分治策略

在2.3.1节中，我们介绍了归并排序，它利用了分治策略。回忆一下，在分治策略中，我们递归地求解一个问题，在每层递归中应用如下三个步骤：

分解(Divide)步骤将问题划分为一些子问题，子问题的形式与原问题一样，只是规模更小。

解决(Conquer)步骤递归地求解出子问题。如果子问题的规模足够小，则停止递归，直接求解。

合并(Combine)步骤将子问题的解组合成原问题的解。

当子问题足够大，需要递归求解时，我们称之为**递归情况**(recursive case)。当子问题变得足够小，不再需要递归时，我们说递归已经“触底”，进入了**基本情况**(base case)。有时，除了与原问题形式完全一样的规模更小的子问题外，还需要求解与原问题不完全一样的子问题。我们将这些子问题的求解看做合并步骤的一部分。

在本章中，我们将看到更多基于分治策略的算法。第一个算法求解最大子数组问题，其输入是一个数值数组，算法需要确定具有最大和的连续子数组。然后我们将看到两个求解 $n \times n$ 矩阵乘法问题的分治算法。其中一个的运行时间为 $\Theta(n^3)$，并不优于平凡算法。但另一算法(Strassen算法)的运行时间为 $O(n^{2.81})$，渐近时间复杂性击败了平凡算法。

递归式

递归式与分治方法是紧密相关的，因为使用递归式可以很自然地刻画分治算法的运行时间。一个**递归式**(recurrence)就是一个等式或不等式，它通过更小的输入上的函数值来描述一个函数。 65
例如，在2.3.2节中，我们用递归式描述了MERGE-SORT过程的最坏情况运行时间 $T(n)$：

$$T(n) = \begin{cases} \Theta(1) & \text{若 } n = 1 \\ 2T(n/2) + \Theta(n) & \text{若 } n > 1 \end{cases} \tag{4.1}$$

求解可得 $T(n) = \Theta(n \lg n)$。

递归式可以有很多形式。例如，一个递归算法可能将问题划分为规模不等的子问题，如2/3对1/3的划分。如果分解和合并步骤都是线性时间的，这样的算法会产生递归式 $T(n) = T(2n/3) + T(n/3) + \Theta(n)$。

子问题的规模不必是原问题规模的一个固定比例。例如，线性查找的递归版本(练习2.1-3)仅生成一个子问题，其规模仅比原问题的规模少一个元素。每次递归调用将花费常量时间再加上下一层递归调用的时间，因此递归式为 $T(n) = T(n-1) + \Theta(1)$。

本章介绍三种求解递归式的方法，即得出算法的“Θ”或“O”渐近界的方法：

- **代入法** 我们猜测一个界，然后用数学归纳法证明这个界是正确的。
- **递归树法** 将递归式转换为一棵树，其结点表示不同层次的递归调用产生的代价。然后采用边界和技术来求解递归式。
- **主方法** 可求解形如下面公式的递归式的界：

$$T(n) = aT(n/b) + f(n) \tag{4.2}$$

其中 $a \geqslant 1$，$b > 1$，$f(n)$是一个给定的函数。这种形式的递归式很常见，它刻画了这样一个分治算法：生成 a 个子问题，每个子问题的规模是原问题规模的 $1/b$，分解和合并步骤总共花费时间为 $f(n)$。

为了使用主方法，必须要熟记三种情况，但是一旦你掌握了这种方法，确定很多简单递归式的渐近界就变得很容易。在本章中，我们将使用主方法来确定最大子数组问题和矩阵相乘问题的分治算法的运行时间，本书中其他使用分治策略的算法也将用主方法进行分析。 66

我们偶尔会遇到不是等式而是不等式的递归式，例如 $T(n) \leqslant 2T(n/2)+\Theta(n)$。因为这样一种递归式仅描述了 $T(n)$的一个上界，因此可以用大 O 符号而不是 Θ 符号来描述其解。类似地，如果不等式为 $T(n) \geqslant 2T(n/2)+\Theta(n)$，则由于递归式只给出了 $T(n)$的一个下界，我们应使用 Ω 符号来描述其解。

递归式技术细节

在实际应用中，我们会忽略递归式声明和求解的一些技术细节。例如，如果对 n 个元素调用 MERGE-SORT，当 n 为奇数时，两个子问题的规模分别为$\lfloor n/2 \rfloor$和$\lceil n/2 \rceil$，准确来说都不是 $n/2$，因为当 n 是奇数时，$n/2$ 不是一个整数。技术上，描述 MERGE-SORT 最坏情况运行时间的准确的递归式为

$$T(n) = \begin{cases} \Theta(1) & \text{若 } n = 1 \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n) & \text{若 } n > 1 \end{cases} \tag{4.3}$$

边界条件是另一类我们通常忽略的细节。由于对于一个常量规模的输入，算法的运行时间为常量，因此对于足够小的 n，表示算法运行时间的递归式一般为 $T(n)=\Theta(1)$。因此，出于方便，我们一般忽略递归式的边界条件，假设对很小的 n，$T(n)$为常量。例如，递归式(4.1)常被表示为

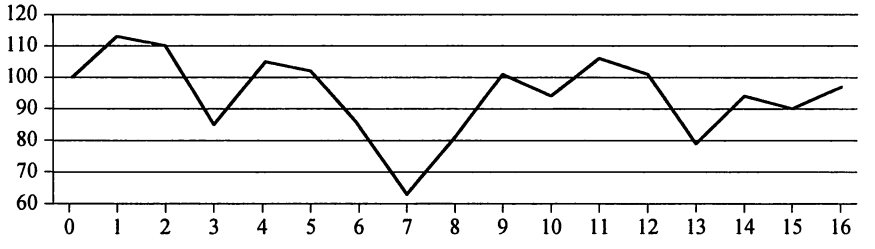
$$T(n) = 2T(n/2) + \Theta(n) \tag{4.4}$$

去掉了 n 很小时函数值的显式描述。原因在于，虽然改变 $T(1)$的值会改变递归式的精确解，但改变幅度不会超过一个常数因子，因而函数的增长阶不会变化。

当声明、求解递归式时，我们常常忽略向下取整、向上取整及边界条件。我们先忽略这些细节，稍后再确定这些细节对结果是否有较大影响。通常影响不大，但你需要知道什么时候会影响不大。这一方面可以依靠经验来判断，另一方面，一些定理也表明，对于很多刻画分治算法的递归式，这些细节不会影响其渐近界(参见定理 4.1)。但是，在本章中，我们会讨论某些细节，展
67 示递归式求解方法的要点。

4.1 最大子数组问题

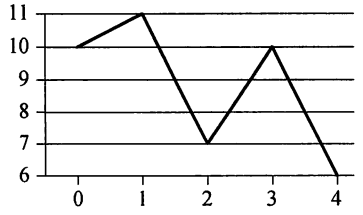
假定你获得了投资挥发性化学公司的机会。与其生产的化学制品一样，这家公司的股票价格也是不稳定的。你被准许可以在某个时刻买进一股该公司的股票，并在之后某个日期将其卖出，买进卖出都是在当天交易结束后进行。为了补偿这一限制，你可以了解股票将来的价格。你的目标是最大化收益。图 4-1 给出了 17 天内的股票价格。第 0 天的股票价格是每股 100 美元，你可以在此之后任何时间买进股票。你当然希望“低价买进，高价卖出”——在最低价格时买进股票，之后在最高价格时卖出，这样可以最大化收益。但遗憾的是，在一段给定时期内，可能无法做到在最低价格时买进股票，然后在最高价格时卖出。例如，在图 4-1 中，最低价格发生在第 7 天，而最高价格发生在第 1 天——最高价在前，最低价在后。



天	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
价格	100	113	110	85	105	102	86	63	81	101	94	106	101	79	94	90	97
变化		13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7

图 4-1 17 天内，每天交易结束后，挥发性化学公司的股票价格信息。横轴表示日期，纵轴表示股票价格。表格的最后一行给出了股票价格相对于前一天的变化

你可能认为可以在最低价格时买进，或在最高价格时卖出，即可最大化收益。例如，在图 4-1 中，我们可以在第 7 天股票价格最低时买入，即可最大化收益。如果这种策略总是有效的，则确定最大化收益是非常简单的：寻找最高和最低价格，然后从最高价格开始向左寻找之前的最低价格，从最低价格开始向右寻找之后的最高价格，取两对价格中差值最大者。但图 4-2 给出了一个简单的反例，显示有时最大收益既不是在最低价格时买进，也不是在最高价格时卖出。 68



天	0	1	2	3	4
价格	10	11	7	10	6
变化		1	-4	3	-4

图 4-2 本例说明最大收益并不一定从最低价格开始或者到最高价格结束。与图 4-1 一样，横轴表示日期，纵轴表示价格。在本例中，最大收益为每股 3 美元，第 2 天买进，第 3 天卖出可获得此最大收益。第 2 天的价格 7 美元并非最低价格，而第 3 天的价格 10 美元也并非最高价格

暴力求解方法

我们可以很容易地设计出一个暴力方法来求解本问题：简单地尝试每对可能的买进和卖出日期组合，只要卖出日期在买入日期之后即可。n 天中共有 $\binom{n}{2}$ 种日期组合。因为 $\binom{n}{2}=\Theta(n^2)$，而处理每对日期所花费的时间至少也是常量，因此，这种方法的运行时间为 $\Omega(n^2)$。有更好的方法吗？

问题变换

为了设计出一个运行时间为 $o(n^2)$ 的算法，我们将从一个稍微不同的角度来看待输入数据。我们的目的是寻找一段日期，使得从第一天到最后一天的股票价格净变值最大。因此，我们不再从每日价格的角度去看待输入数据，而是考察每日价格变化，第 i 天的价格变化定义为第 i 天和第 $i-1$ 天的价格差。图 4-1 中的表格的最后一行给出了每日价格变化。如果将这一行看做一个数组 A，如图 4-3 所示，那么问题就转化为寻找 A 的和最大的非空连续子数组。我们称这样的连续子数组为**最大子数组**(maximum subarray)。例如，对图 4-3 中的数组，$A[1..16]$的最大子数组为 $A[8..11]$，其和为 43。因此，你可以在第 8 天(7 天之后)买入股票，并在第 11 天后卖出，获得每股收益 43 美元。

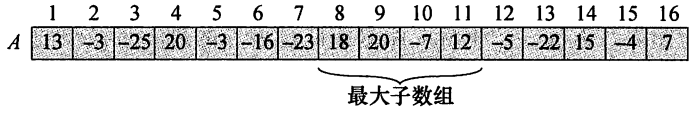


图 4-3 股票价格变化值的最大子数组问题。本例中，子数组 $A[8..11]$的和是 43，是 A 的所有连续子数组中和最大的

乍一看，这种变换对问题求解并没有什么帮助。对于一段 n 天的日期，我们仍然需要检查 $\binom{n-1}{2}=\Theta(n^2)$个子数组。练习 4.1-2 要求证明，虽然计算一个子数组之和所需的时间是线性 69 的，但当计算所有 $\Theta(n^2)$个子数组和时，我们可以重新组织计算方式，利用之前计算出的子数组和来计算当前子数组的和，使得每个子数组和的计算时间为 $O(1)$，从而暴力求解方法所花费的时间仍为 $\Theta(n^2)$。

接下来，我们寻找最大子数组问题的更高效的求解方法。在此过程中，我们通常说“一个最

大子数组”而不是“最大子数组”，因为可能有多个子数组达到最大和。

只有当数组中包含负数时，最大子数组问题才有意义。如果所有数组元素都是非负的，最大子数组问题没有任何难度，因为整个数组的和肯定是最大的。

使用分治策略的求解方法

我们来思考如何用分治技术来求解最大子数组问题。假定我们要寻找子数组 $A[low..high]$ 的最大子数组。使用分治技术意味着我们要将子数组划分为两个规模尽量相等的子数组。也就是说，找到子数组的中央位置，比如 mid，然后考虑求解两个子数组 $A[low..mid]$ 和 $A[mid+1..high]$。如图 4-4(a)所示，$A[low..high]$ 的任何连续子数组 $A[i..j]$ 所处的位置必然是以下三种情况之一：

