

方法。

- R-11.7 图 11-12 和图 11-14 中的 trinode 重组是不是会导致单旋转或双旋转?
- R-11.8 根据图 11-14b 绘制插入键为 52 的条目后的 AVL 树。
- R-11.9 根据图 11-14b 绘制移除键为 62 的条目后的 AVL 树。
- R-11.10 解释为什么使用 8.3.2 节的基于数组的表示对一个 n-node 二叉树执行一个旋转需要 $\Omega(n)$ 的时间。
- R-11.11 按照图 11-13 的风格给出原理图, 展示对 AVL 树进行删除操作过程中子树的高度变化, y 节点的两个子节点以相同高度开始的情况下引发了 trinode 重组。执行删除操作后重新平衡子树的结果是什么?
- R-11.12 重复前面的问题, 考虑其中 y 的子节点从不同的深度开始。
- R-11.13 AVL 树的删除规则中特别要求当表示为 y 的节点的两个子树具有相同深度时, x 子节点应该和 y “对齐”(所以 x 和 y 均为左子节点或右子节点)。为了更好地理解这一要求, 假设选择了错误的 x , 重复练习 R11.11。说明为什么用那种选择恢复 AVL 性能可能会有问题?
- R-11.14 在最初为空的顺序伸展树中执行以下操作后绘制树。
- 插入键 0、2、4、6、8、10、12、14、16、18 (按照这个顺序)。
 - 查找键 1、3、5、7、9、11、13、15、17、19 (按照这个顺序)。
 - 删除键 0、2、4、6、8、10、12、14、16、18 (按此顺序)。
- R-11.15 如果按照键的增加来访问, 伸展树会是什么样子?
- R-11.16 图 11-23a 所示的搜索树是不是一棵 (2, 4) 树? 回答后请给出相应的原因。
- R-11.17 对 (2, 4) 树的节点 w 的另一种分裂是把 w 分成 w' 和 w'' , w' 成了 2-node 而 w'' 成了 3-node。我们会把 k_1 、 k_2 、 k_3 、 k_4 中的哪一个存储在 w 的父节点中? 为什么?
- R-11.18 Amongus 博士声称, 存储一组条目的 (2, 4) 树总是具有相同的结构, 不管在其中插入的条目的顺序如何。请证明他的观点是错的。
- R-11.19 绘制 4 种不同的对应于相同 (2, 4) 树的红黑树。
- R-11.20 假设有一组键 $K = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15\}$ 。
- 用最少的节点绘制 (2, 4) 树, 将 K 中的键作为 (2, 4) 树的键。
 - 用最多的节点绘制 (2, 4) 树, 将 K 中的键作为 (2, 4) 树的键。
- R-11.21 假设有一组键 $(5, 16, 22, 45, 2, 10, 18, 30, 50, 12, 1)$, 绘制插入这些键的项 (按给定顺序) 的结果。
- 最初为空的 (2, 4) 树。
 - 最初为空的红黑树。
- R-11.22 根据下面有关红黑树陈述, 证明每一个为真的语句。对于为假的语句, 请举出反例。
- 红黑树的子树就是一棵红黑树。
 - 没有兄弟节点的节点是红色的。
 - 与给定红黑树相关联的 (2, 4) 树是唯一的。
 - 与给定 (2, 4) 树相关联的红黑树是唯一的。
- R-11.23 在一棵二叉搜索树 T 中, 无论 T 是 AVL 树、伸展树还是红黑树, 中序遍历得到的条目都是相同的输出结果。
- R-11.24 考虑一棵存储 100 000 个条目的树 T , 下面列举的选项哪个有最坏情况的高度?
- T 是二叉搜索树。

- 2) T 是 AVL 树。
- 3) T 是伸展树。
- 4) T 是 $(2, 4)$ 树。
- 5) T 是红黑树。

R-11.25 画一棵是红黑树但不是 AVL 树的例子。

R-11.26 假设 T 是一棵红黑树, 设 p 为该树经过标准搜索树删除算法被删除节点的父节点。试证明: 如果 p 没有子节点, 则删除的节点为红色叶子。

R-11.27 假设 T 是一棵红黑树, 设 p 为该树经过标准搜索树删除算法被删除节点的父节点。试证明: 如果 p 只有一个孩子, 除了一个保留的子结点是红色叶子的情况, 该删除将会在 p 的位置导致黑色不足。

R-11.28 假设 T 是一棵红黑树, 设 p 为该树经过标准搜索树删除算法被删除节点的父节点。试证明: 如果 p 有两个子节点, 则删除的节点是黑色并且有一个红色子节点。

创新

C-11.29 说明如何用 AVL 树或者红黑树对 n 个可比较的元素进行排序, 并且在最坏情况下的时间复杂度为 $O(n \log n)$ 。

C-11.30 能用伸展树对 n 个可比较的元素进行排序, 并且在最坏情况下的时间复杂度达到 $O(n \log n)$ 吗? 为什么?

C-11.31 对 TreeMap 类重复练习 C-10.28。

C-11.32 说明任何 n -node 的二叉树都可以经过 $O(n)$ 次旋转被转换成其他 n -node 的二叉树。

C-11.33 对于一个在二叉搜索树 T 中没有搜索到的键 k , 证明小于 k 的最大键和大于 k 的最小键都位于 k 的搜索路径上。

C-11.34 在 11.1.2 节中, 我们声明了一个二叉搜索树的 `find_range` 方法执行的时间复杂度为 $O(s + h)$, 其中 s 为搜索范围内的元素个数, h 为树的高度。实现在代码段 11-6 开始对开始节点搜索, 并且重复调用 `after` 方法, 直到搜索完整个范围。每次调用 `after` 方法都保证运行时间在 $O(h)$ 以内。这表明了 `find_range` 方法的一个更小的 $O(sh)$ 界限, 因为它包括了 $O(s)$ 的 `after` 调用。证明该实验实现了更大的时间界限 $O(s + h)$ 。

C-11.35 描述如何进行 `remove_range(start, stop)` 操作, 删除所有以二叉搜索树实现的有序映射中落在范围 $(start, stop)$ 之间的键, 并表明该方法运行的时间复杂度为 $O(s + h)$, 其中 s 为删除的元素个数, h 为 T 的高度。

C-11.36 用 AVL 树重新解决上述问题, 实现运行时间复杂度为 $O(s \log n)$ 。为什么原来问题的解决方法对 AVL 树不会直接产生一个 $O(s + \log n)$ 的算法。

C-11.37 假设希望支持一个新的能确定有多少有序映射的键落在一个特定的范围内的计数范围方法 `count_range(start, stop)`。我们很明确地采用我们的 `find_range` 方法实现该操作, 其耗时 $O(s + h)$ 。描述如何修改该搜索树结构使得其用 `count_range` 方法搜索时, 最坏情况下时间复杂度为 $O(h)$ 。

C-11.38 如果在前面的问题中描述的方法前作为 TreeMap 类实现的一部分, 那么为了支持新方法, 必须附加哪些修改 (有可能的话) 作为一个子类 (比如 `AVLTreeMap`)?

C-11.39 为了恢复高度平衡属性, 绘制 AVL 树的原理图, 说明一个单独的移除操作需要 $\Omega(\log n)$ 的从叶子节点到根的 trinode 重组 (或旋转)。

C-11.40 在我们的 AVL 实现中, 每个节点存储其子树的高度, 它是一个任意的大整数。通过存储一

个节点的平衡因子来减少一个 AVL 树的存储空间，其中平衡因子被定义为该节点左子树的高度减去右子树的高度。因此，一个节点的平衡因子总是取 $-1, 0$ 或者 1 。除了在插入或者删除阶段它临时等于 -2 或者 $+2$ 的情况。重新实现存储平衡因子而不是子树高度的 AVLTreeMap 类。

- C-11.41 如果保留一个二叉搜索树最左边节点的引用，那么 `find_min` 操作执行时间会是 $O(1)$ 。描述如何修改其他映射方法从而保留最左边位置的指针。
- C-11.42 如果描述前面问题的方法作为 TreeMap 类实现的一部分，那么必须附加哪些修改（如果可以的话）给一个如 AVLTreeMap 的子类，以精确地保留最左边位置的引用？
- C-11.43 描述一个对二叉搜索树的修改，没有其他方法渐近的不利影响的情况下，通过 `after(p)` 和 `before(p)` 两个方法实现了最坏时间复杂度 $O(1)$ 。
- C-11.44 如果前面问题描述的方法作为 TreeMap 类实现的一部分，为了保持效率，对子类（例如 AVLTreeMap）来说什么样的额外修改（如果有的话）是必要的？
- C-11.45 对于一个标准二叉搜索树，表 11-1 表明了 `delete(p)` 方法需要用 $O(h)$ 的时间复杂度。证明如果对练习 C-11.43 给出一个解决方案，为什么 `delete(p)` 方法运行时间将会是 $O(1)$ ？
- C-11.46 描述一个对二叉搜索树数据结构进行的修改，使其对一个有序映射支持以下两个基于索引的操作，所用时间复杂度为 $O(h)$ ，其中 h 是树的高度。
 - `at_index(i)`: 返回有序映射中索引为 i 的项的位置 p 。
 - `index_of(p)`: 返回有序映射中位置为 p 的项的索引 i 。
- C-11.47 绘制一棵伸展树 T_1 以及产生它的更新的序列，同时绘制一棵红黑树 T_2 ，在同一组设置 10 个条目，使得 T_1 的先序遍历将和 T_2 的先序遍历相同。
- C-11.48 请展示，在 AVL 树中，在插入操作期间暂时成为不平衡的节点可能在从新插入节点到根节点这条路径上不连续。
- C-11.49 请展示，在 AVL 树中，经过标准 `__delitem__map` 操作删除一个节点后至多一个节点暂时失去平衡。
- C-11.50 记 T 和 U 为 $(2, 4)$ 树，分别存储 n 和 m 个条目，使得所有 T 中的条目拥有的键少于 U 中所有条目拥有的键。描述将 T 和 U 合并成单一的树来存储 T 和 U 的所有元素的方法，使得该方法的时间复杂度为 $O(\log n + \log m)$ 。
- C-11.51 用红黑树 T 和 U 重复上述的问题。
- C-11.52 证明命题 11-7。
- C-11.53 当有不同的键时，在红 - 黑树中使用布尔指示器标记节点是“红”还是“黑”并不是很严格。描述一棵现实方案，使得无须添加任何额外的空间就能将一棵准二叉搜索树变成红黑树。
- C-11.54 记 T 是一个有 n 个条目的红黑树， k 为 T 中一个条目的键。展示如何根据 T 在 $O(\log n)$ 的时间里构建两个红黑树 T' 和 T'' ，使得 T' 包含 T 中的所有小于 k 的， T'' 包含 T 中所有大于 k 的键。这个操作会破坏 T 。
- C-11.55 展示任何一个 AVL 树 T 的节点通过标记成红和黑都能成为一个红黑树。
- C-11.56 标准伸展步骤需要两步：首先向下延伸找到待扩展节点 x 。然后向上延伸扩展 x 。描述一个在向下的延伸中伸展并搜索 x 的方法。每个子步骤都需要你考虑接在下降到 x 的路径中接下来的两个节点，并且可能在最后使用 zig 子步骤。描述如何进行 zig-zig, zig-zag 和 zig 步骤。

- C-11.57 考虑一个伸展树的变形，叫作半伸展树。只要到达伸展树 $d/2$ 的深度，将停止伸展深度为 d 的节点。对半伸展树进行摊销分析。
- C-11.58 试述 n -node 伸展树 T 的一系列的访问，其中 n 为奇数。这将导致 T 由一个单链节点组成，使得到 T 的路径中节点交替出现在左子节点和右子节点之间。
- C-11.59 作为一个位置结构，TreeMap 的实现有一点瑕疵。例如一个与位置 p 关联的键 - 值对 (k, v) ，只要其条目保存在映射中，就应该保留其有效性。特别是，该位置不能受到在集合中调用插入或者删除其他项的影响。但是我们的算法在删除一个二叉搜索树时不能提供这样一个保证。因为所定的规则是在删除一个有两个子节点的键时用其前面的键代替它。给出一系列明确的 Python 命令，演示这样的瑕疵。
- C-11.60 如何改变 TreeMap 从而避免上述问题中提到的缺点？

项目

- P-11.61 使用各种不同序列的操作研究比较 AVL 树、伸展树和红黑树的实现速度。
- P-11.62 重复上述练习，包括跳跃表（见练习 P-10.53）的实现。
- P-11.63 使用一棵 $(2, 4)$ 树（见 10.1.1 节）实现 ADT 映射。
- P-11.64 重复上述练习，用到有序 ADT 映射的所有方法（见 10.3 节）。
- P-11.65 重复练习 P-11.63 提供二叉搜索树（11.1.1 节）的位置支持，用到 first()、last()、before(p)、after(p) 和 find_position(kb) 方法。理论上每个条目都有不同的位置，即使许多条目可能存储在一棵树的同一个节点上。
- P-11.66 编写一个 Python 类，能将要给红黑树转换成相应的 $(2, 4)$ 树，同时也能将一个 $(2, 4)$ 树转换成其对应的红黑树。
- P-11.67 在 10.5.3 节描述多集合和多重映射时，我们描述了一个一般的方法来改变传统的映射，即通过在二级容器中存储所有的副本。给出一个可选择的使用二叉搜索树的多重映射的实施方案，使得映射中每个条目存储在树的不同节点中。由于存有副本，因此重新定义搜索树的属性，使得位置 p 的左子树所有条目的键小于等于 k ，位置 p 的右子树的所有条目的键大于等于 k 。使用在代码段 10-17 中给出的公共接口。
- P-11.68 像练习 C-11.56 描述的一样使用从上到下扩展的方法实现伸展树。在本章节进行广泛的实验研究，比较该方法与标准自底向上伸展的性能差别。
- P-11.69 可合并堆 ADT 是优先队列 ADT 的一个扩展，包括的操作有 add(k, v)、min()、remove_min() 和 merge(h)。merge(h) 操作是对一个可合并堆 h 集合中当前元素进行的，将所有条目合并到该元素直到 h 为空。描述一个可合并堆 ADT 的具体实现，该 ADT 的所有操作时间复杂度都在 $O(\log n)$ 以内实现， n 表示合并操作生成的堆的大小。
- P-11.70 编写执行一个简单的 n 体模拟，称为程序“跳妖精灵。”这个模拟包含了 n 个精灵，编号从 1 到 n 。它为每个精灵 i 保留了一个黄金价 g_i ，开始每个精灵价值一百万，即对于 $i = 1, 2, \dots, n$ ， $g_i = 1\,000\,000$ 。另外，该模拟器为每个精灵 i 在水平方向上保留一个空间，代表了一个双精度浮点型数 x_i 。模拟器的每次迭代都使精灵有序。在每次迭代过程中产生一个精灵并且通过以下公式为 i 计算一个水平的空间：

$$x_i = x_i + rg_i$$

r 为 -1 到 1 之间的随机浮点数。然后精灵 i 获取离它最近精灵的一半黄金，并且加到自己的黄金价值 g_i 中。请编写一个程序，通过给出精灵个数 n 来实现这一系列的精灵。你必须使用一个本章中的有序映射数据结构来维持水平位置的集合。

扩展阅读

本章中讨论的许多数据结构在 Knuth 的 *Sorting and Searching*^[65] 书中广泛涉及，并且被 Mehlhorn 在文献 [76] 中用到。AVL 树是由 Adel'son-Vel'skii 和 Landis^[2] 于 1962 年发明的平衡搜索树。二叉搜索树、AVL 树和哈希都在 Knuth 的 *Sorting and Searching*^[65] 书中有讲述。二叉搜索树的平均高度分析来自 Aho、Hopcroft 和 Ullman^[6]，以及 Cormen、Leiserson、Rivest 和 Stein 的书^[29]。Gonnet andaeza-Yates^[44] 的手稿保留了许多映射实现的理论和实验的比较。Aho、Hopcroft 和 Ullman^[5] 讨论了 (2,3) 树，这种树类似 (2, 4) 树。红黑树是由 Bayer^[10] 定义的。Guibas 和 Sedgewick^[48] 的论文展示了红黑树的各种有趣属性。有兴趣想了解更多有关不同平衡树数据结构的读者，可以阅读 Mehlhorn^[76] 和 Tarjan^[95] 的书，本章参见 Mehlhorn 和 Tsakalidis^[78]。Knuth^[65] 是优秀的附加读物，包含了早期平衡树的研究方法。伸展树是由 Sleator 和 Tarjan^[89]（也可参见文献 [95]）发明的。

排序与选择

12.1 为什么要学习排序算法

本章的重点是针对对象集进行排序的算法。我们要对一个集合的元素进行重新排列，以使它们按照从小到大的顺序进行排列（或以此顺序生成一个新的副本）。我们假设存在一个这样的一致次序，就如同我们在学习优先级队列时所做的（参见 9.4 节）。在 Python 中，对象的自然顺序一般使用 `<` 操作符定义，该运算符具有以下性质：

- 非自反性： $k \not< k$ 。
- 可传递性：若 $k_1 < k_2$ 且 $k_2 < k_3$ ，则 $k_1 < k_3$ 。

可传递性是很重要的。它使我们在不花费时间执行比较的情况下，能够直接推断出某些比较的结果，从而得到一个更高效的算法。

排序是已被很多学者充分研究过的有关计算的最重要的问题之一。数据集合通常按照排好序的顺序存储以便进行高效搜索，举个例子，在已有序的数据集合上可以使用二分查找算法（参见 4.1.3 节）来检索。许多解决不同问题的高级算法都依赖于排序。

Python 对数据排序提供了内置支持，其中包括重新对列表内容进行排序的 `list` 类的 `sort` 方法，还有以排好的顺序生成一个包含任意元素集合的内置的 `sorted` 函数。这些内置函数使用了一些高级算法（其中的一些我们将在本章描述），并且是高度优化的。由于很少有需要从头开始实现排序的特殊情况出现，因此编程人员往往会调用内置排序函数。

这就表示，对排序算法有深刻的理解是十分重要的。当务之急，在调用这些内置函数的时候，最好弄清楚预期的效率是多少以及它是如何依赖于元素的初始顺序或者排序对象的类型的。一般而言，这些引领排序算法发展进步的思想和方法使得计算机应用其他领域的算法也得到了发展。

我们在本书中已经介绍了一些排序算法：

- 插入排序（参见 5.5.2 节、7.5 节和 9.4.1 节）
- 选择排序（参见 9.4.1 节）
- 冒泡排序（参见练习 C-7.38）
- 堆排序（参见 9.4.2 节）

在本章中，我们展示了四种其他的排序算法：归并排序、快速排序、桶排序和基数排序，之后我们将在 12.5 节中讨论这些排序算法的优缺点。

12.2 归并排序

12.2.1 分治法

我们在本章中先描述前两个算法——归并排序和快速排序，它们在分治法的算法设计模式当中使用了递归的方法。我们已经知道，递归可以十分简练地描述一个算法（参见第 4 章）。分治法设计模式包含以下三个步骤：

1) 分解：如果输入值的规格小于确定的阈值（比如一个或者两个元素），我们就通过使用直截了当的方法来解决这些问题并返回所获得的答案。否则，我们把输入值分解为两个或者更多的互斥子集。

2) 解决子问题：递归地解决这些与子集相关的子问题。

3) 合并：整理这些子问题的解，然后把它们合并成一个整体用以解决最开始的问题。

使用分治法进行排序

我们首先在一个很高的层次上描述归并算法，而不是去关注数据是基于数组（Python）的表还是链表，之后，我们将给出对于每一种数据的具体实现。我们使用分治法的三个步骤来对一个有 n 个元素的序列 S 进行排序，归并排序的过程如下：

1) 分解：若 S 只有 0 个或 1 个元素，直接返回 S ；此时它已经完成排序了。否则（若 S 有至少 2 个元素），从 S 中移除所有的元素，然后将它们放在 S_1 、 S_2 两个序列中，每一个序列包含 S 中一半的元素。这就是说， S_1 包含 S 前一半的元素， S_2 包含 S 后一半的元素。

2) 解决子问题：递归地对 S_1 和 S_2 进行排序。

3) 合并：把这些分别在 S_1 和 S_2 中排好序的元素拿出并按照顺序合并到 S 序列中。

关于分解的步骤，我们用 $\lfloor x \rfloor$ 符号来表示取 x 的底（floor），即有最大的整数 k 使得 $k \leq x$ 。类似地，我们用 $\lceil x \rceil$ 表示取 x 的顶（ceiling），即有最小的整数 m 使得 $x \leq m$ 。

可以用一个二叉树 T 来形象化一个归并排序算法的执行过程，称这个二叉树为归并排序树。 T 的每一个节点表示归并排序算法的一个递归调用（或引用）。我们将 T 中的每个节点 v ，通过调用和序列 S 关联起来。节点 v 的子节点通过递归调用相关联，该递归调用可以处理 S 的子序列 S_1 和 S_2 。 T 的外部节点是与 S 中的单个元素相关联的，与无递归调用的算法实例一致。

图 12-1 通过展示对归并排序树每个节点处理得到的输入输出序列，总结了归并排序算法的执行过程。归并排序树的逐步演变在图 12-2 ~ 图 12-4 中展示。

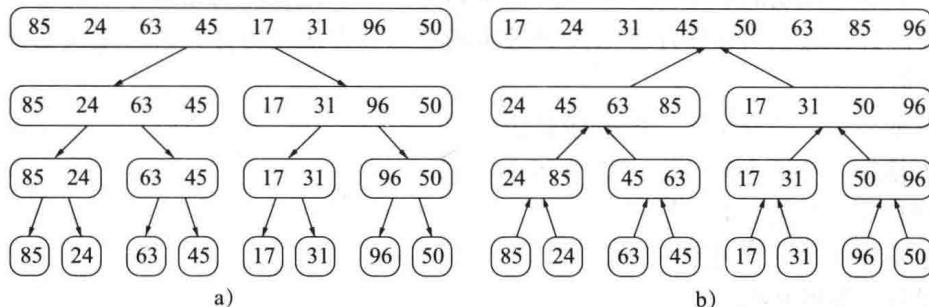


图 12-1 8 元素序列的归并排序算法执行过程的归并排序树 T ：a) 对 T 的每个节点处理得到的输入序列；b) T 的每个节点生成的输出序列

归并排序树算法的可视化，可以帮助我们分析归并排序算法的运行时间。特别地，由于输入序列的大小大约是归并排序中每个递归调用的一半，因此归并排序树的高度大约是 $\log n$ （如果 \log 的底被省略，则以 2 为底）。

命题 12-1： 在大小为 n 的序列上执行归并算法，与其相关联的归并排序树的高度为 $\lceil \log n \rceil$ 。

我们把命题 12-1 的证明留作一个简单的练习 R-12.1。我们将使用这一命题来分析归并

排序算法的运行时间。

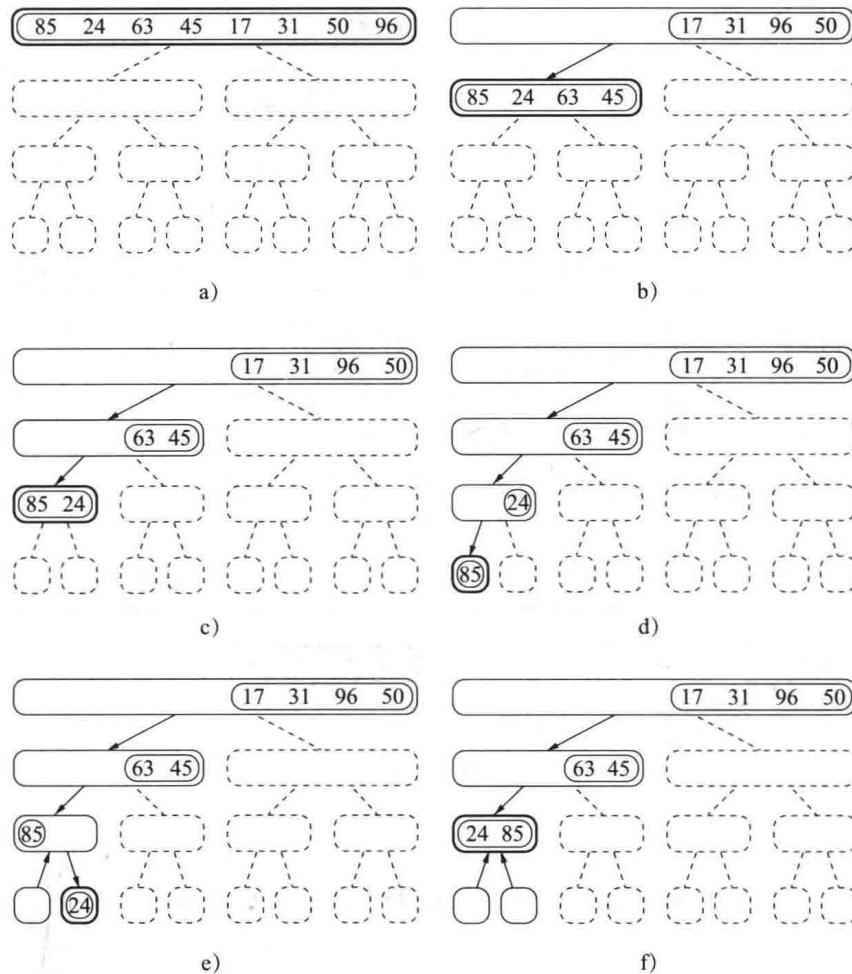


图 12-2 一个可视化的归并排序执行过程。树的每一个节点表示一个归并排序的递归调用。虚线所画的节点表示该节点的调用仍未形成。粗线所画的节点表示当前调用的节点。用细线画出的空节点表示该节点已经完成调用。剩下的节点（细线所画的但是非空的）表示该调用正在等待子节点调用的返回值（下接图 12-3）

结合已经给出的关于归并排序的概述，以及其工作方式的说明，让我们更详细地思考分治法中的每一个步骤。把一个长度为 n 的序列在其位置为 $\lceil n/2 \rceil$ 的元素处进行分解，然后可以通过把较小的序列作为参数开始递归调用。比较复杂的步骤则是将两个已排序的子序列合并成一个单独的序列。因此，在我们进行关于归并排序的分析之前，需要知道更多有关它是如何完成的内容。

12.2.2 基于数组的归并排序的实现

我们以一个被表示为 Python 列表（基于数组）的序列开始。merge 函数（见代码段 12-1）负责将之前提到的两个已排序的序列 S_1 和 S_2 合并，并将输出复制到序列 S 中的子任务。我们在每次进入 while 循环时复制一个元素，有条件地决定下一个元素将会取自 S_1 或 S_2 中的哪一个。分治法的归并排序算法已经给出，参见代码段 12-2。

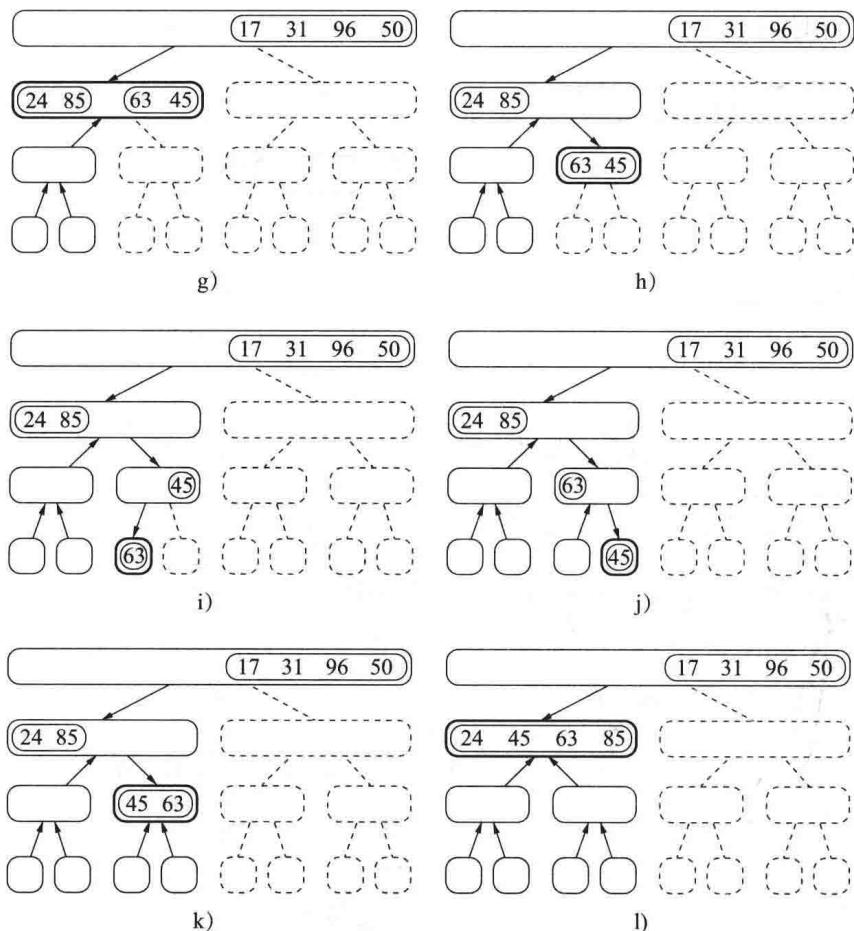


图 12-3 可可视化的归并排序执行过程（结合图 12-2 和图 12-4）

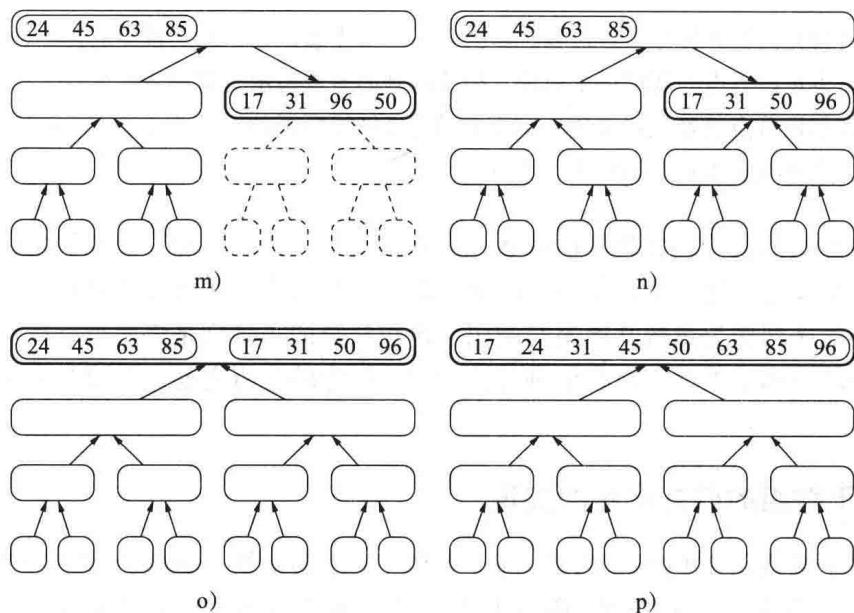


图 12-4 可可视化的归并排序执行过程（上接图 12-3）。许多在图 m 和 n 之间的调用被省略了。在步骤 p 中，请注意这两个部分的合并过程

代码段 12-1 Python 中基于数组的 list 类的合并操作的执行过程

```

1 def merge(S1, S2, S):
2     """ Merge two sorted Python lists S1 and S2 into properly sized list S. """
3     i = j = 0
4     while i + j < len(S):
5         if j == len(S2) or (i < len(S1) and S1[i] < S2[j]):
6             S[i+j] = S1[i]           # copy ith element of S1 as next item of S
7             i += 1
8         else:
9             S[i+j] = S2[j]           # copy jth element of S2 as next item of S
10            j += 1

```

代码段 12-2 Python 中基于数组的 list 类的递归归并排序算法的执行过程（使用了代码段 12-1 中定义的 merge 函数）

```

1 def merge_sort(S):
2     """ Sort the elements of Python list S using the merge-sort algorithm. """
3     n = len(S)
4     if n < 2:
5         return                      # list is already sorted
6     # divide
7     mid = n // 2
8     S1 = S[0:mid]                 # copy of first half
9     S2 = S[mid:n]                 # copy of second half
10    # conquer (with recursion)
11    merge_sort(S1)               # sort copy of first half
12    merge_sort(S2)               # sort copy of second half
13    # merge results
14    merge(S1, S2, S)             # merge sorted halves back into S

```

下面我们来说明图 12-5 中合并过程的一个步骤。在整个过程中，索引 i 表示 S_1 中已经被复制到 S 中的元素个数，同时，索引 j 表示 S_2 中已经被复制到 S 中的元素个数。假设 S_1 和 S_2 都至少有 1 个未复制元素，我们考虑复制两个元素中较小的那个元素。因为 $i+j$ 个对象之前已经复制过了，所以下一个元素会被放置到 $S[i+j]$ （例如，当 $i+j$ 为 0，则下一元素就被复制到 $S[0]$ ）。如果我们达到了某一个序列的最后，就必须从另一序列开始复制下一个元素。

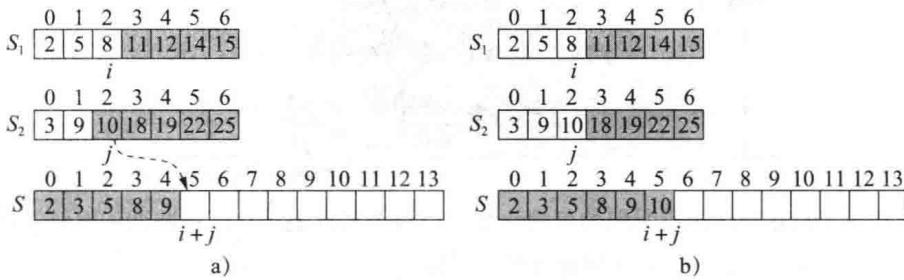


图 12-5 两个已排序数组的合并步骤 ($S_2[j] < S_1[i]$)。我们展示了数组在复制前 (a) 与复制后 (b) 的情况

12.2.3 归并排序的运行时间

我们来分析 merge 算法的运行时间。令 n_1 和 n_2 分别为 S_1 和 S_2 的元素数。很明显，在每个 while 循环中执行的操作消耗 $O(1)$ 的时间。需要注意的是，在每一次迭代循环的过程

中，元素始终是从 S_1 或者 S_2 复制到 S 中的（并且认为这个元素没有做更进一步的复制）。因此，循环的迭代次数是 $n_1 + n_2$ 。也就是说，merge 算法的运行时间是 $O(n_1 + n_2)$ 。

分析了用于合并子问题的 merge 算法的运行时间，对于一个含有 n 个元素的输入序列，我们可以分析其整个归并排序算法的运行时间。为简单起见，我们只考虑 n 是 2 的乘方的情况。当 n 不是 2 的乘方时分析结果依旧成立，我们把它留作练习 R-12.3。

在评估归并排序的递归时，我们依赖于 4.2 节中介绍的分析技术。我们对每一次递归调用的时间消耗进行计算，但是排除等待成功的递归调用终止所花费的时间。至于 merge_sort 函数，我们计算把一个序列分为两个子序列，以及调用 merge 函数来合并这两个已排序的序列所耗费的时间，但排除了两个对 merge_sort 函数的递归调用。

一棵归并排序树 T ，如同图 12-2 ~ 图 12-4 所描绘的，可以指引我们的分析。考虑一个已经关联了归并排序树 T 的节点 v 的递归调用。在节点 v 处分解的步骤是直截了当的；基于切片的使用来创造两个 list 的二分副本，这一步运行的时间与 v 所在序列的大小成比例。我们已经看出，合并步骤在已合并序列的大小中，同样也花费线性的时间。如果我们让 i 表示节点 v 的深度，则在节点 v 处的时间花费为 $O(n/2^i)$ ，原因是关联了 v 的递归调用所处理的序列的长度为 $n/2^i$ 。

更全局地看这棵树 T ，如图 12-6，我们看到，基于“在节点处的时间花费”的定义，归并排序的运行时间等于在树 T 的节点处时间花费的总和。注意， T 在深度为 i 处，显然有个节点。这一简单的现象得出很重要的结论，它意味着在树 T 的深度为 i 处的所有节点的全部时间花费为 $O(2^i \cdot n/2^i)$ ，即 $O(n)$ 。由命题 12-1 可知，树 T 的高为 $\lceil \log n \rceil$ 。也就是说，因为在树 T 的每个 $\lceil \log n \rceil + 1$ 处，时间花费均为 $O(n)$ ，所以有如下结论。

