

类数据成员

当有一些值（如常量），被一个类的所有实例共享时，我们就会经常用到类级的数据成员。在这种情况下，在每个实例的命名空间中存储这个值就会造成不必要的浪费。例如，我们回顾一下 2.4.1 节中介绍的 PredatoryCreditCard 类，在该类中会因为信用卡额度限制而使试图支付 5 美元费用的操作失败。我们选择 5 美元的费用是有点随意的，如果使用命名变量，而不是将文字值嵌入代码中，我们的编码风格会更好。通常，这些费用的数额是由银行的政策决定的，对每个客户都一样。这种情况下，我们可像如下样式定义和使用类数据成员：

```
class PredatoryCreditCard(CreditCard):
    OVERLIMIT_FEE = 5 # this is a class-level member

    def charge(self, price):
        success = super().charge(price)
        if not success:
            self._balance += PredatoryCreditCard.OVERLIMIT_FEE
        return success
```

数据成员 OVERLIMIT_FEE 直接进入 PredatoryCreditCard 类命名空间，因为赋值在类定义的直接范围内发生，并且没有任何限定标识符。

嵌套类

在另一个类的范围内嵌套一个类定义也是可行的。这是一个有用的结构，我们在本书的数据结构实现中多次予以探讨。可以使用如下语法完成：

```
class A: # the outer class
    class B: # the nested class
    ...
```

在这种情况下，B 类是嵌套类。标识符 B 是进入了 A 类的命名空间相关联的一个新定义的类。我们注意到这种技术与继承的概念无关，因为 B 类不继承 A 类。

在一个类中嵌套另一个类，这表明嵌套类的存在需要外部类的支持。此外，它有助于减少潜在的命名冲突，因为它允许类似的命名类存在于另一个上下文中。例如，我们稍后将介绍链表的数据结构，它通过定义一个嵌套节点类来存储列表的各个组件。我们还介绍树的数据结构，这取决于其自身的嵌套节点类。这两个结构根据不同的节点定义，我们可以通过在各自的容器类中嵌套各自的节点定义来避免歧义。

将一个类嵌套为另一个类的成员还有一个优点，就是它允许更高级形式的继承，使外部类的子类重载嵌套类的定义。我们将在 11.2 节中实现树结构的节点时使用这种技术。

字典和 __slots__ 声明

默认情况下，Python 中的每个命名空间均代表内置 dict 类的一个实例（参见 1.2.3 节），即将范围内识别的名称与相关联的对象映射起来。虽然字典结构支持相对有效的名称查找，但它需要的额外内存使用量超出了它存储原始数据的内存（我们将在第 10 章探讨实现字典的数据结构）。

Python 提供了一种更直接的机制来表示实例命名空间，以避免使用一个辅助字典。使用流表示一个类的所有实例，类定义必须提供一个名为 __slots__ 的类级别的成员分配给一个固定的字符串序列以此服务于变量。例如，在 CreditCard 类中，声明如下：

```
class CreditCard:
    __slots__ = '_customer', '_bank', '_account', '_balance', '_limit'
```

在这个例子中，赋值的右边是一组元组（见 1.9.3 节元组的自动打包）。

如果使用继承时，基类声明了 `__slots__`，那么为了避免字典实例的创建，子类也必须声明 `__slots__`。子类的声明只需包含新创建的补充方法的名称。例如，`PredatoryCreditCard` 的声明如下：

```
class PredatoryCreditCard(CreditCard):
    __slots__ = '_apr'           # in addition to the inherited members
```

我们可以选择使用 `__slots__` 简化本书中每个类的声明，但并不会这样做，因为这样将使 Python 程序非典型。也就是说，这本书里有几个类，我们希望有大量的实例，每个代表一个轻量级构造。例如，当讨论嵌套类，我们建议链表和树作为数据结构通常来组成大量的个体节点。为了更好地提升内存使用效率，我们将在所有期望有很多实例的嵌套类中使用显式的 `__slots__` 声明。

2.5.2 名称解析和动态调度

在上一节中，我们讨论了各种命名空间以及建立访问命名空间的机制。在本节中，我们将研究在 Python 的面向对象框架中检索名称的过程。当用点运算符语法访问现有的成员（如 `obj.foo`）时，Python 解释器将开始一个名称解析的过程，描述如下：

1) 在实例命名空间中搜索，如果找到所需的名称，关联值就可以使用。

2) 否则在该实例所属的类的命名空间中搜索，如果找到名称，关联值可以使用。

3) 如果在直接的类的命名空间中没有，搜索仍在继续，通过继承层次结构向上，检查每一个父类的类名称空间（通常通过检查超类，接着是超类的超类，等等）。第一次找到这个名字，它的关联值可以使用。

4) 如果还没有找到该名称，就会引发一个 `AttributeError` 异常。

举一个实际的例子，假设 `mycard` 标识的 `PredatoryCreditCard` 类的一个实例。考虑以下可能的使用模式：

- `mycard._balance` (等价于内部方法体中的 `self._balance`)：在 `mycard` 实例命名空间中找到 `_balance` 方法。
- `mycard.process_month()`：开始搜索实例命名空间，但是在这个名称空间没有找到 `process_month()`。因此，在 `PredatoryCreditCard` 类命名空间搜索；在本例中，这个名字找到了，方法也调用了。
- `mycard.make_payment(200)`：没有在实例命名空间和 `PredatoryCreditCard` 类命名空间中找到 `make_payment`，该名称是在超类 `CreditCard` 中解析出来的，继承方法也被调用了。
- `mycard.charge(50)`：在实例命名空间中搜索 `charge` 名称失败。接着检查 `PredatoryCreditCard` 类的命名空间，因为这是实例的真实类型。在该类中有一个 `charge` 函数的定义，该方法也可以调用。

最后一个案例显示，`PredatoryCreditCard` 类的 `charge` 函数重载了 `CreditCard` 命名空间中 `charge` 函数的版本。在传统的面向对象术语中，Python 使用动态调度（或动态绑定）来确定运行时基于调用其对象的类型实现函数的调用，这与一些使用静态调度的语言相似，即在编译时基于变量声明的类型来决定调用函数的版本。

2.6 深拷贝和浅拷贝

在第 1 章中，我们曾强调，一个赋值语句 `foo = bar` 使对象 `bar` 有一个别名 `foo`。在本节

中，我们考虑的是拷贝对象的一个副本，而不是一个别名。在应用程序中，当我们想以一种独立的方式修改原始的或拷贝的内容时，这是非常必要的。

考虑这样一个场景：在该场景中，我们各种列表的颜色，每个颜色代表假定颜色类的一个实例。我们让标识符 `warmtones` 表示现有的颜色（如橙色、棕色）列表。在这个应用程序中，我们希望创建一个名为 `palette` 的新列表，复制一份 `warmtones` 列表。不过，我们想随后可以在 `palette` 中添加额外的颜色，或修改、删除一些现有的颜色，而不影响 `warmtones` 的内容。如果执行命令

```
palette = warmtones
```

就创建了一个别名，如图 2-9 所示，没有创建新的列表。相反，新的标识符 `palette` 参考原先的列表。

不幸的是，这不符合我们的期望，因为如果随后在 `palette` 中添加或删除颜色，我们修改的列表为 `warmtones`。

我们可以用以下语法创建一个新的列表实例：

```
palette = list(warmtones)
```

在这种情况下，我们显式调用列表构造函数，将第一个列表作为参数，这将导致一个新的列表被创建，如图 2-10 所示，这被称为浅拷贝。新的列表被初始化，以便其内容与原来的序列相同。然而，Python 的列表是用作参考的（见 1.2.3 节），所以新列表与原列表代表了引用相同元素的顺序。

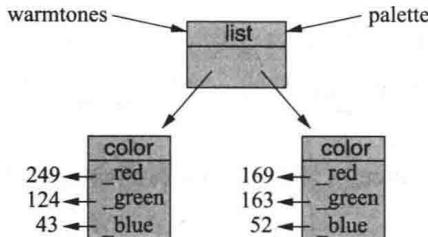


图 2-9 相同颜色列表的两个别名

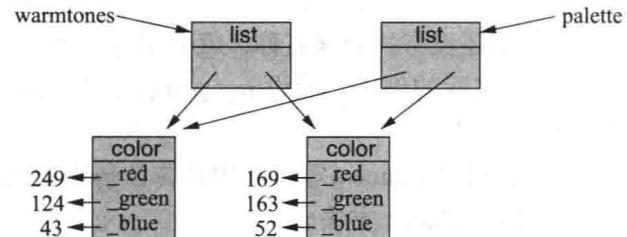


图 2-10 颜色列表的浅拷贝

这比第一次尝试的情况更好，我们可以合理地从 `palette` 添加或删除元素而不影响 `warmtones`。然而，如果编辑 `palette` 中的颜色实例列表，则相对改变了 `warmtones` 的内容。尽管 `palette` 和 `warmtones` 是不同的列表，但仍有间接的混叠，例如，`palette[0]` 和 `warmtones[0]` 为相同颜色实例的别名。

我们更希望 `palette` 是 `warmtones` 的深拷贝。在深拷贝中，新副本引用的对象也是从原始版本中复制过来的（见图 2-11）。

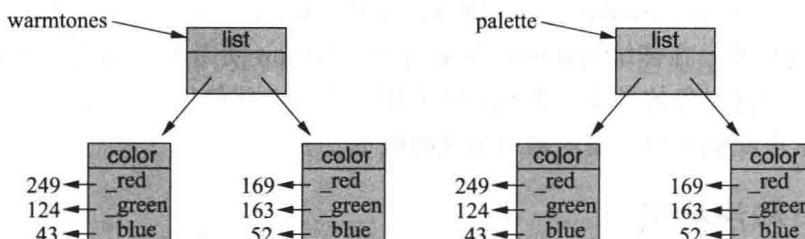


图 2-11 颜色列表的深拷贝

Python 的 copy 模块

要创建一个深拷贝，可以通过显式复制原始颜色实例来填充列表，但这需要知道如何复制颜色（而不是别名）。Python 提供了一个很方便的模块，即 `copy`，它能产生任意对象的浅拷贝和深拷贝。

该模块提供两个函数：`copy` 函数和 `deepcopy` 函数。`copy` 函数创建对象的浅拷贝，`deepcopy` 函数创建对象的深拷贝。引入模块后，我们可以为例子创建一个深拷贝，如图 2-11 所示，所使用的命令如下：

```
palette = copy.deepcopy(warmtones)
```

2.7 练习

请访问 www.wiley.com/college/goodrich 以获得练习帮助。

巩固

- R-2.1 给出三个生死攸关的软件应用程序的例子。
- R-2.2 给出一个软件应用程序的例子，其中适应性意味着产品销售和破产的生命周期期间的不同。
- R-2.3 描述文本编辑器 GUI 的组件和它封装的方法。
- R-2.4 编写一个 Python 类 `Flower`。该类有 `str`、`int`、`float` 类型的三种实例变量，分别代表花的名字、花瓣的数量和价格。该类必须包含一个构造函数，该构造函数给每个变量初始化一个合适的值。该类应该包含设置和检索每种类型值的方法。
- R-2.5 使用 1.7 节的技术修订 `CreditCard` 类的 `charge` 和 `make_payment` 方法确保调用方可以将一个数字作为参数传递。
- R-2.6 如果 `CreditCard` 类的 `make_payment` 方法接收到的参数是负数，这将影响账户的余额。修改实现，使得传递的参数值如果为负数，即抛出 `ValueError` 异常。
- R-2.7 2.3 节的 `CreditCard` 类将一个新账户的余额初始化为零。修改这个类，使构造函数具有第五个参数作为可选参数，它可以初始化一个余额不为零的新账户。而原来的四参数构造函数仍然可以用来生成余额为零的新账户。
- R-2.8 在代码段 2-3 的 `CreditCard` 类测试中修改第一个 `for` 循环的声明，使三张信用卡的其中之一超过其信用额度。哪张信用卡会出现这种情况？
- R-2.9 实现 2.3.3 节 `Vector` 类的 `__sub__` 方法，使表达式 `u - v` 返回一个代表两矢量间差异的新矢量实例。
- R-2.10 实现 2.3.3 节 `Vector` 类的 `__neg__` 方法，使表达式 `- v` 返回一个新的矢量实例。新矢量 `v` 的坐标值都是负值。
- R-2.11 在 2.3.3 节中，我们注意到 `Vector` 类支持形如 `v = u + [5, 3, 10, -2, 1]` 这样的语法形式，向量和列表的总和返回一个新的向量。然而，语法 `v = [5, 3, 10, -2, 1] + u` 确是非法的。解释应该如何修改 `Vector` 类的定义使得上述语法能够生成新的向量。
- R-2.12 实现 2.3.3 节中的 `Vector` 类的 `__mul__` 方法，使得表达式 `v*3` 返回一个新的矢量实例，新矢量 `v` 的坐标值都是以前的 3 倍。
- R-2.13 练习 R-2.12 要求对 2.3.3 节中的 `Vector` 类实现 `__mul__` 方法，以提供对语法 `v*3` 的支持。试实现 `__rmul__` 方法，提供对语法 `3*v` 的支持。
- R-2.14 实现 2.3.3 节 `Vector` 类的 `__mul__` 方法，使表达式 `u*v` 返回一个标量代表向量点运算的结果，即 $\sum_{i=1}^d u_i \cdot v_i$ 。

R-2.15 2.3.3节的Vector类提供接受一个整数d的构造函数，并产生一个d维向量，它的所有坐标等于0。另一种创建矢量的便捷方式是给构造函数传递一个参数，一些迭代类型可以代表一系列的数字，创建一个向量，它的维度等于序列的长度，坐标值等于序列值。例如，`Vector([7, 4, 5])`会产生一个三维向量，坐标为`<4, 7, 5>`。修改构造函数，使它可以接受任何形式的参数。也就是说，如果一个整数被传递，它就产生了一个所有坐标值为零的向量。但是如果提供了一个序列，它就产生了一个坐标值等于序列值的向量。

R-2.16 2.3.5节的Range类按照如下公式

```
max(0, (stop - start + step - 1) // step)
```

去计算范围内元素的数量。即使假设一个正的step大小，也并不能很明显地看出为什么这个公式提供了正确的计算。可以用你自己的方式证明这个公式。

R-2.17 从下面类的集合中画一个类的继承图：

- Goat类扩展了object类，增加了实例变量_tail以及方法milk()和jump()。
- Pig类扩展了object类，增加了实例变量_nose以及方法eat(food)和wallow()。
- Horse类扩展了object类，增加了实例变量_height和_color以及方法run()和jump()。
- Racer类扩展了Horse类，增加了方法race()。
- Equestrian类扩展了Horse类，增加了实例变量_weight以及方法trot()与is_trained()。

R-2.18 给出一个来自Python代码的简短片段，使用2.4.2节的Progression类，找到那个以2开始且以2作为前两个值的斐波那契数列的第8个值。

R-2.19 利用2.4.2节的ArithmeticProgression类，以0开始，增量为128，在到达整数 2^{63} 或者更大的数时，我们需要执行多少次的调用？

R-2.20 拥有一棵非常深的继承树会有哪些潜在的效率劣势？也就是说，有一个很大的类的集合，A、B、C……，其中B继承自A、C继承自B、D继承自C……

R-2.21 拥有一棵非常浅的继承树会有哪些潜在的效率劣势？也就是说，有一个很大的类的集合，A、B、C……所有的这些类扩展来自一个单一的类Z。

R-2.22 collections.Sequence抽象基类不提供对两个序列的比较支持，从代码段2-14中修改Sequence类，使其定义包含__eq__方法，使两个序列中的元素相等时，表达式seq1 == seq2返回True。

R-2.23 在之前的问题中有相似的问题，使用方法__lt__参数化Sequence类，使其支持字典比较seq1 < seq2。

创新

C-2.24 假设你在一个新的电子书阅读器的设计团队。你的读者将需要Python软件哪些主要的类和方法？你应该为这段代码设计一个继承关系图，但你不需要写任何实际的代码。你的软件体系结构至少应该包括顾客购买新书的方式、查看他们购买书的清单以及阅读他们购买的书籍。

C-2.25 练习R-2.12使用__mul__方法支持使用一个数字乘以Verctor类，而练习R-2.14使用__mul__方法支持运用点运算计算两个向量。给出Verctor.__mul__的一个简单实现，使用运行时类型来检查是否支持这两种语法`u*v`和`u*k`，u和v表示向量实例，k代表一个数字。

C-2.26 2.3.4节的SequenceIterator类提供众所周知的前向迭代器。实现一个名为ReversedSequenceIterator的类，以此作为任何Python序列的反向迭代器。第一次调用next返回序列的最后一个元素，第二次调用next返回倒数第二个元素，以此类推。

C-2.27 在2.3.5节中对于Range r，“`k in r`”，我们注意到Range类的版本隐式地支持迭代，因为它显式支持__len__和__getitem__。该类也接受对布尔类型的隐式支持。这个测试通过范围基于

前向迭代器进行评估，通过试验证明 2 in Range(10000000) 对比 9 999 999 in Range(10000000) 的相对速度。请提供一种 `_contains_` 方法更有效的实现，以确定特定的值是否属于给定范围内。所提供方法的运行时间应独立于范围的大小。

- C-2.28 2.4.1 节的 `PredatoryCreditCard` 类提供了 `process_month` 方法，可使模型完成每月一次的循环。请修改该类，实现这样的功能：在本月内，一旦用户完成十次呼叫，就需要对其收取费用。每增加一个额外的呼叫，收取 1 美元的附加费。
- C-2.29 请修改 2.4.1 节的 `PredatoryCreditCard` 类，实现这样的功能：给用户分配一个每月最低付款额，作为账户的一部分，如果客户在下一个月周期之前没有连续地支付最低金额，则要评估延迟的费用。
- C-2.30 在 2.4.1 节的末尾，我们认为一个 `CreditCard` 类支持非公有制的方法模型 `_set_balance(b)`，可以被子类使用以影响余额的改变，而不直接访问数据成员 `_balance`。相应地修改 `CreditCard` 类和 `PredatoryCreditCard` 类，实现这样一个模型。
- C-2.31 写一个扩展自 `Progression` 类的 Python 类，使 `Progression` 中的每个值都是前两个值差的绝对值。其中应包括一个构造函数，以接受一对数字作为第一和第二个值，使用 2 和 200 作为默认值。
- C-2.32 写一个扩展自 `Progression` 类的 Python 类，使 `Progression` 中的每个值是前一个数值的平方根（注意：你不能用一个整数来表示每个值）。构造函数应该接受一个可选参数用于指定开始值，使用 65536 作为默认值。

项目

- P-2.33 写一个 Python 程序，如输入标准的代数多项式，则输出该多项式的一阶导数。
- P-2.34 写一个 Python 程序，如输入一个文件，则输出一个柱形图表，以显示文档中每个字母字符出现的频率。
- P-2.35 写一组 Python 类，可以模拟网络应用程序的其中一方 Alice，定期创建一组她想发给 Bob 的包。互联网进程不断检查是否 Alice 有想要发送的包，如果有，就发送至 Bob 的计算机，Bob 定期检查自己的计算机，以确定是否收到来自 Alice 的包，如果有，他将阅读并删除包。
- P-2.36 写一个 Python 程序来模拟生态系统，其中包含两种类型的动物——熊与鱼。生态系统还包括一条河流，它被建模为一个比较大的列表。列表中的每一个元素应该是一个 `Bear` 对象、一个 `Fish` 对象或者 `None`。在每一个时间步长，基于随机过程，每一个动物都试图进入一个相邻的列表位置或停留在原处。如果两只相同类型的动物竞争同一单元格，那么它们留在原处，但它们创造了这种类型动物的新实例，实例放置在列表中的一个随机（即以前为 `None`）位置。如果一头熊和一条鱼竞争，那么鱼就会死亡（即它消失了）。
- P-2.37 在之前的项目中写一个模拟器，但添加一个布尔值 `gender` 字段和一个浮点 `strength` 字段到每一个动物，使用 `Animal` 类作为基础类。如果两只同一类型的动物竞争，如果它们是不同性别的动物，那么这种类型只创建一个新的实例；否则，如果两只相同类型和性别的动物竞争，那么只有力量更大的动物才会生存。
- P-2.38 写一个 Python 程序，模拟一个支持电子书阅读器的功能系统。你应该为用户在系统中提供“买”新书、查看他们所购买书的名单以及阅读所购买的书籍的方法。系统应该使用实际的书籍（其版权已经过期并可在互联网上获得），为系统用户“购买”和阅读提供可用的书籍。
- P-2.39 基于拥有抽象方法 `area()` 和 `perimeter()` 的 `Polygon` 类发展继承层次结构。实现扩展自基类的 `Triangle`、`Quadrilateral`、`Pentagon`、`Hexagon` 和 `Octagon` 类，伴随着具有明显意义的 `area()` 和

perimeter() 方法。同时实现 IsoscelesTriangle、EquilateralTriangle、Rectangle 和 Square 类，它们有适当的继承关系。最后，写一个简单的程序，允许用户创建各种类型的多边形，输入它们的几何尺寸，输出面积和周长。附加功能：允许用户通过指定顶点坐标输入多边形，并能够测试两个多边形是否相似。

扩展阅读

对于计算机科学与工程发展的广泛概述，请阅读《The Computer Science and Engineering Handbook》^[96]。关于 Therac-25 事件更多的信息，详见 Leveson 和 Turner^[69] 的文章。

有兴趣学习面向对象编程的读者，可以参考由 Booch^[17]、Budd^[20]、Liskov 和 Guttag^[71] 编写的书。Liskov 和 Guttag 提供了关于抽象数据类型很精彩的讨论，Cardelli 和 Wegner^[23] 撰写了调研论文，Demurjian^[33] 参与编写了《The Computer Science and Engineering Handbook》^[96]一书的相关章节。书中描述的设计模式由 Gamma 等人^[41] 完成的。

重点介绍 Python 中面向对象编程的图书包括由 Goldwasser 和 Letscher^[43] 编写的入门书籍，以及由 Phillips^[83] 编写的进阶书籍。

算法分析

有一个经典的故事，国王委托著名的数学家阿基米德判断黄金王冠是否如声称的那样是纯金的而没有掺白银。当阿基米德进入浴盆洗澡时，他发现了一个解决方法。他注意到，自己身体进入浴盆的体积与水溢出浴盆的数量成比例。这给了阿基米德启示，他立刻跳出浴盆，赤裸着身体奔跑在大街上大喊“找到了！找到了！”。他发现了一个分析工具（排水量）。只要用一个简单的天平，就可以判断国王的新王冠是不是纯金的。具体做法就是：阿基米德把王冠和同等质量的黄金分别沉到一碗水里，观察两者的排水量是否一样。这个发现对金匠来说是不幸的，因为如果阿基米德进行分析后，发现王冠溢出的水比同等质量的纯金块所溢出的水多，那就意味着王冠不是纯金的。

在本书中，我们对设计“优秀”的数据结构和算法感兴趣。简言之，数据结构是组织和访问数据的一种系统化方式，算法是在有限的时间里一步步执行某些任务的过程。这些概念对计算极为重要，为了分辨哪些数据结构和算法是“优秀”的，我们需要一些精确分析算法的方法。

我们在本书中用到的主要分析方法包括算法和数据结构的运行时间和空间利用表示。运行时间是一个很好的度量，因为时间是宝贵的资源——计算机解决方案应该运行得尽可能快。一般来说，一个算法或数据结构操作的运行时间随着输入大小而增加，尽管它可能对相同大小的不同输入也有所变化。另外，运行时间也受硬件环境（例如，处理器、时钟频率、内存、硬盘）以及算法实施和执行的软件环境（例如，操作系统、程序设计语言）的影响。当其他所有因素不变时，如果计算机有更快的处理器，或者程序编译到本机代码来执行而不是解释执行，有相同输入数据的相同算法的运行时间会更少。我们将在本章的开始部分讨论进行实验研究的工具，并讨论将其作为评估算法效率的一种主要方法的局限性。

要研究运行时间这一度量，要求我们会用一些数学工具。尽管可能存在来自不同环境因素的干扰，但是我们主要关注算法的运行时间和其输入大小的关系。我们希望将算法的运行时间表示为输入大小的函数。但是，度量它的合适途径是什么？在本章中，我们将自己动手开发一种分析算法的数学方法。

3.1 实验研究

如果算法已经实现了，我们可以通过在不同的输入下执行它并记录每一次执行所花费的时间来研究它的运行时间。Python 中一个简单的实现方法是使用 time 模块的 time() 函数。这个函数传递的是自新纪元基准时间后已经过去的秒数或分数（新纪元是指 1970 年）。当我们可以通过记录算法运行前的那一刻以及算法执行完毕后的那一刻，并且计算它们之间的差（如下所示）来判定消逝的时间时，新纪元的选择不影响测试时间的结果。

```
from time import time
start_time = time()           # record the starting time
run algorithm
end_time = time()             # record the ending time
elapsed = end_time - start_time # compute the elapsed time
```

在第 5 章，我们将演示这种方法的使用，即在 Python list 类的效率上收集实验数据。用

这样的方法测量消逝的时间很好地反映了算法效率，但绝不意味着完美。`time()` 函数的测量是相对于“挂钟”的。因为许多进程共享使用计算机的中央处理器（CPU），所以算法执行过程花费的时间依赖于在作业执行时正运行在计算机上的其他进程。一个更公正的度量是算法使用的 CPU 周期的数量。即使用相同的输入重复相同的算法可能没有保持一致性，也要使用 `time` 模块的 `clock()` 函数，并且它的粒度依赖于计算机系统。Python 包含了一个更高级的模块（名叫 `timeit`），它可以自动地做多次重复实验来评估差异。

通常我们认为运行时间依赖于输入的大小和结构，所以应该在各种大小的不同测试输入上执行独立实验。接下来我们可以通过绘制算法每次运行的性能图来可视化结果， x 坐标表示输入大小 n ， y 坐标表示运行时间 t 。图 3-1 显示了这样的假设性数据。这种可视化可以提供关于算法的问题大小和执行时间的直观描述。这可用于对实验数据做统计分析，以寻找符合实验数据的最好的输入大小函数。为了使得分析更有意义，要求选择好的样本输入并且对其进行足够多次的测试，使算法运行时间的统计更准确。

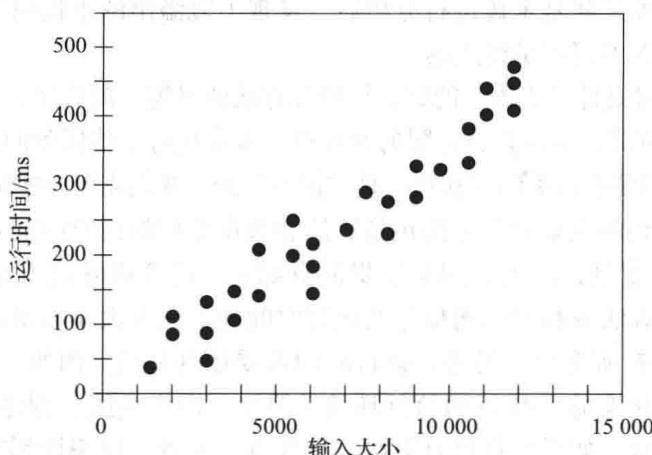


图 3-1 一个算法运行时间的实验研究结果。坐标 (n, t) 中的点表示对于输入大小 n 所测出的算法的运行时间 t (ms)

实验分析的挑战

虽然执行时间的实验研究是有用的，使用算法分析有 3 个主要的局限性（尤其是在优化生产质量代码时）：

- 很难直接比较两个算法的实验运行时间，除非实验在相同的硬件和软件环境中执行。
- 实验只有在有限的一组测试输入下才能完成，因此它们忽略了不包括在实验中的输入的运行时间（这些输入可能是重要的）。
- 为了在实验上执行算法来研究它的执行时间，算法必须完全实现。

最后一个要求是实验研究应用中最严重的缺点。在设计的初期，当考虑数据结构或算法的选择时，花费大量的时间实现一个显然低劣的算法是不明智的。

进一步的实验分析

我们的目标是开发一种分析算法效率的方法：

- 1) 在软硬件环境独立的情况下，在某种程度上允许我们评价任意两个算法的相对效率。
- 2) 通过研究不需要实现的高层次算法描述来执行算法。
- 3) 考虑所有可能的输入。

计算原子操作

为了在没有执行实验时分析一个算法的执行时间，我们用一个高层次的算法描述直接进行分析（可以是真实的代码片段，也可以是独立于语言的伪代码）。我们定义了一系列原子操作，如下所示：

- 给对象指定一个标识符
- 确定与这个标识符相关联的对象
- 执行算术运算（例如，两个数相加）
- 比较两个数的大小
- 通过索引访问 Python 列表的一个元素
- 调用函数（不包括函数内的操作执行）
- 从函数返回

从形式上说，一个原子操作相当于一个低级别指令，其执行时间是常数。理想情况下，这可能是被硬件执行的基本操作类型，尽管许多原子操作可能被转换成少量的指令。我们并不是试着确定每一个原子操作的具体执行时间，而是简单地计算有多少原子操作被执行了，用数字 t 作为算法执行时间的度量。

操作的计数与特定计算机中真实的运行时间相关联，每个原子操作相当于固定数量的指令，并且该操作只有固定数量的原子操作。这个方法中的隐含假设是不同原子操作的运行时间是非常相似的。因此算法执行的原子操作数 t 与算法的真实运行时间成正比。

随着输入函数的变化进行测量操作

为了获取一个算法运行时间的增长情况，我们把每一个算法和函数 $f(n)$ 联系起来，其中把执行的原子操作的数量描述为输入大小 n 的函数 $f(n)$ 。3.2 节将会介绍 7 个最常见的函数。3.3 节将介绍一个函数之间相互比较的数学框架。

最坏情况输入的研究

对于相同大小的输入，算法针对某些输入的运行速度比其他的更快。因此，我们不妨把算法的运行时间表示为所有可能的相同大小输入的平均值的函数。不幸的是，这样的平均情况分析是相当具有挑战性的。它要求定义一组输入的概率分布，这通常是一个困难的工作。图 3-2 表明，根据输入分布，算法的运行时间可以在最坏和最好情况运行时间之间的任何地方。例如，假设实际上输入只有“A”或“D”类型将会怎么样？

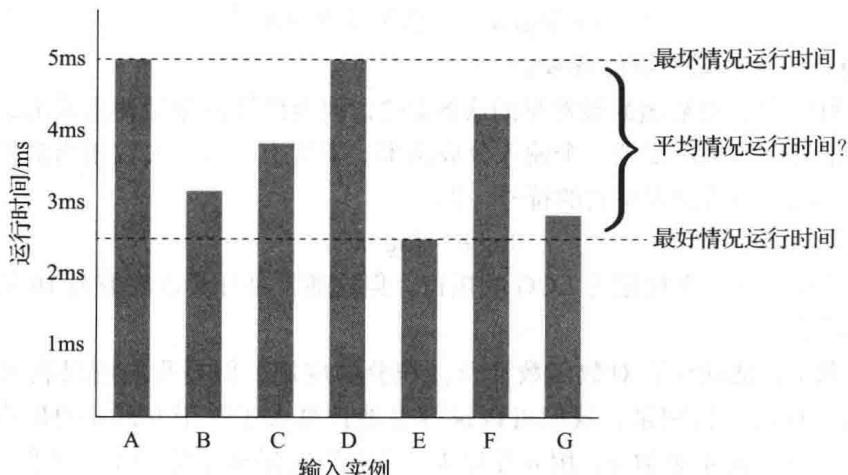


图 3-2 最好情况和最坏情况运行时间的不同。每个条柱代表一些算法在不同输入时的运行时间

平均情况分析通常要求计算基于给定输入分布的预期运行时间，这通常涉及复杂的概率理论。因此，在本书的其余部分，除非特别指明，一般我们都按照最坏情况把算法的运行时间表示为输入大小 n 的函数。

最坏情况分析比平均情况分析容易很多，它只需要有识别最坏情况输入的能力，这通常是很简单的。另外，这个方法通常会导出更好的算法。算法在最坏情况下很好执行的标准必然是该算法在每一个输入情况都能很好地执行。也就是说，最坏情况的设计会使得算法更加健壮，这很像一个飞毛腿总是在斜坡上练习跑步。

3.2 本书使用的 7 种函数

在这一节，我们将简要讨论用在算法分析中最重要的 7 种函数。我们把这 7 种简单的函数用在本书的几乎所有分析中。事实上，某些章节使用的函数不同于这 7 种的将会被标记为星号 (*)，以表明这是可选的。除了这 7 种基本的函数，附录 B 包含了其他被应用在数据结构和算法分析中的一系列有用的数学定理。

3.2.1 常数函数

我们能想起的最简单的函数是常数函数。这个函数是

$$f(n) = c$$

对一些固定的常数 c ，例如 $c = 5$ 、 $c = 7$ 或 $c = 2^{10}$ 。也就是说，对任意参数 n ，常数函数 $f(n)$ 的值都是 c 。换言之， n 的值是什么并不重要， $f(n)$ 总是为定值 c 。

我们对整数函数最感兴趣，因此最基本的常数函数是 $g(n) = 1$ ，这是用在本书中最经典的常数函数。注意，任何其他的函数 $f(n) = c$ 都可以被写成常数 c 乘以 $g(n)$ ，即 $f(n) = cg(n)$ 。