- 完全位于子数组 $A[low..mid]$ 中，因此 $low \leqslant i \leqslant j \leqslant mid$。
- 完全位于子数组 $A[mid+1..high]$ 中，因此 $mid < i \leqslant j \leqslant high$。
- 跨越了中点，因此 $low \leqslant i \leqslant mid < j \leqslant high$。

因此，$A[low..high]$ 的一个最大子数组所处的位置必然是这三种情况之一。实际上，$A[low..high]$ 的一个最大子数组必然是完全位于 $A[low..mid]$ 中、完全位于 $A[mid+1..high]$ 中或者跨越中点的所有子数组中和最大者。我们可以递归地求解 $A[low..mid]$ 和 $A[mid+1..high]$ 的最大子数组，因为这两个子问题仍是最大子数组问题，只是规模更小。因此，剩下的
70 全部工作就是寻找跨越中点的最大子数组，然后在三种情况中选取和最大者。

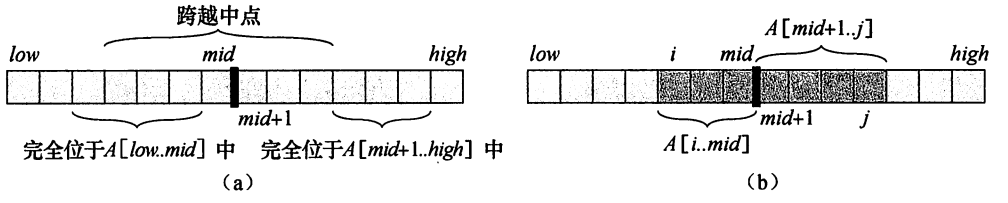


图 4-4 (a)$A[low..high]$ 的子数组的可能位置：完全位于 $A[low..mid]$ 中，完全位于 $A[mid+1..high]$ 中，或者跨越中点 mid。(b)$A[low..high]$ 的任何跨越中点的子数组由两个子数组 $A[i..mid]$ 和 $A[mid+1..j]$ 组成，其中 $low \leqslant i \leqslant mid$ 且 $mid < j \leqslant high$

我们可以很容易地在线性时间(相对于子数组 $A[low..high]$ 的规模)内求出跨越中点的最大子数组。此问题并非原问题规模更小的实例，因为它加入了限制——求出的子数组必须跨越中点。如图 4-4(b)所示，任何跨越中点的子数组都由两个子数组 $A[i..mid]$ 和 $A[mid+1..j]$ 组成，其中 $low \leqslant i \leqslant mid$ 且 $mid < j \leqslant high$。因此，我们只需找出形如 $A[i..mid]$ 和 $A[mid+1..j]$ 的最大子数组，然后将其合并即可。过程 FIND-MAX-CORSSING-SUBARRAY 接收数组 A 和下标 low、mid 和 $high$ 为输入，返回一个下标元组划定跨越中点的最大子数组的边界，并返回最大子数组中值的和。

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
 1  left-sum = −∞
 2  sum = 0
 3  for i = mid downto low
 4      sum=sum+A[i]
 5      if sum > left-sum
 6          left-sum = sum
 7          max-left = i
 8  right-sum = −∞
 9  sum = 0
10  for j = mid + 1 to high
```

```
11      sum=sum + A[j]
12      if sum > right-sum
13          right-sum = sum
14          max-right = j
15  return (max-left, max-right, left-sum + right-sum)
```

71

此过程的工作方式如下所述。第1～7行求出左半部$A[low..mid]$的最大子数组。由于此子数组必须包含$A[mid]$，第3～7行的**for**循环的循环变量i是从mid开始，递减直至达到low，因此，它所考察的每个子数组都具有$A[i..mid]$的形式。第1～2行初始化变量$left\text{-}sum$和sum，前者保存目前为止找到的最大和，后者保存$A[i..mid]$中所有值的和。每当第5行找到一个子数组$A[i..mid]$的和大于$left\text{-}sum$时，我们在第6行将$left\text{-}sum$更新为这个子数组的和，并在第7行更新变量$max\text{-}left$来记录当前下标i。第8～14求右半部$A[mid+1..high]$的最大子数组，过程与左半部类似。此处，第10～14行的**for**循环的循环变量j是从$mid+1$开始，递增直至达到$high$，因此，它所考察的每个子数组都具有$A[mid+1..j]$的形式。最后，第15行返回下标$max\text{-}left$和$max\text{-}right$，划定跨越中点的最大子数组的边界，并返回子数组$A[max\text{-}left..max\text{-}right]$的和$left\text{-}sum+right\text{-}sum$。

如果子数组$A[low..high]$包含n个元素(即$n=high-low+1$)，则调用FIND-MAX-CROSSING-SUBARRAY(A，low，mid，$high$)花费$\Theta(n)$时间。由于两个**for**循环的每次迭代花费$\Theta(1)$时间，我们只需统计一共执行了多少次迭代。第3～7行的**for**循环执行了$mid-low+1$次迭代，第10～14行的**for**循环执行了$high-mid$次迭代，因此总循环迭代次数为

$$(mid-low+1)+(high-mid)=high-low+1=n$$

有了一个线性时间的FIND-MAX-CROSSING-SUBARRAY在手，我们就可以设计求解最大子数组问题的分治算法的伪代码了：

```
FIND-MAXIMUM-SUBARRAY(A, low, high)
 1  if high == low
 2      return (low, high, A[low])              // base case: only one element
 3  else mid=⌊(low+high)/2⌋
 4      (left-low, left-high, left-sum) =
            FIND-MAXIMUM-SUBARRAY(A, low, mid)
 5      (right-low, right-high, right-sum) =
            FIND-MAXIMUM-SUBARRAY(A, mid+1, high)
 6      (cross-low, cross-high, cross-sum) =
            FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
 7      if left-sum ≥ right-sum and left-sum ≥ cross-sum
 8          return (left-low, left-high, left-sum)
 9      elseif rightr-sum ≥ left-sum and right-sum ≥ cross-sum
10          return (right-low, right-high, right-sum)
11      else return (cross-low, cross-high, cross-sum)
```

72

初始调用FIND-MAXIMUM-SUBARRAY(A，1，$A.length$)会求出$A[1..n]$的最大子数组。

与FIND-MAX-CROSSING-SUBARRAY相似，递归过程FIND-MAXIMUM-SUBARRAY返回一个下标元组，划定了最大子数组的边界，同时返回最大子数组中的值之和。第1行测试基本情况，即子数组只有一个元素的情况。在此情况下，子数组只有一个子数组——它自身，因此第2行返回一个下标元组，开始和结束下标均指向唯一的元素，并返回此元素的值作为最大和。第3～11行处理递归情况。第3行划分子数组，计算中点下标mid。我们称子数组$A[low..mid]$为**左子数组**，$A[mid+1..high]$为**右子数组**。因为我们知道子数组$A[low..high]$至少包含两个元

素，则左、右两个子数组各至少包含一个元素。第 4 行和第 5 行分别递归地求解左右子数组中的最大子数组。第 6～11 行完成合并工作。第 6 行求跨越中点的最大子数组(回忆一下，第 6 行求解的子问题并非原问题的规模更小的实例，因为我们将其看做合并部分)。第 7 行检测最大和子数组是否在左子数组中，若是，第 8 行返回此子数组。否则，第 9 行检测最大和子数组是否在右子数组中，若是，第 10 行返回此子数组。如果左、右子数组均不包含最大子数组，则最大子数组必然跨越中点，第 11 行将其返回。

分治算法的分析

接下来，我们建立一个递归式来描述递归过程 FIND-MAXIMUM-SUBARRAY 的运行时间。如 2.3.2 节中分析归并排序那样，对问题进行简化，假设原问题的规模为 2 的幂，这样所有子问题的规模均为整数。我们用 $T(n)$表示 FIND-MAXIMUM-SUBARRAY 求解 n 个元素的最大子数组的运行时间。首先，第 1 行花费常量时间。对于 $n=1$ 的基本情况，也很简单：第 2 行花费常量时间，因此，

$$T(1) = \Theta(1) \tag{4.5}$$

当 $n>1$ 时为递归情况。第 1 行和第 3 行花费常量时间。第 4 行和第 5 行求解的子问题均为 $n/2$ 个元素的子数组(假定原问题规模为 2 的幂，保证了 $n/2$ 为整数)，因此每个子问题的求解时间为 $T(n/2)$。因为我们需要求解两个子问题——左子数组和右子数组，因此第 4 行和第 5 行给
73 总运行时间增加了 $2T(n/2)$。而我们前面已经看到，第 6 行调用 FIND-MAX-CROSSING-SUBARRAY 花费 $\Theta(n)$时间。第 7～11 行仅花费 $\Theta(1)$时间。因此，对于递归情况，我们有

$$T(n) = \Theta(1) + 2T(n/2) + \Theta(n) + \Theta(1) = 2T(n/2) + \Theta(n) \tag{4.6}$$

组合式(4.5)和式(4.6)，我们得到 FIND-MAXIMUM-SUBARRAY 运行时间 $T(n)$的递归式：

$$T(n) = \begin{cases} \Theta(1) & \text{若 } n = 1 \\ 2T(n/2) + \Theta(n) & \text{若 } n > 1 \end{cases} \tag{4.7}$$

此递归式与式(4.1)归并排序的递归式一样。我们在 4.5 节将看到用主方法求解此递归式，其解为 $T(n)=\Theta(n\lg n)$。你也可以重新回顾一下图 2-5 中的递归树，来理解为什么解是 $T(n)=\Theta(n\lg n)$。

因此，我们看到利用分治方法得到了一个渐近复杂性优于暴力求解方法的算法。通过归并排序和本节的最大子数组问题，我们开始对分治方法的强大能力有了一些了解。有时，对某个问题，分治方法能给出渐近最快的算法，而其他时候，我们(不用分治方法)甚至能做得更好。如练习 4.1-5 所示，最大子数组问题实际上存在一个线性时间的算法，并未使用分治方法。

练习

4.1-1 当 A 的所有元素均为负数时，FIND-MAXIMUM-SUBARRAY 返回什么？

4.1-2 对最大子数组问题，编写暴力求解方法的伪代码，其运行时间应该为 $\Theta(n^2)$。

4.1-3 在你的计算机上实现最大子数组问题的暴力算法和递归算法。请指出多大的问题规模 n_0 是性能交叉点——从此之后递归算法将击败暴力算法？然后，修改递归算法的基本情况——当问题规模小于 n_0 时采用暴力算法。修改后，性能交叉点会改变吗？

74 **4.1-4** 假定修改最大子数组问题的定义，允许结果为空子数组，其和为 0。你应该如何修改现有算法，使它们能允许空子数组为最终结果？

4.1-5 使用如下思想为最大子数组问题设计一个非递归的、线性时间的算法。从数组的左边界开始，由左至右处理，记录到目前为止已经处理过的最大子数组。若已知 $A[1..j]$的最大子数组，基于如下性质将解扩展为 $A[1..j+1]$的最大子数组：$A[1..j+1]$的最大子数组要么是 $A[1..j]$的最大子数组，要么是某个子数组 $A[i..j+1](1\leqslant i\leqslant j+1)$。在已知 $A[1..j]$的最大子数组的情况下，可以在线性时间内找出形如 $A[i..j+1]$的最大子数组。

4.2 矩阵乘法的 Strassen 算法

如果你以前曾经接触过矩阵，可能了解如何进行矩阵乘法(否则，请阅读 D.1 节)。若 $A=(a_{ij})$和 $B=(b_{ij})$是 $n\times n$ 的方阵，则对 i，$j=1$，2，…，n，定义乘积 $C=A\cdot B$ 中的元素 c_{ij} 为：

$$c_{ij}=\sum_{k=1}^{n}a_{ik}\cdot b_{kj} \tag{4.8}$$

我们需要计算 n^2 个矩阵元素，每个元素是 n 个值的和。下面过程接收 $n\times n$ 矩阵 A 和 B，返回它们的乘积——$n\times n$ 矩阵 C。假设每个矩阵都有一个属性 *rows*，给出矩阵的行数。

