## 第6章 分解、合并、解决

*分而治之，为至理名言；而合而御之，则更进一步。*

——选自Johann Wolfgang von Goethe[[1]](#footnote-1)的诗集

在接下来的三章中，我们将会为您介绍一些著名的*设计策略*（*design strategy*）。本章介绍的是*分治法*（*divide and conquer*，*D&C*）。这种设计策略致力于通过某种分解问题的方式来改善程序性能：即先将问题实例分解成若干个子问题，然后递归解决这些子问题，并将结果合并，以求最终获得原问题的答案——这也是本章标题中所描述的解题模式。[[2]](#footnote-2)

### 树状问题：即平衡问题

之前，我们曾经提出过一种设计思路：即以子问题为节点、并以它们之间的依赖（或归简）关系为边构建一个子问题图。而对于这种结构来说，树状结构无疑是最简单的一个例子了。因为在该结构中，每个子问题都可能依赖于一个或多个别的子问题，而这些问题本身又都可以被独立解决（到了第8章中，我们还会为您介绍去掉这种独立性之后，接踵而来的各种重叠、纠缠的关系）。这就意味着。只要我们能利用这种相对简单的结构找出某种合适的归简方案，就可以直接套用递归公式来解决问题了。

事实上，我们目前已经掌握了设计出分治类算法所需要的所有要件，它们分别是以下三个概念性思维：

* 第3章中讨论的分治递归式
* 第4章中讨论的强归纳法
* 第5章中讨论的递归遍历法

其中，递归式关注的是算法的性能，归纳法则是我们用来理解算法具体工作方式的一种工具，而递归遍历法（即树上DFS）则是构成这些算法的基本骨架。

现在对我们来说，实现各个归纳步骤的递归形式早已不是新问题了，因为我们早在第4章中就曾为了一些简单排序详细介绍过这类实现的具体过程。但除此之外，*平衡*（*balance*）也是我们在设计分治类算法时要考虑的一个关键问题，而这里我们需要用到强归纳法：因为在这里，我们不是要从*n*-1推导出*n*，而是要从*n*/2推导出*n*。[[3]](#footnote-3) 而且，这里要解决的子问题规模也不再被假定为*n*-1了，我们现在要处理任何规模小于*n*的子问题。

但这与平衡问题有何关系？对于这个问题，还是让我们先来回顾一下弱归纳法的设计思路。在弱归纳法中，我们通常会将问题分解成两个部分：一部分的问题规模为*n*-1，另一部分的规模则为1。那么，如果假设其归纳步骤是个线性级操作的话（这种情况相当常见），我们得出的递归式就应该是*T*(*n*) = *T*(*n*-1) + *T*(1) + *n*。显然，这两个递归调用是极其不平衡的，而且我们最终基本上会获得一个解决握手问题的递归式，结果是个平方级算法。接下来，如果我们想让这两个递归调用的分布更均衡一些的话，应该怎么做呢？也就是说，我们要怎样才能让这两个子问题具有同等规模呢？因为只有在这种情况下，相应递归式才会转变成*T*(*n*) = 2*T*(*n*/2) + *n*。想必您对该递归式也非常熟悉，它就是典型的分治递归式，其运行时间为线性对数级（Θ(*n* lg *n*）——这算得上是一次*飞跃*了。

在下面的图6-1与图6-2中，我们用递归树的形式诠释了这两种方法间的不同之处。您可以看到，它们在节点数量上是相同的——而主要的效应则来自这些节点上的*工作*分配情况。然后就像看魔术师的把戏一样，您能看出我们都把工作分配到哪儿了吗？在这过程中最重要的是要理解，对于相对简单，但会逐步失衡的那个方法来说（图6-1），我们会看到它的许多节点被赋予了高额的工作量。而对于能保持平衡的分治法来说（图6-2），它的大多数节点上的工作量都*非常少*。例如在不平衡的递归操作中，似乎总是有大约*n*/4的调用会至少要耗费整体*n*/2的开销，而在一个平衡的递归操作中，*无论n的值是多少*，同样开销的节点都只会有三个。这就是它们之间的一个显著区别。

（图）

***图6-1.****一次不平衡的分解处理，其分解/合并为线性级操作，整体运行时间为平方级。*

（图）

***图6-2.****分治法：一次平衡的分解处理，其分解/合并为线性级操作、整体运行时间为线性对数级。*

下面，我们来看看这种划分模式的实际运用。*天际线问题*（*skyline problem*）*[[4]](#footnote-4)*在这方面算是一个相对简单的问题。在该问题中，我们需要设置一个有序的三元组序列(*L*,*H*,*R*)：其中，L代表了建筑物左侧的x坐标，H代表的则是建筑物的高度，最后R代表的是建筑物右侧的x坐标。也就是说，该序列中的每个三元组都代表了一个建筑物相对于某个既定原点的（矩形）轮廓。而我们现在的任务就是要根据这些轮廓构造出这些建筑物的天际线。

我们可以用图6-3和图6-4来描述一下这个问题。如图6-4所示，我们在现有的天际线中增加了一栋新的建筑。在这种情况下，如果该天际线的存储形式是一个三元组的水平线序列的话，那么添加新建筑就应该是这样一个线性级操作：先（1）查看其左侧坐标在天际线序列中的位置，然后（2）再提升所有低于该建筑的地方，最后（3）再看其右侧坐标的情况。如果其左右坐标都位于某些水平线序列的中间，我们就会将这些水平线一分为二。

但问题的重点并不完全在合并细节上，“在线性时间内将一个新建筑加入到现有天际线中”这件事本身也是个关键。我们不妨先试试用相对简单些的（弱）归纳法来构建算法：从单一建筑入手，然后持续添加新建筑，直至完成任务。很明显，这必然是一个平方级算法。想要改善这种情况的话，我们就得改用强归纳法——也就是分治法。由于我们注意到合并两个天际线并不会比将一个建筑与一个天际线合并来得更困难，因此只要我们能“同步”遍历它们的两侧，并始终在一者比另一者更高的时候，总是使用那个最大值（这儿需要将其水平线做分割处理）。而且顺着这个思路，我们又迎来了第二轮的算法改进：即为了获取所有建筑的天际线，我们也可以先（递归地）将相关建筑对半分，然后分别计算它们各自的天际线，最后再进行合并。这就得到了一个线性对数级算法（想必您也已经看出来了，在习题6-1中，我们将会让您来亲自实现一下该算法。）

（图） （图）

建筑群 天际线

***图6-3.*** *一组建筑的轮廓及其产生的天际线*

（图）

***图6-4.*** *在现有天际线（实线）上添加新建筑（虚线）*

### 经典分治算法

在上一节中，我们以计算天际线的递归算法为原型，诠释了分治类算法的工作过程。在该算法中，输入是一个元素集合（或元素序列），这些元素将会被对半分成两个规模大致相同的集合（这里最多是线性级操作），然后该算法会在这两半元素持续递归下去，并最终合并出结果（这最多也是线性级操作）。当然，其标准形式是可以有一些修改变化（我们在下一节中就会看到它的一个重要变体），但其中所包含的核心思路基本都是一致的。

在清单6-1中，我们给出了一个通用分治函数的基本架构。尽管，我们可能需要根据自己的算法来实现相应的版本，而不是直接使用这样的通用函数。但这个函数能很好的诠释这类算法的工作过程。当然在这里，我们的设定是函数在遇到基本情况时会直接返回S，这具体得取决于combine函数的工作方式。[[5]](#footnote-5)

***清单6-1.*** *分治语义的一种通用性实现*

def divide\_and\_conquer(S, divide, combine):

if len(S) == 1: return S

L, R = divide(S)

A = divide\_and\_conquer(L, divide, combine)

B = divide\_and\_conquer(R, divide, combine)

return combine(A, B)

另外，我们也用图6-5对该模式做了诠释。该图的上半部分表示的是递归调用的过程，而下半部分表示的是其返回值被合并的过程。对于某些算法（譬如本章稍后会提到的快速排序法）来说，其重点在于图的*上半部分*（即分解部分）。但另一些算法则更着重于图的*下半部分*（即合并部分），这其中最著名的一个例子可能就要数归并排序法了（本章后面也会介绍这一算法），那也是一个典型的分治类算法。

（图）

***图6-5.*** *分治类算法中的分解、递归与合并*

### 折半搜索

在研究如何解决一些符合这种通用模式的更多例子之前，让我们先来看一个*与之相关*的模式。该模式在每次调用中都会放弃一个递归分支。我们早前介绍的二分搜索法（二分法）就是这样的算法。该算法会将问题对半分成两部分，然后*只选一部分*来继续递归调用。其核心原则依然是平衡。请想想，如果这是在一个完全不平衡的搜索操作中，情况会如何呢？想必您还记得第3章的那个“猜粒子”游戏吧？一个不平衡的解决方案就相当于您针对宇宙中的每一种粒子都问一遍：“*这*是您要找的粒子吗？”这其中的区别显然仍如图6-1和6-2所示，只不过（对于该问题而言）每个节点上的工作量是常数，另一点区别是，我们只需要从根节点到某一个叶节点走一遍即可。