命题 12-2：假设一个大小为 n 的序列 S ，其两个元素可以在 $O(1)$ 的时间内完成比较，那么归并排序算法对 S 进行排序消耗的时间为 $O(n \log n)$ 。

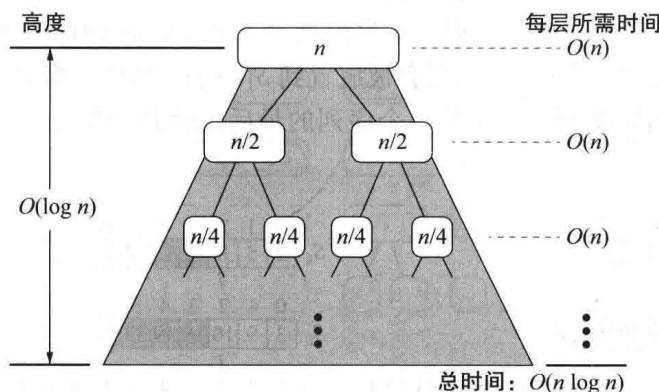


图 12-6 归并排序运行时间的可视化分析。每个节点表示自己递归调用时所花费的时间，并与其子问题规模一同标注

12.2.4 归并排序与递归方程 *

有另一种证明归并排序算法运行时间为 $O(n \log n)$ （由命题 12-2 得出的）的方法。换句话说，我们可以更直接地处理归并排序算法的递归性。在本节中，我们提出一种关于归并排序运行时间的分析，介绍递归方程（也称为递归关系）的数学概念。

我们用函数 $t(n)$ 来表示一个规模为 n 的输入序列的归并排序在最坏情况下的运行时间。

因为归并排序是递归的，所以我们可以用一个方程来描述函数 $t(n)$ ，在该方程中，函数 $t(n)$ 可以根据其自身递归地表达。为了简化 $t(n)$ 的描述，我们只考虑 n 为 2 的乘方的情况（这个问题的渐近特性在一般情况下依旧成立，我们把这个留作练习）。在这种情况下，我们可以把 $t(n)$ 的定义详细化：

$$t(n) = \begin{cases} b & n \leq 1 \\ 2t(n/2) + cn & \text{其他} \end{cases}$$

一个如上所示的表达式被称作递归方程，是因为这个函数同时出现了等号的左支和右支。尽管这样的描述是正确且精确的，然而我们希望得出的是一个关于 $t(n)$ 且不包含 $t(n)$ 自己的大 O 类型的描述。这就是说，我们需要一个关于 $t(n)$ 的封闭性描述。

我们在假设 n 比较大的情况下通过递归方程的定义获得了一个封闭的解决方案。例如在上式的再次应用后，我们可以写出一个新的递归式如下：

$$t(n) = 2(2t(n/2^2) + (cn/2)) + cn = 2^2 t(n/2^2) + 2(cn/2) + cn = 2^2 t(n/2^2) + 2cn$$

如果我们再次应用这个方程，会得到 $t(n) = 2^3 t(n/2^3) + 3cn$ 。从这个角度，我们可以看出一个新模式，即在应用这个表达式 i 次之后可以得到：

$$t(n) = 2^i t(n/2^i) + icn$$

之后剩下的问题就是决定何时终止这个过程。为了知道何时停止这个过程，再次调用我们设置的开关，即当 $2^i = n$ 时将会出现的 $t(n) = b$ ($n \leq 1$) 这个情况。换句话说，这种情况将在 $i = \log n$ 时出现。使用这个代换之后，会得到：

$$t(n) = 2^{\log n} t(n/2^{\log n}) + (\log n)cn = nt(1) + cn\log n = nb + cn\log n$$

也就是说，我们得到了 $t(n)$ 就是 $O(n \log n)$ 这个事实的一个可供替代的证明。

12.2.5 归并排序的可选实现

排序链表

归并排序算法由于其容器类型而很容易适用于使用一个基本队列的任何形式。在代码段 12-3 中，我们基于 7.1.2 节提到的 Linked Queue 类的使用给出了上述内容的实现。命题 12-2 中归并排序的界 $O(n \log n)$ 同样可以应用于这种实现，因为在用一个链表实现时，每个基本操作均消耗 $O(1)$ 的时间。我们在图 12-7 中展示了这种 merge 算法的执行过程。

代码段 12-3 使用基本队列的归并排序实现

```

1 def merge(S1, S2, S):
2     """Merge two sorted queue instances S1 and S2 into empty queue S."""
3     while not S1.is_empty() and not S2.is_empty():
4         if S1.first() < S2.first():
5             S.enqueue(S1.dequeue())
6         else:
7             S.enqueue(S2.dequeue())
8         while not S1.is_empty():          # move remaining elements of S1 to S
9             S.enqueue(S1.dequeue())
10        while not S2.is_empty():         # move remaining elements of S2 to S
11            S.enqueue(S2.dequeue())
12
13 def merge_sort(S):
14     """Sort the elements of queue S using the merge-sort algorithm."""
15     n = len(S)

```

```

16 if n < 2:
17     return # list is already sorted
18     # divide
19     S1 = LinkedQueue( ) # or any other queue implementation
20     S2 = LinkedQueue()
21     while len(S1) < n // 2: # move the first n//2 elements to S1
22         S1.enqueue(S.dequeue())
23     while not S.is_empty(): # move the rest to S2
24         S2.enqueue(S.dequeue())
25     # conquer (with recursion)
26     merge_sort(S1) # sort first half
27     merge_sort(S2) # sort second half
28     # merge results
29     merge(S1, S2, S) # merge sorted halves back into S

```

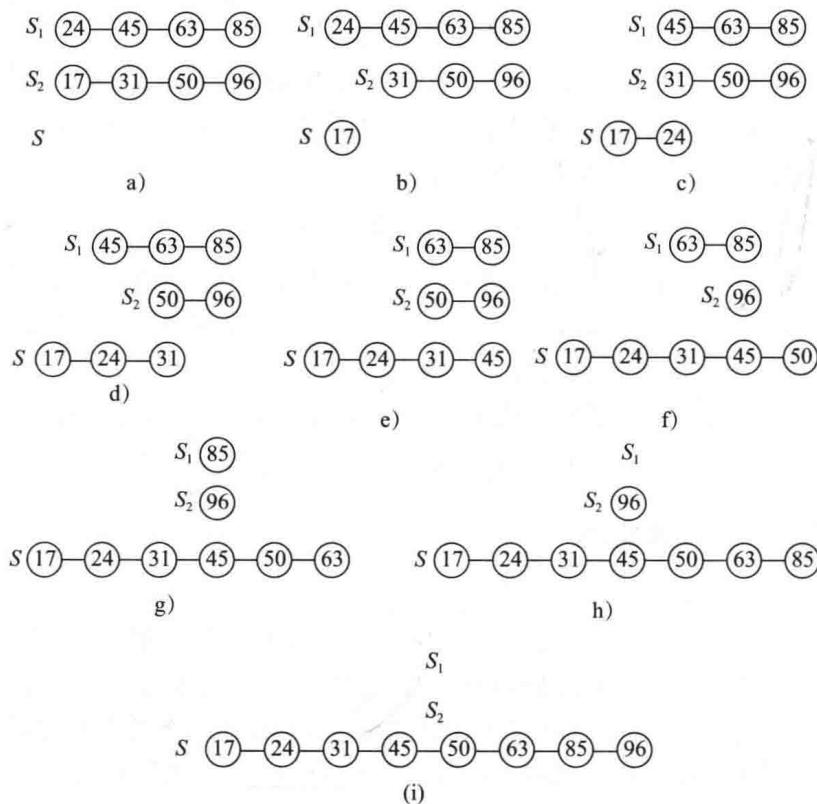


图 12-7 在代码段 12-3 中使用队列实现的归并排序的执行示例

自底向上的（非递归的）归并排序

这是一个基于数组的非递归版本的归并排序，运行时间为 $O(n \log n)$ 。在实践中，它会比递归的归并排序略快一些，因为它回避了递归调用的额外开销并且在每一级都有临时存储器。这种算法的主要思想是执行自底向上的归并排序，即对整个归并排序树自底向上逐层执行合并。给出元素的一个输入数组，我们将每个连续的元素对合并成有序的，以长度为 2 开始执行。然后再合并至长度为 4、长度为 8 等，以此类推，直到整个数组已经排序完毕。为了保持合理的空间使用情况，我们使用一个二维数组来存储这些合并的执行过程（在每次迭代完成后交换输入输出数组）。我们在代码段 12-4 中给出一个 Python 语言的实现。一个近似的自底向上的方法可以被用于排序链表（见练习 C-12.29）。

代码段 12-4 一个非递归的归并排序算法的实现

```

1 def merge(src, result, start, inc):
2     """ Merge src[start:start+inc] and src[start+inc:start+2*inc] into result. """
3     end1 = start+inc           # boundary for run 1
4     end2 = min(start+2*inc, len(src))    # boundary for run 2
5     x, y, z = start, start+inc, start    # index into run 1, run 2, result
6     while x < end1 and y < end2:
7         if src[x] < src[y]:
8             result[z] = src[x]; x += 1      # copy from run 1 and increment x
9         else:
10            result[z] = src[y]; y += 1      # copy from run 2 and increment y
11            z += 1                         # increment z to reflect new result
12    if x < end1:
13        result[z:end2] = src[x:end1]    # copy remainder of run 1 to output
14    elif y < end2:
15        result[z:end2] = src[y:end2]    # copy remainder of run 2 to output
16
17 def merge_sort(S):
18     """ Sort the elements of Python list S using the merge-sort algorithm. """
19     n = len(S)
20     logn = math.ceil(math.log(n, 2))
21     src, dest = S, [None] * n          # make temporary storage for dest
22     for i in (2**k for k in range(logn)):  # pass i creates all runs of length 2i
23         for j in range(0, n, 2*i):       # each pass merges two length i runs
24             merge(src, dest, j, i)       # merge(src, dest, j, i)
25             src, dest = dest, src        # reverse roles of lists
26     if S is not src:                  # additional copy to get results to S
27         S[0:n] = src[0:n]

```

12.3 快速排序

接下来，我们将讨论快速排序。如同归并排序，这个算法同样是基于分治法的典范，但是它在使用这项技术时运用了相反的方式，即把所有的复杂操作在递归调用之前做完。

快速排序的高阶描述

快速排序算法使用一个简单的递归方法将序列 S 排序。主要思想是应用分治法把序列 S 分解为子序列，递归地排序每个子序列，然后通过简单串联的方式合并这些已排序的子序列。特别地，快速排序算法由以下 3 个步骤组成（见图 12-8）。

1) 分解：如果 S 有至少 2 个元素（如果 S 只有 1 个或 0 个元素，什么都不用做），从 S 中选择一个特定的元素 x ，称之为基准值。一般情况下，选择 S 中最后一个元素作为基准值 x 。从 S 中移除所有的元素，并把它们放在 3 个序列中：

- L 存储 S 中小于 x 的元素
- E 存储 S 中等于 x 的元素
- G 存储 S 中大于 x 的元素

当然，如果 S 中的元素是互异的，那么 E 将只含有一个元素——基准值自己。

2) 解决子问题：递归地排序序列 L 和 G 。

3) 合并：把 S 中的元素按照先插入 L 中的元素、然后插入 E 中的元素、最后插入 G 中

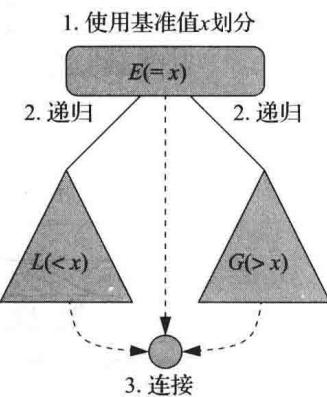


图 12-8 快速排序算法的原理图

的元素的顺序放回。

和归并排序一样，快速排序的执行也可以用二叉递归树来模拟，称作快速排序树。图 12-9 通过展示对快速排序树的每个节点处理得到的输入输出序列，总结了快速排序算法的执行情况。快速排序树的逐步评估在图 12-10 ~ 图 12-12 中展示。

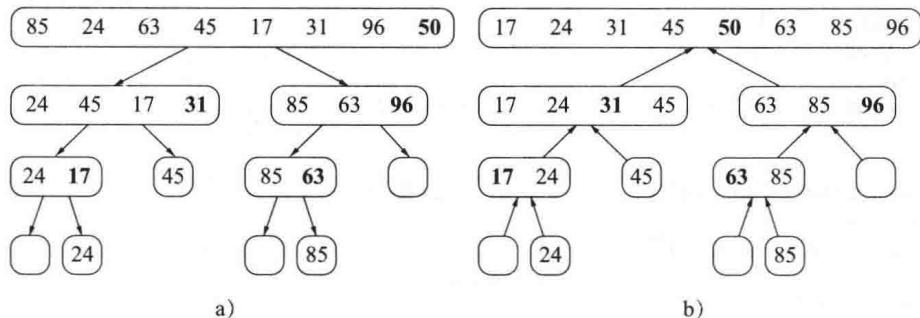


图 12-9 对 8 元素序列执行快速排序算法产生的快速排序树 T ：a) 对 T 的每个节点处理得到的输入序列；b) 对 T 的每个节点生成的输出序列。每一级递归所使用的基准值用粗体标出

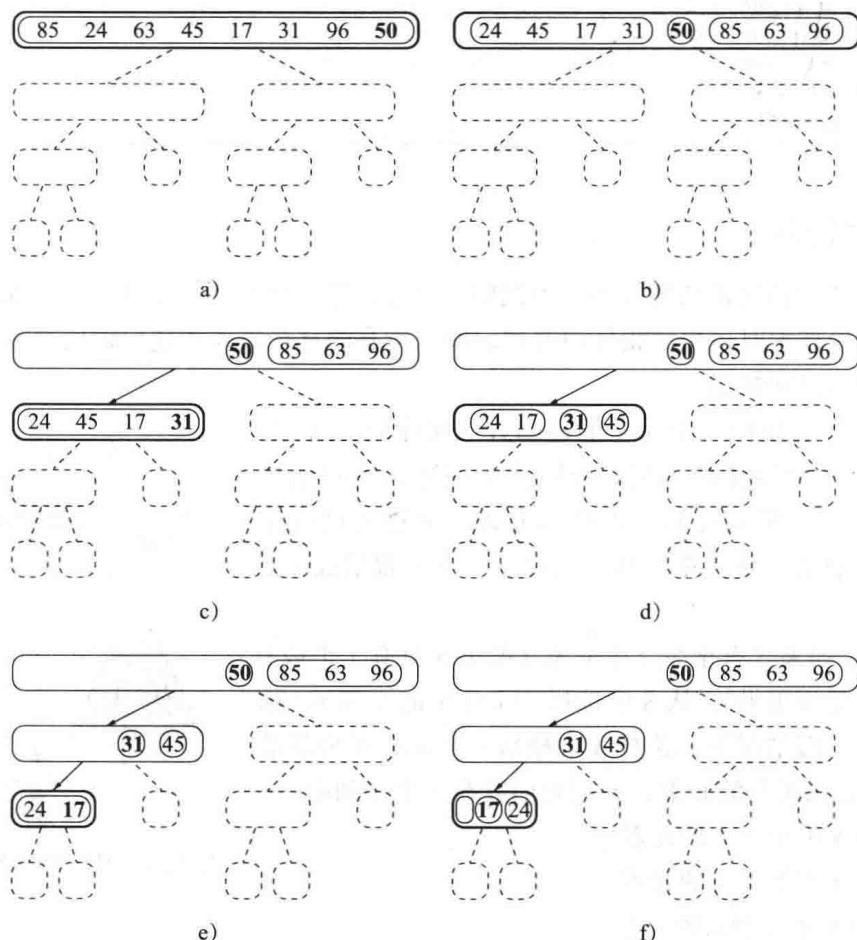


图 12-10 快速排序执行过程的模拟。树的每一个节点表示一个递归调用。虚线画出的节点表示还没访问，粗线画出的节点表示正在调用，细线画出的节点表示已经访问过，剩下的节点表示延迟访问。注意在 b、d 和 f 中执行的分解步骤（接图 12-11）

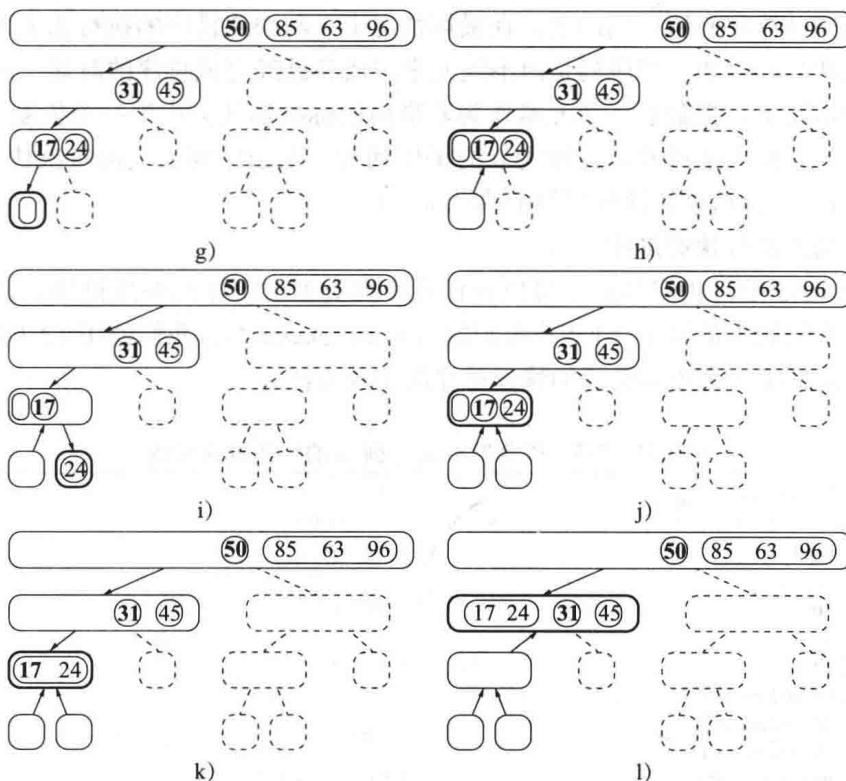


图 12-11 快速排序执行过程的模拟。注意在 k 上执行的串联步骤（接图 12-12）

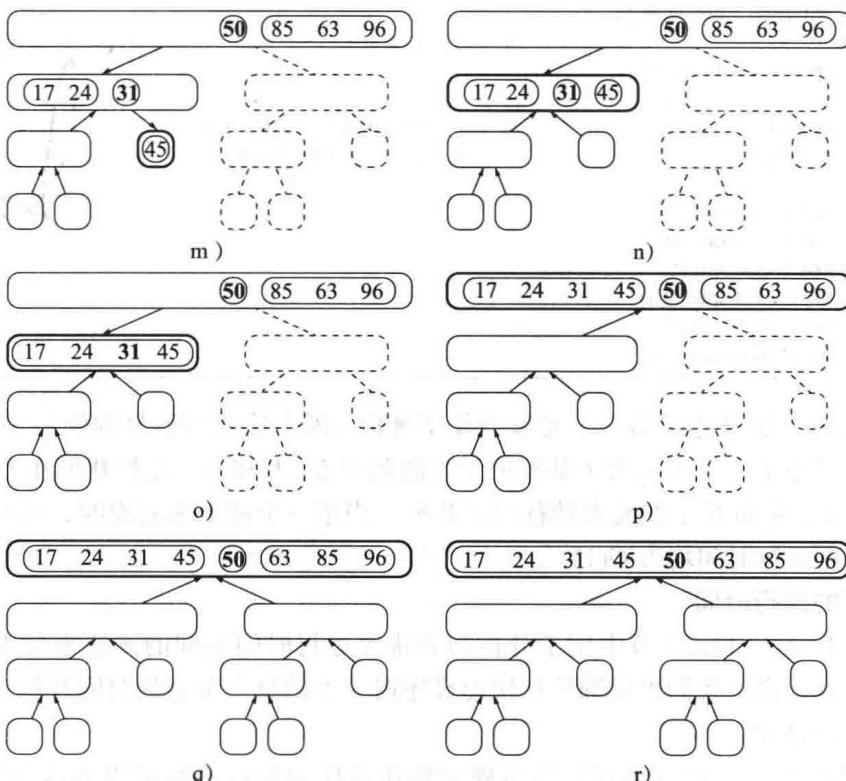


图 12-12 快速排序执行过程的模拟。在 p 和 q 之间的一些调用被省略了。注意在 o 和 r 上执行的串联步骤（上接图 12-11）

但是，与归并排序有所区别的是，在最坏情况下，与快速排序的执行相关联的快速排序树的高度是线性的。例如，当序列是由不同元素组成且已经完成排序的时候，这种情况就会发生。在这种情况下，把最后一个元素作为基准值的标准选法会产生一个长度为 $n - 1$ 的子序列 L 、长度为 1 的子序列 E 和长度为 0 的子序列 G 。在子序列 L 上的每次快速排序调用， L 的长度都减 1。所以这棵快速排序树的高为 $n - 1$ 。

在一般序列上执行快速排序

在代码段 12-5 中，我们给出了可以在任意序列类型上工作的快速排序算法作为队列的实现。这个特别的版本依赖于 7.1.2 节提到的 Linked Queue 类，我们使用 12.3.2 节提到的基于数组的序列提供了一个更为精简的快速排序的实现方法。

代码段 12-5 作为队列的序列 S 的快速排序实现

```

1 def quick_sort(S):
2     """Sort the elements of queue S using the quick-sort algorithm."""
3     n = len(S)
4     if n < 2:
5         return                               # list is already sorted
6     # divide
7     p = S.first()                         # using first as arbitrary pivot
8     L = LinkedQueue()
9     E = LinkedQueue()
10    G = LinkedQueue()
11    while not S.is_empty():               # divide S into L, E, and G
12        if S.first() < p:
13            L.enqueue(S.dequeue())
14        elif p < S.first():
15            G.enqueue(S.dequeue())
16        else:                           # S.first() must equal pivot
17            E.enqueue(S.dequeue())
18    # conquer (with recursion)           # sort elements less than p
19    quick_sort(L)                      # sort elements greater than p
20    quick_sort(G)
21    # concatenate results
22    while not L.is_empty():
23        S.enqueue(L.dequeue())
24    while not E.is_empty():
25        S.enqueue(E.dequeue())
26    while not G.is_empty():
27        S.enqueue(G.dequeue())

```

我们的实现方法是选择第一个元素当作基准值（因为这比较容易理解），然后用它将序列 S 分解为分别小于、等于和大于基准值元素的队列 L 、 E 和 G 。之后我们在 L 和 G 表上递归，并把列表 L 、 E 和 G 上的元素转移回 S 队列。当用一个链表来实现时，所有的队列操作在最坏情况下的运行时间为 $O(1)$ 。

快速排序的运行时间

我们可以使用与 12.2.3 节中用于分析归并排序运行时间相同的方法来分析快速排序的运行时间。也就是说，我们可以确认在快速排序树 T 上的每个节点的时间开销，并求出所有节点的运行时间总和。

测试代码段 12-5，可以看到分解步骤和快速排序的最终串联可以在线性时间内实现。因此，在 T 的节点 v 上，时间开销是与 v 的输入规模 $s(v)$ 成比例的，根据与节点 v 相联系的快速排序调用所处理的序列大小来定义。因为子序列 E 至少有一个元素（基准值），所以节

点 v 的子节点的总输入长度最多为 $s(v) - 1$ 。

用 S_i 表示一棵特定的快速排序树在深度为 i 处节点的输入长度总和。很明显，由于树 T 的根 r 与整个序列有联系，所以 $S_0 = n$ 。同样地，由于基准值不会传给 r 的子节点，所以 $S_1 \leq n - 1$ 。一般来说，会有 $s_i < s_{i-1}$ ，这是因为在深度为 i 处的子序列的所有元素均来自不同的深度为 $(i-1)$ 的子序列，另外，至少会有一个深度为 $i-1$ 的元素不会传递到深度 i 处，这是因为它处于集合 E 中（事实上，任何一个深度为 $i-1$ 的节点的元素都不会传递到深度 i 上）。

我们可以由此给出一个形如 $O(n \cdot h)$ 的快速排序执行的整体运行时间范围（其中 h 为该执行的快速排序树 T 的整体高度）。不幸的是，在最坏的情况下，如同我们在 12.3 节看到的，快速排序树的高为 $\Theta(n)$ 。因此，快速排序在最坏情况下运行时间为 $O(n^2)$ 。然而自相矛盾的是，如果我们选择基准值作为序列的最后一个元素，这一最坏情况行为在排序很容易完成的时候就会发生，即在序列已经有序的时候。

正如它的名字一样，我们期望快速排序可以运行得很快，而且事实上它确实很快。快速排序的最好情况发生在序列由不同的元素组成，且子序列 L 与 G 的大小大致相等的时候。在这种情形下，如同归并排序一样，排序树的高度为 $O(\log n)$ ，所以快速排序运行时间为 $O(n \log n)$ ，我们把这个事实的证明留作练习 R-12.10。更重要的是，即使 L 和 G 的分割不是那么完美，我们也还是可以观察到形如 $O(n \log n)$ 的运行时间。例如，如果每个分解步骤都造成一个包含了 $1/4$ 总元素的子序列，那么其他的步骤则包含了剩下 $3/4$ 的元素，因此树的剩余高度为 $O(\log n)$ ，总的执行代价为 $O(n \log n)$ 。

我们将在下一节看到，基准值选择的随机化引入将使得快速排序通常以这种方式表现，即能达到期望的运行时间 $O(n \log n)$ 。

12.3.1 随机快速排序

分析快速排序的一般方法是假设基准值总是能将序列以合理的、平衡的方式分解。尽管我们预先假设了有关输入分布的知识，但这些输入分布通常是不可用的。例如，我们将不得不假设得到一个几乎排好的序列去进行排序是极少见的情况，这在许多应用中很常见。幸运的是，并不需要该假设去匹配我们对于快速排序行为的直觉。

一般来说，我们希望一些方法可以使快速排序的运行时间更接近最好情况的运行时间。当然，这种接近最优运行时间的方法，就是使得基准值近乎平均分输入序列 S 。如果这一结果发生，将导致运行时间趋近于最好的运行时间。这就是说，让基准值尽量接近元素集合的“中间”，会使快速排序的运行时间达到 $O(n \log n)$ 。

随机选择基准值

因为快速排序方法划分步骤的目的是把序列 S 分解得足够平衡，因此我们为算法引入随机化的概念并且选择输入序列的一个随机元素作为基准值。也就是说，为了代替选择 S 的第一个或最后一个元素作为基准值，我们选择 S 中的一个随机元素作为基准值，并且保持算法的其余部分不变。这种变化的快速排序称为随机快速排序。以下命题展示了一个 n 元素序列的随机化快速排序的期望运行时间是 $O(n \log n)$ 。这个期望涵盖了算法造成的所有可能的随机选择，并且独立于算法包含的任何关于可能的输入序列分布的假设。

命题 12-3：一个大小为 n 的序列 S ，其随机化快速排序的期望运行时间为 $O(n \log n)$ 。

证明：我们假设 S 中的两个元素可以在 $O(1)$ 的时间内比较。考虑一个单独的随机化快速排序的递归调用，然后用 n 表示该调用的输入序列大小。如果基准值的选择使得每个子序

列 L 和 G 均有至少 $n/4$ 、至多 $3n/4$ 的长度，我们可以称之为“好”的选择，否则，我们称之为“坏”的选择。

现在，考虑用随机法均匀地选择基准值带来的影响。注意，对于任意给出的随机快速排序算法大小为 n 的调用，将有 $n/2$ 种基准值选择可能是好的选择。因此，任意的调用都是好的可能性为 $1/2$ 。更加值得注意的是，一个好的调用至少将一个大小为 n 的列表分割为两个大小为 $3n/4$ 和 $n/4$ 的列表，同时，一个不好的调用有可能产生一个单独的大小为 $n - 1$ 的调用一样不好。

现在考虑一个随机化快速排序的递归追踪。这个追踪定义了一个二叉树 T ，这样 T 的每个节点相当于在排序部分原始列表的子问题上的一个不同的递归调用。

我们说如果 v 的子问题大小大于 $(3/4)^{i+1}n$ 且最大为 $(3/4)^i n$ ， T 的节点 v 就在尺寸组 i 中。我们来分析一下在尺寸组 i 内节点的所有子问题上工作花费的期望时间。根据期望的线性性质（命题 B-19），在这些子问题上工作的期望时间是所有期望时间的总和。其中的一些节点对应好的调用，而另一些则对应不好的调用。但是值得注意的是，因为一个好的调用出现的可能性为 $1/2$ ，所以在得到一个好的调用之前，那些我们不得不做的连续调用的期望数量是 2。另外值得注意的是，一旦我们对在尺寸组 i 中的一个节点产生了好的调用，那它的孩子将会出现在高于 i 的尺寸组中。因此，对于来自输入列表的任意元素 x ，在其子问题中，包含 x 的尺寸组 i 中的期望节点的数量为 2。换句话说，所有尺寸组 i 的子问题的期望总规模为 $2n$ 。由于我们为一些子问题执行的非递归的工作是与其规模成比例的，这意味着处理尺寸组 i 中节点子问题的总体期望时间是 $O(n)$ 。

因为重复地乘以 $3/4$ 与重复地除以 $4/3$ 是等价的，所以这些尺寸组的数量为 $\log_{4/3} n$ 。这就是说，这些尺寸组的数量是 $O(\log n)$ 。因此，随机化快速排序的总的期望运行时间为 $O(n \log n)$ （见图 12-13）。■

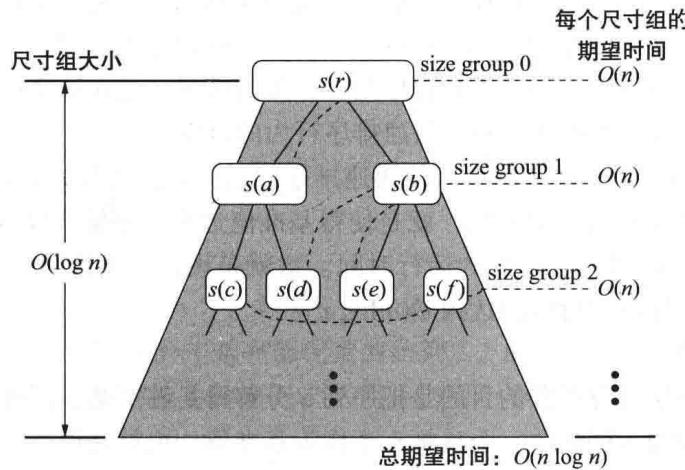


图 12-13 快速排序树 T 的时间分析。每个节点均用其子问题的大小标记显示

实际上存在很大的可能性，使得随机化快速排序的运行时间为 $O(n \log n)$ （见练习 C-12.54）。

12.3.2 快速排序的额外优化

对一个算法来说，如果它除了原始所需的内存以外，仅仅只使用少量的内存，则该算法

是就地算法。我们对于在 9.4.2 节中提到的堆排序的实现就是一个就地排序算法的例子。因为当我们在每一步递归调用中分解序列 S 时，使用了额外的容器 L 、 E 和 G ，所以代码段 12-5 中的快速排序的实现不是就地算法的例子。一个基于数组的序列的快速排序可以适配为就地的，并且这样的优化被用于大多数的部署实现。

然而，就地执行快速排序算法需要一些技巧，对于所有的递归调用，我们必须使用输入序列本身来存储其子序列。我们给出执行就地快速排序的算法 `inplace_quick_sort`，详见代码段 12-6。我们假设输入序列 S 的元素是以 Python list 的形式呈现的。就地快速排序通过使用元素交换的方法改变输入序列，并且隐式地创建新的子序列。相反，输入序列的子序列却隐式地通过一个被最左索引 a 和最右索引 b 所指定的位置范围表示出来。分解步骤是通过使用向前移动的本地变量 `left` 和向后移动的本地变量 `right` 同时扫描数组，并交换逆序的元素对实现的（见图 12-14）。当这两个索引互相经过时，分解的步骤就完成了，并且该算法会在这两个子序列上递归完成。这里没有明确的“合并”步骤是因为这两个子表的串联对于原始表的就地使用来说是隐式的。

代码段 12-6 对 Python 列表 S 的就地快速排序

```

1 def inplace_quick_sort(S, a, b):
2     """Sort the list from S[a] to S[b] inclusive using the quick-sort algorithm."""
3     if a >= b: return
4         pivot = S[b]
4             # range is trivially sorted
5         left = a
5             # will scan rightward
6         right = b - 1
6             # will scan leftward
7         while left <= right:
8             # scan until reaching value equal or larger than pivot (or right marker)
9             while left <= right and S[left] < pivot:
10                 left += 1
11             # scan until reaching value equal or smaller than pivot (or left marker)
12             while left <= right and pivot < S[right]:
13                 right -= 1
14             if left <= right:                      # scans did not strictly cross
15                 S[left], S[right] = S[right], S[left]      # swap values
16                 left, right = left + 1, right - 1          # shrink range
17
18     # put pivot into its final place (currently marked by left index)
19     S[left], S[b] = S[b], S[left]
20     # make recursive calls
21     inplace_quick_sort(S, a, left - 1)
22     inplace_quick_sort(S, left + 1, b)

```

值得注意的是，如果一个序列有重复的值，我们就不会像对原始快速排序的描述那样，明确地创建三个子序列 L 、 E 和 G 。相反，我们会允许等于基准值的元素（除了基准值本身）分散地分布在这两个子表中。练习 R-12.11 探索了我们在重复的关键值出现时所做的处理的精妙之处，练习 C-12.33 则描述了一个严格分区为三个子表 L 、 E 和 G 的就地算法。

尽管我们在这章描述的将一个序列分解为两部分的实现方法是就地的，但仍要注意，完整的快速排序算法需要的栈空间与递归树的深度是成正比的，在这种情况下树的深度最大可为 $n - 1$ 。毫无疑问，我们期望的栈的深度是比 n 要小的 $O(\log n)$ 。一个简单的技巧使得我们可以保证这个栈的大小是 $O(\log n)$ 。其主要想法是，设计一个非递归的就地快速排序版本，使用一个明确的栈来迭代地处理子问题（每一个子问题可以用标记子数组边界的索引对来表

示)。每个迭代过程都包含抛出最顶端的子问题，并将其分成两半(如果足够大的话)，然后将两个子问题入栈。这个技巧是，当入栈一个新的子问题时，我们应该首先入栈更大的子问题，然后才将较小的子问题入栈。用这种方法，子问题的规模将至少是栈的两倍，因此，这个栈的深度至多可以达到 $O(\log n)$ 。我们将这个方法的具体实现留作练习 P-12.56。

基准值的选择

在这一章，我们的实现方法在每一层快速排序的递归中，盲目地使用最后一个元素作为基准值。这使其易受到 $\Theta(n^2)$ 这种最坏情况的影响，尤其是当原始序列是一个已经有序、逆序或近乎有序的序列时。

如同在 12.3.1 节所描述的，这可以通过使用在每个分区步骤随机选择基准值的方法进行改进。在实践中，另一种选择基准值的常用技巧是使用从数组的头部、中部和尾部各自取得的树的值的中位数。这种三数取中的启发式搜索法将更多地选择到好的基准值，并且计算一棵树的中值可能相比通过生成随机数来选择基准值而言，需要更少的开销。对于更大的数据集合，可能需要计算多于三个潜在基准值的中值。

混合方法

虽然快速排序在大量的数据集合上有着非常好的性能，但却在相关的小数据集合上有着更高的开销。例如，用快速排序的方法处理 8 个元素的序列，正如图 12-10 ~ 图 12-12 中阐明的一样，包含了相当大的统计记录。在实际操作中，当我们需要排序一个很短的序列时，像插入排序(7.5 节)这样的简单算法就可以更快地执行。

因此，在最优的排序实现中，使用混合方法是屡见不鲜的：利用分治算法使子序列的大小降到某个阈值(假设 50 个元素)以下；当处于这个阈值以下时，使用插入排序直接调用上面的部分。在比较多种排序算法的性能时，我们将在 12.5 节中更多地讨论这种实际性的考虑。

12.4 再论排序：算法视角

重述一下我们对于这个视角下的排序的讨论，前面已经描述了一些方法，要么是处于最坏的情况，要么是在长度为 n 的输入序列上，预期运行时间为 $O(n \log n)$ 。这些方法包括我们在本章描述的归并排序和快速排序，同时也包括堆排序(9.4.2 节)。在这一节，我们把排序作为算法问题来学习，处理排序算法的一般问题。

