的算法排序的:

- 冒泡排序
- 选择排序
- 插入排序

- 谢尔排序（增量由你决定）
- 归并排序
- 快速排序（取中值由你决定）

7. 考虑下面的整数列表：[10,9,8,7,6,5,4,3,2,1]，表示该列表是怎样被下列的算法排序的:
   - 冒泡排序
   - 选择排序
   - 插入排序
   - 谢尔排序（增量由你决定）
   - 归并排序
   - 快速排序（取中值由你决定）

8. 考虑下面的字母列表：['P','Y','T','H','O','N']，表示该列表是怎样被下列的算法排序的:
   - 冒泡排序
   - 选择排序
   - 插入排序
   - 谢尔排序（增量由你决定）
   - 归并排序
   - 快速排序（取中值由你决定）

9. 为快速排序中选择中值设计一些可选择的策略。例如，选择中间项。实现新的快排算法并且使用一些随机的数据集进行测试。在什么条件下，你的新算法表现的比本章所给的算法更优或更劣？

5.7.编程练习

1. 设置一个随机实验，测试在整数列表中顺序搜索和二分搜索的区别。
2. 使用教材中给出的二分搜索（递归和迭代）。生成一个随机的、有序的整数列表。
   对每个函数做一个标准检查程序分析。你的结果是什么？你能解释一下吗？
3. 用递归算法实现二分搜索，避免用切片操作。记得，对子列表传递的除了列表外，还要有开始以及结束位置在列表中的索引值。通过一个随机产生的有序整数列表来与课件中采用切片操作的二分搜索比较性能。
4. 为散列表实现的 ADT Map 实现 len 方法（__len__）。
5. 为散列表实现的 ADT Map 实现 in 方法（__contains__）。
6. 你如何从一个使用数据项链的方法来解决冲突的散列表中删除项目？如果使用的是开放定址的方法呢？必须处理的特殊情况是什么？实现实现 Hashtable 类的 del 方法。
7. 在散列表实现的 Map 中，散列表的大小被定为 101。如果散列表被填满，它需要增长。重新实现 put 方法，使得当负载因子达到某一预定值时（你可以

决定这一值基于你对负载与性能的评估），散列表会自动调整自身的大小。

8. 实现二次探查作为再散列手段。
9. 使用一个随机数生成器，产生一个 500 个整数的列表。使用本章中的一些排序算法进行基准分析。比较它们在执行速度上的快慢。
10. 使用同时赋值的方法实现冒泡排序。
11. 冒泡排序可以被修改为向两个方向冒泡。第一次操作向上冒泡，第二次操作向下冒泡。这种交替模式一直持续到不需要更多的操作。实现这种改变并描述在什么情况下这种改变是合适的。
12. 使用同时赋值的方法实现选择排序。
13. 使用不同的增量设置在同一个列表上，进行一个谢尔排序的基准分析。
14. 不使用切片操作实现归并排序函数。
15. 提高快速排序的一种方法是在较短长度（称之为“分裂限制”）的列表中使用插入排序。为什么这是有意义的？重新实现快速排序并用它来为随机整数列表排序。使用不同大小的分裂限制进行分析。
16. 实现“三点取样法”，用来在快速排序中确定中值的改进，进行一个实验来比较两种方法。