当然，二分搜索也有些看起来不是很有趣的地方。这种搜索确实很有效率，但它只能在一个有序序列上执行——这是否就显得它的应用范围很受限制呢？其实不是这样的。首先，该操作本身通常就是其它算法的一个非常重要的组成部分。其次同样重要的是，二分搜索还能被应用于一些更为通用领域的查找行为，例如在某些数值优化思路（譬如牛顿方法）中，或在我们自身代码的调试过程中都存在着二分搜索。不仅“二分法调试”常常被用来提高某些手动式调试的效率（譬如“看看相关代码执行到某个print语句前会不会崩溃”之类的），而且它在许多版本控制系统（revision control system，简称RCS）也有大量运用，例如Mercurial和Git。

其在RCS中的作用是这样的：我们会用RCS持续跟踪代码的变化，并将这些变化存储为多个不同的版本。我们可以“按时间遍历”它们，并将代码恢复到之前的任意时间点的版本。接下来，假设我们现在遇到了一个新bug，想要把它找出来（能理解这种想法吧）。这时候RCS该如何发挥作用呢？首先，我们要先基于测试套件写一个测试——以便找出该bug所在的位置（这在调试中通常都是第一步动作）。此外，我们还得确保RCS能访问到该测试。然后，我们就可以要求RCS查找该bug在历史记录中出现的位置了。但这点又是如何做到的呢？二分搜索！惊喜吧？比如说，如果我们知道该bug位于版本号349到574之间的话，RCS就会先将代码恢复到461版本（即中间位置），然后运行测试代码，看看是否存在该bug。如果存在，那么bug的出现就位于349到461之间，否则就位于462到574之间。然后选择一个分支，继续相同操作。

这个例子不仅简单地示范了二分法的运用，还对另外两点内容做了很好的诠释。首先，该例子说明了我们对一些已知算法的实现不能公式化，即使在算法真的不需要修改时也是如此。譬如在上面的例子中，RCS背后的实现者就很可能需要自己来实现二分搜索操作。其次，这个例子很好地演示了在什么情况下减少关键基本操作的执行次数是件非常重要的事——这比起有效率地实现这些操作来得更重要。毕竟，编译代码与执行测试套件往往都是一些较慢的操作，所以我们总会希望能尽可能地少做几次这样的事情。

|  |
| --- |
| **黑盒子专栏之：bisect** |
| **二分搜索在许多环境中都有应用，但在Python标准库中，“在一个有序序列上搜索某个值”的语义被直接做成了bisect模块。其中就包含了bisect函数，它的作用完全符合我们的预期：**  >>> from bisect import bisect  >>> a = [0, 2, 3, 5, 6, 8, 8, 9]  >>> bisect(a, 5)  4  **好吧，也许不是*那一种*预期……因为它并没有返回现有的那个5的位置，返回了可以插入一个新的5的位置，并确保其位于（那个值相等的）现有项*之后*。而事实上，bisect也只是bisect\_right函数的一个别名，该模块中还有一个bisect\_left函数：**  >>> from bisect import bisect\_left  >>> bisect\_left(a, 5)  3  **如今由于速度原因，bisect模块是用C来实现的，但在早期版本中（Python 2.4之前）它其实是一个纯Python模块，其bisect\_right函数的代码如下（注释是我加的）：**  def bisect\_right(a, x, lo=0, hi=None):  if hi is None: # Searching to the end  hi = len(a)  while lo < hi: # More than one possibility  mid = (lo+hi)//2 # Bisect (find midpoint)  if x < a[mid]: hi = mid # Value < middle? Go left  else: lo = mid+1 # Otherwise: go right  return lo  **如您所见，该实现采用的是迭代操作，但它与递归版实现完全等效。**  **此外，在该模块中还有一对非常实用的函数：即insort（insort\_right函数的别名）与insort\_left。这两个函数也会像bisect函数一样找出相关元素的正确位置，并且会将其插入进去。虽然插入操作本身依然是线性级的，但至少其中的搜索操作是对数级的（其实，插入操作的实际实现也非常高效）。**  **但不幸的是，bisect库中的各种函数都不支持key参数的操作，就像我们在list.sort()函数里所做的那样。我们可以将类似这样的功能称之为“装饰→排序→去除装饰”（其实在这里，应该是“装饰→搜索→去除装饰”）模式，简称DSU（Decorate→Sort→Undecorate）。像这样：**  >>> seq = "I aim to misbehave".split()  >>> dec = sorted((len(x), x) for x in seq)  >>> keys = [k for (k, v) in dec]  >>> vals = [v for (k, v) in dec]  >>> vals[bisect\_left(keys, 3)]  'aim'  **或者，我们还可以再简化一下：**  >>> seq = "I aim to misbehave".split()  >>> dec = sorted((len(x), x) for x in seq)  >>> dec[bisect\_left(dec, (3, ""))][1]  'aim'  **如您所见，这段代码涉及一个装饰列表的新建，这本身是一个线性操作。显然，如果我们在每一轮搜索之前都执行该操作，那么使用bisect模块就没有什么意义了。但是，如果我们在多次搜索过程中维持同一个装饰列表，上面提到的那种模式就会有用。而如果该序列本身一开始就没有被排序过，那么我们就会将DSU模式当作是排序的一部分来处理，就像上面例子中所做的那样。** |

#### 搜索树的遍历……及其剪枝

二分搜索的确是一个非常优秀的算法。虽然它是这里最简单的算法之一，但依然能够独挡一面。不过其中也存在着陷阱：即该算法只能在一组有序的值中执行。现在，如果我们真的能够对这些值维护一个链表，那自然不会有什么问题。而且对于任何我们想要插入的对象来说，我们也只需用二分法找到相应的位置（对数级操作），然后将对象插入即可（常数级操作）。但问题是——这行不通。二分法需要在常数时间内检测序列的中间值，这在链表中是做不到的。而且，即使改用数组形式（譬如Python中的list）也无补于事。因为这种形式仅有助于执行二分法，对插入操作却毫无帮助。

如果我们想要得到一种有利于高效搜索的可修改结构，就需要进行某种折中处理。这里需要的是一种类似于链表（使得插入操作能在常数时间内完成）但又要有利于进行二分搜索的结构。根据本节的标题，相信您已经想到了问题的答案，但请耐心听我说完。首先，由于我们在搜索过程中要能常数时间内访问目标序列内的中间项，所以我们先设有一个指针指向它。而从这个节点开始，我们既可以向左也可以向右移动，但下一个指向的也必须是其左半部分或右半部分的中间项。因此……我们又得在首节点上设置并维护两个指针，一个指向左边，一个指向右边。

换而言之，我们或许只须将二分搜索的操作结构表示成一个明确的树结构就可以了！这种树结构会易于修改，并且从其根节点到某一叶节点的遍历操作能在对数时间内完成。因此我们有时也会说，搜索其实就是我们的老朋友遍历操作——只不过其中还伴随着一些剪枝操作。在这里，我们不必遍历整个树结构（也就是我们所谓的线性扫描）。除非我们根据某有序序列构建出一个树结构，否则我们说“左半部分的中间项”这种表述，对说明问题几乎没有什么帮助。作为替换，我们其实可以从实现剪枝操作的角度来思考一下我们所需要的东西。当查找操作从根节点上开始时，我们所要做的就是将其两个子树中的一个剪掉。（当然了，如果我们在某个内部节点上找到了目标值，而树上又不存在重复值的话，该节点的*两个*子树就可以都被剪掉。）

我们需要的就是所谓的“搜索树属性”：对于某子树的根节点*r*来说，其*左子树*上所有的值都应该*小于*（或等于）*r*的值，而右子树上的值则都要*大于*它。换而言之，就是该根节点上的值*将该子树一分为二*了。例如图6-6中所显示的树结构就具有这样的属性，其中的节点标签代表的是我们所要搜索的值。这种树结构在*set类型*的实现中非常有用（我们可以用它来检测某个值是否存在）。但如果要实现*映射表类型*的话，我们就得为其每个节点都设置一个键/值对（其中，键用于搜索，而值则是我们所要找的目标）。

通常来说，我们并不总是从头构建一棵完整的树（尽管有时会很有用）。我们使用树的动机主要还是来自它们的动态性，这使得我们可以一个一个往上添加节点。如果我们想添加一个节点，就只需找到属于它的位置，然后将其添加为新的叶节点即可。例如在图6-6中，那个树结构的构建过程中可能发生了这样的事：最初加入的节点是8，然后依次是12、14、4和6。而且，不同的添加顺序所构建出来的树也会不尽相同。

（图）

***图6-6.*** *一个完全平衡的二分搜索树，其中高亮部分为节点11的搜索路径*

在清单6-2中，我们给出了一个简单的二分搜索树，它的封装方式使其看起来有点像dict类型。我们可以像下面这样使用它：

>>> tree = Tree()

>>> tree["a"] = 42

>>> tree["a"]

42

>>> "b" in tree

False

如您所见，我们将插入和搜索操作都实现成了独立的函数，而非方法。这使得它们能在None节点上操作。（当然，您完全不必这样做。）

***清单6-2.*** *二分搜索树中的插入与搜索*

class Node:

lft = None

rgt = None

def \_\_init\_\_(self, key, val):

self.key = key

self.val = val

def insert(node, key, val):

if node is None: return Node(key, val) # Empty leaf: add node here

if node.key == key: node.val = val # Found key: replace val

elif key < node.key: # Less than the key?