```
SQUARE-MATRIX-MULTIPLY(A, B)
1  n = A.rows
2  let C be a new n×n matrix
3  for i = 1 to n
4      for j = 1 to n
5          c_ij = 0
6          for k = 1 to n
7              c_ij = c_ij + a_ik · b_kj
8  return C
```

过程 SQUARE-MATRIX-MULTIPLY 工作过程如下。第 3～7 行的 **for** 循环计算每行中的元素，在第 i 行中，第 4～7 行的 **for** 循环计算每列中的每个元素 c_{ij}。第 5 行将 c_{ij} 初始化为 0，开始公式(4.8)中的求和计算，第 6～7 行的 **for** 循环的每步迭代将公式(4.8)中的一项累加进来。 75

由于三重 **for** 循环的每一重都恰好执行 n 步，而第 7 行每次执行都花费常量时间，因此过程 SQUARE-MATRIX-MULTIPLY 花费 $\Theta(n^3)$时间。

你最初可能认为任何矩阵乘法都要花费 $\Omega(n^3)$时间，因为矩阵乘法的自然定义就需要进行这么多次的标量乘法。但这是错误的：我们有方法在 $o(n^3)$时间内完成矩阵乘法。在本节中，我们将看到 Strassen 的著名 $n\times n$ 矩阵相乘的递归算法。我们将在 4.5 节证明其运行时间为 $\Theta(n^{\lg 7})$。由于 lg7 在 2.80 和 2.81 之间，因此，Strassen 算法的运行时间为 $O(n^{2.81})$，渐近复杂性优于简单的 SQUARE-MATRIX-MULTIPLY 过程。

一个简单的分治算法

为简单起见，当使用分治算法计算矩阵积 $C=A\cdot B$ 时，假定三个矩阵均为 $n\times n$ 矩阵，其中 n 为 2 的幂。我们做出这个假设是因为在每个分解步骤中，$n\times n$ 矩阵都被划分为 4 个 $n/2\times n/2$ 的子矩阵，如果假定 n 是 2 的幂，则只要 $n\geqslant 2$ 即可保证子矩阵规模 $n/2$ 为整数。

假定将 A、B 和 C 均分解为 4 个 $n/2\times n/2$ 的子矩阵：

$$A=\begin{bmatrix}A_{11} & A_{12}\\ A_{21} & A_{22}\end{bmatrix},\quad B=\begin{bmatrix}B_{11} & B_{12}\\ B_{21} & B_{22}\end{bmatrix},\quad C=\begin{bmatrix}C_{11} & C_{12}\\ C_{21} & C_{22}\end{bmatrix} \tag{4.9}$$

因此可以将公式 $C=A\cdot B$ 改写为：

$$\begin{bmatrix}C_{11} & C_{12}\\ C_{21} & C_{22}\end{bmatrix}=\begin{bmatrix}A_{11} & A_{12}\\ A_{21} & A_{22}\end{bmatrix}\cdot\begin{bmatrix}B_{11} & B_{12}\\ B_{21} & B_{22}\end{bmatrix} \tag{4.10}$$

公式(4.10)等价于如下 4 个公式：

$$C_{11}=A_{11}\cdot B_{11}+A_{12}\cdot B_{21} \tag{4.11}$$

$$C_{12}=A_{11}\cdot B_{12}+A_{12}\cdot B_{22} \tag{4.12}$$

$$C_{21}=A_{21}\cdot B_{11}+A_{22}\cdot B_{21} \tag{4.13}$$

$$C_{22}=A_{21}\cdot B_{12}+A_{22}\cdot B_{22} \tag{4.14}$$

每个公式对应两对 $n/2\times n/2$ 矩阵的乘法及 $n/2\times n/2$ 积的加法。我们可以利用这些公式设计一个

76 直接的递归分治算法：