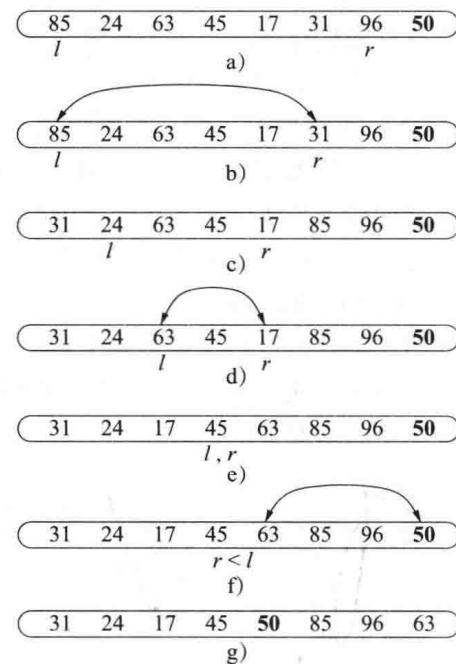


图 12-14 使用索引作为标识符 left 的简写、索引 r 作为标识符 right 的简写的就地快速排序的分解步骤。索引 l 从左到右扫描整个序列，索引 r 则从右到左扫描整个序列。当 l 所处的元素和基准值一样大，且 r 所处的元素与基准值一样小的时候发生交换。最后和基准值发生交换，然后完成分解步骤

12.4.1 排序下界

首先要问的是，我们是否可以使排序所用时间比 $O(n \log n)$ 小。有趣的是，如果排序算法的原始计算是两个元素的比较，其实这是我们可以做到的，基于比较的排序算法在最坏的情况下有 $\Omega(n \log n)$ 的运行时间下界（回想 3.3.1 节的 $\Omega(\cdot)$ 符号）。出于运行时间下界的缘故，我们只考虑比较所花费的时间，从而着重考虑基于比较的排序算法的主要花费。

设想现在给你一个 $S = (x_0, x_1, \dots, x_{n-1})$ 的序列去排序，并且假设 S 中所有元素是不同的（因为我们得到了一个下限，所以这不是一个真正的限制）。出于下界的缘故，我们不关心 S 是作为数组还是链表实现的，因为我们在那里只统计比较次数。每一次排序算法都比较两个元素 x_i 和 x_j （是否 $x_i < x_j$ ），有两种输出结果：“是”与“否”。基于这次比较的结果，排序算法可能会执行一些内部计算（我们没有统计这些时间开销），并且最后算法会执行 S 中另外两个元素的比较，这里我们将再次得到两个输出结果。因此我们可以使用描述树 T 来代表一个基于比较的排序算法（回想例题 8-6），即在 T 中的每个内部节点 v 对应一个比较，并且从位置 v 到它孩子的边对应来自“是”或者“否”答案的计算结果。值得注意的是，问题中假设的排序算法对树 T 没有明确的概念。树仅仅代表从第一次比较开始到最后一次比较结束所有可能被排序算法执行的序列。

对于每个可能的初始序列或者排列， S 中的元素将导致我们假设的排序算法执行一系列比较，遍历 T 中一条从根到一些外部节点的路径。我们关联树 T 中的每个外部节点 v ，那么 S 的排列的集合会造成排序算法在 v 中结束。在有关下界的讨论过程中，最重要的意见是 T 中的每一个外部节点都可以表示这个排列 S 中比较次数最多的序列。这个结论的证明是非常简单的：如果 S 的两个不同的排列 P_1 和 P_2 与相同的外部节点相关联，那么至少有两个对象 x_i 和 x_j ，在 P_1 中， x_i 在 x_j 的前面，在 P_2 中， x_i 在 x_j 的后面。同时，无论把 x_i 和 x_j 哪一个放在前面，与 v 相关联的输出必定是一个 S 的特定的重排序列。但是，如果 P_1 和 P_2 都导致排序算法按此顺序输出 S 中的元素，那就意味着有一个方法使得算法以错误的顺序输出 x_i 和 x_j 。因为这不被正确的排序算法允许，所以 T 中每一个外部节点都必须和一个正确的 S 序列相关联。我们使用与排序算法相关联的决策树的属性来证明以下结果。

命题 12-4：任何基于比较的排序算法对有 n 个元素的序列排序所花费时间都是 $\Omega(n \log n)$ 。

证明：按照上面的描述（见图 12-15），一个基于比较的排序算法必要的运行时间大于或等于与此排列相关联的判定树 T 的高度。通过上面的分析，每一个判定树 T 中外部节点必须与 S 中的一个排列关联。更进一步来说， S 的每一个排列必须产生一个不同的 T 中的外部节点。 n 个对象的排列的个数为 $n! = n(n-1)(n-2) \cdots 2 \cdot 1$ 。因此， T 必须具有至少 $n!$ 个外部节点。由命题 8-8， T 的高度至少为 $\log(n!)$ 。这立刻证明了命题，因为至少有 $n/2$ 项在结果 $n!$ 中是大于或等于 $n/2$ 的。因此：

$$\log(n!) \geq \log\left(\left(\frac{n}{2}\right)^{\frac{n}{2}}\right) = \frac{n}{2} \log \frac{n}{2}$$

空间复杂度为 $\Omega(n \log n)$ 。

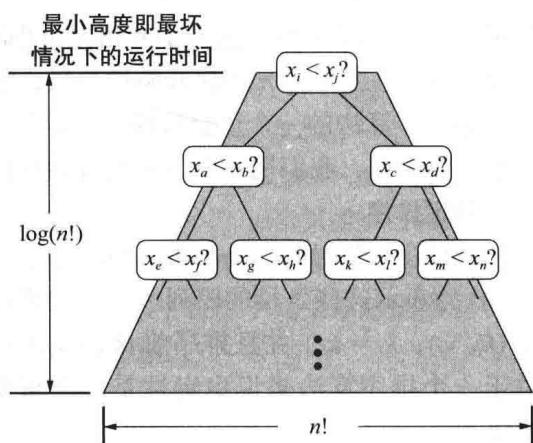


图 12-15 基于比较的排序算法的下界

12.4.2 线性时间排序：桶排序和基数排序

在上一节中，我们发现，在最坏的情况下基于比较排序算法去排序一个含有 n 个元素的序列必须花费 $\Omega(n \log n)$ 的时间。一个很自然想到的问题是，是否有其他类型的排序算法运行的渐近速度比 $O(n \log n)$ 快？有趣的是，这样的算法存在，但它们需要对输入的特殊假设序列进行排序。即使如此，这样的情况下还是经常出现在实际中，例如，在已知的范围内排序整数或排序字符串的时候，这样的讨论就是值得的。在本节中，我们考虑排序条目序列的问题，每一个键值对中的键有一个限制的类型。

桶排序

对于一个由 n 个条目构成的序列 S ， S 中的键值由 $[0, N - 1]$ 中的整数构成，并且整数 $N \geq 2$ ，并且对于序列 S 来说我们应该根据每一项中的键值来排序。在这个例子中，在 $O(n + N)$ 时间内排完序有很大的可能性。令人惊讶的是这似乎意味着，如果 N 是 $O(n)$ ，那么我们就可以在 $O(n)$ 时间内排完序。当然，非常重要的一点是由于严格限制了元素的格式，使得我们在排序过程中避免了比较。

其主要思想是使用所谓的桶排序的算法，它不是基于比较来排序，而是使用键值作为插入桶数组 B 的目录，数组 B 具有从 0 到 $N - 1$ 进行索引的网格。具有关键字 k 的项被放置在“桶” $B[k]$ 中，这个桶本身就是序列（包含键值为 k 的条目）。在将输入序列 S 的每个条目插入它的桶中之后，我们可以通过按序枚举 $B[0], B[1], \dots, B[N - 1]$ 把这些项放回 S 中。在代码段 12-7 中描述了桶排序算法。

代码段 12-7 桶排序

```
Algorithm bucketSort(S):
    Input: Sequence S of entries with integer keys in the range [0, N - 1]
    Output: Sequence S sorted in nondecreasing order of the keys
    let B be an array of N sequences, each of which is initially empty
    for each entry e in S do
        k = the key of e
        remove e from S and insert it at the end of bucket (sequence) B[k]
    for i = 0 to N-1 do
        for each entry e in sequence B[i] do
            remove e from B[i] and insert it at the end of S
```

很容易看出，桶排序运行需要 $O(n + N)$ 的时间，并且使用 $O(n + N)$ 的空间。因此，当键的值的范围 N 与序列大小 n 相比很小时，桶排序是高效的，可以说 $N = O(n)$ 和 $N = O(n \log n)$ 。而当 N 与 n 相比开始增长时，它的性能会降低。

桶排序算法的一个重要特性是：即使许多不同的元素有相同的键值，它也能得到正确的结果。事实上，我们用一些预测特殊情况的方法来描述它。

稳定排序

在排序键值对时，一个重要的问题是相等的键值是如何处理的。令 $S = ((k_0, v_0), \dots, (k_{n-1}, v_{n-1}))$ 为表示这些条目的序列。一个稳定的排序算法是指，对于 S 中任意的两个条目 (k_i, v_i) 和 (k_j, v_j) ， $k_i = k_j$ ，并且排序前 (k_i, v_i) 在 (k_j, v_j) 的前面，排序后 (k_i, v_i) 也在 (k_j, v_j) 的前面。对于一个排序算法来说稳定性是非常重要的，因为应用程序或许想用相同的键保留原始顺序。

只要我们保证把所有的序列当作元素从序列尾插入从序列头删除的队列来看待，就能保

证在代码段 12-7 中简洁描述的桶排序算法的稳定性。即当最初将 S 中的元素放置到桶的时候，我们应该从头到尾处理 S ，然后把所有的元素添加到桶的尾部。随后，当从桶中传递元素回到 S 的时候，我们应该从头到尾处理每个 $B[i]$ 元素，然后把元素添加到 S 的尾部。

基数排序

排序的稳定性如此重要的一个原因是，它允许桶排序方法应用到更加普通的名字排序而不仅仅局限于整数排序。设想一下，我们的排序项由 (k, l) 构成， k 和 l 是在 $[0, N - 1]$ 范围内的整数 ($N \geq 2$)。在这样的背景下，使用字典序来定义这些键的顺序是很常见的，如果 $k_1 < k_2$ 或者 $k_1 = k_2$ 且 $l_1 < l_2$ 时， $(k_1, l_1) < (k_2, l_2)$ 。这是一个字典比较函数的成对的版本，可以把它应用到相同长度的字符串或者长度为 d 的元组中。

基数排序算法通过对序列应用两次稳定的桶排序算法从而对具有成对键的条目的序列 S 进行排序。首先使用成对的键中的第一项作为键来排序，然后使用第二项作为键来排序。但是这种顺序是否正确呢？我们是否应该首先对 k (键对的第一项) 进行排序，然后对 l (键对的第二项) 进行排序，或者反过来？

为了在回答这个问题前获得一些直观感受，我们考虑一下下面的例题。

例题 12-5：考虑下面的序列 S (我们只显示了键)：

$$S = ((3, 3), (1, 5), (2, 5), (1, 2), (2, 3), (1, 7), (3, 2), (2, 2))$$

如果我们对 S 中键对的第一项进行稳定排序，将得到序列

$$S_1 = ((1, 5), (1, 2), (1, 7), (2, 5), (2, 3), (2, 2), (3, 3), (3, 2))$$

如果我们接着对序列 S_1 中键对的第二项进行稳定排序的话，将得到序列

$$S_{1,2} = ((1, 2), (2, 2), (3, 2), (2, 3), (3, 3), (1, 5), (2, 5), (1, 7))$$

可惜它并不是一个有序序列。另一方面，如果我们首先对 S 中键对的第二项进行稳定排序，会获得序列

$$S_2 = ((1, 2), (3, 2), (2, 2), (3, 3), (2, 3), (1, 5), (2, 5), (1, 7))$$

如果我们接着对 S_2 中键对的第一项进行稳定排序，将获得序列

$$S_{2,1} = ((1, 2), (1, 5), (1, 7), (2, 2), (2, 3), (2, 5), (3, 2), (3, 3))$$

这的确是序列 S 的字典序排序结果。

所以，从这个例子中，我们相信应该按照先第二项、后第一项的顺序来排序。这种直觉是完全正确的。按照先第二项、后第一项的顺序，我们可以保证，如果两个条目在第二次排序（按第一项）中是相等的，那么它们的起始序列（按第二项排序得到的）中的相对顺序将被保留下来。因此，所产生的序列可以保证每次都是按照字典序排序。我们留一个简单的练习 R-12.18：如何将这种方法扩展到可以测定三元组和数字的其他 d 元组。我们可以将这一节内容总结如下。

命题 12-6：假设 S 是一个有 n 个键值对的序列，序列中的每个元素都有一个键值 (k_1, k_2, \dots, k_d) ， k_i 是 0 到 $N - 1$ 的整数（其中 $N \geq 2$ ）。我们可以使用基数排序在时间复杂度 $O(d + N)$ 下得到字典序排列。

基数排序可以应用于任何键都可以被看作以字典序排序得到的小规模排序的情形。例如，我们可以将其应用于对长度适中的字符串进行排序，要求字符串中每个单独的字符可以表示为一个整数值。（一些不同长度的字符串需要进行适当处理。）

12.5 排序算法的比较

在这一节，花一点时间去思考本书中学习的所有对 n 元素序列进行排序的算法会有助于

我们更好地理解这些内容。

考虑运行时间和其他因素

我们已经学习了几种方法，如插入排序和选择排序，这两种算法在平均和最坏情况下具有 $O(n^2)$ 的时间复杂度。我们还研究了几种时间复杂度为 $O(n \log n)$ 的方法，包括堆排序、归并排序和快速排序。最后，桶排序和基数排序方法在某些类型键值下能在线性时间内运行。当然，选择排序算法在任何应用程序中都是一个糟糕的选择，因为它即使在最好的情况下运行也需要 $O(n^2)$ 的时间。但是，对于其余的排序算法，哪个是最好的呢？

很多时候，我们也无法从其余的候选中明确“最好”的排序算法。这涉及效率、内存使用和稳定性的权衡。最适合某特定应用程序的排序算法取决于该应用程序的属性。事实上，计算语言和系统使用的默认排序算法随着时间的推移已经发生了很大的变化。所以，基于一些“好”序算法的已知属性，我们可以提供一些指导和意见。

插入排序

如果情况好的话，插入排序的运行时间是 $O(n + m)$ ，其中 m 是逆序的数量（即无序元素对数目）。因此，插入排序是一种进行小序列排序的优秀算法（比如，少于 50 个元素），因为插入排序是很简单的程序，而且小序列最多只有几个逆序。此外，插入排序对“几乎”已经排序好的序列是很有效的。“几乎”是指逆序的数目很小。但是插入排序 $O(n^2)$ 的时间性能使它在这些特定情况之外成为一种糟糕的选择。

堆排序

另一方面，堆排序在最坏的情况下运行时间是为 $O(n \log n)$ ，对于基于比较的排序方法是最佳的选择。当输入的数据可以适应主存时，堆排序很容易就地执行，并且在小或中型的序列上是一个理所当然的选择。然而，堆排序在更大的序列上往往优于快速排序和归并排序。标准的堆排序由于元素的交换，并不能提供稳定排序。

快速排序

快速排序在最坏情况下的时间复杂度为 $O(n^2)$ ，虽然在一些必须保证按时完成排序操作的实时应用中它是可以接受的，但是我们仍然期待它的时间复杂度达到 $O(n \log n)$ 。并且实验研究表明，在许多测试中它优于堆排序和归并排序。由于分块步骤中存在元素交换，所以快速排序自然不能提供稳定的排序。

几十年来，快速排序是一种通用的内存排序算法的默认选择。快速排序被包含在 C 语言库中提供的 `qsort` 排序实用程序中，并且是多年来在 Unix 操作系统上的排序实用程序的基础。这也是 Java 中语言版本 6 以后的数组排序的标准算法。（我们下面讨论 Java7。）

归并排序

归并排序最坏情况下的运行时间为 $O(n \log n)$ 。做到数组的合并排序的就地操作很难，并且对于分配临时数组的额外开销无法实现最优化，而且在数组之间复制相比堆排序的就地实现和可以在计算机主存中完全适合的对序列的快速排序而言没有优势。即便如此，对于输入在计算机的各级存储器层次结构（例如，高速缓存、主存储器、外部存储器）之间被分层的情况，归并排序仍然是一个优秀的算法。在这些语境下，归并排序在很长的合并流中处理数据的方法，最好地利用了在各级存储器中以块存储的所有数据，因而减少了内存交换的总数。

GNU 排序实用程序（Linux 操作系统中的最新版本）依赖于对多路归并排序的修改。自 2003 年以来，Python 的 `list` 类的标准 `sort` 方法已经成为一种名为 Tim-sort（由 Tim Peters 设计）的混合方法。它本质上是一种自下而上的归并排序，利用一些数据的初始运行，之后进

行额外的插入排序。Tim-sort 也成为 Java7 中数组排序的默认算法。

桶排序和基数排序

最后, 如果一个应用程序用小的整型键、字符串或者来自离散范围的 d 元组键对条目进行排序, 那么桶排序和基数排序是很好的选择, 因为它的运行时间为 $O(d(n+N))$, 其中, $[0, N-1]$ 是整型键的范围 (对于桶排序来说, $d=1$)。因此, 如果 $d(n+N)$ 明显 “低于” $n \log n$ 的函数, 那么这个分类方法要比快速排序、堆排序、归并排序更快。

12.6 Python 的内置排序函数

Python 提供了两个内置的方式来给数据排序。首先是 list 类的 sort 方法。举个例子, 假设我们定义以下列表:

```
colors = ['red', 'green', 'blue', 'cyan', 'magenta', 'yellow']
```

该方法给列表中的元素进行排序, 这些元素按小于号 < 的自然定义确定顺序。上述例子中, 元素为字符串, 那么自然顺序就是按照字母表的顺序。那么调用 colors.sort(), 列表顺序变为

```
['blue', 'cyan', 'green', 'magenta', 'red', 'yellow']
```

Python 还支持一个叫作 sorted 的内置函数, 可用于产生一个新的包含任何现有的迭代容器中元素的有序表。回到我们最初的例子, 语法 sorted(colors) 将返回一个新的按字母顺序排列的 colors 列表, 而留下的原始清单的内容不变。第二种更为普遍, 因为它可以应用于任何可迭代对象作为参数的情况, 例如, sorted('green') 返回 ['e', 'e', 'g', 'n', 'r']。

键函数排序

在有很多情况下, 我们希望对元素进行不同于 < 操作符定义的自然顺序的排序。例如, 我们可能希望从短到长地排序字符串列表 (而不是按字母顺序排列)。两种 Python 的内置排序函数都允许调用者控制排序时使用的顺序的定义。这可以通过提供一个可选的关键字参数而实现, 该参数是一个二次函数的引用, 二次函数可以为原始队列的每个元素计算一个键, 之后原始元素基于它们的键值的自然顺序进行排序。(详情请看 1.5.1 节关于内置的 min 和 max 函数的技术讨论。)

键函数必须是单参数函数, 它接受一个元素作为参数并且返回一个键。例如, 我们在按字符串长度排序时可以使用内置的函数 len, 比如对字符串 s 调用 len(s) 返回其长度。为了对数组 colors 按照字符串长度进行排序, 我们用句法 colors.sort(key = len) 改变列表, 或者使用 sorted(colors, key = len) 来生成一个新的有序数组, 舍弃原始数组。当以字符串长度作为键进行排序时, 内容变成

```
['red', 'blue', 'cyan', 'green', 'yellow', 'magenta']
```

这些内置函数还支持关键字参数 reverse, 它可以设置为 True, 使得排序顺序是从最大到最小。

装饰 - 排序 - 取消设计模式

使用装饰 - 排序 - 取消设计模式实现排序

时, Python 支持键函数。它按照下面三个步骤执行:

1) 列表中的每个元素暂时地被包含应用于元素的键函数的结果的“装饰”版本所替代。

2) 列表必须按照键的自然顺序进行排序 (图 12-16)。

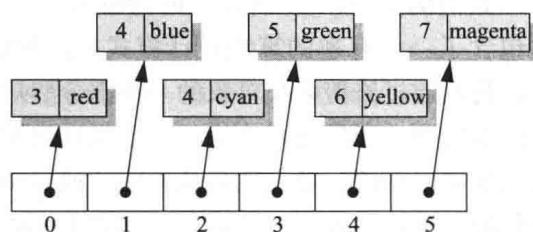


图 12-16 一个使用长度作为装饰的“装饰”字符串列表。列表按照这些键进行排序

3) 装饰的元素被原始的元素替换。

虽然 Python 中已经支持这种算法，但是如果我们要自己实现这种算法的话，表示“装饰”元素最自然的方法就是利用与用优先队列表示键 - 值对一样的策略。在代码段 9-1 中包含了这样一个 `_Item` 类，以便条目的 `<` 运算符依赖于给定的键。有这样一个组合，我们可以对任何排序算法使用装饰 - 排序 - 取消设计模式，如代码段 12-8 的归并排序所示。

代码段 12-8 基于数组的归并排序的装饰 - 排序 - 取消设计模式实现方法。`_Item` 类和 `PriorityQueueBase` 类中使用的相同（见代码段 9-1）

```

1 def decorated_merge_sort(data, key=None):
2     """ Demonstration of the decorate-sort-undecorate pattern."""
3     if key is not None:
4         for j in range(len(data)):
5             data[j] = _Item(key(data[j]), data[j])      # decorate each element
6     merge_sort(data)                                # sort with existing algorithm
7     if key is not None:
8         for j in range(len(data)):
9             data[j] = data[j].value                  # undecorate each element

```

12.7 选择

重要的是，对元素集合所要处理的各种顺序关系来说，排序不是唯一有趣的问题。对于大量的应用，相对于整个集合的排序顺序，我们对根据元素的级别来识别单个元素更感兴趣。比如确定最小和最大元素，但是我们对确定中位数更感兴趣，即除了中位数之外的一半元素比它小，剩下的一半元素比它大。一般情况下，从一个有序的列表中查找指定等级元素的查询称为顺序统计量。

定义选择问题

在这一部分，我们讨论一般的顺序统计量问题，即从未排序的 n 个可比较元素中选择第 k 个最小的元素。这被称为选择问题。当然，我们可以通过对集合进行排序然后在已排序序列的索引为 $k - 1$ 的地方插入索引来解决这个问题。我们可以使用最好的比较排序算法，它的时间复杂度是 $O(n \log n)$ ，这显然舍弃了 $k = 1$ 或 $k = n$ （或者 $k = 2, k = 3, k = n - 1, k = n - 5$ ）的情况，因为我们可以在 $O(n)$ 内为上述索引 k 的这些值解决选择问题。因此，一个自然的问题是我们是否可以在运行时间 $O(n)$ 以内解决 k 的所有值的选择问题（包括当 $k = \lfloor n/2 \rfloor$ 时寻找中位数的情况）。

12.7.1 剪枝搜索

我们确实可以在 $O(n)$ 以内为所有 k 值解决选择问题。此外，我们用来实现该结果的技术包含了一个有趣的算法设计模式。这种设计模式称为剪枝搜索或减治。应用这种设计模式，我们通过修剪 n 个对象的一小部分然后递归地解决更小的问题来解决定义在 n 个对象上的已知问题。当最终减小为一个定义在常数大小对象集合上的问题时，我们使用一些蛮力方法来解决出现的问题。然后从所有的递归调用递推回来得到结果。在一些情况下，我们可以避免使用递归，这种情况下我们只是简单地重复剪枝搜索还原步骤，直到可以使用蛮力方法，然后停止执行。顺便说一句，在 4.1.3 节描述的二分搜索方法是剪枝搜索设计模式的一个示例。

12.7.2 随机快速选择

在对 n 个元素的未排序序列应用剪枝搜索模式去寻找第 k 个最小的元素时，我们用到一种简单实用的算法，称为随机快速选择。考虑到所有由该算法生成的可能的随机选择，该算法预期的时间复杂度是 $O(n)$ ，这种期望不依赖于任何输入分配的随机性假设。注意到随机快速选择在最坏情况下的时间复杂度是 $O(n^2)$ ，其证明留作练习 R-12.24。我们还提供了练习 C-12.55，通过修改随机快速选择算法以定义确定性的选择算法，使得在最坏情况下的时间复杂度是 $O(n)$ 。然而，这种确定性算法只在理论上存在，因为大 O 表示法的隐藏常数因子在该情况下相对来说非常大。

假设我们已知一个有 n 个可比较元素并且整数 $k \in [1, n]$ 的序列 S 。某种意义上，在 S 中搜寻第 k 小的元素的快速选择算法和 12.3.1 节中描述的随机快速选择算法类似。我们从 S 中随机地选择一个“基准值”元素，然后使用此基准值将 S 细分成三个子序列 L 、 E 和 G ，分别存储 S 中比基准值小的元素、等于基准值的元素和大于基准值的元素。在修剪步骤中，我们基于 k 的值和那些子集的大小来确定这些子集包含的元素。我们又在合适的子集上重复上述步骤，注意子集中元素的等级可能和整个集合中该元素的等级不同。随机快速选择算法的实现如代码段 12-9 所示。

代码段 12-9 随机快速选择算法

```

1 def quick_select(S, k):
2     """ Return the kth smallest element of list S, for k from 1 to len(S). """
3     if len(S) == 1:
4         return S[0]
5     pivot = random.choice(S)           # pick random pivot element from S
6     L = [x for x in S if x < pivot]   # elements less than pivot
7     E = [x for x in S if x == pivot]  # elements equal to pivot
8     G = [x for x in S if pivot < x]  # elements greater than pivot
9     if k <= len(L):
10        return quick_select(L, k)      # kth smallest lies in L
11    elif k <= len(L) + len(E):
12        return pivot                 # kth smallest equal to pivot
13    else:
14        j = k - len(L) - len(E)       # new selection parameter
15        return quick_select(G, j)      # kth smallest is jth in G

```

12.7.3 随机快速选择分析

运行时间为 $O(n)$ 的随机快速选择需要一个简单的概率参数。该参数基于期望的线性，这里规定，如果 X 和 Y 是随机变量， c 是一个数字，那么

$$E(X + Y) = E(X) + E(Y) \quad \text{且} \quad E(cX) = cE(X)$$

这里我们用 $E(Z)$ 定义 Z 的期望。

假设 $t(n)$ 是大小为 n 的序列的随机快速选择的运行时间。由于该算法取决于随机事件，其运行时间 $t(n)$ 是一个随机变量。我们想要界定 $t(n)$ 的期望 $E(t(n))$ 。如果算法将 S 进行分区使得每个 L 和 G 的大小至多为 $3n/4$ ，则该算法的递归调用是“好”的。显然，一个好的递归调用的可能性至少是 $1/2$ 。假设 $g(n)$ 表示在我们得到一个好的递归调用之前递归调用的次数，包括前一个递归调用。那么我们可以使用下面的递推方程表示 $t(n)$ ：

$$t(n) \leq bn \cdot g(n) + t(3n/4)$$

其中 b 是一个大于等于 1 的常数。对于 $n > 1$ 应用期望的线性，我们得到：

$$E(t(n)) \leq E(bn \cdot g(n) + t(3n/4)) = bn \cdot E(g(n)) + E(t(3n/4))$$

由于一个递归调用是好的概率至少是 $1/2$ ，并且一个递归调用是好或者不好是独立于它的父调用的， $g(n)$ 的期望值至多是我们扔一枚硬币时在它出现正面朝上之前的次数。就是说， $E(g(n)) \leq 2$ 。因此，如果我们让 $T(n)$ 表示 $E(t(n))$ 的简写，那么对于 $n > 1$ ，有

$$T(n) \leq T(3n/4) + 2bn$$

为了将这种关系转换为一个封闭形式，假设 n 很大，我们以迭代方式应用该不等式。所以，在应用两次迭代之后，

$$T(n) \leq T((3/4)^2 n) + 2b(3/4)n + 2bn$$

在这一点上，我们应该看到，一般情况下是

$$T(n) \leq 2bn \sum_{i=0}^{\lceil \log_{4/3} n \rceil} (3/4)^i$$

换句话说，预计运行时间至多是基数小于 1 的正数的几何和的 $2bn$ 倍。因此，由命题 3-5， $T(n)$ 是 $O(n)$ 。

命题 12-7： 大小为 n 的序列 S 的随机快速选择的预期运行时间是 $O(n)$ ，假设 S 的两个元素可以在 $O(1)$ 时间内进行比较。

12.8 练习

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

巩固

- R-12.1 给出命题 12-1 的完整证明。
- R-12.2 在图 12-2 ~ 图 12-4 中的归并排序树中，一些边被画成了箭头。那些向下的箭头是什么意思？向上的箭头又是什么意思？
- R-12.3 说明为何归并排序算法在一个 n 元素序列上的运行时间为 $O(n \log n)$ ，即使 n 不是 2 的幂的时候。
- R-12.4 我们于 12.2.2 节中给出的基于数组的归并排序算法实现是否是稳定的？请解释为什么是或者为什么不是。
- R-12.5 我们在代码段 12-3 中给出的基于链表的归并排序算法实现是否是稳定的？请解释为什么是或者为什么不是。
- R-12.6 一个通过键来排序键值条目的算法被称为是离散的，如果任何时候，两个条目 e_i 和 e_j 均有相等的键，但在输入中， e_i 出现在 e_j 之前，然后在输出中，该算法将 e_i 放在 e_j 之后。描述一种方法使得 12.2 节提到的归并排序算法变得离散。
- R-12.7 假设我们有 2 个已排序的 n 元素序列 A 和 B ，并且序列中的每个元素均不相同，但其中可能有一些元素同时存在于两个序列中。描述用于计算将 A 和 B 的并集 $A \cup B$ （没有重复元素）表示为一个已排序序列的运行时间为 $O(n)$ 的方法。
- R-12.8 假设我们修改了已经确定的快速排序版本，以便选择处于 $\lfloor n/2 \rfloor$ 的元素（替代选择最后一个元素作为基准值的方法）。这种版本的快速排序在一个已排序序列上的运行时间是什么？
- R-12.9 考虑对目前确定的选择处于 $\lfloor n/2 \rfloor$ 位置的元素作为基准值的快速排序算法的版本进行改动。描述能使这个版本的快速排序运行时间为 $\Omega(n^2)$ 的序列种类。
- R-12.10 说明对于长度为 n 且元素互不相同的序列，对其快速排序的最佳运行时间在 $\Omega(n \log n)$ 之内的原因。

- R-12.11 假设函数 `inplace_quick_sort` 在有重复元素的序列上执行。证明这个算法仍然可以正确地对输入序列进行排序。在划分阶段，当有元素和基准值相同时，会发生什么？如果全部的元素均相等，那么该算法的运行时间如何？
- R-12.12 如果函数 `inplace_quick_sort` 的最外层 while 循环（代码段 12-6 的第 7 行）的条件变为 $\text{left} < \text{right}$ （而非 $\text{left} \leq \text{right}$ ），将会出现一些瑕疵。解释造成这种瑕疵的原因并给出一个会使其执行失败的特定输入序列。
- R-12.13 如果代码段 12-6 中的函数 `inplace_quick_sort` 在其第 14 行的条件变为 $\text{left} < \text{right}$ （而非 $\text{left} \leq \text{right}$ ），将会出现一些瑕疵。解释造成这种瑕疵的原因并给出一个会使其执行失败的特定输入序列。
- R-12.14 接着我们在 12.3.1 节中关于随机化快速排序的分析，请说明一个给定的输入元素 x 在尺寸组 i 中属于超过 $2\log n$ 子问题的可能性至多是 $1/n^2$ 。
- R-12.15 对于有 $n!$ 种可能的基于比较的排序算法的输入来说，在只进行 n 次比较的情况下能够实现正确排序所要求的输入的绝对上限是多少？
- R-12.16 Jonathan 有一个基于比较的排序算法可以在 $O(n)$ 的时间内对一个大小为 n 的序列的前 k 个元素进行排序。请给出一个当 k 达到最大时的大 O 表达。
- R-12.17 桶排序是否是就地的？请给出原因。
- R-12.18 描述一个基数排序，以字典序排序三元组 (k, l, m) 序列 S 。其中的 k, l, m 均为整数且属于 $[0, N-1]$ ($N \geq 2$)。如何使该组合可以延伸至 d 元组 (k_1, k_2, \dots, k_d) ，其中 k_i 是一个在 $[0, N-1]$ 中的整数。
- R-12.19 假设 S 是一个由 n 个值为 0 或 1 的元素组成的序列，使用归并排序算法对其进行排序将花费多少时间？快速排序呢？
- R-12.20 假设 S 是一个由 n 个值为 0 或 1 的元素组成的序列，使用桶排序算法对其进行稳定排序将花费多少时间？
- R-12.21 已知一个由 n 个值为 0 或 1 的元素组成的序列 S ，请描述一个算法对 S 进行就地排序。
- R-12.22 给出一个示例输入列表，归并排序和堆排序需要消耗 $O(n \log n)$ 的时间进行排序，但插入排序却只需要 $O(n)$ 的时间。如果将该列表翻转，情况如何？
- R-12.23 对以下情况来说，最好的排序算法是什么？证明你的答案。
- 一般的可比较对象
 - 长字符串
 - 32 位整型
 - 双精度浮点型数
 - 字节型数
- R-12.24 说明为何对 n 元素序列进行快速选择在最坏情况下运行时间是 $\Omega(n^2)$ 。
- 创新**
- C-12.25 Linda 要求有一个算法能接收一个输入序列 S 并生成一个输出序列 T ， T 是 n 元素序列 S 的已排序结果。
- 给出一个算法 `is_sorted`，在 $O(n)$ 的时间内测试 T 是否是有序的。
 - 解释为何该算法不足以证明一个特定的输出 T 在 Linda 的算法中是由 S 经排序而得出的。
 - 描述 Linda 的算法可以输出何种额外信息，以使得其算法的正确性可以建立在 $O(n)$ 时间内任意给定的 S 和 T 上。
- C-12.26 描述并分析一个用来移除 n 元素集合 A 中所有重复项的有效方法。

- C-12.27 扩展 PersonalList 类（详见 7.4 节），使其支持包含以下行为的 merge 方法。如果 A 和 B 是 PersonalList 类的实例，且其元素已被排序，语法 A.merge(B) 会将 B 中所有元素合并至 A 中使得 A 保持有序，B 被清空。你的方法必须通过再次连接所有已存在的节点来完成归并，但不能创建新的节点。
- C-12.28 扩展 PersonalList 类（详见 7.4 节），使其支持通过重连接已存在节点对列表元素进行排序的 sort 方法。你不能创建新的节点。你可以使用自己选择的排序算法。
- C-12.29 通过将每个元素放在各自的队列来对该元素集实现一个自底向上的归并排序，然后重复地合并队列组直到所有元素在一个队列中被排序。
- C-12.30 修改代码段 12-6 中的就地快速排序的实现方法，使其成为该算法的随机化版本，即我们在 12.3.1 节中讨论的。
- C-12.31 考虑一个确定的快速排序的版本，我们把 n 元素的输入队列中最后 d 个元素的中值作为基准值， d 是固定的奇数且 $d \geq 3$ 。在这种情况下渐近的最坏情况下的运行时间是多少？
- C-12.32 另一种分析随机化快速排序的方法是使用递归方程式。这种情况下，我们用 $T(n)$ 表示随机化快速排序的期望运行时间，然后观察该运行时间，因为其在最坏情况下对好的和坏的部分进行划分，我们可以写出

$$T(n) \leq \frac{1}{2}(T(3n/4) + T(n/4)) + \frac{1}{2}(T(n-1)) + bn$$

其中 bn 是通过给定的基准值分割列表并且在递归结束后返回的连接结果子列表所需的时间。通过归纳法说明 $T(n)$ 是 $O(n \log n)$ 。

- C-12.33 我们关于快速排序的高阶描述将元素划分成三个集 L 、 E 和 G ，分别存放小于、等于和大于基准值的关键值。然而，我们在代码段 12-6 中实现的就地快速排序不把所有等于基准值的元素收集在集合 E 中。对于就地三分割的这种方法，一个可供选择的策略如下。从左到右循环通过所有元素以维持其检索 i 、 j 和 k ，同时，所有 $S[0:i]$ 中的元素都严格地小于基准值，所有 $S[i:j]$ 中的元素都等于基准值，所有 $S[j:k]$ 中的元素都严格地大于基准值， $S[k:n]$ 中的元素均尚未归类。每次通过循环，将归类一个额外的元素，执行一个常数次的交换。请使用这种策略来实现一个就地快速排序。
- C-12.34 假设我们有一个 n 元素的序列 S 使得每个 S 中的元素都表示一位不同的总统候选人所获得的选票，每个选票作为一个整数，代表一个特定的候选人，但这些整数可能是任意大的（即使不是候选人的数量）。设计一个具有 $O(n \log n)$ 时间复杂度的算法来显示谁将赢得这场 S 位代表参与的选举，假定得票最多的候选人获胜。
- C-12.35 考虑练习 C-12.34 中的选举问题，但现在假设我们知道候选人的数量 $k < n$ ，即使这些整数 ID 会尽可能大。请描述一个具有时间复杂度 $O(n \log k)$ 的算法来决定谁将赢得这次选举。
- C-12.36 考虑练习 C-12.34 中的选举问题，但现在假设我们用整数 1 到 k 来标记 $k < n$ 位候选人。请设计一个具有时间复杂度 $O(n)$ 的算法来决定谁将赢得这次选举。
- C-12.37 说明任意的基于比较的排序算法都可以在不影响其渐近运行时间的前提下被做成稳定的算法。
- C-12.38 假设我们有两个存在全序关系定义的 n 元素序列 A 和 B ，其中可能有重复的元素。描述一个有效率的算法来决定 A 和 B 是否包含相同的元素集合。这个方法的运行时间是多少？
- C-12.39 一个 n 整型元素数组 A 的范围为 $[0, n^2 - 1]$ ，描述一个简单的方法使得对 A 的排序的运行时间为 $O(n)$ 。