node.lft = insert(node.lft, key, val) # Go left

else: # Otherwise...

node.rgt = insert(node.rgt, key, val) # Go right

return node

def search(node, key):

if node is None: raise KeyError # Empty leaf: it's not here

if node.key == key: return node.val # Found key: return val

elif key < node.key: # Less than the key?

return search(node.lft, key) # Go left

else: # Otherwise...

return search(node.rgt, key) # Go right

class Tree: # Simple wrapper

root = None

def \_\_setitem\_\_(self, key, val):

self.root = insert(self.root, key, val)

def \_\_getitem\_\_(self, key):

return search(self.root, key)

def \_\_contains\_\_(self, key):

try: search(self.root, key)

except KeyError: return False

return True

|  |
| --- |
| **请注意：**清单6-2中的实现不允许树结构中出现重复的键。如果您插入的新值拥有一个现存的键，那么该键的原值就会被直接重写。要改变这一行为也很容易，因为树结构本身是能够接受重复的。 |

|  |
| --- |
| **有序数组、树以及字典之间，到底选择哪一种？** |
| **虽然，（有序数组上的）二分法、二分搜索树以及字典（即散列表）都实现了一项基本功能：高效率的搜索。但它们之间有着一些重要区别。其中，数组二分法速度很快，开销也较小，但依赖于相关数组（例如Python中的list）的排序情况。而且维护起来会很麻烦，因为在数组中添加元素是一个线性操作。而搜索树的开销会更大一些，但由于它的动态性，插入和移除元素会很方便。不过在很多情况下，散列表（即字典形式）会显得更有优势。因为它理论上的平均时间复杂度是常数级的（相比之下，数组的二分法和搜索树则都是对数级时间），而且在实践中也近乎如此，开销很小。**  **但散列表会要求我们计算出相关对象的哈希值。在实践中，我们几乎总能实现哈希值的计算，但在理论上，数组二分法与搜索树在这方面会更灵活一些——它们只需要比较相关对象的大小即可[[6]](#footnote-6)。这种对于排序的关注也意味着搜索树将允许我们按照既定顺序来访问——无论是整体访问还是某部分的访问。而且，树还可以被扩展成多维结构的操作（譬如在某个超矩形网络区域中搜索某个点）或其它更奇特的搜索形式，而用散列表来处理这些情况就会非常麻烦。除此之外，我们可以举出更多不适合用散列表来处理的情况，例如，如果我们想找出目标序列中最接近被搜索键的那个元素，就基本上只能用搜索树来做。** |

#### 选取算法

下面，我打算用一个算法来结束“折半搜索”这一节的内容。尽管该算法没有什么机会在实践中被大量使用，但它代表了一个有趣的二分法运用方向。除此之外，它也是另一个经典算法——快速排序法的基础算法（这是我们下一节要讨论的话题）。

该算法要解决的问题是：在*线性时间*内找出一个无序序列中第*k*大的数。或许，该问题最重要的用途是找出中间值——也就是该序列完成排序后*位于中间*（即(n+1)//2）的那个元素值[[7]](#footnote-7)。有意思的是，该算法还有一个附带作用：即它也能找出所有*比目标元素小*的元素。这意味着，该算法也可以用于在Θ(*n*)时间内找出序列中*k*个值最小的元素（以及*n-k*个值最大的元素）[[8]](#footnote-8)，且时间复杂度与*k*的具体取值无关。

以上描述乍看之下可能会让人觉得有些糊涂。我们对运行时间的规定似乎就已经将可能的排序算法排除在外了（除非我们能够做某种计数处理，使得我们可以用到第4章中所讨论的计数排序法）。而其它*公认*能用于查找*k*个最小值的算法都是通过某种数据结构对这些对象的持续跟踪来实现的。例如我们可以用某种类似于插入排序的方式，在序列首端或另在一个序列中持续跟踪目前已插入的*k*个值最小的对象。

如果我们持续跟踪了其中值最大的对象，那么检测出主序列中各“大”对象的速度是很快的（只需一个常数级检测即可）。但如果我们想在已经跟踪了*k*个对象的情况下再添加一个对象，就会需要移除掉一个对象。当然，我们要移除的肯定是跟踪结构中值最大的元素，但接下来您得再找到剩下的元素中最大的那个才行。我们可以让其一直保持有序状态（该做法非常类似于插入排序法），但这样一来，该算法就无论如何都需要Θ(*nk*)级的运行时间。

对于该算法而言，（在渐进时间上的）可以进一步采取的一种改进是改用*堆结构*来处理，将其中的“部分插入排序法”替换成“部分堆排序法”，并确保相关堆中的元素数量始终不超过k个（关于堆的更多信息，请参考稍后的黑盒子专栏：二分堆、heapq与Heapsort）。在这种情况下，其运行时间就变成了Θ(*n* lg *k*)，且在*k*值较小的情况下，基本非常接近于Θ(*n*)。除此之外，这种方式还能使我们在实现主序列遍历时避免内存跳转操作。因而在实践中，堆结构时常是个不错的选择。

|  |
| --- |
| **提示：**在Python中，如果我们要在一个可迭代容器中查找*k*个值最小（或值最大）的对象，且当*k*相对于对象总数量来说取值很小时，可以用heapq模块中的nsmallest（或nlargest）函数来处理。而当*k*的取值较大时，我们可能就需要先将序列排序（这里既可以调用相关容器的sort方法，也可以调用独立的sorted函数），然后再选取前*k*个元素了。我们可以根据具体的计时来选择最佳的工作方式，当然，您也可以简单地选择一个能让你的代码最清晰的实现版本。 |

那么下一步，我们该如何消除该算法在渐进时间上对于*k*的依赖性呢？看起来，在这种情况下想要确保其最坏情况为线性级时间会有些难度，所以我们还是先关注其平均情况吧。如果现在我说这里应该用分治法来处理，您会怎么做呢？我们的第一个线索是线性时间的目标，那么怎样的一个“对半分”递归过程有这种属性呢？很简单，我们之前在讨论淘汰赛制的时候已经得出了它的递归式，它只有一个递归调用：*T*(*n*) = *T*(*n*/2) + *n*。换句话说，我们需要运用某种线性操作将现有问题一分为二（或者更准确地说，是在平均情况下是“平分”），就像我们之前在经典分治法中做的一样，但接下来，我们只处理其中的一半问题，使其更接近于二分搜索。也就是说，在该算法的设计中，我们需要找到一种方法，在线性时间内将问题数据分成两半，而所有要找的对象在某一半中。

和之前一样，到了这一步我们得系统性地看一下自己手里究竟拥有哪些工具，这样才能尽可能清晰地分析问题，并且也更易于提出问题的解决方案。总而言之，我们现在已经将目标序列分成了两半，其中一半的值偏小而一半则偏大。我们也不必确保这两半的规模一样大——只要平均情况下是相等的即可。一种简单的分割方式就是找出这些值中所谓的分割点，以此来完成划分。例如所有小于（或等于）该分割值的，我们都放在左半部分，而所有大于分割值的都放在右半部分。在接下来的清单6-3中，我们为这种划分和选取的算法思路提供了一种可能的实现。需要注意的是，该版本是比较偏重于可读性的。在习题6-11中，我们会让您试着移除一些不必要的开销。现在，选取方式也已经写好了，它将返回序列中第*k*小的元素。但如果我们想返回的是序列中*k*个值最小的元素，就只需将返回变量pi改换成lo即可。

***清单6-3.*** *对这种划分与选取算法的一种简单实现*

def partition(seq):

pi, seq = seq[0], seq[1:] # Pick and remove the pivot

lo = [x for x in seq if x <= pi] # All the small elements

hi = [x for x in seq if x > pi] # All the large ones

return lo, pi, hi # pi is "in the right place"

def select(seq, k):

lo, pi, hi = partition(seq) # [<= pi], pi, [> pi]

m = len(lo)

if m == k: return pi # We found the kth smallest

elif m < k: # Too far to the left

return select(hi, k-m-1) # Remember to adjust k

else: # Too far to the right

return select(lo, k) # Just use original k here

|  |
| --- |
| **在线性时间内完成选取操作，我们打包票！** |
| **对于本节所实现的选取算法，业界通常称之为*随机选取*（尽管在真正的随机版本中，我们对分割点的选取还要比这里更随机一些，详细情况可参考习题6-13）。虽然通过该算法，我们通常能预期在线性时间内完成某种选取操作（例如找出中间项），但如果各步骤中所选的分割点不太合适，也有可能最终会导致一个握手循环（这是一种归简规模为1的线性操作），使其变成平方级操作。当然，这种极端的情况在实践中并不多见（对此，建议您再次参考一下习题6-13），而且这种最糟糕的情况在实际操作中是完全可以避免的。**  **相关研究结果表明，只要该算法选择的分割点能按照一个小小的固定比例来分割目标序列（也就是说，只要分割点不位于序列的两端，或者与两端的距离是一个常数），它就一定是个线性级算法。一群算法学家在1973年首先提出了一个能做出这个保证的算法（这些人分别是Blum、Floyd、Pratt、Rivest和Tarjan）。**  **尽管他们所实现的这一算法版本有些复杂，但其核心思路依然是很简单的：即首先将目标序列分成五个一组（也可以是其它较小的常数）。然后用某种简单的排序法找出每一份的中间项。到目前为止，我们执行的都是线性操作，但接下来我们要递归调用这一线性选取算法，以找出这些中间项中的中间项（这是可行的，因为这些中间项的数量规模肯定要小于原序列——尽管会让人觉得有些费解）。其得到的结果就是我们要找的、保证能避免低效递归的分割点，我们将把它用在选取操作中。**  **换而言之，该算法的递归调用将分两步走：第一步是先在其中间项序列中找出一个合适的分割点；然后第二步则是在原目标序列中使用该分割点。**  **当然了，尽管该算法的理论地位非常重要（因为它证明了我们可以在线性时间内完成选取操作），但在实践中被用到的几率很小。** |