正因为常数函数简单，所以它在算法分析中是很有用的，它描述了在计算机上需要做的基本操作的步数，例如两个数相加、给一些变量赋值或者比较两个数的大小。

3.2.2 对数函数

数据结构和算法分析中令人感兴趣甚至惊奇的是无处不在的对数函数， $f(n) = \log_b n$ ，常数 $b > 1$ 。此函数定义如下：

$$x = \log_b n \quad \text{当且仅当 } b^x = n$$

按照定义， $\log_b 1 = 0$ 。 b 是对数的底数。

在计算机科学中，对数函数最常见的底数是 2，因为计算机存储整数采用二进制，并且许多算法中的常见操作是反复把一个输入分成两半。事实上，这个底数相当常见，以至于当底数等于 2 时，我们通常会省略它的符号，即

$$\log n = \log_2 n$$

大多数手持计算器上有一个标记为 LOG 的按钮，但这通常是计算以底数为 10 的对数，而不是底数为 2 的对数。

对任意整数 n ，准确计算对数函数涉及微积分的应用，但是我们可以利用近似值来足够好地实现这一目的。特别是，我们可以很容易地计算大于等于 $\log_b n$ 的最小整数（即向上取整， $\lceil \log_b n \rceil$ ）。对正整数 n ，用 n 除以 b ，只有当结果小于等于 1 时才停止除法操作， $\lceil \log_b n \rceil$ 的值即为 n 除以 b 的次数。例如， $\lceil \log_3 27 \rceil$ 等于 3，因为 $((27/3)/3)/3 = 1$ 。同样，

$\lceil \log_4 64 \rceil$ 等于 3，因为 $((64/4)/4) = 1$ ，并且 $\lceil \log_2 12 \rceil$ 是 4，因为 $((12/2)/2)/2 = 0.75 \leq 1$ 。

对于大于 1 的底数，接下来的命题描述了对数的几个重要特性。

命题 3-1 (对数规则): 给定实数 $a > 0, b > 1, c > 0, d > 1$ ，有：

- 1) $\log_b(ac) = \log_b a + \log_b c$
- 2) $\log_b(a/c) = \log_b a - \log_b c$
- 3) $\log_b(a^c) = c \log_b a$
- 4) $\log_b a = \log_d a / \log_d b$
- 5) $b^{\log_d a} = a^{\log_d b}$

按照惯例，没有括号的符号 $\log n^c$ 指 $\log(n^c)$ 的值。我们用简写符号 $\log^c n$ 表示 $(\log n)^c$ ，在 $(\log n)^c$ 中对数的结果以幂级增大。

上面的特性可以推导出取幂的相反规则，这将在本节后面给出。我们用一些例子描述这些特性。

例题 3-2： 我们用示例演示一下命题 3-1 提到的算法规则（按照惯例，对数的底若省略了，底数即为 2）。

- $\log(2n) = \log 2 + \log n = 1 + \log n$ ，由对数规则 1 得出。
- $\log(n/2) = \log n - \log 2 = \log n - 1$ ，由对数规则 2 得出。
- $\log n^3 = 3\log n$ ，由对数规则 3 得出。
- $\log 2^n = n \log 2 = n \cdot 1 = n$ ，由对数规则 3 得出。
- $\log_4 n = (\log n) / \log 4 = (\log n) / 2$ ，由对数规则 4 得出。
- $2^{\log n} = n^{\log 2} = n^1 = n$ ，由对数规则 5 得出。

作为一个实际问题，对数规则 4 给出了用计算器上以 10 为底的对数按钮 (LOG) 来计算以 2 为底的对数的方法，即

$$\log_2 n = \text{LOG } n / \text{LOG } 2$$

3.2.3 线性函数

另一个简单却很重要的函数是线性函数，

$$f(n) = n$$

即，给定输入值 n ，线性函数 f 就是 n 本身。

这个函数出现在我们必须对所有 n 个元素做基本操作的算法分析的任何时间。例如，比较数字 x 与大小为 n 的序列中的每一个元素，需要做 n 次比较。线性函数也实现了用任何算法处理不在计算机内存中的 n 个对象的最快运行时间，因为读 n 个对象已经需要 n 次操作了。

3.2.4 $n \log n$ 函数

接下来要讨论的函数是 $n \log n$ 函数，

$$f(n) = n \log n$$

对于一个输入值 n ，这个函数是 n 倍的以 2 为底的 n 的对数。这个函数的增长速度比线性函数快，比二次函数慢。因此，与运行时间是二次的算法相比较，我们更喜欢运行时间与 $n \log n$ 成比例的算法。我们会看到一些运行时间与 $n \log n$ 成比例的重要算法。例如，对 n 个任意数进行排序且运行时间与 $n \log n$ 成比例的最快可能算法。

3.2.5 二次函数

另一个经常出现在算法分析中的函数是二次函数，

$$f(n) = n^2$$

即，给定输入值 n ，函数 f 的值为 n 与自身的乘积（即“ n 的平方”）。

二次函数用在算法分析中的主要原因是，许多算法中都有嵌套循环，其中内存循环执行一个线性操作数，外层循环则表示执行线性操作数的次数。因此，在这个情况下，算法执行了 $n \cdot n = n^2$ 个操作。

嵌套循环和二次函数

二次函数也可能出现在嵌套循环中，第一次循环迭代使用的操作数为 1，第二次为 2，第三次为 3，等等。即操作数为

$$1 + 2 + 3 + \cdots + (n - 2) + (n - 1) + n$$

换言之，如果内层循环的操作数随外层循环的每次迭代逐次加 1，这个函数即表示嵌套循环总的操作数。这个数量也有一个有趣的典故。

1787 年，一个德国教师让 9 ~ 10 岁大的小学生计算从 1 ~ 100 所有整数之和。立刻有一个孩子说自己已经有答案了！老师很怀疑，因为这个孩子的答题板上只有一个答案。但是，他的答案 5050 却是正确的。这个孩子长大后成了那个时代最伟大的数学家之一，他就是卡尔·高斯。我们推测年轻的高斯用了下面的恒等式。

命题 3-3：对于任何一个整数 $n \geq 1$ ，我们有：

$$1 + 2 + 3 + \cdots + (n - 2) + (n - 1) + n = \frac{n(n + 1)}{2}$$

图 3-3 中所示即为命题 3-3 的两个“可视化”的证明。

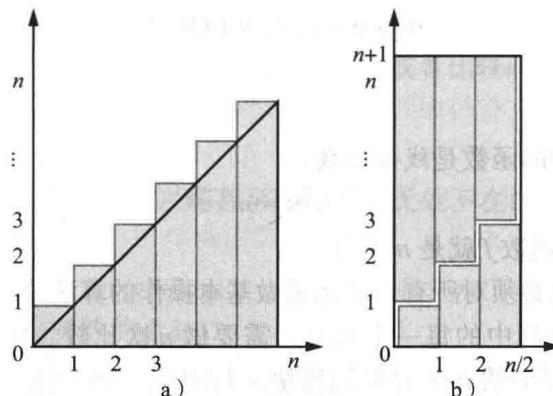


图 3-3 命题 3-3 的可视化的证明。通过 n 个单位宽度并且高度分别为 $1, 2, \dots, n$ 的矩形的总面积，两个分图都可视化了上述的等式。在图 3-a 中，这些矩形被表示成一个面积为 $n \cdot n / 2$ 的大三角形（底为 n 、高为 n ）加上 n 个面积 $1/2$ 为的小三角形（底为 1、高为 1）。在图 3-b 中，这仅适用于当 n 为偶数的情形，所述矩形被表示成一个底为 $n/2$ 、高为 $n+1$ 的大矩形。

从命题 3-3 得到的结论是，如果我们执行一个含有嵌套循环的算法，那么在内循环中每次增加一个操作，执行外循环时操作的总数是 n 的平方。更确切地说，操作的总数是 $n \cdot n / 2 + n/2$ ，所以与一个在内循环执行时每次使用 n 个操作的算法相比，这仅仅是这个算法操作总数的一半多一些。但增长的阶数仍然是 n 的平方。

3.2.6 三次函数和其他多项式

继续我们对函数输入能力的讨论，我们考虑三次函数（cubic function）

$$f(n) = n^3$$

这个函数分配一个输入值 n ，可以得到 n 的三次方这样一个输出。与前面提到的常数函数、线性函数和平方函数相比，这个函数在算法分析文章中出现的频率较低，但它确实会时不时地出现。

多项式

到目前为止，我们已经列出的大多数函数可以看作一个更大的类函数（多项式）的一部分。一个多项式函数有如下的形式，

$$f(n) = a_0 + a_1n + a_2n^2 + a_3n^3 + \dots + a_dn^d$$

其中 a_0, a_1, \dots, a_d 都是常数，称为多项式的系数，并且整数 d 表示多项式中的最高幂次，称为多项式的次数。

例如，下列所有函数都是多项式：

- $f(n) = 2 + 5n + n^2$
- $f(n) = 1 + n^3$
- $f(n) = 1$
- $f(n) = n$
- $f(n) = n^2$

因此，我们可能会质疑，在用于算法分析时本书仅仅提出了 4 个重要的函数，但之所以我们坚持说有 7 个函数，那是因常量函数、线性函数和二次函数太重要而不能与其他多项式放在一起。而且较小次数的多项式的运行时间一般比较大次数的多项式的运行时间要好。

求和

在数据结构和算法的分析中一次又一次出现的表示法就是求和，其定义如下：

$$\sum_{i=a}^b f(i) = f(a) + f(a+1) + f(a+2) + \dots + f(b)$$

其中 a 和 b 都是整数，并且 $a \leq b$ 。之所以出现在数据结构与算法分析中，是因为循环的运行时间自然会引起求和。

使用求和，我们可以把命题 3-3 的公式改写为

$$\sum_{i=1}^n i = \frac{n(n+1)}{2}$$

同样，我们可以写一个系数为 a_0, \dots, a_d 次数为 d 的多项式为

$$f(n) = \sum_{i=0}^d a_i n^i$$

如此一来，求和符号就为我们表达越来越多项的和提供一种简便方法，其中这些项都是规则的结构。

3.2.7 指数函数

用在算法分析中的另一个函数是指数函数，

$$f(n) = b^n$$

其中 b 是一个正的常数，称为底，参数 n 是指数。也就是说，函数 $f(n)$ 分配给输入参数 n 的值是通过底数 b 乘以它自己 n 次获得的。考虑到对数函数的情况，在算法分析中，指数函数最基本的情况是 $b = 2$ 。例如，含有 n 位的整数字可以表示小于 2^n 的所有非负整数。如果通过执行一个操作开始一个循环，然后每次迭代所执行的操作数目翻倍，则在第 n 次迭代所执行的操作数目为 2^n 。

然而，我们有时会有除了 n 的其他指数，因此，对于我们来说知道一些便捷的处理指数的规则是有用的。以下这些指数规则是相当有帮助的。

命题 3-4 (指数规则): 对于给定正整数 a 、 b 和 c ，我们有

$$1) (b^a)^c = b^{ac}$$

$$2) b^a b^c = b^{a+c}$$

$$3) \frac{b^a}{b^c} = b^{a-c}$$

例如，我们有以下例子：

$$\bullet 256 = 16^2 = (2^4)^2 = 2^{4*2} = 2^8 = 256 \text{ (指数规则 1)}$$

$$\bullet 243 = 3^5 = 3^{2+3} = 3^2 3^3 = 9 * 27 = 243 \text{ (指数规则 2)}$$

$$\bullet 16 = \frac{1024}{64} = \frac{2^{10}}{2^6} = 2^{10-6} = 2^4 = 16 \text{ (指数规则 3)}$$

如下所述，我们可以把指数函数扩展到指数是分数和实数的情况或者负指数的情况。给出一个正整数 k ，我们定义 $b^{\frac{1}{k}}$ 为 b 的 k 次根，即存在一个数 r ，使得 $r^k = b$ 。例如 $25^{\frac{1}{2}} = 5$ ，即 $5^2 = 25$ 。同样， $27^{\frac{1}{3}} = 3$ ， $16^{\frac{1}{4}} = 2$ 。通过指数规则 1，这种方法允许我们定义任意次幂的指数 $b^{\frac{a}{c}} = (b^a)^{\frac{1}{c}}$ ，该指数可以表示为一个分数，例如， $9^{\frac{3}{2}} = (9^3)^{\frac{1}{2}} = 729^{\frac{1}{2}} = 27$ 。因此， $b^{\frac{a}{c}}$ 实际上正是整数指数 b^a 的 c 次根。

我们可以进一步把指数函数 b^x 扩展到参数为任意实数 x 的指数，通过计算一系列形为 $b^{\frac{a}{c}}$ 的值，分数 $\frac{a}{c}$ 逐渐得到越来越接近 x 的值。任意一个实数 x 可以通过分数来实现任意程度的近似，因此，我们可以用分数 $\frac{a}{c}$ 作为 b 的指数来任意程度地接近指数 b^x 。例如，数 2^u 是一个很好的定义。最后，给定一个负指数 d ，我们定义 $b^d = \frac{1}{b^{-d}}$ ，这对应于指数规则 3，其中 $a = 0$ 和 $c = -d$ 。例如， $2^{-3} = \frac{1}{2^3} = \frac{1}{8}$ 。

几何求和

假设有一个循环，它的每次迭代需要一个比前一个更长时间的乘法因子。那么这个循环可以使用下列命题进行分析。

命题 3-5: 对于任意整数 $n \geq 0$ 和任意实数 a ，比如 $a > 0$ 和 $a \neq 1$ ，考虑下述的和

$$\sum_{i=0}^n a^i = 1 + a + a^2 + \cdots + a^n$$

(记住如果 $a > 0$ ，那么 $a^0 = 1$ 。) 这个总和等于

$$\frac{a^{n+1} - 1}{a - 1}$$

命题 3-5 所示的求和被称为几何求和，因为如果 $a > 1$ ，在几何规模上每一项都比它的

前一项大。例如，从事计算工作的每个人都应该知道

$$1 + 2 + 4 + 8 + \dots + 2^{n-1} = 2^n - 1$$

因为这是在二进制表示法中使用 n 位可以表示的最大整数。

3.2.8 比较增长率

综上所述，按顺序给出的算法分析使用的 7 个常用函数如表 3-1 所示。

表 3-1 函数的类型（这里我们假设 $a > 1$ 并且是一个常数）

常数函数	对数函数	线性函数	$n \log n$ 函数	二次函数	三次函数	指数函数
1	$\log n$	n	$n \log n$	n^2	n^3	a^n

理想情况下，我们希望数据结构的操作运行时间与常数函数或者对数函数成正比，而且我们希望算法以线性函数或 $n \log n$ 函数来运行。运行时间为二次或者三次的算法不太实用，除最小输入规模的情况外，运行时间为指数的算法是不可行的。7 个函数的增长率如图 3-4 所示。

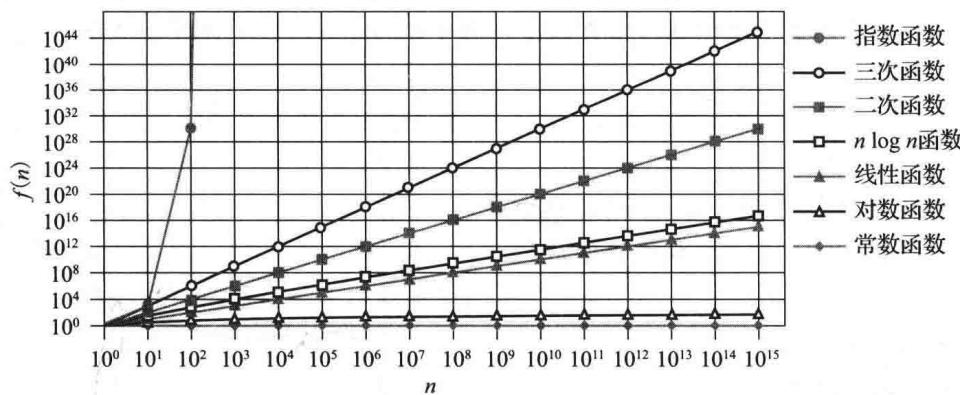


图 3-4 在算法分析中使用的 7 个基本函数的增长率。对于指数函数我们用底 $a = 2$ 的函数表示。这些函数绘制在双对数图上，主要是通过图形的坡度来比较增长率。即使如此，指数函数因增长过快而不能在图表上显示其所有值

向下取整和向上取整函数

以上函数还有一个方面要额外考虑。在讨论到对数时我们指出，对数的值通常不是一个整数，然而一个算法的运行时间通常是通过一个整数来表示的，比如操作的数量。因此，一个算法的分析有时可能涉及向下取整和向上取整函数的使用，它们分别定义如下：

- $\lfloor x \rfloor$ = 小于或者等于 x 的最大整数。
- $\lceil x \rceil$ = 大于或者等于 x 的最小整数。

3.3 渐近分析

在算法分析中，我们重点研究运行时间的增长率，采用宏观方法把运行时间视为输入大小为 n 的函数。例如，通常只要知道算法的运行时间为按比例增长到 n 就足够了。

我们用函数的数学符号（不考虑那些不变因子）来分析算法。换句话说，我们这样用函

数描述算法的运行时间：输入一个 n 值，对应输出一个数据，用这个数据来反映决定关于 n 的增长率的主要因素。这种方法表明：在伪代码描述或高级语言执行的每一个基本步骤中可以用几个微指令来描述。因此，我们能够通过估计执行不变因子的微指令的个数来执行算法分析，而不必再苦恼于在特定语言或者特定硬件下分析执行在计算机上的操作的精准个数。

作为一个实际的例子，我们再来回想一下 1.4.2 节中在 Python 列表中寻找最大数的要求。如代码段 3-1 所示，在介绍循环时，第一次引入了这个例子来表示一个找列表中最大值的函数。

代码段 3-1 返回 Python 列表最大值的函数

```

1 def find_max(data):
2     """Return the maximum element from a nonempty Python list."""
3     biggest = data[0]          # The initial value to beat
4     for val in data:           # For each value:
5         if val > biggest:    # if it is greater than the best so far,
6             biggest = val    # we have found a new best (so far)
7     return biggest            # When loop ends, biggest is the max

```

这是运行时间按比例增长到 n 这种算法的一个经典例子，当循环中的每一个数据元素执行一次时，一些相应数量的微指令也在这个过程中执行了一次。在本节的末尾，我们提供了一个框架来规范这个声明。

3.3.1 大 O 符号

令 $f(n)$ 和 $g(n)$ 作为正实数映射正整数的函数。如果有实型常量 $c > 0$ 和整型常量 $n_0 \geq 1$ 满足

$$f(n) \leq cg(n), \text{ 当 } n \geq n_0$$

我们就说 $f(n)$ 是 $O(g(n))$ 。

这种定义就是通常说的大 O 符号，因为它有时被说成“ $f(n)$ 是 $g(n)$ 的大 O”。图 3-5 展示了一般的定义。

例题 3-6： 函数 $8n + 5$ 是 $O(n)$ 。

证明：通过大 O 的定义，我们需要找到一个实型常量 $c > 0$ 和一个整型常量 $n_0 \geq 1$ ，对于任意一个整数 $n \geq n_0$ ，满足 $8n + 5 \leq cn$ 。很容易找到一个可能的选择： $c = 9$ ， $n_0 = 5$ 。当然，这是无限种可选择组合中的一个，因为在 c 和 n_0 之间有一个权衡。比如说，我们能够设定常数 $c = 13$ ， $n_0 = 1$ 。

大 O 符号的含义是：当给定一个常数因子且在渐近意义上 n 趋于无穷时，函数 $f(n)$ “小于或等于” 函数 $g(n)$ 。这种思想来源于这样一个事实：从渐近的角度来说，当 $n \geq n_0$ 时，规定用“ \leq ” 来比较 $f(n)$ 和 $g(n)$ 与一个常数的乘积。然而，如果说“ $f(n) \leq O(g(n))$ ” 未免显得不合适，因为大 O 已经有“小于或等于”的意思。同样，如果用“ $=$ ” 关系的一般理解来说，“ $f(n) \leq O(g(n))$ ” 也不完全正确，尽管这很常见，因为没办法说明白“ $O(g(n)) = f(n)$ ” 这种对称语句。最好是说

$$f(n) \text{ 是 } O(g(n))$$

或者，我们可以说“ $f(n)$ 是 $g(n)$ 的量级”。如果用更倾向于数学的语言，这样说也是正确的：

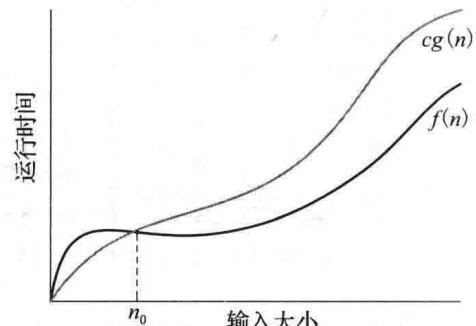


图 3-5 解释大 O 符号。当 $n \geq n_0$ 时，因为 $f(n) \leq c \cdot g(n)$ ，所以函数 $f(n)$ 是 $O(g(n))$

“ $f(n) \in O(g(n))$ ”，从技术上说，大O符号代表多个函数的集合。在本书中，我们仍将大O声明为“ $f(n)$ 是 $O(g(n))$ ”。即使这样解释，我们在如何使用大O符号参与算术运算上仍有相当大的自由，以及由这种自由所带来的一定的责任。

使用大O符号描述运行时间

通过设定一些参数 n ，大O符号被广泛用于描述运行时间和空间界限，虽然参数的设定依据问题的不同而有所不同，但（大O符号）仍是一种测量问题“尺寸”的可选择的方法。例如，假如我们对在一个序列中找最大数感兴趣，当使用找最大数算法时，我们应该用 n 表示这个集合中元素的个数。运用大O符号，我们能够为任意一台计算机写出关于找最大数算法（代码段3-1）的运行时间的数学化的精准语句。

命题3-7：找最大数算法（即计算一系列数中的最大数）的运行时间为 $O(n)$ 。

证明：在循环开始之前初始化时，仅仅需要固定数量的基本操作。循环的每一次重复也仅仅需要固定的基本操作，并且循环执行 n 次。因此，我们可以通过选择适当的常数 c' 和 c'' （这两个常数在初始化和循环体中能分别反应执行状况），就可计算出基本操作的数量，即 $c' + c''$ 。因为每一个基本操作的运行时间是固定的，我们可以通过输入一个 n 值来计算找最大数算法的运行时间，运行时间最多也就是一个常数乘以 n 。所以，我们得出结论，找最大数算法的运行时间为 $O(n)$ 。■

大O符号的一些性质

大O符号使得我们忽视常量因子和低阶项，转而关注函数中影响增长的主要成分。

例题3-8： $5n^4 + 3n^3 + 2n^2 + 4n + 1$ 是 $O(n^4)$ 。

证明：注意， $5n^4 + 3n^3 + 2n^2 + 4n + 1 \leq (5+3+2+4+1)n^4 = cn^4$ ，令 $c=15$ ，当 $n \geq n_0=1$ 时即满足题意。■

事实上，我们可以描述任何多项式函数的增长速率。

命题3-9：如果 $f(n)$ 是一个指数为 d 的多项式，即

$$f(n) = a_0 + a_1n + \dots + a_dn^d$$

且 $a_d > 0$ ，则 $f(n)$ 是 $O(n^d)$ 。

证明：注意，当 $n \geq 1$ 时，我们有 $1 \leq n \leq n^2 \leq \dots \leq n^d$ 。因此，

$$a_0 + a_1n + a_2n^2 + \dots + a_dn^d \leq (|a_0| + |a_1| + |a_2| + \dots + |a_d|)n^d$$

令 $c = |a_0| + |a_1| + |a_2| + \dots + |a_d|$ ， $n_0 = 1$ ，即可得出 $f(n)$ 是 $O(n^d)$ 。■

因此，多项式中的最高阶项决定了该多项式的渐近增长速率。在练习中，我们考虑大O符号另外一些性质。接下来让我们来进一步考虑一些例子，这些例子主要集中在用于算法设计中的7个基本函数的结合上。我们依据当 $n \geq 1$ 时， $\log n \leq n$ 这样的一个数学定理。

例题3-10： $5n^2 + 3n \log n + 2n + 5$ 是 $O(n^2)$ 。

证明： $5n^2 + 3n \log n + 2n + 5 \leq (5+3+2+5)n^2 = cn^2$ ，令 $c=15$ ，当 $n \geq n_0=1$ 时（满足题意）。■

例题3-11： $20n^3 + 10n \log n + 5$ 是 $O(n^3)$ 。

证明：当 $n \geq 1$ 时， $20n^3 + 10n \log n + 5 \leq 35n^3$ 。■

例题3-12： $3 \log n + 2$ 是 $O(\log n)$ 。

证明：当 $n \geq 2$ 时， $3 \log n + 2 \leq 5 \log n$ 。注意，当 $n=1$ 时， $\log n=0$ 。这就是为何在此处用 $n \geq n_0=2$ 。■

例题 3-13: 2^{n+2} 是 $O(2^n)$ 。

证明: $2^{n+2} = 2^n \cdot 2^2 = 4 \cdot 2^n$; 因此, 这种情况下我们令 $c = 4$, $n_0 = 1$ 。 ■

例题 3-14: $2n + 100 \log n$ 是 $O(n)$ 。

证明: 当 $n \geq n_0 = 1$ 时, $2n + 100 \log n \leq 102n$; 因此, 此时我们令 $c = 102$ 。 ■

用最简单的术语描述函数

总的来说, 我们应该用大 O 符号尽可能接近地描述函数。虽然函数 $f(n) = 4n^3 + 3n^2$ 是 $O(n^5)$ 或者甚至是 $O(n^4)$, 但说 $f(n)$ 是 $O(n^3)$ 更精确。通过类比考虑一个场景:

一位饥饿的旅行者在一条漫长的乡村小路开车, 突然遇到一位刚从集市回家的农民。假设旅行者问农民自己还要开多久才能找到食物, 虽然农民回答“当然不会再超过 12 个小时”也是对的, 但告诉旅行者“沿着这条路再行驶几分钟就会看到一个超市”却更精确 (且更有用)。因此, 即便是使用大 O 符号, 我们仍然需要尽可能地还原整个真相。

如果在大 O 符号里使用常数因子和低阶项, 也会被认为不得体。例如, 函数 $2n^2$ 是 $O(4n^2 + 6n \log n)$, 尽管说法完全正确, 但却不常用。我们应尽力用最简单的术语来描述函数。

3.2 节列举的 7 个函数最常和大 O 符号结合起来描述算法的运行时间和空间使用情况。事实上, 我们通常用函数的名称来引用其所描述的算法的运行时间。因此, 例如, 我们可以说以 $O(n^2)$ 运行的二次算法的最坏运行时间为 $4n^2 + n \log n$ 。同样, 若一个算法运行时间最大为 $5n + 20 \log n + 4$, 则这样的算法被称为线性算法。

大 Ω

正如大 O 提供了一种渐近说法: 一个函数的增长速率“小于或等于”另一个函数, 接下来的符号提供了另一种渐近说法: 一个函数的增长速率“大于或等于”另一个函数。

设 $f(n)$ 和 $g(n)$ 为正实数映射正整数的函数, 如果 $g(n)$ 是 $O(f(n))$, 即存在实常数 $c > 0$ 和整型常数 $n_0 \geq 1$ 满足

$$f(n) \geq cg(n), \text{ 当 } n \geq n_0 \text{ 时},$$

我们就说 $f(n)$ 是 $\Omega(g(n))$, 表述为“ $f(n)$ 是 $g(n)$ 的大 Ω ”。这个定义允许我们采用渐近的说法: 当给定一个常数因子时, 一个函数大于或等于另一个函数。

例题 3-15: $3n \log n - 2n$ 是 $\Omega(n \log n)$ 。

证明: 当 $n \geq 2$ 时, $3n \log n - 2n = n \log n + 2n(\log n - 1) \geq n \log n$ 。因此, 此时我们令 $c = 1$, $n_0 = 2$ 。 ■

大 Θ

此外, 有一个符号允许我们说: 当给定一个常数因子时, 两个函数的增长速率相同。如果 $f(n)$ 是 $O(g(n))$, 且 $f(n)$ 是 $\Omega(g(n))$, 即存在实常数 $c' > 0$ 、 $c'' > 0$ 和一个整型常数 $n_0 \geq 1$ 满足

$$c'g(n) \leq f(n) \leq c''g(n), \text{ 当 } n \geq n_0 \text{ 时},$$

我们就说 $f(n)$ 是 $\Theta(g(n))$, 描述为“ $f(n)$ 是 $g(n)$ 的大 Θ ”。

例题 3-16: $3n \log n + 4n + 5 \log n$ 是 $\Theta(n \log n)$ 。

证明: 当 $n \geq 2$ 时, $3n \log n \leq 3n \log n + 4n + 5 \log n \leq (3 + 4 + 5)n \log n$ 。 ■

3.3.2 比较分析

假设有两个算法都能解决同一个问题: 一个算法 A, 其运行时间为 $O(n)$; 另一个算法 B, 其运行时间 $O(n^2)$ 。哪一个算法更好呢? 我们知道 n 是 $O(n^2)$, 这就意味着算法 A 比算法

B 更具有渐近性，虽然当 n 的值较小时，算法 B 的运行时间可能低于 A。

我们使用大 O 符号依据渐近增长率来为函数排序。在下面的序列中，我们将 7 个函数按升序排序，即，假如函数 $f(n)$ 在函数 $g(n)$ 之前，那么 $f(n)$ 就是 $O(g(n))$ ： $1, \log n, n, n \log n, n^2, n^3, 2^n$ 。

我们举例说明一下表 3-2 中 7 个函数的增长速率（也可以参考 3.2.1 节的图 3-4）。

表 3-2 从基本函数的算法分析中选择的值

n	$\log n$	n	$n \log n$	n^2	n^3	2^n
8	3	8	24	64	512	256
16	4	16	64	256	4 096	65 536
32	5	32	160	1 024	32 768	4 294 967 296
64	6	64	384	4 096	262 144	1.84×10^{19}
128	7	128	896	16 384	2 097 152	3.40×10^{38}
256	8	256	2 048	65 536	16 777 216	1.15×10^{77}
512	9	512	4 608	262 144	134 217 728	1.34×10^{154}

在表 3-3 中，我们进一步举例说明渐近观点的重要性。该表探讨了允许一个输入实例的最大值，该实例由某个算法分别运行在 1 秒、1 分钟和 1 小的时候产生。该表显示了一个好的算法设计的重要性：缓慢渐近算法由于运行时间长从而被快速渐近算法所击败，尽管常数因子对于快速渐近算法而言可能更糟。

表 3-3 对于以微秒为单位的不同运行时间，一个问题分别在 1 秒、1 分钟和 1 小时所能解决的最大问题量

运行时间 (μs)	最大问题量 (n)		
	1 second	1 minute	1 hour
$400n$	2 500	150 000	9 000 000
$2n^2$	707	5 477	42 426
2^n	19	25	31

然而，好的算法设计的重要性不仅仅是在一台给定的计算机上高效地解决问题。如表 3-4 所示，即使硬件更新速度飞快，我们仍不能克服一个缓慢渐近算法的弊端。假设给定运行时间的算法运行在比以往计算机快 256 倍的计算机上，该表给出了在任意的常量时间所能解决的最大问题量。

表 3-4 在固定的时间，利用一台比以往计算机快 256 倍的计算机，(显示出)新的可供解决的最大问题量。每一个条目都是一个先前 m 倍的函数

运行时间 (μs)	新的最大问题量
$400n$	$256m$
$2n^2$	$16m$
2^n	$m + 8$

一些注意事项

这里就渐近符号做一些提醒。首先，注意大 O 符号和其他符号在使用时可能会被误导，因为它们“隐藏”的常数因子可能非常大。例如，虽然函数 $10^{100}n$ 是 $O(n)$ 是，但与运行时

间为 $10n \log n$ 的算法相比，虽然线性算法渐近速度更快，我们可能更倾向于选择运行时间为 $O(n \log n)$ 的算法。之所以这样，是因为常数 10^{100} 被称为“天文数字”，在观测宇宙时，许多天文学家一致认为该数字是原子数目的上限。所以，我们不可能得到一个像输入大小一样大的现实问题，因此，在使用大 O 符号时，我们应该注意被“隐藏”的常数因子和低阶项。

上述观测引发了这样的问题：什么是“快速”算法。一般来说，任何算法的运行时间为 $O(n \log n)$ （在给定一个合理的常数因子的情况下），都应被认为是高效的，甚至运行时间为 $O(n^2)$ 的算法在一些情形下，比如 n 很小时，也被认为是快速的。但如果算法的运行时间为 $O(2^n)$ ，则大多数情况不会被认为是高效的。

指数运行时间

有一个著名的关于国际象棋发明者的故事。他要求国王在象棋的第一个格只需支付 1 粒米，第二格 2 粒，第三格 4 粒，第四格 8 粒，以此类推。如果使用编程技巧编写一个程序来精确计算国王应支付的米粒数量，这将会是一个有趣的测试。

如果必须在高效和不高效算法之间划清界限，那么很自然，多项式运行时间和指数运行时间将会是一个明显的区别。也就是说，区分运行时间 $O(n^c)$ 是否为快速算法，只需看常数 c 是否满足 $c > 1$ ；区分运行时间 $O(b^n)$ 是否为快速算法，只需看常数 b 是否满足 $b > 1$ 。本节讨论的许多概念也应该看作“盐粒”，因为运行时间为 $O(n^{100})$ 的算法可能不被认为是“高效”的。即便如此，多项式运行时间和指数运行时间的区别仍被认为是健壮易处理的方式。

3.3.3 算法分析示例

既然我们用大 O 符号能进行算法分析，接下来给出使用该符号来描述一些简单算法的运行时间的若干示例。此外，为了和之前的约定保持一致，我们将介绍本章给出的 7 个函数是如何被用于描述算法实例的运行时间的。

在本节中，我们不再使用伪代码，而是给出完整的能够实现的 Python 代码。我们用 Python 的 list 类自然地表示数组的值。在第 5 章，我们将深入研究 Python 的 list 类以及该类所提供的各种方法的效率。在本节中，我们仅仅介绍几种方法来讨论它们的效率。