```
SQUARE-MATRIX-MULTIPLY-RECURSIVE(A, B)
 1  n = A.rows
 2  let C be a new n×n matrix
 3  if n==1
 4      c11=a11 · b11
 5  else partition A, B, and C as in equations (4.9)
 6      C11=SQUARE-MATRIX-MULTIPLY-RECURSIVE(A11, B11)
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A12, B21)
 7      C12=SQUARE-MATRIX-MULTIPLY-RECURSIVE(A11, B12)
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A12, B22)
 8      C21=SQUARE-MATRIX-MULTIPLY-RECURSIVE(A21, B11)
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A22, B21)
 9      C22=SQUARE-MATRIX-MULTIPLY-RECURSIVE(A21, B12)
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A22, B22)
10  return C
```

这段伪代码掩盖了一个微妙但重要的实现细节。在第 5 行应该如何分解矩阵？如果我们真的创建 12 个新的 $n/2\times n/2$ 矩阵，将会花费 $\Theta(n^2)$时间复制矩阵元素。实际上，我们可以不必复制元素就能完成矩阵分解，其中的诀窍是使用下标计算。我们可以通过原矩阵的一组行下标和一组列下标来指明一个子矩阵。最终表示子矩阵的方法与表示原矩阵的方法略有不同，这就是我们省略的细节。这种表示方法的好处是，通过下标计算指明子矩阵，执行第 5 行只需 $\Theta(1)$的时间(虽然我们将看到是否通过复制元素来分解矩阵对总渐近运行时间并无影响)。

现在，我们推导出一个递归式来刻画 SQUARE-MATRIX-MULTIPLY-RECURSIVE 的运行时间。令 $T(n)$表示用此过程计算两个 $n\times n$ 矩阵乘积的时间。对 $n=1$ 的基本情况，我们只需进行一次标量乘法(第 4 行)，因此

$$T(1) = \Theta(1) \tag{4.15}$$

当 $n>1$ 时是递归情况。如前文所讨论，在第 5 行使用下标计算来分解矩阵花费 $\Theta(1)$时间。第 6～9 行，我们共 8 次递归调用 SQUARE-MATRIX-MULTIPLY-RECURSIVE。由于每次递归调用完成两个 $n/2\times n/2$ 矩阵的乘法，因此花费时间为 $T(n/2)$，8 次递归调用总时间为 $8T(n/2)$。我们还需要计算第 6～9 行的 4 次矩阵加法。每个矩阵包含 $n^2/4$ 个元素，因此，每次矩阵加法花
77 费 $\Theta(n^2)$时间。由于矩阵加法的次数是常数，第 6～9 行进行矩阵加法的总时间为 $\Theta(n^2)$(这里我们仍然使用下标计算方法将矩阵加法的结果放置于矩阵 C 的正确位置，由此带来的额外开销为每个元素 $\Theta(1)$时间)。因此，递归情况的总时间为分解时间、递归调用时间及矩阵加法时间之和：

$$T(n) = \Theta(1) + 8T(n/2) + \Theta(n^2) = 8T(n/2) + \Theta(n^2) \tag{4.16}$$

注意，如果通过复制元素来实现矩阵分解，额外开销为 $\Theta(n^2)$，递归式不会发生改变，只是总运行时间将会提高常数倍。

组合公式(4.15)和公式(4.16)，我们得到 SQUARE-MATRIX-MULTIPLY-RECURSIVE 运行时间的递归式：

$$T(n) = \begin{cases} \Theta(1) & 若\ n = 1 \\ 8T(n/2) + \Theta(n^2) & 若\ n > 1 \end{cases} \tag{4.17}$$

我们在 4.5 节将会看到利用主方法求解递归式(4.17)，得到的解为 $T(n)=\Theta(n^3)$。因此，简单的分治算法并不优于直接的 SQUARE-MATRIX-MULTIPLY 过程。

在继续介绍 Strassen 算法之前，让我们先回顾一下公式(4.16)的几个组成部分都是从何而来

的。用下标计算方法分解每个 $n\times n$ 矩阵花费 $\Theta(1)$时间，但有两个矩阵需要分解。虽然你可能认为分解两个矩阵需要 $\Theta(2)$时间，但实际上 Θ 符号中已经包含常数 2 在内了。假定每个矩阵包含 k 个元素，则两个矩阵相加需花费 $\Theta(k)$时间。由于每个矩阵包含 $n^2/4$ 个元素，每次加法花费 $\Theta(n^2/4)$时间。但是同样，Θ 符号已经包含常数因子 1/4，因此，两个 $n/2\times n/2$ 矩阵相加花费 $\Theta(n^2)$时间。我们需要进行 4 次矩阵加法，再次，我们并不说花费了 $\Theta(4n^2)$时间，而是 $\Theta(n^2)$时间。(当然，你可能发现我们可以说 4 次矩阵加法花费了 $\Theta(4n^2/4)$时间，而 $4n^2/4=n^2$，但此处的要点是 Θ 符号已经包含了常数因子，无论怎样的常数因子均可省略。)因此，我们最终得到两项 $\Theta(n^2)$，可以将它们合二为一。

但是，当分析 8 次递归调用时，就不能简单省略常数因子 8 了。换句话说，我们必须说递归调用共花费 $8T(n/2)$时间，而不是 $T(n/2)$时间。至于这是为什么，你可以回顾一下图 2-5 中的递归树，它对应递归式(2.1)(与递归式(4.7)相同)，其递归情况为 $T(n)=2T(n/2)+\Theta(n)$。因子 2 决定了树中每个结点有几个孩子结点，进而决定了树的每一层为总和贡献了多少项。如果省略 78
公式(4.16)中的因子 8 或递归式(4.1)中的因子 2，递归树就变为线性结构，而不是“茂盛的”了，树的每一层只为总和贡献了一项。

因此，切记，虽然渐近符号包含了常数因子，但递归符号(如 $T(n/2)$)并不包含。

Strassen 方法

Strassen 算法的核心思想是令递归树稍微不那么茂盛一点儿，即只递归进行 7 次而不是 8 次 $n/2\times n/2$ 矩阵的乘法。减少一次矩阵乘法带来的代价可能是额外几次 $n/2\times n/2$ 矩阵的加法，但只是常数次。与前文一样，当建立递归式刻画运行时间时，常数次矩阵加法被 Θ 符号包含在内。

Strassen 算法不是那么直观(这可能是本书陈述最不充分的地方了)。它包含 4 个步骤：

1. 按公式(4.9)将输入矩阵 A、B 和输出矩阵 C 分解为 $n/2\times n/2$ 的子矩阵。采用下标计算方法，此步骤花费 $\Theta(1)$时间，与 SQUARE-MATRIX-MULTIPLY-RECURSIVE 相同。

2. 创建 10 个 $n/2\times n/2$ 的矩阵 S_1，S_2，…，S_{10}，每个矩阵保存步骤 1 中创建的两个子矩阵的和或差。花费时间为 $\Theta(n^2)$。

3. 用步骤 1 中创建的子矩阵和步骤 2 中创建的 10 个矩阵，递归地计算 7 个矩阵积 P_1，P_2，…，P_7。每个矩阵 P_i 都是 $n/2\times n/2$ 的。

4. 通过 P_i 矩阵的不同组合进行加减运算，计算出结果矩阵 C 的子矩阵 C_{11}，C_{12}，C_{21}，C_{22}。花费时间 $\Theta(n^2)$。

我们稍后会看到步骤 2～4 的细节，但现在可以建立 Strassen 算法的运行时间递归式。假定一旦矩阵规模从 n 变为 1，就进行简单的标量乘法计算，正如 SQUARE-MATRIX-MULTIPLY-RECURSIVE 的第 4 行那样。当 $n>1$ 时，步骤 1、2 和 4 共花费 $\Theta(n^2)$时间，步骤 3 要求进行 7 次 $n/2\times n/2$ 矩阵的乘法。因此，我们得到如下描述 Strassen 算法运行时间 $T(n)$的递归式：

$$T(n)=\begin{cases}\Theta(1) & \text{若 } n=1\\ 7T(n/2)+\Theta(n^2) & \text{若 } n>1\end{cases} \tag{4.18}$$

79

我们用常数次矩阵乘法的代价减少了一次矩阵乘法。一旦我们理解了递归式及其解，就会看到这种交换确实能带来更低的渐近运行时间。利用 4.5 节的主方法，可以求出递归式(4.18)的解为 $T(n)=\Theta(n^{\lg 7})$。

我们现在来介绍 Strassen 算法的细节。在步骤 2 中，创建如下 10 个矩阵：

$$S_1=B_{12}-B_{22}$$

$$S_2=A_{11}+A_{12}$$

$$S_3 = A_{21} + A_{22}$$
$$S_4 = B_{21} - B_{11}$$
$$S_5 = A_{11} + A_{22}$$
$$S_6 = B_{11} + B_{22}$$
$$S_7 = A_{12} - A_{22}$$
$$S_8 = B_{21} + B_{22}$$
$$S_9 = A_{11} - A_{21}$$
$$S_{10} = B_{11} + B_{12}$$

由于必须进行 10 次 $n/2 \times n/2$ 矩阵的加减法，因此，该步骤花费 $\Theta(n^2)$时间。

在步骤 3 中，递归地计算 7 次 $n/2 \times n/2$ 矩阵的乘法，如下所示：

$$P_1 = A_{11} \cdot S_1 = A_{11} \cdot B_{12} - A_{11} \cdot B_{22}$$
$$P_2 = S_2 \cdot B_{22} = A_{11} \cdot B_{22} + A_{12} \cdot B_{22}$$
$$P_3 = S_3 \cdot B_{11} = A_{21} \cdot B_{11} + A_{22} \cdot B_{11}$$
$$P_4 = A_{22} \cdot S_4 = A_{22} \cdot B_{21} - A_{22} \cdot B_{11}$$
$$P_5 = S_5 \cdot S_6 = A_{11} \cdot B_{11} + A_{11} \cdot B_{22} + A_{22} \cdot B_{11} + A_{22} \cdot B_{22}$$
$$P_6 = S_7 \cdot S_8 = A_{12} \cdot B_{21} + A_{12} \cdot B_{22} - A_{22} \cdot B_{21} - A_{22} \cdot B_{22}$$
$$P_7 = S_9 \cdot S_{10} = A_{11} \cdot B_{11} + A_{11} \cdot B_{12} - A_{21} \cdot B_{11} - A_{21} \cdot B_{12}$$

注意，上述公式中，只有中间一列的乘法是真正需要计算的。右边这列只是用来说明这些乘积与步骤 1 创建的原始子矩阵之间的关系。

步骤 4 对步骤 3 创建的 P_i 矩阵进行加减法运算，计算出 C 的 4 个 $n/2 \times n/2$ 的子矩阵，首先，

80

$$C_{11} = P_5 + P_4 - P_2 + P_6$$

利用每个 P_i 的展开式展开等式右部，每个 P_i 的展开式位于单独一行，并将可以消去的项垂直对齐，我们可以看到 C_{11}等于

$$\begin{array}{rrrr}
A_{11} \cdot B_{11} & + A_{11} \cdot B_{22} & + A_{22} \cdot B_{11} & + A_{22} \cdot B_{22} \\
 & - A_{22} \cdot B_{11} & & + A_{22} \cdot B_{21} \\
- A_{11} \cdot B_{22} & & & - A_{12} \cdot B_{22} \\
 & & - A_{22} \cdot B_{22} - A_{22} \cdot B_{21} & + A_{12} \cdot B_{22} + A_{12} \cdot B_{21} \\
\hline
A_{11} \cdot B_{11} & & & + A_{12} \cdot B_{21}
\end{array}$$

与公式(4.11)相同。类似地，令

$$C_{12} = P_1 + P_2$$

则 C_{12}等于

$$\begin{array}{rrr}
A_{11} \cdot B_{12} & - A_{11} \cdot B_{22} & \\
 & + A_{11} \cdot B_{22} & + A_{12} \cdot B_{22} \\
\hline