### 折半排序

最后，我们到达了一个与分治策略最密切相关的领域：排序法。我们在这里并不打算太过深入地探讨这一话题。因为Python本身已经为我们提供了业界最好的排序算法（具体请参考稍后的黑盒子专栏：Timsort），其实现方式也很高效。而且事实上，list.sort()的效率足够好，以至于大多数场合都可以充当我们的第一选择，甚至好过一些渐进时间比它好的算法（例如对于选取操作而言）。但本节依然会对一些业界最著名的排序算法进行一些介绍，以便让人们深入理解它们的具体工作方式，毕竟它们也是运用分治法进行算法设计的典型实例。

下面，让我们先来看一下由算法设计界的名家之一C. A. R. Hoare提出的*快速排序法*（*quicksort*）。该算法与上一节介绍的选取算法有着密切的联系，当然后者也是由Hoare提出的（所以有时候也被称为*快速选取法*，即*quickselect*）。这种扩展其实非常简单：如果说快速选取法所代表的是剪枝式的遍历操作——即在递归树中找出一条通往第*k*小元素的路径——那么，快速排序法就是一个完全遍历操作，也就是说，它会针对每个*k*提出一个解决方案。找出序列最小的元素、第二小的元素，一直找下去，并将这些元素放到各自合适的位置上，这个序列就完成排序了。在下面的清单6-4中，我们给出了快速排序法的一个版本：

***清单6-4.*** *快速排序*

def quicksort(seq):

if len(seq) <= 1: return seq # Base case

lo, pi, hi = partition(seq) # pi is in its place

return quicksort(lo) + [pi] + quicksort(hi) # Sort lo and hi separately

如您所见，该算法非常简单，我们只要拥有partition函数即可（在习题6-11和习题6-12中，我们会要求您将quicksort和partition函数重写，使其成为一个就地排序算法）。首先我们用pi将目标序列明确分成左右两部分。然后（在归纳前提为真的情况下）对这两部分持续递归该排序算法。最后以中间项为分割点连接这两部分，以确保产生一个有序序列。而由于我们无法保证各个分区中递归属性的平衡，我们只能确定快速排序法在*平均*情况下是个线性对数级算法——而在最坏情况下则是个平方级算法[[9]](#footnote-9)。

快速排序法是一个典型的分治类算法，其主要工作在递归调用开始*之前*，也就是在数据*划分过程*（即在分区阶段）中就已经做完了。而合并部分的操作则相对没有那么重要。虽然，我们也可以采用其它方式来做：譬如从中间位置将数据一分为二，使其成为一个平衡的递归调用（以获得一个不错的最坏运行时间），然后再在合并阶段做些排序努力，或者直接*归并出*结果，这也正是*归并排序*的做法：就像在本章开头所介绍的天际线算法中，从插入单座建筑开始，到归并两条天际线为止的整个过程一样，归并排序也是先在一个有序序列中插入单个元素（即插入排序），最后把两个有序序列归并起来。

虽然我们在第3章中已经见过了归并排序的代码（见清单3-2），但我们在此基础上加了一些注释，并把它重新贴在下面（见清单6-5）。

***清单6-5.*** *归并排序*

def mergesort(seq):

mid = len(seq)//2 # Midpoint for division

lft, rgt = seq[:mid], seq[mid:]

if len(lft) > 1: lft = mergesort(lft) # Sort by halves

if len(rgt) > 1: rgt = mergesort(rgt)

res = []

while lft and rgt: # Neither half is empty

if lft[-1] >= rgt[-1]: # lft has greatest last value

res.append(lft.pop()) # Append it

else: # rgt has greatest last value

res.append(rgt.pop()) # Append it

res.reverse() # Result is backward

return (lft or rgt) + res # Also add the remainder

现在，我们理解该算法的具体工作过程要比在第3章中容易一些了。请注意其归并部分所写的内容，看看它具体做了哪些事。另外，如果您在Python中需要实际使用归并排序（或者其它类似算法），或许也可以用heapq.merge来完成合并操作。

|  |
| --- |
| **黑盒子专栏之：Timsort** |
| **在Python中，list.sort()后台采用的是由Tim Peters提出的算法，Tim是Python社区中最为著名的人物之一[[10]](#footnote-10)。该算法被命名为*Timsort*，它取代了我们早期所用的一个非常复杂的算法，后者的实现非常的繁琐，通常会针对特殊情况来处理一些诸如升序或降序分区之类的东西。而在Timsort中，这些特殊情况的处理被通用化、机制化，所以性能上依然会保持原样（甚至在某些情况下还会有所改善），但算法本身显得更清晰、更简单。当然了，该算法也有一些需要详细说明的细节问题，我们会在这里为您做一个快速预览。如果想知道更多细节信息，您需要查看源代码[[11]](#footnote-11)。**  **Timsort与归并排序的关系非常密切。这是一个*就地型*算法，其归并分区、产生结果的过程都会在原数组中进行（尽管在具体归并过程中还是会用到一些辅助的内存空间）。它不是简单地将数组分成一半一半，分别排序后再加以归并了；而是会在开始阶段查找一下该数组中有哪些*已处于有序状态*的分区（包括可能的反序情况），我们称这种分段为“*run*”。尽管这种分区在随机数组中并不多见，但其在真实数据中却往往大量存在——这样一来，与归并排序相比，Timsort就有了显著的性能优势，即在最好的情况下它可以拥有线性级运行时间（并且还涵盖了许多比输入序列完全有序更为复杂的情况）。**  **Timsort会在遍历目标序列的过程中将其中的各个run标识出来，并将它们的边界位置放进一个栈结构中，然后再根据某种经验法则来决定何时合并这些run。这种思路可以避免各种合并操作的不平衡而形成一个平方级运行时间，同时又充分利用数据原有结构（即run）的特性。首先，所有短小的run都会被手动扩展并排序（利用稳定的插入排序法）。其次，我们针对栈上的三个顶层run：A、B和C（其中A在最顶层）设置下面两个不变式：len(A) > len(B) + len(C)、len(B) > len(C)。当第一个不变式不成立时，A和C中较小的那一个将会与B合并，并用结果替换掉堆栈中被合并的run。如果这时第二个不变式也不成立，就继续合并，直至两个不变式都成立为止。**  **此外，该算法中也用了一些其它方面的技巧，以便尽可能多地提升速度。如果您感兴趣的话，我推荐您去找一些源代码来看看[[12]](#footnote-12)。如果不想看C代码的话，在PyPy项目[[13]](#footnote-13)的相关组件中，您也可以找一版用纯Python实现的Timsort算法。该版实现有详尽的注释，代码也非常清晰（关于PyPy项目，您也可以参考附录A中的相关讨论）。** |

#### 排序操作究竟可以有多快？

对于排序来说，一个很重要的结论是归并排序这类分治算法已经属于*最优状态*了；一个对于任何值都可用的算法（当然，前提是这些值要能比较大小），最坏情况下至少需要消耗Ω(*n* lg *n*)的时间。这之中最重要的情况是，该算法能支持我们对于任意实数的排序[[14]](#footnote-14)。

|  |
| --- |
| **请注意：**计数排序法及其相关算法（我们在第4章讨论过）似乎不符合上面所说的结论。但值得注意的是，在该类算法中我们无法对任意值进行排序——我们需要其能被计数，这意味着待排序的对象必须要能被散列处理，并且还需要在线性时间内完成整个值区间的遍历。 |

那么，我们是如何得出这一结论的呢？其实很简单。第一个理由是：由于我们针对的是任意值，而且这些值是可以比较出大小的，所以这里的每轮对象比较归根结底都是一个yes/no的问题。接下来是第二个理由：n个数应该有n!种排列，而（在最糟糕的情况下）我们要找的只是其中一种。但这个理由又能得出什么结果呢？这时我们又回到了之前那个“猜粒子”的问题上，或者结合当前情况来说，我们现在是要“猜出一种排列”。这意味着我们能做到的最好情况是在Ω(lg *n*!)时间内根据这些yes/no的问题（即比较操作）来获取正确的排列（即对相关数字进行排序）。而碰巧的是lg *n*!在渐进意义上等效于*n* lg *n* [[15]](#footnote-15)。换句话说，其在最糟糕的情况下的运行时间应该是Ω(lg *n*!) = Ω(*n* lg *n*)。