- C-12.40 令 S_1, S_2, \dots, S_k 为 k 个不同的序列，其元素含有整型关键值，范围是 $[0, N - 1]$ ，参数 $N \geq 2$ 。描述一个算法使其可以在 $O(n + N)$ 的时间内生成 k 个各自已排好序的序列， n 表示这些序列的尺寸总和。
- C-12.41 给定一个具有全序关系的 n 元素序列，描述一个有效率的方法来决定 S 中是否存在两个相等的元素。你的方法的运行时间是多少？
- C-12.42 定义 S 为一个有全序关系的 n 元素序列。回想发生在序列 S 中的倒置是发生在一对元素 x 和 y 上的，使得在 S 中 x 先于 y 出现，但 $x > y$ 。描述一个运行时间在 $O(n \log n)$ 以内的算法来决定 S 中的倒置数量。
- C-12.43 定义一个 n 元素整型序列 S 。描述一个方法用以在 $O(n + k)$ 的时间内打印 S 中所有的倒置对， k 是这些倒置对的数量。
- C-12.44 S 是 n 个互相独立的整数的随机置换。证明对 S 进行插入排序的运行时间是 $\Omega(n^2)$ 。（提示：注意，已按序排好的一半元素最好放在 S 的前半部分。）
- C-12.45 定义 A 和 B 是两个 n 元素整型序列。给定一个整数 m ，请描述一个时间复杂度为 $O(n \log n)$ 的算法来决定是否在 A 中有一个整数 a 且 B 中有一个整数 b ，使得 $m = a + b$ 。
- C-12.46 给定一个 n 整数集合，描述并分析一个最快的方法来找出最接近中值的 $\lceil \log n \rceil$ 整数。
- C-12.47 Bob 有 n 个螺母，称为集合 A ，还有对应的 n 个螺钉，称为集合 B ， A 中的每个螺母仅能唯一对应 B 中的一个螺钉。不幸的是， A 中的螺母全部长得像， B 中的螺钉也长得像。Bob 唯一能进行比较的是配成 (a, b) 这样的对，使得 a 在 A 中且 b 在 B 中，并且测试 a 是大了、小了还是刚好匹配 b 。描述并分析一个有效率的算法来让 Bob 匹配所有的螺母和螺钉。
- C-12.48 我们关于快速选择的实现可以通过首先计算集合 L 、 E 和 G 的 count 数以使算法更加有空间效率，同时仅需要创建新的将用于递归的子集合。请实现这种版本的算法。
- C-12.49 用伪代码描述一个就地快速选择算法，假设允许改变元素的顺序。
- C-12.50 说明如何用一个确定的、时间复杂度为 $O(n)$ 的选择算法在最坏情况下为 $O(n \log n)$ 的条件下对一个 n 元素序列进行排序。
- C-12.51 给定一个未排序 n 个可比元素的序列 S 和一个整数 k ，给出一个期望时间为 $O(n \log k)$ 的算法来寻找 $O(k)$ 元素，顺序为 $\lceil n/k \rceil, 2\lceil n/k \rceil, 3\lceil n/k \rceil$ 等。
- C-12.52 函数 alien_split 可以取得 n 元素序列 S ，并将其在的 $O(n)$ 时间内划分成每一个的最大规模为 $\lceil n/k \rceil$ 的序列 S_1, S_2, \dots, S_k ，使得 S_i 中的每个元素都小于或等于 S_{i+1} 中的每个元素。 $i = 1, 2, \dots, k - 1$ ， $k < n$ 。说明如何使用 alien_split 在 $O(n \log n/\log k)$ 的时间内对 S 进行排序。
- C-12.53 阅读关于 Python 的排序函数中 reverse 关键字的文档，描述如何使用 decorate-sort-undecorate 实现该排序函数，而不用假设任何关键字的类型。
- C-12.54 通过回答以下问题来说明运行时间为 $O(n \log n)$ 的随机化快速排序有至少 $1 - 1/n$ 种可能，即有高可能性。
- 对每一个输入元素 x ，定义 $C_{i,j}(x)$ 为 0 或 1 的随机变量，当且仅当元素 x 属于尺寸组 i 中的 $j + 1$ 个子问题中时为 1，给出我们不需要在 $j > n$ 上定义 $C_{i,j}$ 的理由。
 - 使 $X_{i,j}$ 作为 0 或 1 的随机变量，有 $1/2^j$ 的可能性是 1，独立于任何其他的事件，并且使得 $L = \lceil \log_{4/3} n \rceil$ 。给出 $\sum_{i=0}^{L-1} \sum_{j=0}^n C_{i,j}(x) \leq \sum_{i=0}^{L-1} \sum_{j=0}^n X_{i,j}$ 成立的原因。
 - 请说明为何 $\sum_{i=0}^{L-1} \sum_{j=0}^n X_{i,j}$ 的期望值是 $(2 - 1/2^n)L$ 。

d) 说明为何 $\sum_{i=0}^L \sum_{j=0}^n X_{i,j} > 4L$ 的可能性最多是 $1/n^2$ 。使用切诺夫界，其声明了如果 X 是独立的 0/1 随机变量的有限数量之和，且期望值 $\mu > 0$ ，那么当 $e = 2.71828128\cdots$ 时， $\text{pr}(X > 2\mu) < (4/e)^{-\mu}$ 。

e) 为何之前说的可以证明随机化快速排序运行在 $O(n \log n)$ 内的可能性至少有 $1 - 1/n$ 种？

C-12.55 通过以下方法选择 n 元素序列的基准值，我们可以使快速选择算法变得确定化。

划分集合 S 为每个大小为 5 的 $\lceil n/5 \rceil$ 组（除了可能为 1 组的情况）。对每个小集合进行排序并标记该集合的中值元素。对于这个 $\lceil n/5 \rceil$ “小” 中值，应用选择算法递归地找出这些小中值的中值。使用该元素作为基准值，并在快速选择算法中进行。

通过回答以下问题来说明这个确定化的快速选择算法是运行在 $O(n)$ 时间内的（请忽略向上或向下取整函数以简化数学计算）。

- a) 有多少小中值小于等于选择的基准值？有多少大于等于基准值？
- b) 对每个小于等于基准值的小中值来说，有多少其他元素小于等于基准值？对于那些大于等于基准值的元素来说是否有同样的结论？
- c) 说明为何寻找确定的基准值的方法和用它对 S 进行划分的操作将花费 $O(n)$ 的时间。
- d) 基于这些评估，为这个选择算法写一个递归等式来限定最坏情况运行时 $t(n)$ 。（注意，在最坏情况下将有两个递归调用——一个是寻找小中值的中值，另一个是在更大的 L 和 G 中递归寻找。）
- e) 使用该递归等式，通过归纳法来说明 $t(n)$ 是 $O(n)$ 。

项目

- P-12.56 实现一个非递归的、就地的快速排序算法。该算法曾在 12.3.2 节末描述过。
- P-12.57 比较就地快速排序和非就地快速排序的性能。
- P-12.58 执行一系列基准测试来决定归并排序和快速排序哪个执行得更快。你的测试不但应当包含“随机”序列，还应包含“几乎”已排序的序列。
- P-12.59 实现确定化的和随机化的快速排序算法并执行一系列基准测试，以显示哪个更快。你的测试应当包含非常“随机”的序列，以及基本上已经有序的序列。
- P-12.60 实现一个就地插入排序算法和一个就地快速排序算法。执行基准测试来决定 n 的值的范围，其中快速排序 n 值范围平均比插入排序的 n 值范围大。
- P-12.61 设计并实现一个桶排序算法，用来排序列表。该列表有 n 个条目，均为整型，且来自范围 $[0, N-1]$ ($N \geq 2$)。该算法必须运行于 $O(n+N)$ 的时间内。
- P-12.62 为本章所提到的一种排序算法设计并实现一个动画。你的动画应当以直观的方式阐明该算法的关键性质。

扩展阅读

Knuth 的关于排序和查找^[65]的经典文献包含了广泛的关于排序问题的历史及解决它们的算法。Huang 和 Langston^[53]说明了如何在线性时间内就地合并两个已排序列表。快速排序算法的标准应当归功于 Hoare^[51]。很多快速排序的最优化均由 Bentley 和 McIlroy^[16] 描述。更多的关于随机化的描述，包括 Chernoff 约束，可以在附录以及 Motwani 和 Raghavan^[80] 的书中找到。本章中给出的快速排序分析是基于本书早先的 Java 版本，并结合了来自 Kleinberg 和 Tardos^[60] 的分析。练习 C-12.32 归功于 Littman。Gonnet 和 Baeza-Yates^[44] 分析并实验性地比较了多种排序算法。术语“剪枝搜索”源自于计算机几何学的著作（诸如 Clarkson^[26] 和 Megiddo^[75] 的工作）。术语“减治”来自于 Levitin^[70]。

文本处理

13.1 数字化文本的多样性

虽然多媒体信息很丰富，但文本处理依然是计算机的一个主要功能。计算机可用于编辑、存储和显示文件，并通过互联网传送文件。此外，数字系统用于归档广泛的文本信息，并且新数据正在以很快的增长速度产生。一个大型的语料库可以轻而易举地拥有超过 PB 级的数据（相当于一千万亿字节，或者一百万兆字节）。包括文本信息集合的常见例子如下：

- 万维网的快照，以互联网文本格式 HTML 和 XML 为主要文本格式，它们用于为多媒体内容添加标签。
- 在用户计算机上本地存储的所有文件。
- 电子邮件归档。
- 顾客评论。
- 社交网站状态更新的编辑，如 Facebook。
- 微博网站的供稿，例如 Twitter 和 Tumblr。

这些集合包括数百种国际语言的书面文本。此外，还有即使不是语言，也可以从计算上视为“串”的大数据集（例如 DNA）。

在本章中，我们会探讨一些可以用来有效地分析和处理大数据文字集的基本算法。除了一些有趣的应用程序之外，文字处理算法还突出了一些重要的算法设计模式。

首先考虑在文章的较长一段文字中搜索子串时产生的问题，例如，搜寻文件中的一个字时产生的问题。解决模式匹配问题可以使用穷举法（brute-force method），这种方法虽然具有广泛的适用性，但往往是低效的。

接着，我们引入了动态规划（dynamic programming）算法，它可以在特定条件下解决多项式时间内的问题，而这些问题刚开始出现的时候需要指数时间去解决。我们在字符串匹配（即部分字符串相同，但不是完全相同）问题上展示了这种技术的运用。这种问题出现在对单词拼写错误提出建议或者试图匹配相关遗传样本的时候。

由于文本数据集十分庞杂，因此对其进行压缩十分重要，通过减少网络传输的字节数来降低对文档长期存储的需求。对于文本压缩，我们可以采用贪心算法（greedy method），这往往能就困难的问题得到近似的解决方案，并且对于一些问题（例如文本压缩）可以得到优化算法。

最后，我们梳理了一些有特殊用途的数据结构。这些数据结构用于更好地组织文本数据，从而支持更高效的查询。

字符串的表示法和 Python 的 str 类

我们在讨论文本处理的算法时使用字符串作为文本模型。字符串可能来源于科学、语言和互联网等各种应用。比如下面这些字符串：

```
S = "CGTAAACTGCTTTAATCAAACGC"
T = "http://www.wiley.com"
```

第一个字符串 S 来自 DNA 应用程序，第二个字符串 T 是本书出版商的 URL。我们可以参阅附录 A 了解 Python 的 str 类支持的操作。

为了便于算法描述，假设字符串中的字符来自已知的字母表 (alphabet)，我们把字母表表示为 Σ 。例如，在 DNA 的背景下，标准字母表中有四个符号， $\Sigma = \{A, C, G, T\}$ 。这个字母表 Σ 可能是 ASCII 或 Unicode 字符集的一个子集，但也有可能是其他更一般的字符集。尽管假设一个字母表有固定的有限尺寸（表示为 $|\Sigma|$ ），但尺寸也可以是不确定的（非平凡的），就像 Python 对 Unicode 字母表的处理，它允许多于一百万个不同的字符。因此，我们在文本处理算法的渐近分析中要考虑 $|\Sigma|$ 的影响。

一些字符串处理操作涉及把大字符串分成一些小字符串。为了能够讨论从这些操作中产生的结果，我们需要依赖于 Python 的索引 (indexing) 和切片 (slicing) 符号。为了标记方便，用 S 表示一个长度为 n 的字符串。在这种情况下，用 $S[j]$ 表示索引为 j 的符号，其中 $0 \leq j \leq n - 1$ 。用 $S[j:k]$ 表示由 $S[j]$ 到 $S[k - 1]$ 构成的子串（注意，不是 $S[k]$ ），构成 S 的一部分（或者子串 (substring)）。按照这个定义，应注意子串 $S[j:j + m]$ 的长度为 m ，子串 $S[j:j]$ 一般为长度为 0 的空串 (null string)。按照 Python 的约定，当 $k < j$ 时，子串 $S[j:k]$ 也是空子串。

为了区分一些特殊类型的子串，我们需要把 $S[0:k]$ ($0 \leq k \leq n$) 这种形式的任意子串作为 S 的前缀 (prefix)，当 Python 的切片符号中省略第一个索引时也会产生这样的前缀，如 $S[:k]$ 。同样， $S[j:n]$ ($0 \leq j \leq n$) 这种形式的任意子串是 S 的后缀 (suffix)，当 Python 的切片符号中省略第二个索引时会产生这样的后缀，如 $S[j:]$ 。举个例子，如果再次把 S 作为上面给出的 DNA 的字符串，“CGTAA” 就是 S 的一个前缀，“CGC” 是 S 的一个后缀，“C” 既是 S 的前缀也是后缀。注意，空串是任何字符串的前缀和后缀。

13.2 模式匹配算法

在经典的模式匹配问题中，我们给出了长度为 n 的文本字符串 T 和长度为 m 的模式字符串 P，并希望明确是否 P 是 T 的一个子串。如果是，则希望找到 P 在 T 中开始位置的最低索引 j ，比如 $T[j:j + m]$ 和 P 匹配，或者从 T 中找到所有 P 的开始位置索引。

模式匹配问题在 Python 的 str 类中有许多内在的行为，例如 P in T、T.find(P)、T.index(P) 及 T.count(P)，这些行为是更复杂的行为中的子任务，例如 T.partition(P)、T.split(P) 和 T.replace(P, Q)。

在本节中，我们将提出三种模式匹配算法，这三种算法的困难程度逐渐增加。为简单起见，我们在字符串类的 find 方法上对函数的外部语义进行建模，在该模式开始的时候返回最低的索引，如果模式没有找到，则返回 -1。

13.2.1 穷举

如要搜索或者优化某些功能，穷举算法设计模式是一种强大的技术。在一般情况下运用这种技术时，我们通常会列举输入相关的所有可能情况，并挑出列举的所有情况的最优情况。

在运用这种技术来设计一个穷举模式匹配算法时，我们推导出了可能是所要解决的第一个算法——我们简单地测试了 P 相对于 T 产生的所有可能性。该算法实现如代码段 13-1 所示。

代码段 13-1 穷举模式匹配算法的实现

```

1 def find_brute(T, P):
2     """Return the lowest index of T at which substring P begins (or else -1)."""
3     n, m = len(T), len(P)          # introduce convenient notations
4     for i in range(n-m+1):        # try every potential starting index within T
5         k = 0                      # an index into pattern P
6         while k < m and T[i + k] == P[k]:    # kth character of P matches
7             k += 1
8         if k == m:                  # if we reached the end of pattern,
9             return i                # substring T[i:i+m] matches P
10    return -1                   # failed to find a match starting with any i

```

性能

对穷举模式匹配算法的分析很简单。它由两个嵌套的循环组成：一个是在文本模式所有可能的开始索引进行外部循环索引；另一个是在模式的每个字符之间进行内部循环索引，并将它和文章中潜在对应的字符进行比较。因此，通过穷举搜索方法，穷举模式匹配算法的正确性立刻就能得到保证。

在最坏的情况下，穷举模式匹配的运行时间很长，因为对于 T 中的每个索引，无论如何都要对 m 个字符进行比较，最后却可能发现在当前的索引下 P 和 T 并不匹配。参考代码段 13-1，我们看到外部 for 循环至多被执行了 $n - m + 1$ 次，内部 while 循环至多执行了 m 次。因此，穷举方法最坏情况下的运行时间是 $O(nm)$ 。

例题 13-1：假设给出如下的文本字符串

$T = "abacaabaccabacabaabb"$

模式字符串为

$P = "abacab"$

图 13-1 说明了穷举模式匹配算法在 T 和 P 上的执行过程。

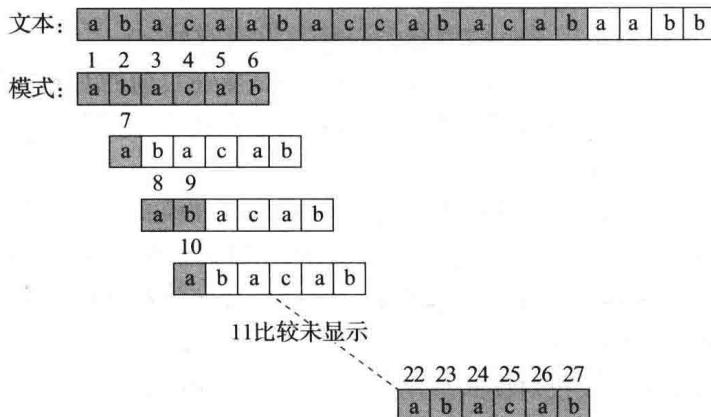


图 13-1 穷举模式匹配算法的运行示例。该算法对 27 个字符进行比较，字符上方用数字标签表示

13.2.2 Boyer-Moore 算法

起初，为了找出作为子串的模式 P 或者排除它存在的可能性，检查 T 中的每个字符似乎是非常必要的。但并不总是如此。我们在本节中研究的 Boyer-Moore 模式匹配算法有时可以避免对 P 和 T 中占很大比例的字符进行比较。在本节中，我们会描述 Boyer 和 Moore 提出的简化版原始算法。

Boyer-Moore 算法的主要思想是通过增加两个可能省时的启发式算法来提升穷举算法的运行时间。这些启发式算法大致如下：

- 镜像启发式 (looking-glass heuristic)：当测试 P 相对于 T 可能的位置时，可以从 P 的尾部开始比较，然后从后向前移动直到 P 的头部。
- 字幕跳跃启发式 (character-jump heuristic)：在测试 P 在 T 中可能的位置时，有着相应模式字符 $P[k]$ 的文本字符 $T[i] = c$ 的不匹配情况按如下方法处理。如果 P 中任何位置都不包含 c，则将 P 完全移动到 $T[i]$ 之后（因为它不能匹配 P 中任何一个字符）；否则，直到 P 中出现字符 c 并与 $T[i]$ 一致才移动 P。

我们将会尽快形式化这些启发式算法，但直观上来讲，它们作为一个完整的团体进行工作。镜像启发式通过设置其他启发式来避免 P 和 T 整个群组之间的所有字符进行比较。至少在这种情况下，通过倒着匹配可以更快地到达目的，因为如果在考虑 P 在 T 中的确定位置时遇到了不匹配，我们可以利用字符跳跃启发式算法相对于 T 大幅度移动 P 来避免大量无用的比较。如果及早运用字符跳跃启发式算法测试 P 相对于 T 的位置，它将起到很大的作用。图 13-2 展示了这些启发式的一些简单应用。

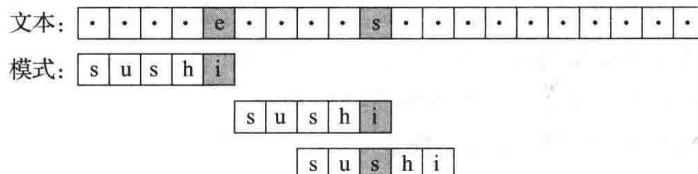


图 13-2 直观展示 Boyer-Moore 模式匹配算法的一个简单示例。原来的比较结果导致了文本字符 e 的不匹配。因为那个字符不在模式中，整个模式从当前的位置跳了过去。第二个比较同样是不匹配的，但是不匹配字符 s 在模式的其他地方出现了。模式向下移动，因此 s 的最后一次出现与文本中相应的 s 对齐。该方法的其余部分并没有在该图中显示

图 13-2 的例子是相当基础的，因为它仅仅涉及该模式最后一个字符的不匹配情况。一般情况下，当最后一个字符的匹配被找到时，该算法试图在目前的对齐情况下对该模式的倒数第二个字符扩展匹配。这个进程持续进行，直到整个模式匹配完全或者在模式的某些内部位置发现不匹配时才停止进行。

如果发现不匹配，并且文章中不匹配的字符没有出现在模式中，则直接将整个模式字符串从当前位置跳过去，就像图 13-2 最初陈述的那样。如果不匹配字符发生在模式的其他位置，则必须根据它最后出现的位置是在不匹配对齐的模式的字符之前还是之后来考虑两种子情况。这两种情况如图 13-3 所示。

在图 13-3b 的情况下，仅仅对模式移动一个单元。直到找到不匹配字符 $T[i]$ 在模式中的另一个出现的位置才向右移动，这样是更加有效的，但是我们不希望花费时间去寻找另一个出现的位置。Boyer-Moore 算法的高效依赖于创建一个查阅的表，使其能够更快地定位模式中不匹配的字符发生在其他哪个地方。特别地，我们定义一个函数 $\text{last}(c)$ 如下：

- 如果 c 在 P 中， $\text{last}(c)$ 是 c 在 P 中最后一次出现的索引；否则，默认定义 $\text{last}(c) = -1$ 。

如果假设字母表是固定的、有限大小的，并且那些字符可以转变成一个数组的索引（例如，通过使用它们的字符代码），可以简单地将最后一个功能实现为一个查找表，该表在查找 $\text{last}(c)$ 函数值的时候，最坏情况下的时间复杂度是 $O(1)$ 。然而，这个表的长度和字母表的大小相等（而不是模式的大小），并且还需要初始化整个表的时间。

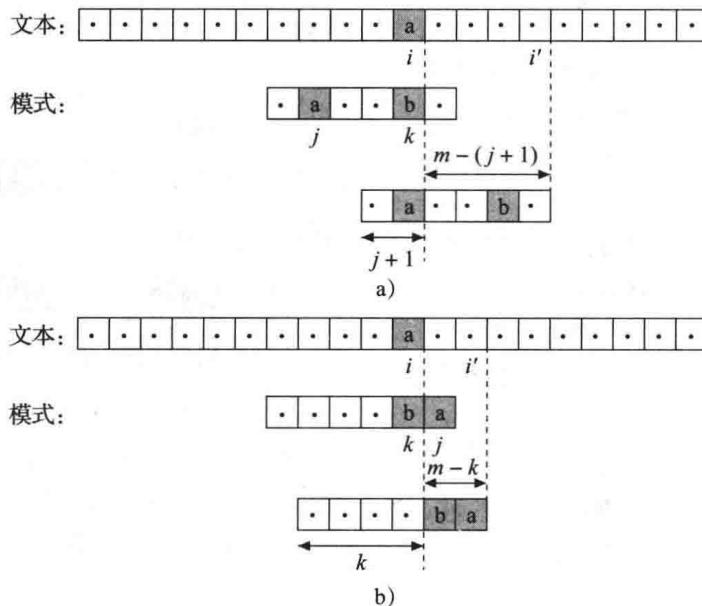


图 13-3 Boyer-Moore 算法的字符跳跃启发式的附加规则。令 i 代表文章中不匹配字符的索引, k 代表模式中出现的索引, j 代表 $T[i]$ 在模式中最后一次出现位置的索引。我们区分两种情况: ① $j < k$, 这种情况下对模式移动 $k - j$ 个单元, 因此索引 i 会前进 $m - (j + 1)$ 个单元; ② $j > k$, 这种情况下对模式移动 1 个单元, 索引 i 会前进 $m - k$ 个单元

我们用哈希表来实现, 仅仅包含在结构中出现的来自模式的那些字符。这种方法使用的空间与模式中不同字符的数量成正比, 因此空间复杂度为 $O(m)$ 。期望的查询时间与问题的规模无关 (尽管最坏情况的界限是 $O(m)$)。我们在代码段 13-2 中给出了 Boyer-Moore 模式匹配算法的完整实现。

代码段 13-2 Boyer-Moore 算法的实现

```

1 def find_boyer_moore(T, P):
2     """Return the lowest index of T at which substring P begins (or else -1)."""
3     n, m = len(T), len(P)
4     if m == 0: return 0
5     last = {}
6     for k in range(m):
7         last[P[k]] = k
8     # align end of pattern at index m-1 of text
9     i = m-1
10    k = m-1
11    while i < n:
12        if T[i] == P[k]:
13            if k == 0:
14                return i
15            else:
16                i -= 1
17                k -= 1
18        else:
19            j = last.get(T[i], -1)
20            i += m - min(k, j + 1)
21            k = m - 1
22    return -1

```

Boyer-Moore 模式匹配算法的正确性是通过这样的方式来保证的, 即该方法的每一次移

位都保证不会“跳过”任何可能的匹配。因为 $\text{last}(c)$ 是 c 在 P 中最后一次出现的位置。在图 13-4 中，我们将说明 Boyer-Moore 模式匹配算法在类似例题 13-1 的一个输入字符串的情况下执行过程。

性能

如果使用传统的查找表，在最坏的情况下 Boyer-Moore 算法的运行时间是 $O(nm + |\Sigma|)$ 。即，最后一个功能的计算需要花费时间 $O(m + |\Sigma|)$ ，并且该模式的实际搜寻在最坏的情况下花费时间为 $O(nm)$ ，和穷举算法的花费时间一样 ($|\Sigma|$ 的依赖在有哈希表的情况下被移除。) 一个文本模式达到最坏的情况的一个例子是

c last(c)	a b c d
	4 5 3 -1

文本: [a | b | a | c | a | a | b | a | d | c | a | b | a | c | a | b | a | a | b | b]
1

模式: [a | b | a | c | a | **b**]
4 3 2
[a | b | a | **c** | a | **b**]
13 12 11 10 9 8
[a | b | a | c | a | **b**]
5
[a | b | a | c | a | **b**]
6
[a | b | a | c | a | **b**]
7

图 13-4 Boyer-Moore 模式匹配算法的说明，包括 $\text{last}(c)$ 函数的概述。该算法执行了 13 个字符的比较，字符上方用数字标签表示

$$\begin{aligned} T &= \underbrace{\text{aaaaaa}\cdots\text{a}}_n \\ P &= \underbrace{\text{baa}\cdots\text{a}}_{m-1} \end{aligned}$$

然而，英文文本不太可能有最坏的情况，因为在这种情况下，Boyer-Moore 算法往往能够跳过文本的大部分。英文文本的实验证据表明，每个字符作比较的平均数量是每 5 个字符模式字符串中有 0.24 次比较。

我们实际上提出了 Boyer-Moore 算法的简化版本。每当原始算法改变模式超过字符跳跃启发式时，原始算法通过对部分匹配的文本字符串使用替代转变启发式达到的运行时间为 $O(n + m + |\Sigma|)$ 。这个替代转变启发式是基于借鉴 Knuth-Morris-Pratt 模式匹配算法的主要思想。

13.2.3 Knuth-Morris-Pratt 算法

如例题 13-1 所示，在特定实例情况下，测试穷举算法和 Boyer-Moore 匹配算法的最差性能。对于模式的一个确定的调整，如果发现一些匹配的字符但后来又发现不匹配，在模式下一次重新匹配时，我们忽略所有由成功的比较获得的信息。

在本节讨论的 Knuth-Morris-Pratt（或者“KMP”）算法，避免了信息的浪费，并且它能达到的运行时间为 $O(n + m)$ ，这是渐近最优运行时间。即在最坏的情况下，任何模式匹配算法将对文本的所有字符和模式的所有字符检查至少一次。KMP 算法的主要思想是预先计算模式部分之间的自重叠，从而当不匹配发生在一个位置时，我们在继续搜寻之前就能立刻知道移动模式的最大数目。一个很好的例子如图 13-5 所示。

失败函数

为了实现 KMP 算法，我们会预先计算失败函数 f ，该函数用于表示匹配失败时 P 对应的位移。具体地，失败函数 $f(k)$ 定义为 P 的最长前缀的长度，它是 $P[1:k+1]$ 的后缀（注意，我们这里没有包含 $P[0]$ ，因为至少会移动一个单元）。直观地说，如果在字符 $P[k+1]$ 中找到不匹配，函数 $f(k)$ 会告诉我们多少紧接着的字符可以用来重启模式。例题 13-2 描述了图 13-5 例子中模式的失败函数的值。

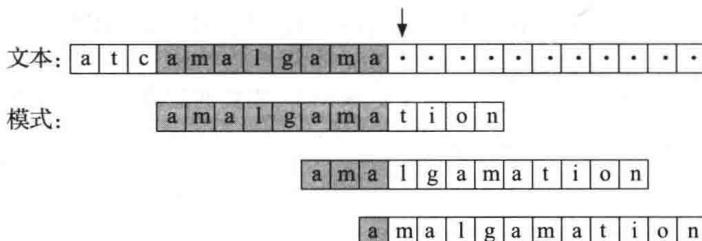


图 13-5 Knuth-Morris-Pratt 算法的一个示例。如果一个不匹配发生在指定的位置，模式应该被移动到第二个对齐的位置，并不特别需要用 AMA 前缀去重新检查部分匹配。如果不匹配的字符不是 1，下一次匹配将充分利用已经匹配的公共字母 a

例题 13-2：考虑从图 13-5 得到的模式 $P = \text{"amalgamation"}$ 。对于在下面展示的字符串 P ，Knuth-Morris-Pratt(KMP) 失败函数为 $f(k)$ 。

k	0	1	2	3	4	5	6	7	8	9	10	11
$P[k]$	a	m	a	l	g	a	m	a	t	i	o	n
$f(k)$	0	0	1	0	0	1	2	3	0	0	0	0

实现

KMP 模式匹配算法的实现如代码段 13-3 所示。它依赖于一个有效的函数 `compute_kmp_fail` 计算 KMP 的失败，该函数可以有效计算失败函数。

代码段 13-3 KMP 模式匹配算法的实现。`compute_kmp_fail` 效用函数在代码段 13-4 中给出

```

1 def find_kmp(T, P):
2     """Return the lowest index of T at which substring P begins (or else -1)."""
3     n, m = len(T), len(P)                                # introduce convenient notations
4     if m == 0: return 0                                    # trivial search for empty string
5     fail = compute_kmp_fail(P)                           # rely on utility to precompute
6     j = 0                                                 # index into text
7     k = 0                                                 # index into pattern
8     while j < n:
9         if T[j] == P[k]:                               # P[0:k] matched thus far
10            if k == m - 1:                             # match is complete
11                return j - m + 1
12            j += 1                                     # try to extend match
13            k += 1
14        elif k > 0:
15            k = fail[k-1]                            # reuse suffix of P[0:k]
16        else:
17            j += 1
18    return -1                                         # reached end without match

```

KMP 算法的主要部分是它的 while 循环，每一次迭代会对 T 中索引 j 的字符和 P 中索引 k 的字符进行比较。如果这次比较的结果是匹配，算法在 T 和 P 中会移动到下一个字符（或者，如果到达模式的最后，将报告一个匹配的结果）。如果比较失败了，算法会对 P 中的新候选字符导出失败函数；否则，就从 T 中的下一个索引开始（因为没有东西可以被重复使用）。

KMP 失败函数的构建

为了构建失败函数，我们使用代码段 13-4 中的方法，这是一个“引导过程”，它将模式与 KMP 中的模式进行比较。每次有两个匹配的字符，我们设置 $f(j) = k + 1$ 。注意，因为在

整个算法中已经有 $j > k$, 当使用它的时候, $f(k - 1)$ 总是被定义得很好。

代码段 13-4 compute_kmp_fail 的实现, 用于支持 KMP 模式匹配算法。注意算法是如何使用失败函数之前的值去有效地计算新值

```

1 def compute_kmp_fail(P):
2     """Utility that computes and returns KMP 'fail' list."""
3     m = len(P)
4     fail = [0] * m           # by default, presume overlap of 0 everywhere
5     j = 1
6     k = 0
7     while j < m:
8         if P[j] == P[k]:
9             fail[j] = k + 1
10        j += 1
11        k += 1
12    elif k > 0:
13        k = fail[k-1]
14    else:
15        j += 1
16    return fail

```

性能

除去失败函数的计算外, KMP 算法的运行时间明显正比于 while 循环的迭代次数。为了方便分析, 令 $s = j - k$ 。直观地说, s 是模式 P 关于文本 T 移动的总数。需要注意的是, 在整个算法执行过程中, $s \leq n$ 。以下三种情况中的某一种发生在循环的每次迭代时。

- 如果 $T[j] = P[k]$, j 和 k 每次增加 1, 因此, s 不发生改变。
- 如果 $T[j] \neq P[k]$ 且 $k > 0$, j 不改变并且 s 至少增加 1, 因为在这种情况下, s 在 $j - k$ 到 $j - f(k - 1)$ 之间发生改变, 这是 $k - f(k - 1)$ 的附加, 因为 $f(k - 1) < k$ 是确定的。
- 如果 $T[j] \neq P[k]$ 且 $k = 0$, 因为 k 不会改变, 所以 j 和 s 每次增加 1。

因此, 在循环的每次迭代中, j 或者 s 每次至少增加 1 (也可能两个都会增加)。因此, 在 KMP 模式匹配算法中, while 循环的迭代总次数至多为 $2n$ 。当然, 为了实现这一约束, 应假设已经计算出了 P 的失败函数。

计算失败函数的算法的运行时间为 $O(m)$ 。它的分析方法类似于主要的 KMP 算法, 有一个长度为 m 的模式和它自己进行比较。因此, 我们得出:

命题 13-3: Knuth-Morris-Pratt 算法执行长度为 n 的文本字符串和长度为 m 的模式字符串的匹配, 所需的运行时间为 $O(n+m)$ 。

该算法的正确性是根据失败函数的定义而来的。任何跳过的比较其实都是没有必要的, 因为失败函数保证了所有忽略的比较都是多余的——会多次涉及比较相同的匹配字符。

在图 13-6 中, 我们说明了 KMP 模式匹配算法对例 13-1 中输入字符串的执行。

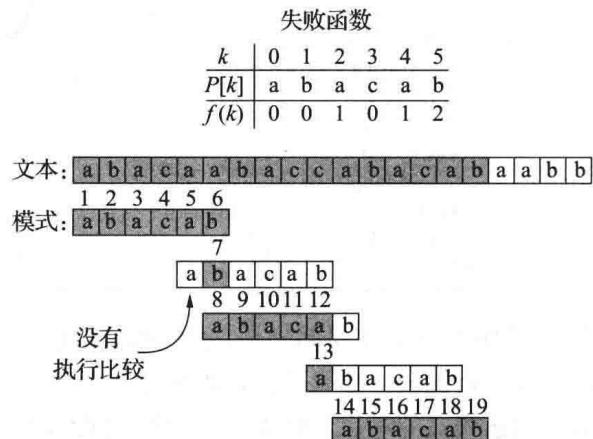


图 13-6 KMP 模式匹配算法的说明。该基本算法对 19 个字符进行比较, 用数字标签进行表示 (在失败函数的计算中会执行附加的比较)