A_{11} \cdot B_{12} & & + A_{12} \cdot B_{22}
\end{array}$$

与公式(4.12)相同。令

$$C_{21} = P_3 + P_4$$

使 C_{21}等于

$$\begin{array}{rrr}
A_{21} \cdot B_{11} & + A_{22} \cdot B_{11} & \\
 & - A_{22} \cdot B_{11} & + A_{22} \cdot B_{21} \\
\hline
A_{21} \cdot B_{11} & & + A_{22} \cdot B_{21}
\end{array}$$

与公式(4.13)相同。最后，令

$$C_{22} = P_5 + P_1 - P_3 - P_7$$

则 C_{22} 等于

$$\begin{array}{llll} A_{11}\cdot B_{11} + A_{11}\cdot B_{22} + A_{22}\cdot B_{11} + A_{22}\cdot B_{22} & & & \\ -A_{11}\cdot B_{22} & & +A_{11}\cdot B_{12} & \\ & -A_{22}\cdot B_{11} & & -A_{21}\cdot B_{11} \\ -A_{11}\cdot B_{11} & & -A_{11}\cdot B_{12} + A_{21}\cdot B_{11} + A_{21}\cdot B_{12} & \\ \hline & A_{22}\cdot B_{22} & & +A_{21}\cdot B_{12} \end{array}$$

81

与公式(4.14)相同。在步骤4中，共进行了8次 $n/2\times n/2$ 矩阵的加减法，因此花费 $\Theta(n^2)$ 时间。

因此，我们看到由4个步骤构成的Strassen算法，确实生成了正确的矩阵乘积，递归式(4.18)刻画了它的运行时间。由于我们将在4.5节看到此递归式的解为 $T(n)=\Theta(n^{\lg 7})$，Strassen方法的渐近复杂性低于直接的SQUARE-MATRIX-MULTIPLY过程。本章注记会讨论Strassen算法实际应用方面的一些问题。

练习

注意：虽然练习4.2-3、4.2-4和4.2-5是关于Strassen算法的变形的，但你应该先阅读4.5节，然后再尝试求解这几个问题。

4.2-1 使用Strassen算法计算如下矩阵乘法：

$$\begin{bmatrix} 1 & 3 \\ 7 & 5 \end{bmatrix}\begin{bmatrix} 6 & 8 \\ 4 & 2 \end{bmatrix}$$

给出计算过程。

4.2-2 为Strassen算法编写伪代码。

4.2-3 如何修改Strassen算法，使之适应矩阵规模 n 不是2的幂的情况？证明：算法的运行时间为 $\Theta(n^{\lg 7})$。

4.2-4 如果可以用 k 次乘法操作(假定乘法的交换律不成立)完成两个 3×3 矩阵相乘，那么你可以在 $o(n^{\lg 7})$ 时间内完成 $n\times n$ 矩阵相乘，满足这一条件的最大的 k 是多少？此算法的运行时间是怎样的？

4.2-5 V. Pan发现一种方法，可以用132 464次乘法操作完成 68×68 的矩阵相乘，发现另一种方法，可以用143 640次乘法操作完成 70×70 的矩阵相乘，还发现一种方法，可以用155 424次乘法操作完成 72×72 的矩阵相乘。当用于矩阵相乘的分治算法时，上述哪种方法会得到最佳的渐近运行时间？与Strassen算法相比，性能如何？

82

4.2-6 用Strassen算法作为子进程来进行一个 $kn\times n$ 矩阵和一个 $n\times kn$ 矩阵相乘，最快需要花费多长时间？对两个输入矩阵规模互换的情况，回答相同的问题。

4.2-7 设计算法，仅使用三次实数乘法即可完成复数 $a+bi$ 和 $c+di$ 相乘。算法需接收 a、b、c 和 d 为输入，分别生成实部 $ac-bd$ 和虚部 $ad+bc$。

4.3 用代入法求解递归式

我们已经看到如何用递归式刻画分治算法的运行时间，下面将学习如何求解递归式。我们从“代入”法开始。

代入法求解递归式分为两步：

1. 猜测解的形式。

2. 用数学归纳法求出解中的常数，并证明解是正确的。

当将归纳假设应用于较小的值时，我们将猜测的解代入函数，因此得名“代入法”。这种方法很强大，但我们必须能猜出解的形式，以便将其代入。

我们可以用代入法为递归式建立上界或下界。例如，我们确定下面递归式的上界：

$$T(n) = 2T(\lfloor n/2 \rfloor) + n \tag{4.19}$$

该递归式与递归式(4.3)和(4.4)相似。我们猜测其解为 $T(n)=O(n\lg n)$。代入法要求证明，恰当选择常数 $c>0$，可有 $T(n)\leqslant cn\lg n$。首先假定此上界对所有正数 $m<n$ 都成立，特别是对于 $m=\lfloor n/2 \rfloor$，有 $T(\lfloor n/2 \rfloor)\leqslant c\lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)$。将其代入递归式，得到

$$\begin{aligned} T(n) &\leqslant 2(c\lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)) + n \leqslant cn\lg(n/2) + n \\ &= cn\lg n - cn\lg 2 + n \\ &= cn\lg n - cn + n \leqslant cn\lg n \end{aligned}$$

83

其中，只要 $c\geqslant 1$，最后一步都会成立。

数学归纳法要求我们证明解在边界条件下也成立。为证明这一点，我们通常证明对于归纳证明，边界条件适合作为基本情况。对递归式(4.19)，我们必须证明，通过选择足够大的常数 c，可以使得上界 $T(n)\leqslant cn\lg n$ 对边界条件也成立。这一要求有时可能引起问题。例如，为了方便讨论，假设 $T(1)=1$ 是递归式唯一的边界条件。对 $n=1$，边界条件 $T(n)\leqslant cn\lg n$ 推导出 $T(1)\leqslant c1\lg 1=0$，与 $T(1)=1$ 矛盾。因此，我们的归纳证明的基本情况不成立。

我们稍微多付出一点努力，就可以克服这个障碍，对特定的边界条件证明归纳假设成立。例如，在递归式(4.19)中，渐近符号仅要求我们对 $n\geqslant n_0$ 证明 $T(n)\leqslant cn\lg n$，其中 n_0 是我们可以自己选择的常数，我们可以充分利用这一点。我们保留麻烦的边界条件 $T(1)=1$，但将其从归纳证明中移除。为了做到这一点，首先观察到对于 $n>3$，递归式并不直接依赖 $T(1)$。因此，将归纳证明中的基本情况 $T(1)$ 替换为 $T(2)$ 和 $T(3)$，并令 $n_0=2$。注意，我们将递归式的基本情况($n=1$)和归纳证明的基本情况($n=2$ 和 $n=3$)区分开来了。由 $T(1)=1$，从递归式推导出 $T(2)=4$ 和 $T(3)=5$。现在可以完成归纳证明：对某个常数 $c\geqslant 1$，$T(n)\leqslant cn\lg n$，方法是选择足够大的 c，满足 $T(2)\leqslant c2\lg 2$ 和 $T(3)\leqslant c3\lg 3$。事实上，任何 $c\geqslant 2$ 都能保证 $n=2$ 和 $n=3$ 的基本情况成立。对于我们所要讨论的大多数递归式来说，扩展边界条件使归纳假设对较小的 n 成立，是一种简单直接的方法，我们将不再总是显式说明这方面的细节。

做出好的猜测

遗憾的是，并不存在通用的方法来猜测递归式的正确解。猜测解要靠经验，偶尔还需要创造力。幸运的是，你可以使用一些启发式方法帮助你成为一个好的猜测者。你也可以使用递归树来做出好的猜测，我们将在 4.4 节看到这一方法。

如果要求解的递归式与你曾见过的递归式相似，那么猜测一个类似的解是合理的。例如，考虑如下递归式：

$$T(n) = 2T(\lfloor n/2 \rfloor + 17) + n$$

84

看起来很困难，因为在等式右边 T 的参数中增加了“17”。但直观上，增加的这一项不会显著影响递归式的解。当 n 较大时，$\lfloor n/2 \rfloor$和$\lfloor n/2 \rfloor+17$ 的差距不大：都是接近 n 的一半。因此，我们猜测 $T(n)=O(n\lg n)$，你可以使用代入法验证这个猜测是正确的(见练习 4.3-6)。

另一种做出好的猜测的方法是先证明递归式较松的上界和下界，然后缩小不确定的范围。例如，对递归式(4.19)，我们可以从下界 $T(n)=\Omega(n)$ 开始，因为递归式中包含 n 这一项，还可以证明一个初始上界 $T(n)=O(n^2)$。然后，我们可以逐渐降低上界，提升下界，直至收敛到渐近紧确界 $T(n)=\Theta(n\lg n)$。

微妙的细节

有时你可能正确猜出了递归式解的渐近界，但莫名其妙地在归纳证明时失败了。问题常常出在归纳假设不够强，无法证出准确的界。当遇到这种障碍时，如果修改猜测，将它减去一个低阶的项，数学证明常常能顺利进行。

考虑如下递归式：

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1$$

我们猜测解为 $T(n)=O(n)$，并尝试证明对某个恰当选出的常数 c，$T(n)\leqslant cn$ 成立。将我们的猜测代入递归式，得到

$$T(n) \leqslant c\lfloor n/2 \rfloor + c\lceil n/2 \rceil + 1 = cn + 1$$

这并不意味着对任意 c 都有 $T(n)\leqslant cn$。我们可能忍不住尝试猜测一个更大的界，比如 $T(n)=O(n^2)$。虽然从这个猜测也能推出结果，但原来的猜测 $T(n)=O(n)$ 是正确的。然而为了证明它是正确的，我们必须做出更强的归纳假设。

直觉上，我们的猜测是接近正确的：只差一个常数 1，一个低阶项。但是，除非我们证明与归纳假设严格一致的形式，否则数学归纳法还是会失败。克服这个困难的方法是从先前的猜测中减去一个低阶项。新的猜测为 $T(n)\leqslant cn-d$，d 是大于等于 0 的一个常数。我们现在有

$$\begin{aligned} T(n) &\leqslant (c\lfloor n/2 \rfloor - d) + (c\lceil n/2 \rceil - d) + 1 \\ &= cn - 2d + 1 \leqslant cn - d \end{aligned}$$

85

只要 $d\geqslant 1$，此式就成立。与以前一样，我们必须选择足够大的 c 来处理边界条件。

你可能发现减去一个低阶项的想法与直觉是相悖的。毕竟，如果证明上界失败了，就应该将猜测增加而不是减少，更松的界难道不是更容易证明吗？不一定！当利用归纳法证明一个上界时，实际上证明一个更弱的上界可能会更困难一些，因为为了证明一个更弱的上界，我们在归纳证明中也必须使用同样更弱的界。在当前的例子中，当递归式包含超过一个递归项时，将猜测的界减去一个低阶项意味着每次对每个递归项都减去一个低阶项。在上例中，我们减去常数 d 两次，一次是对 $T(\lfloor n/2 \rfloor)$ 项，另一次是对 $T(\lceil n/2 \rceil)$ 项。我们以不等式 $T(n)\leqslant cn-2d+1$ 结束，可以很容易地找到一个 d 值，使得 $cn-2d+1$ 小于等于 $cn-d$。

避免陷阱

使用渐近符号很容易出错。例如，在递归式(4.19)中，我们可能错误地“证明”$T(n)=O(n)$：猜测 $T(n)\leqslant cn$，并论证

$$T(n) \leqslant 2(c\lfloor n/2 \rfloor) + n \leqslant cn + n = O(n) \qquad \Leftarrow \text{错误!!}$$

因为 c 是常数。错误在于我们并未证出与归纳假设严格一致的形式，即 $T(n)\leqslant cn$。因此，当要证明 $T(n)=O(n)$ 时，需要显式地证出 $T(n)\leqslant cn$。

改变变量

有时，一个小的代数运算可以将一个未知的递归式变成你所熟悉的形式。例如，考虑如下递归式：

$$T(n) = 2T(\lfloor \sqrt{n} \rfloor) + \lg n$$

它看起来很困难。但我们可以通过改变变量来简化它。为方便起见，我们不必担心值的舍入误差问题，只考虑 $\sqrt{n}$ 是整数的情形即可。令 $m=\lg n$，得到

$$T(2^m) = 2T(2^{m/2}) + m$$

现在重命名 $S(m)=T(2^m)$，得到新的递归式：

$$S(m) = 2S(m/2) + m$$