接下来我们又会问：这个等效性又是怎么来的呢？这里最简单的解释就是*斯特林渐进式*（*Stirling's approximation*）[[16]](#footnote-16)，该公式认为*n*!应该等效于Θ(*nn*)。然后只要对双方取对数，该等式就显而易见了[[17]](#footnote-17)。现在，我们得出了问题在*最坏情况*下的性能边界。而且，通过信息理论（这里就不深入阐述了）我们可以知道，这实际上也极有可能是该问题的*平均*性能水平。换而言之，从非常现实的意义上来说，除非我们能确实了解相关数据的分布情况，或者其取值范围，否则线性对数级就是我们所能达到的最佳性能。

### 三个额外实例

在本章结束之前。我打算再带大家来看三个例子。其中前两个用于处理几何计算（这类问题非常适合用分治策略），而最后则是一个相对简单的数字序列问题（但也会让我们经历一些有趣的曲折）。当然，这里只会草拟这些问题的解决方案，因为阐述其中的设计原则才是我们的重点。

#### 最近点对问题

问题是这样的：某平面中存在着一些点，而我们要在其中找出距离最接近的一对点。对此，人们首先会想到用暴力破解法：即检查每一点到其它点的距离（或者至少要检查那些还没有被检查过的点对）。显然，（根据握手问题的求和式来看）这应该是个平方级算法。但通过分治法，我们可以将其运行时间降为线性对数级。

这是一个相当有吸引力的问题，所以如果您对解谜感兴趣的话，也可以在继续阅读我们相关解说之前先试试看是否能自己解决掉它。尽管在这里我们已经给了您一个强烈的暗示，即我们应该用分治法（设计一个线性对数级算法），但这并不意味着解决方案就显而易见了。

从结构上来说，该算法与我们之前用分治法所设计的线性对数级算法（譬如归并排序）相差无几：我们依然会将这些点分成两个子集，然后递归性地找出每个子集中距离最接近的那对点，并在线性时间内合并出最终结果。然而，尽管我们可以用归纳法/递归法（以及分治模式）将问题简化成一般性的归并操作，但在进一步发挥创意之前，我们可能还需要再剥离掉一些东西：即在我们的合并结果中，最近点对必须要么（1）来自左侧，要么（2）存在于右侧，或是（3）*同时来自两侧。*也就是说，我们对最近点对的查找也应该要“横跨”分割线两侧。尽管每当我们这样做的时候，通常会为这种距离设置一个上限（来自左右两侧的最近点对中的最小值）。

现在开始要深入问题的本质了，让我们来看看这里究竟有多少麻烦。先假设中间区域中（宽度为2*d*的）所有的点都已经按y坐标排好了序。然后我们要按顺序逐步考察各点到其它点的距离，找出所有距离小于*d*的点对（以作为目前为止找到的最近点对）。但问题是，我们需要考察多少“相邻点”呢？

这正是该解决方案的关键所在：我们都知道，中间线两侧的所有点之间都至少有一个*d*距离。因为我们要找的是一对距离*最远*不超过d且横跨中间线的点，所以我们每次都只需要考虑一块高度为*d*（且宽度为2*d*）的垂直切片。那么究竟会有多少个点落在该区域内呢？

下面，我们用图6-7来做个说明。由于我们没有为左右之间的距离设置下界，所以在最坏的情况下，有可能会有两个点正好重合在中间线上（图中高亮部分）。但除此之外，我们可以轻而易举地证明在一个最短距离为*d*的*d* ×*d*正方形区域内最多只能容纳四个点，中间线两侧都是如此（详见习题6-15）。也就是说，我们最多只须考虑切片中8个这样的点，这意味着每个点最多只能与其下面的7个相邻点进行比较（其实比较*5个*相邻点就够了，请参考习题6-16）。

现在，该做的都*差不多*已经做完，剩下的就只有其x、y坐标的排序问题了。对此，我们要对x坐标进行排序，以便逐步将问题对半分；然后在合并时，则用y坐标的排序来完成线性遍历。期间，我们会分别用两个数组来进行：在面向x坐标的数组中，我们进行的是递归划分操作，这显然相当简单。而y坐标的处理虽然没有那么直截了当，但也依然很简单：我们在划分x数组数据的同时也基于x坐标替y数组做了分区。所以当我们进行数据合并时，这两个数组也就自然被合并起来了。与归并排序一样，我们在维持顺序的同时确保了算法的线性级运行时间。

（图）

***图6-7.*** *最坏的情况：在中间区域，某大小为d × 2d的垂直切片中存在着八个点，其中（高亮部分的）两个中间点各表示一对重合点。*

|  |
| --- |
| **请注意：**对于该算法操作来说，我们会在各轮递归调用中返回针对每次递归调用来排序的全体子集。所以对离中间线过远的点的过滤工作就必须在其副本中完成。 |

另外，为了得出一个期望的运行时间，我们可以将该算法看作是一种加强版的归纳前提法（相关讨论请参考第4章）：只不过这里不止假设了我们能在某个较小点集中找到最近点对，*也*假设了自己得到的将是重新*排序过*的点集。

#### 凸包问题

这是另一个几何学问题。试想一下，如果我们在一块板上钉了n个钉子，然后用一个橡皮圈将其围了起来。那么被橡皮圈围起来的这个形状就被叫做这些点（钉子）的凸包（convex hull）。要计算的是能容纳这些点的最小凸形[[18]](#footnote-18)区域，也就是说，这是一个由这些点的“最外层”所连成的凸多边形。其具体实例如图6-8所示。

现在，我肯定您一定在怀疑我们是否还能采用之前那种解决方式：先将这些点平均分成两半，然后分别递归解决它们，最后再用一个线性操作合并这两个解决方案。对此，图6-9给了我们相关的提示：即关键是要找出这些点上下边界的*共同切线*。（既然是切线，那简单地说就意味着其经过这些点之前和之后的角度都应该是向内曲折的）

（图）

***图6-8.*** *一组点元素将其组成的凸包*

（图）

***图6-9.*** *通过寻找相关点元素的上下共同切线（虚线部分）来合并两个小型凸包*

下面我们来谈谈除具体实现细节以外的东西：假设我们可以检查某条线是否为其左右两部分的任意一部分的上切线的话（下切线的情况完全类似），那么我们就可以从*左半部分最右边*的点开始，画一条线到*右半部分最左边*的点。只要这条线不是左半部分的上切线，我们就可以沿着该子凸包逆时针移动到下一个点继续画线，然后对右半部分也进行一样的操作，这个过程可能需要重复很多次。一旦两个顶点被确定了，我们就可以用同样的方式找出其下切线。最后，我们只需要移除这两条切线包围着的所有线段，就大功告成了。

|  |
| --- |
| **寻找凸包的速度究竟能有多快？** |
| **通常情况下，分治类算法都具有*O*(*n* lg*n*)级运行时间，尽管也有些极个别的凸包算法会略快一些，其时间复杂度可达到*O*(*n* lg*h*)级，h指的是落在凸包边缘线上的点元素数目。当然，在最坏的情况下，所有的点元素都会落在其边缘线上，这时候算法的运行时间又会变回*O*(*n* lg*n*)。而且事实上，最坏的情况下，*O*(*n* lg*n*)是所有可能的算法里面最好的了——但这个结论又是如何得出的呢？**  **对此，我们可以用第4章中的*归简思维*来证明它的“硬度”。如本章之前所述，实数排序在最坏情况下是个Ω(*n* lg*n*)级操作。这不是我们所采用的算法能决定的，我们就是不可能做得比这更好了。**  **现在，我们观察到排序操作可以被化简为凸包问题。在对*n*个实数进行排序时，我们会直接以该数为x坐标，并为其添加相应的y坐标，以便将它们置于一个柔和曲线上。例如，设*y* = *x* 2，而当我们在查找一些点元素的凸包时，这些值最终也需要按照一定的顺序排列。所以，我们可以通过遍历这些点之间的边来对其进行排序，这里，化简步骤的时间复杂度只是线性级而已。**  **现在，姑且先假设我们手里有一个时间复杂度好于线性对数级的凸包算法。那么通过上面的线性化简法，我们理应会立即得到一个时间复杂度好于线性对数级的*排序*算法。但这显然是不可能的！换句话说，因为如果真存在着一种简单的（线性级）方法，可以把排序问题化简成凸包查找问题，那就说明后者的问题难度至少与前者相同。所以……该问题的解决方案最多也只能做到线性对数级。** |

#### 最大切片问题

下面是最后一个问题：这回我们需要在一个实数序列中截取一个切片（或者片段）A[i:j]，使A[i:j]的和值是该序列所有片段中最大的。当然，我们不能直接选取整个序列，因为其中可能存在着一些负数[[19]](#footnote-19)。该问题有时会以股票交易为背景来表达——用以表示股价的波动情况，这时我们通常会希望能找出能使该股票收益最大化的时间段。（当然，这个类比是有点瑕疵的，因为这意味着我们要能提前预知股票的走势。）

对此，我们也可以采用类似下面这种显而易见的解决方案（其中n=len(A)）：

result = max((A[i:j] for i in range(n) for j in range(i+1,n+1)), key=sum)

在上面这个生成器表达式中，我们分别对起点和终点这两个条件进行了直接遍历，然后以A[i;j]（键值）的和值为准找出最大值。该解决方案体现的短小精悍或许让人觉得“聪明”，但其实并没有什么聪明之处。这不过是一个很简单的暴力破解方案而已，而且还是个立方级算法（即运行时间为Θ(*n*3)）！换句话说，该方案*实际上*很糟糕。