注意，之所以使用失败函数，是为了避免对模式中的字符和文本中的字符进行重复比较。同样需要注意的是，在对相同的字符串进行所有比较时，该算法比贪心算法运行的次数更少（见图 13-1）。

13.3 动态规划

在本节中，我们将讨论动态规划算法设计技术。该技术和分而治之算法（12.2.1 节）类似，可应用于各种不同的问题。动态规划经常用于解决一些问题，这些问题可能需要用指数时间和多项式时间算法去解决。另外，由动态规划技术的应用所产生的算法通常是相当简单的，只需要比几行代码多一点的代码去描述填写在表中的一些嵌套循环。

13.3.1 矩阵链乘积

我们首先给出一个经典、具体的例子，而不是先对动态规划技术的通用部分进行说明。假设给定 n 个二维矩阵的集合，用来计算这 n 个矩阵的数学乘积。

$$\mathbf{A} = \mathbf{A}_0 \cdot \mathbf{A}_1 \cdot \mathbf{A}_2 \cdot \cdots \cdot \mathbf{A}_{n-1}$$

其中， \mathbf{A}_i 是一个 $d_i \times d_{i+1}$ 矩阵，其中 $i = 0, 1, 2, \dots, n-1$ 。在标准矩阵乘法算法中（将会使用的一个算法），乘以一个 $d \times e$ 的矩阵 \mathbf{B} 再乘以 $e \times f$ 的矩阵 \mathbf{C} ，计算结果 \mathbf{A} 为

$$\mathbf{A}[i][j] = \sum_{k=0}^{e-1} \mathbf{B}[i][k] \cdot \mathbf{C}[k][j]$$

这个定义意味着矩阵乘法具有结合律，也就是说， $\mathbf{B} \cdot (\mathbf{C} \cdot \mathbf{D}) = (\mathbf{B} \cdot \mathbf{C}) \cdot \mathbf{D}$ 。因此，可以对 \mathbf{A} 的表达式以任何方式加圆括号，并且得到相同的答案。然而，没有必要对每一个圆括号表达式执行相同数量的原始（即标量）增加，如以下示例所示。

例题 13-4：令 \mathbf{B} 是一个 2×10 的矩阵， \mathbf{C} 是一个 10×50 的矩阵， \mathbf{D} 是一个 50×20 的矩阵。计算 $\mathbf{B} \cdot (\mathbf{C} \cdot \mathbf{D})$ 需要 $2 \cdot 10 \cdot 20 + 10 \cdot 50 \cdot 20 = 10\,400$ 次乘法，而计算 $(\mathbf{B} \cdot \mathbf{C}) \cdot \mathbf{D}$ 需要 $2 \cdot 10 \cdot 50 + 2 \cdot 50 \cdot 20 = 3000$ 次乘法。

矩阵链乘积问题是决定定义乘积 \mathbf{A} 的表达式的圆括号表达式，用以减少执行乘法标量的总数量。正如上面的例子所示，圆括号表达式之间的差异可能很大，因此找到一个好的解决方案可能会明显提高速度。

定义子问题

解决矩阵链乘法的一个方式是简单地列举 \mathbf{A} 的圆括号表达式的所有可能，并且确定每一个执行的乘法的数量。不幸的是， \mathbf{A} 的所有不同的圆括号表达式的设置和所有不同的具有 n 个叶子二进制树的设置相同。这个数是 n 的指数。因此，这个简单（“贪心”）算法运行时间为指数时间，因为有指数数量种为组合算术表达式加圆括号的方法。

可以通过贪心算法显著改善实现的性能，不过，也可以通过对矩阵乘积链问题的性质进行一些观测来改善实现的性能。首先，这个问题可以被分成子问题。在这种情况下，可以定义多个不同的子问题，每一个都是为了计算子表达式 $\mathbf{A}_i \cdot \mathbf{A}_{i+1} \cdot \cdots \cdot \mathbf{A}_j$ 最好的圆括号表达式。作为一个简要的表示法，使用 $N_{i,j}$ 来表示计算这个子表达式需要的乘法的最小数量。因此，原始矩阵链乘法问题可以被定性为计算 $N_{0,n-1}$ 的值。这个观察是重要的，但是为了应用动态规划技术，我们需要进行更多观察。

表征最优解

另外一个可以对矩阵链乘法问题做的重要的观察是：就一个特别的子问题的最佳解决方案

而言，对它的子问题表征一个最佳解决方案是可能的。我们把这个属性叫作子问题最优条件。

在矩阵链乘法问题的情况下，我们观察到，无论怎样对一个子表达式加圆括号，最终必然会执行一些矩阵乘法运算。也就是说，子表达式 $A_i \cdot A_{i+1} \cdot \dots \cdot A_j$ 完整的圆括号表达式必须是 $(A_i \cdots A_k) \cdot (A_{k+1} \cdots A_j)$ ，其中， $k \in \{i, i+1, \dots, j-1\}$ 。此外，对于任意确切的 k ，乘积 $(A_i \cdots A_k)$ 和 $(A_{k+1} \cdots A_j)$ 都必须被最优解决。如果不是这样，那将是全局最优，即每个子问题都被有效解决。但这是不可能的，因为接下来可能通过一个子问题的最优解决方案重置当前子问题的解决方案来减少乘法的总数量。就其他子问题优化解决方案而言，这个发现提出了一种对于 $N_{i,j}$ 明确定义最优问题的方式。也就是说，我们可以通过考虑每个 k 的位置来计算 $N_{i,j}$ ，在 k 的位置可以放置最后的乘法并从中取最小值。

设计动态规划算法

我们可以表征子问题最优解决方案 $N_{i,j}$ 为

$$N_{i,j} = \min_{i \leq k < j} \{N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1}\}$$

其中， $N_{i,j} = 0$ ，因为对单个矩阵不需要进行任何操作。也就是说， $N_{i,j}$ 是最小值，它占据所有可能的位置去执行最终的乘法，即计算每个子表达式需要的乘法的数目加上执行最后的乘法需要的数目。

注意，有一类问题会禁止我们把其分解成独立的子问题（这是为了应用分而治之技术）。不过，可以通过计算自底向上方式产生的 $N_{i,j}$ 值和在 $N_{i,j}$ 值的表中存储中间解决方案的方式，使用 $N_{i,j}$ 的方程得到一个高效的算法。我们可以通过指定 $N_{i,i} = 0 (i = 0, 1, \dots, n-1)$ 简单地开始。我们可以应用 $N_{i,j}$ 的一般方程去计算 $N_{i+1,i+1}$ 的值，因为它们仅仅需要可用的 $N_{i,i}$ 和 $N_{i,j}$ 的值。 $N_{i,i+1}$ 的值已经给出，我们接下来可以计算 $N_{i,i+2}$ 的值，并以此类推。因此，直到最终计算出一直在寻找的 $N_{0,n-1}$ 的值，才可以从以前计算得到的值中推导出 $N_{i,j}$ 的值。这种动态规划解决方案的 Python 实现在代码段 13-5 中给出。

代码段 13-5 矩阵链乘积的动态规划算法

```

1 def matrix_chain(d):
2     """d is a list of n+1 numbers such that size of kth matrix is d[k]-by-d[k+1].
3
4     Return an n-by-n table such that N[i][j] represents the minimum number of
5     multiplications needed to compute the product of Ai through Aj inclusive.
6     """
7     n = len(d) - 1                                # number of matrices
8     N = [[0] * n for i in range(n)]               # initialize n-by-n result to zero
9     for b in range(1, n):                          # number of products in subchain
10        for i in range(n-b):                      # start of subchain
11            j = i + b                             # end of subchain
12            N[i][j] = min(N[i][k] + N[k+1][j] + d[i]*d[k+1]*d[j+1] for k in range(i,j))
13    return N

```

因此，可以用主要包含了三个嵌套循环（第三个嵌套循环计算最小项）的算法来计算 $N_{0,n-1}$ 。每个这样的循环每次执行时最多迭代 n 次，它的内部具有恒定数量的附加工作。因此，这个算法的总运行时间为 $O(n^3)$ 。

13.3.2 DNA 和文本序列比对

一个常见的出现在遗传学和软件工程中的文本处理问题是测试两个文本字符串的相似性。在遗传学中，两个字符串对应于 DNA 的两条链。同样，在软件工程中，两个字符串可

能是来自相同程序的两个不同版本的代码源，为此，我们需要确定一个版本和下一版本之间所做的改变。事实上，确定两个字符串之间的相似性如此普遍，以至于 UNIX 和 Linux 操作系统各有一个用来比较文本文件但名称不同的内置程序。

给定一个字符串 $X = x_0 x_1 x_2 \dots x_{n-1}$, X 的一个子序列是任何具有 $x_{i_1} x_{i_2} \dots x_{i_k}$ 形式的字符串，其中 $i_j < i_{j+1}$ ；也就是说，这是一个字符序列，不必连续，但却是从 X 中按顺序取得的。例如，字符串 AAAG 是字符串 CGATAATTGAGA 的子序列。

这里讨论的 DNA 和文本相似性的问题是最长公共子序列 (LCS) 问题。在这个问题中，通过一些字母表（例如在计算遗传学中常见的字母表 {A, C, G, T}）给出两个字符串， $X = x_0 x_1 x_2 \dots x_{n-1}$ 和 $Y = y_0 y_1 y_2 y_{m-1}$ ，然后要求找出最长的字符串 S , S 是 X 和 Y 共同的子序列。一种解决最长公共子序列问题的方法是列举 X 的所有子序列，并找出同样是 Y 的子序列中最大的一个。由于每个 X 中的字符无论在不在子序列中，都有可能有 2^n 个不同的 X 的子序列，每个子序列确定其是否是 Y 的子序列需要的时间是 $O(m)$ 。因此，这种蛮力方法产生了一个非常低效的具有指数时间的算法，其运行时间为 $O(2^{nm})$ 。幸运的是，用动态规划可以有效地解决 LCS 问题。

动态规划解决方案的组件

如上所述，动态规划技术主要应用在希望找到做某事的最优解的优化问题中。如果问题具有一定的属性，我们可以在这样的情况下运用动态规划技术：

- 简单子问题：必须有一些方式将全局优化问题重复地划分为子问题。而且，应该有只用一些索引来参数化子问题的方式。
- 子问题优化：全局问题的优化解决方案必须是由子问题优化解决方案组成的。
- 子问题重复：无关子问题的优化解决方案可以包含共同的子问题。

对 LCS 问题应用动态规划

回想一下，在 LCS 问题中所得到的两个字符串，即长度为 n 的 X 和长度为 m 的 Y ，并且要求找到一个最长的字符串 S , S 是 X 和 Y 的子序列。因为 X 和 Y 都是字符串，我们有一个定义子问题的固有索引设置——字符串 X 和 Y 的索引。定义一个子问题，因此，作为计算值 $L_{j,k}$ ，我们将用它来表示最长字符串的长度，最长字符串是前缀 $X[0:j]$ 和 $Y[0:k]$ 的一个子序列。这个定义允许我们针对子问题优化解决方案重写 $L_{j,k}$ 。其定义取决于图 13-7 所示的两种情况。

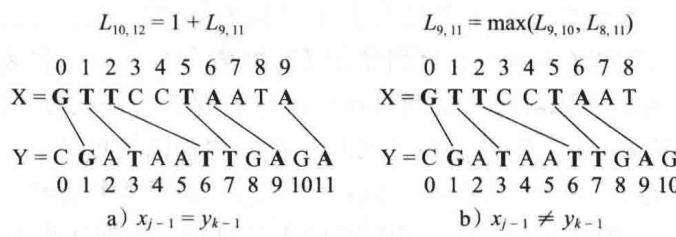


图 13-7 最长公共子序列算法在计算 $L_{j,k}$ 时的两种情况

- $x_{j-1} = y_{k-1}$ 。在这种情况下，我们对 $X[0:j]$ 的最后一个字符和 $Y[0:k]$ 的最后一个字符进行匹配。声明这个字符属于 $X[0:j]$ 和 $Y[0:k]$ 的最长共同子序列。为了证明这个声明，先假设它是不正确的，则必有最长共同子序列 $x_{a_1} x_{a_2} \dots x_{a_c} = y_{b_1} y_{b_2} \dots y_{b_c}$ 。如果 $x_{a_c} = x_{j-1}$ 或者 $y_{b_c} = y_{k-1}$ ，则通过设置 $a_c = j-1, b_c = k-1$ 得到相同的子序列。接下来，如果 $x_{a_c} \neq x_{j-1}$ 并且 $y_{b_c} \neq y_{k-1}$ ，则通过增加 $x_{j-1} = y_{k-1}$ 到最后甚至可以得到更长的公共子序列。这样的

话, $X[0:j]$ 和 $Y[0:k]$ 的最长公共子序列以 x_{j-1} 结束。因此, 可以设定

$$L_{j,k} = 1 + L_{j-1,k-1}, \text{ 如果 } x_{j-1} = y_{k-1}$$

- $x_{j-1} \neq y_{k-1}$ 。在这种情况下, 不能得到同时包含 x_{j-1} 和 y_{k-1} 的共同子序列。也就是说, 可以得到一个以 x_{j-1} 结束的或者以 y_{k-1} 结束的共同子序列 (或者可能都不会得到), 但是这两者不可能同时得到。因此, 设定

$$L_{j,k} = \max\{L_{j-1,k}, L_{j,k-1}\}, \text{ 如果 } x_{j-1} \neq y_{k-1}$$

我们注意到: 因为切片 $Y[0:0]$ 是空字符串, $L_{j,0} = 0$, $j = 0, 1, \dots, n$; 类似地, 因为切片 $X[0:0]$ 是空字符串, $L_{0,k} = 0$, 其中 $k = 0, 1, \dots, m$ 。

LCS 算法

$L_{j,k}$ 满足子问题优化的定义, 因为既没有最长公共子序列, 也没有子问题的最长公共子序列。此外, 它使用子问题重复, 因为子问题解决方案 $L_{j,k}$ 可以在一些其他的问题中使用 (即问题 $L_{j+1,k}$ 、 $L_{j,k+1}$ 和 $L_{j+1,k+1}$)。将 $L_{j,k}$ 的定义转变为一个算法实际上非常简单。我们创建一个 $(n+1) \times (m+1)$ 维数组 L , 定义 $0 \leq j \leq n$ 并且 $0 \leq k \leq m$ 。初始化所有项为 0, 特意使形式 $L_{j,0}$ 和 $L_{0,k}$ 的所有项为 0, 然后反复地建立 L 的值直至得到 X 和 Y 的最长共同子序列的长度 $L_{n,m}$ 。这个算法的 Python 实现在代码段 13-6 中给出。

代码段 13-6 LCS 问题的动态规划算法

```

1 def LCS(X, Y):
2     """Return table such that L[j][k] is length of LCS for X[0:j] and Y[0:k]."""
3     n, m = len(X), len(Y)                      # introduce convenient notations
4     L = [[0] * (m+1) for k in range(n+1)]      # (n+1) x (m+1) table
5     for j in range(n):
6         for k in range(m):
7             if X[j] == Y[k]:                      # align this match
8                 L[j+1][k+1] = L[j][k] + 1
9             else:                                # choose to ignore one character
10                L[j+1][k+1] = max(L[j][k+1], L[j+1][k])
11    return L

```

LCS 算法的运行时间非常容易分析, 因为它由两个嵌套循环控制, 外部循环迭代 n 次, 内部循环迭代 m 次。因为每个循环内的 if 语句和分配需要 $O(1)$ 的基本操作, 所以这个算法的运行时间为 $O(nm)$ 。因此, 动态规划技术可运用于最长共同子序列问题, 并通过 LCS 问题的指数时间的蛮力解决方案得到显著改善。

代码段 13-6 的 LCS 函数计算了最长公共子序列的长度 (记为 $L_{n,m}$), 但不是子序列自己。幸运的是, 如果通过 LCS 函数计算出来的 $L_{j,k}$ 的值完全列在一张表中, 则提取实际最长公共子序列是很容易的。该解决方案通过逆向进行长度 $L_{n,m}$ 的估算可以从后往前地重建。在任何位置 $L_{j,k}$, 如果 $x_j = y_k$, 则基于在公共的字符 x_j 之前的长度 $L_{j-1,k-1}$ 的公共子序列的长度。可以将 x_j 记作为子序列的一部分, 然后从 $L_{j-1,k-1}$ 继续进行分析。如果 $x_j \neq y_k$, 则可以移动到 $L_{j,k-1}$ 和 $L_{j-1,k}$ 中较大的一个。我们继续上述过程, 直到某个 $L_{j,k} = 0$ (例如, j 或者 k 是 0 作为边界的情况)。这一策略的 Python 实现在代码段 13-7 中给出。这个函数构造了在 $O(n+m)$ 的附加时间里构建了一个最长的公共子序列, 因为无论 j 或者 k (或者两个都), while 循环的每次执行都会递减。计算最长公共子序列算法的说明如图 13-8 所示。

代码段 13-7 最长公共子序列的重建

```

1 def LCS_solution(X, Y, L):
2     """Return the longest common substring of X and Y, given LCS table L."""

```

```

3  solution = []
4  j,k = len(X), len(Y)
5  while L[j][k] > 0:           # common characters remain
6    if X[j-1] == Y[k-1]:
7      solution.append(X[j-1])
8      j -= 1
9      k -= 1
10   elif L[j-1][k] >= L[j][k-1]:
11     j -= 1
12   else:
13     k -= 1
14  return ''.join(reversed(solution))  # return left-to-right version

```

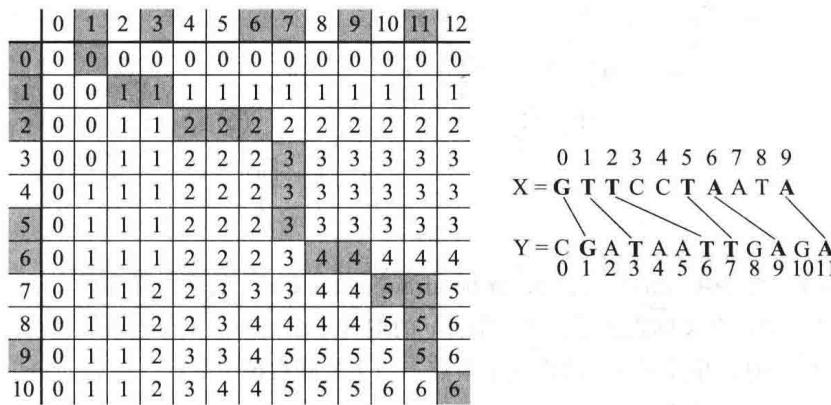


图 13-8 从数组 L 中重建最长公共子序列算法的说明。在着重显示路径上的对角线步骤代表公共字符的使用（在序列中字符的各指标在边缘着重显示）

13.4 文本压缩和贪心算法

本节讨论一个重要的文本处理任务——文本压缩。在这个问题中，我们给定一个由一些字母组成的字符串，例如选用 ASCII 或者 Unicode 字符集，此外，我们想高效地将 X 编码成一个很小的二进制字符串 Y （仅使用字符 0 和 1）。当希望降低数字通信的带宽时，文本压缩是非常有用的，这样做可以减少传输文本所需的时间。同样，文本压缩可以更有效地存储大文档，这样做可以允许一个固定容量的存储装置尽可能地包含更多的文件。

本节探讨的文本压缩方法是霍夫曼编码。标准的编码方案（例如 ASCII）是使用固定长度的二进制字符串去编码字符（在传统的或者扩展的 ASCII 系统中分别用 7 位或者 8 位来编码）。Unicode 系统最初由 16 位固定长度来表示，然而常见的编码通过允许公共组字符（例如那些来自 ASCII 系统，由更少位编码的字符）来减少空间的使用。霍夫曼编码在有固定长度的编码情况下，使用短码字符串对高频字符进行编码，并用长码字符串对低频字符进行编码，以节省空间。另外，霍夫曼编码充分使用一个可变长度的编码对任何字母表上给出的字符串 X 进行编码。基于对字符频率的使用进行优化，其中，对于每个字符 c ，计数 $f(c)$ 是 c 出现在字符串 X 中的次数。

为了对字符串 X 进行编码，我们将 X 中的每一个字符转换为一个可变长度编码文字，并且为了减少 Y 对 X 的编码，我们联结所有编码文字。为了避免产生歧义，应确保在编码中没有任何编码文字是另一个编码文字的前缀。这样的代码被称为前缀码，并且为了检索 X

而简化了 Y 的解码 (见图 13-9)。即使有这样的限制, 由可变长度的前缀码产生的节省也是十分显著的, 尤其是在字符频率差异较大的情况下 (如自然语言文本在所有书面语言中的情况)。

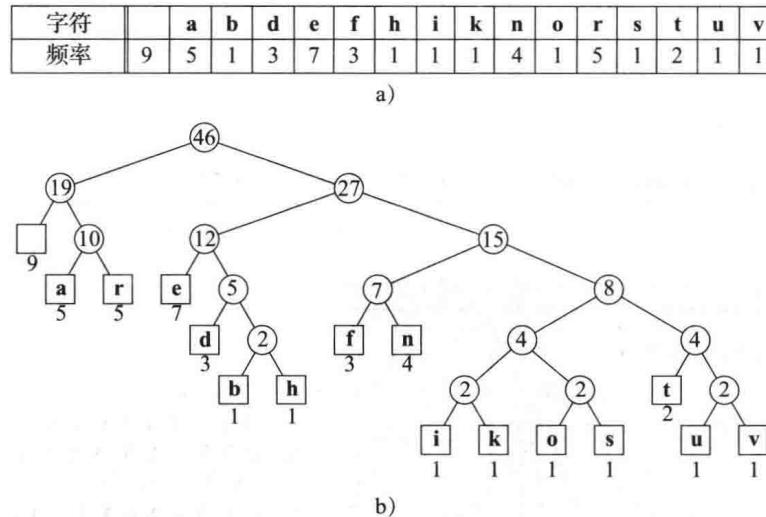


图 13-9 字符串 $X = "a \text{ fast runner need never be afraid of the dark"$ 的霍夫曼编码。a) X 中每个字符的频率; b) 字符串 X 的霍夫曼树 T 。字符 c 的编码是由根节点 T 和存储 c 的叶子节点之间的跟踪路径得到的, 并且用 0 关联左孩子节点, 用 1 关联右孩子节点。例如, “r”的编码是 011, “h”的编码是 10111

霍夫曼算法对 X 产生的最优可变长度前缀码, 是基于代表该代码的二进制树 T 的建立。 T 的每一条边代表代码字的一位, 到左孩子节点的那条边代表“0”, 到右孩子节点的那条边代表“1”。每一个叶子 v 和一个特殊的字符相关联, 并且该字符的代码字由与从 T 的根到 v 之间的边相关联的位的子序列定义 (见图 13-9)。每个叶子 v 有一个频率 $f(v)$, 它仅仅是 X 中与 v 相关联字符的频率。另外, 我们给 T 中的每一个内部节点 v 一个频率 $f(v)$, 它是以 v 为根的子树中所有叶子的频率的总和。

13.4.1 霍夫曼编码算法

霍夫曼编码算法将字符串 X 的每个不同字符 d 解码成一个单节点的二进制树的根节点。该算法会执行很多轮。在每一轮, 该算法取两个具有最小频率的二进制树, 然后把它们合并为一棵二进制树。重复这一过程, 直到只有一棵树 (见代码段 13-8)。

代码段 13-8 霍夫曼编码算法

Algorithm $\text{Huffman}(X)$:

Input: String X of length n with d distinct characters

Output: Coding tree for X

Compute the frequency $f(c)$ of each character c of X .

Initialize a priority queue Q .

for each character c in X **do**

- Create a single-node binary tree T storing c .
- Insert T into Q with key $f(c)$.

while $\text{len}(Q) > 1$ **do**

- $(f_1, T_1) = Q.\text{remove_min}()$
- $(f_2, T_2) = Q.\text{remove_min}()$

```

Create a new binary tree  $T$  with left subtree  $T_1$  and right subtree  $T_2$ .
Insert  $T$  into  $Q$  with key  $f_1 + f_2$ .
 $(f, T) = Q.\text{remove\_min}()$ 
return tree  $T$ 

```

霍夫曼算法的每一个 while 循环的迭代通过使用由堆表示的优先队列在 $O(\log d)$ 时间内实现。另外，每个迭代从 Q 中取出两个节点并且向 Q 中添加一个节点，在正好一个节点被留在 Q 之前，程序将会重复 $d - 1$ 次。因此，这个算法运行时间为 $O(n + d \log d)$ 。尽管关于这个算法的正确性的全部验证不在本书所述范围之内，但是需要注意，它的灵感来自一个简单的想法——任何一个最佳的节点可以被转换为对两个最不频繁的字母 a 和 b 的 code-words，仅仅在它们的最后一个比特不同的最理想的节点。对一个有 a 并且 b 被替换为 c 的字符串进行重复的讨论，得到以下观点。

命题 13-5：霍夫曼算法为一个长度为 n 并且有 d 个不同字符的字符串构造一个最优的前缀代码的时间复杂度为 $O(n + d \log d)$ 。

13.4.2 贪心算法

用于构建最优编码的霍夫曼算法是贪心算法的设计模式示例之一。这个设计模式应用于优化问题，我们试图在最小化或者最大化该结构的一些特性的时候构造一些结构。

贪心算法模式的一般公式和蛮力方法的几乎一样简单。为了使用贪心算法解决给出的优化问题，我们选择一个序列进行。序列从一些很好理解的开始条件开始，然后计算那些初始条件的花费。这个模式要求通过识别从所有当前可能的选择中实现最优成本改善的决定来迭代地做出附加的选择。这个方法并不总能产生最优的解决方案。

但是有几个问题是不可以解决的，并且这些问题可以说都具有贪心选择的特性，即全局最优的性质可以通过一系列局部最优选择来实现（即选择是当时可用的可能性之中每一个当前最优的选择）。计算最优可变长度的前缀代码的问题只是具有贪心选择特性的问题的示例之一。

13.5 字典树

13.2 节的模式匹配算法通过对模式进行预处理来加速在文本中的搜索（在 Knuth-Morris-Pratt 算法中计算失败函数或者在 Boyer-Moore 算法中计算最后函数）。本节采取了一个互补的方法，即呈现了预处理文本的字符串搜寻算法。这个方法非常适合对一个固定文本执行一系列请求的应用，因此预处理文本的原始花费通过在每个随后的查询中加速来获得补偿（例如，对莎士比亚的《哈姆雷特》提供模式匹配的网站或者提供关于“哈姆雷特”主题的搜索引擎）。

字典树是为了支持最快模式匹配的存储字符串的基于树的数据结构。字典树主要应用于信息检索中。事实上，名字“tries”来自于单词“retrieval”。在信息检索应用中，例如在一个染色体组的数据库中搜索一个确定的 DNA 序列，我们已经得到了字符串的集合 S ，所有定义使用了相同的字母表。字典树支持的主要查询操作是模式匹配和前缀匹配。接下来的操作为：给定一个字符串 X ，查找以 X 作为前缀的所有在 S 中的字符串。

13.5.1 标准字典树

令 S 为一个来自字母表 Σ 的 s 个字符串的集合，而且 S 中的字符串不是其他字符串的前缀。 S 的标准字典树是一棵具有下列特性的有序树 T （见图 13-10）：

- 除了根之外的 T 的每个节点，都用 Σ 中的字符作标签。
- T 的内部节点的孩子节点有不同的标签。
- T 有 s 个叶子节点，每个叶子节点和 S 中的一个字符串相关联，从根到 T 的一个叶子节点 f 的路径的标签的串联产生了和 v 相关联的 S 的字符串。

因此，一棵字典树表示拥有从根到 T 的叶子路径的 S 的字符串。需要注意的是， S 中没有一个字符串是另一个字符串的前缀，这一点非常重要。它确保了 S 的每个字符串和 T 中的一个叶子是唯一相关的（这和 13.4 节描述的霍夫曼编码的前缀代码的限制是类似的）。我们总是可以通过在每个字符串的末尾添加一个不在原始的字母表 Σ 中的特殊字符来满足这个假设。

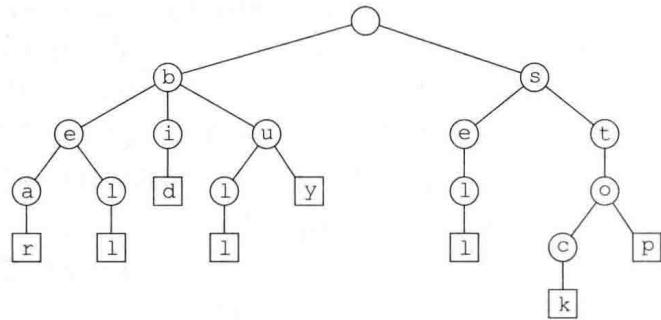


图 13-10 字符串 {bear, bell, bid, bull, buy, sell, stock, stop} 的标准字典树

一棵标准字典树的内部节点可以在任何地方有 $1 \sim |\Sigma|$ 个孩子节点。对每个在集合 S 中的字符串的第一个字符来说，都有一条从根节点 r 到它的其中一个孩子节点的边。此外，从 T 的根节点到深度为 k 的内部节点 v 之间的路径相当于 S 的一个字符串 X 的 k 字符前缀 $X[0:k]$ 。

事实上，对每个可以跟随 S 集合的字符串中的前缀 $X[0:k]$ 的字符 c ，有一个用字符 c 作标签的 v 的孩子节点。这样，字典树就简明地存储了存在于一系列的字符串之间的共同前缀。

作为一种特殊情况，如果在字母表中仅有两个字符，那么字典树本质上是一棵二进制树，它可能仅包含一个孩子节点的一些内部节点（即它可能是一棵不标准的二进制树）。一般来说，尽管一个内部节点可能最多能有 $|\Sigma|$ 个孩子，但实际上这样的节点的平均度数可能会更小。例如，图 13-10 中的字典树有一些仅包含一个孩子节点的内部节点。在更大的数据集合中，节点的平均度数可能在更大深度的树中会更小，因为可能会有更少的分享共同前缀的字符串，所以该模式会有更少的延续。此外，在更多的语言中，将会有不可能自然发生的字符组合。

接下来的命题提供了一些关于标准字典树的重要的结构特性。

命题 13-6：一个存储来自字母表 Σ 的总长度为 n 的 s 个字符串的集合 S 的标准字典树有以下的特性：

- T 的高度和 S 中最长的字符串的长度相等。
- T 的每个内部节点至多有 $|\Sigma|$ 个孩子。
- T 有 s 个叶子节点。
- T 的节点的数目至多是 $n + 1$ 。

对于字典树节点的数目而言，最坏的情况发生在没有两个字符串分享一个共同的非空前缀时，即除了根节点，所有内部节点都只有一个孩子节点。

字符串的集合 S 的一棵字典树 T 可以用来实现主键是 S 的字符串的集合或者图。也就是说，我们通过追踪由 X 中的字符指示的从根开始的路径，在 T 中对字符串 X 执行搜索。如果该路径可以被追踪并且在一个叶子节点结束，那么 X 是图的主键。例如，在

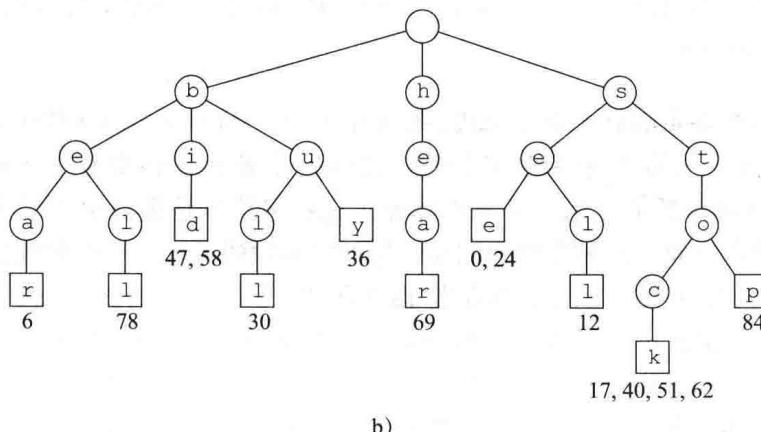
图 13-10 的字典树中，追踪“bull”的路径在一个叶子节点结束。如果路径不能被追踪或者路径能被追踪但是在一个内部节点结束，那么 X 不是图的主键。在图 13-10 所示的例子中，对“bet”的路径不能被追踪并且对“be”的路径在一个内部节点结束。在图中没有这样的单词。

可以很容易地知道搜索长度为 m 的字符串所需的运行时间为 $O(m \cdot |\Sigma|)$ ，因为我们访问了至多 $m + 1$ 个 T 的节点，并且在每个节点上确定孩子节点有后续字符作为标签所花费的时间是 $O(|\Sigma|)$ 。在 $O(|\Sigma|)$ 的上限时间内去定位一个具有给定标签的孩子节点是可以实现的，即使一个节点的孩子节点是无序的，因为至多有 $|\Sigma|$ 个孩子节点。可以将花费在一个节点上的时间提高到 $O(\log |\Sigma|)$ 或者期望的 $O(1)$ (如果 $|\Sigma|$ 非常小 (就像 DNA 字符串的情况一样))，方法是对每一个节点使用一个次级搜索表或者哈希表，或者对每一个节点使用一个大小是 $|\Sigma|$ 的有向查阅表将字符映射到孩子节点。因为这些原因，通常预计搜索一个长度为 m 的字符串的运行时间是 $O(m)$ 。

综上所述，我们可以使用一棵字典树去执行模式匹配的特殊类型，这称为词汇匹配，即判定一个给定的模式是否能正确地匹配文本中的一个单词。词汇匹配和标准模式匹配有所不同，因为模式不能匹配文本的任意一个子串——仅匹配单词的其中一个。为了实现词汇匹配，原始文献的每个单词必须都加到字典树中（见图 13-11）。这个方案的简单扩展支持前缀匹配的查询。然而，文本中模式的随机发生（例如，模式是单词的正确的前缀或者跨越两个单词）不能高效执行。

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22
s e e a b e a r ? s e l l s t o c k !
23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45
s e e a b u l l ? b u y s t o c k !
46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68
b i d s t o c k ! b i d s t o c k !
69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88
h e a r t h e b e l l ? s t o p !

a)



b)

图 13-11 标准字典树的词汇匹配：a) 被搜索的文本（文章和介词，也称为禁用词）；b) 在文本中单词的标准字典树，给定的单词在开始工作的索引中突出显示了叶子节点。例如单词 stock 节点的叶子，单词在文本的索引 17、40、51 和 62 处开始

为了构建一个字符串集合 S 的标准字典树，可以使用一次插入一个字符串的增量算法。回想 S 中的字符串没有一个是另一个字符串的前缀的假设，为了在当前字典树 T 中插入一个字符串 X ，追踪在 T 中和 X 相关联的路径，当陷入僵局的时候创建一个新的节点链去存储 X 的剩余字符。插入长度为 m 的 X 的运行时间和搜索时间类似，最坏情况下时间复杂度为 $O(m \cdot |\Sigma|)$ ，或者如果对每个节点使用次级哈希表，则期望时间为 $O(m)$ 。因此，对 S 集合构建完整的字典树花费了期望的 $O(n)$ 时间，其中 n 是 S 的字符串的总长度。

标准字典树潜在的空间效率低下促进了压缩字典树的发展。压缩字典树（因为历史的原因）也称为基数树，即在只有一个孩子节点的标准字典树中可能有许多节点，并且这种节点的存在是浪费的。接下来讨论压缩字典树。