86

它与递归式(4.19)非常像。这个新的递归式确实与(4.19)具有相同的解：$S(m)=O(m\lg m)$。再从 $S(m)$ 转换回 $T(n)$，我们得到 $T(n)=T(2^m)=S(m)=O(m\lg m)=O(\lg n \lg\lg n)$。

练习

4.3-1 证明：$T(n)=T(n-1)+n$ 的解为 $O(n^2)$。

4.3-2 证明：$T(n)=T(\lceil n/2\rceil)+1$ 的解为 $O(\lg n)$。

4.3-3 我们看到 $T(n)=2T(\lfloor n/2\rfloor)+n$ 的解为 $O(n\lg n)$。证明 $\Omega(n\lg n)$也是这个递归式的解。从而得出结论：解为 $\Theta(n\lg n)$。

4.3-4 证明：通过做出不同的归纳假设，我们不必调整归纳证明中的边界条件，即可克服递归式(4.19)中边界条件 $T(1)=1$ 带来的困难。

4.3-5 证明：归并排序的“严格”递归式(4.3)的解为 $\Theta(n\lg n)$。

4.3-6 证明：$T(n)=2T(\lfloor n/2\rfloor+17)+n$ 的解为 $O(n\lg n)$。

4.3-7 使用 4.5 节中的主方法，可以证明 $T(n)=4T(n/3)+n$ 的解为 $T(n)=\Theta(n^{\log_3 4})$。说明基于假设 $T(n)\leqslant cn^{\log_3 4}$ 的代入法不能证明这一结论。然后说明如何通过减去一个低阶项完成代入法证明。

4.3-8 使用 4.5 节中的主方法，可以证明 $T(n)=4T(n/2)+n$ 的解为 $T(n)=\Theta(n^2)$。说明基于假设 $T(n)\leqslant cn^2$ 的代入法不能证明这一结论。然后说明如何通过减去一个低阶项完成代入
87 法证明。

4.3-9 利用改变变量的方法求解递归式 $T(n)=3T(\sqrt{n})+\log n$。你的解应该是渐近紧确的。不必担心数值是否是整数。

4.4 用递归树方法求解递归式

虽然你可以用代入法简洁地证明一个解确是递归式的正确解，但想出一个好的猜测可能会很困难。画出递归树，如我们在 2.3.2 节分析归并排序的递归式时所做的那样，是设计好的猜测的一种简单而直接的方法。在**递归树**中，每个结点表示一个单一子问题的代价，子问题对应某次递归函数调用。我们将树中每层中的代价求和，得到每层代价，然后将所有层的代价求和，得到所有层次的递归调用的总代价。

递归树最适合用来生成好的猜测，然后即可用代入法来验证猜测是否正确。当使用递归树来生成好的猜测时，常常需要忍受一点儿“不精确”，因为稍后才会验证猜测是否正确。但如果在画递归树和代价求和时非常仔细，就可以用递归树直接证明解是否正确。在本节中，我们将使用递归树生成好的猜测，并且在 4.6 节中，我们将使用递归树直接证明主方法的基础定理。

我们以递归式 $T(n)=3T(\lfloor n/4\rfloor)+\Theta(n^2)$为例来看一下如何用递归树生成一个好的猜测。首先关注如何寻找解的一个上界。因为我们知道舍入对求解递归式通常没有影响(此处即是我们需要忍受不精确的一个例子)，因此可以为递归式 $T(n)=3T(\lfloor n/4\rfloor)+cn^2$ 创建一棵递归树，其中已将渐近符号改写为隐含的常数系数 $c>0$。

图 4-5 显示了如何从递归式 $T(n)=3T(\lfloor n/4\rfloor)+cn^2$ 构造出递归树。为方便起见，我们假定 n 是 4 的幂(忍受不精确的另一个例子)，这样所有子问题的规模均为正数。图 4-5(a)显示了 $T(n)$，它在图 4-5(b)中扩展为一棵等价的递归树。根结点中的 cn^2 项表示递归调用顶层的代价，根的三棵子树表示规模为 $n/4$ 的子问题所产生的代价。图 4-5(c)显示了进一步构造递归树的过程，将图 4-5(b)中代价为 $T(n/4)$的结点逐一扩展。我们继续扩展树中每个结点，根据递归式确定的关
88 系将其分解为几个组成部分(孩子结点)。

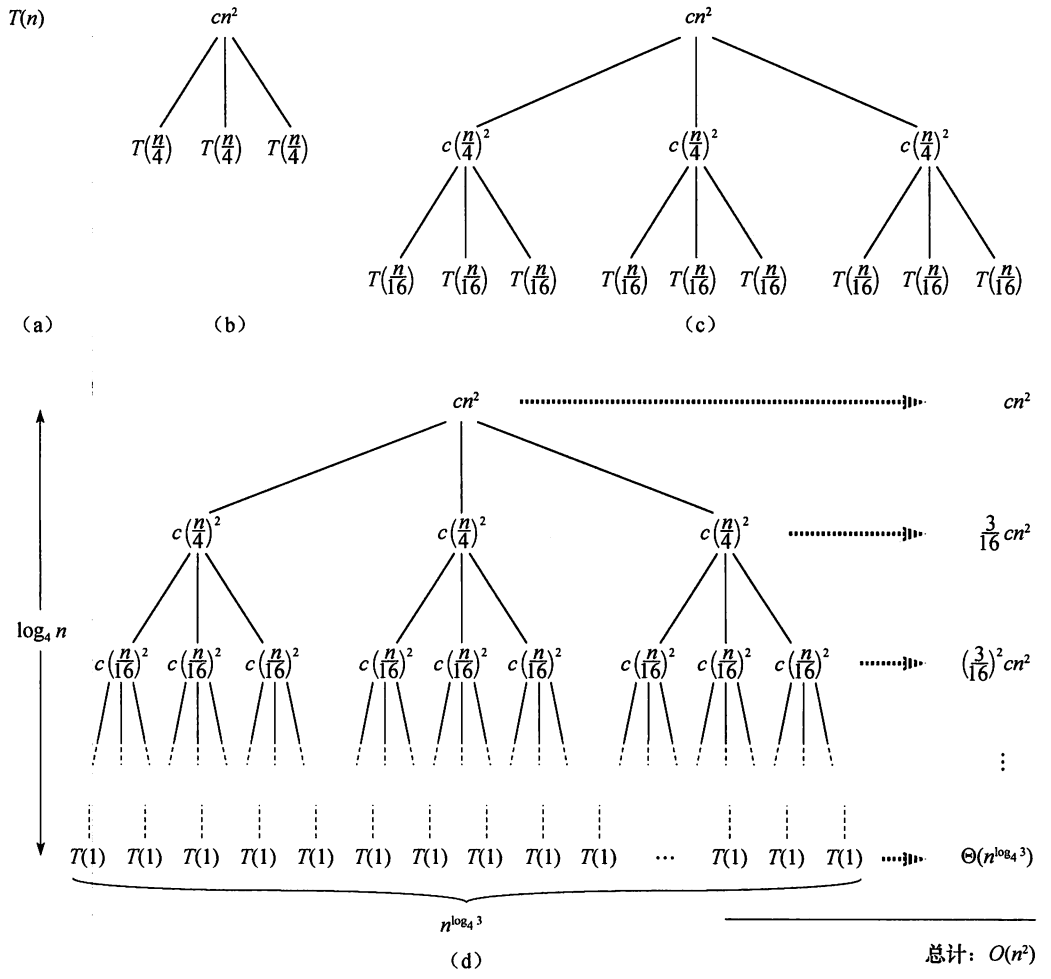


图 4-5 为递归式 $T(n)=3T(\lfloor n/4\rfloor)+cn^2$ 构造递归树。(a)显示了 $T(n)$，在(b)～(d)中逐步扩展为递归树的形式。(d)中显示了扩展完毕的递归树，其高度为 $\log_4 n$(有 $\log_4 n+1$ 层)

89

因为子问题的规模每一步减少为上一步的 1/4，所以最终必然会达到边界条件。那么根结点与距离为 1 的子问题距离多远呢？深度为 i 的结点对应规模为 $n/4^i$ 的子问题。因此，当 $n/4^i=1$，或等价地 $i=\log_4 n$ 时，子问题规模变为 1。因此，递归树有 $\log_4 n+1$ 层(深度为 0，1，2，…，$\log_4 n$)。

接下来确定树的每一层的代价。每层的结点数都是上一层的 3 倍，因此深度为 i 的结点数为 3^i。因为每一层子问题规模都是上一层的 1/4，所以对 $i=0$，1，2，…，$\log_4 n-1$，深度为 i 的每个结点的代价为 $c(n/4^i)^2$。做一下乘法可得，对 $i=0$，1，2，…，$\log_4 n-1$，深度为 i 的所有结点的总代价为 $3^i c(n/4^i)^2=(3/16)^i cn^2$。树的最底层深度为 $\log_4 n$，有 $3^{\log_4 n}=n^{\log_4 3}$ 个结点，每个结点的代价为 $T(1)$，总代价为 $n^{\log_4 3}T(1)$，即 $\Theta(n^{\log_4 3})$，因为假定 $T(1)$ 是常量。

现在我们求所有层次的代价之和，确定整棵树的代价：

$$
\begin{aligned}
T(n) &= cn^2+\frac{3}{16}cn^2+\left(\frac{3}{16}\right)^2 cn^2+\cdots+\left(\frac{3}{16}\right)^{\log_4 n-1} cn^2+\Theta(n^{\log_4 3}) \\
&= \sum_{i=0}^{\log_4 n-1}\left(\frac{3}{16}\right)^i cn^2+\Theta(n^{\log_4 3}) \\
&= \frac{(3/16)^{\log_4 n}-1}{(3/16)-1}cn^2+\Theta(n^{\log_4 3}) \quad \text{(根据公式(A.5))}
\end{aligned}
$$

最后的这个公式看起来有些凌乱，但我们可以再次充分利用一定程度的不精确，并利用无限递减几何级数作为上界。回退一步，应用公式(A.6)，我们得到

$$\begin{aligned}T(n) &= \sum_{i=0}^{\log_4 n-1}\left(\frac{3}{16}\right)^i cn^2+\Theta(n^{\log_4 3}) < \sum_{i=0}^{\infty}\left(\frac{3}{16}\right)^i cn^2+\Theta(n^{\log_4 3})\\ &= \frac{1}{1-(3/16)}cn^2+\Theta(n^{\log_4 3})\\ &= \frac{16}{13}cn^2+\Theta(n^{\log_4 3})\\ &= O(n^2)\end{aligned}$$

这样，对原始的递归式 $T(n)=3T(\lfloor n/4\rfloor)+\Theta(n^2)$，我们推导出了一个猜测 $T(n)=O(n^2)$。在本
90 例中，cn^2 的系数形成了一个递减几何级数，利用公式(A.6)，得出这些系数的和的一个上界——常数 16/13。由于根结点对总代价的贡献为 cn^2，所以根结点的代价占总代价的一个常数比例。换句话说，根结点的代价支配了整棵树的总代价。

实际上，如果 $O(n^2)$确实是递归式的上界(稍后就会证明这一点)，那么它必然是一个紧确界。为什么？因为第一次递归调用的代价为 $\Theta(n^2)$，因此 $\Omega(n^2)$必然是递归式的一个下界。

现在用代入法验证猜测是正确的，即 $T(n)=O(n^2)$是递归式 $T(n)=3T(\lfloor n/4\rfloor)+\Theta(n^2)$的一个上界。我们希望证明 $T(n)\leqslant dn^2$ 对某个常数 $d>0$ 成立。与之前一样，使用常数 $c>0$，我们有

$$\begin{aligned}T(n) &\leqslant 3T(\lfloor n/4\rfloor)+cn^2 \leqslant 3d\lfloor n/4\rfloor^2+cn^2 \leqslant 3d(n/4)^2+cn^2\\ &= \frac{3}{16}dn^2+cn^2 \leqslant dn^2\end{aligned}$$

当 $d\geqslant(16/13)c$ 时，最后一步推导成立。

在另一个更复杂的例子中，图 4-6 显示了如下递归式的递归树：

$$T(n) = T(n/3)+T(2n/3)+O(n)$$

(为简单起见，再次忽略了舍入问题。)与之前一样，令 c 表示 $O(n)$项中的常数因子。对图中显示出的递归树的每个层次，当求代价之和
91 时，我们发现每层的代价均为 cn。从根到叶的最长简单路径是 $n\to(2/3)n\to(2/3)^2n\to\cdots\to1$。由于当 $k=\log_{3/2}n$ 时，$(2/3)^kn=1$，因此树高为 $\log_{3/2}n$。

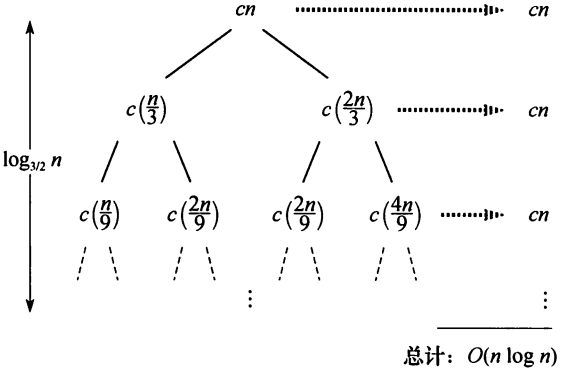


图 4-6 递归式 $T(n)=T(n/3)+T(2n/3)+cn$

直觉上，我们期望递归式的解最多是层数乘以每层的代价，即 $O(cn\log_{3/2}n)=O(n\lg n)$。但图 4-6 仅显示了递归树的顶部几层，并不是递归树中每个层次的代价都是 cn。考虑叶结点的代价。如果递归树是一棵高度为 $\log_{3/2}n$ 的完全二叉树，则叶结点的数量应为 $2^{\log_{3/2}n}=n^{\log_{3/2}2}$。由于每个叶结点的代价为常数，因此所有叶结点的总代价为 $\Theta(n^{\log_{3/2}2})$，由于 $\log_{3/2}n$ 是严格大于 1 的常数，因此叶结点代价总和为 $\Omega(n\lg n)$。但递归树并不是完全二叉树，因此叶结点数量小于 $n^{\log_{3/2}2}$。而且，当从根结点逐步向下走时，越来越多的内结点是缺失的。因此，递归树中靠下的层次对总代价的贡献小于 cn。我们可以计算出所有代价的准确值，但记住我们只是希望得到一个猜测，用于代入法。我们还是忍受一些不精确，尝试证明猜测的上界 $O(n\lg n)$是正确的。

我们确实可以用代入法验证 $O(n\lg n)$是递归式解的一个上界。我们来证明 $T(n)\leqslant dn\lg n$，其中 d 是一个适当的正常数。我们有

$$\begin{aligned}T(n) &\leqslant T(n/3)+T(2n/3)+cn\\ &\leqslant d(n/3)\lg(n/3)+d(2n/3)\lg(2n/3)+cn\end{aligned}$$

$$
\begin{aligned}
&= (d(n/3)\lg n - d(n/3)\lg 3) + (d(2n/3)\lg n - d(2n/3)\lg(3/2)) + cn \\
&= dn\lg n - d((n/3)\lg 3 + (2n/3)\lg(3/2)) + cn \\
&= dn\lg n - d((n/3)\lg 3 + (2n/3)\lg 3 - (2n/3)\lg 2) + cn \\
&= dn\lg n - dn(\lg 3 - 2/3) + cn \\
&\leqslant dn\lg n
\end{aligned}
$$

只要 $d \geqslant c/(\lg 3-(2/3))$。因此，无需对递归树的代价进行更精确的计算。

练习

4.4-1 对递归式 $T(n)=3T(\lfloor n/2 \rfloor)+n$，利用递归树确定一个好的渐近上界，用代入法进行验证。

4.4-2 对递归式 $T(n)=T(n/2)+n^2$，利用递归树确定一个好的渐近上界，用代入法进行验证。 92

4.4-3 对递归式 $T(n)=4T(n/2+2)+n$，利用递归树确定一个好的渐近上界，用代入法进行验证。

4.4-4 对递归式 $T(n)=T(n-1)+1$，利用递归树确定一个好的渐近上界，用代入法进行验证。

4.4-5 对递归式 $T(n)=T(n-1)+T(n/2)+n$，利用递归树确定一个好的渐近上界，用代入法进行验证。

4.4-6 对递归式 $T(n)=T(n/3)+T(2n/3)+cn$，利用递归树论证其解为 $\Omega(n\lg n)$，其中 c 为常数。

4.4-7 对递归式 $T(n)=4T(\lfloor n/2 \rfloor)+cn$（$c$ 为常数），画出递归树，并给出其解的一个渐近紧确界。用代入法进行验证。