尽管我们暂时无法找到一种明确的思路来避免上面那两个显式循环，但我们可以试着从求和式中那个隐性循环入手。关于这一点，我们可以选择在一次迭代操作中考虑所有*k*长度的区间，然后下一次迭代再考虑所有*k*+1长度的区间，依此类推下去。这样一来，尽管依然还是要面对平方数量的区间检查，但接下来我们就技巧性地确定了其线性级成本的扫描操作——在这种情况下，尽管第一个区间中的和值还是要照常计算，但随后该区间每次只向右移动了一个位置，所以我们只需要减去被移出去的元素，然后再加上新增元素就可以了。

best = A[0]

for size in range(1,n+1):

cur = sum(A[:size])

for i in range(n-size):

cur += A[i+size] -= A[i]

best = max(best, cur)

虽然这样做也没能使性能得到大量改善，但至少我们已经将运行时间降到了平方级。当然，这不是我们停止前进的理由。

下面，我们来看看能否对其运用一点分治法。基本上来说，当我们明确搜寻目标时，相关算法——或者说至少是其大致轮廓——就差不多已经自行完成了：即将序列分成两半，分别（递归性地）找出各自的最大切片。然后看看是否存在横跨中间点的更大切片（这与最近点对问题相同）。换句话说，解决该问题唯一要做的创造性思维是找出横跨中间点的最大切片。对此，我们可以进一步将问题化简——该最大切片可以被分成从左边到中间点的部分与从中间点到右边的部分，我们可以通过直接线性遍历来计算出两侧到中间点的各个和值，以此来实现分别查找。

因此，我们可以自己设计线性对数级方案来解决问题。但在结束本节内容之前，我还不得不提一下，事实上该问题还有一个*线性级*解决方案（详见习题6-18）。

|  |
| --- |
| **真正的分工协作：多重处理** |
| **分治法的设计目标主要是为了在尽可能少的时间内平衡各层递归调用的工作量。通过将这些工作量分配给各个独立处理器（或核心），我们还能进一步增强这一效果。如果我们手里有着大量的处理器，理论上就可以做到很多具有吸引力的事情，譬如说，在对数时间内计算出某序列的和值或找出其中的最大值等。（您知道怎么做吗？）**  **当然，在更为现实的情况下是不太可能有那么多处理器可用的，但如果我们想要取得这些能力，multiprocessing（多重处理）模块会成为我们的好帮手。并行编程通常是利用并行的（操作系统）*线程*来实现的。尽管Python本身也提供了线程机制，但它并不支持真正意义上的并行执行。这种情况下，我们*可以*用并行*进程*来做这些事，并行进程在现代操作系统中的实现效率很高。multiprocessing模块为我们提供了一整套与多线程比较相似的并行处理接口。** |

### 树的平衡与再平衡[[20]](#footnote-20)

通常情况下，如果我们在一个二分搜索树中插入随机值，它大致都能维持一个平均意义上的平衡。但如果运气不好的话，我们*也可能*会得到一个非常不平衡的树结构，基本上就是一个像图6-1那样的链表。在现实世界中，我们所用的大多数搜索树结构中都含有某种再平衡的形式。它们其实是一组用于重组树结构、确保其*平衡状态*的操作（当然，前提是不能破坏搜索树本身的属性）。

尽管各种树结构及其再平衡方法都不尽相同，但它们通常都由以下两类基本操作发展而来：

* + *节点的分割（与合并）*：节点可以拥有两个以上的子节点（和一个以上的键），并且在某些特定情况下，一个节点也有可能会出现*溢出现象*（*overfull*），这时它就可以被分割成两个节点（这也有可能会使它们的父节点*溢出*）。
  + *节点的翻转*：该类操作则依旧作用于二叉树，只不过我们会对边进行切换。即如果*x*是*y*的父节点，我们现在就要让*y*成为*x*的父节点。另外在这项操作中，*x*还必须接管*y*的另一个子节点。

尽管前面的描述让人有些困惑，但我们将深入地谈谈细节，然后您就明白这其中所有的原理了。让我们先来看一个被称为2-3树的结构。在一个普通的二叉树结构中，每个节点通常都只有两个子节点，且各自都只能有一个键。而在2-3树中，我们允许一个节点拥有*一到两个*键，最多*三个*子节点。并且，该节点的*左子树*上的节点都要小于其*最小*键值，同时*右子树*上的节点都要大于其*最大*键值，而中间子树上的节点值则必须落在这两者之间。下面，我们来看看2-3树的这两种节点类型，如图6-10所示：

（图） （图）

*双节点型 三节点型*

***图6-10.*** *2-3树的节点类型*

|  |
| --- |
| **请注意：**2-3树其实是B树的一种特殊情况。B树是几乎所有数据库，以及基于磁盘的树形系统（例如地理信息系统、图像检索系统等）的基石。B树结构带来的一个重要扩展就是它可以拥有上千个键（和子树），其每个节点通常会在磁盘上被存储为一个连续的块。使用大块存储的目的主要是为了尽可能地减少我们访问磁盘的次数。 |

在2-3树中，节点的搜索相当简单——只需进行剪枝式的递归遍历即可，这与普通二分搜索树没有什么不同。但在插入方面，倒是有一些需要额外注意的地方。在往二分搜索树中插入新值时，我们通常先寻找适合插入该值的叶节点。在二分搜索树中，这些位置总是None引用（即空子节点），相当于是在现有节点上“追加”新的节点。而在2-3树中，我们则总是要往现有叶节点内添加新的值（当然，在往树上添加第一个值的时候必然要创建新节点，这在所有树结构中都一样）。也就是说，只要该节点还有空间（也就是说它是双节点型的），我们就直接将值添加进去。如果没有空间了，我们则要考虑三个键的情况了（即已有的两个，加上我们要新增的值）。

其解决方案就是*分割*节点，将三个值中的*中间值*向上移到父节点中（如果我们分割的是根节点，那就创建一个新的根节点）。如果其*父节点*也溢出了，那么就*继续分割*下去。最后，这种分割操作将产生一个重要结果，即该树上所有的叶节点最终都将位于同一层，这意味着它将处于完全平衡的状态。

到目前为止，虽然节点分割的思路相对比较容易理解，我们眼下可以将其贯彻到更简单的二叉树中去。但是，这里只不过应用了2-3树的*思路*，并没有真正把它*实现*成2-3树。我们可以只用二叉节点结构来模拟整个过程。这样做有两个优点：第一，该结构更为简单，也更为一致；第二，我们可以在不需要学习一个全新的平衡方案的情况下学习了解节点翻转操作（这也是一个很重要的通用技术）。

上述“模拟结构”就是我们接下去要讲的AA树，它是根据其创建者的名字Arne Andersson.来命名的[[21]](#footnote-21)。在众多基于翻转的平衡方案中，AA树因其简单性而真正地占据了一席之地（尽管新手还是会觉得有些头痛）。然而AA树是一种二叉树，所以我们需要学习一下怎样用它来模拟三节点型的平衡法。我们可以从图6-11中看到它的工作方式：

（图） （图）

被反转的三节点型结构 有效三节点型结构

***图6-11.*** *用AA树模拟的两种三节点型结构（高亮部分），值得注意的是左图是一个被反转的结构，必须要进行修复。*

上图说明了以下几个要点：首先，我们有了一种模拟三节点型结构的方式，即将那两个节点连接起来模拟单一的*伪节点*（图中高亮部分）。其次，该图还诠释了一种*分层*思路，树上的每个节点都会被分配有一个层次（或者说一个数字），其中所有叶节点的层次都为1。当我们让上面一层的两个节点参与伪装一个三节点型结构时，我们就直接赋予它们相同的层次，就像上图垂直方向上所展现的那样。第三，三节点型结构的“内部”边线（即*横向*边）只能*指向右边*。这意味着上面的左子图中所诠释的是一个*非法*节点，它必须要*向右翻转一下*，加以修正。也就是将*c*变成*d*的左子树，而*d*变成*b*的右子树，最后，再将*d*原有的父节点变成*b*的父节点。瞬间，我们就得到了右子图中所示的结构（而这是一个有效结构）。换句话说，现在指向中间子树的边线和横向边线都做了水平翻转。我们称这种操作为*偏斜（skew）*。

此外，还有一种可能出现的非法情况，我们也得通过翻转操作来修复它。这种情况就是伪节点溢出（也就是四节点型结构），如图6-12所示。在该图中，我们有三个节点链在了同一层（即*c*、*e*、*f*）。所以接下来，我们要模拟出节点的分割：和2-3树一样，将中间键（*e*）移动到父节点中（*e*）。在这种情况下，做法很简单，就是向左翻转*c*和*e*。该操作正好与图6-11中的情况相反。换句话说，c的子节点由e变成了它下面的d，而e的子节点由d变成了它上面的c。最后，a的子节点也由c变成了e。为了记录下a和e现在组成了一个新的三节点型结构这一事实，我们提升了e所处的层次（见图6-12）。我们自然地把这一操作称为*分割*（split）。