常量时间操作

给出一个 Python 的 list 类的实例，将其命名为 data，调用函数 len(data)，在固定的时间内对其进行评估。这是一个非常简单的算法，因为对于每一个列表，list 类包含一个能记录列表当前长度的实例变量。这就使得该算法能立即得出列表的长度，而不用再花时间迭代计算列表中的每个元素。使用渐近符号，我们说函数的运行时间为 $O(1)$ ，也就是说，函数的运行时间是独立于列表长度 n 的。

Python 的 list 类的另一个重要特征是能使用整数索引 j 写出 data[j] 来访问列表中的任意元素。因为 Python 列表是基于数组序列执行的，列表中的元素存储在连续的内存块内。之所以能搜索到列表中的第 j 个元素，不是靠迭代列表中的元素得到的，而是通过验证索引，并把该索引作为底层数组的偏移量得到的。反过来，对于某一元素，计算机硬件支持基于内存地址的常量时间访问。因此，我们说 Python 列表的 data[j] 元素的运行时间被估计为 $O(1)$ 。

回顾在序列中找最大数的问题

在开始下一个示例之前，我们先来回顾一下代码段 3-1 中的 find_max 算法，即在列表

中找最大值。在命题 3-7 中，我们得出该算法的运行时间为 $O(n)$ 。这符合我们之前的分析：语法 `data[0]` 的初始化运行时间为 $O(1)$ 。该循环执行 n 次，在每次循环中，都执行一次比较，可能也会执行一次赋值语句（以及维持循环变量）。最后，我们注意到 Python 返回语句机制运行时间也为 $O(1)$ 。综上所述，我们得出算法 `find_max` 的运行时间为 $O(n)$ 。

进一步分析找最大值算法

关于 `find_max` 算法，有一个更有趣的问题：我们要更新多少次当前“最大”值？在最坏的情况下，即给出的顺序按升序排列，最大值将会被重新赋值 $n - 1$ 次。但如果给出的是随机序列，即任何情况都可能出现，在这种情况下，如何预测最大值将会被更新多少次？要回答这个问题，应注意在循环的每一次迭代中，只有当前元素比以往所有元素都更大时才会更新当前最大值。如果给出的是随机序列，则第 j 个元素比前 j 个元素更大的概率是 $1/j$ （假定元素唯一）。因此，我们更新最大值（包括初始化）的预期次数是 $H_n = \sum_{j=1}^n 1/j$ ，这就是著名的 n 调和数。

这（见附录中的命题 B-16）表明 H_n 的运行时间是 $O(\log n)$ 。因此，在 `find_max` 算法中，基于随机序列，该算法的最大值被更新的预期次数是 $O(\log n)$ 。

前缀平均值

我们要讨论的下一个问题是著名的计算一个序列的前缀平均值。换句话说，给出一个包含 n 个数的序列 S ，我们想计算出序列 A ，该序列满足的条件为：当 $j = 0, \dots, n - 1$ 时， $A[j]$ 是 $S[0], \dots, S[j]$ 的平均值，即

$$A[j] = \frac{\sum_{i=0}^j S[i]}{j+1}$$

在经济学和统计学中，有很多计算前缀平均值的方法。比如，给出一个公共资金的每年收益，并把这些收益从过去到现在依次排列，投资者往往关注最近一年、三年或五年等的年平均收益。同样，给出一连串的日常网络使用日志，网站管理者可能希望能追踪不同时期的平均使用趋势。我们将分析三种能用于解决这些问题的方法，且该三种方法的运行时间截然不同。

二次 - 时间算法

为了计算前缀平均值，我们给出第一个算法（如代码段 3-2 所示），并将其命名为 `prefix_average1`。该算法使用内部循环计算部分和，因而能独立计算出序列 A 的每一个元素。

代码段 3-2 算法 `prefix_average1`

```

1 def prefix_average1(S):
2     """Return list such that, for all j, A[j] equals average of S[0], ..., S[j]."""
3     n = len(S)
4     A = [0] * n                      # create new list of n zeros
5     for j in range(n):
6         total = 0                    # begin computing S[0] + ... + S[j]
7         for i in range(j + 1):
8             total += S[i]
9         A[j] = total / (j+1)          # record the average
10    return A

```

为了分析算法 `prefix_average1`，我们对每步执行情况进行讨论。

- 在本节开始处已给出 $n = \text{len}(S)$, 且执行时间固定。
- 语句 $A = [0] * n$ 用于创建和初始化 Python 列表, 列表长度为 n , 每个元素值为 0。因每个元素都执行相同次数的原子操作, 故该算法的运行时间为 $O(n)$ 。
- for 循环有两层嵌套, 分别由计数器 j 和 i 独自约束。外层循环被计数器 j 约束, j 从 0 增长到 $n - 1$, 共执行 n 次。因此, 语句 $\text{total} = 0$ 和 $A[j] = \text{total}/(j + 1)$ 各被执行 n 次。这表明这两条语句加上 j 在此范围的执行, 使得原子操作的次数按比例增长到 n , 即其运行时间为 $O(n)$ 。
- 内层循环被计数器 i 约束, 执行 $j + 1$ 次, 具体执行次数取决于外层循环 j 的值。因此, 内层循环中的语句 $\text{total} += S[i]$ 共执行 $1 + 2 + 3 + \dots + n$ 次。通过回顾命题 3-3, 我们知道 $1 + 2 + 3 + \dots + n = n(n + 1)/2$, 这就表明内层循环的语句使得该算法运行时间为 $O(n^2)$ 。对于和计数器 i 相关的原子操作, 也可以做类似的论证, 其运行时间为 $O(n^2)$ 。

将上述三项运行时间相加, 即可得出执行算法 prefix_average1 的执行时间。第一项和第二项的运行时间为 $O(n)$, 第三项的运行时间为 $O(n^2)$ 。通过简单运用命题 3-9, 得出算法 prefix_average1 的运行时间为 $O(n^2)$ 。

接下来介绍第二种计算前缀平均值的算法 prefix_average2, 如代码段 3-3 所示。

代码段 3-3 算法 prefix_average2

```

1 def prefix_average2(S):
2     """Return list such that, for all j, A[j] equals average of S[0], ..., S[j]."""
3     n = len(S)
4     A = [0] * n           # create new list of n zeros
5     for j in range(n):
6         A[j] = sum(S[0:j+1]) / (j+1)  # record the average
7     return A

```

该方法本质上和算法 prefix_average1 一样, 都属于高级算法, 只是不再使用内层循环, 转而使用单一表达式 $\text{sum}(S[0:j + 1])$ 来计算部分和 $S[0] + \dots + S[j]$ 。利用 sum 函数极大地简化了算法的规模, 但是否对效率有影响值得思考。从渐近的角度来说, 没有比该算法更好的了。虽然表达式 $\text{sum}(S[0:j + 1])$ 看起来似乎是一条指令, 但它却是一个函数调用, 并能评估出该函数在算法中的运行时间为 $O(j + 1)$ 。从技术上讲, 这一句计算 $S[0:j + 1]$ 运行时间也为 $O(j + 1)$, 因为它构造了一个新的实例存储列表。因此算法 prefix_average2 的运行时间仍被一系列步骤所决定, 这些步骤按比例运行时间为 $1 + 2 + 3 + \dots + n$, 因此仍为 $O(n^2)$ 。

线性时间算法

接下来给出最后一个算法 prefix_average3, 如代码段 3-4 所示。

就像前两个算法一样, 我们热衷于对每个 j 计算前缀和 $S[0] + S[1] + \dots + S[j]$, 并在代码中以 total 表示, 以便能够进一步计算前缀平均值 $A[j] = \text{total}/(j + 1)$ 。不过, 与前两个算法不同的是该算法更高效。

代码段 3-4 算法 prefix_average3

```

1 def prefix_average3(S):
2     """Return list such that, for all j, A[j] equals average of S[0], ..., S[j]."""
3     n = len(S)
4     A = [0] * n           # create new list of n zeros
5     total = 0             # compute prefix sum as S[0] + S[1] + ...

```

```

6   for j in range(n):
7       total += S[j]           # update prefix sum to include S[j]
8       A[j] = total / (j+1)    # compute average based on current sum
9   return A

```

在前两个算法中，对每一个 j ，都要对前缀和重新进行计算。因每一个 j 都需要 $O(j)$ 的运行时间，从而导致该算法运行时间变为二次。在算法 `prefix_average3` 中，我们动态保存当前的前缀和，用 `total + S[j]` 高效计算 $S[0] + S[1] + \dots + S[j]$ ，这里 `total` 的值就等于先前算法循环执行到 j 时的和 $S[0] + S[1] + \dots + S[j - 1]$ 。对算法 `prefix_average3` 运行时间的分析如下：

- 初始化变量 n 和 `total`，用时 $O(1)$ 。
- 初始化列表 `A`，用时 $O(n)$ 。
- 只有一个 `for` 循环，用计数器 j 来约束。计数器在循环范围内持续迭代，使得 `total` 用时 $O(n)$ 。
- j 从 0 到 $n - 1$ ，循环体被执行 n 次。因此，语句 `total += S[j]` 和 `A[j] = total/(j + 1)` 各被执行 n 次。因为这两条语句每次迭代用时 $O(1)$ ，所以共用时 $O(n)$ 。

通过对上述四项求和便可得出算法 `prefix_average3` 的运行时间。第一项是 $O(1)$ ，剩余三项是 $O(n)$ 。通过对命题 3-9 的简单运用，得出 `prefix_average3` 的运行时间为 $O(n)$ ，比二次算法 `prefix_average1` 和 `prefix_average2` 运行效率更高。

三集不相交

假设我们给出三个序列 A 、 B 、 C 。假定任一序列没有重复值，但不同序列间可以重复。三集不相交问题就是确定三个序列的交集是否为空，即不存在元素 x 满足 $x \in A$ 、 $x \in B$ 且 $x \in C$ 。代码段 3-5 给出了一个简单的 Python 函数来确定这个性质。

代码段 3-5 算法 `disjoint1` 测试三集不相交

```

1 def disjoint1(A, B, C):
2     """Return True if there is no element common to all three lists."""
3     for a in A:
4         for b in B:
5             for c in C:
6                 if a == b == c:
7                     return False      # we found a common value
8     return True                # if we reach this, sets are disjoint

```

这个简单的算法将遍历三个序列任一组可能的三个值并且确定这些值是否相等。假如最初序列每一个长度都为 n ，在最坏情况下，该函数的运行时间为 $O(n^3)$ 。

我们可以用一个简单的观测来提高渐近性。一旦在循环 B 中发现此时的元素 a 和 b 不相等，再去遍历 C 为了找三个相等的数，则就浪费时间了。在代码段 3-6 中，利用观测思想，给出了解决该问题的改进方案。

代码段 3-6 算法 `disjoint2` 测试三集不相交

```

1 def disjoint2(A, B, C):
2     """Return True if there is no element common to all three lists."""
3     for a in A:
4         for b in B:
5             if a == b:          # only check C if we found match from A and B
6                 for c in C:
7                     if a == c:      # (and thus a == b == c)
8                         return False  # we found a common value
9     return True                # if we reach this, sets are disjoint

```

在改进方案中，如果运气好，则不仅能节省时间。对于 disjoint2，我们声明在最坏情况下的运行时间为 $O(n^2)$ 。这里要考虑许多二次对 (a, b) 。假如 A 和 B 均为没有重复值的序列，最多会有 $O(n)$ 。因此，最内层的循环 C 最多执行 n 次。

为了计算总的运行时间，我们检测每一行代码的执行时间。for 循环在 A 上需要运行 $O(n)$ ，在 B 上共需要 $O(n^2)$ ，因为该循环被执行在 n 个不同的时间段。预计语句 $a == b$ 的运行时间为 $O(n^2)$ 。剩下的运行时间取决于找到多少匹配的 (a, b) 对。因为我们已经注意到，最多有 n 对，因此 for 循环在 C 上以及循环体内的执行最多用时 $O(n^2)$ 。通过规范运用命题 3-9，得出总的运行时间为 $O(n^2)$ 。

元素唯一性

与三集不相交紧密相关的便是元素唯一性问题。前面我们给出三个集合并假定任一集合内元素不重复。在元素唯一性问题中，我们给出一个有 n 个元素的序列 S ，求该集合内的所有元素是否都彼此不同。

代码段 3-7 用于测试元素唯一性的算法 unique1

```

1 def unique1(S):
2     """Return True if there are no duplicate elements in sequence S."""
3     for j in range(len(S)):
4         for k in range(j+1, len(S)):
5             if S[j] == S[k]:
6                 return False           # found duplicate pair
7     return True                  # if we reach this, elements were unique

```

我们对此问题的第一个解决方案便是采用一个简单的迭代算法。在代码段 3-7 中给出函数 unique1，用于解决元素唯一性问题。该函数通过遍历所有下标 $j < k$ 的不同组合，检查是否有任一组合两元素相等。该算法使用两层循环，外层循环的第一次迭代致使内层循环 $n - 1$ 次迭代，外层循环的第二次迭代致使内层循环 $n - 2$ 次，以此类推。因此，在最坏情况下，该函数的运行时间按比例增长到

$$(n - 1) + (n - 2) + \cdots + 2 + 1$$

通过命题 3-3，我们得出总运行时间仍为 $O(n^2)$ 。

以排序作为解决问题的工具

解决元素唯一性问题更优的一个算法是以排序作为解决问题的工具。在此情况下，通过对序列的元素进行排序，我们确定任何相同元素将会被排在一起。因此，为了确定是否有重复值，我们所要做的就是遍历该排序的序列，查看是否有连续的重复值。该算法的一个 Python 实现方法如代码段 3-8 所示：

代码段 3-8 用于测试元素唯一性的算法 unique2

```

1 def unique2(S):
2     """Return True if there are no duplicate elements in sequence S."""
3     temp = sorted(S)           # create a sorted copy of S
4     for j in range(1, len(temp)):
5         if temp[j-1] == temp[j]:
6             return False          # found duplicate pair
7     return True                  # if we reach this, elements were unique

```

如 1.5.2 节所述，内置函数 sorted 的基本功能是对原始列表的元素进行一次有序排序后产生的一个新列表。该函数保证在最坏情况下其运行时间为 $O(n \log n)$ ；详见第 12 章对常见

排序算法的讨论。一旦数据被排序，下面的循环运行时间就变为 $O(n)$ ，因此算法 unique2 的总运行时间为 $O(n \log n)$ 。

3.4 简单的证明技术

有时，我们会想做关于一个算法的声明，如显示它是正确的或者它的运行速度很快。为了使声明更加严谨，我们必须使用数学语言。为了证实这样的说法，我们必须对声明加以证明。幸运的是，有几种简单的方法可以做到这一点。

3.4.1 示例

有些声明的一般形式为：“在集合 S 中，存在具有性质 P 的元素 x ”。为了证明这个说法，我们只需要生成一个特定的元素 x ，它在集合 S 中并具有性质 p 。同样，一些难以置信的声明的一般形式为：“在集合 S 中，任一元素 x 都具有性质 P ”。为了证明这种声明是错误的，我们只需生成一个特定的元素 x ，它在集合 S 中并不具有性质 P 。这样的实例就是一个反例。

例题 3-17: Amongus 教授声称，当 i 是大于 1 的整数时，每个形如 $2^i - 1$ 的数是一个素数。Amongus 教授的说法是错误的。

证明: 为了证明 Amongus 教授是错误的，我们找出一个反例。幸运的是，我们不需要找太大的数，例如 $2^4 - 1 = 15 = 3 \times 5$ 。 ■

3.4.2 反证法

另一种证明技术涉及否定的使用。两个主要的这类方法是逆否命题和矛盾的使用。逆否命题方法的使用就像透过镜子的反面进行观察。为了证明命题“如果 p 为真，那么 q 为真”，我们使用命题“如果 q 非真，那么 p 非真”来代替。从逻辑上讲，这两个命题是相同的，但是后者，也就是第一个命题的逆否命题，可能更容易思考。

例题 3-18: 设 a 和 b 都是整数，如果 ab 是偶数，那么 a 是偶数或者 b 是偶数。

证明: 为了证明这个结论，我们来考虑它的逆否命题，“如果 a 是奇数并且 b 是奇数，那么 ab 也是奇数”。所以，假设 $a = 2j + 1$ 和 $b = 2k + 1$ ，那么， $ab = 4jk + 2j + 2k + 1 = 2(2jk + j + k) + 1$ ，其中， j 和 k 为整数，可证 ab 为奇数。 ■

除了显示逆否命题证明方式的使用，在前面的例子中还含有一个德摩根定律 (DeMorgan's Law) 的应用。这个定律能帮助我们处理否定，因为它说明了“ p 或者 q ”的否定形式是“非 p 并非 q ”。同样，它也说明了“ p 并 q ”的否定形式是“非 p 或者非 q 。”

矛盾

另一个反证方法是通过矛盾来证明，这也常常涉及德摩根定律的使用。通过矛盾的方法进行证明时，我们建立一个声明 q 是真的，首先假设 q 是假的，然后显示出由这个假设导致的矛盾（如 $2 \neq 2$ 或 $1 > 3$ ）。通过这样的一个矛盾，我们可以得出如果 q 是假的，那么没有一致的情况存在，所以 q 必须是真的。当然，为了得出这个结论，在假设 q 是假的之前，必须确保我们的情况是一致的。

例题 3-19: 设 a 和 b 都是整数，如果 ab 是奇数，那么 a 是奇数并且 b 也是奇数。

证明: 设 ab 是奇数。我们希望得到 a 是奇数并且 b 也是奇数。所以，希望出现与假设相反的矛盾，即假设 a 是偶数或者 b 是偶数。事实上，为了不失一般性，我们甚至可以假设 a 是偶数的情况（因为 b 的情况是对称的）。然后我们设 $a = 2j$ ，其中 j 是整数。因此， $ab =$

($2j$), $b = 2(jb)$, 得出 ab 是偶数。但这是一个矛盾: ab 不能既是奇数又是偶数。因此, a 是奇数并且 b 也是奇数。 ■

3.4.3 归纳和循环不变量

我们所做出的关于运行时间或空间约束的大多数声明都包括一个整数参数 n (通常表示该问题的“大小”的直观概念)。此外, 大多数的这些声明相当于“对于所有 $n \geq 1$, $q(n)$ 为真”这样的语句。由于这是一个关于无限组数字的声明, 我们不能以直接的方式穷尽证明这一点。

归纳

但是, 通过使用归纳的方法, 我们通常可以证明上述声明是正确的。这种方法表明, 对于任何特定的 $n \geq 1$, 有一个有限序列的证明, 从已知为真的东西开始, 最终得出 $q(n)$ 为真的结论。具体地说, 通过证明当 $n = 1$ 时, $q(n)$ 为真, 我们开始用归纳法证明 (可能还有其他一些值 $n = 2, 3, \dots, k$, k 为一个常数)。然后, 我们证明当 $n > k$ 时归纳“步骤”为真, 即表明“对于所有 $j < n$, 如果 $q(j)$ 为真, 那么 $q(n)$ 为真。”将这两块部分合起来即可完成归纳的证明。

命题 3-20: 考虑斐波那契函数 $F(n)$, 它定义 $F(1) = 1, F(2) = 2, F(n) = F(n - 2) + F(n - 1)$, 其中 $n > 2$ (参见 1.8 节), 由此推断 $F(n) < 2^n$ 。

证明: 通过归纳法, 我们将证明上述命题是正确的。

递推的基础: ($n \leq 2$)。 $F(1) = 1 < 2 = 2^1$ 并且 $F(2) = 2 < 4 = 2^2$ 。

递推的依据: ($n > 2$)。对于所有 $n' < n$, 假设结论是正确的。考虑 $F(n)$ 。因为 $n > 2$, $F(n) = F(n - 2) + F(n - 1)$ 。而且, 因为 $n - 2$ 和 $n - 1$ 都小于 n , 我们可以应用归纳假设 (有时称为“递归假说”) 得到 $F(n) < 2^{n-2} + 2^{n-1}$, 因为

$$2^{n-2} + 2^{n-1} < 2^{n-1} + 2^{n-1} = 2 \cdot 2^{n-1} = 2^n$$

让我们做另外一个归纳论证, 这次是我们之前已经看到的事实。

命题 3-21: 它和命题 3-3 的定义相同。

$$\sum_{i=1}^n i = \frac{n(n+1)}{2}$$

证明: 我们将用归纳法证明这个等式。

递推的基础: $n = 1$ 最简单的, 如果 $n = 1$, 那么 $1 = n(n + 1)/2$ 。

递推的依据: $n \geq 2$ 对于所有 $n' < n$ 。考虑 n 。

$$\sum_{i=1}^n i = n + \sum_{i=1}^{n-1} i$$

通过归纳假说, 有

$$\sum_{i=1}^n i = n + \frac{(n-1)n}{2}$$

我们可以把上式简化为

$$n + \frac{(n-1)n}{2} = \frac{2n + n^2 - n}{2} = \frac{n^2 + n}{2} = \frac{n(n+1)}{2}$$

对于所有 $n \geq 1$ 的情况, 我们有时会感到证明一些事情为真的任务让我们不堪重负。但是, 我们应该记住归纳法的具体步骤。这表明, 对于任何特定的 n , 通过一系列的有限、逐

步的证明，从已知为真的东西开始，最终得出关于 n 的真实性。总之，归纳论证为一系列直接证明提供了模板。

循环不变量

在本节中，我们最后讨论的证明方法是循环不变量。为了证明一些关于循环的语句 L 是正确的，我们依据一系列较小的语句 L_0, L_1, \dots, L_k 来定义 L ，其中：

- 1) 在循环开始前，最初要求 L_0 是真的。
- 2) 如果在迭代 j 之前 L_{j-1} 为真，那么在迭代 j 之后 L_j 也会为真。
- 3) 最后的语句 L_k 意味着想要证明的语句 L 为真。

让我们以一个简单的用到循环不变量参数的例子来证明算法的正确性。尤其是，用一个循环不变量来证明函数 `find`（参见代码段 3-9），

找出出现在序列 S 中元素 val 的最小索引值。

代码段 3-9 寻找一个给定的元素在 Python 列表中出现的第一个索引值的算法

```

1 def find(S, val):
2     """Return index j such that S[j] == val, or -1 if no such element."""
3     n = len(S)
4     j = 0
5     while j < n:
6         if S[j] == val:
7             return j           # a match was found at index j
8         j += 1
9     return -1

```

为了说明 `find` 函数为真，我们归纳定义一系列的语句 L_j ，来推断算法的正确性。具体地说，在 `while` 循环迭代 j 的开始，我们认为以下的叙述为真：

L_j : val 不等于序列 S 中的第一个元素 j 的任何一个

循环的第一次迭代开始时，这种声明为真，因为 j 是 0，序列 S 中的第一个 0 中没有元素（这样一个非常真实的声明被称为空存）。在第 j 次迭代中，我们比较元素 val 和元素 $S[j]$ ，如果这两种元素是相等的，那么返回索引值 j ，在这种情况下，这显然是正确的并且可以完成这个算法。如果两个元素 val 和 $S[j]$ 是不相等的，那么我们可以多一个不等于 val 的元素，并把索引值 j 加一。因此，对于这个新的索引值 j ，这个声明 L_j 会变成真，于是在下一次迭代开始时它为真。如果 `while` 循环终止而没有返回序列 S 中的任何一个索引值，则有 $j = n$ 。也就是说， L_n 为真——在序列 S 中没有与 val 相等的元素。因此，该算法准确的返回 -1 ，以指示在序列 S 中没有元素 val 。

3.5 练习

请访问 www.wiley.com/college/goodrich 以获得练习帮助。

巩固

- R-3.1 画出函数 $8n$ 、 $4n \log n$ 、 $2n^2$ 、 n^3 和 2^n 的图形，其中 x 轴和 y 轴均为对数刻度。也就是说，若函数 $f(n)$ 的值为 y ，则 x 坐标为 $\log(n)$ ， y 坐标为 $\log(y)$ ，其中， (x, y) 为一个点。
- R-3.2 算法 A 和 B 执行的操作个数分别为 $8n \log(n)$ 和 $2n^2$ 。确定 n_0 ，满足：当 $n \geq n_0$ 时，A 比 B 更优。
- R-3.3 算法 A 和 B 执行的操作个数分别为 $40n^2$ 和 $2n^3$ 。确定 n_0 ，满足：当 $n \geq n_0$ 时，A 比 B 更优。
- R-3.4 请给出一个函数示例，该函数在双对数坐标轴和标准坐标轴中的图形相同。
- R-3.5 试解释：在双对数坐标轴中，斜率为 c 的函数 n^c ，为何其图形为一条直线？

R-3.6 对于任意的正整数 n , $0 \sim 2n$ 范围内, 所有偶数的和是多少?

R-3.7 证明下面两个语句等价:

1) 算法 A 的运行时间为 $O(f(n))$ 。

2) 在最坏的情况下, 算法 A 的运行时间为 $O(f(n))$ 。

R-3.8 根据渐近增长速率对下面的函数进行排序。

$$\begin{array}{lll} 4n \log n + 2n & 2^{10} & 2^{\log n} \\ 3n + 100 \log n & 4n & 2n \\ n^2 + 10n & n^3 & n \log n \end{array}$$

R-3.9 证明: 若 $d(n)$ 为 $O(f(n))$, 对于任意的常数 $a > 0$, $ad(n)$ 为 $O(f(n))$ 。

R-3.10 证明: 若 $d(n)$ 为 $O(f(n))$, $e(n)$ 为 $O(g(n))$, 则 $d(n)e(n)$ 为 $O(f(n)g(n))$ 。

R-3.11 证明: 若 $d(n)$ 为 $O(f(n))$, $e(n)$ 为 $O(g(n))$, 则 $d(n) + e(n)$ 为 $O(f(n) + g(n))$ 。

R-3.12 证明: 若 $d(n)$ 为 $O(f(n))$, $e(n)$ 为 $O(g(n))$, 则 $d(n) - e(n)$ 不一定为 $O(f(n) - g(n))$ 。

R-3.13 证明: 若 $d(n)$ 为 $O(f(n))$, $f(n)$ 为 $O(g(n))$, 则 $d(n)$ 为 $O(g(n))$ 。

R-3.14 证明: $O(\max\{f(n), g(n)\}) = O(f(n) + g(n))$ 。

R-3.15 证明: 当且仅当 $g(n)$ 为 $\Omega(f(n))$ 时, $f(n)$ 为 $O(g(n))$ 。

R-3.16 证明: 若 $p(n)$ 为 n 的多项式, 则 $\log p(n)$ 为 $O(\log n)$ 。

R-3.17 证明: $(n+1)^5$ 为 $O(n^5)$ 。

R-3.18 证明: 2^{n+1} 为 $O(2^n)$ 。

R-3.19 证明: n 为 $O(n \log n)$ 。

R-3.20 证明: n^2 为 $\Omega(n \log n)$ 。

R-3.21 证明: $n \log n$ 为 $\Omega(n)$ 。

R-3.22 证明: 若 $f(n)$ 为正的、非递减函数, 且恒大于 1, 则 $\lceil f(n) \rceil$ 为 $O(f(n))$ 。

R-3.23 对代码段 3-10 中给出的函数 example1, 使用 n 对其运行时间做大 O 描述。

R-3.24 对代码段 3-10 中给出的函数 example2, 使用 n 对其运行时间做大 O 描述。

R-3.25 对代码段 3-10 中给出的函数 example3, 使用 n 对其运行时间做大 O 描述。

R-3.26 对代码段 3-10 中给出的函数 example4, 使用 n 对其运行时间做大 O 描述。

R-3.27 对代码段 3-10 中给出的函数 example5, 使用 n 对其运行时间做大 O 描述。

R-3.28 在下表中, 对于任一函数 $f(n)$ 和时间 t , 若针对问题 P 的算法运行 $f(n)$ 微秒, 确定在 t 时间内 P 被解决的最大的 n 为多少 (其中一项已给出结果)。

	1 Second	1 Hour	1 Month	1 Century
$\log n$	$\approx 10^{300\,000}$			
n				
$n \log n$				
n^2				
2^n				

R-3.29 算法 A 对包含 n 个元素的序列中的每个元素都执行 $O(\log n)$ 的计算时间。算法 A 的最坏运行时间是多少?

R-3.30 给出一个包含 n 个元素的序列 S , 算法 B 在 S 中随机选择 $\log n$ 个元素, 并对每个元素都执行 $O(n)$ 的计算时间。算法 B 的最坏运行时间是多少?

- R-3.31 给出一个包含 n 个整数的序列 S , 算法 C 对 S 中的每个偶数执行 $O(n)$ 的计算时间, 每个奇数执行 $O(\log n)$ 的运算时间。算法 C 的最好和最坏运行时间分别是多少?

代码段 3-10 用于做分析的一些示例算法

```

1 def example1(S):
2     """Return the sum of the elements in sequence S."""
3     n = len(S)
4     total = 0
5     for j in range(n):           # loop from 0 to n-1
6         total += S[j]
7     return total
8
9 def example2(S):
10    """Return the sum of the elements with even index in sequence S."""
11    n = len(S)
12    total = 0
13    for j in range(0, n, 2):      # note the increment of 2
14        total += S[j]
15    return total
16
17 def example3(S):
18    """Return the sum of the prefix sums of sequence S."""
19    n = len(S)
20    total = 0
21    for j in range(n):           # loop from 0 to n-1
22        for k in range(1+j):       # loop from 0 to j
23            total += S[k]
24    return total
25
26 def example4(S):
27    """Return the sum of the prefix sums of sequence S."""
28    n = len(S)
29    prefix = 0
30    total = 0
31    for j in range(n):
32        prefix += S[j]
33        total += prefix
34    return total
35
36 def example5(A, B):          # assume that A and B have equal length
37    """Return the number of elements in B equal to the sum of prefix sums in A."""
38    n = len(A)
39    count = 0
40    for i in range(n):           # loop from 0 to n-1
41        total = 0
42        for j in range(n):       # loop from 0 to n-1
43            for k in range(1+j):   # loop from 0 to j
44                total += A[k]
45        if B[i] == total:
46            count += 1
47    return count

```

- R-3.32 给出一个包含 n 个元素的序列 S , 算法 D 对每个元素 $S[i]$ 都调用算法 E。算法 E 被调用时, 对于每个元素 $S[i]$, 运行时间为 $O(i)$ 。算法 D 的最坏运行时间是多少?
- R-3.33 Al 和 Bob 在争论各自的算法。Al 认为自己的运行时间为 $O(n \log n)$ 的算法总是比 Bob 的运行时间为 $O(n^2)$ 的算法要快。为了解决这个问题, 他们做了一系列实验。令 Al 沮丧的是, 他们发现: 若 $n < 100$, 则运行时间为 $O(n^2)$ 的算法较快, 仅当 $n \geq 100$ 时, 运行时间为 $O(n \log n)$ 的算法才更快。请解释为何会这样。
- R-3.34 有一个著名的城市 (这里可能是无名的), 城市居民享有这样的声誉: 仅当一顿饭的质量是他们有生以来所体验的最好的, 他们才会享受这顿饭; 否则, 就厌恶它。假设饭菜的质量均匀分布于一个人的一生, 描述此城市的居民对饭菜满意的预期次数。

创新

- C-3.35 假设在 $O(n \log n)$ 时间内可以完成对 n 个数字的排序，证明在 $O(n \log n)$ 的时间内能够解决三集不相交问题。
- C-3.36 描述一种有效算法：在大小为 n 的序列中找到前十个最大元素。你的算法的运行时间是多少？
- C-3.37 给出一个正函数 $f(n)$ 的例子，满足： $f(n)$ 的运行时间既不是 $O(n)$ 也不是 $\Omega(n)$ 。
- C-3.38 证明： $\sum_{i=1}^n i^2$ 是 $O(n^3)$ 。
- C-3.39 证明： $\sum_{i=1}^n i / 2^i < 2$ 。（提示：根据几何级数逐项求和。）
- C-3.40 证明：若 $b > 1$ ，且为常数， $\log_b f(n)$ 为 $\Theta(\log f(n))$ 。
- C-3.41 描述一种算法：从 n 个数字中找到最小值和最大值，要求比较次数少于 $3n/2$ 次。（提示：首先，选出一组候选的最小值和一组候选的最大值。）
- C-3.42 Bob 开发了一个 Web 网站，仅仅把 URL 给了他的 n 个朋友，并把这 n 个朋友从 1 到 n 进行了编号。他告诉编号为 i 的朋友，他或她对该 Web 网站最多访问 i 次。现在，Bob 有一个计数器 C ，能够记录此网站的总访问量（但不能辨别访问者是谁）。 C 最小为多少，能够使得 Bob 知道他其中一个朋友的访问次数已达上限？
- C-3.43 当 n 为奇数时，对命题 3-3 给出一个类似于图 3-3b 的可视化的理由。
- C-3.44 在计算机网络中，通信安全是极其重要的，许多网络协议实现安全的一种策略便是加密信息。确保信息在网络中安全传输的典型加密方案基于这样一个事实：没有已知的有效算法来分解大的整数。因此，若使用一个大的素数 p 表示秘密信息，我们就可以在网络中传输数字 $r = p \cdot q$ ，这里 q 为另一个大的素数，且 $q > p$ ，用于作为密钥。假如窃听者在网络中获取到传输数字 r ，但若要找出秘密信息 p ，则必须分解 r 。
使用分解法找信息时，若不知道密钥 q ，则非常困难。为了搞清楚原因，先来考虑如下天真的分解算法：
- ```
for p in range(2,r):
 if r % p == 0: # if p divides r
 return 'The secret message is p!'
```
- 1) 假设窃听者采用上述算法，并拥有一台计算机，能够在 1ms（1s 的百万分之一）时间内，对两个整数做一次除法，其中每个整数都超过 100 位。若传输信息  $r$  有 100 位，估计在最坏情况下解读信息  $p$  需要多少时间。
- 2) 上述算法的最坏时间复杂度是什么？因为算法输入的仅仅是一个大的数字  $r$ ，假设输入大小  $n$  表示存储  $r$  所需要的字节位数，即  $n = \lceil (\log_2 r)/8 \rceil + 1$ ，且每次除法运行时间为  $O(n)$ 。
- C-3.45 序列  $S$  包含  $n - 1$  个唯一的整数，整数范围为  $[0, n - 1]$ ，即此范围内有一个数不属于  $S$ 。设计一个  $O(n)$  - 时间算法找出此数。除了  $S$  本身所占的内存外，仅允许你使用  $O(1)$  的额外空间。
- C-3.46 AI 说他能证明：在一个羊群内所有绵羊都是同一种颜色。
- 基本情况：一只绵羊。很明显，同种颜色即是它本身。
- 归纳步骤：一群绵羊，共  $n$  只。取出绵羊  $a$ ，通过归纳，剩余的  $n - 1$  只都是同一种颜色。现在，把绵羊  $a$  放回去，并取出一只不同的绵羊  $b$ 。通过归纳，剩余的  $n - 1$  只绵羊（包括绵羊  $a$ ）都是同一种颜色。因此，一个羊群内的所有绵羊都是同一种颜色。AI 的“理由”错在哪里？
- C-3.47 设  $S$  为包含  $n$  条直线的集合， $n$  条直线位于同一平面，任意两条直线都不平行，任意三条直线都不相交于一点。归纳证明： $S$  中的直线能够确定  $\Theta(n^2)$  个交点。