13.5.2 压缩字典树

压缩字典树和标准字典树类似，但是它确保字典树中的每个内部节点至少有两个孩子节点。它通过压缩单个孩子节点的链为单条边执行规则（见图 13-12）。定义 T 是一个标准字典树。如果 T 的一个内部节点 v 有一个孩子节点并且不是根节点，则说 v 是多余的。例如，图 13-10 的字典树有 8 个多余的节点。同样，对于 $k \geq 2$ 条边的链，

$$(v_0, v_1)(v_1, v_2) \cdots (v_{k-1}, v_k)$$

是多余的，如果：

- 对于 $i = 1, \dots, k-1$, v_i 是多余的。
- v_0 和 v_k 不是多余的。

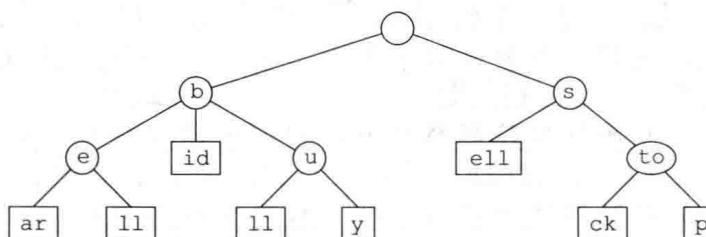


图 13-12 字符串 $\{\text{bear}, \text{bell}, \text{bid}, \text{bull}, \text{buy}, \text{sell}, \text{stock}, \text{stop}\}$ 的压缩字典树（将该表和图 13-10 所示的标准字典树进行比较）。除了在叶子节点的压缩，请注意有标签的内部节点被 `stock` 和 `stop` 这两个单词分享

可以通过将每个多余的有 $k \geq 2$ 条边的链 $(v_0, v_1)(v_1, v_2) \cdots (v_{k-1}, v_k)$ 替换为一条单独的边 (v_0, v_k) ，并用节点 v_1, \dots, v_{k-1} 的标签的串联来重新标记 v_k ，来将 T 转变为一个压缩字典树。

因此，压缩字典树的节点用字符串作标签，这些字符串是集合中的子字符串或者字符串，而不是用单独的字符。压缩字典树相对于标准字典树的优势是节点的数目和字符串的数目成正比，而不是和总长度成正比，如命题 13-7 所述。

命题 13-7： 存储大小为 d 的取自字母表的 s 个字符串的集合 S 的压缩字典树有以下特性：

- T 的每个内部节点至少有两个孩子节点，至多有 d 个孩子节点。
- T 有 s 个叶子节点。
- T 的节点的数目是 $O(s)$ 。

细心的读者可能想知道路径的压缩是否有某种显著优势，因为它被相应节点标签的扩充

所抵消了。事实上，压缩字典树仅当在已经存储在基本结构中的字符串集合之上作为辅助索引结构，以及不需要实际存储在集合中的字符串的所有字符时才有优势。

假设字符串的集合 S 是字符串 $S[0], S[1], \dots, S[s-1]$ 的数组。不是显式地存储节点的标签 X ，而是用三个数字 $(i, j:k)$ 的组合隐式地表示它，就像 $X = S[i][j:k]$ ，即 X 是包含第 j 个以后但不包含第 k 个字符的 $S[i]$ 的切片（见图 13-13 的例子。同样和图 13-11 的标准字典树进行比较）。

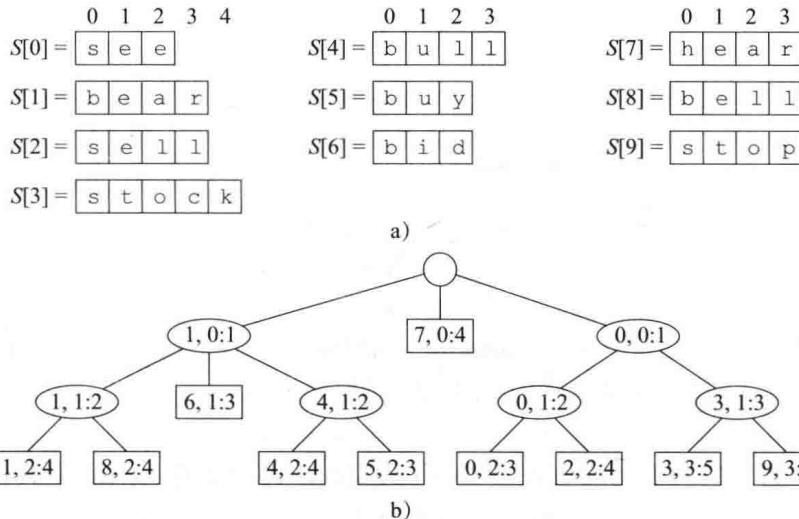


图 13-13 a) 存储在一个数组中的字符串的集合 S ; b) S 的压缩字典树的简单表示

这个附加的压缩方案将空间复杂度从 $O(n)$ 降低到了 $O(s)$ ，其中 n 是 S 中字符串的总长度，并且 s 是 S 中字符串的总数目。我们必须仍旧存储 S 中的不同字符串，当然这并没有降低字典树的空间复杂度。

在压缩字典树中搜索不一定比在标准字典树中更快，因为在字典树中遍历路径时，仍旧需要比较期望模式的每个字符和潜在的多字符标签。

13.5.3 后缀字典树

字典树主要应用于集合 S 中的字符串都是字符串 X 的后缀的情况。这样的字典树称为字符串 X 的后缀字典树（又称为后缀树或者位置树）。例如，图 13-14a 展示了字符串“minimize”8 个后缀的后缀字典树。对于一棵后缀字典树，上一节提出的结构进一步简化。也就是说，每个顶点的标签是一对指示字符串 $X[j:k]$ 的 (j, k) （见图 13-14b）。为了满足 X 的后缀都不是另一个后缀的前缀这一规则，可以增加一个用 $\$$ 表示的特殊字符，它在 X 的最后但不在原始的字母表 Σ 中（同时对每个后缀来说）。也就是说，如果字符串 X 的长度为 n ，则为 n 个字符串 $X[j:n]$ 的集合建立一个字典树，其中 $j = 0, \dots, n-1$ 。

节省空间

后缀字典树允许我们通过使用一些空间压缩技巧（包括为压缩字典树使用的技巧），在一个标准字典树上节省空间。

现在，字典树的简明表示的优势对后缀字典树变得明显。因为长度为 n 的字符串 X 的后缀的总长度为

$$1 + 2 + \dots + n = n(n+1)/2$$

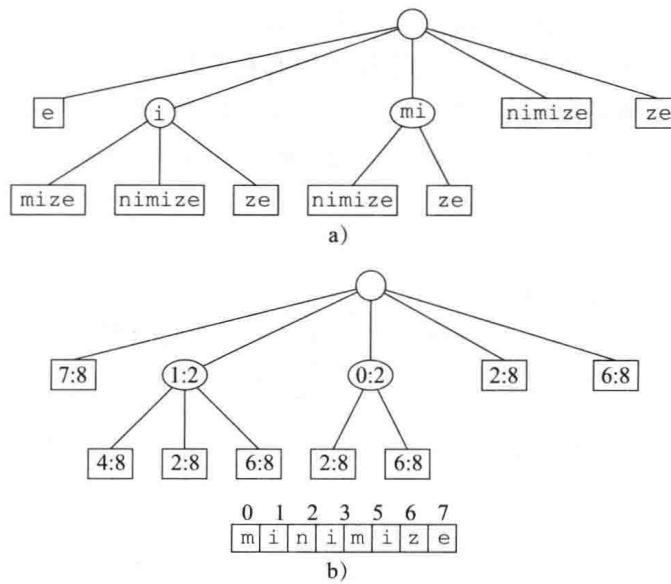


图 13-14 a) 字符串 $X = \text{"minimize"}$ 的后缀字典树 T ；b) T 的简单表示，其中 $j:k$ 对表示引用字符串中的切片 $X[j:k]$

所以显式存储 X 的所有后缀会花费 $O(n^2)$ 的空间。即便如此，后缀字典树也在 $O(n)$ 的空间内隐式地表示了这些字符串，如命题 13-8 所述。

命题 13-8： 长度为 n 的字符串 X 的后缀字典树的简明表示使用了 $O(n)$ 的空间。

构造

可以像 13.5.1 节给出的那样使用递增算法为长度为 n 的字符串建立后缀字典树。这个构造花费了 $O(|\Sigma|n^2)$ 的时间，因为后缀的总长度是 n 的平方。然而，长度为 n 的字符串的后缀字典树可以用不同于一般字典树的递增算法在 $O(n)$ 时间内创建。这个线性时间构造算法非常复杂，这里不做讨论。但是在使用一个后缀字典树去解决其他问题时，仍然可以利用这个快速构造算法的优点去实现。

使用后缀字典树

字符串 X 的后缀字典树 T 可用于在文本 X 上高效地执行模式匹配查询。也就是说，可以通过追踪在 T 中与 P 相关联的路径来确定模式 P 是否是 X 的子串。 P 是 X 的一个子串——当且仅当追踪到这样的路径时。在字典树 T 上执行的搜索假设 T 中的节点存储了一些附加的信息，关于后缀字典树的简明表示有：

如果节点 v 有标签 (j, k) ，并且 Y 是与从根到 v （包括）的路径相关联的长度为 y 的字符串，那么 $X[k - y:k] = Y$ 。

这个特性确保匹配发生时容易计算文本中该模式的开始索引。

13.5.4 搜索引擎索引

万维网是包含了文本文献（网页）的巨大集合。关于这些网页的信息由称为网络爬虫的程序汇聚起来，可以将这些信息存储在一个特别的字典数据库里。网络搜索引擎允许用户从这个数据库里检索相关信息，从而在包含给定关键词的网络中识别相关页面。本节将呈现一个搜索引擎的简易模型。

反序文件

存储核心信息的搜索引擎是字典，叫作反序索引或者反序文件，存储主键 – 值的对 (w, L)，其中 w 是一个单词， L 是包含单词 w 的页面的集合。这个字典中的主键（单词）称为索引词，并且应该成为一个尽可能大的词汇条目和专有名词的集合。这个字典中的元素称为出现列表，并且应该覆盖尽可能多的网页。

可以用包含以下特点的数据结构高效地实现一个反向索引：

- 一个存储元素出现列表的数组（无先后顺序）。
- 索引词集合的压缩字典树的每个叶子节点存储着相关词的发生列表的索引。

之所以在字典树之外存储出现列表，是为了使字典树数据结构的大小保持足够小以适应内存。相反，因为它们总的大小过大，所以出现列表必须存储在硬盘上。

根据这种数据结构，对单个关键词的查询和单词匹配查询类似（13.5.1 节）。也就是说，在字典树中找到了关键词并且返回了关联发生列表。

当多个关键词给出并且期望的输出是包含所有给定关键词的页面时，使用字典树检索每个关键词的出现列表并且返回它们的交集。为了使交集的计算变得容易，允许高效地设置操作，每个发生列表应该用被地址或者地图分类过的序列来实现。

除了返回包含给定关键词的页面列表这一基本任务之外，搜索引擎还有一项重要的附加服务，即通过对按照相关性返回的页面进行排名。对计算机研究者和电子商务公司来说为搜索引擎设计快速而准确的排名算法是一个主要挑战。

13.6 练习

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

巩固

- R-13.1 列出字符串 $P = "aaabbaaa"$ 的前缀和后缀。
- R-13.2 字符串 "cgtacgttgcgtacg" 的最长的（适当的）前缀同时也是该字符串的后缀是什么？
- R-13.3 请画图说明对文本 "aaabaadaabaaa" 和模式 "aabaa" 使用蛮力模式匹配所进行的比较。
- R-13.4 用 Boyer-Moore 算法重复先前的问题，不计算 $\text{last}(c)$ 函数进行的比较计数。
- R-13.5 用 Knuth-Morris-Pratt 算法重复练习 R-13.3，不计算失败函数进行的比较计数。
- R-13.6 在模式字符串中对字符计算代表使用在 Boyer-Moore 算法中的 last 函数的映射：
"the quick brown fox jumped over a lazy cat"。
- R-13.7 对模式字符串 "cgtacgttgcgtac" 计算代表 Knuth-Morris-Pratt 失败函数的表。
- R-13.8 对 10×5 、 5×2 、 2×20 、 20×12 、 12×4 和 4×60 的矩阵链进行乘法的最好的方式是什么？
请阐述具体过程。
- R-13.9 在图 13-8 中，GTTAA 是给定字符串 X 和 Y 的最长的公共子序列。然而，这个答案不是唯一的。给出其他关于 X 和 Y 的长度为 6 的公共子序列。
- R-13.10 给出下面两个字符串的最长公共子序列数组 L ：

$$\begin{aligned} X &= \text{"skullandbones"} \\ Y &= \text{"lullabybabies"} \end{aligned}$$

这两个字符串的最长公共子序列是什么？

- R-13.11 画出下列字符串的频率数组和霍夫曼编码：
"dogs do not spot hot pots or cats".

R-13.12 画出下面字符串集合的标准字典树:

{abab, baba, ccccc, bbaaa, caa, bbaacc, cbcc, cbca}.

R-13.13 画出先前问题中给出的字符串的压缩字典树。

R-13.14 画出下列字符串的前缀字典树的简明表示:

"minimize minime"

创新

C-13.15 描述一个例子: 长度为 n 的文本 T 和长度为 m 的模式, 并且使蛮力模式匹配算法的运行时间达到 $\Omega(nm)$ 。

C-13.16 为了实现一个函数 rfind_brute(T, P), 改编蛮力模式匹配算法, 该函数返回了一个索引, 该索引表示文本 T 中模式 P 最右边的出现 (如果有的话)。

C-13.17 重做上面的练习, 改编 Boyer-Moore 模式匹配算法, 以正确地实现函数 rfind_boyer_moore(T, P)。

C-13.18 重做 C-13.16, 改编 Knuth-Morris-Pratt 模式匹配算法, 以正确地实现函数 rfind_kmp(T, P)。

C-13.19 Python 的 str 类的计数方法报告了一个字符串中一个模式不重叠出现的最大次数。例如, 调用 'abababa'.count('aba') 返回 2 (不是 3)。改编蛮力模式匹配算法以实现函数 count_brute(T, P), 它和示例有一样的结果。

C-13.20 重做上面的练习, 改编 Boyer-Moore 模式匹配算法以实现函数 count_boyer_moore(T, P)。

C-13.21 重做 C-13.19, 改编 Knuth-Morris-Pratt 模式匹配算法以正确地实现函数 count_kmp(T, P)。

C-13.22 给出 compute_kmp_fail 函数 (见代码段 13.4) 对长度为 m 的模式的运行时间为 $O(m)$ 的理由。

C-13.23 设 T 是一个长度为 n 的文本, 设 P 是一个长度为 m 的模式。描述一个寻找 T 的子串 P 的最长前缀并且时间为 $O(n + m)$ 的方法。

C-13.24 如果 P 是 T 的 (正常的) 子串, 或者 P 和 T 的一个后缀和 T 的一个前缀的串联相等, 即有一个索引 $0 \leq k < m$, 使得 $P = T[n - m + k:n] + T[0:k]$, 则长度为 m 的模式 P 是长度 $n > m$ 的文本 T 的循环子串, 给出 $O(n + m)$ 时间的判定 P 是否是 T 的一个循环子串的算法。

C-13.25 Knuth-Morris-Pratt 模式匹配算法可以通过重定义失败函数使得改良后在二进制字符串上运行得更快, 重定义的失败函数为:

$$f(k) = \text{最大的 } j < k, \text{ 使得 } P[0:j] \hat{P}_j \text{ 是 } P[1:k+1] \text{ 的后缀}$$

其中 \hat{P}_j 表示 P 的第 j 个比特的补集。试描述如何修改 KMP 算法以利用这个新函数, 并且同样给出计算这个失败函数的方法。证明这个方法在文本和模式之间至多进行了 n 次比较 (与在 13.2.3 节给出的需要标准 KMP 算法进行 $2n$ 次比较截然相反)。

C-13.26 修改呈现在本章的使用 KMP 算法思想的简化版 Boyer-Moore 算法, 使得其运行时间为 $O(n + m)$ 。

C-13.27 为矩阵链乘法问题设计一个高效的算法, 目标是使用最少的操作, 要求输出一个完整的加上括号的表达式。

C-13.28 澳大利亚人 Anatjari 希望穿越沙漠, 他只拿着一个水壶, 并有一张沿路标记了所有水洞的地图。假设有一壶水的情况下他能走 k 英里, 试设计一个高效的算法, 判定在尽可能少停顿的情况下, Anatjari 应该在哪里重新装满水壶。

C-13.29 为了使用最少的硬币来完成找零, 试描述一个高效的贪心算法。假设有四种硬币的面额 (quarters、dimes、nickels 和 pennies), 它们各自的价值为 25、10、5 和 1。试说明你的算法

正确的理由。

- C-13.30 给出一个硬币面额集合的例子，使得贪心找零算法将无法使用硬币的最小数目。
- C-13.31 在艺术画廊守卫问题中，已经得到在艺术画廊中代表一条长走廊的线 L ，还得到一个在这条长廊进行喷绘的指定位置的真实数字的集合 $X = \{x_0, x_1, \dots, x_{n-1}\}$ ，假设一个守卫可以保护距离他至多为 1 的距离中（两边都可以）的所有喷绘。试设计一个确定守卫位置的算法，其中使用守卫的最小数目去防卫在 X 位置中的所有喷绘。
- C-13.32 设 P 是一个凸多边形， P 的三角剖分是指连接 P 的顶点的对角线的增加，每个内部的面是一个三角形。三角剖分的权重是对角线长度的总和。假设可以计算长度并且在固定的时间内添加和比较它们，给出一个计算 P 的三角剖分的最小权重的高效算法。
- C-13.33 设 T 是一个长度为 n 的文本字符串。试描述一个寻找 T 的最长前缀的方法， T 的最长前缀也是 T 的遍历的子串。
- C-13.34 描述一个找到最长回文结构的高效的算法，最长回文结构是长度为 n 的字符串 T 的后缀。回文结构是一个和它的遍历相等的字符串。这个方法的运行时间是多少？
- C-13.35 已知一个数字序列 $S = (x_0, x_1, \dots, x_{n-1})$ ，描述一个寻找数字的最长序列 $T = (x_{i_0}, x_{i_1}, \dots, x_{i_{k-1}})$ 的时间为 $O(n^2)$ 的算法，其中 $i_j < i_{j+1}$ 且 $x_{i_j} > x_{i_{j+1}}$ ，即 T 是 S 的最长递减子序列。
- C-13.36 试给出一个高效算法，判定模式 P 是否是文本 T 的子序列（不是子串）。该算法的运行时间是多少？
- C-13.37 在长度为 n 的字符串 X 和长度为 m 的字符串 Y 之间定义编辑距离，该距离是使 X 变成 Y 的编辑操作的数目。编辑操作包括字符插入、字符删除和字符置换。例如，字符串 "algorithm" 和 "rhythm" 的编辑距离是 6。为计算 X 和 Y 之间的编辑距离，请设计一个时间为 $O(nm)$ 的算法。
- C-13.38 设 X 是长度为 n 的字符串， Y 是长度为 m 的字符串。设 $B(j, k)$ 是后缀 $X[n-j:n]$ 和后缀 $Y[m-k:m]$ 的最长公共子串的长度。为计算所有 $B(j, k)$ 的值，试设计一个时间为 $O(nm)$ 的算法，其中 $j = 1, \dots, n$, $k = 1, \dots, m$ 。
- C-13.39 Anna 赢得了竞赛，她可以在免费糖果之外多拿 n 块糖果。Anna 已经知道一些糖果较贵，而其他糖果相对便宜。装糖果的瓶子分别被标号为 $0, 1, \dots, m-1$ ，瓶子 j 有 n_j 块糖果，每一块的价格是 c_j 。设计一个时间为 $O(n + m)$ 的算法，该算法允许 Anna 最大化因赢得奖励所拿走糖果的价值。试为 Anna 提供最大化价值的算法。
- C-13.40 定义三个数组 A 、 B 和 C ，每个数组的大小为 n 。已知一个任意的数字 k ，设计一个时间为 $O(n^2 \log n)$ 的算法，判定是否存在这样的数字，即对于 A 中的 a 、 B 中的 b 和 C 中的 c ，有 $k = a + b + c$ 。
- C-13.41 为上面的练习给出一个时间为 $O(n^2)$ 的算法。
- C-13.42 已知长度为 n 的字符串 X 和长度为 m 的字符串 Y ，试描述一个为寻找 X 的最长前缀同时是 Y 的后缀的时间为 $O(n + m)$ 的算法。
- C-13.43 为从一棵标准字典树中删除一个字符串，试给出一个高效的算法并分析它的运行时间。
- C-13.44 为从一棵压缩字典树中删除一个字符串，试给出一个高效的算法并分析它的运行时间。
- C-13.45 为构建一棵后缀字典树的简明表示描述一个算法，已知它的非简明表示，并且分析该算法的运行时间。

项目

- P-13.46 使用 LCS 算法在 DNA 字符串之间计算最好的序列队列，DNA 字符串可以从基因库中在线

查找。

- P-13.47 写一个工程，有两个字符串（例如 DNA 双旋链的表示）并且计算它们的编辑距离，同时显示相应的内容。
- P-13.48 为可变长度的模式执行关于蛮力算法和 KMP 模式匹配算法的效率（执行的字符比较的数目）的实验性分析。
- P-13.49 为可变长度的模式执行关于蛮力算法和 Boyer-Moore 模式匹配算法的效率（执行的字符比较的数目）的实验性分析。
- P-13.50 对蛮力算法、KMP 和 Boyer-Moore 模式匹配算法的相对速度进行实验性分析。在使用可变长度模式搜索的巨大文本文档上记录相对的运行时间。
- P-13.51 针对 Python 的 str 类的 find 方法的效率进行实验，并且构建一个关于它使用的模式匹配算法的假设。试图使用可能对不同的算法同时造成最好情况和最坏情况的输入。
- P-13.52 实现一个基于霍夫曼编码的压缩和解压的方案。
- P-13.53 创建一棵对 ASCII 字符串的集合实现标准字典树的类。这个类应该有将一系列字符串作为一个参数的构造函数，并且这个类应该有测试给定的字符串是否存储在字典树中的方法。
- P-13.54 创建一个对 ASCII 字符串的集合实现压缩字典树的类。这个类应该有将一系列的字符串作为一个参数的构造函数，并且这个类应该有测试给定的字符串是否存储在字典树中的方法。
- P-13.55 创建一个对 ASCII 字符串实现一个前缀字典树的类。这个类应该有将字符串作为一个参数的构造函数，并且这个类应该有在字符串上进行模式匹配的方法。
- P-13.56 为一个小网站的网页实现在 13.5.4 节中描述的简化的搜索引擎。使用在网站页面中的所有单词作为索引词，除了一些诸如文章、介词、名词这样的停词。
- P-13.57 通过对在 13.5.4 节描述的简化的搜索引擎添加一个页面排序特征，对一个小网站的页面实现搜索引擎。页面排序特征应该首先返回最相关的页面。使用网站页面的所有单词作为索引词，除了诸如冠词、介词和代词这样的停顿词。

拓展阅读

KMP 算法是由 Knuth、Morris 和 Pratt 在期刊文章^[66] 中描述的，并且 Boyer 和 Moore 在同年出版的期刊文章中也描述了他们的算法^[18]。在文章中，Knuth 等人^[66] 还证明了 Boyer-Moore 算法以线性时间运行。最近 Cole^[27] 展示了 Boyer-Moore 算法在最坏情况下至多进行 $3n$ 次字符比较，并且这个界限很窄。上述所有讨论的算法同样被 Aho^[4] 讨论了，虽然在更多的理论框架中包括普通表达式的模式匹配方法。对字符串模式匹配的深层学习感兴趣的读者可阅读 Stephen^[90] 的书，还有 Aho^[4]、Crochemore 和 Lecroq^[30] 的书。

动态规划在运筹学的团体中不断发展，并且被 Bellman^[13] 正式化。

字典树由 Morrison^[79] 发明并且在 Knuth^[65] 的《经典排序和搜索》一书中被广泛讨论。“Patricia”是“Practical Algorithm to Retrieve Information Coded in Alphanumeric”^[79] 的简称。McCreight^[73] 展示了如何在线性时间内构建后缀字典树。信息检索领域的介绍，包括对网络的搜索引擎的介绍，在 Baeza-Yates 和 Ribeiro-Neto^[8] 的书刊中有所提及。

图 算 法

14.1 图

图是表示对象之间存在关系的一种方式。即图是对象的一个集合，称为顶点，顶点之间的成对连接称为边。图在许多领域中都有应用，包括绘图、运输、计算机网络和电气工程。顺便说一下，这里的“图”的概念不应该与条形图和函数图混淆，因为这些形形色色的“图”和本章的话题无关。

抽象地看，图 G 仅仅是顶点的集合 V 和 V 中的成对的顶点（称为边）的集合 E 。因此，图是表示一些集合 V 中的成对对象的连接或关系的一种方式。此外，一些书籍中对图形使用不同的术语，如将顶点称为节点，并且将边称为圆弧。我们使用术语“顶点”和“边”。

在图中边被定义为有向或者无向的。如果顶点对 (u, v) 是有序的， u 在 v 之前，可以说边 (u, v) 从顶点 u 到顶点 v 是有向的。如果顶点对 (u, v) 是无序的，可以说边 (u, v) 是无向的。无向边有时候用数学符号表示为 $\{u, v\}$ ，但是为简单起见，我们使用顶点对符号 (u, v) ，注意无向情况下 (u, v) 和 (v, u) 是一样的。通常通过将顶点绘制为椭圆形或矩形，并将边作为连接椭圆形和矩形对的线段或曲线来显示图形。下面是有向或者无向图的一些例子。

例题 14-1： 我们可以通过构造一个图来可视化某一学科研究者之间的协作，这些图的顶点与研究者本身相关联，边用于连接与共同研究论文或书的研究人员相关联的顶点对（见图 14-1）。这样的边是无向的，因为合著是一种对称关系；也就是说，如果 A 与 B 合著了某些文献，那么 B 必然与 A 合著了同样的文献。

例题 14-2： 我们将面向对象的程序想象为顶点代表定义在程序中的类，边表示类之间的继承关系的图。如果 v 的类继承 u 的类，就有从顶点 v 到顶点 u 的边。这样的边是有向的，因为继承关系仅仅只有一个方向（即它是不对称的）。

如果图形中的所有边都是无向的，那么我们说这个图是无向图。同样，一个有方向的图也称为有向图，其所有边都是有向的。一个图若同时有有方向的边和无方向的边，则通常称为混合图。注意，无向图或者混合图可以通过将每一个无向边 (u, v) 重置为成对的有向边 (u, v) 和 (v, u) 而转换成有向图。这通常是有用的，但是，为了保持所表示的无向图和混合图，对于这类图有多种应用，如下面的例子。

例题 14-3： 城市地图可以模拟成一个顶点是十字路口或者死角，边是没有交点的街道延伸的曲线图。该图既有无向边对应双行道，也有有向边对应单行道。因此，在这种方式下，城市地图的构造图是混合图。

例题 14-4： 图的物理实例的代表是建筑物的电气布线和管道网络。这种网络可以被建

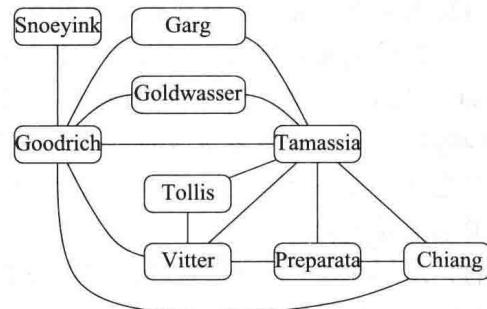


图 14-1 一些作者之间的合著关系图

模成图，其中每一个连接器、钢筋或者出口被看作为顶点，并且每个电线或管道的不间断延伸被看作边。这些图实际上是更大的图形，即当地的电力和供水网络的组成部分。根据在这些图中感兴趣的具体方面，可以考虑它们的边是无向或者有向的，因为，在原则上，水在管道中或电流在导线中可以沿任一方向流动。

由边缘连接的两个顶点被称为边的端部顶点（或端点）。如果边是有向的，它的第一个端点是起点，并且另一个端点是边的终点。两个端点 u 和 v 之间如果有一条边，则称它们是相邻的。如果顶点是边的端点之一，则这条边被称为入射到这个顶点。一个顶点的输出边是起点为该顶点的有向边。输入边是其终点为该顶点的有向边。顶点 v 的度表示为 $\deg(v)$ ，是 v 的入射边的数目。顶点 v 的入度和出度是 v 的输入边和输出边的数目，分别表示为 $\text{indeg}(v)$ 和 $\text{outdeg}(v)$ 。

例题 14-5： 我们可以通过构造一个图 G 来研究航空运输，图 G 称为飞行网络，其顶点和机场相关联，边和航班相关联（见图 14-2）。在图 G 中，边是有向的，因为给出的航班有具体的行驶方向。 G 中每个边 e 的端点分别与 e 对应的航班的出发地和目的地对应。两个机场在 G 中相邻的条件是，有航班在它们之中飞行，并且边 e 入射到图 G 中的顶点 v 的条件是，对应 e 的航班飞向 v 或者是从对应 v 的机场飞来。顶点 v 的输出边对应 v 机场的出站航班。最后， G 的顶点 v 的入度对应 v 机场的进站航班的数目， G 的顶点 v 的出度对应出站航班的数目。

图的定义是指将边作为一个集合（collection 而非 set），从而允许两个无向边具有相同的端点，对于两个有向边可以有相同的起点和终点。这种边称为平行边或者多重边。飞行网络中可以包含平行边（见例题 14-5），这样，同一对顶点之间的多条边可以指示在一天的不同时间的同一路线上运行的不同航班。另一种边的特殊类型是顶点和自己连接。也就是说，如果两个顶点重合，我们称这样的边（无向的或者有向的）为自循环。自循环可能出现在城市地图（见例题 14-3），它可能对应“圆”（一个返回其出发点的环形的街道）。

除了少数例外，图没有平行边和自循环。这类图被认为是简单的。因此，我们通常说简单图的边是一组顶点对（set 而不仅仅是 collection）。在本章中，我们假设图是简单的，除非另有规定。

路径是交替的顶点和边的序列，其开始于一个顶点，结束于一个顶点，使得每条边入射到它的前继顶点和后继顶点。循环是指开始和结束在同一个顶点，并且至少包含一条边的路径。如果路径中的每个顶点都是不同的，我们称这条路径是简单的。如果循环中的每个顶点都是不同的（除去第一个和最后一个顶点），则称这个循环是简单的。有向路径是指所有的边都是有向的并且沿其方向运行的路径。有向循环也是类似的定义。例如，在图 14-2 中，（BOS, NW 35, JFK, AA 1387, DFW）是有向简单路径，而（LAX, UA 120, ORD, UA 877, DFW, AA 49, LAX）是有向简单循环。注意到，有向图可以包含同一对顶点之间两条方向相反的边，例如在图 14-2 中的（ORD, UA 877, DFW, DL 355, ORD）。如果有向图中没有有向循环，则它是非循环的。例如，如果我们在图 14-2 中移除边 UA 877，则剩下

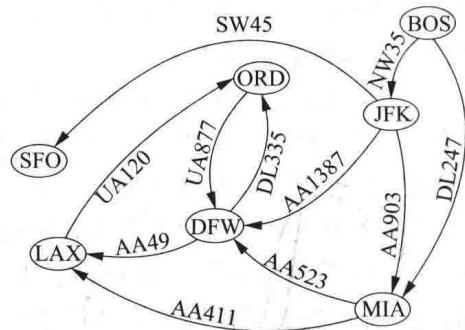


图 14-2 飞行网络的有向图。边 UA 120 的端点是 LAX 和 ORD，因此，LAX 和 ORD 是相邻的。DFW 的入度是 3，DFW 的出度是 2

的图是一个循环。如果图是简单的，在描述路径 P 或者循环 C 时，在 P 是相邻顶点的列表和 C 是相邻顶点的循环的情况下，我们需要省略边，因为这些已定义得很好。

例题 14-6：代表城市地图的图 G 已经给出了（见例题 14-3），我们可以模拟一对夫妇驾车按照 G 中的路径行驶到推荐的餐厅去吃晚饭。如果他们知道路，并且不走入相同的路口两次，那么他们在 G 中行驶了简单路径。因此，我们可以模拟出这对夫妇所需要的全部旅程，从他们的家到餐厅再返回家，作一个循环。如果他们从餐厅回家的路和去餐厅的路完全不同，甚至没有经过相同的路口两次，那么他们的整个行程就是一个简单循环。最后，如果他们在整个行程中沿着单行道进行，我们可以模拟他们的夜晚出行作为一个有向循环。

(有向) 图 G 中已经给出了顶点 u 和 v ，如果 G 中从 u 到 v 有一条(有向)路径，我们称 u 到达 v ，并且 v 是从 u 可达的。在无向图中，可达性的概念是对称的，也就是说，假设 v 可达 u ，则 u 可达 v 。然而，在有向图中，可能 u 可达 v ，但是 v 不可达 u ，因为有向路径必须根据边各自的方向进行遍历。如果一个图是连通的，则意味着对于任何两个顶点，它们中间都是有路径的。如果对于 G 的任何两个顶点 u 和 v ，都有 u 可达 v 并且 v 可达 u ，则有向图 G 是强连通的（见图 14-3 的一些例子。）

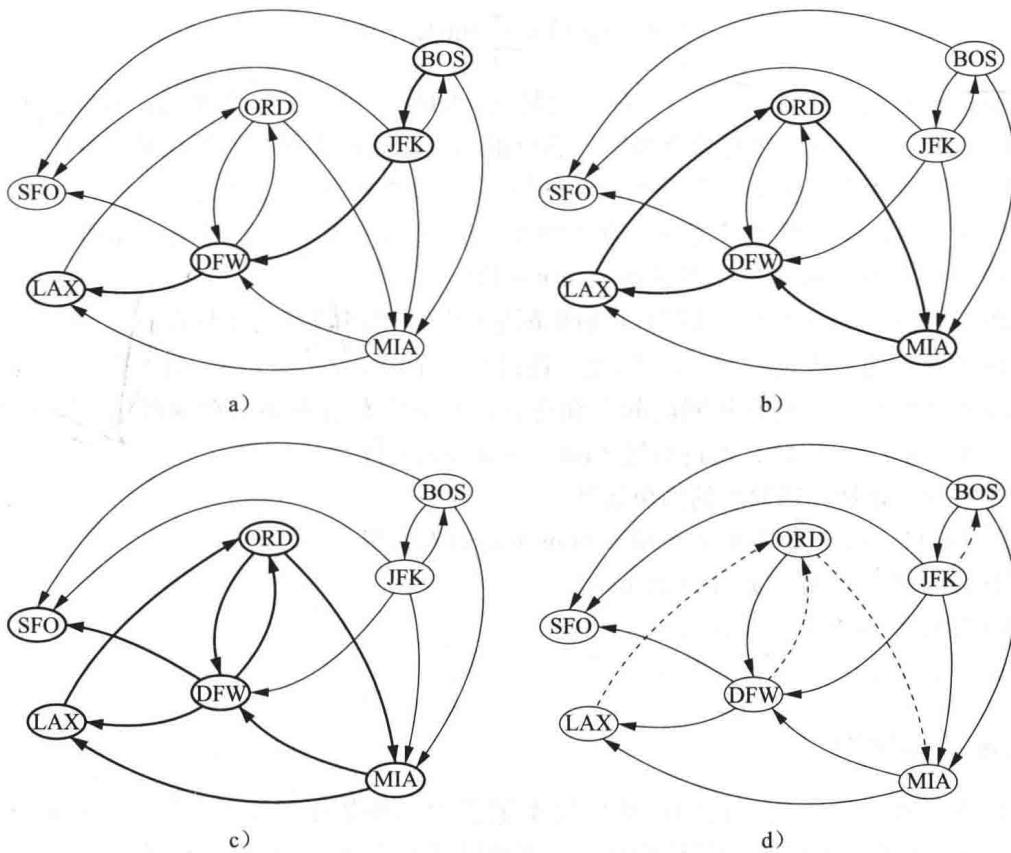


图 14-3 有向图可达的例子：a) 从 BOS 到 LAX 突出显示有向路径；b) 突出显示的有向循环（ORD, MIA, DFW, LAX, ORD），它的顶点可产生强连通子图；c) 突出显示来自 ORD 的顶点和边的子图；d) 虚线边的移除产生了一个循环有向图

图 G 的子图是顶点和边是 G 的顶点和边的各自的子集的图 H 。 G 的生成子图是包含图 G 的所有顶点的图。如果图 G 是不连通的，它的最大连通子图称为 G 的连通分支。森林是