（图）

***图6-12.*** *一个溢出的伪节点，及其经历左翻转修复（即交换(e,d)、(c,e)这两条边），e变成a新的子节点后的情况。*

我们在AA树中插入节点的情况与在标准不平衡二叉树结构中基本相同，唯一的不同是我们后期还会进行一些清理（应用偏斜、分割这两种操作）。读者可以在清单6-6中找到这些操作的完整代码。您可以看到，其中的清理操作（一个调用的是skew()，而另一个则是split()）都是在递归性回溯部分中执行的——这样一来，相关节点就会在其回溯到根节点的路径上被修复。这个逻辑具体执行起来会是怎样的呢？

这些沿着路径进行下去的操作实际上只会给我们带来一个影响：即有可能会把另一个节点放到“我们”当前的模拟节点中来。在叶节点层，每次添加节点时这种情况都会发生，因为该层所有节点都在层次1中。而如果当前节点在树上的位置更高一些，在分割操作中有节点被上移的情况下，我们可以在当前（模拟）节点中获得这个节点。无论是哪一种情况，该突然出现在我们当前层的节点可能呈现为*左/右子节点*。如果它是一个*左子节点*，我们可采用偏斜操作（即做右翻转）来修正这个问题。如果它是一个*右子节点*，那就没有任何问题。但如果它是个右*孙节点*，我们就有了一个溢出的节点，这就需要用分割操作（即左翻转）来将我们的四节点型模拟结构的中间节点移动到其父节点层。

这些操作的文字表述实在有点复杂——只能希望下面这段代码能有助于我们理解这些内容。（尽管很可能会让您苦思冥想一段时间。）

***清单6-6.*** *用AA树结构实现再平衡的二分搜索树*

class Node:

lft = None

rgt = None

lvl = 1 # We've added a level...

def \_\_init\_\_(self, key, val):

self.key = key

self.val = val

def skew(node): # Basically a right rotation

if None in [node, node.lft]: return node # No need for a skew

if node.lft.lvl != node.lvl: return node # Still no need

lft = node.lft # The 3 steps of the rotation

node.lft = lft.rgt

lft.rgt = node

return lft # Switch pointer from parent

def split(node): # Left rotation & level incr.

if None in [node, node.rgt, node.rgt.rgt]: return node

if node.rgt.rgt.lvl != node.lvl: return node

rgt = node.rgt

node.rgt = rgt.lft

rgt.lft = node

rgt.lvl += 1 # This has moved up

return rgt # This should be pointed to

def insert(node, key, val):

if node is None: return Node(key, val)

if node.key == key: node.val = val

elif key < node.key:

node.lft = insert(node.lft, key, val)

else:

node.rgt = insert(node.rgt, key, val)

node = skew(node) # In case it's backward

node = split(node) # In case it's overfull

return node

那么，我们能否确定AA树将来一定会是平衡的呢？事实上是可以的，因为它忠实地模拟了2-3树的结构（以及代表了该结构中实际树层次的“level”属性）。而且实际上，由于在模拟的三节点型结构上的任何搜索路径最多只会访问两个节点的缘故，其渐进搜索时间将依然是对数级。

|  |
| --- |
| **黑盒子专栏之：二分堆结构与heapq、heapsort** |
| ***优先级队列（priarity queue）*是第5章中所讨论的LIFO队列及FIFO队列的一种推广。与只根据元素项添加的时刻来决定顺序不同，该队列的每个元素都会有一个优先级，我们总是会获取剩余元素中优先级最低的那一个（也可以是优先级最高的，但这两种用法通常不能同时出现在同一个结构中）。该结构提供的这种功能是几种算法的重要组成部分，譬如Prim提出的最小生成树算法（见第7章）、Dijkstra提出的最短路径搜索算法（见第9章）等。尽管优先级队列的实现可以有很多种方式，但其中最常用于这类目标的数据结构恐怕还是非*二分堆（binary heap）*莫属。（虽然堆结构不止这一种，但我们通常提到*堆* [*heap*] 这个词时，往往指的就是二分堆。）**  **二分堆是一个*完整的*二叉树结构。这意味着它会始终保持平衡。该树除最底层可能不满以外，每一层都必须是满的，并且会从左边开始尽可能地填满最底层。对于这种结构来说，所谓的*堆属性*（*heap property*）无疑是个重点：该属性规定每个父节点的值都必须要小于其所有的子节点（这针对的是*最小值堆*，对于*最大值堆*来说，当然是父节点要大于子节点）。据此推断，根节点自然就应该是堆中的最小值。该属性与搜索树非常类似，但也不完全相同。事实上，堆属性更易于在不破坏树结构的情况下维持平衡。我们从来不需要在堆中通过翻转、分割节点来修复树结构，而只需根据堆属性对父节点与子节点进行交换即可。例如，为了“修复”某子树的根节点（假设它的值太大了），我们只需要直接在当前子树中拿它与最小的子节点进行交换，并递归下去即可。**  **heapq模块实现了一个非常有效率的堆结构。该结构用list来实现，并采用了一种很常见的“编码”形式：即如果a是一个堆，那么a[i]的子节点就位于a[2\*i+1]与a[2\*i+2]这两个位置。同时，这也意味着其根节点（即最小元素）将始终位于a[0]的位置。我们既可以用heappush()和heappop()这两个函数从头开始构建一个堆结构；也可以从一个拥有大量元素项的列表开始，使之成为一个堆结构。对于后一种情况，我们可以用heapify()函数来做[[22]](#footnote-22)。简单地说，该函数会从最靠下边和右边的子树根节点开始，依次向左、向上对每个子树的根节点进行修复（事实上，由于它总是跳过叶节点层的缘故，它只需要对该数组的左半部分进行操作即可）。其运行时间应该为线性级（见习题6-9）。另外，如果这是个有序列表的话，它就已经是一个有效堆了，无需再对其多做什么。**  **下面的这段代码演示了怎样一块一块地构建出堆结构：**  >>> from heapq import heappush, heappop  >>> from random import randrange  >>> Q = []  >>> for i in range(10):  ... heappush(Q, randrange(100)) ...  >>> Q  [15, 20, 56, 21, 62, 87, 67, 74, 50, 74]  >>> [heappop(Q) for i in range(10)]  [15, 20, 21, 50, 56, 62, 67, 74, 74, 87]  **与bisect一样，虽然heapq也是一个用C语言实现的模块，但它过去曾是一个普通的Python模块。例如在下面这段（来自Python2.3的）函数代码中，我们将一个对象下移到了其小于所有子节点的位置（与之前一样，注释依然是我加的）。**  def sift\_up(heap, startpos, pos):  newitem = heap[pos] # The item we're sifting up  while pos > startpos: # Don't go beyond the root  parentpos = (pos - 1) >> 1 # The same as (pos - 1) // 2  parent = heap[parentpos] # Who's your daddy?  if parent <= newitem: break # Valid parent found  heap[pos] = parent # Otherwise: copy parent down  pos = parentpos # Next candidate position  heap[pos] = newitem # Place the item in its spot  **值得注意的是，这个函数的原名是\_siftdown，因为它整理值的方向是（沿着元素的下标）*向下*的（即下标越来越小）。不过，我更愿意将其看作是对堆的隐性树结构的一种*向上*整理。另外，请注意这里的实现方式与bisect\_right()一样，使用的是循环，而不是递归。**  **除heappop()外，还有heapreplace函数，它可以在弹出堆中最小元素的同时插入一个新元素（而且还比在调用heappop()之后接着调用heappush()更有效率一些）。heappop操作会返回根节点（即首元素）。在这过程中，为了维持堆的形状，我们会先将当前最后一个元素移到根节点的位置，再让它与下面的元素进行持续交换（每一步都与其最小的子节点交换），直至其小于其所有子节点为止。而heappush操作则与之正好相反，每当一个新元素被追加到列表尾端时，该元素就会与其父节点进行持续交换，直至它的值大于它的父节点为止。两者都属于对数级操作（即使在最坏情况下也是如此，因为堆结构始终能维持平衡）。**  **最后要说的是，该模块（自Python2.6以来）含有merge()、nlargest()及nsmallest()这些工具函数，它们分别被用于：归并多组已排序的输入、寻找可迭代序列中的前*n*大和*n*小的元素项。其中（与模块中其他函数不同的是）后两个函数都接受一个key参数，它与list.sort()的key参数含义相同。（当然，正如bisect那篇专栏中提到的那样，我们也可以对其余函数应用DSU模式来模拟这些操作。）**  **尽管我们也许不会在Python中直接用到这些堆操作，但它们也能构成一种简单、高效且渐进时间达到最优化的排序算法，我们称之为*堆排序*（*heapsort*）。该算法往往会先实现一个最大值堆（max-heap）结构，先对相关序列执行heapify操作，然后反复（用heappop()）弹出该结构的根节点，最终将其放入最后的空槽（empty slot）。随着堆结构的慢慢收缩，原数组就会从右边开始被逐渐填入，先是最大的元素，接着是第二大的，以此类推。换句话说，堆排序法是一种基于选择排序的算法，其中堆结构被用来充当选取器。由于该算法的初始化操作是线性级的，而n个数的选取操作中的每一个都是对数级的，所以其整体是线性对数级的（也就是最优的）。** |