C-3.48 考虑下述的“理由”：Fibonacci 函数， $F(n)$ （见命题 3-20）是  $O(n)$ 。

基本情况 ( $n \leq 2$ )： $F(1) = 1$ ,  $F(2) = 2$ 。

归纳步骤 ( $n > 2$ )：假设当  $n' < n$  时，结论正确。考虑  $n$ ,  $F(n) = F(n - 2) + F(n - 1)$ 。通过归纳， $F(n - 2)$  是  $O(n - 2)$  且  $F(n - 1)$  是  $O(n - 1)$ 。之后，根据在 R-3.11 中得出的一致性原理， $F(n)$  是  $O((n - 2) + (n - 1))$ 。因此， $F(n)$  的运行时间是  $O(n)$ 。

该“理由”错在哪里？

C-3.49 考虑 Fibonacci 函数， $F(n)$ （见命题 3-20）。归纳证明  $F(n)$  的运行时间是  $\Omega((3/2)^n)$ 。

C-3.50 设  $p(x)$  为  $n$  次多项式，即  $p(x) = \sum_{i=0}^n a_i x^i$ 。

1) 给出一个简单  $O(n^2)$  时间的算法计算  $p(x)$ 。

2) 通过对  $x^i$  进行更有效的计算，给出一个简单  $O(n \log n)$  时间的算法计算  $p(x)$ 。

3) 现在考虑对  $p(x)$  进行改写：

$$p(x) = a_0 + x(a_1 + x(a_2 + x(a_3 + \cdots + x(a_{n-1} + x a_n) \cdots)))$$

这就是著名的霍纳法。使用大 O 符号，描述该方法执行的算术运算次数。

C-3.51 证明求和公式  $\sum_{i=1}^n \log i$  的运行时间是  $O(n \log n)$ 。

C-3.52 证明求和公式  $\sum_{i=1}^n \log i$  的运行时间是  $\Omega(n \log n)$ 。

C-3.53 一位坏国王有  $n$  瓶酒，一个间谍对其中的一瓶下了毒。不幸的是，他们都不知道哪一瓶已被下毒。毒药非常致命，仅仅稀释一滴，配成 10 亿 : 1 的溶液，也能将人杀死。虽然如此，但若要毒药起作用，也需要一整个月的时间。设计一个方案，在一个月的时间内，通过  $O(n \log n)$  个测试者，准确确定哪个酒瓶已被下毒。

C-3.54 序列  $S$  包含  $n$  个整数，整数范围为  $[0, 4n]$ ，允许有重复值。描述一个有效算法，确定在  $S$  中值为  $k$  的整数出现次数最多。该算法的运行时间是多少？

## 项目

P-3.55 对 3.3.3 节的三个算法 prefix\_average1、prefix\_average2 和 prefix\_average3 执行实验分析。将它们的运行时间形象化为一个输入大小的函数，并以双对数图的形式表示。

P-3.56 执行实验分析：比较代码段 3-10 中给出的函数的相对运行时间。

P-3.57 执行实验分析：验证 Python 的 sorted 方法平均运行时间为  $O(n \log n)$  这一假设。

P-3.58 对解决元素唯一性问题的三个算法 unique1、unique2 和 unique3 中的任意一个执行实验分析，确定最大值  $n$ ，使得给出的算法运行时间小于等于 1min。

## 扩展阅读

大 O 符号关于其合适的使用已在参考文献<sup>[19, 49, 63]</sup>中给出了一些评论。Knuth<sup>[64, 63]</sup> 使用  $f(n) = O(g(n))$  进行定义，但是，说“相等”仅仅是“一种方式”。我们选择了更标准的相等概念，即把大 O 符号看作集合，这一思想来自于 Brassard<sup>[19]</sup>。Vitter 和 Flajolet<sup>[101]</sup> 的文章中提到了热衷于学习平均情况分析的读者。对于其他一些数学工具，参见附录 B。

# 递 归

在计算机程序中，描述迭代的一种方法是使用循环，比如在 1.4.2 节中描述的 Python 语言的 while 循环和 for 循环。另一种完全不同的迭代实现方法就是递归。

递归是一种技术，这种技术通过一个函数在执行过程中一次或者多次调用其本身，或者通过一种数据结构在其表示中依赖于相同类型的结构更小的实例。在艺术和自然界中有很多递归的例子。例如分形图是自然方面的递归。一个在艺术中应用递归的例子是俄罗斯套娃。每个娃娃要么是实心的，要么是空心的，并且空心的娃娃里面包含了另一个俄罗斯套娃。

在计算中，递归提供了用于执行迭代任务的优雅并且强大的替代方案。事实上，一些编程语言（例如 Scheme、Smalltalk）不明确支持循环结构，而是直接依靠递归来表示迭代。大多数现代编程语言都通过和传统函数调用相同的机制支持函数的递归调用。当函数的一次调用需要进行递归调用时，该调用被挂起，直到递归调用完成。<sup>⊖</sup>

在数据结构和算法的研究中，递归是一种重要的技术。我们将在本书的后面几个章节中多次使用递归（尤其是第 8 章和第 12 章）。在本章中，我们将从以下四个递归使用例证开始，并给出了每个例证的 Python 实现。

- 阶乘函数（通常表示为  $n!$ ）是一个经典的数学函数，它有一个固有的递归定义。
- 英式标尺具有的递归模式是分形结构的一个简单例子。
- 二分查找是最重要的计算机算法之一。在一个拥有数十亿以上条目的数据集中，它能让我们有效地定位所需的某个值。
- 计算机的文件系统有一个递归结构，在该结构中，目录能够以任意深度嵌套在其他目录上。递归算法被广泛用于探索和管理这些文件系统。

我们接下来讨论如何进行一个递归算法的运行时间的形式化分析，并且讨论在定义递归时一些潜在的缺陷。在内容的选择上，我们提供了更多的递归算法的例子，强调了一些常见的设计形式。

## 4.1 说明性的例子

### 4.1.1 阶乘函数

为了说明递归的机制，我们首先介绍一个计算阶乘函数的值的简单数学示例。一个正整数  $n$  的阶乘表示为  $n!$ ，它被定义为整数从 1 到  $n$  的乘积。如果  $n = 0$ ，那么按照惯例  $n!$  被定义为 1。更正式的定义是，对于任何整数  $n \geq 0$ ，

$$n! = \begin{cases} 1 & n = 0 \\ n \times (n-1) \times (n-2) \cdots 3 \times 2 \times 1 & n \geq 1 \end{cases}$$

例如， $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$ 。阶乘函数很重要，其结果等于  $n$  个元素全排列的个数。例如，三个字符 a、b 和 c 有  $3! = 3 \times 2 \times 1 = 6$  种不同的排列方式：abc、acb、bac、bca、cab

<sup>⊖</sup> 函数调用时局部变量和执行位置信息压栈，调用结束后恢复。递归调用也是这样的过程。——译者注

和 cba。

阶乘函数有一个固有的递归定义。可以看到  $5! = 5 \times (4 \times 3 \times 2 \times 1) = 5 \times 4!$ 。通常，对于一个正整数  $n$ ，我们可以定义  $n! = n \times (n - 1)!$ 。这个递归定义可以形式化为

$$n! = \begin{cases} 1 & n = 0 \\ n \times (n-1)! & n \geq 1 \end{cases}$$

在许多递归定义中，这个定义是很典型的。首先，它包含一个或多个基本情况，根据定量，这些基本情况通常被直接定义。在这个定义中， $n = 0$  是一个基本情况。它还包含一个或多个递归情况，这些情况的定义服从被定义函数的定义。

### 阶乘函数的递归实现

递归不仅是一个数学符号，也可以用于设计一个阶乘函数，如代码段 4-1 所示。

代码段 4-1 阶乘函数的递归实现

```
1 def factorial(n):
2 if n == 0:
3 return 1
4 else:
5 return n * factorial(n-1)
```

这个函数不使用任何显式循环。迭代是通过函数的重复递归调用来实现的。在这个定义中没有循环，因为函数每被调用一次，它的参数就会变小一次，当达到基本情况的时候，递归调用就会停止。

我们用递归跟踪的形式来说明一个递归函数的执行过程。跟踪的每个条目代表着一个递归调用。每一个新的递归函数调用用一个向下的箭头指向新的调用表示。函数返回时，用一个弯曲的箭头表示，并将返回值标在箭头的旁边。图 4-1 所示为一个对阶乘函数进行跟踪的示例。

递归跟踪密切反映了编程语言对于递归的执行。在 Python 中，每当一个函数（递归或其他方式）被调用时，都会创建一个被称为活动记录或框架的结构来存储信息，这些信息是关于函数调用的过程的。这个活动记录包含一个用来存储函数调用的参数和局部变量的命名空间（参见 1.10 节命名空间的讨论），以及关于在这个函数体中当前正在执行的命令的信息。

如果一个函数的执行导致嵌套函数的调用，那么前者调用的执行将被挂起，其活动记录将存储源代码中的位置，这个位置是被调用函数返回后将继续执行的控制流。该过程可以在标准情况下一个函数调用另一个不同的函数，或用在一个函数调用自身的递归情况下。关键的一点是对于每个有效的调用都有一个不同的活动记录。

### 4.1.2 绘制英式标尺

在计算阶乘的情况下，也可以采用循环来实现迭代，不一定必须采用递归的方法。举一个更复杂的使用递归的例子，考虑如何绘制出一个典型的英式标尺的刻度。对于每一英寸 (in, 1in = 2.54cm)，我们用一个数字标签做上刻度标记。我们表示刻度的长度并且指定一个

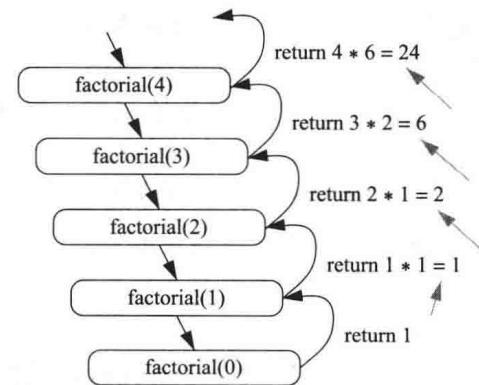


图 4-1 `factorial(5)` 函数调用的递归跟踪

英寸作为主刻度线的长度。在整个英寸刻度之间，标尺包含一系列较小的刻度线，如  $1/2$  英寸、 $1/4$  英寸，等等。当间隔的大小减少了一半时，刻度线的长度也减 1。图 4-2 展示了几个这样的具有不同主刻度长度的标尺（虽然不是按比例绘制）。

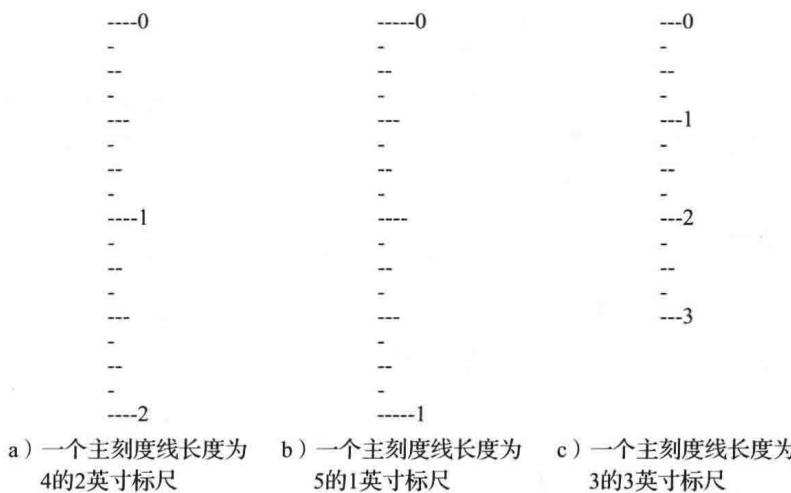


图 4-2 一个英式标尺绘制的三个示例输出

### 绘制标尺的递归方法

英式标尺模式是分形的一个简单示例，也就是具有在各级放大的自递归结构的形状。考虑在图 4-2b 中所示的刻度线长度为 5 的标尺，忽略包含 0 和 1 的刻度线，考虑如何绘制这些刻度线之间的刻度序列。中央刻度线（在  $1/2$  英寸处）的长度为 4。观察中央刻度线上面和下面两个部分的刻度，发现它们是相同的，并且每部分有一个长度为 3 的中央刻度线。

一般情况下，中央刻度线长度  $L \geq 1$  的刻度间隔的组成如下：

- 一个中央刻度线长度为  $L - 1$  的刻度间隔
- 一个长度为  $L$  的单独的刻度线
- 一个中央刻度线长度  $L - 1$  的刻度间隔

虽然可以使用一个迭代过程绘制这样的标尺（参见练习 P-4.25），但是这个任务用递归完成更加容易。如代码段 4-2 所示，代码实现包括三个函数。主函数 `draw_ruler` 管理整个标尺的构建。它的参数指定标尺的总长度以及主刻度线的长度。功能函数 `draw_line` 用指定数量的破折号绘制一个单独的刻度线（并且在刻度线之后打印一个可选的字符串标签）。

最重要的工作是由递归函数 `draw_interval` 来完成的。这个函数根据刻度间隔中中央刻度线的长度来绘制刻度间隔之间副刻度线的序列。根据本节开始时列出的  $L \geq 1$  的规律，当  $L = 0$  这个基本情况，不再绘制任何东西。对于  $L \geq 1$ ，第一步和最后一步都是通过递归调用 `draw_interval(L-1)` 进行的，中间的步骤是通过递归调用函数 `draw_interval(L)` 进行的。

### 代码段 4-2 绘制一个标尺的函数的递归实现

```

1 def draw_line(tick_length, tick_label=''):
2 """Draw one line with given tick length (followed by optional label)."""
3 line = '-' * tick_length

```

```

4 if tick_label:
5 line += ' ' + tick_label
6 print(line)
7
8 def draw_interval(center_length):
9 """ Draw tick interval based upon a central tick length. """
10 if center_length > 0: # stop when length drops to 0
11 draw_interval(center_length - 1) # recursively draw top ticks
12 draw_line(center_length) # draw center tick
13 draw_interval(center_length - 1) # recursively draw bottom ticks
14
15 def draw_ruler(num_inches, major_length):
16 """ Draw English ruler with given number of inches, major tick length. """
17 draw_line(major_length, '0') # draw inch 0 line
18 for j in range(1, 1 + num_inches): # draw interior ticks for inch
19 draw_interval(major_length - 1) # draw interior ticks for inch
20 draw_line(major_length, str(j)) # draw inch j line and label

```

### 用递归追踪说明标尺的绘制

用一个递归追踪可以使递归函数 `draw_interval` 的执行变得可视化。然而，`draw_interval` 的追踪比阶乘函数追踪的例子更复杂，因为每个实例进行了两次递归调用。为了说明这一点，我们将以排列的形式展示递归跟踪，这个形式非常类似一个文档大纲，如图 4-3 所示。

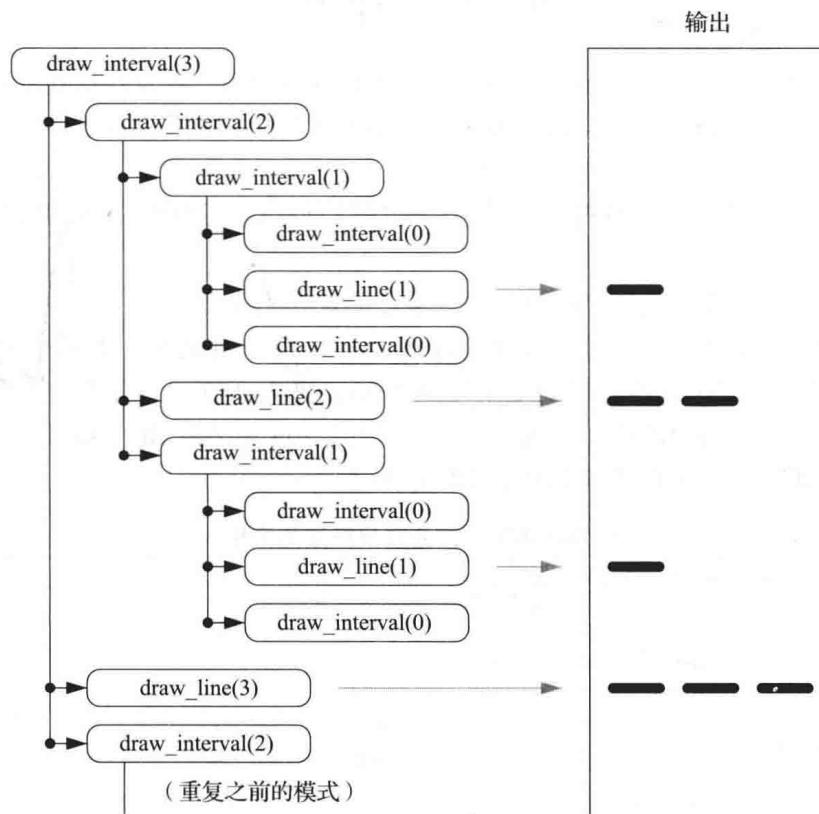


图 4-3 对于调用 `draw_interval(3)` 的局部递归追踪。对于调用 `draw_interval(2)` 的第二个追踪模式没有予以展示，但它与第一个是相同的

### 4.1.3 二分查找

本节将介绍一个典型的递归算法——二分查找。该算法用于在一个含有  $n$  个元素的有序

序列中有效地定位目标值。这是最重要的计算机算法之一，也是我们经常顺序存储数据（见图 4-4）的原因。

|   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    |
|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6  | 7  | 8  | 9  | 10 | 11 | 12 | 13 | 14 | 15 |
| 2 | 4 | 5 | 7 | 8 | 9 | 12 | 14 | 17 | 19 | 22 | 25 | 27 | 28 | 33 | 37 |

图 4-4 值以索引序列顺序存储，比如 Python 列表，顶部的数字是索引

当序列无序时，寻找一个目标值的标准方法是使用循环来检查每一个元素，直至找到目标值或检查完数据集的每个元素。这就是所谓的顺序查找算法。因为最坏的情况下每个元素都需要检查，这个算法的时间复杂度是  $O(n)$ （即线性的时间）。

当序列有序并且可通过索引访问时，有一个更有效的算法（直觉上，想想你如何手工完成这个任务！）。对于任意索引  $j$ ，我们知道在索引  $0, \dots, j - 1$  上存储的所有值都小于索引  $j$  上的值，并且在索引  $j + 1, \dots, n - 1$  上存储的所有值都大于或等于索引  $j$  上的值。在搜索目标时，这种观察使我们能够迅速定位目标值。在查找时，如果不能排除一个元素与目标值相匹配，那么称序列的这个元素为候选项。该算法维持两个参数  $\text{low}$  和  $\text{high}$ ，这样可使所有候选条目的索引位于  $\text{low}$  和  $\text{high}$  之间。首先， $\text{low} = 0$  和  $\text{high} = n - 1$ 。然后我们比较目标值和中间值候选项，即索引项  $[\text{mid}]$  的数据。

$$\text{mid} = \lfloor (\text{low} + \text{high}) / 2 \rfloor$$

考虑以下三种情况：

- 如果目标值等于  $[\text{mid}]$  的数据，然后找到正在寻找的值，则查找成功并且终止。
- 如果目标值  $< [\text{mid}]$  的数据，对前半部分序列重复这一过程，即索引的范围从  $\text{low}$  到  $\text{mid} - 1$ 。
- 如果目标值  $> [\text{mid}]$  的数据，对后半部分序列重复这一过程，即索引的范围从  $\text{mid} + 1$  到  $\text{high}$ 。

如果  $\text{low} > \text{high}$ ，说明索引范围  $[\text{low}, \text{high}]$  为空，则查找不成功。

该算法被称为二分查找。代码段 4-3 给出了一个 Python 实例，其算法执行过程的说明如图 4-5 所示。而顺序查找的时间复杂度是  $O(n)$ ，更为高效的二分查找的时间复杂度是  $O(\log n)$ 。这是一个显著的改进，因为假设  $n$  是十亿， $\log n$  仅为 30。（对于二分查找运行时间的问题，我们将在 4.2 节命题 4-2 做正式的分析。）

#### 代码段 4-3 二分查找算法的实现

---

```

1 def binary_search(data, target, low, high):
2 """Return True if target is found in indicated portion of a Python list.
3
4 The search only considers the portion from data[low] to data[high] inclusive.
5 """
6 if low > high:
7 return False # interval is empty; no match
8 else:
9 mid = (low + high) // 2
10 if target == data[mid]: # found a match
11 return True
12 elif target < data[mid]:
13 # recur on the portion left of the middle
14 return binary_search(data, target, low, mid - 1)
15 else:
16 # recur on the portion right of the middle
17 return binary_search(data, target, mid + 1, high)

```

---

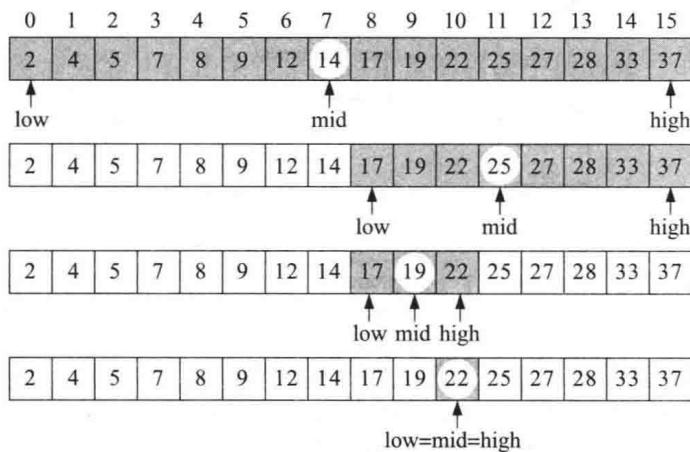


图 4-5 对于目标值 22 的二分查找的例子

#### 4.1.4 文件系统

现代操作系统用递归的方式来定义文件系统目录（有时也称为“文件夹”）。也就是说，一个文件系统包括一个顶级目录，这个目录的内容包括文件和其他目录，其他目录又可以包含文件和其他目录，以此类推。虽然必定会有一些基本的目录只包含文件而没有下一级目录，但是操作系统允许嵌套任意深度的目录（只要在内存中有足够的空间）。图 4-6 所示即为此类文件系统的一部分。

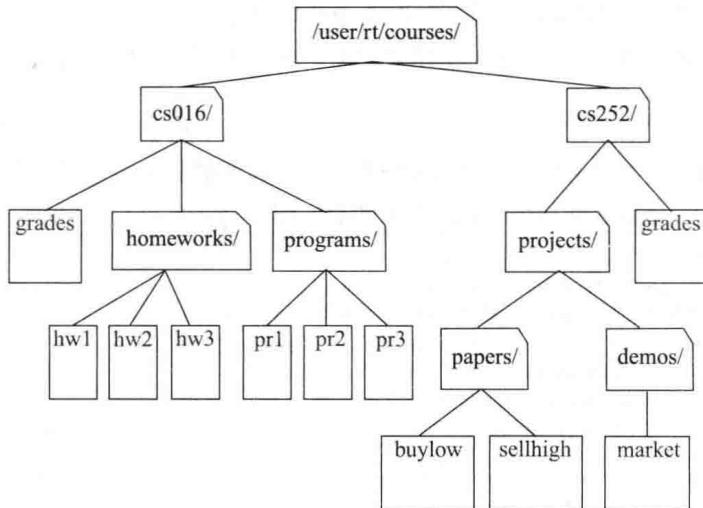


图 4-6 用文件系统的一部分展示嵌套结构

考虑到文件系统表示的递归特性，操作系统中许多常见的行为，比如目录的复制或删除，都可以很方便地用递归算法来实现。在本节中，我们考虑这样一个算法：计算嵌套在一个特定目录中的所有文件和目录的总磁盘使用情况。

为了说明空间的使用情况，图 4-7 显示了样例文件系统中所有条目使用的磁盘空间。我们对每个条目所使用的即时磁盘空间以及由该条目和所有嵌套目录所使用的累计磁盘空间加以区分。例如，目录 cs016 仅仅使用了 2K 的即时空间，但使用了 249K 的累计磁盘空间。

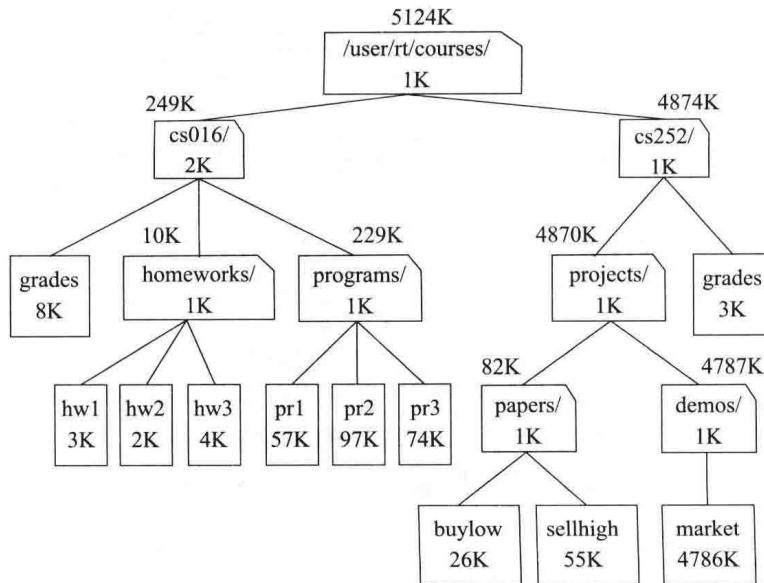


图 4-7 和图 4-6 的文件系统局部有相同的部分，但增加了用于描述所使用磁盘空间量的注释。每个文件或目录的图标里面是该模块镜像直接使用空间的总量。每个目录的图标上面是由该目录及其所有（递归）内容使用的累计磁盘空间

一个条目的累计磁盘空间可以用简单的递归算法来计算。它等于条目使用的直接磁盘空间加上直接存储在该条目中所有条目使用的累计磁盘空间之和。例如，cs016 的累计磁盘空间是 249K，因为它本身使用 2K 的磁盘空间，grades 使用 8K 的累计磁盘空间，homeworks 使用 10K 的累计磁盘空间，programs 使用 229K 的累计磁盘空间。代码段 4-4 给出了这个算法的伪代码。

#### 代码段 4-4 计算嵌套在一个文件系统条目中的累积磁盘空间使用的算法。Size 函数返回一个条目的即时磁盘空间

---

**Algorithm** DiskUsage(path):  
**Input:** A string designating a path to a file-system entry  
**Output:** The cumulative disk space used by that entry and any nested entries  
 total = size(path) {immediate disk space used by the entry}  
**if** path represents a directory **then**  
     **for** each child entry stored within directory path **do**  
         total = total + DiskUsage(child) {recursive call}  
**return** total

---

#### Python 的操作系统模块

为了给出一个计算磁盘使用情况的递归算法 Python 实例，我们需要借助 Python 的操作系统模块，在程序执行的过程中，该模块提供了强大的与操作系统交互的工具。这是一个丰富的函数库，但我们只需要以下四个函数：

- `os.path.getsize(path)`

返回由字符串路径（例如：/user/rt/courses）标识的文件或者目录使用的即时磁盘空间大小（单位是字节）。

- `os.path.isdir(path)`

如果字符串路径指定的条目是一个目录，则返回 True；否则，返回 false。

- `os.listdir(path)`

返回一个字符串列表，它是字符串路径指定的目录中所有条目的名称。在样例文件系统中，如果参数是 `/user/rt/courses`，那么返回字符串列表 `['cs016', 'cs252']`。

- `os.path.join(path, filename)`

生成路径字符串和文件名字符串，并使用一个适当的操作系统分隔符在两者之间分隔（例如：Unix/Linux 系统中的 '/' 字符和 Windows 系统中的 '\' 字符）。返回表示文件完整路径的字符串。

### Python 实例

通过使用 `os` 模块，现在我们把代码段 4-4 中的算法转换成代码段 4-5 中的 Python 实例。

**代码段 4-5 报告一个文件系统磁盘使用情况的递归函数**

---

```

1 import os
2
3 def disk_usage(path):
4 """Return the number of bytes used by a file/folder and any descendants."""
5 total = os.path.getsize(path) # account for direct usage
6 if os.path.isdir(path): # if this is a directory,
7 for filename in os.listdir(path): # then for each child:
8 childpath = os.path.join(path, filename) # compose full path to child
9 total += disk_usage(childpath) # add child's usage to total
10
11 print ('{0:<7}'.format(total), path) # descriptive output (optional)
12 return total # return the grand total

```

---

### 递归跟踪

为了产生另一种格式的递归跟踪，我们在 Python 实例加入了额外的 `print` 语句（代码段 4-5 的第 11 行）。该输出的准确格式有意模仿由一个名为 `du` 的典型的 Unix/Linux 实用程序（对于“disk usage”）生成的输出。如图 4-8 所示，它报告一个目录及其中嵌套的所有内容使用的磁盘空间的总量，并能生成详细的报告。

在图 4-7 所示的样例文件系统上执行时，`disk_usage` 函数的实例产生一个相同的结果。在算法执行期间，对于文件系统的每一个条目，正好使用一次递归调用。因为 `print` 语句是在递归调用之前执行的，所以图 4-8 所示的输出反映了递归调用完成的顺序。需要特别强调的是，在可以计算和报告所有包含的条目的累计磁盘空间之前，我们必须完成嵌套在该条目之下的所有条目的递归调用。例如，我们在计算包含 `grades`、`homeworks` 和 `programs` 条目的递归调用完成后，才知道条目 `/user/rt/courses/cs016` 的累计磁盘空间大小。

|      |                                                 |
|------|-------------------------------------------------|
| 8    | /user/rt/courses/cs016/grades                   |
| 3    | /user/rt/courses/cs016/homeworks/hw1            |
| 2    | /user/rt/courses/cs016/homeworks/hw2            |
| 4    | /user/rt/courses/cs016/homeworks/hw3            |
| 10   | /user/rt/courses/cs016/homeworks                |
| 57   | /user/rt/courses/cs016/programs/pr1             |
| 97   | /user/rt/courses/cs016/programs/pr2             |
| 74   | /user/rt/courses/cs016/programs/pr3             |
| 229  | /user/rt/courses/cs016/programs                 |
| 249  | /user/rt/courses/cs016                          |
| 26   | /user/rt/courses/cs252/projects/papers/buylow   |
| 55   | /user/rt/courses/cs252/projects/papers/sellhigh |
| 82   | /user/rt/courses/cs252/projects/papers          |
| 4786 | /user/rt/courses/cs252/projects/demos/market    |
| 4787 | /user/rt/courses/cs252/projects/demos           |
| 4870 | /user/rt/courses/cs252/projects                 |
| 3    | /user/rt/courses/cs252/grades                   |
| 4874 | /user/rt/courses/cs252                          |
| 5124 | /user/rt/courses/                               |

图 4-8 图 4-7 中所示文件系统的磁盘使用情况报告——由 Unix/Linux 实用程序 `du`（带命令行选项 `-ak`）或代码段 4-5 中的 `disk_usage` 函数生成

## 4.2 分析递归算法

在第3章中，我们介绍了一种分析算法效率的数学方法，该方法基于算法执行的基本操作次数的估计值。我们使用符号（比如 big-Oh）来概括操作次数和问题输入大小之间的关系。本节将演示如何执行这种类型的递归算法的时间复杂度分析。

对于递归算法，我们将解释基于函数的特殊激活并且被执行的每个操作，该函数在被执行期间管理控制流。换句话说，对于每次函数调用，我们只解释被调用的主体内执行的操作的数目。然后，通过在每个单独调用过程中执行的操作数的总和，即所有调用次数，我们可以解释被作为递归算法的一部分而执行的操作的总数。（顺便说一句，这也是我们分析非递归函数的方式。这些非递归函数从它们的函数体中调用其他函数）

为了说明这种分析的模式，我们回顾一下 4.1.1 ~ 4.1.4 节介绍的四个递归算法：阶乘函数、绘制一个英式标尺、二分查找以及文件系统累计磁盘空间大小的计算。一般来说，在识别有多少递归调用发生，以及每个调用的参数化可以用来估计其调用的主体内发生的基本操作次数方面，我们可以借助于递归追踪提供的客观事实。但是，每一个递归算法都具有独特的结构和形式。