4.4-8 对递归式 $T(n)=T(n-a)+T(a)+cn$，利用递归树给出一个渐近紧确解，其中 $a \geqslant 1$ 和 $c>0$是常数。

4.4-9 对递归式 $T(n)=T(\alpha n)+T((1-\alpha)n)+cn$，利用递归树给出一个渐近紧确解，其中 $0<\alpha<1$ 和 $c>0$ 是常数。

4.5 用主方法求解递归式

主方法为如下形式的递归式提供了一种“菜谱”式的求解方法

$$T(n) = aT(n/b) + f(n) \tag{4.20}$$

其中 $a \geqslant 1$ 和 $b>1$ 是常数，$f(n)$是渐近正函数。为了使用主方法，需要牢记三种情况，但随后你就可以很容易地求解很多递归式，通常不需要纸和笔的帮助。 93

递归式(4.20)描述的是这样一种算法的运行时间：它将规模为 n 的问题分解为 a 个子问题，每个子问题规模为 n/b，其中 a 和 b 都是正常数。a 个子问题递归地进行求解，每个花费时间 $T(n/b)$。函数 $f(n)$包含了问题分解和子问题解合并的代价。例如，描述 Strassen 算法的递归式中，$a=7$，$b=2$，$f(n)=\Theta(n^2)$。

从技术的正确性方面看，此递归式实际上并不是良好定义的，因为 n/b 可能不是整数。但将 a 项 $T(n/b)$都替换为 $T(\lfloor n/b \rfloor)$或 $T(\lceil n/b \rceil)$并不会影响递归式的渐近性质(我们将在下一节证明这个断言)。因此，我们通常发现当写下这种形式的分治算法的递归式时，忽略舍入问题是很方便的。

主定理

主方法依赖于下面的定理。

定理 4.1(主定理) 令 $a \geqslant 1$ 和 $b>1$ 是常数，$f(n)$是一个函数，$T(n)$是定义在非负整数上的递归式：

$$T(n) = aT(n/b) + f(n)$$

其中我们将 n/b 解释为 $\lfloor n/b \rfloor$ 或 $\lceil n/b \rceil$。那么 $T(n)$ 有如下渐近界：

1. 若对某个常数 $\varepsilon > 0$ 有 $f(n) = O(n^{\log_b a - \varepsilon})$，则 $T(n) = \Theta(n^{\log_b a})$。

2. 若 $f(n) = \Theta(n^{\log_b a})$，则 $T(n) = \Theta(n^{\log_b a} \lg n)$。

3. 若对某个常数 $\varepsilon > 0$ 有 $f(n) = \Omega(n^{\log_b a + \varepsilon})$，且对某个常数 $c < 1$ 和所有足够大的 n 有 $af(n/b) \leqslant cf(n)$，则 $T(n) = \Theta(f(n))$。 ■

在使用主定理之前，我们花一点儿时间尝试理解一下它的含义。对于三种情况的每一种，我们将函数 $f(n)$ 与函数 $n^{\log_b a}$ 进行比较。直觉上，两个函数较大者决定了递归式的解。若函数 $n^{\log_b a}$ 更大，如情况 1，则解为 $T(n) = \Theta(n^{\log_b a})$。若函数 $f(n)$ 更大，如情况 3，则解为 $T(n) = \Theta(f(n))$。若两个函数大小相当，如情况 2，则乘上一个对数因子，解为 $T(n) = \Theta(n^{\log_b a} \lg n) = \Theta(f(n) \lg n)$。

在此直觉之外，我们需要了解一些技术细节。在第一种情况中，不是 $f(n)$ 小于 $n^{\log_b a}$ 就够了，
94 而是要多项式意义上的小于。也就是说，$f(n)$ 必须渐近小于 $n^{\log_b a}$，要相差一个因子 n^{ε}，其中 ε 是大于 0 的常数。在第三种情况中，不是 $f(n)$ 大于 $n^{\log_b a}$ 就够了，而是要多项式意义上的大于，而且还要满足“正则”条件 $af(n/b) \leqslant cf(n)$。我们将会遇到的多项式界的函数中，多数都满足此条件。

注意，这三种情况并未覆盖 $f(n)$ 的所有可能性。情况 1 和情况 2 之间有一定间隙，$f(n)$ 可能小于 $n^{\log_b a}$ 但不是多项式意义上的小于。类似地，情况 2 和情况 3 之间也有一定间隙，$f(n)$ 可能大于 $n^{\log_b a}$ 但不是多项式意义上的大于。如果函数 $f(n)$ 落在这两个间隙中，或者情况 3 中要求的正则条件不成立，就不能使用主方法来求解递归式。

使用主方法

使用主方法很简单，我们只需确定主定理的哪种情况成立，即可得到解。

我们先看下面这个例子

$$T(n) = 9T(n/3) + n$$

对于这个递归式，我们有 $a = 9$，$b = 3$，$f(n) = n$，因此 $n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$。由于 $f(n) = O(n^{\log_3 9 - \varepsilon})$，其中 $\varepsilon = 1$，因此可以应用主定理的情况 1，从而得到解 $T(n) = \Theta(n^2)$。

现在考虑

$$T(n) = T(2n/3) + 1$$

其中 $a = 1$，$b = 3/2$，$f(n) = 1$，因此 $n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$。由于 $f(n) = \Theta(n^{\log_b a}) = \Theta(1)$，因此应用情况 2，从而得到解 $T(n) = \Theta(\lg n)$。

对于递归式

$$T(n) = 3T(n/4) + n \lg n$$

我们有 $a = 3$，$b = 4$，$f(n) = n \lg n$，因此 $n^{\log_b a} = n^{\log_4 3} = O(n^{0.793})$。由于 $f(n) = \Omega(n^{\log_4 3 + \varepsilon})$，其中 $\varepsilon \approx 0.2$，因此，如果可以证明正则条件成立，即可应用情况 3。当 n 足够大时，对于 $c = 3/4$，$af(n/b) = 3(n/4)\lg(n/4) \leqslant (3/4)n \lg n = cf(n)$。因此，由情况 3，递归式的解为 $T(n) = \Theta(n \lg n)$。

主方法不能用于如下递归式：

$$T(n) = 2T(n/2) + n \lg n$$

虽然这个递归式看起来有恰当的形式：$a = 2$，$b = 2$，$f(n) = n \lg n$，以及 $n^{\log_b a} = n$。你可能错误地
95 认为应该应用情况 3，因为 $f(n) = n \lg n$ 渐近大于 $n^{\log_b a} = n$。问题出在它并不是多项式意义上的大于。对任意正常数 ε，比值 $f(n)/n^{\log_b a} = (n \lg n)/n = \lg n$ 都渐近小于 n^{ε}。因此，递归式落入了情况 2 和情况 3 之间的间隙（此递归式的解参见练习 4.6-2）。

我们利用主方法求解在 4.1 节和 4.2 节中曾见过的递归式(4.7)，

$$T(n) = 2T(n/2) + \Theta(n)$$

它刻画了最大子数组问题和归并排序的分治算法的运行时间（按照通常的做法，我们忽略了递归

式中基本情况的描述)。这里，我们有 $a=2$，$b=2$，$f(n)=\Theta(n)$，因此 $n^{\log_b a}=n^{\log_2 2}=n$。由于 $f(n)=\Theta(n)$，应用情况 2，于是得到解 $T(n)=\Theta(n\lg n)$。

递归式(4.17)，

$$T(n) = 8T(n/2)+\Theta(n^2)$$

它描述了矩阵乘法问题第一个分治算法的运行时间。我们有 $a=8$，$b=2$，$f(n)=\Theta(n^2)$，因此 $n^{\log_b a}=n^{\log_2 8}=n^3$。由于 n^3 多项式意义上大于 $f(n)$(即对 $\varepsilon=1$，$f(n)=O(n^{3-\varepsilon})$)，应用情况 1，解为 $T(n)=\Theta(n^3)$。

最后，我们考虑递归式(4.18)，

$$T(n) = 7T(n/2)+\Theta(n^2)$$

它描述了 Strassen 算法的运行时间。这里，我们有 $a=7$，$b=2$，$f(n)=\Theta(n^2)$，因此 $n^{\log_b a}=n^{\log_2 7}$。将 $\log_2 7$ 改写为 $\lg 7$，由于 $2.80<\lg 7<2.81$，我们知道对 $\varepsilon=0.8$，有 $f(n)=O(n^{\lg 7-\varepsilon})$。再次应用情况 1，我们得到解 $T(n)=\Theta(n^{\lg 7})$。

练习

4.5-1 对下列递归式，使用主方法求出渐近紧确界。

a. $T(n)=2T(n/4)+1$

b. $T(n)=2T(n/4)+\sqrt{n}$

c. $T(n)=2T(n/4)+n$

d. $T(n)=2T(n/4)+n^2$

96

4.5-2 Caesar 教授想设计一个渐近快于 Strassen 算法的矩阵相乘算法。他的算法使用分治方法，将每个矩阵分解为 $n/4\times n/4$ 的子矩阵，分解和合并步骤共花费 $\Theta(n^2)$时间。他需要确定，他的算法需要创建多少个子问题，才能击败 Strassen 算法。如果他的算法创建 a 个子问题，则描述运行时间 $T(n)$的递归式为 $T(n)=aT(n/4)+\Theta(n^2)$。Caesar 教授的算法如果要渐近快于 Strassen 算法，a 的最大整数值应是多少?

4.5-3 使用主方法证明：二分查找递归式 $T(n)=T(n/2)+\Theta(1)$的解是 $T(n)=\Theta(\lg n)$。(二分查找的描述见练习 2.3-5)。

4.5-4 主方法能应用于递归式 $T(n)=4T(n/2)+n^2\lg n$ 吗? 请说明为什么可以或者为什么不可以。给出这个递归式的一个渐近上界。

***4.5-5** 考虑主定理情况 3 的一部分：对某个常数 $c<1$，正则条件 $af(n/b)\leqslant cf(n)$是否成立。给出一个例子，其中常数 $a\geqslant 1$，$b>1$ 且函数 $f(n)$满足主定理情况 3 中除正则条件外的所有条件。

*4.6 证明主定理

本节给出主定理(定理 4.1)的证明。但如果只是为了使用主定理，你不必理解这个证明。

证明分为两部分。第一部分分析主递归式(4.20)，为简单起见，假定 $T(n)$仅定义在 $b(b>1)$的幂上，即仅对 $n=1$，b，b^2，…定义。这一部分给出了为理解主定理是正确的所需的所有直觉知识。第二部分显示了如何将分析扩展到所有正整数 n；这一部分应用了处理向下和向上取整问题的数学技巧。

在本节中，我们有时会稍微滥用渐近符号，用来描述仅仅定义在 b 的幂上的函数的行为。回忆一下，渐近符号的定义要求对所有足够大的数都证明函数的界，而不是仅仅对 b 的幂。因为可以定义出仅仅应用于集合$\{b^i: i=0, 1, 2, \cdots\}$上而不是所有非负数上新的渐近符号，所以这种滥用问题不大。

97

然而，当我们在一个局限的值域上使用渐近符号时，必须时刻小心，避免得到错误的结论。例如，对 n 是 2 的幂的情况证明 $T(n)=O(n)$并不保证 $T(n)=O(n)$。函数 $T(n)$可能是这样定义的：

$$T(n)=\begin{cases}n & \text{若 } n=1,2,4,8,\cdots\\ n^2 & \text{其他}\end{cases}$$

此例中适用于所有 n 值的最佳上界为 $T(n)=O(n^2)$。由于可能导致这种严重后果，在并不绝对清楚应用环境的情况下，永远也不要在一个有限的值域上使用渐近符号。

4.6.1 对 b 的幂证明主定理

主定理证明的第一部分分析主定理的递归式(4.20)：

$$T(n)=aT(n/b)+f(n)$$

假定 n 是 $b(b>1)$的幂，b 不一定是一个整数。我们将分析过程分解为三个引理。第一个引理将求解主递归式的问题归约为一个求和表达式的求值问题。第二个引理确定这个和式的界。第三个引理将前两个引理合二为一，证明 n 为 b 的幂的情况下的主定理。

引理 4.2 *令 $a\geqslant 1$ 和 $b>1$ 是常数，$f(n)$是一个定义在 b 的幂上的非负函数。$T(n)$是定义在 b 的幂上的递归式：*

$$T(n)=\begin{cases}\Theta(1) & \text{若 } n=1\\ aT(n/b)+f(n) & \text{若 } n=b^i\end{cases}$$

其中 i 是正整数。那么

$$T(n)=\Theta(n^{\log_b a})+\sum_{j=0}^{\log_b n-1}a^j f(n/b^j) \tag{4.21}$$