没有循环的图。树是连通的森林，即没有循环的连通图。图的生成树是树的生成子图。(请注意，树的定义和第 8 章给出的树的定义有略微不同，因为不一定是特定的根。)

例题 14-7：也许现在最受关注的图是因特网，它可以被看作顶点是计算机，(无向)边是互联网上一对计算机之间的通信连接的图。计算机及其在域上的联系，就像 wiley.com，形成了因特网的子图。如果这个子图是连通的，那么当两个用户在这个域上使用计算机发送电子邮件给对方时，不需要信息报离开这个域。假设这个子图的边形成了一个生成树。这意味着即使只要一个连接断开(例如，因为有人在该域中将计算机的通信电缆断开了)，那么这个子图将不再连通。

在后续的命题中，我们探索图的几个重要特性。

命题 14-8：如果 G 是有 m 条边和顶点集 V 的图，那么

$$\sum_{v \in V} \deg(v) = 2m$$

证明：一旦通过其端点 u 和通过其端点 v 一次，在上述求和计算中边 (u, v) 就计算了两次。因此，边对顶点度数的总贡献数是边数目的两倍。■

命题 14-9：如果 G 是有 m 条边和顶点集 V 的有向图，那么

$$\sum_{v \in V} \text{in deg}(v) = \sum_{v \in V} \text{out deg}(v) = m$$

证明：在有向图中，边 (u, v) 对它的起点 u 的出度贡献了一个单元，对终点 v 的入度贡献了一个单元。因此，边对顶点出度的总贡献和边的数目相等，入度也是一样的。■

我们接下来展示一个有 n 个顶点， $O(n^2)$ 条边的简单图。

命题 14-10：给定 G 为具有 n 个顶点和 m 条边的简单图。如果 G 是无向的，那么 $m \leq n(n - 1)/2$ ，如果 G 是有向的，那么 $m \leq n(n - 1)$ 。

证明：假设 G 是无向的。因为没有两条边可以有相同的端点并且没有自循环，在这种情况下 G 的顶点的最大度是 $n - 1$ 。因此，通过命题 14-8， $2m \leq n(n - 1)$ 。现在假设 G 是有向的。因为没有两条边具有相同的起点和终点，并且没有自循环，在这种情况下 G 的顶点的最大入度是 $n - 1$ 。因此，通过命题 14-9， $m \leq n(n - 1)$ 。■

有许多树、森林和连通图的简单属性。

命题 14-11：给定 G 是有 n 个顶点和 m 条边的无向图。

- 如果 G 是连通的，那么 $m \geq n - 1$ 。
- 如果 G 是一棵树，那么 $m = n - 1$ 。
- 如果 G 是森林，那么 $m \leq n - 1$ 。

图的抽象数据结构

图是顶点和边的集合。我们将抽象模型定义为三种数据类型的组合：Vertex、Edge 和 Graph。Vertex 是存储由使用者提供的任意元素的轻量级的对象(例如，机场节点)，我们假设它提供一个用来检索所存储元素的方法 element()。Edge 同样存储相关联的对象(例如，航班号、行程距离、费用)，用 element() 方法进行检索。此外，我们假设 Edge 提供以下方法：

- endpoint(): 返回元组 (u, v) ，顶点 u 是边的起点，顶点 v 是终点；对于一个无向图，方向是任意的。
- opposite(): 假设顶点 v 是边的一个端点(起点或者终点)，返回另一个端点。

图的基本抽象表示为 Graph ADT。我们假设一个图是有向或者无向的，定义在构造时指定；回想混合图可以代表一个有向图，构造边 $\{u, v\}$ 作为一对有向边 (u, v) 和 (v, u) 。Graph ADT 包含以下几种方法：

- `vertex_count()`: 返回图的顶点的数目。
- `vertices()`: 迭代返回图中所有顶点。
- `edge_count()`: 返回图的边的数目。
- `edges()`: 迭代返回图的所有边。
- `get_edge(u, v)`: 返回从顶点 u 到顶点 v 的边，如果其中一个存在；否则返回 `None`。
对于无向图，`get_edge(u, v)` 和 `get_edge(v, u)` 之间没有区别。
- `degree(v, out = True)`: 对于一个无向图，返回边入射到顶点 v 的数目。对于一个有向图，返回入射到顶点 v 的输出（或输入）边的数目，由可选参数指定。
- `incident_edges(v, out = True)`: 返回所有边入射到顶点 v 的迭代循环。在有向图的情况下，通过默认报告输出边；如果可选参数设置为 `False`，则报告输入边。
- `insert_vertex(x = None)`: 创建和返回一个新的存储元素 x 的 `Vertex`。
- `insert_edge(u, v, x = None)`: 创建和返回一个新的从顶点 u 到顶点 v 的存储元素 x 的 `Edge`（默认 `None`）。
- `remove_vertex(v)`: 移除顶点 v 和图中它的所有入射边。
- `remove_edge(e)`: 移除图中的边 e 。

14.2 图的数据结构

在本节中，我们介绍四种表示图的数据类型。对于每一种表示，我们维护一个集合去存储图的顶点。然而，这四种表示在它们组织边的方式上有显著不同。

- 在边列表中，我们对所有边采用无序的列表。这个最低限度就足够了，但是还没有有效的方法来找到特定的边 (u, v) ，或者将所有的边入射到顶点 v 。
- 在邻接列表中，我们为每个顶点维护一个单独的列表，包括入射到顶点的那些边。可以通过取较小集合的并集来确定完整的边集合，然而我们可以更高效地找到所有入射到给出顶点的边。
- 邻接图和邻接列表非常相似，但是所有入射到顶点的边的次级容器被组织成一个图，而不是一个列表，用相邻的顶点作为键。这允许在 $O(1)$ 的预期时间内访问特定边 (u, v) 。
- 邻接矩阵通过对有 n 个顶点的图维持一个 $n \times n$ 矩阵来提供最坏的情况下访问特定边 (u, v) 的时间 $O(1)$ 。每一项专用于为顶点 u 和 v 的特定对存储一个参考边 (u, v) ；如果没有这样的边存在，该表项即为空。

这些结构的性能的总结在表 14-1 中给出。我们在本节的剩余部分给出结构的进一步说明。

表 14-1 在本节讨论的图的表示中对 Graph ADT 方法运行时间的总结。令 n 表示顶点的数目， m 表示边的数目， d_v 表示顶点 v 的度。注意邻接矩阵使用 $O(n^2)$ 的空间，而所有其他的结构使用 $O(n + m)$ 的空间

操作	边列表	邻接列表	邻接图	邻接矩阵
<code>vertex_count()</code>	$O(1)$	$O(1)$	$O(1)$	$O(1)$
<code>edge_count()</code>	$O(1)$	$O(1)$	$O(1)$	$O(1)$

(续)

操作	边列表	邻接列表	邻接图	邻接矩阵
vertices()	$O(n)$	$O(n)$	$O(n)$	$O(n)$
edges()	$O(m)$	$O(m)$	$O(m)$	$O(m)$
get_edge(u, v)	$O(m)$	$O(\min(d_u, d_v))$	$O(1)\exp.$	$O(1)$
degree(v)	$O(m)$	$O(1)$	$O(1)$	$O(n)$
incident_edges(v)	$O(m)$	$O(d_v)$	$O(d_v)$	$O(n)$
insert_vertex(x)	$O(1)$	$O(1)$	$O(1)$	$O(n^2)$
remove_vertex(v)	$O(m)$	$O(d_v)$	$O(d_v)$	$O(n^2)$
insert_edge(u, v, x)	$O(1)$	$O(1)$	$O(1)\exp.$	$O(1)$
remove_edge(e)	$O(1)$	$O(1)$	$O(1)\exp.$	$O(1)$

14.2.1 边列表结构

边列表结构作为图 G 的表示方式可能是最简单的，但是却不是最有效的。所有顶点存储在一个无序的列表 V 中，并且所有边对象存储在一个无序的列表 E 中。我们在图 14-4 中举例说明图 G 的边列表结构。

为了支持 Graph ADT (14.1 节) 的许多方法，我们假设边列表代表以下附加特征。集合 V 和 E 用双向链表表示，该双向链表使用第 7 章的 PositonalList 类。

- 顶点对象。存储元素 x 的顶点 v 的顶点对象有以下实例变量：
- 对元素 x 的引用，为了支持 element() 方法。
- 对列表 V 中顶点实例位置的引用，如果 v 从图中移除了，由此允许 v 有效地从 V 中移除。
- 边对象。存储元素 x 的边 e 的边对象有以下实例变量：
- 对元素 x 的引用，为了支持 element() 方法。
- 和 e 的端点相关联的顶点对象的引用，从而允许边实例为方法 endpoints() 和 opposite(v) 提供固定时间支持。
- 列表 E 中边实例的位置的引用，如果 e 在图中被移除了，由此允许 e 更高效地从 E 中移除。

边列表结构的性能

在实现 Graph ADT 过程中边列表结构的性能总结在表 14-2 中。我们首先讨论空间的使用，表示一个有 n 个顶点和 m 条边的图的使用

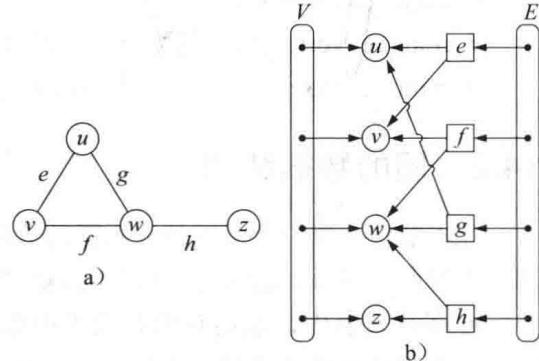


图 14-4 a) 图 G ；b) G 的边列表结构的概要表示。注意到边对象指的是对于它的端点的两个顶点的对象，但是该顶点不指事件边

表 14-2 用边列表结构实现图的表示的运行时间。空间使用是 $O(n + m)$ ，其中 n 是顶点的数目， m 是边的数目

操作	运行时间
vertex_count(), edge_count()	$O(1)$
vertices()	$O(n)$
edges()	$O(m)$
get_edge(u, v), degree(v), incident_edges(v)	$O(m)$
insert_vertex(x), insert_edge(u, v, x), remove_edge(e)	$O(1)$
remove_vertex(v)	$O(m)$

空间是 $O(n + m)$ 。每一个单独的顶点或者边实例使用 $O(1)$ 的空间，附加列表 V 和 E 使用的空间与它们的条目数成比例。

就运行时间而言，边列表结构在报告顶点和边的数目，或者在产生那些顶点或者边的循环中，并不如人们希望的一样好。通过查询相应的列表 V 或者 E ，顶点和边的计算方法的运行时间是 $O(1)$ ，并且在正确的列表中循环，该方法对顶点和边的运行时间分别是 $O(n)$ 和 $O(m)$ 。

边列表结构最显著的局限性，尤其是和其他图形表示方法相比较而言，是 $\text{get_edge}(u, v)$ 、 $\text{degree}(v)$ 和 $\text{incident_edges}(v)$ 方法的运行时间 $O(m)$ 。问题是，图的所有边在无序列表 E 中，能响应那些查询的唯一方法是通过对所有边进行详尽的排查。在本节中介绍的其他数据结构将会更有效地实现这些方法。

最后，我们考虑了一种更新图的方法。在 $O(1)$ 时间内很容易添加一个新的顶点或者一条新的边到图中。例如，通过一个存储给定元素作为数据的 Edge 实例将新边添加到图中，在位置列表 E 中添加那个实例，在 E 中记录它的结果 Position 作为边的属性。之后，存放的位置在 $O(1)$ 时间内可以被用来定位和删除这条边，从而实现方法 $\text{remove_edge}(e)$ 。

讨论为什么 $\text{remove_vertex}(v)$ 方法的运行时间是 $O(m)$ 是值得的。如在 Graph ADT 中所示，当顶点 v 在图中被移除的时候，所有入射到 v 的边同样也必须移除（否则，我们可能会有不是该图的一部分但是指向该顶点的边的矛盾）。为了找到该顶点的入射边，我们必须研究 E 中的所有边。

14.2.2 邻接列表结构

与图的边列表表示方法相比，邻接列表结构将通过将图形的边存储在较小的位置来对其进行分组，从而和每个单独的顶点相关联的次级容器集合起来。具体地，对每个顶点 v 维持一个集合 $I(v)$ ，该集合被称为 v 的入射集合，其中全部都是入射到 v 的边。（在有向图的情况下，输出边和输入边分别存储在两个单独的集合 $\text{lout}(v)$ 和 $\text{lin}(v)$ 中。）传统意义上，顶点 v 的入射集合 $I(v)$ 是一个列表，这就是为什么我们称这种图的表示方法为邻接列表结构。

我们要求邻接列表的基本结构在某种程度上保持顶点集合 V ，因此我们可以在 $O(1)$ 的时间内为给出的顶点 v 找出次级结构 $I(v)$ 。这可以通过使用位置列表来表示 V ，同时每个顶点实例对它的入射集合 $I(v)$ 维持一个有向的引用实现，在图 14-5 中我们说明了这样的一个图的邻接列表结构。如果顶点可以从 0 到 $n - 1$ 进行唯一编号，我们可以代替使用基本的基于数组的结构去访问适当的次级列表。

邻接列表主要的好处是集合 $I(v)$ 正好包含那些应该用 $\text{incident_edges}(v)$ 方法报告的边。因此，我们可以通过在 $O(\deg(v))$ 时间内对 $I(v)$ 的边进行迭代来实现这种方法，其中 $\deg(v)$ 是顶点 v 的度。对于任何图的表示方式这是最好的可能的输出，因为有 $\deg(v)$ 的边进行报告。

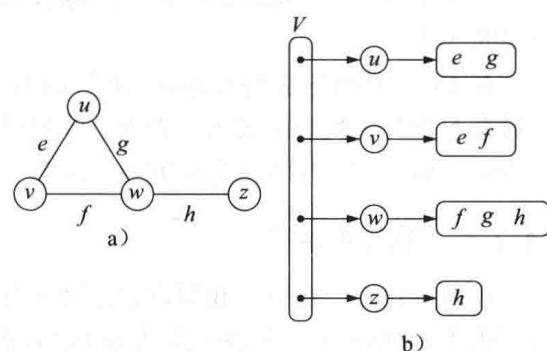


图 14-5 a) 一个无向图 G ；b) G 的邻接列表结构的概要表示。集合 V 是顶点的基本列表，并且每个顶点都有一个入射边的相关联的列表。尽管没有图解，我们推测图的每个边用一个维持其端点的引用的唯一 Edge 实例表示

邻接列表结构的性能

表 14-3 总结了图的邻接列表结构的性能，假设主要集合 V 和所有的次级集合 $I(v)$ 都由双向链表实现。

表 14-3 用邻接列表结构实现图的表示方法的运行时间。空间使用为 $O(n + m)$ ，其中 n 是顶点的数目， m 是边的数目

操作	运行时间
Vertex_count(), edge_count()	$O(1)$
vertices()	$O(n)$
edges()	$O(m)$
get_edge(u, v)	$O(\min(\deg(u), \deg(v)))$
degree(v)	$O(1)$
incident_edges(v)	$O(\deg(v))$
insert_vertex(x), insert_edge(u, v, x)	$O(1)$
remove_edge(e)	$O(1)$
remove_vertex(v)	$O(\deg(v))$

渐近地，邻接列表的所需空间和边列表结构一样，对有 n 个顶点和 m 条边的图需要使用 $O(n + m)$ 的空间。主要的顶点列表使用 $O(n)$ 的空间。所有次级列表长度的总和是 $O(m)$ ，原因在命题 14-8 和命题 14-9 中已经给出了形式化的描述。简而言之，无向边 (u, v) 引用在 $I(u)$ 和 $I(v)$ 中，但是在图中它的存在仅使用了定量的附加空间。

我们已经注意到，incident_edges(v) 方法根据 $I(v)$ 的使用可以实现 $O(\deg(v))$ 的时间。我们可以使用 $O(1)$ 的时间去实现 Graph ADT 的 degree(v) 方法，假设 $I(v)$ 集合可以在类似的时间内报告它的大小。在实现 get_edge(u, v) 中为了找到特定的边，我们可以通过 $I(u)$ 或者 $I(v)$ 搜索。通过选择两个中较小的一个，我们得到 $O(\min(\deg(u), \deg(v)))$ 的运行时间。

表 14-3 中的其余部分可以额外地实现。为了有效地支持边的缺失，edge(u, v) 需要在 $I(u)$ 和 $I(v)$ 之中的位置维持一个引用，因此在 $O(1)$ 时间内，它可以从那些集合中被删除。为了移除一个顶点 v ，我们必须同样移除任何一个人射边，但是至少可以在 $O(\deg(v))$ 时间内找到那些边。

在 $O(m)$ 时间内支持 edges() 和在 $O(1)$ 时间内计算 edges() 最简单的方法是维护边的辅助列表 E 作为边列表的表示。否则，我们可以通过访问每个辅助列表并报告它们的边，从而在 $O(n + m)$ 时间内实现 edges 方法，注意不要报告无向边 (u, v) 两次。

14.2.3 邻接图结构

在邻接列表结构中，我们假设次级入射集合作为无序链表被实现。这样集合 $I(v)$ 的用空间正比于 $O(\deg(v))$ ，允许一条边在 $O(1)$ 时间内被添加或者被移除，并且在 $O(\deg(v))$ 的时间内允许所有入射到顶点 v 的边的迭代。然而，get_edge(u, v) 最好的实现需要 $O(\min(\deg(u), \deg(v)))$ 的时间，因为我们必须在 $I(u)$ 或者 $I(v)$ 中搜寻。

我们可以使用基于哈希的映射为每个顶点 v 实现 $I(v)$ 来提高性能。具体而言，我们让每个人射边的相反的端点作为图的主键，用边结构作为值。我们称这种图的表示方法为邻接图（见图 14-6）。邻接图的空间使用保持为 $O(n + m)$ ，因为 $I(v)$ 对每个顶点 v 使用 $O(\deg(v))$ 的空间，和邻接列表一样。

相对于邻接列表，邻接图的优势是方法 `get_edge(u, v)` 可以通过将顶点 u 作为关键字在 $l(v)$ 中搜索以达到在预期时间 $O(1)$ 内实现。这为邻接列表提供了可能的改善，同时保持在 $O(\min(\deg(u), \deg(v)))$ 的最坏情况的范围内。

在比较邻接图的性能和其他表达方式的性能的过程中（见表 14-1），我们发现它本质上对所有方法实现了最佳的运行时间，成为图表示中一种优秀的通用选择。

14.2.4 邻接矩阵结构

图 G 的邻接矩阵结构对边列表结构增加了一个矩阵 A （即一个二维数组，节 5.6 节），这允许我们在最坏情况的固定时间内找出一对给定顶点之间的边。在邻接矩阵表示方式中，我们考虑顶点以集合 $\{0, 1, \dots, n - 1\}$ 中的数字来表示，边以这些数字中的其中一对来表示。这允许在二维数组 A 的单元格内存储边的引用。特别地，单元格 $A[i, j]$ 存储边 (u, v) 的引用（如果它存在的话），其中 u 是索引为 i 的顶点， v 是索引为 j 的顶点。如果没有这样的边，那么 $A[i, j] = \text{None}$ 。我们需要注意到，如果图 G 是无向的，则数组 A 是对称的，例如对所有的一对 i 和 j 来说， $A[i, j] = A[j, i]$ （见图 14-7）。

邻接矩阵最显著的优点是任何边 (u, v) 可以在最坏情况下 $O(1)$ 时间内被访问到，而邻接图支持在 $O(1)$ 的预期时间内操作。然而，用邻接矩阵进行的一些操作效率很低。例如，为了找到入射到顶点 v 的边，我们必须大概检测与 v 相关联的行的所有 n 个条目，而邻接列表或者邻接图可以在最佳的 $O(\deg(v))$ 时间内找到这些边。从一个图中添加或者移除顶点是不确定的，因为矩阵必须调整大小。

此外，邻接矩阵 $O(n^2)$ 的使用空间通常远远大于其他表示方法所需的 $O(n + m)$ 空间。虽然，在最坏的情况下，密集图的边的数目将正比于 n^2 ，然而大多数现实世界的图是稀疏的。在这样的情况下，邻接矩阵的使用是低效的。不过，如果图是密集的，邻接矩阵的比例常数将比邻接列表和邻接图的要更小。事实上，如果边没有辅助数据，布尔邻接矩阵可以在每个边的位置使用一个位，则 $A[i, j] = \text{True}$ 当且仅当相关的 (u, v) 是一条边。

14.2.5 Python 实现

在本节中，我们提供了 Graph ADT 的实现方法。我们的实现过程将会支持有向或者无

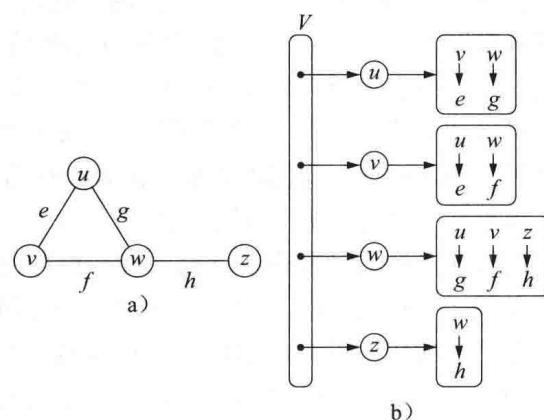


图 14-6 a) 一个无向图 G ；b) G 的邻接图结构的概要表示。每个顶点保持为一个次级图，其中邻接的点作为主键，连接的边作为相关联的值。尽管没有用图表示出来，但我们假设图的每一条边都有一个唯一的 Edge 实例，并且对它的端点维持引用

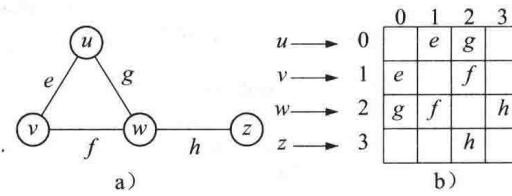


图 14-7 a) 一个无向图 G ；b) G 的邻接辅助矩阵结构的概要表示，其中 n 个顶点映射到索引 0 到 $n - 1$ 。尽管没有用图表表示，我们假设对每一条边有一个唯一的 Edge 实例，并且它对每个端点维持一个引用。我们同样假设有一个次级边列表（没有画出来），对有 m 条边的图，允许 `edges()` 方法在 $O(m)$ 时间内运行

向的图，但是对于解释说明的情况，我们首先在一个无向图的前提下进行描述。

我们使用了邻接图表示方法的变化情况。对于每个顶点 v ，我们用 Python 字典表示次级入射图 $I(v)$ 。然而，我们不能明确地维持列表 V 和 E ，就像在边列表表示方法中的原始描述一样。列表 V 被顶级字典代替，顶级字典 D 将每个顶点 v 映射到其入射图 $I(v)$ 。注意，我们可以通过生成字典 D 的主键的集合来迭代所有的顶点。通过使用这样的字典 D 将顶点映射到次级入射图，我们不需要对作为顶点结构的一部分的入射图维持引用。同样，一个顶点不需要在 D 中对它的位置明确地维持引用，因为它可以在 $O(1)$ 的预期时间内被决定。这很好地简化了我们的实现。然而，我们设计的结果是图 ADT 操作在最坏情况下运行时间的一些界限，并在表 14-1 中给出，这变成了预期的界限。除了维持列表 E ，我们对得到在各种各样的入射图中发现的边的联合很满意。在理论上，它的运行时间为 $O(n + m)$ 而不是严格的 $O(m)$ 时间，因为字典 D 有 n 个主键，即使一些入射图是空的。

Graph ADT 的实现在代码段 14-1 ~ 代码段 14-3 中给出。Vertex 类和 Edge 类在代码段 14-1 中给出，非常简单，并且可以嵌入更多复杂的图类里面。注意我们对 Vertex 和 Edge 都定义了哈希的方法，这样那些实例可以被当作主键用于 Python 的基于哈希的集合和字典里。剩余的 Graph 类在代码段 14-2 和代码段 14-3 中给出。图默认情况下是无向的，但是可以用构造函数里可选择的参数声明为有向的。

代码段 14-1 Vertex 和 Edge 类（嵌套在 Graph 类中）

```

1 #----- nested Vertex class -----
2 class Vertex:
3     """Lightweight vertex structure for a graph."""
4     __slots__ = '_element'
5
6     def __init__(self, x):
7         """Do not call constructor directly. Use Graph's insert_vertex(x)."""
8         self._element = x
9
10    def element(self):
11        """Return element associated with this vertex."""
12        return self._element
13
14    def __hash__(self):          # will allow vertex to be a map/set key
15        return hash(id(self))
16
17 #----- nested Edge class -----
18 class Edge:
19     """Lightweight edge structure for a graph."""
20     __slots__ = '_origin', '_destination', '_element'
21
22     def __init__(self, u, v, x):
23         """Do not call constructor directly. Use Graph's insert_edge(u,v,x)."""
24         self._origin = u
25         self._destination = v
26         self._element = x
27
28     def endpoints(self):
29         """Return (u,v) tuple for vertices u and v."""
30         return (self._origin, self._destination)
31
32     def opposite(self, v):
33         """Return the vertex that is opposite v on this edge."""
34         return self._destination if v is self._origin else self._origin
35

```

```

36 def element(self):
37     """Return element associated with this edge."""
38     return self._element
39
40 def __hash__(self):      # will allow edge to be a map/set key
41     return hash( (self._origin, self._destination) )

```

代码段 14-2 Graph 类的定义 (在代码段 14-3 中继续)

```

1 class Graph:
2     """Representation of a simple graph using an adjacency map."""
3
4     def __init__(self, directed=False):
5         """Create an empty graph (undirected, by default)."""
6
7         Graph is directed if optional parameter is set to True.
8         """
9         self._outgoing = { }
10        # only create second map for directed graph; use alias for undirected
11        self._incoming = { } if directed else self._outgoing
12
13    def is_directed(self):
14        """Return True if this is a directed graph; False if undirected.
15        """
16        Property is based on the original declaration of the graph, not its contents.
17        """
18        return self._incoming is not self._outgoing # directed if maps are distinct
19
20    def vertex_count(self):
21        """Return the number of vertices in the graph."""
22        return len(self._outgoing)
23
24    def vertices(self):
25        """Return an iteration of all vertices of the graph."""
26        return self._outgoing.keys()
27
28    def edge_count(self):
29        """Return the number of edges in the graph."""
30        total = sum(len(self._outgoing[v]) for v in self._outgoing)
31        # for undirected graphs, make sure not to double-count edges
32        return total if self.is_directed() else total // 2
33
34    def edges(self):
35        """Return a set of all edges of the graph."""
36        result = set()      # avoid double-reporting edges of undirected graph
37        for secondary_map in self._outgoing.values():
38            result.update(secondary_map.values())      # add edges to resulting set
39        return result

```

代码段 14-3 Graph 类的定义 (上接代码段 14-2)。我们为了简便起见省略了参数的错误检查

```

40    def get_edge(self, u, v):
41        """Return the edge from u to v, or None if not adjacent."""
42        return self._outgoing[u].get(v)      # returns None if v not adjacent
43
44    def degree(self, v, outgoing=True):
45        """Return number of (outgoing) edges incident to vertex v in the graph.
46        """
47        If graph is directed, optional parameter used to count incoming edges.
48        """
49        adj = self._outgoing if outgoing else self._incoming

```

```

50     return len(adj[v])
51
52 def incident_edges(self, v, outgoing=True):
53     """Return all (outgoing) edges incident to vertex v in the graph.
54
55     If graph is directed, optional parameter used to request incoming edges.
56     """
57     adj = self._outgoing if outgoing else self._incoming
58     for edge in adj[v].values():
59         yield edge
60
61 def insert_vertex(self, x=None):
62     """Insert and return a new Vertex with element x."""
63     v = self.Vertex(x)
64     self._outgoing[v] = { }
65     if self.is_directed():
66         self._incoming[v] = { }      # need distinct map for incoming edges
67     return v
68
69 def insert_edge(self, u, v, x=None):
70     """Insert and return a new Edge from u to v with auxiliary element x."""
71     e = self.Edge(u, v, x)
72     self._outgoing[u][v] = e
73     self._incoming[v][u] = e

```

在内部，我们通过有两个最高层级的字典实例 `_outgoing` 和 `_incoming` 来管理有向的情况，以便 `_outgoing[v]` 映射到代表 $I_{\text{out}}(v)$ 的另一个字典，`_incoming[v]` 映射到 $I_{\text{in}}(v)$ 的表示。为了统一对有向图和无向图进行处理，我们继续在无向的情况下使用 `_outgoing` 和 `_incoming` 标识，作为同一字典的别名。为了方便，我们定义一个通用名 `is_directed`，允许我们在这两种情况下进行分辨。

对于方法 `degree` 和 `incident_edges`，每个都接收一个可选参数在输出和传入方向进行区分，我们在进程开始前选择适当的图。对于方法 `insert_vertex`，我们对每个新的顶点 `v` 的空字典的 `_outgoing[v]` 进行初始化。对于无向的情况，这个步骤并不是必要的，因为 `_outgoing` 和 `_incoming` 是别名。我们将方法 `remove_vertex` 和 `remove_edge` 的实现过程作为练习 C-14.37 和 C-14.38。

14.3 图遍历

希腊神话讲述了一个为了安置一部分是牛一部分是人的巨大人身牛头怪物而精心制作迷宫的故事。这个迷宫非常复杂，以至于没有野兽或者人能逃离它。希腊英雄提修斯在国王女儿阿丽雅德妮的帮助下，决定去实现图的遍历算法。提修斯将捏成团的线固定在迷宫的门上，然后当他为了寻找怪兽而穿过扭曲的通道时解开它。提修斯明确地知道什么是好算法，因为，当他寻找到并战胜野兽之后，提修斯可以轻松地跟随这条线走出迷宫并返回阿丽雅德妮身边。

形式上，遍历是通过检查所有的边和顶点来探索图的系统化的步骤。如果遍历访问的所有顶点和边与它们的数目成正比，即在线性的时间内，则遍历是高效的。

图的遍历算法是回答许多涉及可达性概念的有关于图的问题的关键，即跟随图的路径时，决定如何从一个顶点到达另一个顶点。在无向图中处理可达性的有趣问题包括以下几个方面：

- 计算从顶点 u 到顶点 v 的路径，或者报告这样的路径存在。
- 已知 G 的开始顶点 s ，对每个 G 的顶点 v 计算在 s 和 v 之间的边的最小数目的路径，或者报告没有这样的路径存在。
- 测试是否 G 是连通的。
- 如果 G 是连通的，计算 G 的生成树。
- 计算 G 的连通分支。
- 计算 G 中的循环，或者报告 G 没有循环。

解决有向图 G 的可达性的有趣问题主要包括以下几个方面：

- 计算从顶点 u 到顶点 v 的有向路径，或者报告没有这样的路径存在。
- 找出 G 中从已知顶点 s 可达的顶点。
- 判定 G 是否是非循环的。
- 判定 G 是否是强连通的。

本节剩余的部分，我们展示了两种图的遍历算法，分别叫作深度优先搜索和广度优先搜索。

14.3.1 深度优先搜索

在本节我们考虑第一个遍历算法深度优先搜索（DFS）。深度优先搜索对测试图的性能，包括是否从一个顶点到另一个顶点有路径和是否该图是一个连通图是非常有用的。

图 G 的深度优先搜索和拿着一条绳子和一罐涂料在不迷路的情况下在迷宫中漫步很类似。我们以 G 中的特殊的开始顶点 s 开始，通过将绳子的一端固定到 s 并且喷涂 s 作为“访问”来初始化。顶点 s 是我们现在的“当前”顶点——称为当前顶点 u 。那么我们通过考虑一条入射到当前顶点 u 的（任意的）边 (u, v) 来遍历 G 。如果边 (u, v) 指引我们到已经访问过（即被喷绘过）的顶点 v ，则忽略这条边。相反，如果 (u, v) 指向一个没有被访问过的顶点 v ，那么我们展开绳子并走向 v 。然后将 v 喷涂成“被访问过”，并将它变成当前顶点，重复上面的计算。最终，我们将会到达“尽头”，即对于当前顶点 v ，所有入射到 v 的边指向的顶点都已经访问过了。为了摆脱这种僵局，我们将绳子卷起来，沿着带我们去 v 的边原路返回，直到先前访问过的顶点 u 。然后我们将 u 作为当前顶点，并且对还没有考虑过的所有入射到 u 的边重复上述计算。如果 u 的所有入射边都指向已经被访问过的顶点，那么我们再次卷起绳子，然后返回到从那个顶点来且去向 u 的顶点，并对那个顶点进行重复的步骤。因此，我们沿着迄今为止追踪到的路径返回，直到找到还没有被探索到的边的顶点，找到一条这样的边后将继续遍历。当回溯过程指引我们返回到开始顶点 s ，并且没有未被探索的入射到 s 的边的时候，这个过程就结束了。

以顶点 u 开始的深度优先搜索遍历的伪代码（见代码段 14-4）遵循绳子和涂料的类比。我们用递归来实现字符串的类比，并且假设有原理（的类比）来判定是否一个顶点或者一条边是先前探索过的。

代码段 14-4 DFS 算法

Algorithm DFS(G, u): {We assume u has already been marked as visited}

Input: A graph G and a vertex u of G

Output: A collection of vertices reachable from u , with their discovery edges

```

for each outgoing edge  $e = (u, v)$  of  $u$  do
  if vertex  $v$  has not been visited then
    Mark vertex  $v$  as visited (via edge  $e$ ).
    Recursively call DFS( $G, v$ ).
  