### 计算阶乘

正如 4.1.1 节所描述的，分析计算阶乘的函数的效率是比较容易的。图 4-1 给出了阶乘函数的一个示例递归追踪。为了计算  $\text{factorial}(n)$ ，共执行了  $n + 1$  次函数调用。参数从第一次调用时的  $n$  下降到第二次调用时的  $n - 1$ ，以此类推，直至达到参数为 0 时的基本情况。

代码段 4-1 给出了函数体的检测，同样清楚的是，阶乘的每个调用执行了一个常数级别的运算。因此，我们得出这样的结论：计算  $\text{factorial}(n)$  的操作总次数是  $O(n)$ ，因为有  $n + 1$  次函数的调用，所以每次调用占的操作次数为  $O(1)$ 。

### 绘制一个英式标尺

在 4.1.2 节分析英式标尺的应用程序中，我们考虑共有多少行输出这一基本问题。该输出是通过初始调用 `draw_interval(c)` 产生的，其中  $c$  表示中央刻度线长度。这是该算法的整体效率的合理基准，因为输出的每一行是基于一个对 `draw_line` 函数的调用，以及对 `draw_interval` 的每次非零参数递归调用恰好产生一个对 `draw_line` 的直接调用。

通过检验源代码和递归追踪可以获得直观认识。我们知道对 `draw_interval(c)` ( $c > 0$ ) 的一个调用产生两个对 `draw_interval(c - 1)` 的调用和一个单独的 `draw_line` 的调用。我们将依赖这些客观事实来证明以下的声明。

**命题 4-1：**对于  $c \geq 0$ ，调用 `draw_interval(c)` 函数刚好产生  $2^c - 1$  行输出。

**证明：**通过归纳法（参见 3.4.3 节），我们给出了这种声明的正式证明。事实上，归纳法是用于证明递归过程正确性和有效性的自然数学技术。在标尺的例子中，我们注意到，`draw_interval(0)` 的应用程序没有输出，并以此作为证明的基本情况。■

更一般的是，通过调用 `draw_interval(c)` 函数打印的行数比通过调用 `draw_interval(c - 1)` 函数产生的行数的两倍还多 1——因为在两个这样的递归调用之间打印一个中心线。通过归纳法，我们计算出行数  $1 + 2(2^{c-1} - 1) = 1 + 2^c - 2 = 2^c - 1$ 。

这个证明表明，一个更严格的被称为递归方程的数学工具可用于分析递归算法的运行时间。在 12.2.4 节对递归排序算法的分析中，我们会讨论这种技术。

### 执行二分查找

如在 4.1.3 节提到的，考虑到二分查找算法的运行时间，我们观察到二分查找方法的每

次递归调用中被执行的基本操作次数是恒定的。因此，运行时间与执行递归调用的数量成正比。我们会证明在对含有  $n$  个元素的队列进行二分查找过程中至多进行  $\lfloor \log n \rfloor + 1$  次递归调用，并且得出以下声明。

**命题 4-2：**对于含有  $n$  个元素的有序序列，二分查找算法的时间复杂度是  $O(\log n)$ 。

**证明：**为了证明这一命题，一个重要的事实是：在每次递归调用中，需要被查找的候选条目的数量是由一个值给出的。这个值为

$$\text{high} - \text{low} + 1$$

此外，每次递归调用之后，剩下的候选条目的数量至少减少一半。具体来讲，从  $\text{mid}$  的定义可知，剩下的候选条目的数量是

$$(\text{mid} - 1) - \text{low} + 1 = \left\lfloor \frac{\text{low} + \text{high}}{2} \right\rfloor - \text{low} \leq \frac{\text{high} - \text{low} + 1}{2}$$

或者是

$$\text{high} - (\text{mid} + 1) + 1 = \text{high} - \left\lfloor \frac{\text{low} + \text{high}}{2} \right\rfloor \leq \frac{\text{high} - \text{low} + 1}{2}$$

候选条目最初为  $n$ ；在进行一次二分查找调用之后，它至多是  $n/2$ ；在进行第二次调用后，它至多为  $n/4$ ；以此类推。一般情况下，在进行第  $j$  次二分查找调用之后，剩下的候选条目的数量至多是  $n/2^j$ 。在最坏的情况下（一次不成功的查找），当没有更多的候选条目时递归调用停止。因此，进行递归调用的最大次数，有最小整数  $r$ ，使得

$$\frac{n}{2^r} < 1$$

换言之（回想一下，当对数底数是 2 时，省略对数底数）， $r > \log n$ 。因此，有  $r = \lfloor \log n \rfloor + 1$ ，这意味着二分查找的时间复杂度为  $O(\log n)$ 。 ■

### 计算磁盘空间使用情况

4.1 节最后一个递归算法是计算在文件系统的特定部分整体磁盘空间使用情况。为了描述所分析的“问题大小”，我们用  $n$  表示所考虑文件系统的特定部分的文件系统条目的数量。（例如，图 4-6 所示的文件系统有  $n(n=19)$  个条目）

为了描述 `disk_usage` 函数初始调用的累计时间开销，我们必须分析所执行的递归调用的总数以及在这些调用中执行的操作次数。

首先显示刚好有  $n$  次函数调用的递归过程，尤其是文件系统的相关部分的每个条目对应一次递归调用的过程。直观来讲，这是因为对于文件系统的特定条目  $e$  仅进行一次 `disk_usage` 调用，在代码段 4-5 的 `for` 循环中处理包含  $e$  的父目录时，将只检索一次该条目。

为了形式化地证明上述论证，我们可以定义每个条目的嵌套级别，比如定义起始条目的嵌套级别为 0，定义直接存储在该条目中所有条目的嵌套级别为 1，定义存储在这些条目中所有条目的嵌套级别为 2，以此类推。我们可以通过归纳法证明在嵌套等级为  $k$  的各条目上恰好有一个对 `disk_usage` 函数的递归调用。作为一种基本情况，当  $k = 0$  时，唯一进行的递归调用是初始调用。就归纳步骤来说，一旦知道在嵌套级别为  $k$  的每个条目上恰好只有一次递归调用，我们可以证明对于嵌套级别为  $k$  的条目下的条目  $e$ ，仅从处理包含  $e$  的  $k+1$  级条目的 `for` 循环中调用一次。

在确定了文件系统的每个条目有一个递归调用之后，我们回到对于算法整体计算时间的问题上来。或许我们认为在任何单一的函数调用上花  $O(1)$  的时间将是非常好的，但事实并

非如此。虽然可以在该条目上用固定数量的步骤调用函数 `os.path.getsize` 来直接计算的磁盘使用情况，但当这个条目是一个目录时，`disk_usage` 函数的主体包含一个 `for` 循环，将遍历这个目录包含的所有条目。在最坏的情况下，一个条目可能包含  $n - 1$  个其他条目。

基于这种推理，我们可以得出这样的结论：有  $O(n)$  个递归调用，并且每个调用运行的时间为  $O(n)$ ，从而导致总的运行时间为  $O(n^2)$ 。虽然这个时间上限在技术上是正确的，但它不是一个严格意义上的上限。值得注意的是，我们可以证明更强的约束：对于 `disk_usage` 函数的递归算法可以在  $O(n)$  的时间内完成！较弱的约束是悲观的，因为它假设了每个目录所有条目在最坏情况下的数量。虽然可能一些目录包含的条目数量与  $n$  成正比，但它们不可能每个都含有那么多的条目。为了证明这个更有力的声明，我们选择考虑在所有递归调用中 `for` 循环迭代的总数。我们断言刚好有  $n - 1$  个该循环的这种迭代。这一声明基于这样一个事实，即该循环的每次迭代进行一次对 `disk_usage` 函数的递归调用，并且已经得出结论，即对 `disk_usage` 函数共进行了  $n$  次调用（包括最初的调用）。因此，我们得出这样的结论：有  $O(n)$  次递归调用，每次递归调用在循环外部使用  $O(1)$  的时间，并且循环操作的总数是  $O(n)$ 。总结所有这些限制条件，操作的总数是  $O(n)$ 。

我们已经得出的观点比前面递归的例子更先进。有时可以通过考虑累积效应获得一系列操作更严格的约束，而不是假设每个操作都是最坏的情况，这种思想就是被叫作分期偿还的技术（在 5.3 节我们会看到更多的例子）。此外，文件系统是隐式地使用“树”这一数据结构的例子，磁盘使用（disk usage）算法实际是树遍历算法的一种表现。树是第八章的重点，并且关于磁盘使用（disk usage）算法时间复杂度是  $O(n)$  这一论据将在 8.4 节中树的遍历中加以推广。

### 4.3 递归算法的不足

虽然递归是一种非常强大的工具，但它也很容易被误用。在本节中，我们检查了几个问题，其中一个糟糕的递归实现导致严重的效率低下，并讨论了一些用于识别和避免这种陷阱的策略。

首先回顾 3.3.3 节的定义：元素唯一性问题。我们可以用下面的递归公式来确定序列中所有的  $n$  个元素是否都是唯一的。作为一种基本情况，当  $n = 1$  时，明显元素是唯一的。对于  $n \geq 2$ ，当且仅当第一个  $n - 1$  个元素是唯一的、最后的  $n - 1$  项是唯一的并且第一个元素和最后一个元素不同时，元素是唯一的（因为这是唯一一对子情况中没有被检查的元素）。代码段 4-6 给出了基于这种思想的递归实例，称其为 `unique3`（与第 3 章的 `unique1` 和 `unique2` 区分开来）。

代码段 4-6 测试元素唯一性的递归函数 `unique3`

---

```

1 def unique3(S, start, stop):
2 """Return True if there are no duplicate elements in slice S[start:stop]."""
3 if stop - start <= 1: return True # at most one item
4 elif not unique3(S, start, stop-1): return False # first part has duplicate
5 elif not unique3(S, start+1, stop): return False # second part has duplicate
6 else: return S[start] != S[stop-1] # do first and last differ?

```

---

不幸的是，这是一个效率非常低的递归使用。非递归部分的每次调用所使用的时间为  $O(1)$ ，所以总的运行时间将正比于递归调用的总数。为了分析这个问题，我们用  $n$  表示所考虑的条目总数，即  $n = stop - start$ 。

如果  $n = 1$ , 则 unique3 的运行时间为  $O(1)$ , 因为在这种情况下, 不进行递归调用。一般情况下, 最重要的发现是, 对于一个大小为  $n$  的问题, 对 unique3 函数的单一调用可能导致对两个大小为  $n - 1$  的问题的 unique3 函数调用。反过来, 这两个大小为  $n - 1$  的调用可能又产生 4 个大小为  $n - 2$  的调用 (各两个), 然后是 8 个大小为  $n - 3$  的调用, 以此类推。因此, 在最坏的情况下, 函数调用的总数由如下几何求和公式给出

$$1 + 2 + 4 + \cdots + 2^{n-1}$$

这等于是由命题 3-5 给出的。因此函数 unique3 的时间复杂度为  $O(2^n)$ 。难以置信, 这个函数解决元素唯一性问题的效率如此低下。其低效率不是因为使用递归, 而是缘于所使用的递归不佳这样一个事实, 这是我们在练习 C-4.11 中要解决的问题。

### 一个低效的计算斐波那契数的递归算法

在 1.8 节中, 我们介绍了生成斐波纳契数的过程, 可以递归地定义如下:

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_n &= F_{n-2} + F_{n-1} \quad \text{for } n > 1 \end{aligned}$$

恰巧, 基于上述定义的直接实现就是代码段 4-7 中所示的函数 bad\_fibonacci, 该函数通过执行两个非基本情况的递归调用来计算斐波纳契数。

#### 代码段 4-7 使用二分递归计算第 $n$ 个斐波那契数列

---

```

1 def bad_fibonacci(n):
2 """Return the nth Fibonacci number."""
3 if n <= 1:
4 return n
5 else:
6 return bad_fibonacci(n-2) + bad_fibonacci(n-1)

```

---

不幸的是, 这样的斐波那契数公式的直接实现会导致函数的效率非常低。以这种方式计算第  $n$  个斐波纳契数需要对这个函数进行指数级别的调用。具体来说, 用  $C_n$  表示在 bad\_fibonacci(n) 执行中进行的调用次数。然后, 我们可以得到以下的一系列值:

$$\begin{aligned} c_0 &= 1 \\ c_1 &= 1 \\ c_2 &= 1 + c_0 + c_1 = 1 + 1 + 1 = 3 \\ c_3 &= 1 + c_1 + c_2 = 1 + 1 + 3 = 5 \\ c_4 &= 1 + c_2 + c_3 = 1 + 3 + 5 = 9 \\ c_5 &= 1 + c_3 + c_4 = 1 + 5 + 9 = 15 \\ c_6 &= 1 + c_4 + c_5 = 1 + 9 + 15 = 25 \\ c_7 &= 1 + c_5 + c_6 = 1 + 15 + 25 = 41 \\ c_8 &= 1 + c_6 + c_7 = 1 + 25 + 41 = 67 \end{aligned}$$

如果遵循这个模式继续下去, 我们可以看到, 对于每两个连续的指标, 后者调用的数量将是前者的 2 倍以上。也就是说,  $c_4$  是  $c_2$  的两倍以上,  $c_5$  是  $c_3$  的两倍以上,  $c_6$  是  $c_4$  的两倍以上, 以此类推。因此  $c_n > 2^{n/2}$  意味着 bad\_fibonacci(n) 使得调用的总数是  $n$  的指数组。

### 一个高效的计算斐波那契数列的递归算法

我们之所以尝试使用这个不好的递归公式, 是因为第  $n$  个斐波那契数取决于前两个值, 即  $F_{n-2}$  和  $F_{n-1}$ 。但是请注意, 计算出  $F_{n-2}$  之后, 计算  $F_{n-1}$  的调用需要其自身递归调用以

计算  $F_{n-2}$ ，因为它不知道先前级别的调用中被计算的  $F_{n-2}$  的值。这是一个重复的操作。更糟的是，这两个调用都需要（重新）计算  $F_{n-3}$  的值， $F_{n-1}$  的计算也一样。正是这种滚雪球效应，导致 `bad_fibonacci` 函数有指数倍的运行时间。

我们可以更有效地使用递归来计算  $F_n$ ，这种递归的每次调用只进行一次递归调用。要做到这一点，我们需要重新定义函数的期望值。我们定义了一个递归函数，该函数返回一对连续的斐波那契数列 ( $F_n, F_{n-1}$ )，并且使用约定  $F_{n-1} = 0$ ，而不是让函数返回第  $n$  个斐波那契数这一单一数值。用返回一对连续的斐波那契数列代替返回一个值，虽然这似乎是一个更大的负担，但从递归这一级来看，通过这个额外的信息之后使得递归更容易继续这一进程（它可以让我们避免再计算第二个值，这个值在递归中是已知的）。代码段 4-8 给出了基于这种策略的一个实例。

代码段 4-8 使用线性递归计算第  $n$  个斐波那契数

---

```

1 def good_fibonacci(n):
2 """Return pair of Fibonacci numbers, F(n) and F(n-1)."""
3 if n <= 1:
4 return (n, 0)
5 else:
6 (a, b) = good_fibonacci(n-1)
7 return (a+b, a)

```

---

就效率而言，对于这个问题，效率低的递归和效率高的递归之间的区别就像黑夜和白天。`bad_fibonacci` 函数使用指数数量级的时间。我们认为函数 `good_fibonacci(n)` 使用的时间为  $O(n)$ 。每次对 `good_fibonacci(n)` 函数的递归调用都使参数  $n$  减小 1，因此，递归追踪包括一系列的  $n$  个函数调用。因为每个调用的非递归工作使用固定的时间，所以整体的运算执行在  $O(n)$  的时间内完成。

## Python 中的最大递归深度

在递归的误用中，另一个危险就是所谓的无限递归。如果每个递归调用都执行另一个递归调用，而最终没有达到一个基本情况，那我们就有一个无穷级数的此类调用。这是一个致命的错误。无限递归会迅速耗尽计算资源，这不仅是因为 CPU 的快速使用，而且是由于每个连续的调用会创建需要额外内存的活动记录。一个明显不合语法的递归示例如下：

```

def fib(n):
 return fib(n) # fib(n) equals fib(n)

```

然而，还有更微小的错误会导致无限递归。回顾我们在代码段 4-3 中二分查找的实现，在最后的情况下（第 17 行），我们在序列的右半部分，特别是索引从  $mid + 1$  到  $high$  这部分，进行一个递归调用。那一行反而被写成

```
return binary_search(data, target, mid, high) # note the use of mid
```

这可能导致一个无限递归。尤其是在搜索范围内的两个元素时，有可能在同一范围内进行递归调用。

程序员应该确保每个递归调用以某种方式逐步向基本情况发展（例如，通过使用随每次调用减少的参数值）。然而，为了避免无限递归，Python 的设计者做了一个有意的决定来限制可以同时有效激活的函数的总数。这个极限的精确值取决于 Python 分配，但典型的默认值是 1000。如果达到这个限制，Python 解释器就生成了一个 `RuntimeError` 消息：超过最大

递归深度 (maximum recursion depth exceeded)。

对于许多合法的递归应用, 1000 层嵌套函数的限制调用足够了。例如, `binary_search` 函数 (见 4.1.3 节) 的递归深度为  $O(\log n)$ , 所以要达到默认递归的限制, 需要有  $2^{1000}$  个元素 (远远超过宇宙中原子数量的估计值)。然而, 在下一节中, 我们将讨论一些递归深度与  $n$  成正比的算法。Python 在递归深度上的人为限制可能会破坏这些其他的合法计算。

幸运的是, Python 解释器可以动态地重置, 以更改默认的递归限制。这是用一个名为 `sys` 的模块来实现的, 该模块支持 `getrecursionlimit` 函数和 `setrecursionlimit` 函数。这些函数的使用示例如下:

```
import sys
old = sys.getrecursionlimit() # perhaps 1000 is typical
sys.setrecursionlimit(1000000) # change to allow 1 million nested calls
```

## 4.4 递归的其他例子

本章的剩余部分将给出使用递归的其他例子。我们通过考虑在一个激活的函数体内开始的递归调用的最大数量来组织我们的介绍。

- 如果一个递归调用最多开始一个其他递归调用, 我们称之为线性递归 (linear recursion)。
- 如果一个递归调用可以开始两个其他递归调用, 我们称之为二路递归 (binary recursion)。
- 如果一个递归调用可以开始三个或者更多其他递归调用, 我们称之为多重递归 (multiple recursion)。

### 4.4.1 线性递归

如果一个递归函数被设计成使得所述主体的每个调用至多执行一个新的递归调用, 这被称为线性递归。到目前为止, 在我们已经看到的递归函数中, 阶乘函数的实现 (见 4.1.1 节) 和 `good_fibonacci` 函数 (见 4.3 节) 是线性递归鲜明的例子。更有趣的是, 尽管在名称中有 “binary”, 二分查找算法 (见 4.1.3 节) 也是线性递归的一个例子。二分查找的代码 (见代码段 4-3) 包括一个具有两个分支的情况分析, 这两个分支产生递归调用, 但在函数体的一个具体执行期间只有其中一个调用可以被执行。

正如在 4.1.1 节描绘的阶乘函数 (见图 4-1) 一样, 线性递归定义的一个结果是任何递归追踪将表现为一个单一的调用序列。注意, 线性递归术语反映递归追踪的结构, 而不是运行时间的渐近分析, 例如, 我们已经看到二分查找的时间复杂度为  $O(\log n)$ 。

#### 元素序列的递归求和

线性递归可以作为一个有用的工具来处理数据序列, 例如 Python 列表。例如, 假设想要计算一个含有  $n$  个整数的序列  $S$  的和。我们可以使用线性递归解决这个求和问题。通过观察发现, 如果  $n = 0$ ,  $S$  中所有  $n$  个整数的总和是 0; 否则, 序列  $S$  的和应为  $S$  中的前  $n - 1$  个整数的总和加上  $S$  中最后一个元素 (见图 4-9)。

|   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |
|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| 4 | 3 | 6 | 2 | 8 | 9 | 3 | 2 | 8 | 5 | 1  | 7  | 2  | 8  | 3  | 7  |

图 4-9 通过前  $n - 1$  个整数的总和加上最后一个数, 递归地计算序列的和

基于这个客观事实，代码段 4-9 实现了计算数字序列和的递归算法

#### 代码段 4-9 使用线性递归计算序列元素的和

```

1 def linear_sum(S, n):
2 """Return the sum of the first n numbers of sequence S."""
3 if n == 0:
4 return 0
5 else:
6 return linear_sum(S, n-1) + S[n-1]

```

图 4-10 给出了 `linear_sum` 函数递归追踪的一个小例子。对于大小为  $n$  的输入，`linear_sum` 算法执行了  $n + 1$  次函数调用。因此，这将需要  $O(n)$  的时间，因为它花费恒定的时间执行每次调用的非递归部分。此外，我们还可以看到，这个算法使用的内存空间（除了序列  $S$ ）也是  $O(n)$ ，正如在做出最后一次的递归调用（当  $n = 0$ ）时的递归追踪中，对  $n + 1$  个活动记录的任何一个我们都使用固定数量的内存空间。

#### 使用递归逆置序列

接下来，让我们考虑逆置含有  $n$  个元素的序列  $S$  的问题，即第一个元素成为最后一个元素，第二个元素成为倒数第二个元素，以此类推。我们可以使用线性递归解决这个问题，通过观察，通过对调第一个元素和最后一个元素，之后递归地反置剩余元组，这样就可以完成序列的逆置。按照约定，我们把第一次调用的算法记作 `reverse(S, 0, len(S))`。代码段 4-10 给出了这个算法的一个实现。

#### 代码段 4-10 使用线性递归逆置序列的元素

```

1 def reverse(S, start, stop):
2 """Reverse elements in implicit slice S[start:stop]."""
3 if start < stop - 1: # if at least 2 elements:
4 S[start], S[stop-1] = S[stop-1], S[start] # swap first and last
5 reverse(S, start+1, stop-1) # recur on rest

```

需要注意的是，有两个隐含的基本情况场景：当  $start == stop$  时，这个隐含的范围是空的；当  $start == stop - 1$  时，这个隐含的范围仅含有一个元素。这两种情况中的任何一个，都不需要再执行任何操作，因为含有零个或者一个元素的序列与它的逆置序列是完全相等的。当其他情况调用递归时，我们都保证使过程朝着一个基本情况发展，不同的是， $stop - start$  每次调用减小两个值（见图 4-11）。如果  $n$  是偶数，最终将达到  $start == stop$  这种情况；如果  $n$  是奇数，最终会达到  $start == stop - 1$  这种情况。

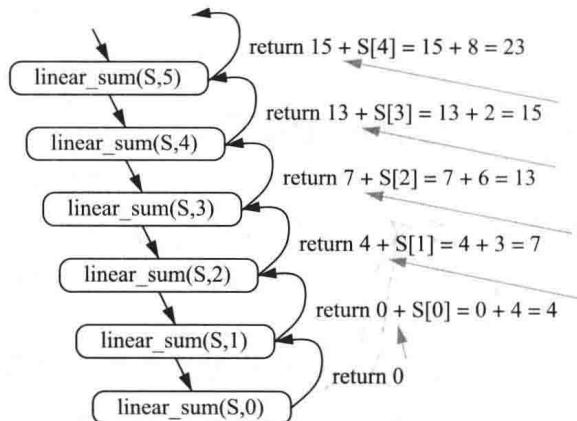


图 4-10 对 `linear_sum(S, 5)` 执行的递归追踪，其中输入的参数是  $S = [4, 3, 6, 2, 8]$

|   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| 4 | 3 | 6 | 2 | 8 | 9 | 5 |
| 5 | 3 | 6 | 2 | 8 | 9 | 4 |
| 5 | 9 | 6 | 2 | 8 | 3 | 4 |
| 5 | 9 | 8 | 2 | 6 | 3 | 4 |
| 5 | 9 | 8 | 2 | 6 | 3 | 4 |

图 4-11 逆置一个序列的递归追踪。  
阴影部分是尚未被逆置的

上面的观点意味着代码段 4-10 的递归算法确保在进行  $1 + \left\lfloor \frac{n}{2} \right\rfloor$  次递归调用后递归终止。

因为每次调用包含固定数量的工作，所以整个过程运行时间为  $O(n)$ 。

### 用于计算幂的递归算法

再举一个线性递归应用的例子，即数  $x$  的  $n$  次幂问题，其中  $n$  是任意的非负整数。也就是说，我们希望计算幂函数（power function），其定义为  $\text{power}(x, n) = x^n$ 。（对于这个讨论，我们使用的名字为“power”，以便与同样能提供这个功能的 built-in 函数区分）对于这个问题，我们将考虑两个不同的递归公式，这两个公式会导致算法有不同的性能。

对于  $n > 0$ ，遵从  $x^n = x \cdot x^{n-1}$  这个事实的一个简单的递归定义。

$$\text{power}(x, n) = \begin{cases} 1 & n=0 \\ x \cdot \text{power}(x, n-1) & \text{其他} \end{cases}$$

代码段 4-11 给出了这个定义产生的一个递归算法。

代码段 4-11 用简单的递归计算幂函数

---

```

1 def power(x, n):
2 """ Compute the value x**n for integer n."""
3 if n == 0:
4 return 1
5 else:
6 return x * power(x, n-1)

```

---

这个版本的  $\text{power}(x, n)$  函数递归调用的时间复杂度为  $O(n)$ 。它的递归追踪和图 4-1 中阶乘函数的递归追踪的结构非常相似，都是每次调用参数减 1，并且每  $n+1$  层执行固定数量的工作。

不过，有一种更快的方法用以计算幂函数，即采用了平方技术的定义。用  $k = \left\lfloor \frac{n}{2} \right\rfloor$  表示递归的层数（Python 中表示为  $n//2$ ）。我们考虑  $(x^k)^2$  这种表示：当  $n$  是偶数时， $\left\lfloor \frac{n}{2} \right\rfloor = \frac{n}{2}$ ，因此  $(x^k)^2 = \left( x^{\frac{n}{2}} \right)^2 = x^n$ ；当  $n$  是奇数时， $\left\lfloor \frac{n}{2} \right\rfloor = \frac{n-1}{2}$  且  $(x^k)^2 = x^{n-1}$  因此  $x^n = x \cdot (x^k)^2$ ，比如  $2^{13} = 2 \cdot 2^6 \cdot 2^6$ 。通过这个分析，我们可以得出如下的递归定义：

$$\text{Power}(x, n) = \begin{cases} 1 & n=0 \\ x \cdot \left( \text{power}\left(x, \left\lfloor \frac{n}{2} \right\rfloor \right) \right)^2 & n>0 \text{ 是奇数} \\ \left( \text{power}\left(x, \left\lfloor \frac{n}{2} \right\rfloor \right) \right)^2 & n>0 \text{ 是偶数} \end{cases}$$

如果要执行两个递归调用来计算  $\text{power}\left(x, \left\lfloor \frac{n}{2} \right\rfloor\right) \cdot \text{power}\left(x, \left\lfloor \frac{n}{2} \right\rfloor\right)$ ，那么实现这个递归的追踪表示要进行  $O(n)$  次调用。我们可以通过计算  $\text{power}\left(x, \left\lfloor \frac{n}{2} \right\rfloor\right)$  作为部分结果，然后乘以它本身来显著地减少执行的操作。代码段 4-12 给出了基于这种递归定义的一个示例。

代码段 4-12 使用重复的平方计算幂函数

---

```

1 def power(x, n):
2 """Compute the value x**n for integer n."""
3 if n == 0:
4 return 1
5 else:
6 partial = power(x, n // 2) # rely on truncated division
7 result = partial * partial
8 if n % 2 == 1: # if n odd, include extra factor of x
9 result *= x
10 return result

```

---

为了说明改进算法的执行，图 4-12 给出了计算  $\text{power}(2, 13)$  函数的递归追踪。

为了分析修正算法的运行时间，我们观察到函数  $\text{power}(x, n)$  每个递归调用中的指数最多是之前调用的一半。如我们在二分查找的分析中所看到的，在变成 1 或者更少之前，我们可以用 2 除  $n$  的时间的数量级是  $O(\log n)$ 。因此，新构想的幂函数产生  $O(\log n)$  次递归调用。每个单独的函数的激活执行  $O(1)$  个操作（不包括递归调用），所以计算函数  $\text{power}(x, n)$  操作的总数是  $O(\log n)$ 。在原始的时间复杂度为  $O(n)$  的算法上这是一个显著的改进。

在减少内存使用方面，改进版本显著节约了内存。第一个版本的递归深度为  $O(n)$ ，因此  $O(n)$  个激活记录同时被存储在内存中。因为改进版本的递归深度是  $O(\log n)$ ，其所用内存也是  $O(\log n)$ 。

#### 4.4.2 二路递归

当一个函数执行两个递归调用时，我们就说它使用了二路递归。我们已经列举了二路递归的几个例子，最具代表性的是绘制一个英式标尺（见 4.1.2 节），或者是 4.3 节的 `bad_fibonacci` 函数。作为二路递归的另一个应用，让我们回顾一下计算序列  $S$  的  $n$  个元素之和问题。计算一个或零个元素的总和是微不足道的。在有两个或者更多元素的情况下，我们可以递归地计算前一半元素的总和和后一半元素的总和，然后把这两个和加在一起。在代码段 4-13 中，对于这样一个算法，实现最初是以 `binary_sum(A, 0, len(A))` 而被调用的。

代码段 4-13 用二路递归计算一个序列的元素之和

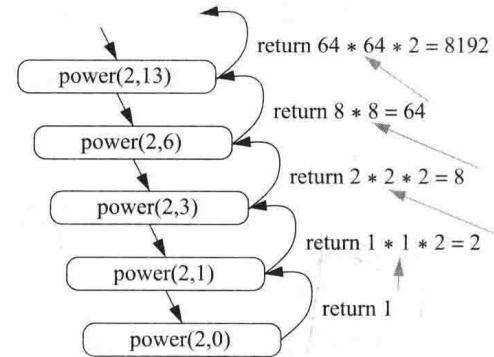
---

```

1 def binary_sum(S, start, stop):
2 """Return the sum of the numbers in implicit slice S[start:stop]."""
3 if start >= stop: # zero elements in slice
4 return 0
5 elif start == stop - 1: # one element in slice
6 return S[start]
7 else: # two or more elements in slice
8 mid = (start + stop) // 2
9 return binary_sum(S, start, mid) + binary_sum(S, mid, stop)

```

---

图 4-12 对  $\text{power}(2, 13)$  函数执行的递归追踪

为了分析算法 `binary_sum`，且为了方便起见，我们考虑当  $n$  为 2 的整数次幂的情况。图 4-13 显示了 `binary_sum(0, 8)` 函数执行的递归追踪。我们在每个圆角矩形中添加一个标

签，这个标签是所调用的参数 `start:stop` 的值。每次递归调用后，范围的尺寸减小一半，因此递归的深度为  $1 + \log_2 n$ 。因此，`binary_sum` 函数使用  $O(\log n)$  数量级的额外空间，与代码段 4-9 中 `linear_sum` 函数使用  $O(n)$  数量级的空间相比，这是一个巨大的进步。然而，`binary_sum` 函数的时间复杂度是  $O(n)$ ，因为有  $2n - 1$  函数次调用，每次都需要恒定的时间。

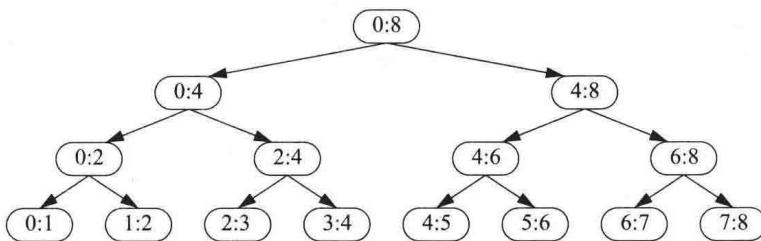


图 4-13 `binary sum(0, 8)` 执行的递归追踪

#### 4.4.3 多重递归

从二路递归可知，我们将多重递归定义为一个过程，在这个过程中，一个函数可能会执行多于两次的递归调用。对于一个文件系统磁盘空间使用状况分析的递归（见 4.1.4 节）是多重递归的一个例子，因为在调用期间，递归调用执行的次数等于在文件系统给定目录中条目的数量。

另一个多重递归的常见应用是通过枚举各种配置来解决组合谜题的情况。例如，以下是所谓的求和谜题的所有实例：

pot + pan = bib  
 dog + cat = pig  
 boy + girl = baby

为了解决这样的谜题，我们需要分配唯一的数字（即  $0, 1, \dots, 9$ ）给方程中的每个字母，以便使方程为真。通常情况下，我们通过人工对特殊问题的观察解决这样一个谜题，这个特殊问题即解决并测试每个配置的正确性以消除配置（也就是数字与字母的可能部分分配），直到可以得出可行的配置。

但是，如果可能配置的数量不是太大，我们可以用计算机简单地列举所有可能性，并测试每一个可能，而不需要任何人工的观察。此外，这种算法可以以一种系统的方式使用多重递归得出正确的配置。代码段 4-14 给出了这样一个算法的伪代码。为了确保描述足以被其他问题使用，这个算法枚举并测试所有长度为  $k$  的序列，而且不与给定全集  $U$  的元素重复。我们通过以下步骤创建含有  $k$  个元素的序列：

- 1) 递归生成含有  $k - 1$  个元素的序列。
- 2) 附加一个元素到每个这样的未包含该元素的序列中。

在算法执行的整个过程中，我们使用一个集合  $U$  来跟踪不包含在当前序列中的元素，从而当且仅当元素  $e$  在  $U$  中时，它还未被使用。

看待代码段 4-14 中算法的另一种方式是它列举  $U$  所有可能大小为  $k$  的子集，并且测试每个子集，这些子集是问题的可能解决方案之一。

对于求和问题， $U = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ ，并且序列中的每个位置对应一个给定的字母。例如，第一个位置可以代表  $b$ ，第二个位置代表  $o$ ，第三个位置代表  $y$ ，以此类推。