### 本章小结

对于*分治法*来说，其算法设计策略主要分为三个步骤：首先我们要将一个大问题分解成一系列规模大小基本相同的子问题；然后解决这些子问题（这里通常会用到一些递归法）；最后再将其结果合并。这种策略可用的主要基础来自其工作量的*平衡性*，典型情况下，它能帮助我们将一个平方级复杂度的问题降为线性级问题，我们所介绍的归并排序、快速排序以及点的集合中的最近点对问题或凸包问题，都是这方面的重要例子。在某些情况下（譬如在某有序序列中搜索，或选取中间项时），我们还可以*剪枝掉*相关子问题以外的分支问题，以获取从根节点到相关叶节点之间的子问题路径，从而产生一些更为有效的算法。

另外，这些子问题的结构也是可以用二分搜索树来明确表示的。该树上的每个节点都大于其左子树上的所有节点，而小于其右子树上的所有节点。这意味着二分搜索树可以通过从根节点的遍历来实现。另外，如果我们直接向其中插入一些随机值，搜索树本身在通常情况下是依然能维持平衡的（即其搜索时间仍然为对数级），但也可能通过某种节点划分或转换操作来对该树结构进行*再平衡*，以确保在最糟糕的情况下仍只需对数级运行时间。

### 如果您感兴趣……

如果您对二分法有兴趣的话，还可以去看看一些与*插值搜索法*（*interpolation search*）有关的资料，这个算法针对数据均匀分布的数组的平均运行时间为*O*(lg lg *n*)。而对于有序列表、搜索树以及散列表以外的集合实现（即有效的成员搜索操作），我们建议您看一些与*布隆筛选器*（*Bloom filters*）相关的资料。另外，如果您对搜索树及其相关结构有兴趣的话，那可就有大量的资料可看了。您可以找到许多不同的平衡机制（例如*红黑树*、*AVL树*、*伸展树* [splay tree] 等），其中还包含着一些随机性的（譬如*树堆* [treap] 结构）和一些抽象的树形式（譬如*跳表* [skip list] 之类的）。此外，还有一些专门用于提供多维坐标索引（用于空间访问的方法）和距离（用于*度量访问*的方法）的树结构，以及*区间树* [interval tree]*、四分树* [quadtree]*、八分树* [octtree] 等其它结构。

### 练习题

1. 请用Python实现一个天际线问题的解决方案。
2. 二分搜索树在每个递归步骤中都会将目标序列分割成两个规模相近的部分。而*三分搜索*树则会将该序列分成三部分。那么请问：后者的渐进复杂度会是多少？二分搜索与三分搜索各自需要的比较次数又是多少？
3. 请问多路搜索树与二分搜索树有哪些不同点？
4. 请问我们应该如何在线性时间内按顺序获取一棵二分搜索树中所有的键值？
5. 请问我们应该如何删除二分搜索树中的节点？
6. 假设我们在一个初始状态为空的二分搜索树中插入了n个随机值，那么请问：最左边那个节点（即最小值）的平均深度应该是多少？
7. 在一个最小堆中，每当我们要向下移动一个大节点时，始终要先选择*最小的*子节点，为什么这一点如此重要？
8. 在堆结构中，我们的编码工作是如何进行的？为什么能这样做？
9. 为什么说堆结构的构建是一个线性级操作？
10. 为什么我们不会简单地用平衡二分搜索树来代替堆结构？
11. 请写一个能就地分割序列（即让相关元素只在原始序列中移动）的分割算法。您能让它运行得比清单6-3中的版本快吗？
12. 请用习题6-11中完成的就地型分割算法重写快速排序法，令其实现元素的就地排序。
13. 假设我们用random.choice这样的函数重写了对于分割点选择的实现，会带来怎样的不同？（请注意：这种策略也可以用来建立*随机化的快速排序法*）
14. 请实现一个接受key函数的快速排序算法，使其功能接近于list.sort。
15. 请证明，在任意两点间距离都至少为d的前提下，一个边长为d的正方形最多只能包含四个顶点。
16. 在利用分治法解决最近点对问题的过程中，我们在按y坐标排序的点序列中最多只需检测7个点。请证明我们其实很容易地就能将该点数降为5。
17. *元素唯一性*问题主要用于判断目标序列中所有元素的唯一性。该问题在最坏情况下处理实数的性能下界是线性对数级的。请据此证明最近点对问题在最坏情况下的性能下界*也是*线性对数级的。
18. 请问如何在线性时间内解决最大切片问题？

### 参考资料

* Andersson, A. (1993). Balanced search trees made simple. In *Proceedings of the Workshop on Algorithms and Data Structures* (WADS), pages 60-71.
* Bayer, R. (1971). Binary B-trees for virtual memory. In Proceedings of the ACM SIGFIDET Workshop on Data Description, Access and Control, pages 219-235.
* Blum, M., Floyd, R. W., Pratt, V., Rivest, R. L., and Tarjan, R. E. (1973). Time bounds for selection. *Journal of Computer and System Sciences*, 7(4):448-461.
* de Berg, M., Cheong, O., van Kreveld, M., and Overmars, M. (2008). *Computational Geometry: Algorithms and Applications*. Springer, third edition.

1. 译者注：约翰·沃尔夫冈·冯·歌德（Johann Wolfgang von Goethe）：德国著名诗人、戏剧家、自然学家以及政治活动家。 [↑](#footnote-ref-1)
2. 请注意，有些作者会用解决这一术语来描述归纳法中的基本情况，因此产生了有所不同的顺序，变成了分解、解决、合并。 [↑](#footnote-ref-2)
3. 也就是说，我们要将问题规模为n/2解决方案构建成一个问题规模为n的解决方案。 [↑](#footnote-ref-3)
4. 相关说明可参考Udi Manber《Introduction to Algorithms》（见第４章的“参考资料”部分）一书。 [↑](#footnote-ref-4)
5. 例如在天际线问题中，我们很可能会将基本情况下的元素(L,H,R)分成两部分(L,H)与(R,H)，然后combine函数就可以构建一个由点元素组成的序列了。 [↑](#footnote-ref-5)
6. 事实上，这里说“更灵活”也未必完全确切，因为有许多对象（譬如说复数对象）虽然可以被散列，但未必能比较大小。 [↑](#footnote-ref-6)
7. 在统计学中，中间值在长度为偶数的序列中也有明确定义：即为两个中间元素的平均值。但这并不是我们在这里需要担心的问题。 [↑](#footnote-ref-7)
8. 译者注：此处原文为*k* smallest elements，与前文中提到的*k*th smallest element极易混淆。根据上下文，译者窃以为前者指的是应该是“序列中k个值最小的元素”，而后者是“序列中值第k小的元素”。 [↑](#footnote-ref-8)
9. 从理论上而言，我们确实可以通过之前那个可以确保在线性时间内完成的选取算法来找出目标序列的中间项，并将其用作分割点。当然，这种做法在实践中出现的几率并不高 [↑](#footnote-ref-9)
10. 事实上，Timsort算法在Java SE 7的数组排序中也有运用。 [↑](#footnote-ref-10)
11. 例如您可以参考源代码中的listsort.txt文件（或者访问http://svn.python.org/projects/python/trunk/Objects/listsort.txt）。 [↑](#footnote-ref-11)
12. 您可以从以下链接中获取想用的C代码：<http://svn.python.org/projects/python/trunk/Objects/listobject.c>。 [↑](#footnote-ref-12)
13. 详见：<http://codespeak.net/pypy/dist/pypy/rlib/listsort.py>。 [↑](#footnote-ref-13)
14. 当然，实数通常也并没有那么任意——只要我们的数字不超过固定的位数，就能用（第4章中提到的）基数排序法对它们进行线性时间的排序。 [↑](#footnote-ref-14)
15. 我觉得这太酷了，以至于其实很想在句后面加个感叹号，但又担心这种做会让其跟阶乘号混淆起来，于是只好作罢。 [↑](#footnote-ref-15)
16. 译者注：斯特林渐进式（Stirling's approximation）是一条用来获取n!的近似值的数学公式。在通常情况下，当n的值很大时，n!的计算量非常庞大，斯特林公式十分好用，而即使在n的值很小时，斯特林公式的取值也有很高的准确性。 [↑](#footnote-ref-16)
17. 实际上，该渐进式并不完全是自然渐进的。如果想知道该渐进式的更多细节问题，建议您参考一下相关的数学专著。 [↑](#footnote-ref-17)
18. 如果一个区域中的任意两点之间的直线都在该区域内，我们就说它是个凸形区域。 [↑](#footnote-ref-18)
19. 在这里，我会始终假设我们要找的是一个非空区间，但如果它被证明有一个负数和值，那么我们也可以将其替换成空区间。 [↑](#footnote-ref-19)
20. 本节内容会有些难度，而且也并不是理解本书其余内容的必要基础。所以读者可以随意浏览一遍，甚至也可以完全跳过它。您在本节中也许想阅读黑盒子专栏：“二分堆与heapq、Heapsort”中的内容。 [↑](#footnote-ref-20)
21. 从某种程度上说，AA树其实也可以被认为是BB树（即binary B树，二叉B树）的一个版本，后者是Rudolph Bayer在1971年提出的一种2-3树结构的二叉树表示法。 [↑](#footnote-ref-21)
22. 这个操作常被称作*构建堆*（build-heap），同时保留heapify（堆化）这个动词来表示修复单个节点的操作，这种情况下，构建堆结构操作会在叶节点以外的所有节点上运行heapify。 [↑](#footnote-ref-22)