```

用DFS对图的边进行分类

深度优先搜索的执行可以用来分析图的结构，这依赖于在遍历的过程中被探索的边的路径。DFS进程很自然地识别出以开始顶点 s 作为根的深度优先搜索树。无论任何时候，边 $e = (u, v)$ 用于发现代码段 14-4 中的新顶点 v ，那条边叫作发现边或者树的边，以从 u 到 v 为方向。所有 DFS 实现的过程中被考虑的其他边则称为非树的边，可以带我们到先前访问过的点。在无向图的情况下，我们会发现所有被探索的非树的边连通了当前顶点和 DFS 树中它的祖先。我们将这样的边称为 back 边。当对一个有向图执行 DFS 时，会有三种可能的非树边：

- back 边，连通了顶点和 DFS 树的祖先。
- forward 边，连通了顶点和 DFS 树的孩子。
- cross 边，连通了顶点和另一个既不是其祖先也不是其孩子的顶点。

有向图的 DFS 算法的例子说明展示在图 14-8 中，演示了非树边的每一种类型。无向图的 DFS 算法的例子说明展示在图 14-9 中。

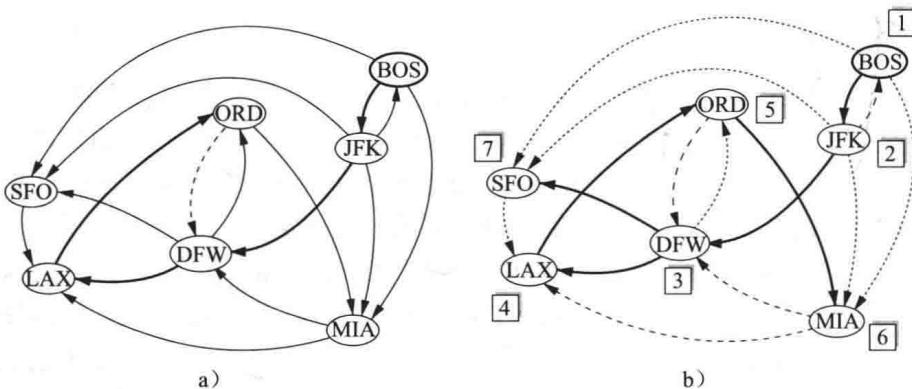


图 14-8 有向图的 DFS 的例子，以顶点 (BOS) 开始：a) 中间步骤，首先，考虑一条边指向一个已经访问过的顶点 (DFW)；b) 完全的 DFS。树的边用粗线表示，back 边用虚线表示，forward 和 cross 边用带点的线表示。每个顶点的访问顺序由每个顶点旁边的标签指示。边 (ORD, DFW) 是 back 边，但是 (DFW, ORD) 是 forward 边。边 (BOS, SFO) 是 forward 边，(SFO, LAX) 是 cross 边

深度优先搜索的特性

我们对深度优先搜索算法做了大量的研究，许多研究都是通过将图 G 的边划分为组的方式来获得的。我们从最重要的特性开始。

命题 14-12：定义 G 为以顶点 s 为开始的 DFS 遍历已经执行过的无向图。那么这个遍历在 s 的连通分支中访问了所有的顶点，并且发现边生成了一个 s 的连通分支的生成树。

证明：假设 s 的连通分支中至少有一个顶点 w 没有被访问， v 是从 s 到 w 的一些路径中第一个没有被访问的顶点（有 $v = w$ ）。因为 v 是这条路径的第一个未被访问的顶点，它有一个没有被访问的邻居 u 。但是当我们访问 u 时，必须考虑边 (u, v) ；因此， v 是未被访问的可能是不正确的。因此，在 s 的连通分支中没有未被访问的顶点。

因为仅仅当我们走向一个未被访问的顶点的时候才能跟随发现边，所以永远不会形成这样的边的循环。因此，发现边形成了一个没有循环的连通子图，是一棵树。此外，这是一个生成树，因为我们已经知道，深度优先搜索在 s 的连通分支中访问了每个顶点。■

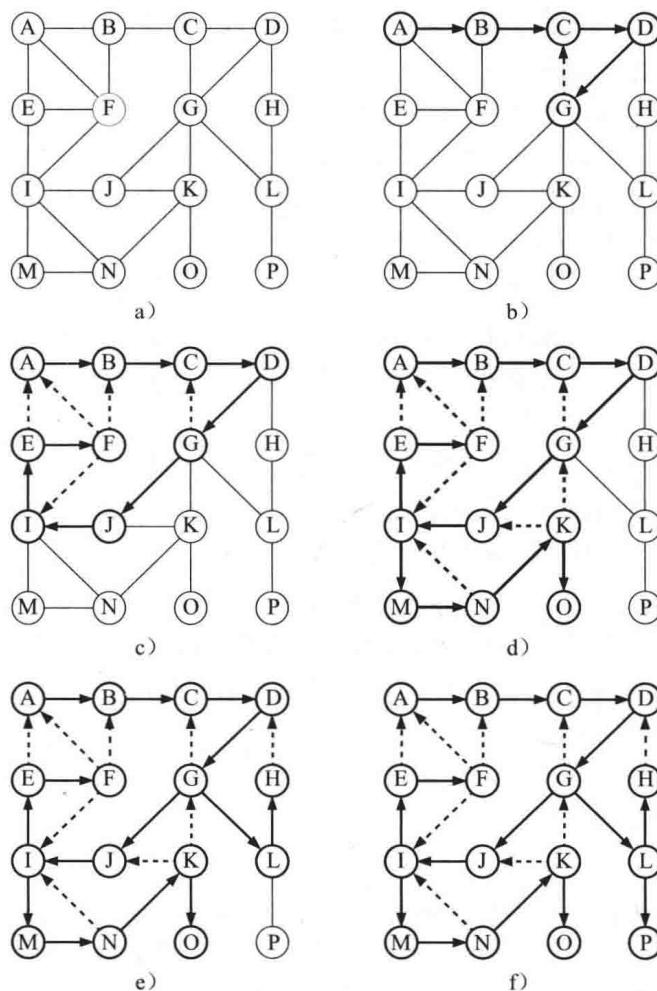


图 14-9 以顶点 A 开始的无向图的深度优先搜索的例子。我们假设顶点的邻接以字母的顺序考虑。被访问过的顶点和探索过的边突出显示，发现边用实线表示，非树（back）边用虚线表示：
 a) 输入图；b) 从 A 开始追踪直到 back 边 (G, C) 被检查到的树的边的路径；c) 到达尽头 F；
 d) 返回到 I 之后，重新恢复边 (I, M) ，在 O 碰上另一个尽头；e) 返回到 G 之后，以边 (G, L) 继续，然后在 H 碰上另一个尽头；f) 最后的结果

命题 14-13： \vec{G} 是一个有向图。以顶点 s 开始的在 \vec{G} 上的深度优先搜索访问了所有的从 s 可达的 \vec{G} 的顶点。同样，DFS 树包含了从 s 到每个可达 s 的顶点的有向路径。

证明： V_s 是以顶点 s 开始的 DFS 中被访问的 G 的顶点的子集。我们想展示 V_s 包含了 s 和每个属于 V_s 的从 s 可达的顶点。现在假设（为了反驳），有一个从 s 可达的顶点 w 不在 V_s 中。考虑从 s 到 w 的一条有向路径，并且令 (u, v) 是带我们从 V_s 中出来的路径的第一条边，即 u 是 V_s 中的，但是 v 不在 V_s 中。当 DFS 到达 u 时，它探索了 u 的所有输出边，因此必须通过边 (u, v) 到达顶点 v 。因此， v 应该在 V_s 中，所以我们得到了矛盾。因此， V_s 必须包含每一个从 s 可达的顶点。

我们通过算法步骤的归纳证明了第二个事实。我们声明发现边 (u, v) 都是可识别的，在 DFS 树中从 s 到 v 存在有向路径。因为 u 必须在先前被发现，从 s 到 u 存在一条路径，因此通过将边 (u, v) 附加到该路径，我们有从 s 到 v 的有向路径。 ■

需要注意的是，因为 back 边总是连通顶点 v 和先前访问的顶点的 u ，所以每个 back 边

暗示了 G 中的一个由 u 到 v 的发现边加上 back 边 (u, v) 构成的循环。

深度优先搜索的运行时间

就运行时间而言，深度优先搜索是遍历图的高效方法。需要注意的是，对每一个顶点 DFS 至多被调用一次（因为它会被标记为被访问过），并且因此对一个无向图来说每条边至多被检查两次，一次来自它的每个端点，并且从它的原始顶点开始，一个有向图至多有一次。如果我们让 $n_s \leq n$ 作为顶点 s 的可达顶点的数目， $m_s \leq m$ 作为这些顶点的入射边的数目，从 s 开始的 DFS 的运行时间为 $O(n_s + m_s)$ ，给出以下满足的条件：

- 用数据结构表示图，通过花费 $O(\deg(v))$ 时间的 `incident_edges(v)` 来创建和迭代，并且 `e.opposite(v)` 方法花费 $O(1)$ 时间。邻接列表结构就是这样的结构，但是邻接矩阵结构不是这样的。
- 我们有一种方法来“标记”被探查的顶点或边，并且测试在 $O(1)$ 时间是否已经探索了顶点的边。我们在下一节讨论实现这个目标的 DFS 实现方式。

已知上面的假设，我们可以解决很多有意思的问题。

命题 14-14：定义 \vec{G} 是一个有 n 个顶点和 m 条边的无向图。 \vec{G} 的 DFS 遍历可以在 $O(n + m)$ 时间内执行完，并且可以在 $O(n + m)$ 时间内被用来解决下面的问题：

- 计算 G 的两个已知顶点之间的路径（如果有一条存在的话）。
- 测试 G 是否是连通的。
- 计算 G 的生成树（如果 G 是连通的）。
- 计算 G 的连通分支。
- 计算 G 的循环，或者报告 G 没有循环。

命题 14-15：定义 \vec{G} 是一个有 n 个顶点和 m 条边的有向图。 \vec{G} 的 DFS 遍历可以在 $O(n + m)$ 的时间内执行完，并且可以在 $O(n + m)$ 的时间内用来解决以下问题：

- 计算 \vec{G} 的两个已知顶点之间的有向路径（如果有存在的话）。
- 计算从已知的顶点 s 可达的 \vec{G} 的顶点的集合。
- 测试 \vec{G} 是否是强连通的。
- 计算 \vec{G} 的有向循环，或者报告 \vec{G} 是非循环的。
- 计算 \vec{G} 的传递闭包（见 14.4 节）。

命题 14-14 和命题 14-15 的证明依赖于将 DFS 算法稍微修改的版本作为子程序的算法。我们将会在本节的剩余部分探索一些扩展。

14.3.2 深度优先搜索的实现和扩展

我们从提供基本深度优先探索算法的 Python 实现开始，伪代码的原始描述在代码段 14-4 中。DFS 功能呈现在代码段 14-5 中。

代码段 14-5 以指定的顶点 v 开始的图的深度优先搜索的递归实现

```

1 def DFS(g, u, discovered):
2     """ Perform DFS of the undiscovered portion of Graph g starting at Vertex u.
3
4     discovered is a dictionary mapping each vertex to the edge that was used to
5     discover it during the DFS. (u should be "discovered" prior to the call.)
6     Newly discovered vertices will be added to the dictionary as a result.
7     """
8     for e in g.incident_edges(u):          # for every outgoing edge from u

```

```

9     v = e.opposite(u)
10    if v not in discovered:           # v is an unvisited vertex
11        discovered[v] = e            # e is the tree edge that discovered v
12        DFS(g, v, discovered)       # recursively explore from v

```

为了追踪哪个顶点被访问过并建立生成 DFS 树的表示，我们的实现引入了叫作 `discovered` 的第三个参数。这个参数应该是 Python 字典，可以将图的顶点映射到用于发现那个顶点的树的边。在此，我们假设源顶点 u 作为字典的主键产生，`None` 作为它的值。因此，调用可以像下面这样开始遍历：

```

result = {u : None}      # a new dictionary, with u trivially discovered
DFS(g, u, result)

```

字典为两个目的服务。内在地，该字典提供了用于识别访问的顶点的机制，因为顶点将会作为主键出现在字典中。外部地，DFS 函数在其继续进行时添加这个字典，因此字典里的值是进程结论中的 DFS 树的边。

因为字典是基于哈希的，测试 `if v not in discovered` 和记录步骤 `discovered[v] = e` 在 $O(1)$ 期望时间内运行，而不是最坏情况下的时间。实际上，这是一个我们愿意接受的妥协，但是它违反了算法的正式的分析。如果我们假设顶点可以用 0 到 $n - 1$ 进行编号，那么这些数字可以用来作为基于数组的查找表的索引而不是基于哈希的映射。或者，我们可以存储每一个顶点的发现状态并且将树的边直接关联成顶点距离的一部分。

从 u 到 v 的路径重建

我们可以使用基本 DFS 函数作为一个工具来鉴定从顶点 u 通往 v 的（有向）路径（如果 v 是从 u 可达的）。这个路径可以很容易地通过遍历期间记录在发现字典里的信息而重建。代码段 14-6 提供了在 u 到 v 的路径中产生的顶点的顺序列表的二级函数的实现。

代码段 14-6 重建 u 到 v 的有向路径的函数，给出了从 u 开始的 DFS 的发现的踪迹。这个函数返回了路径中顶点的顺序列表

```

1 def construct_path(u, v, discovered):
2     path = []                      # empty path by default
3     if v in discovered:
4         # we build list from v to u and then reverse it at the end
5         path.append(v)
6         walk = v
7         while walk is not u:
8             e = discovered[walk]          # find edge leading to walk
9             parent = e.opposite(walk)
10            path.append(parent)
11            walk = parent
12            path.reverse()             # reorient path from u to v
13    return path

```

为了重建这条路径，我们从这条路径的最后开始，检查发现字典以决定哪条边被用来到达顶点 v ，以及那条边的另一个顶点是什么。我们将那个顶点加入一个列表，然后重复这个进程以决定哪条边被用来发现。一旦我们追踪到了返回开始顶点 u 的所有的路，就可以颠倒列表使得它从 u 到 v 被正确地调整，然后将它返回给调用者。这个进程的花费时间和路径的长度成正比，因此它在 $O(n)$ 的时间内运行（最初还有调用 DFS 花费的时间）。

连通性的测试

我们可以用基本 DFS 函数去判定图是否是连通的。在无向图的情况下，我们在任意的

顶点简单地开始深度优先搜索，然后测试是否 $\text{len}(\text{discovered})$ 和结论中的 n 相等。如果图是连通的，那么根据命题 14-12，所有的顶点都能被发现；相反，如果图是不连通的，那么必须至少有一个顶点 v 不可达 u ，而且将不会被发现。

对于有向图 \vec{G} ，我们可能希望测试它是否是强连通的，即是否对每一对顶点 u 和 v ， u 能到达 v 并且 v 能到达 u 。如果我们从每个顶点对 DFS 开始一个独立的调用，便可以判定是否是这种情况，但是 n 个调用组合运行的时间为 $O(n(n + m))$ 。然而，我们可以比这更快地判定 \vec{G} 是否是强连通的，仅仅需要两次深度优先搜索。

我们通过对以任意顶点 s 开始的有向图 \vec{G} 执行深度优先搜索开始。如果根据这次遍历 \vec{G} 中的任何一个顶点都没有被访问，并且不可达 s ，那么图不是强连通的。如果第一次深度优先搜索访问了 \vec{G} 的每个顶点，然后我们需要检查是否 s 从其他所有顶点都可达。在概念上，我们可以通过复制图 \vec{G} 来完成，但是要在所有边相反的方向上。在反向图中以 s 开始的深度优先搜索将会到达可能到达原始图中的 s 的每个顶点。实际上，比制作一个新的图更好的方法是重新实现一个版本的 DFS 方法，该方法将所有输入边循环到当前顶点，而不是所有的输出边。因为该算法仅仅做了两次 \vec{G} 的 DFS 遍历，所以它的运行时间是 $O(n + m)$ 。

计算所有的连通分支

当图是不连通的时候，我们的下一个目标是识别出无向图的所有连通分支，或者有向图的强连通分支。我们首先讨论无向的情况。

如果 DFS 的初始调用不能到达图的所有顶点，我们可以在那些未被访问的顶点重新开始一个新的 DFS 调用。这种综合 `DFS_complete` 方法的实现在代码段 14-7 中给出。

代码段 14-7 返回全部图的 DFS 森林的高级函数

```

1 def DFS_complete(g):
2     """ Perform DFS for entire graph and return forest as a dictionary.
3
4     Result maps each vertex v to the edge that was used to discover it.
5     (Vertices that are roots of a DFS tree are mapped to None.)
6     """
7     forest = { }
8     for u in g.vertices():
9         if u not in forest:
10            forest[u] = None                    # u will be the root of a tree
11            DFS(g, u, forest)
12
13 return forest

```

尽管 `DFS_complete` 函数对原始 DFS 函数进行了多次调用，但调用 `DFS_complete` 花费的总时间为 $O(n + m)$ 。对于一个无向图，回想我们最初的分析，对以顶点 s 开始的 DFS 的单独调用的运行时间为 $O(n_s + m_s)$ ，其中 n_s 是从 s 可达的顶点的数目， m_s 是这些顶点的入射边的数目。因为每一次 DFS 的调用探索了不同的分支， $n_s + m_s$ 的总和是 $n + m$ 。 $O(n + m)$ 的总界限也可以应用于有向的情况，即使可达顶点的集合不一定是不相交的。然而，因为相同的发现字典对所有的 DFS 调用作为一个参数传递，我们知道 DFS 子程序对每个顶点只调用一次，那么在这个过程中每个输出边仅仅只被探索一次。

`DFS_complete` 函数可以被用来分析无向图的连通分支。发现字典的返回代表整个图的 DFS 森林。我们称之为森林而不是树，因为图可能是不连通的。连通分支的数目可以通过用 `None` 作为发现边（这些是 DFS 树的根）的发现字典中顶点的数目来判定。核心 DFS 方法的

微小的修正被用来在发现顶点时标记该顶点的分支数目（见练习 C-14.44）。

找到一个有向图的强连通分支的情况更复杂。存在在 $O(n + m)$ 时间内计算这些连通分支的方法，使用两次单独的深度优先搜索遍历，但是细节不在本书的范围内。

用 DFS 发现循环

对于无向的和有向的图来说，循环的存在当且仅当和图的 DFS 遍历相关的 back 边存在。很容易发现，如果 back 边存在，通过从祖先的孩子得到 back 边并且跟随树的边返回到祖先，这样便可以说明循环存在。相反，如果图中存在循环，那么必须有和 DFS 相关的 back 边（尽管这里没有证明这个事实）。

在算法上，在无向的情况下探索 back 边是容易的，因为所有的边不是树的边就是 back 边。在有向图的情况下，核心 DFS 实现的细微修改需要正确地将非树的边分类为 back 边。若被探索的有向边指向先前访问过的顶点，我们必须认出该顶点是否是当前顶点的祖先。这需要额外的记录，例如，依据 DFS 的递归调用是否依旧活跃来标记顶点。我们把细节留作练习 C-14.43。

14.3.3 广度优先搜索

如先前部分中所描述的，深度优先搜索的前进和回溯定义了通过物理上的跟踪来探索图的遍历。在本节，我们考虑另一种遍历图的连通分支的算法，叫作广度优先搜索（BFS）。BFS 算法更类似于在所有方向上发送以协调方式共同遍历图的许多探索者。

BFS 以回合的方式进行并且将顶点分成不同级别。BFS 以顶点 s 开始，它的级别是 0。在第一轮标记“被访问过”，对于所有和开始顶点 s 邻近的顶点——这些顶点和开始有一步之远，我们将其置为级别 1。在第二轮，我们允许所有的探索者从开始顶点走两步（也就是边）远。这些新的顶点和级别 1 的顶点邻近但以前没有被设置过级别，现在将其置为级别 2 并且标记为“被访问过”。这个过程以类似的方式继续进行，当在级别中没有新的顶点被找到时进程结束。

BFS 的 Python 实现在代码段 14-8 中给出。我们遵守和 DFS（代码段 14-5）类似的约定，使用发现字典去识别被访问过的顶点和记录 BFS 树的发现边。我们在图 14-10 中举例说明了 BFS 遍历。

代码段 14-8 以任意顶点 s 开始的图的广度优先搜索的实现

```

1 def BFS(g, s, discovered):
2     """Perform BFS of the undiscovered portion of Graph g starting at Vertex s.
3
4     discovered is a dictionary mapping each vertex to the edge that was used to
5     discover it during the BFS (s should be mapped to None prior to the call).
6     Newly discovered vertices will be added to the dictionary as a result.
7     """
8     level = [s]                      # first level includes only s
9     while len(level) > 0:
10        next_level = []              # prepare to gather newly found vertices
11        for u in level:
12            for e in g.incident_edges(u): # for every outgoing edge from u
13                v = e.opposite(u)
14                if v not in discovered:   # v is an unvisited vertex
15                    discovered[v] = e    # e is the tree edge that discovered v
16                    next_level.append(v) # v will be further considered in next pass
17        level = next_level          # relabel 'next' level to become current

```

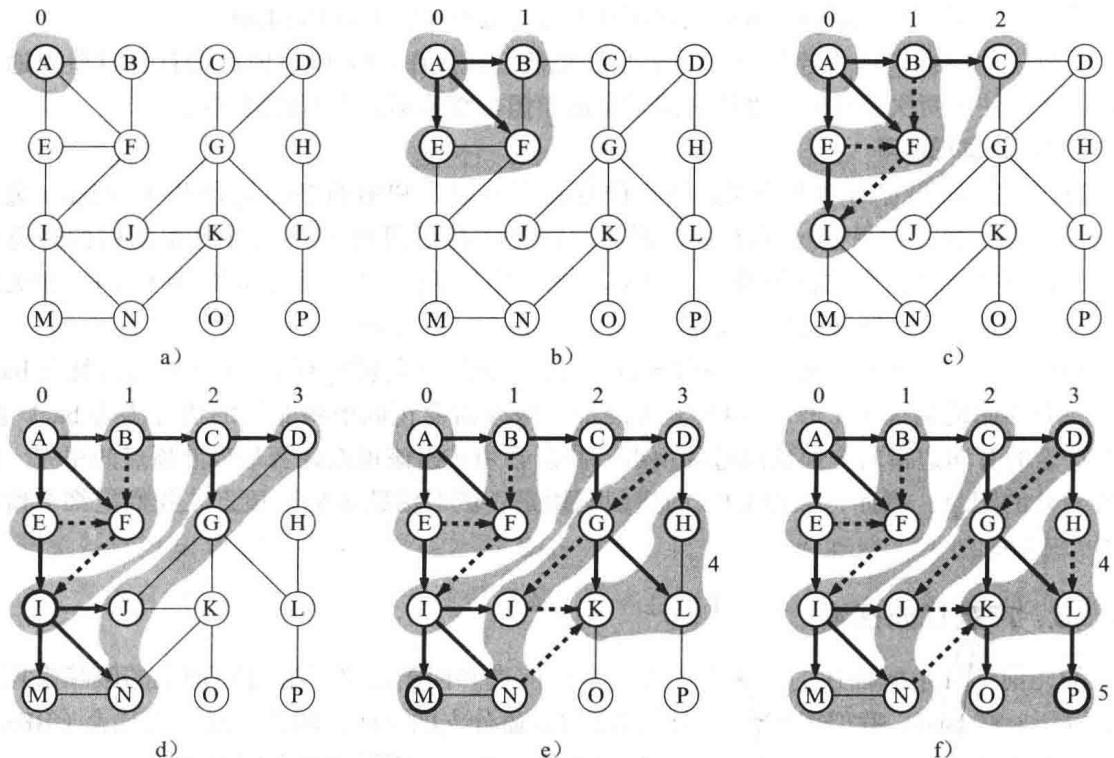


图 14-10 广度优先搜索遍历的例子，其中以相邻顶点的字母顺序考虑入射到顶点的边。发现边用实线表示，非树的（cross）边用虚线表示：a) 从 A 开始搜索；b) 发现级别 1；c) 发现级别 2；d) 发现级别 3；e) 发现级别 4；f) 发现级别 5

讨论 DFS 的时候，我们将非树的边的分类描述为 back 边、（连通一个顶点和它的一个祖先）、forward 边（连通另一个顶点和它的一个祖先）或者 cross 边（连通一个顶点到另一个顶点或它的祖先或它的孩子）。对于无向图的 BFS，所有非树的边都是 cross 边（见练习 C-14.47）。对于有向图的 BFS，所有非树的边都是 back 边或者 cross 边（见练习 C-14.48）。

BFS 遍历算法有大量的有趣的特性，我们在接下来的命题中会进一步探索。最显而易见的是，以顶点 s 为根的到其他任何顶点 v 的广度优先搜索树的路径就边的数目而言保证是从 s 到 v 的最短路径。

命题 14-16： 定义 G 是一个无向或者有向图，在 G 上执行了以顶点 s 为开始的 BFS 遍历。那么

- 遍历访问了所有从 s 可达的 G 的顶点。
- 对每个在 i 阶层的顶点 v ，在 s 和 v 之间 BFS 树 T 的路径有 i 条边，并且任何其他的从 s 到 v 的 G 的路径至少有 i 条边。
- 如果 (u, v) 是不在 BFS 树的边，那么 v 的级别数字至多为 1 并且比 u 的级别数字大。我们把这个命题的证明留作练习 C-14.50。

BFS 运行时间的分析和 DFS 的很类似，BFS 运行时间为 $O(n + m)$ ，或者更特别地，如果 n_s 是从顶点 s 可达的顶点的数目， $m_s \leq m$ 是入射到这些顶点的边的数目，则运行时间为 $O(n_s + m_s)$ 。为了探索整个图，这个进程可以在另一个顶点重新开始，和代码段 14-7 的 `DFS_complete` 函数类似。同样，从顶点 s 到顶点 v 的实际路径可以使用代码段 14-6 的 `construct_path` 函数重建。

命题 14-17： 定义 G 是一个有 n 个顶点和 m 条边，用邻接列表结构表示的图。 G 的

BFS 遍历需要 $O(n + m)$ 时间。

尽管代码段 14-8 中 BFS 的实现一层一层地进行，但 BFS 算法同样可以使用单个的 FIFO 队列去代表搜索的当前边来实现。在队列中以源顶点开始，我们重复地从队列的前面移出顶点并在队列的后端插入任何它的未被访问的邻近点（见练习 C-14.51）。

在比较 DFS 和 BFS 的性能的过程中，两个都能很高效地找到从给定源可达的顶点的集合，然后判定到这些顶点的路径。然而，BFS 可保证这些路径尽可能少地使用边。对于无向图，两个算法都能用来测试连通性，识别连通分支或者找出循环。对于有向图来说，DFS 算法更适合一些任务，例如在图中寻找有向循环，或者识别强连通分支。

14.4 传递闭包

我们已经看到图的遍历可以用来回答有向图可达性的基本问题。特别的，如果你对在图中顶点 u 和顶点 v 之间是否有路径很感兴趣，我们可以从 u 开始执行 DFS 或者 BFS 遍历并且观察是否 v 会被发现。如果用一个邻接列表或者邻接图来表示一个图，我们可以在 $O(n + m)$ 的时间内回答 u 和 v 的可达性（见命题 14-15 和命题 14-17）。

在某些应用中，我们可能希望更高效地回答许多可达性的需求，在这种情况下对图预计算一个更高效的表示方式是非常值得的。例如，这个服务的第一步就是计算起点到终点的行驶方向，从而评定终点是否可达。类似的，在网络通信中，我们可能希望能够快速决定从一个特别的点到另一个点之间是否能流通。受此类应用的启发，我们介绍了下面的定义。有向图 \vec{G} 的传递闭包是有向图 \vec{G}^* ，使得 \vec{G}^* 的顶点和 \vec{G} 的顶点一样，并且无论是否从 u 到 v 有一条有向路径（包括 (u, v) 是原始 \vec{G} 的一条边的情况）。

如果一个图用邻接列表或者邻接图表示，我们在 $O(n(n + m))$ 时间内可以通过从每一个开始顶点进行 n 次图的遍历来计算它的传递闭包。例如，从顶点 u 开始的 DFS 可以被用来决定从 u 到所有顶点的可达性，因此在传递闭包中构成了以 u 开始的边的集合。

本节剩余的部分，我们将为计算有向图的传递闭包探索一种替代技术，尤其是当有向图使用支持在 $O(1)$ 时间内的 $\text{get_edge}(u)$ 方法查找的数据结构时（例如，邻接矩阵结构），这种技术特别适合。定义一个有 n 个顶点和 m 条边的有向图 \vec{G} 。在一系列的界限内计算 \vec{G} 的传递闭包。初始化 $\vec{G}_0 = \vec{G}$ 。任意地将 \vec{G} 的顶点编号为 v_1, v_2, \dots, v_n 。然后开始循环计算，从 1 开始循环。在一般的循环 k ，我们用 $\vec{G}_k = \vec{G}_{k-1}$ 开始构建有向图 \vec{G} ，并且如果有向图 \vec{G}_{k-1} 同时包含 (v_i, v_k) 和 (v_k, v_j) 时，向 \vec{G}_k 添加边 (v_i, v_j) 。以这种方式，我们实施的简单规则会呈现在接下来的命题中。

命题 14-18：对于 $i = 1, \dots, n$ ，当且仅当有向图 \vec{G} 从 v_i 到 v_j 有一条有向路径时，有向图 \vec{G}_k 有边 (v_i, v_j) ，其中中间的顶点（如果任何）在集合 $\{v_1, \dots, v_k\}$ 中。特别的， \vec{G}_n 和 \vec{G}^* 相等， \vec{G}^* 是 \vec{G} 的传递闭包。

命题 14-18 为计算 \vec{G} 的依赖于一系列界限的每个 \vec{G}_k 传递闭包提出了一个简单算法。这个算法被称为 Floyd_Warshall 算法，它的伪代码在代码段 14-9 中给出。我们在图 14-11 中说明了 Floyd_Warshall 算法的例子。

代码段 14-9 Floyd_Warshall 算法的伪代码。这个算法通过递增地计算一系列有向图 $\vec{G}_0, \vec{G}_1, \dots, \vec{G}_n$ ，
 $k = 1, \dots, n$ ，来计算 \vec{G} 的传递闭包 \vec{G}^*

Algorithm FloydWarshall(\vec{G}):

Input: A directed graph \vec{G} with n vertices

Output: The transitive closure \vec{G}^* of \vec{G}

```

let  $v_1, v_2, \dots, v_n$  be an arbitrary numbering of the vertices of  $\vec{G}$ 
 $\vec{G}_0 = \vec{G}$ 
for  $k = 1$  to  $n$  do
     $\vec{G}_k = \vec{G}_{k-1}$ 
    for all  $i, j$  in  $\{1, \dots, n\}$  with  $i \neq j$  and  $i, j \neq k$  do
        if both edges  $(v_i, v_k)$  and  $(v_k, v_j)$  are in  $\vec{G}_{k-1}$  then
            add edge  $(v_i, v_j)$  to  $\vec{G}_k$  (if it is not already present)
return  $\vec{G}_n$ 

```

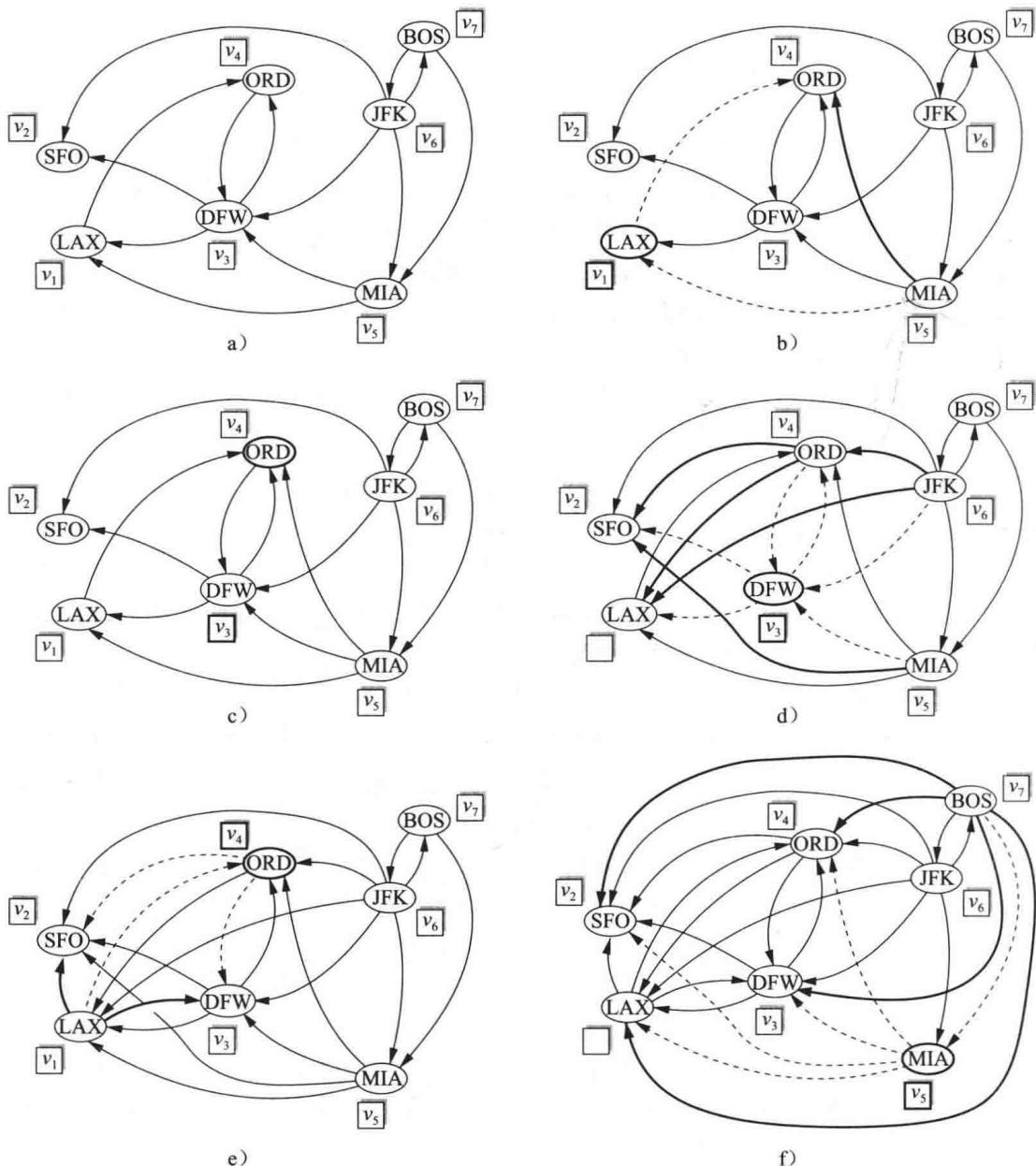


图 14-11 由 Floyd_Warshall 算法计算的有向图的序列：a) 初始化有向图 $\vec{G} = \vec{G}_0$ 和顶点的编号；b) 有向图 \vec{G}_0 ；c) \vec{G}_2 ；d) \vec{G}_3 ；e) \vec{G}_4 ；f) \vec{G}_5 。注意 $\vec{G}_5 = \vec{G}_6 = \vec{G}_7$ 。如果有向图 \vec{G}_{k-1} 有边 (v_i, v_k) 和 (v_k, v_j) ，但是不是边 (v_i, v_j) ，在有向图 \vec{G}_k 的绘制中，我们用虚线展示边 (v_i, v_k) 和 (v_k, v_j) ，而边 (v_i, v_j) 用粗线表示。例如，在 b 中，边 (MIA, LAX) 和 (LAX, ORD) 产生了新的边 (MIA, ORD)

从这个伪代码中，假设表示 G 的数据结构在 $O(1)$ 的时间内完成 `get_edge` 和 `insert_edge` 方法，那么我们可以轻松地分析 Floyd_Warshall 算法的运行时间。主循环执行了 n 次，内部循环考虑每 $O(n^2)$ 对顶点，对每对执行一个固定的时间。因此，Floyd_Warshall 算法的总运行时间为 $O(n^3)$ 。从上述描述和分析中我们可以立刻得出下列命题。

命题 14-19： 定义 \vec{G} 为有 n 个顶点的有向图，并用支持在 $O(1)$ 时间内查找和更新邻接信息的数据结构来表示 \vec{G} 。那么 Floyd_Warshall 算法可以在 $O(n^3)$ 时间内计算 \vec{G} 的传递闭包 \vec{G}^* 。

Floyd_Warshall 算法的性能

渐近地，一旦从每个顶点去计算可达性，Floyd_Warshall 算法 $O(n^3)$ 的运行时间并不比重复地运行 DFS 所实现的好。然而，当图是密集的，或者当图是稀疏的但是用一个邻接矩阵表示的时候，Floyd_Warshall 算法可以匹配重复的 DFS 的渐近边界（见练习 R-14.12）。

Floyd_Warshall 算法的重要性是它比 DFS 更容易实现，并且在实践中非常快，因为它在渐近表示法中隐藏了相对较少的低级操作。这个算法尤其适合邻接矩阵的使用，因为单独的位可以用于指定在传递闭包中可达性模型为方法 `edge(u, v)`。

然而，需要注意的是，当图是稀疏的并且使用邻接列表或者邻接图表示的时候，DFS 的重复响应产生了更好的渐近性能。在这种情况下，一个单独的 DFS 运行时间为 $O(n + m)$ ，因此传递闭包的计算时间为 $O(n^2 + nm)$ ，更好的情况可以达到 $O(n^3)$ 。

Python 实现

我们总结了 Floyd_Warshall 算法的 Python 实现，如代码段 14-10 所示。尽管原始算法用一系列的有向图 $\vec{G}_1, \vec{G}_2, \dots, \vec{G}_n$ ，进行描述，我们对原始图创建了一个单独的副本（使用 Python 副本模块的 `deepcopy` 方法），然后在进行 Floyd_Warshall 算法循环的时候重复地向闭包添加新的边。

代码段 14-10 Floyd_Warshall 算法的 Python 实现

```

1 def floyd_marshall(g):
2     """Return a new graph that is the transitive closure of g."""
3     closure = deepcopy(g)                      # imported from copy module
4     verts = list(closure.vertices())           # make indexable list
5     n = len(verts)
6     for k in range(n):
7         for i in range(n):
8             # verify that edge (i,k) exists in the partial closure
9             if i != k and closure.get_edge(verts[i],verts[k]) is not None:
10                 for j in range(n):
11                     # verify that edge (k,j) exists in the partial closure
12                     if i != j != k and closure.get_edge(verts[k],verts[j]) is not None:
13                         # if (i,j) not yet included, add it to the closure
14                         if closure.get_edge(verts[i],verts[j]) is None:
15                             closure.insert_edge(verts[i],verts[j])
16
return closure

```

这个算法需要对图的顶点进行规范的编号，因此，我们在闭包图中创建了一系列的顶点，然后按照命令对列表添加索引。在最外层的循环中，我们必须考虑所有的 i 和 j 的对。最后，我们仅仅在查实 i 被选择以使得 (v_i, v_k) 存在于闭包的当前的版本之后，通过对所有的 j 的值进行迭代来进行完善和优化。