## 代码段 4-14 通过枚举和测试所有可能的配置来解决组合谜题

**Algorithm** PuzzleSolve( $k, S, U$ ):

**Input:** An integer  $k$ , sequence  $S$ , and set  $U$

**Output:** An enumeration of all  $k$ -length extensions to  $S$  using elements in  $U$  without repetitions

**for each**  $e$  in  $U$  **do**

    Add  $e$  to the end of  $S$

    Remove  $e$  from  $U$

{ $e$  is now being used}

**if**  $k == 1$  **then**

        Test whether  $S$  is a configuration that solves the puzzle

**if**  $S$  solves the puzzle **then**

**return** "Solution found: "  $S$

**else**

        PuzzleSolve( $k - 1, S, U$ )

{a recursive call}

        Remove  $e$  from the end of  $S$

{ $e$  is now considered as unused}

        Add  $e$  back to  $U$

图 4-14 显示了  $\text{PuzzleSolve}(3, S, U)$  函数调用的递归追踪，其中  $S$  为空并且  $U = \{a, b, c\}$ 。这个执行生成并测试了  $a, b, c$  三个字符的所有排列。注意初始调用进行三次递归调用，其中每一个调用又进行两次甚至更多次调用。在一个包含四个元素的集合  $U$  上，如果已经执行了  $\text{PuzzleSolve}(3, S, U)$ ，那么初始调用可能已经进行了四项递归调用，其中每一个调用将有一个追踪——类似于图 4-14 描述的一样。

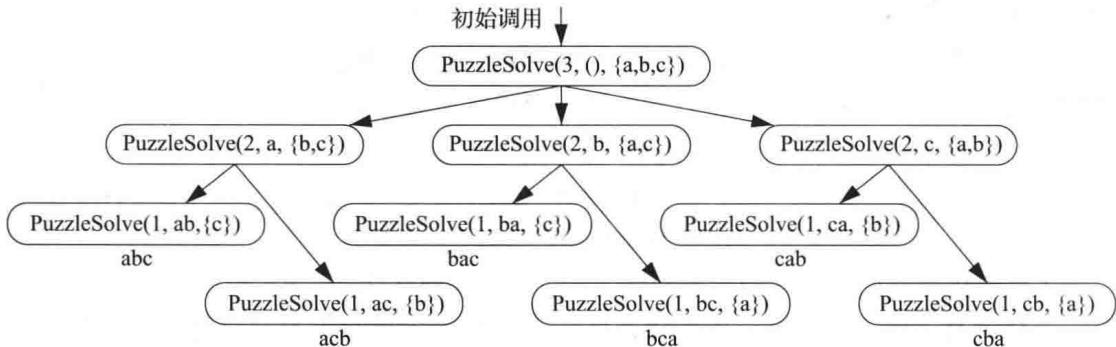


图 4-14  $\text{PuzzleSolve}(3, S, U)$  函数执行的递归追踪

## 4.5 设计递归算法

一般来说，使用递归的算法通常具有以下形式：

- 对于基本情况的测试。首先测试一组基本情况（至少应该有一个）。这些基本情况应该被定义，以便每个可能的递归调用链最终会达到一种基本情况，并且每个基本情况的处理不应使用递归。
- 递归。如果不是一种基本情况，则执行一个或多个递归调用。这个递归步骤可能包括一个测试，该测试决定执行哪几种可能的递归调用。我们应该定义每个可能的递归调用，以便使调用向一种基本情况靠近。

### 参数化递归

要为一个给定的问题设计递归算法，考虑我们可以定义的子问题的不同方式是非常有用的，该子问题与原始问题有相同的总体结构。如果很难找到需要设计递归算法的重复结构，解决一些具体问题有时是有用的，这样可以看出子问题应该如何定义。

一个成功的递归设计有时需要重新定义原来的问题，以便找到看起来相似的子问题。这通常涉及参数化函数的特征码。例如，在一个序列中执行二分查找算法时，对调用者的自然函数特征码将显示为 `binary_search(data, target)`。不过，在 4.1.3 节中，我们调用特征码 `binary_search(data, target, low, high)` 定义函数，并且使用额外的参数说明子列表作为递归过程。对于二分查找来说，在参数化方面的这个改变是至关重要的。如果坚持简便的特征值 `binary_search(data, target)`，在列表的一半进行搜索的唯一方法可能是建立一个只含有这些元素的新列表并且把它作为第一个参数。然而，复制列表的一半已经需要  $O(n)$  的时间，这就否定了二分查找算法全部的优点。

如果希望给一个像二分查找这样的算法提供一个简洁的公共接口，而不会干扰用户的其他参数，那么标准的技术是创建一个有简洁接口的公共函数，比如 `binary_search(data, target)`，然后让它的函数体调用一个非公共的效用函数，这个效用函数含有我们所希望的递归参数。

你会发现我们对本章其他几个例子的递归类似地进行了重新参数化（例如，`reverse`、`linear_sum` 及 `binary_sum`）。在 `good_fibonacci` 函数的实现中，通过有意加强返回的期望（在这种情况下，返回的是一对数字而不是一个数字），我们看到了一种用以重新定义递归的不同方法。

## 4.6 消除尾递归

算法设计的递归方法的主要优点是，它使我们能够简洁地利用重复结构呈现诸多问题。通过使算法描述以递归的方式利用重复结构，我们经常可以避开复杂的案例分析和嵌套循环。这种方法会得出可读性更强的算法描述，而且十分有效。

然而，递归的可用性要基于合适的成本。特别是，Python 解释器必须保持跟踪每个嵌套调用的状态的活动记录。当计算机内存价格昂贵时，在某些情况下，能够从那些递归算法得到非递归算法是很有用的。

在一般情况下，我们可以使用堆栈数据结构，堆栈结构将在 6.1 节介绍，通过管理递归结构自身的嵌套，而不是依赖于解释器，从而把递归算法转换成非递归算法。虽然这只是把内存使用从解释器变换到堆栈，但是也许能够通过只存储最小限度的必要信息来减少内存使用。

更好的情况是，递归的某些形式可以在不使用任何辅助存储空间的情况下被消除。其中一种著名的形式被称为尾递归（tail recursion）。如果执行的任何递归调用是在这种情况下最后操作，而且通过封闭递归，递归调用的返回值（如果有的话）立即返回，那么这个递归是一个尾递归。根据需要，一个尾递归必须是线性递归（因为如果必须立即返回第一个递归调用的结果，那么将无法进行第二次递归调用）。

在本章给出的递归函数中，代码段 4-3 的 `binary_search` 函数和代码段 4-10 的 `reverse` 函数均是尾递归的例子。虽然其他几个线性递归很像尾递归，但技术上并不是如此。例如，代码段 4-1 中的阶乘函数不是一个尾递归。它最后的命令：

```
return n * factorial(n-1)
```

这不是一个尾递归，因为递归调用完成之后进行了额外的乘法运算。出于类似的原因，代码段 4-9 的 `linear_sum` 函数和代码段 4-7 的 `good_fibonacci` 函数也不是尾递归。

在重复循环中，通过封闭函数体，并且通过重新分配现存参数的这些值以及用新的参数来代替一个递归调用，任何尾递归都可以被非递归地重新实现。举一个实例，如代码段 4-15 所示，binary\_search 函数可以被重新实现，仅需要在 while 循环之前，初始化变量 low 和 high 来表示序列的完整的程度。然后，经过每次的循环找到目标值或者缩小候选子序列的范围。

代码段 4-15 二分查找算法的非递归实现

---

```

1 def binary_search_iterative(data, target):
2 """Return True if target is found in the given Python list."""
3 low = 0
4 high = len(data)-1
5 while low <= high:
6 mid = (low + high) // 2
7 if target == data[mid]: # found a match
8 return True
9 elif target < data[mid]: # only consider values left of mid
10 high = mid - 1
11 else: # only consider values right of mid
12 low = mid + 1
13 return False # loop ended without success

```

---

在最初版本中进行递归调用 binary\_search(data, target, low, mid - 1) 函数的地方，仅用 high = mid - 1 进行替换，然后继续下一个循环的迭代。最初的基本情况的条件 low > high 只被相反的循环条件 while low <= high 所取代。在新的实现中，如果 while 循环结束，则用返回 False 来特指查找失败（也就是说，没有从内部返回 True）。

我们同样可以实现代码段 4-10 原始递归逆置 (reverse) 方法的非递归实现，如代码段 4-16 所示。

代码段 4-16 使用迭代逆置一个序列的元素

---

```

1 def reverse_iterative(S):
2 """Reverse elements in sequence S."""
3 start, stop = 0, len(S)
4 while start < stop - 1:
5 S[start], S[stop-1] = S[stop-1], S[start] # swap first and last
6 start, stop = start + 1, stop - 1 # narrow the range

```

---

在新版本中，在每个循环期间，更新 start 和 stop 的值。一旦在这个范围内只有一个或者更少的元素，即退出。

即使许多其他线性递归不是正式的尾递归，它们也可以非常有效地用迭代来表达。例如，对于计算阶乘、求序列元素的和或者有效地计算斐波纳契数，都有简单的非递归实现。事实上，从 1.8 节可以看出，斐波那契数生成器的实现使用  $O(1)$  的时间产生每个子序列的值，因此需要  $O(n)$  的时间来产生该系列中的第  $n$  个条目。

## 4.7 练习

请访问 [www.wiley.com/college/goodrich](http://www.wiley.com/college/goodrich) 以获得练习帮助。

### 巩固

- R-4.1 对于一个含有  $n$  个元素的序列  $S$ ，描述一个递归算法查找其最大值。所给出的递归算法时间复杂度和空间复杂度各是多少？

- R-4.2 使用在代码段 4-11 中实现的传统函数，绘制出 `power(2, 5)` 函数计算的递归跟踪。
- R-4.3 如代码段 4-12 实现的函数所示，使用重复平方算法，绘制出 `power(2, 18)` 函数计算的递归跟踪。
- R-4.4 绘制函数 `reverse(S, 0, 5)`（代码段 4-10）执行的递归追踪，其中  $S = [4, 3, 6, 2, 6]$ 。
- R-4.5 绘制函数 `PuzzleSolve(3, S, U)`（代码段 4-14）执行的递归追踪，其中  $S$  为空并且  $U = \{a, b, c, d\}$ 。
- R-4.6 描述一个递归函数，用于计算第  $n$  个调和数（harmonic number），其中  $H_n = \sum_{i=1}^n 1/i$ 。
- R-4.7 描述一个递归函数，它可以把一串数字转换成对应的整数。例如，'13 531' 对应的整数为 13 531。
- R-4.8 Isabel 用一种有趣的方法来计算一个含有  $n$  个整数的序列  $A$  中的所有元素之和，其中  $n$  是 2 的幂。她创建一个新的序列  $B$ ，其大小是序列  $A$  的一半并且设置  $B[i] = A[2i] + A[2i + 1](i = 0, 1, \dots, (n/2) - 1)$ 。如果  $B$  的大小为 1，那么输出  $B[0]$ ；否则，用  $B$  取代  $A$ ，并且重复这个过程。那么她的这个算法的时间复杂度是多少？

## 创新

- C-4.9 写一个简短的递归 Python 函数，用于在不使用任何循环的条件下查找一个序列中的最小值和最大值。
- C-4.10 在只使用加法和整数除法的情况下，描述一个递归算法，来计算以 2 为底的  $n$  的对数的整数部分。
- C-4.11 描述一个有效的递归函数来求解元素的唯一性问题，在不使用排序的最坏的情况下运行时间最多是  $O(n^2)$ 。
- C-4.12 在只使用加法和减法的情况下，给出一个递归算法，来计算两个正整数  $m$  和  $n$  的乘积。
- C-4.13 在 4.2 节中，我们用归纳法证明调用 `draw_interval(c)` 函数打印的行数是  $2^c - 1$ 。另一个有趣的问题是在此过程中有多少短线被打印出来。通过归纳法证明调用 `draw_interval(c)` 函数打印的短线的数量为  $2^{c+1} - c - 2$ 。

- C-4.14 在汉诺塔问题中，我们给出了一个平台，有三根柱子  $a$ 、 $b$  和  $c$  从这个平台上伸出。在柱子  $a$  上放有  $n$  个盘子，每个都比后放上来的盘子大，因此，最小的盘子在顶部并且最大的盘子在底部。本题是把所有盘子从柱子  $a$  移动到柱子  $b$ ，每次移动一个盘子，并且不会把大一些的盘子放在小一些的盘子的上面。参见图 4-15 中  $n = 4$  的例子。描述一个递归算法，用来求解任意整数  $n$  的汉诺塔问题。（提示：首先考虑这个问题的子问题，即使用第三个柱子把除第  $n$  个盘子之外的所有盘子从柱子  $a$  移动到另一个柱子作为“临时存储”。）

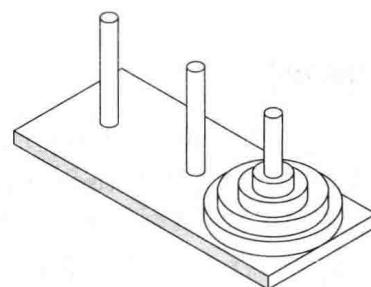


图 4-15 汉诺塔问题的一个示意图

- C-4.15 编写一个递归函数，该函数将输出一个含有  $n$  个元素的集合的所有子集（没有任何重复的子集）。
- C-4.16 编写一个简短的递归 Python 函数，它接受一个字符串  $s$  并且输出其逆置字符串。例如字符串 'pots&pans' 的逆置字符串为 'snap&stop'。
- C-4.17 编写一个简短的递归 Python 函数，确定一个字符串  $s$  是否是它的一个回文字符串，也就是说，该字符串与其逆置字符串相同。例如，字符串 'racecar' 和 'gohangasalamimalasagnahog' 是回文字符串。
- C-4.18 使用递归编写一个 Python 函数，确定字符串  $s$  中是否元音字母比辅音字母多。
- C-4.19 编写一个简短的递归 Python 函数，用于重新排列一个整数值序列，使得所有偶数值出现在所

有奇数值的前面。

- C-4.20 给定一个未排序的整数序列  $S$  和整数  $k$ , 描述一个递归算法, 用于对  $S$  中的元素重新排序, 使得所有小于等于  $k$  的元素在所有大于  $k$  的元素之前。在这个含有  $n$  个值的序列中, 算法的时间复杂度是多少?
- C-4.21 假设给出一个含有  $n$  个元素的序列  $S$ , 这个序列是包含不同元素的升序序列。给定一个数  $k$ , 描述一个递归算法找到  $S$  中总和为  $k$  的两个整数 (如果这样的一对整数存在)。算法的时间复杂度是多少?
- C-4.22 从代码段 4-12 使用重复平方的 power 函数的版本中, 实现一个非递归实例。

### 项目

- P-4.23 实现一个具有特征值 `find(path, filename)` 的递归函数, 该特征值报告在具有指定路径的指定文件名为根的文件系统的所有条目。
- P-4.24 编写一个程序, 通过列举和测试所有可能的配置来解决求和谜题。使用该程序解决 4.4.3 节给出的三个问题。
- P-4.25 对于 4.1.2 节的英式标尺工程, 用 `draw_interval` 函数的一个非递归实现。如果  $c$  代表中心刻度线的长度, 那么应该精确地有  $2^c - 1$  行输出。如果从 0 至增加  $2^c - 2$  个计数器, 每个刻度线中短线的数量应该恰好比在计数器的二进制表示的结尾连续的 1 的数量多 1。
- P-4.26 编写一个程序, 以解决汉诺塔问题的实例 (参见练习 C-4.14)。
- P-4.27 Python 的 `os` 模块提供了一个有特征值 `walk(path)` 的函数, 该特征值对于由字符串路径标识目录的每个子目录来说是三元组 (`dirpath, dirnames, filenames`) 的发生器。比如字符串 `dirpath` 是子目录的完整路径, `dirnames` 是在 `dirpath` 内子目录名称的列表, `filenames` 是 `dirpath` 非目录条目名称的列表。例如, 当查看图 4-6 所示的文件系统的目录 `cs016` 的子目录时, `walk` 会产生 ('/user/rt/courses/cs016', ['homeworks', 'programs'], ['grades'])。给出这样一个 `walk` 函数的实现。

### 扩展阅读

在程序中, 递归的使用属于计算机科学的特色 (参见 Dijkstra 算法的文章<sup>[36]</sup>)。这也是函数编程语言的核心 (参见 H. Abelson、G. J. Sussman 和 J. Sussman 的经典著作<sup>[1]</sup>)。有趣的是, 二分查找首次出版于 1946 年, 但直到 1962 年才发表了一个完全正确的形式。有关本章内容的进一步讨论, 请参阅 Bentley<sup>[14]</sup> 和 Lesuisse<sup>[68]</sup> 的论文。

# 基于数组的序列

## 5.1 Python 序列类型

在本章中，我们探讨 Python 的各种“序列”类，即内嵌的列表类（list）、元组类（tuple）和字符串类（str）。这些类之间有明显的共性，最主要的是：每个类都支持用下标访问序列元素，比如使用语法 `seq[k]`；每个类都使用数组这种低层次概念表示序列。然而，在 Python 中，这些类所表示的抽象以及实例化的方式有着明显的区别。因为这些类被广泛用于 Python 程序中，又因为它们能够成为构件块，用这些构件块可以开发更复杂的数据结构，所以，我们迫切需要搞清楚这些类的公共行为和内部运作机制。

### 行为

一个优秀的程序员有必要正确理解类的外部语义。列表、字符串和元组的使用看似简单，然而在理解与这些类相关的行为上，却有一些重要的细节（比如说复制序列意味着什么，或者取序列的一部分又意味着什么）。对类的行为有误解很容易导致程序中出现无意识的错误。因此，我们要在头脑中为这些类建立准确的模型。这些模型将会帮助我们研究更高级的用法，比如使用多维数据集合表示列表的列表。

### 实现细节

关注这些类的内部实现似乎有悖于面向对象编程的原则。在 2.1.2 节中，我们强调过封装的原则，指出在使用类时不需要知道其内部实现细节。虽然这句话没错，即程序员仅需要理解类的公共接口的语法和语义就能够用类的实例写出合法且准确的代码，但程序的效率很大程度上依赖于其所使用组件的效率。

### 渐近和实验分析

对于 Python 序列类，我们依据在第 3 章给出的渐近分析符号来描述其各种操作的效率。我们也将对主要操作执行实验分析，给出和更具理论化的渐近分析相一致的实验性结论。

## 5.2 低层次数组

为了能准确描述 Python 所表示序列类型的方法，我们必须先讨论计算机体系结构的低层次内容。计算机主存由位信息组成，这些位通常被归类成更大的单元，这些单元则取决于精准的系统架构。一个典型的单元就是一个字节，相当于 8 位。

计算机系统拥有庞大数量的存储字节，为了能跟踪信息存储在哪个字节，计算机采用了一个抽象概念，即我们熟悉的存储地址。实际上，每个存储字节都和一个作为其地址的唯一数字相关联（更正式地说，数字的二进制表示作为地址）。例如，使用这种方式，计算机系统能够将“字节 #2150”中的数据和“字节 #2157”中的数据进行对比。存储地址通常和存储系统的物理设计相协调，我们因此通常以顺序的方式描述这些数字。图 5-1 给出了这样一个图表，在该图表中，每个字节均被指定了存储地址。



图 5-1 计算机内存的部分示例，其中每个字节都被指定了连续的存储地址

尽管编号系统具有顺序性，但计算机硬件也是这样设计的，因此，从理论上说，基于这种存储地址，主存的任何字节都能被有效地访问。从这个意义上说，我们将计算机主存称为随机存儲存储器（Random Access Memory，RAM）。也就是说，检索字节 #8675309 就和检索字节 #309 一样容易。（在实践中，有很多复杂的因素，包括对缓存和外部存储器的使用，我们会在 15 章解决一些这样的问题）使用渐近分析的符号，我们认为存储器的任一单个字节被存储或检索的运行时间为  $O(1)$ 。

一般来说，编程语言记录标识符和其关联值所存储的地址之间的联系。比如，标识符  $x$  可能和存储器中的某一值关联，而标识符  $y$  和另一值关联。常见的编程任务就是记录一系列相关对象。例如，我们可能希望某一视频游戏能够记录此游戏的前十名玩家的分数。在此任务中，我们不会用 10 个变量来记录，而更倾向于为一个组赋以组名，并使用索引值指向该组内的高分。

一组相关变量能够一个接一个地存储在计算机存储器的一块连续区域内。我们将这样的表示法称为数组（array）。举一个实际的例子，一个文本字符串是以一列有序字符的形式存储的。在 Python 中，每个字符都用 Unicode 字符集表示，对于大多数计算机系统，Python 内部用 16 位表示每个 Unicode 字符（即 2 个字节）。因此，一个 6 个字符的字符串，比如 'SAMPLE'，将会被存储在存储器的连续 12 个字节中，如图 5-2 所示。

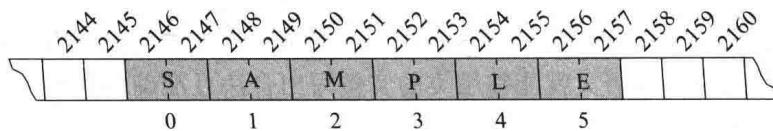
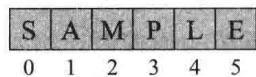


图 5-2 一个 Python 字符串以字符数组的形式存储在计算机存储器中。假定该字符串的每个 Unicode 字符需要两个字节的存储空间。条目下面的数字即是该字符串的索引值

虽然该字符串需要 12 个字节的存储空间，但我们仍把它描述为 6 字符数组。我们会将数组中的每个位置称为单元，并用整数索引值描述该数组，其中单元的开始编号为 0、1、2 等。例如，在图 5-2 中，索引为 4 的数组单元的内容为 L，并且存储在存储器的 2154 和 2155 字节中。

数组的每个单元必须占据相同数量的字节。之所以这样要求，是为了允许使用索引值能够在常量时间内访问数组内的任一单元。尤其是，假如知道某一数组的起始地址（例如，在图 5-2 中，起始地址为 2146），每个元素所占的字节数（例如，每个 Unicode 字符占 2 个字节），和所要求的字符的索引值，通过计算  $\text{start} + \text{cellsize} * \text{index}$  便可得出其正确的内存地址。通过这个公式得出，单元 0 正好起始于数组的起始地址，单元 1 正好起始于数组起始地址后的一个 cellsize 字节，等等。例如，图 5-2 中的单元 4 起始地址为  $2146 + 2 \times 4 = 2146 + 8 = 2154$ 。

当然，在数组内计算内存地址的算法是自动处理的。因此，程序员可以把字符数组理解得更通俗、更抽象，如图 5-3 所示。



字符串的进一步抽象

### 5.2.1 引用数组

再举一个有用的例子：假设想要为某医院开发一套医疗信息系统，来记录当前分配到病床的病人的名字。假定医院有 200 张床，为方便起见，这些床编号为 0 ~ 199。我们可以考虑使用基于数组的数据结构来记录最近分配到这些病床上的病人的名字。例如，在 Python 中，我们可能会用到一张姓名清单，如下所示：

```
[‘Rene’, ‘Joseph’, ‘Janet’, ‘Jonas’, ‘Helen’, ‘Virginia’, …]
```

在 Python 中，为了用数组表示这样的列表，必须要满足数组的每个单元字节数都相同这一条件。然而，元素是字符串，它们串的长度显然不同。Python 可以用最长字符串（不仅目前存储的字符串，将来也可能存储任何字符串）来为每个单元预留足够的空间，但那样太浪费了。

相反，Python 使用数组内部存储机制（即对象引用，来表示一列或者元组实例。在最低层，存储的是一块连续的内存地址序列，这些地址指向一组元素序列。图 5-4 所示即为该列表的高层视图。

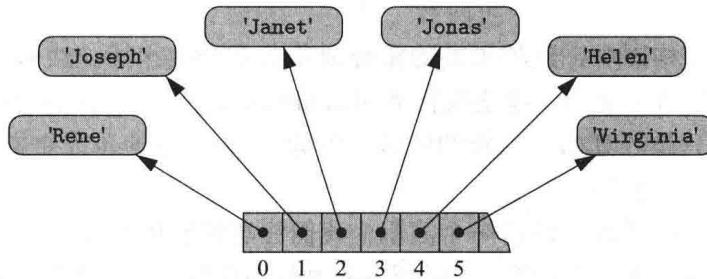


图 5-4 存储字符串引用的数组

虽然单个元素的相对大小可能不同，但每个元素存储地址的位数是固定的（比如，每个地址 64 位）。在这种方式下，Python 可以通过索引值以常量时间访问元素列表或元组。

在图 5-4 中，我们把医院病人的名字描述为字符串列表。当然，更有可能的是，该医疗信息系统可以管理每个病人更全面的信息，也许可以表示成 Patient 类的一个实例。从列表实现的观点看，同样的原则也适用于此，即列表仅保存那些对象引用的序列。同时需要注意，空对象 (None) 的引用能作为列表的元素来表示医院的空床位。

列表和元组是引用结构这一事实对这些类的语义来说是很重要的。一个列表实例可能会以多个指向同一个对象的引用作为列表元素，一个对象也可能被两个或更多列表中的元素所指向，因为列表仅仅存储返回对象的引用。例如，在计算列表的一小段时，结果产生了一个新的列表实例，该新列表指向了和原列表相同的元素，如图 5-5 所示。

当列表的元素是不变的对象时，正如图 5-5 中的整数实例一样，则两张表共享元素就显得没那么重要了，因为任何一张表都不能改变共享对象。比如，若在此结构图中执行语句 `temp[2] = 15`，这并未改变已存在的整数对象，而是将 `temp` 列表单元 2 中的引用指向了不同的对象。图 5-6 所示即为执行后的结构图。

当通过复制创建一个新的列表时也是同样的情况：比如 `backup = list(primes)`，就会对原列表复制出一张新列表。这张新列表即为浅拷贝（见 2.6 节），该列表和原列表指向同样的元

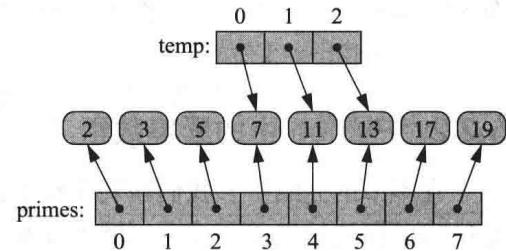


图 5-5 对 `temp` 执行的结果等于 `primes[3:6]`

素。当这些元素不可变时，浅拷贝也没关系。假如列表的元素是可变的，利用 copy 模块的 deepcopy 函数可以复制列表的元素，得到一个具有全新元素的新列表，这种方式称为深拷贝。

再给出一个更有用的例子：在 Python 中，使用诸如 counters = [0]\*8 这样的语句来初始化整数数组，这是很常见的一种做法。该语句构造出一张长度为 8、各元素为 0 的列表。从技术上讲，列表的 8 个单元都指向同一个对象，如图 5-7 所示。

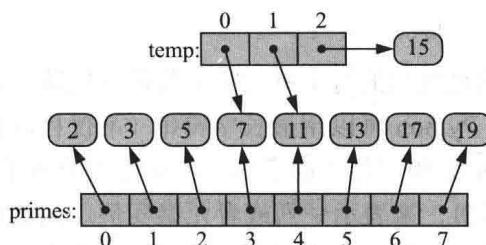


图 5-6 对图 5-5 中给出的结构图执行语句 `temp[2] = 15` 后的结果

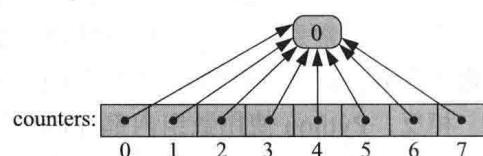


图 5-7 执行 `data = [0]*8` 后的结果

乍一看，在此结构图中，这种极端的重叠现象着实令人担忧。然而，我们可以依据指向的整数是不可变的这一事实。即使执行诸如 `counters[2] += 1` 这样的语句，技术上也不能改变现有的整数。只是计算出一个新的整数，值为  $0 + 1$ ，并使单元 2 指向了这个新的值。图 5-8 所示即为执行后的结构图。

下面对列表引用性质给出最后一个演示，我们注意到使用 extend 命令能将一个列表的所有元素添加到另一张列表的末尾。扩展列表的过程不是将那些元素复制过来，而是将元素的引用复制到末尾。图 5-9 所示即为调用 extend 函数的结果。

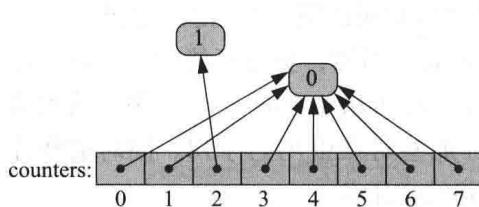


图 5-8 通过对图 5-7 中的列表执行 `counters[2] += 1` 后的结果

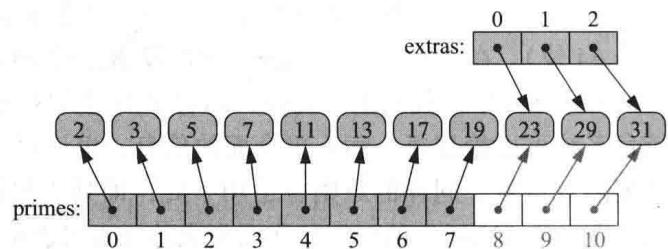
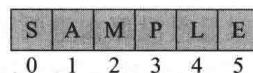


图 5-9 执行 `primes.extend(extras)` 后的结果，如浅灰色部分所示

## 5.2.2 Python 中的紧凑数组

在本节的介绍中，我们强调字符串是用字符数组表示的（而不是用数组的引用）。我们将会谈到更直接的表示方式——紧凑数组（compact array），因为数组存储的是位，这些位表示原始数据（在字符串情况下，这些位即是字符）。



在计算性能方面，紧凑数组比引用结构多几大优势。最重要的是，使用紧凑结构会占用更少的内存，因为在内存引用序列的显示存储上没有开销（原始数据除外），即引用结构通常会将 64 位地址存入数组，无论存储单个元素的对象有多少位。另外，字符串中的每个

Unicode 字节存储在紧凑数组中仅需要两个字节。如果每个字符都以单字符字符串独立存储，那显然将会占用更多字节。

接下来研究另一个案例，假设想要存储 100 万个 64 位整数。理论上，我们或许希望仅仅占用 64 000 000 位，然而通过估计得出 Python 列表将会占用的容量是该容量的 4~5 倍。每个列表元素都将产生一个 64 位存储地址，并将此地址存储于原始数组中，整数实例会被存储于内存的其他地方。Python 允许查询每个对象在主存中实际占用多少位字节——使用系统模块中的 `getsizeof` 函数即可得出。在我们的系统中，一个标准整型对象需要占用 14 字节内存（超出 4 字节的部分用于表示实际 64 位地址）。总之，列表每个条目要占用 18 个字节，而不是像整数紧凑列表那样仅需要 4 个字节。

紧凑结构在高性能计算方面的另一个重要优势是：原始数据在内存中是连续存放的。注意，引用结构没有这种情况。也就是说，即使列表对存储地址做了谨慎的规定，但是这些元素会被存入内存的什么位置并不受该列表所决定。由于缓存的工作性质和计算机的存储层次结构，将数据存到其他可能用于相同计算的数据旁边通常是有利的。

尽管引用结构明显效率低下，但在本书中，我们更看重 Python 列表和元组所提供的便利。我们将会在第 15 章讨论紧凑结构，届时将集中讨论内存使用对数据结构和算法的影响。Python 提供了几种用于创建不同类型的紧凑数组的方法。

紧凑数组主要通过一个名为 `array` 的模块提供支撑。该模块定义了一个类（也命名为 `array`），该类提供了紧凑存储原始数据类型的数组的方法。图 5-10 所示即为这样一个整型数组的描述。

`array` 类的公共接口通常和 Python 的 `list`（列表）一致。然而，该类的构造函数需要以类型代码（`type code`）作为第一个参数，也即一个字符，该

字符表明要存入数组的数据类型。举一个实际的例子，类型代码 '`i`' 表明这是一个（有符号的）整型数组，通常表示每个元素至少 16 位。我们可以把图 5-10 所示的数组声明如下：

```
primes = array('i', [2, 3, 5, 7, 11, 13, 17, 19])
```

类型代码允许解释器确定数组的每个元素需要多少位。正如表 5-1 所示，`array` 模块支持类型代码，这些类型代码主要是基于 C 编程语言（Python 使用最广泛的发布版就是用 C 语言实现）的本地数据类型。C 语言数据的精确位数是跟系统有关的，但可以给出通常的范围。

表 5-1 `array` 模块支持的类型代码

| 代码                 | 数据类型                            | 字节的精确位数 |
|--------------------|---------------------------------|---------|
| ' <code>b</code> ' | <code>signed char</code>        | 1       |
| ' <code>B</code> ' | <code>unsigned char</code>      | 1       |
| ' <code>u</code> ' | <code>Unicode char</code>       | 2 or 4  |
| ' <code>h</code> ' | <code>signed short int</code>   | 2       |
| ' <code>H</code> ' | <code>unsigned short int</code> | 2       |
| ' <code>i</code> ' | <code>signed int</code>         | 2 or 4  |
| ' <code>I</code> ' | <code>unsigned int</code>       | 2 or 4  |
| ' <code>L</code> ' | <code>signed long int</code>    | 4       |
| ' <code>L</code> ' | <code>unsigned long int</code>  | 4       |
| ' <code>f</code> ' | <code>Float</code>              | 4       |
| ' <code>d</code> ' | <code>float</code>              | 8       |

array 模块不支持存储用户自定义数据类型的紧凑数组。这种结构的紧凑数组可以用一个名为 `ctypes` 的底层模块来创建（5.3.1 节将会对 `ctypes` 模块做更多讨论）。

### 5.3 动态数组和摊销

在计算机系统中，创建低层次数组时，必须明确声明数组的大小，以便系统为其存储分配连续的内存。例如，图 5-11 给出了一个 12 字节的数组，该数组被分配在地址 2146 ~ 2157 的位置。



图 5-11 一个 12 字节数组被分配在存储器从地址 2146 ~ 2157 的位置

由于系统可能会占用相邻的内存位置去存储其他数据，因此数组大小不能靠扩展内存单元来无限增加。在表示 Python 元组（tuple）或者字符串（str）实例的情形中，这种限制就没什么问题了。由于这些类的实例变量都是不可变的，因此当对象实例化时，低层数组的大小就已确定了。

Python 列表（list）类提供了更有趣的抽象。虽然列表在被构造时已经有了确定的长度，但该类允许对列表增添元素，对列表的总体大小没有明显的限制。为了提供这种抽象，Python 依赖于一种算法技巧，即我们所熟知的动态数组（dynamic array）。

为了理解动态数组的语义，首先关键的一点是：一张列表通常关联着一个底层数组，该数组通常比列表的长度更长。例如，用户创建了一张具有 5 个元素的列表，系统可能会预留一个能存储 8 个对象引用的底层数组（而不止 5 个）。通过利用数组的下一个可用单元，剩余的长度使得增添列表元素变得很容易。

假如用户持续增添列表元素，所有预留单元最终将被耗尽。此时，列表类向系统请求一个新的、更大的数组，并初始化该数组，使其前面部分能与原来的小数组一样。届时，原来的数组不再需要，因此被系统回收。这种策略直观上就像寄居蟹，当旧的贝壳不足以容纳它时，它便会钻到更大的贝壳里。

经验证明，Python 的 `list` 类确实基于这种策略。我们在代码段 5-1 中给出源代码，在代码段 5-2 中给出程序样例输出，并使用了 `sys` 模块提供的 `getsizeof` 函数。该函数用于给出在 Python 中存储对象的字节数。对于列表，该函数仅给出此列表关联的数组和其他实例变量的字节数之和，而不包括任何分配给被该列表引用的元素的内存。

#### 代码段 5-1 在 Python 中，探究列表长度和底层大小关系的实验

---

```

1 import sys # provides getsizeof function
2 data = []
3 for k in range(n): # NOTE: must fix choice of n
4 a = len(data) # number of elements
5 b = sys.getsizeof(data) # actual size in bytes
6 print('Length: {0:3d}; Size in bytes: {1:4d}'.format(a, b))
7 data.append(None) # increase length by one

```

---

#### 代码段 5-2 代码段 5-1 实验的样例输出

---

```

Length: 0; Size in bytes: 72
Length: 1; Size in bytes: 104

```

---

```

Length: 2; Size in bytes: 104
Length: 3; Size in bytes: 104
Length: 4; Size in bytes: 104
Length: 5; Size in bytes: 136
Length: 6; Size in bytes: 136
Length: 7; Size in bytes: 136
Length: 8; Size in bytes: 136
Length: 9; Size in bytes: 200
Length: 10; Size in bytes: 200
Length: 11; Size in bytes: 200
Length: 12; Size in bytes: 200
Length: 13; Size in bytes: 200
Length: 14; Size in bytes: 200
Length: 15; Size in bytes: 200
Length: 16; Size in bytes: 200
Length: 17; Size in bytes: 272
Length: 18; Size in bytes: 272
Length: 19; Size in bytes: 272
Length: 20; Size in bytes: 272
Length: 21; Size in bytes: 272
Length: 22; Size in bytes: 272
Length: 23; Size in bytes: 272
Length: 24; Size in bytes: 272
Length: 25; Size in bytes: 272
Length: 26; Size in bytes: 352

```

在评估实验结果时，首先注意代码段 5-2 的第一行输出。可以注意到，空列表已经请求了一定量字节的内存（在我们的系统中是 72 个）。事实上，Python 中每个对象都保存了一些状态，例如，标志着该对象属于哪个类的引用。尽管不能直接访问列表的私有实例变量，但可以推测该列表以某种形式保存的一些状态信息，类似于：

|                 |                      |
|-----------------|----------------------|
| <u>n</u>        | 列表当前存储的实际元素的个数       |
| <u>capacity</u> | 当前所分配数组中允许存储的元素最大个数  |
| <u>A</u>        | 当前所分配数组的引用（最初为 None） |

当第 1 个元素添入列表时，我们就会检查底层结构的大小是否改变。特别需要注意的是，字节数从 72 跳到 104，增加了 32 个字节。本实验是在 64 位机器上运行的，这表明每个内存地址是 64 位（即 8 个字节）。我们推测增加的 32 个字节即为分配的用于存储 4 个对象引用的底层数组大小。这一推测符合这样的事实：当对列表增添第 2 个、第 3 个或者第 4 个元素时，我们没有发现在内存占用上有任何改变。

当增添第 5 个元素时，我们注意到内存占用的字节数从 104 跳到 136。假定列表最初占 72 个字节，最后变为总共 136 字节，增加  $64 = 8 \times 8$  个字节，这表明我们提供了 8 个对象引用的扩展空间。另外，当增添第 9 个元素前，内存占用都不再增加，这也跟实验结果相一致。从这个角度来讲，200 个字节可被视为最初的 72 个字节再加上用于存储 16 个对象引用的 128 个字节。当增添第 17 个元素后，整个存储占用将变为  $272 = 72 + 200 = 72 + 25 \times 8$ ，因此，足够存储 25 个对象引用。

因为列表是引用结构，对列表实例使用函数 `getsizeof` 得出的结果仅包括该列表主要结构的大小，不算由对象（即列表元素）所占用的内存。在实验中，我们不断给列表增添 `None` 对象，并不关心单元将会放什么内容，但是我们可以向列表中增添任何类型对象，结果不受元素大小（即 `getsizeof(data)`）所给出的字节数的影响。

假如想继续该实验并做进一步迭代，我们或许想搞清楚：每次当前一个数组使用完后，

Python 会创建一个多大的数组（见练习 R-5.2 和 C-5.13）。在探究使用 Python 创建的精确字节数之前，我们先继续本节学习，接下来给出用于实现动态数组和执行该数组性能渐近分析的一般方法。

### 5.3.1 实现动态数组

尽管 Python 的 list 类给出了动态数组的一种高度优化的实现（我们在本书的后续部分要依赖该实现方法），但学习该类是如何被实现的仍对我们有指导性意义。

关键在于提供能够扩展用于存储列表元素的数组  $A$  的方法。当然，实际上我们不能扩展数组，因为它的大小是固定的。当底层数组已满，而有元素要添入列表时，我们会执行下面的步骤：

- 1) 分配一个更大的数组  $B$ 。
- 2) 设  $B[i] = A[i]$  ( $i = 0, \dots, n - 1$ )，其中  $n$  表示条目的当前数量。
- 3) 设  $A = B$ ，也就是说，我们以后使用  $B$  作为数组来支持列表。
- 4) 在新的数组里增添元素。

图 5-12 中给出了上述步骤的示意图。

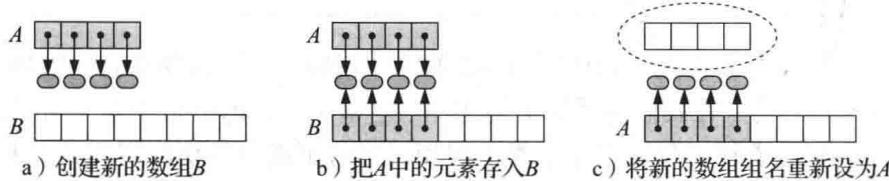


图 5-12 “扩展” 动态数组的三步示意图（没有给出旧数组回收和新数据插入的示意图）

接下来需要思考一个问题：新数组应该多大？通常的做法是：新数组大小是已满的旧数组大小的 2 倍。在 5.3.2 节中，我们会对这种做法进行数学分析。

在代码段 5-3 中，我们使用 Python 给出了动态数组的一种具体实现。DynamicArray 类的设计便是运用了本节所讨论的思想。虽然和 Python 中 list 类的接口一致，但这里仅提供部分功能：append 方法以及访问器 \_\_len\_\_ 和 \_\_getitem\_\_。底层数组的创建由 ctypes 模块提供。因为在本书的剩余部分不会一直使用这种底层结构，所以我们不再对 ctypes 模块进行详细说明。我们在私有实例方法 \_make\_array 中封装了必要的指令，该指令用于声明原始数组。扩展的主要过程在非公开的 \_resize 方法中实现。

代码段 5-3 使用 ctypes 模块提供的原始数组实现 DynamicArray 类

---

```

1 import ctypes # provides low-level arrays
2
3 class DynamicArray:
4 """A dynamic array class akin to a simplified Python list."""
5
6 def __init__(self):
7 """Create an empty array."""
8 self._n = 0 # count actual elements
9 self._capacity = 1 # default array capacity
10 self._A = self._make_array(self._capacity) # low-level array
11
12 def __len__(self):
13 """Return number of elements stored in the array."""

```

```

14 return self._n
15
16 def __getitem__(self, k):
17 """ Return element at index k."""
18 if not 0 <= k < self._n:
19 raise IndexError('invalid index')
20 return self._A[k] # retrieve from array
21
22 def append(self, obj):
23 """ Add object to end of the array."""
24 if self._n == self._capacity: # not enough room
25 self._resize(2 * self._capacity) # so double capacity
26 self._A[self._n] = obj
27 self._n += 1
28
29 def _resize(self, c): # nonpublic utility
30 """ Resize internal array to capacity c."""
31 B = self._make_array(c) # new (bigger) array
32 for k in range(self._n): # for each existing value
33 B[k] = self._A[k]
34 self._A = B # use the bigger array
35 self._capacity = c
36
37 def _make_array(self, c): # nonpublic utility
38 """ Return new array with capacity c."""
39 return (c * ctypes.py_object)() # see ctypes documentation

```

### 5.3.2 动态数组的摊销分析

本节中，我们对动态数组相关操作的运行时间做具体分析。我们用 3.3.1 节介绍的大  $\Omega$  符号，对这些操作的算法或步骤的运行时间给出渐近下界。

使用新的、更大的数组替换旧数组的策略起初似乎很慢，因为单个增添操作可能就需要  $\Omega(n)$  的运行时间，这里的  $n$  是指数组元素的当前数量。然而我们注意到，在数组的替换过程中，由于增大了 1 倍的容量，新数组在被替换之前允许增添  $n$  个新元素。这种方式使得每一次大的代价的替换过程后，对每个元素进行添加操作（见图 5-13）。这一事实让我们意识到：从总的运行时间来看，对初始为空的动态数组执行一系列的操作，其效率也是很高的。

我们使用一种称为摊销（amortization）的算法设计模式进行证明：事实上，在动态数组中执行一系列增添操作效率是非常高的。为了做摊销分析（amortized analysis），我们使用一种会计学技巧：把计算机视为一个投币装置，对每个固定的运行时间均支付一枚网络硬币（cyber-dollar）。当执行一个操作时，当前“银行账户”中要有足够的网络硬币来支付此次操作的运行时间。因此，在任意计算中所花费的网络硬币总数将会和该计算的运行时间成正比。使用该分析方法的妙处在于我们可以增加某些操作的投入，以减低其他操作所需的网络硬币。

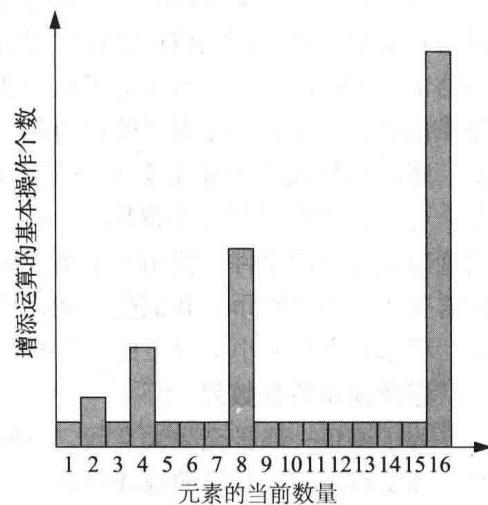


图 5-13 对动态数组执行一系列 append 操作的运行时间

**命题 5-1:** 设  $S$  是一个由具有初始大小的动态数组实现的数组，实现策略为：当数组已满时，将此数组大小扩大为原来的 2 倍。 $S$  最初为空，对  $S$  连续执行  $n$  个增添操作的运行时间为  $O(n)$ 。

**证明：**假定不需要擴大数组的情况下，向  $S$  中增加一个元素所需时间需支付一个网络硬币。另外，假定数组大小从  $k$  增长到  $2k$  时，初始化该新数组需要  $k$  个网络硬币。我们将会对每个增添操作索价 3 个网络硬币。因此，对不需要擴大数组的增添操作我们多付了 2 个网络硬币。在不需要擴大数组的增添操作中，我们多收的 2 个硬币将被视为“存入”该元素所插入的单元中。当数组  $S$  大小为  $2^i$  并且  $S$  中有  $2^i$  个元素时，对于  $i \geq 0$ ，增加元素将会出现溢出。此时，将数组大小擴大 1 倍需要  $2^i$  个网络硬币。幸运的是，这些硬币能够在内存单元从  $2^{i-1}$  到  $2^i - 1$  中找到（见图 5-14）。注意到前一次溢出出现在当元素个数第一次比  $2^{i-1}$  大时，所以，在单元  $2^{i-1}$  到  $2^i - 1$  中存储的网络硬币还未消费。因此，我们有了一个有效的摊销方案：每个操作索价 3 个网络硬币，所有运行时间都用硬币来支付，即我们可以用  $3n$  个网络硬币来支付  $n$  次增添操作。换句话说，每个增添操作的摊销运行时间为  $O(1)$ ；因此， $n$  次增添操作的总体运行时间为  $O(n)$ 。

### 大小按几何增长

虽然在命题 5-1 的证明中，我们每次都是把数组擴大 1 倍，但是对“数组大小以任意几何增长级数（见 2.4.2 节对几何级数的讨论）擴大，每次操作的摊销运行时间仍为  $O(1)$ ”这一结论是可以证明的。当选定了几何基数时，在运行效率和内存使用之间便存在一个折中问题。例如，当基数为 2 时（即数组擴大两倍），假如最后一个插入操作使得数组大小发生改变，则数组的大小本质上会变为其需要的 2 倍来结束该事件。如果不希望最后浪费太多内存，可以让数组当前大小仅增大 25%（即几何基数为 1.25），这种做法会在中间出现更多的调整数组大小的事件。使用一个更大的常数，例如在命题 5-1 的证明中使用的常数为每个操作需要 3 个网络硬币，我们仍可能证明摊销运行时间为  $O(1)$ （见练习 C-5.15）。证明的关键是：增添的内存大小是否正比于当前数组大小。

### 避免使用等差數列

为了避免一次扩充太大空间，可能会对动态数组执行这样的策略：每次要调整数组大小时，都要预留固定数量的额外单元。不幸的是，这种策略的整体性能明显糟糕。在极端情况下，如果每次增加一个单元，则会导致每个增添操作都将调整数组大小，继而就是类似地求和  $1 + 2 + 3 + \dots + n$ ，所以总体运行时间为  $\Omega(n^2)$ 。如图 5-15 所示，如果每次增加 2 个或 3 个单元也只是稍微改善，但总体运行时间仍为  $n^2$ 。

每次调整大小时都采用固定的增量，因此中间数组大小将会成等差數列，正如命题 5-2 所示，总体运行时间在操作个数上表现为平方。从直观上讲，即使每次增加 1000 个单元，对于大数据集来说，也无济于事。

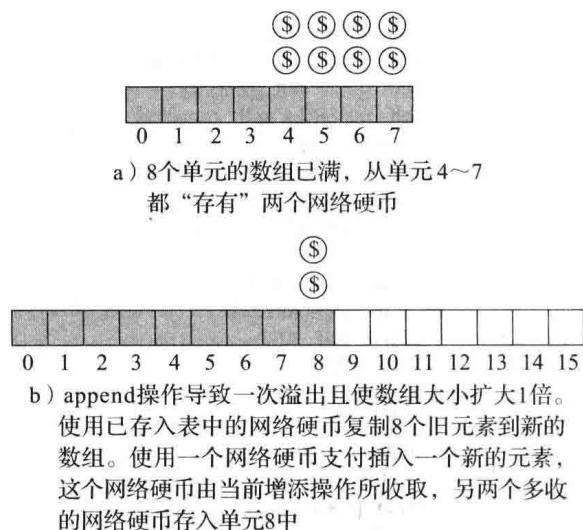
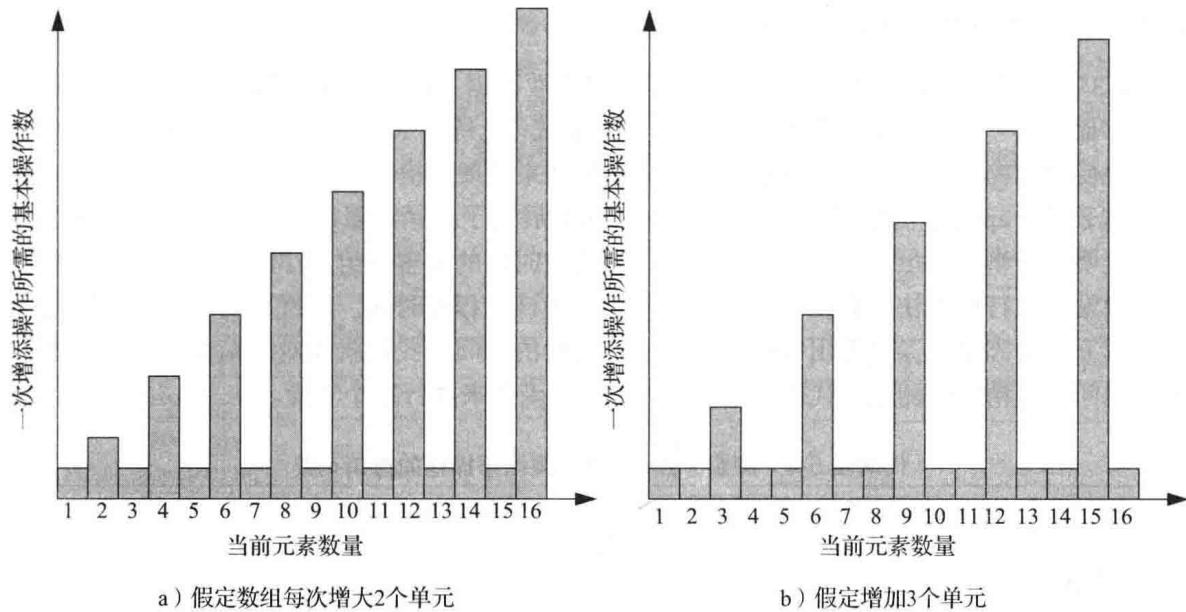


图 5-14 在动态数组中执行一系列增添操作的示意图



a) 假定数组每次增大2个单元

b) 假定增加3个单元

图 5-15 对动态数组采用等差数列进行一系列 append 操作所需要的运行时间

**命题 5-2：**对初始为空的动态数组执行连续  $n$  个增添操作，若每次调整数组大小时采用固定的增量，则运行时间为  $\Omega(n^2)$ 。

**证明：**设  $c > 0$ ，表示每次调整数组大小时的固定增量。在连续的  $n$  个 append 操作中，时间将会花费在分别初始化大小为  $c, 2c, 3c, \dots, mc$  的数组上面，其中  $m = \lceil n/c \rceil$ ，因此，总体运行时间将会正比于  $c + 2c + 3c + \dots + mc$ 。根据命题 3-3，得出和为

$$\sum_{i=1}^m ci = c \cdot \sum_{i=1}^m i = c \frac{m(m+1)}{2} \geq c \frac{\frac{n}{c} \left( \frac{n}{c} + 1 \right)}{2} \geq \frac{n^2}{2c}$$

因此，执行  $n$  个 append 操作花费的时间为  $\Omega(n^2)$ 。 ■

从命题 5-1 和 5-2 中得到一个教训：算法设计中，一个细微的差异在渐近性能上能表现出巨大的不同，细致的分析在设计数据结构中能起到重要的作用。

### 内存使用和紧凑数组

当对动态数组增添数据时，这种按几何增长的模式所带来的另一结果是：最终数组大小都确保能正比于元素总个数。也就是说，数据结构占用  $O(n)$  的内存，这是数据结构一个非常理想的属性。

假如一个容器，例如一张 Python 列表，能够提供删除一个或多个元素的操作，那就更要注意确保动态数组占用  $O(n)$  的内存。风险是：重复的插入操作可能会导致底层数组肆意增大，当许多元素被删除后，元素的实际数量与数组大小之间便不存在正比关系。

有时，会对这种数据结构采用一种健壮的实现方式——紧凑底层数组，在此期间，单个操作都保持  $O(1)$  的摊销绑定。然而，注意确保在扩充和收缩底层数组时，结构不能摊销（改变），因为在这种情况下，摊销绑定将不能实现。在练习 C-5.16 中，我们探索一种策略：无论实际元素个数比数组大小的  $1/4$  少多少，我们都对半平分数组，这样能确保数组大小至少是元素个数的 4 倍。在练习 C-5.17 和 C-5.18 中，我们将探究这种策略的摊销分析。

### 5.3.3 Python 列表类

5.3节开篇处的代码段5-1和5-2给出了实验性证据：Python列表类使用动态数组的形式来存储内容。然而，对中间数组大小的细致测试（见练习R-5.2和C-5.13）表明，Python既不是使用纯粹的几何级数，也不是使用等差数列来扩展数组。

这表明，`append`方法的Python实现很清晰地展现了摊销常量时间的行为。我们可以用实验证明这一事实。虽然我们应该关注一些更花费时间的调整数组大小的操作，但单个增添操作通常都执行得太快，以至于我们很难精确测量该过程的时间。通过对初始为空的列表执行连续 $n$ 个增添操作，并算出每个操作所平均花费的时间，我们就能对摊销花费在每个操作上的时间做更精准的测量。代码段5-4给出了一个函数来执行这个实验。

**代码段 5-4 测量 Python 列表类增添操作的摊销花费**

---

```

1 from time import time # import time function from time module
2 def compute_average(n):
3 """Perform n appends to an empty list and return average time elapsed."""
4 data = []
5 start = time() # record the start time (in seconds)
6 for k in range(n):
7 data.append(None)
8 end = time() # record the end time (in seconds)
9 return (end - start) / n # compute average per operation

```

---

从技术上说，从开始到结束所耗费的时间，除了调用`append`函数的时间，还包括维持循环迭代的时间。如表5-2所示，随着 $n$ 值的增大，给出了该实验的实证结果。我们看到较小的数据集往往会有更大的平均花费时间，也许部分原因在于循环的开销。在使用这种方式测量摊销花费时，也会产生一些自然偏差，因为最终调整大小都跟 $n$ 有关会对其产生影响。从整体来看，每个`append`操作的摊销时间都独立于 $n$ ，这点似乎很明确。

**表 5-2 通过观察开始为空的列表连续执行 $n$ 次调用后，得出增添操作以微秒为单位进行测量的平均运行时间**

| $n$           | 100   | 1000  | 10 000 | 100 000 | 1 000 000 | 10 000 000 | 100 000 000 |
|---------------|-------|-------|--------|---------|-----------|------------|-------------|
| $\mu\text{s}$ | 0.219 | 0.158 | 0.164  | 0.151   | 0.147     | 0.147      | 0.149       |

## 5.4 Python 序列类型的效率

在上一节中，我们依据执行策略和效率，初步学习了Python列表(`list`)类的基础内容。在本节中，我们继续检测所有Python序列类型的性能。

### 5.4.1 Python 的列表和元组类

列表类的`non mutating`行为是由元组(`tuple`)类所支持的。我们注意到元组比列表的内存利用率更高，因为元组是固定不变的，所以没必要创建拥有剩余空间的动态数组。表5-3给出了列表和元组类中`non mutating`行为的渐近效率。下面对其中的内容进行解释。

#### 常量时间操作

实例的长度之所以能在常量时间内得到，是因为该实例明确包含了这一状态信息。通过访问底层数组，保证了`data[j]`的常量时间效率。

表 5-3 列表和元组类中 non mutating 行为的渐近性能。标识符 data、data1 和 data2 表示列表或元组类的实例， $n$ 、 $n_1$ 、 $n_2$  代表它们各自的长度。对于该容器的检索和 index 方法， $k$  表示被搜索值在最左边出现时的索引（假如没有该值，那么  $k = n$ ）。在两个序列间进行比较，当  $n_1$  不等于  $n_2$  时，我们用  $k$  表示最左边的索引；否则，令  $k = \min(n_1, n_2)$

| 操作                                             | 运行时间           |
|------------------------------------------------|----------------|
| len(data)                                      | $O(1)$         |
| data[j]                                        | $O(1)$         |
| data.count(value)                              | $O(n)$         |
| data.index(value)                              | $O(k + 1)$     |
| value in data                                  | $O(k + 1)$     |
| data1 == data2<br>(similarly !=, <, <=, >, >=) | $O(k + 1)$     |
| data[j:k]                                      | $O(k - j + 1)$ |
| data1 + data2                                  | $O(n_1 + n_2)$ |
| C * data                                       | $O(cn)$        |

### 搜寻值的出现

每个 count、index 和 \_\_contains\_\_ 方法均从左往右迭代遍历序列。实际上，2.4.3 节的代码段 2-14 演示了这些行为是怎样被实现的。值得注意的是，当执行 count 方法时，必须循环遍历整个序列。当检索该容器中是否存在某个元素或者确定某个元素下标时，假如该元素存在，一旦从左开始第一次找到它，便立即退出循环。因此，count 方法需要检测序列的  $n$  个元素，而 index 和 \_\_contains\_\_ 方法只有在最坏的情况下才会检测  $n$  个元素，但往往都会更快。我们可以给出实验证据：设  $data = list(range(10\ 000\ 000))$ ，在 data 中找 5，在 data 中找 9 999 995，或者甚至失败的测试，如在 data 中找 -5，比较这些测试之间的相对效率。

### 字典比较

两个序列之间的对比被定义为字典。在最坏的情况下，评估这一情况需要运行时间正比于两序列中长度较短序列的迭代（因为当一个序列结束时，字典结果已能被确定）。而在一些情况下，能更高效地评估测试结果。例如，若评估  $[7, 3, \dots] < [7, 5, \dots]$ ，很明显，不用再测试列表剩余部分便已知结果是 True，因为左运算对象的第二个元素严格小于右运算对象的第二个元素。

### 创建新的实例

表 5-3 的后三个行为是在一个或多个原有实例的基础上构造的一个新实例。在所有情况下，运行时间都取决于构造和初始化实例所耗费的时间，因此，渐近行为正比于该实例的长度。于是，我们发现数据段  $[6\ 000\ 000 : 6\ 000\ 008]$  能够被立即构建成功，因为它仅有 8 个元素。数据段  $[6\ 000\ 000 : 7\ 000\ 000]$  有一百万个元素，因此要花费更多的时间去创建。

### 变异行为

表 5-4 描述了 list 类变异行为的效率。最简单的行为是  $data[j] = val$ ，且该行为被特殊的 \_\_setitem\_\_ 方法所支持。此行为在最坏情况下的运行时间为  $O(1)$ ，因为其仅用一个新值替换列表的一个元素。其他元素不受影响且底层数组的大小不变。值得分析的更有趣的行为是向列表中增添元素或从列表中删除元素。

表 5-4 列表类变异行为的渐近性能。data、data1 和 data2 表示列表类实例， $n_1$ 、 $n_2$  代表它们各自的长度

| 操作                    | 运行时间             |
|-----------------------|------------------|
| data[j] = val         | $O(1)$           |
| data.append(value)    | $O(1)^*$         |
| data.insert(k, value) | $O(n - k + 1)^*$ |
| data.pop()            | $O(1)^*$         |
| data.pop(k)           | $O(n - k)^*$     |
| del data[k]           | $O(n)^*$         |
| data.remove(value)    | $O(n)^*$         |
| data1.extend(data2)   | $O(n_2)^*$       |
| data1 += data2        | $O(n)^*$         |
| data.reverse()        | $O(n)$           |
| data.sort()           | $O(n \log n)$    |

\* 摊销

### 向列表中增添元素

在 5.3 节中，我们充分探讨了 append 方法。在最坏的情况下，因为底层数组需要调整，因此运行时间为  $\Omega(n)$ ，但在摊销情况下，运行时间为  $O(1)$ 。列表同样支持 insert(k, value) 这一方法，此方法将给定的值插入列表索引  $0 \leq k \leq n$  的位置，该位置通过将所有后续元素向前移动一个单位得到。为了解释清楚，在代码段 5-3 介绍的 DynamicArray 类的语义下，代码段 5-5 给出了此方法的一种实现方式，使用了代码段 5-3 的 DynamicArray 类。在分析此过程的效率时有两个复杂因素。首先，我们注意到增添一个元素需要调整动态数组大小。这部分工作对于每个 append 操作来说，在最坏情况下运行时间为  $\Omega(n)$ ，但摊销时间仅为  $O(1)$ 。insert 操作的另一个代价是移动元素来为新元素提供位置。此过程的时间取决于新元素的索引以及由此产生的移动后续元素的个数。

### 代码段 5-5 DynamicArray 类 insert 方法的实现

```

1 def insert(self, k, value):
2 """Insert value at index k, shifting subsequent values rightward."""
3 # (for simplicity, we assume 0 <= k <= n in this version)
4 if self._n == self._capacity: # not enough room
5 self._resize(2 * self._capacity) # so double capacity
6 for j in range(self._n, k, -1): # shift rightmost first
7 self._A[j] = self._A[j-1]
8 self._A[k] = value # store newest element
9 self._n += 1

```

如图 5-16 所示，循环过程将索引  $n - 1$  的引用复制到索引  $n$  内，将索引  $n - 2$  的引用复制到索引  $n - 1$  内，如此往复，直到将索引  $k$  的引用复制到索引  $k + 1$  内。插入索引  $k$  内需要的总摊销时间为  $O(n - k + 1)$ 。

在 5.3.3 节中，当探讨 Python 的 append 方法的效率时，我们做了这样的实验：在不同大小的列表上重复调用，计算耗费时间的平均值（见代码段 5-4 和表 5-2）。我们将用 insert 方法重复该实验，并尝试用三种不同的访问模式。

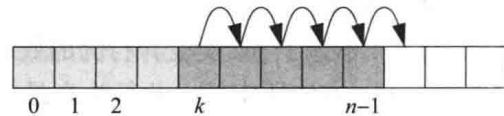


图 5-16 在动态数组中索引为  $k$  的位置开辟空间并插入新元素

- 第一种情况，我们在列表的开始位置进行重复插入，

```
for n in range(N):
 data.insert(0, None)
```

- 第二种情况，我们在列表接近中间的位置进行重复插入，

```
for n in range(N):
 data.insert(n // 2, None)
```

- 第三种情况，我们在列表的结束位置进行重复插入，

```
for n in range(N):
 data.insert(n, None)
```

表 5-5 给出了实验的结果，记录了每个操作的平均时间（不是整个循环的总时间）。正如所预料的那样，我们看到在列表的开始位置做插入是最费时的，每个操作插入时间呈线性，因而运行时间仍为  $\Omega(n)$ 。在结束位置做插入表现为  $O(1)$  的运行时间，类似于增添操作。

表 5-5 通过观察在初始为空的列表内进行的连续  $N$  次调用，得出  $\text{insert}(k, \text{val})$  的平均运行时间，单位为微秒。令  $n$  表示当前列表的大小（与最终列表大小作对照）  
(单位:  $\mu\text{s}$ )

|            | $N$   |       |        |         |           |
|------------|-------|-------|--------|---------|-----------|
|            | 100   | 1 000 | 10 000 | 100 000 | 1 000 000 |
| $k=0$      | 0.482 | 0.765 | 4.014  | 36.643  | 351.590   |
| $k=n // 2$ | 0.451 | 0.577 | 2.191  | 17.873  | 175.383   |
| $k=0$      | 0.420 | 0.422 | 0.395  | 0.389   | 0.397     |

### 从列表中删除元素

Python 的 list 类提供了几种从列表中删除元素的方法。调用 `pop()` 删除列表的最后一个元素。这是最高效的，因为其他所有元素都保持在自己的原有位置。这虽然是一个效率为  $O(1)$  的操作，但由于 Python 不定时地收缩底层数组以节省内存，因此绑定是摊销的。

带参数的方法 `pop(k)` 能够删除列表中索引为  $k < n$  的元素，并把所有后续元素往左移动，以填补由删除操作导致的空缺。该操作的效率为  $O(n < k)$ ，因为移动的数量取决于索引  $k$  的选择，如图 5-17 所示。注意，这表明 `pop(0)` 是最耗时的调用，运行时间为  $\Omega(n)$ 。（见练习 R-5.8 中的实验）

`list` 类提供了另一种名为 `remove` 的方法，该方法允许调用者指定要删除的值（不是值的索引）。正式地说，该方法仅删除列表中第一次出现的指定值，当未找到该值时，生成一个 `ValueError` 异常。在代码段 5-6 中，再次利用 `DynamicArray` 类做说明，给出此行为的一种实现方式。

有趣的是，对于 `remove` 方法来说，没有“高效”的情况：每一次调用都需要  $\Omega(n)$  的运行时间。该过程部分工作用于从列表开头进行搜索，直至找到索引为  $k$  的值，而剩余的从  $k$  到最后的迭代用于往左移动元素。该线性行为能用实验观察到（见练习 C-5.24）。

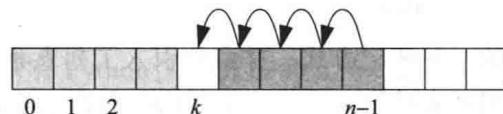


图 5-17 删除动态数组中索引为  $k$  的元素

### 代码段 5-6 对 `DynamicArray` 类的 `remove` 方法的一种实现

```
1 def remove(self, value):
2 """ Remove first occurrence of value (or raise ValueError). """
3
4 for i in range(len(self)):
5 if self[i] == value:
6 self._shift_left(i)
7 break
8
9 else:
10 raise ValueError(f'Value {value} not found')
```

```

3 # note: we do not consider shrinking the dynamic array in this version
4 for k in range(self._n):
5 if self._A[k] == value: # found a match!
6 for j in range(k, self._n - 1): # shift others to fill gap
7 self._A[j] = self._A[j+1]
8 self._A[self._n - 1] = None # help garbage collection
9 self._n -= 1 # we have one less item
10 return # exit immediately
11 raise ValueError('value not found')

```

## 扩展列表

Python 提供了一个名为 `extend` 的方法，该方法能将一张列表的所有元素增添到另一张列表的末尾。在作用上，调用 `data.extend(other)` 输出的结果和如下代码输出的结果相同：

```

for element in other:
 data.append(element)

```

在任何情况下，运行时间都正比于另一张列表的长度，并且之所以摊销，是因为第一张列表的底层数组需要调整大小以容纳增添的元素。

在实践中，相对于重复调用 `append` 方法，我们倾向于选择 `extend` 方法，因为渐近分析中隐含的常数明显更小。`Extend` 方法效率更高缘于三个方面：首先，使用合适的 Python 方法总会有一些优势，因为这些方法通常使用本地编译语言进行执行（不是用作解释 Python 代码）。其次，与调用很多独立的函数相比，调用一个函数完成所有工作的开销更小。最后，`extend` 提升的效率来源于更新列表的最终大小能提前计算出。假如第二个数据集是非常大的，当重复调用 `append` 方法时，底层动态数组会有多次调整大小的风险。若调用一次 `extend` 方法，最多执行一次调整操作。练习 C-5.22 用实验探究了这两种方法的相对效率。

## 构造新列表

有几种用于构造新列表的语法。在几乎所有情况下，该行为的渐近效率在创建列表的长度方面是线性的。然而，与前面讨论的 `extend` 方法的情况类似，在实际效率上有明显的不同。

在 1.9.2 节中，使用一个诸如 `squares = [k*k for k in range(1, n+1)]` 的例子作为

```

squares = []
for k in range(1, n+1):
 squares.append(k*k)

```

的一种速记方式，并由此引入了列表推导式（list comprehension）的话题。实验将会证明使用列表推导式语法比不断增添数据来建表速度明显更快（见练习 C-5.23）。

类似地，使用乘法操作初始化一张具有固定值的列表，也是一种很常见的 Python 风格。例如，语句 `[0]*n` 生成一张长度为  $n$ 、所有值都等于 0 的列表。这样做不但语法简便，而且比逐步构造这样的表效率更高。

### 5.4.2 Python 的字符串类

在 Python 中，字符串是非常重要的。我们在 1.3 节中，通过对不同运算符的讨论，介绍了字符串的使用。在附录 A 的表 A-1 ~ 表 A-4 中，给出了该类已命名方法的综合概要。本节中，我们不再正式分析每个行为的效率，却希望在一些值得注意的问题上做一些批注。一般来说，我们用  $n$  表示字符串的长度。对于那些需要另一个字符串作为样例的操作，我们用  $m$  表示样例字符串的长度。

对许多行为的分析常靠直觉。例如，生成新字符串的方法（如 `capitalize`、`center`、`strip`

方法)需要的时间与该方法所生成的字符串长度之间呈线性关系。字符串的许多行为(例如, `islower`)以布尔条件进行测试,在最坏情况下需要检查 $n$ 个字符,此时运行时间为 $\Omega(n)$ 。但是当结果很明显时,循环就很快结束(例如,若第一个字符是大写字母,`islower`能立即返回`False`)。比较操作符(例如,`==`,`<`)也属于这一种情况。

### 样例匹配

有一些更有趣的行为,从算法角度来说,这些行为在某种程度上取决于在较大的字符串中找到字符样例。所要寻找的目标是这些方法的核心(例如`_contains_`、`find`、`index`、`count`、`replace`和`split`)。字符串算法将会是第13章的课题,13.2节的重心是特别著名的模式匹配(pattern matching)问题。有一种运行时间为 $O(mn)$ 的简单实现方法:我们为此样例考虑了 $n-m+1$ 种可能的起始索引,每个起始索引都需要花费 $O(m)$ 的运行时间用于检查该样例是否匹配。而在13.2节中,我们将会编写一个算法,用于在 $O(n)$ 时间内寻找最大长度为 $n$ 的字符串中长度为 $m$ 的字符串。

### 组成字符串

最后,我们想对几种能组成大字符串的方法进行评论。接下来做一个学术练习,假定有一个较大的字符串`document`,我们的目标是生成一个新的字符串`letters`,该字符串仅包含原字符串的英文字母字符(即,将空格、数字、标点符号除去)。我们或许会采用如下循环来得到结果,

```
WARNING: do not do this
letters = '' # start with empty string
for c in document:
 if c.isalpha():
 letters += c # concatenate alphabetic character
```

虽然上面的代码段实现了该目标,但其效率可能非常低下。因为字符串大小固定,指令`letters += c`很可能计算串联部分`letters + c`,并把结果作为新的字符串实例且重新分配给标识符`letters`。构造新字符串所用时间与该字符串的长度成正比。假如最终结果有 $n$ 个字符,连续串联计算所花费的时间与所谓的求和公式 $1 + 2 + 3 + \dots + n$ 成正比,因此,运行时间为 $\Omega(n^2)$ 。

这类效率低的代码在Python中很普遍,大概跟代码的自然外观有点关系,且容易对`+=`操作符如何与字符串连接产生误解。Python解释器后来的一些实现方法中对开发进行了最优化,能够允许这类代码的运行时间为线性,但不是所有Python实现方法都支持。最优化如下:指令`letters += c`之所以会产生新的字符串实例,是因为假如程序中有另一个变量要引用原字符串,则原字符串必须保持不变。另一方面,假如Python知道在该问题中对该字符串没有其他引用,但通过直接改变字符串(作为一个动态数组)可以更高效地实现`+=`。当发生上述情况时,Python解释器已经为每个对象包含了所谓的引用计数器。该计数器部分用于确定某个对象是否能被垃圾回收(见15.1.2节)。但在此情形中,计数器给出了一种方法,用于检测是否存在对字符串的其他引用,因此,允许最优化。

保证能在线性时间内组成字符串的另一个更标准的Python术语是使用临时表存储单个数据,然后使用字符串类的`join`方法组合最终结果。将此技巧用于我们之前例子将会如下编写:

```
temp = [] # start with empty list
for c in document:
 if c.isalpha():
 temp.append(c) # append alphabetic character
letters = ''.join(temp) # compose overall result
```

该方法能确保运行时间为  $O(n)$ 。首先，我们注意到连续  $n$  次 append 调用共需要  $O(n)$  的运行时间，其运行时间可以根据此操作的摊销花费定义得出。最后对 join 的调用也能保证在组合字符串的最终长度上花费的时间呈线性。

正如在上一节末尾所讨论的那样，我们使用列表推导式语法来创建临时表，而不是重复调用 append 方法，能够进一步提高实际执行速度。方案如下：

```
letters = ''.join([c for c in document if c.isalpha()])
```

还有更好的方法——我们使用生成器理解可以完全避免使用临时表：

```
letters = ''.join(c for c in document if c.isalpha())
```

## 5.5 使用基于数组的序列

### 5.5.1 为游戏存储高分

我们学习的第一个应用是为某款视频游戏存储一列高分条目。这是许多必须存储一系列对象的应用程序的代表。我们可能很容易选择为医院的病人存储记录或者登记某足球队队员的姓名。然而，这里我们将关注存储高分条目，这是一个简单且数据丰富的应用程序，足以表示一些重要的数据结构概念。

刚开始，我们考虑在对象中存储什么信息来表示高分条目。显然，信息中一定含有一个表示分数的整数，我们用 `_score` 来表示。另一个有用的信息是得分者姓名，我们用 `_name` 表示。我们能继续增加字段表示得分的数据或者得分的游戏统计的字段。但我们忽略一些使得例子变得容易的细节。在代码段 5-7 中，我们给出一个 Python 类 `GameEntry`，用于表示游戏条目：

**代码段 5-7 一个简单 GameEntry 类的 Python 代码。其中包括返回游戏条目对象的姓名和分数的方法，还有返回表示该条目的字符串的方法**

---

```

1 class GameEntry:
2 """Represents one entry of a list of high scores."""
3
4 def __init__(self, name, score):
5 self._name = name
6 self._score = score
7
8 def get_name(self):
9 return self._name
10
11 def get_score(self):
12 return self._score
13
14 def __str__(self):
15 return '{0}, {1}'.format(self._name, self._score) # e.g., '(Bob, 98)'

```

---

### 存储高分的类

为了存储一系列高分，我们编写一个类并将其命名为 `Scoreboard`，一个 `scoreboard` 对象只能存储一定数量的高分，一旦达到存储界限，新的分数必须严格大于得分板上最低的“最高分”才能记入 `scoreboard`。理想的 `scoreboard` 的长度取决于游戏，可能为 10、50 或者 500。因为这个长度非常依赖游戏，我们将它指定为 `Scoreboard` 结构的参数。

在内部，我们将会使用名为 `_board` 的 Python 列表（list）来管理表示高分的 `GameEntry` 实例。因为希望 `scordboard` 最终能被填满，所以使初始化列表尽可能大以便能存储最多的分

数，但起初将所有条目都设为 None。最初给列表分配了最大化的容量，因此执行过程中不再需要调整大小。当添加条目时，我们将会从列表的索引 0 开始，从最高分到最低分依次存储。图 5-18 所示即为这种数据结构的一个典型样例。

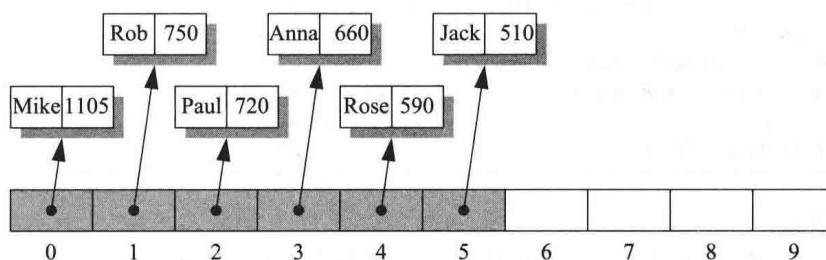


图 5-18 一张长度为 10 有序列表的示意图，从索引 0 ~ 5，共存储 6 个 GameEntry 对象的引用，其余单元仍为 None

代码段 5-8 给出了 Scoreboard 类的一个完整的 Python 实现方法。构造函数非常简单。下面的语句

```
self._board = [None] * capacity
```

创建一张所需长度的列表，而所有条目都是 None。其中还包含一个额外的实例变量 `_n`，用于表示表内当前的实际条目数。为了方便，我们的类也支持 `__getitem__` 方法，此方法通过给定的索引，使用 `board[i]` 能检索获得一个条目（或者假如没有这样的条目就返回 `None`），我们也支持简单的 `__str__` 方法，该方法返回用每行一个条目表示整个 scoreboard 的字符串。

代码段 5-8 Scoreboard 类的 Python 代码，其中包含一系列有序的分数，这些分数代表 GameEntry 对象

```

1 class Scoreboard:
2 """Fixed-length sequence of high scores in nondecreasing order."""
3
4 def __init__(self, capacity=10):
5 """Initialize scoreboard with given maximum capacity.
6
7 All entries are initially None.
8
9 self._board = [None] * capacity # reserve space for future scores
10 self._n = 0 # number of actual entries
11
12 def __getitem__(self, k):
13 """Return entry at index k."""
14 return self._board[k]
15
16 def __str__(self):
17 """Return string representation of the high score list."""
18 return '\n'.join(str(self._board[j]) for j in range(self._n))
19
20 def add(self, entry):
21 """Consider adding entry to high scores."""
22 score = entry.get_score()
23
24 # Does new entry qualify as a high score?
25 # answer is yes if board not full or score is higher than last entry
26 good = self._n < len(self._board) or score > self._board[-1].get_score()

```

```

27
28 if good:
29 if self._n < len(self._board): # no score drops from list
30 self._n += 1 # so overall number increases
31
32 # shift lower scores rightward to make room for new entry
33 j = self._n - 1
34 while j > 0 and self._board[j-1].get_score() < score:
35 self._board[j] = self._board[j-1] # shift entry from j-1 to j
36 j -= 1 # and decrement j
37 self._board[j] = entry # when done, add new entry

```

### 增添一个条目

Scoreboard 类中最有趣的方法是 `add` 方法，该方法能够考虑到将新的条目添加至 scoreboard 中。要记住：每个条目不一定要绑定一个高分。假如 `board` 还没有满，任何一个新的条目都可以被记录。一旦 `board` 已满，新的条目只有严格大于一个或多个分数时，才能被记入，特别是，`scoreboard` 中的最后一个条目是最低的高分。

当考虑新的分数时，我们先要确定该分数是否满足高分的条件。假如满足，若 `board` 未满，我们便增加有效分数的个数 `_n`。假如 `board` 已满，增添新的高分将会导致某个其他高分从 `scoreboard` 中被删除，因此条目的总数保持不变。

为了正确地将新条目放入列表中，最后的工作是将较低分往后移动一个位置（当 `socreboard` 已满时，最低分将会被完全删除）。这个过程与在前面 `list` 类的 `insert` 方法的实现方式很类似。在 `scoreboard` 情形中，不需要移动任何保存在数组尾部的 `None` 引用，因此该过程能按图 5-19 所示的那样进行。

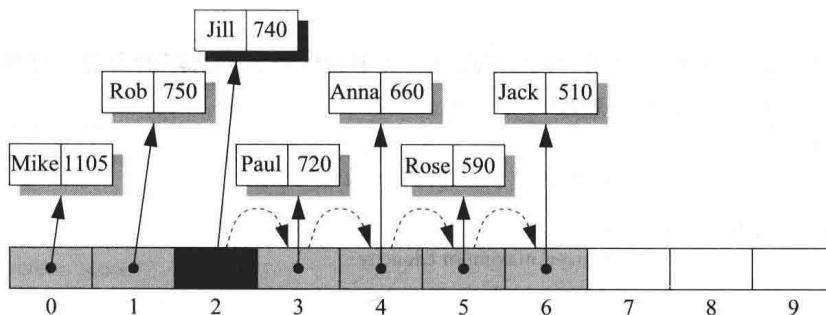


图 5-19 为 Jill 增加一个新的 GameEntry 到 scoreboard。为了给新的引用提供空间，我们必须将分数比新分数低的游戏条目的引用往右移动一个单元，然后我们能将新条目增添到索引 2 的位置

为了完成最后的步骤，我们首先考虑索引 `j = self._n - 1`，当完成此操作后，该索引指向最后一个 `GameEntry` 实例。`j` 要么是新条目的正确索引，要么是一个或者多个暂时有更低分数的条目的索引。当循环执行到第 34 行时，只要索引 `j - 1` 条目的分数比新条目分数低，就把索引往右移动，`j` 自减 1。

### 5.5.2 为序列排序

上一节中，我们学习了一个应用程序：在序列中的给定位置增添对象，通过移动其他元素保持先前顺序不变。本节中，我们使用类似的技巧解决排序问题，即把开始元素无序的序列通过重新排序变成非递减序列。

## 插入排序算法

本书将介绍几种排序算法，其中大部分将在第 12 章中介绍。作为准备，本节将介绍一种友好简单的排序算法——插入排序。对基于数组的序列，该算法按如下方式执行。我们从数组的第一个元素开始。一个元素本身已排序。接着我们考虑数组的下一个元素。假如它比第一个元素小，我们就把这两个数进行交换。之后我们考虑数组中的第三个元素，把它与左边前两个元素进行比较和交换，直至找到自己的位置。考虑第四个元素，把它与左边前三个元素进行比较和交换，直至找到正确的位置。对第五、第六及其余元素继续执行上述操作，直至整个数组被完全排序。我们可以用伪代码描述插入排序算法，示例如代码段 5-9 所示。

代码段 5-9 插入排序算法的高级语言描述

**Algorithm InsertionSort(A):**

**Input:** An array A of n comparable elements

**Output:** The array A with elements rearranged in nondecreasing order

**for** k from 1 to n – 1 **do**

    Insert A[k] at its proper location within A[0], A[1], …, A[k].

这是对插入排序的一种简单高级的描述。假如回顾 5.5.1 节中的代码段 5-8，我们会看到在高分列表中插入新条目的操作与在插入排序算法中插入正被考虑的元素的操作几乎相同（唯一不同是游戏高分从高到低已经排序）。在代码段 5-10 中，我们给出插入排序算法的一种 Python 实现方法，使用外层循环轮流考虑每个元素，内层循环移动正被考虑的元素，将其移动到其左边（已排序）子数组的合适位置。图 5-20 所示即为插入排序算法运行过程的示例。

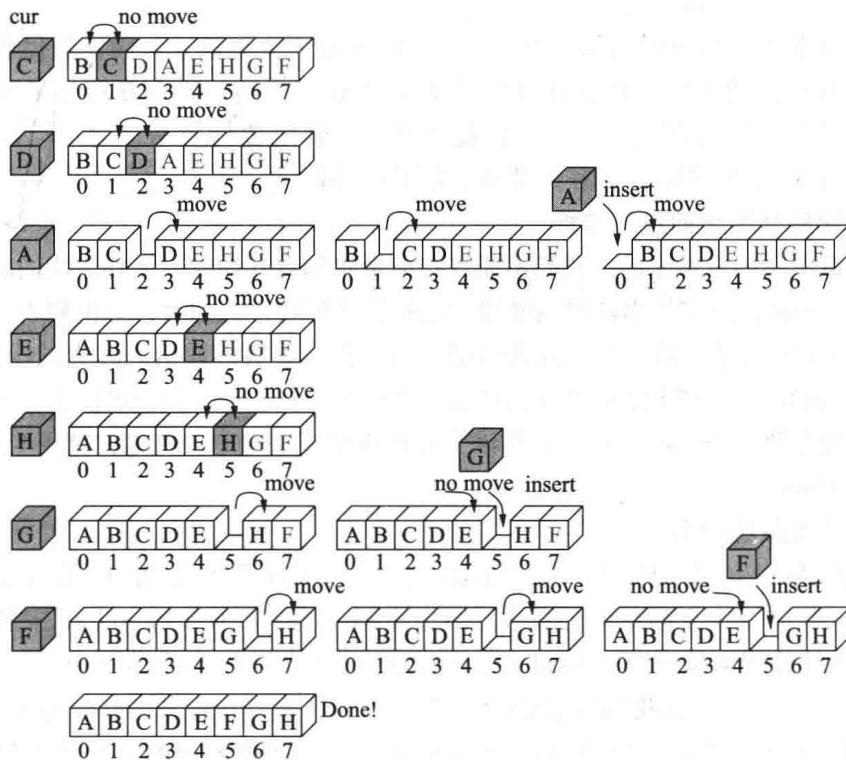


图 5-20 在 8 字符数组中执行插入排序算法。每一行对应外部循环的一次迭代，一行内的每一次复制对应内层循环的一次迭代。正被插入的当前元素在数组中被突出显示，并作为当前值

插入排序的嵌套循环在最坏的情况下会导致 $\Omega(n^2)$ 的运行时间。如果数组最初是反序，则工作量最大。另外，如果初始数组已基本排序或已完全排序，则插入排序运行时间为 $O(n)$ ，因为内层循环迭代次数很少或完全没有迭代。

代码段 5-10 在列表中执行插入排序的 Python 代码

---

```

1 def insertion_sort(A):
2 """Sort list of comparable elements into nondecreasing order."""
3 for k in range(1, len(A)):
4 cur = A[k] # current element to be inserted
5 j = k # find correct index j for current
6 while j > 0 and A[j-1] > cur: # element A[j-1] must be after current
7 A[j] = A[j-1]
8 j -= 1
9 A[j] = cur # cur is now in the right place

```

---

### 5.5.3 简单密码技术

字符串和列表的一个有趣应用是密码学（cryptography），它是一种秘密信息的科学及其应用。这一领域研究加密的方法，即将称为明文的信息转换成称为密文的加密信息。同样，该领域也研究对应的解密方法，即将密文转变回原来的明文。

可以说最早的加密技术是凯撒密码（Caesar cipher），该技术以尤利乌斯·凯撒的名字命名，凯撒使用此技术保护重要军事情报。（所有凯撒情报都是用拉丁语写的，当然，这使得我们大多数人都不能阅读这些情报！）凯撒密码是一种简单隐藏情报的方法，这些情报用字母表组成单词的语言编写。

凯撒密码涉及替换情报中的每一个字母，用在字母表中继该字母固定数目后的字母进行替换。因此，在英语情报中，我们可以把每个 A 都用 D 替换，每个 B 都用 E 替换，每个 C 都用 F 替换，等等，即移动三个字母。以此类推，直到用 Z 替换 W。之后，我们将此替换模式循环，即将 X 用 A 替换，Y 用 B 替换，Z 用 C 替换。

#### 字符串和字符列表之间进行转换

给定的字符串是固定不变的，我们不能直接编辑实例对其加密。另外，我们的目标是产生一个新字符串。一种执行字符串转换的便捷方法是创建等效字符列表，编辑列表，然后将该列表重新组成（新）字符串。第一步可以通过把字符串作为参数传递给列表类的构造函数完成。例如，表达式 `list('bird')` 会得到 `['b', 'i', 'r', 'd']` 这样的结果。相应地，我们可以在空字符串上通过用字符列表作为参数调用 `join` 方法，并将该字符列表组成字符串。例如，调用 `"join(['b', 'i', 'r', 'd'])` 返回字符串 `'bird'`。

#### 使用字符作为数组索引

如果我们像数组索引那样为每个字母编码，那么 A 就是 0、B 是 1、C 是 2，等等，之后我们可以用  $r$  轮转写一个简单的公式表示凯撒密码：用字母  $(i + r) \bmod 26$  来替换每个字母  $i$ ，这里的  $\bmod$  就是模数运算子（modulo operator），当执行整除后返回余数。在 Python 中该运算子用 `%` 表示，这正是我们所需要的运算子，可以在字母表末尾处很容易地执行轮转，因为  $26 \bmod 26$  是 0， $27 \bmod 26$  是 1， $28 \bmod 26$  是 2。凯撒密码的解密算法与此相反——采用轮转，用每个字母前的第  $r$  个字母代替自己（即字母  $i$  被字母  $(i - r) \bmod 26$  替换）。

我们可以用另一个字符串指明替换规则来描述转换过程。举一个具体的例子，假设正在使用一个三字符轮转的凯撒密码。我们应该提前计算出用于替换 A ~ Z 每个字

母的字符串。比如，A 应该被 D 替换，B 被 E 替换，等等。26 个替换字母按顺序就是“DEFGHIJKLMNOPQRSTUVWXYZABC”。之后我们可以使用这个转换的字符串作为向导来加密情报。剩下的任务就是如何为原情报的每个字母快速地找到替换字母。

幸运的是，我们可以依据字符在 Unicode 中用整数代码点表示这一事实，且拉丁字母表中大写字母的代码点是连续的（为简单起见，我们限制只对大写字母加密）。Python 支持在整数代码点和单字符字符串之间进行转换的函数。尤其是将单字符字符串作为参数传递到函数 `ord(c)` 中，能得到该字节的整数代码点。相应地，将整数传入函数 `chr(j)` 中能得到其所对应的单字符字符串。

在凯撒密码中，为了确定某个字节的替换字符，我们需要将字符 'A' ~ 'Z' 分别映射为 0 ~ 25 的整数。执行此变换的公式即为  $j = \text{ord}(c) - \text{ord}('A')$ 。做一次检查，假如  $c$  为 'A'，我们得到  $j=0$ 。当  $c='B'$  时，我们发现其顺序值正好比 'A' 多 1，故它们相差 1。一般来说，由此计算得到的整数  $j$  能够在我们预先翻译的字符串中充当索引，如图 5-21 所示。

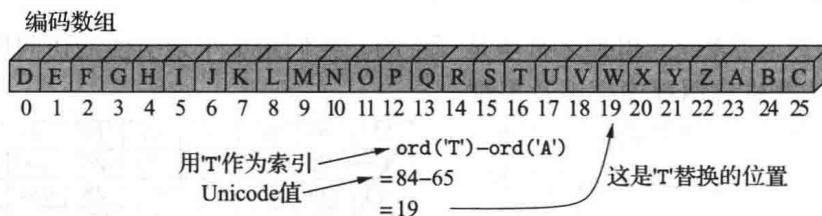


图 5-21 用大写字母作为索引值，演示凯撒密码加密的替换规则

代码段 5-11 给出了一个 Python 类，可以为凯撒密码赋予任意轮转值，并且证明了此用法。运行此程序时（执行一个简单测试），得到的输出结果如下：

```
Secret: WKH HDJOH LV LQ SODB; PHHW DW MRH'V.
Message: THE EAGLE IS IN PLAY; MEET AT JOE'S.
```

该类的构造函数为给定值建立了加密前后的字符串。加密和解密算法就像一双手，本质上是相同的，所以我们用不公开的实例方法 `_transform` 执行这两种算法。

#### 代码段 5-11 凯撒密码的一个完整 Python 类

```

1 class CaesarCipher:
2 """Class for doing encryption and decryption using a Caesar cipher."""
3
4 def __init__(self, shift):
5 """Construct Caesar cipher using given integer shift for rotation."""
6 encoder = [None] * 26 # temp array for encryption
7 decoder = [None] * 26 # temp array for decryption
8 for k in range(26):
9 encoder[k] = chr((k + shift) % 26 + ord('A'))
10 decoder[k] = chr((k - shift) % 26 + ord('A'))
11 self._forward = ''.join(encoder) # will store as string
12 self._backward = ''.join(decoder) # since fixed
13
14 def encrypt(self, message):
15 """Return string representing encrypted message."""
16 return self._transform(message, self._forward)
17
18 def decrypt(self, secret):
19 """Return decrypted message given encrypted secret."""
20 return self._transform(secret, self._backward)
21

```

```

22 def _transform(self, original, code):
23 """Utility to perform transformation based on given code string."""
24 msg = list(original)
25 for k in range(len(msg)):
26 if msg[k].isupper():
27 j = ord(msg[k]) - ord('A') # index from 0 to 25
28 msg[k] = code[j] # replace this character
29 return ''.join(msg)
30
31 if __name__ == '__main__':
32 cipher = CaesarCipher(3)
33 message = "THE EAGLE IS IN PLAY; MEET AT JOE'S."
34 coded = cipher.encrypt(message)
35 print('Secret: ', coded)
36 answer = cipher.decrypt(coded)
37 print('Message: ', answer)

```

## 5.6 多维数据集

在 Python 中，列表、元组和字符串是一维的。我们用单个索引便能访问序列中的每个元素。但许多计算机应用程序都涉及多维数据集。例如，计算机图形通常用二维或三维来建模。地理信息通常表示为二维，医学图像可以给出病人的三维扫描，公司估值通常基于大量的独立金融测试，这些均可以用多维数据来建模。二维数组有时也称为矩阵（matrix），我们可以用两个索引  $i$  和  $j$  指向矩阵中的单元。第一个索引通常表示行号，第二个表示列号，并且在计算机学中，这两个索引习惯上从 0 开始。图 5-22 所示为整数数据的二维数据集。这个数据集可以用来表示美国曼哈顿不同地区的商店数量。

在 Python 中，二维数据集通常表示为列表的列表。我们用多行列表表示二维数组，每行本身表示一张列表。例如，二维数据

|    |     |     |     |     |
|----|-----|-----|-----|-----|
| 22 | 18  | 709 | 5   | 33  |
| 45 | 32  | 830 | 120 | 750 |
| 4  | 880 | 45  | 66  | 61  |

|   | 0   | 1   | 2   | 3   | 4   | 5   | 6   | 7   | 8   | 9   |
|---|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 1 | 22  | 18  | 709 | 5   | 33  | 10  | 4   | 56  | 82  | 440 |
| 2 | 45  | 32  | 830 | 120 | 750 | 660 | 13  | 77  | 20  | 105 |
| 3 | 4   | 880 | 45  | 66  | 61  | 28  | 650 | 7   | 510 | 67  |
| 4 | 940 | 12  | 36  | 3   | 20  | 100 | 306 | 590 | 0   | 500 |
| 5 | 50  | 65  | 42  | 49  | 88  | 25  | 70  | 126 | 83  | 288 |
| 6 | 398 | 233 | 5   | 83  | 59  | 232 | 49  | 8   | 365 | 90  |
| 7 | 33  | 58  | 632 | 87  | 94  | 5   | 59  | 204 | 120 | 829 |
|   | 62  | 394 | 3   | 4   | 102 | 140 | 183 | 390 | 16  | 26  |

图 5-22 二维整数数据集的图示，共 8 行 10 列。行和列都是从 0 开始。假如这个数据集命名为 stores，则 stores[3][5] 的值为 100，stores[6][2] 的值为 632

在 Python 中可以按照如下形式存储：

```
data = [[22, 18, 709, 5, 33], [45, 32, 830, 120, 750], [4, 880, 45, 66, 61]]
```

这样表示的好处是：我们可以很自然地使用诸如 data[1][3] 这样的语法表示 1 行 3 列的数据，比如，外表中第二个条目 data[1] 本身也是一张表，因此是可索引的。

### 创建多维列表

为了快速初始化一张一维列表，一般使用诸如 data = [0]\*n 这样的语句来创建具有  $n$  个 0 的列表。在图 5-7 和图 5-8 中，我们从技术角度做了强调，这种方式创建了一张长度为  $n$  且所有条目都指向同一个整数实例的列表，但是这种混叠方式并没有取得多么有意义的结果，因为在 Python 中，int 类是固定不变的。

在创建列表的列表时，我们要更加小心。假如想要创建一张相当于有  $r$  行  $c$  列的二维整

数列表的列表，并把所有值都初始化为 0，我们可能会采用如下的错误语句

```
data = ([0] * c) * r # Warning: this is a mistake
```

$([0]^c)^r$  确实创建了一张有  $c$  个 0 的列表，但通过将列表乘  $r$ ，只会创建一张长度为  $r*c$  的一维列表，比如  $[2, 4, 6]^2$  创建为  $[2, 4, 6, 2, 4, 6]$  这样的列表。

还有一种更好一点但仍有问题的做法：创建一张包含  $n$  个 0 的列表作为自己唯一元素的列表，然后把这个列表乘以  $r$ ，即使用如下的语句创建：

```
data = [[0] * c] * r # Warning: still a mistake
```

这更加接近了，因为我们实际上确实得到了一个正式的列表的列表的结构。问题却是 `data` 列表的所有  $r$  个条目都指向了同一个实例，该实例即为有  $c$  个 0 的列表。图 5-23 所示即为这种混叠方式的示例。

这确实是一个问题。赋值一个条目，例如 `data[2][0] = 100`，将可能使得第二级列表的第一个条目指向新的值 100。而第二级列表的那个单元也表示 `data[0][0]` 的值，因为第 `data[0]` “行” 和 `data[2]` “行” 指向同一个第二级列表。

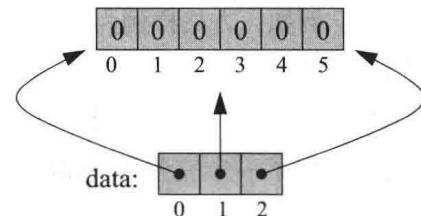


图 5-23 一个用  $3 \times 6$  的数据集作为列表的列表的错误表示，该数据集用语句 `data=[[0]*6]*3` 创建（为了简单起见，我们忽略了第二级列表的值为引用类型）

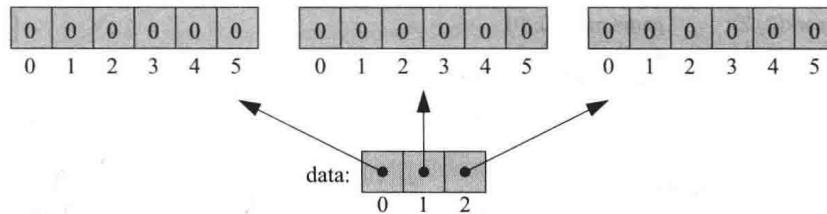


图 5-24  $3 \times 6$  数据集作为列表的列表的一种有效表示方法（为了简便起见，我们忽略了第二级列表值是引用类型这一事实）

为了能正确实例化二维列表，我们必须确保原始列表的每个单元都能指向一个独立的第二级列表。通过运用 Python 列表推导式语法能够实现实例化。

```
data = [[0] * c for j in range(r)]
```

该语句产生了一个有效配置，如图 5-24 所示。使用列表推导式语法，表达式  $[0]^c$  在循环的每次执行中都会重新评估。因此，我们得到  $r$  的不同的第二级列表，这正是我们想要的。（我们注意到语句中的变量  $j$  是不相关的，仅仅需要将循环迭代  $r$  次。）

## 二维数组和位置型游戏

许多计算机游戏，例如策略游戏、模拟游戏或第一人称战斗游戏，都是将对象放置于二维空间中。开发这类位置型游戏需要一种能表示二维“边界”的方法，在 Python 中，很自然就会选择列表的列表。

### 三连棋游戏

大多数学生都知道，三连棋是在  $3 \times 3$  的方格里玩的游戏。两个玩家——X 和 O——从 X 开始，轮流将他们各自的标志符号放入方格的某单元内。假如其中的任一玩家将己方符号