证明 使用图 4-7 中的递归树。树的根结点的代价为 $f(n)$，它有 a 个孩子结点，每个的代价为 $f(n/b)$。(将 a 看做一个整数非常方便，当可视化递归树时尤其如此，但从数学角度并不要求这一点)。每个孩子结点又有 a 个孩子，使得在深度为 2 的层次上有 a^2 个结点，每个的代价为 $f(n/b^2)$。一般地，深度为 j 的层次上有 a^j 个结点，每个的代价为 $f(n/b^j)$。每个叶结点的代价为 $T(1)=\Theta(1)$，深度为 $\log_b n$，因为 $n/b^{\log_b n}=1$。树中共有 $a^{\log_b n}=n^{\log_b a}$ 个叶结点。

98

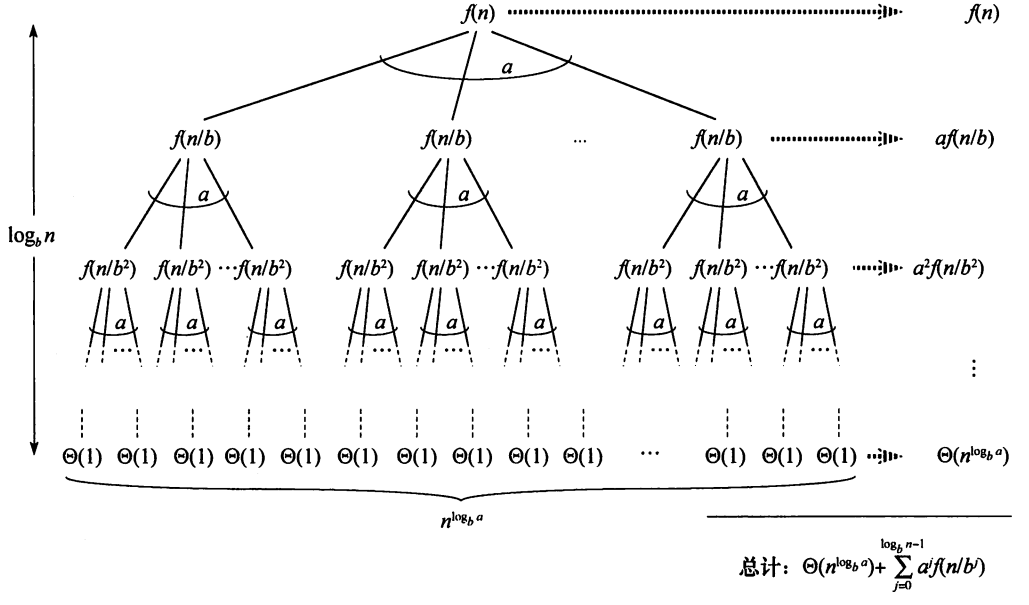


图 4-7 $T(n)=aT(n/b)+f(n)$的递归树。该树是一棵完全 a 叉树，高度为 $\log_b n$，共有 $n^{\log_b a}$ 个叶结点。每层结点的代价显示在右侧，代价和如公式(4.21)所示

我们将图 4-7 所示的递归树中的每层结点的代价求和，得到公式(4.21)。深度为 j 的所有内部结点的代价为 $a^j f(n/b^j)$，所以内部结点的总代价为：

$$\sum_{j=0}^{\log_b n-1} a^j f(n/b^j)$$

在分治算法中，这个和表示分解子问题与合并子问题解的代价。所有叶结点的代价(表示完成所 99
有 $n^{\log_b a}$ 个规模为 1 的子问题的代价)为 $\Theta(n^{\log_b a})$。 ■

从递归树看，主定理的三种情况分别对应以下三种情况：(1)树的总代价由叶结点的代价决定；(2)树的总代价均匀分布在树的所有层次上；(3)树的总代价由根结点的代价决定。

公式(4.21)中的和式描述了分治算法中分解与合并步骤的代价。下一个定理则给出了这个和式增长速度的渐近界。

引理 4.3 令 $a \geqslant 1$ 和 $b > 1$ 是常数，$f(n)$是一个定义在 b 的幂上的非负函数。$g(n)$是定义在 b 的幂上的函数：

$$g(n) = \sum_{j=0}^{\log_b n-1} a^j f(n/b^j) \tag{4.22}$$

对 b 的幂，$g(n)$有如下渐近界：

1. 若对某个常数 $\varepsilon > 0$ 有 $f(n) = O(n^{\log_b a-\varepsilon})$，则 $g(n) = O(n^{\log_b a})$。
2. 若 $f(n) = \Theta(n^{\log_b a})$，则 $g(n) = \Theta(n^{\log_b a} \lg n)$。
3. 若对某个常数 $c < 1$ 和所有足够大的 n 有 $af(n/b) \leqslant cf(n)$，则 $g(n) = \Theta(f(n))$。

证明 对情况 1，我们有 $f(n) = O(n^{\log_b a-\varepsilon})$，这意味着 $f(n/b^j) = O((n/b^j)^{\log_b a-\varepsilon})$。代入公式(4.22)得

$$g(n) = O\left(\sum_{j=0}^{\log_b n-1} a^j \left(\frac{n}{b^j}\right)^{\log_b a-\varepsilon}\right) \tag{4.23}$$

对于 O 符号内的和式，通过提取因子并化简来求它的界，得到一个递增的几何级数：

$$\sum_{j=0}^{\log_b n-1} a^j \left(\frac{n}{b^j}\right)^{\log_b a-\varepsilon} = n^{\log_b a-\varepsilon} \sum_{j=0}^{\log_b n-1} \left(\frac{ab^\varepsilon}{b^{\log_b a}}\right)^j = n^{\log_b a-\varepsilon} \sum_{j=0}^{\log_b n-1} (b^\varepsilon)^j$$

$$= n^{\log_b a-\varepsilon}\left(\frac{b^{\varepsilon \log_b n}-1}{b^\varepsilon-1}\right) = n^{\log_b a-\varepsilon}\left(\frac{n^\varepsilon-1}{b^\varepsilon-1}\right) \quad 100$$

由于 b 和 ε 是常数，因此可以将最后一个表达式重写为 $n^{\log_b a-\varepsilon} O(n^\varepsilon) = O(n^{\log_b a})$。用这个表达式代换公式(4.23)中的和式，得到 $g(n) = O(n^{\log_b a})$，因此情况 1 得证。

由于情况 2 假定 $f(n) = \Theta(n^{\log_b a})$，因此有 $f(n/b^j) = \Theta((n/b^j)^{\log_b a})$。代入公式(4.22)得

$$g(n) = \Theta\left(\sum_{j=0}^{\log_b n-1} a^j \left(\frac{n}{b^j}\right)^{\log_b a}\right) \tag{4.24}$$

采用与情况 1 相同的方式，求出 Θ 符号内和式的界，但这次并未得到一个几何级数，而是发现和式的每一项都是相同的：

$$\sum_{j=0}^{\log_b n-1} a^j \left(\frac{n}{b^j}\right)^{\log_b a} = n^{\log_b a} \sum_{j=0}^{\log_b n-1} \left(\frac{a}{b^{\log_b a}}\right)^j = n^{\log_b a} \sum_{j=0}^{\log_b n-1} 1 = n^{\log_b a} \log_b n$$

用这个表达式替换公式(4.24)中的和式，我们得到

$$g(n) = \Theta(n^{\log_b a} \log_b n) = \Theta(n^{\log_b a} \lg n)$$

情况 2 得证。

情况 3 的证明类似。由于 $f(n)$出现在 $g(n)$的定义(4.22)中，且 $g(n)$的所有项都是非负的，因此可以得出结论：对 b 的幂，$g(n) = \Omega(f(n))$。假定在这个引理中，对某个常数 $c < 1$ 和所有足够大的 n 有 $af(n/b) \leqslant cf(n)$。将这个假设改写为 $f(n/b) \leqslant (c/a)f(n)$并迭代 j 次，得到 $f(n/b^j) \leqslant (c/a)^j f(n)$，或等价地，$a^j f(n/b^j) \leqslant c^j f(n)$，其中假设进行迭代的值足够大。由于最后一个，也

就是最小的值为 n/b^{j-1}，因此假定 n/b^{j-1} 足够大就够了。

代入公式(4.22)并化简，我们得到一个几何级数，但与情况 1 证明中的几何级数不同，这次
101 得到的是递减的几何级数。使用一个 $O(1)$ 项来表示 n 足够大这个假设未覆盖的项：

$$g(n)=\sum_{j=0}^{\log_b n-1} a^j f(n/b^j) \leqslant \sum_{j=0}^{\log_b n-1} c^j f(n)+O(1) \leqslant f(n)\sum_{j=0}^{\infty} c^j+O(1)$$

$$=f(n)\left(\frac{1}{1-c}\right)+O(1)=O(f(n))$$

因为 c 是一个常数。因此可以得到结论：对 b 的幂，$g(n)=\Theta(f(n))$。情况 3 得证，引理证毕。 ■

现在我们来证明 n 为 b 的幂的情况下的主定理。

引理 4.4 *令 $a\geqslant 1$ 和 $b>1$ 是常数，$f(n)$ 是一个定义在 b 的幂上的非负函数。$T(n)$ 是定义在 b 的幂上的递归式：*

$$T(n)=\begin{cases}\Theta(1) & \text{若 } n=1\\ aT(n/b)+f(n) & \text{若 } n=b^i\end{cases}$$

其中 i 是正整数。那么对 b 的幂，$T(n)$ 有如下渐近界：

1. *若对某个常数 $\varepsilon>0$ 有 $f(n)=O(n^{\log_b a-\varepsilon})$，则 $T(n)=\Theta(n^{\log_b a})$。*
2. *若 $f(n)=\Theta(n^{\log_b a})$，则 $T(n)=\Theta(n^{\log_b a}\lg n)$。*
3. *若对某个常数 $\varepsilon>0$，有 $f(n)=\Omega(n^{\log_b a+\varepsilon})$，并且对某个常数 $c<1$ 和所有足够大的 n，有 $af(n/b)\leqslant cf(n)$，则 $T(n)=\Theta(f(n))$。*

证明 利用引理 4.3 中的界对引理 4.2 中的和式(4.21)进行求值。对情况 1，我们有

102 $$T(n)=\Theta(n^{\log_b a})+O(n^{\log_b a})=\Theta(n^{\log_b a})$$

对于情况 2，

$$T(n)=\Theta(n^{\log_b a})+\Theta(n^{\log_b a}\lg n)=\Theta(n^{\log_b a}\lg n)$$

对于情况 3，

$$T(n)=\Theta(n^{\log_b a})+\Theta(f(n))=\Theta(f(n))$$

因为 $f(n)=\Omega(n^{\log_b a+\varepsilon})$。 ■

4.6.2 向下取整和向上取整

为了完成主定理的证明，我们必须将上述分析扩展到主递归式中使用向下取整和向上取整的情况，这样递归式就定义在所有整数上，而非仅仅针对 b 的幂。很容易获得如下递归式的下界：

$$T(n)=aT(\lceil n/b\rceil)+f(n) \tag{4.25}$$

以及如下递归式的上界：

$$T(n)=aT(\lfloor n/b\rfloor)+f(n) \tag{4.26}$$

因为我们可以对第一种情况应用下界 $\lceil n/b\rceil\geqslant n/b$ 来得到所需结果，对第二种情况应用上界 $\lfloor n/b\rfloor\leqslant n/b$。可以使用几乎一样的技术来处理递归式(4.26)的下界和递归式(4.25)的上界，因此我们只给出后一个界的证明。

对图 4-7 中的递归树进行修改，得到图 4-8 中的递归树。当沿着递归树向下时，我们得到如下递归调用的参数序列：

$$\begin{array}{l} n\\ \lceil n/b\rceil\\ \lceil\lceil n/b\rceil/b\rceil\\ \lceil\lceil\lceil n/b\rceil/b\rceil/b\rceil\\ \quad\vdots\end{array}$$

用 n_j 表示序列中第 j 个元素，其中

$$n_j = \begin{cases} n & \text{若 } j = 0 \\ \lceil n_{j-1}/b \rceil & \text{若 } j > 0 \end{cases} \tag{4.27}$$

103

我们的第一个目标是确定 n_k 是常数时的深度 k。利用不等式 $\lceil x \rceil \leqslant x+1$，可得

$$n_0 \leqslant n$$

$$n_1 \leqslant \frac{n}{b} + 1$$

$$n_2 \leqslant \frac{n}{b^2} + \frac{1}{b} + 1$$

$$n_3 \leqslant \frac{n}{b^3} + \frac{1}{b^2} + \frac{1}{b} + 1$$

$$\vdots$$

一般地，我们有

$$n_j \leqslant \frac{n}{b^j} + \sum_{i=0}^{j-1} \frac{1}{b^i} < \frac{n}{b^j} + \sum_{i=0}^{\infty} \frac{1}{b^i} = \frac{n}{b^j} + \frac{b}{b-1}$$

令 $j = \lfloor \log_b n \rfloor$，可得

$$n_{\lfloor \log_b n \rfloor} < \frac{n}{b^{\lfloor \log_b n \rfloor}} + \frac{b}{b-1} < \frac{n}{b^{\log_b n - 1}} + \frac{b}{b-1} = \frac{n}{n/b} + \frac{b}{b-1}$$

$$= b + \frac{b}{b-1} = O(1)$$

因此我们可以看到在深度 $\lfloor \log_b n \rfloor$，问题规模至多是常数。

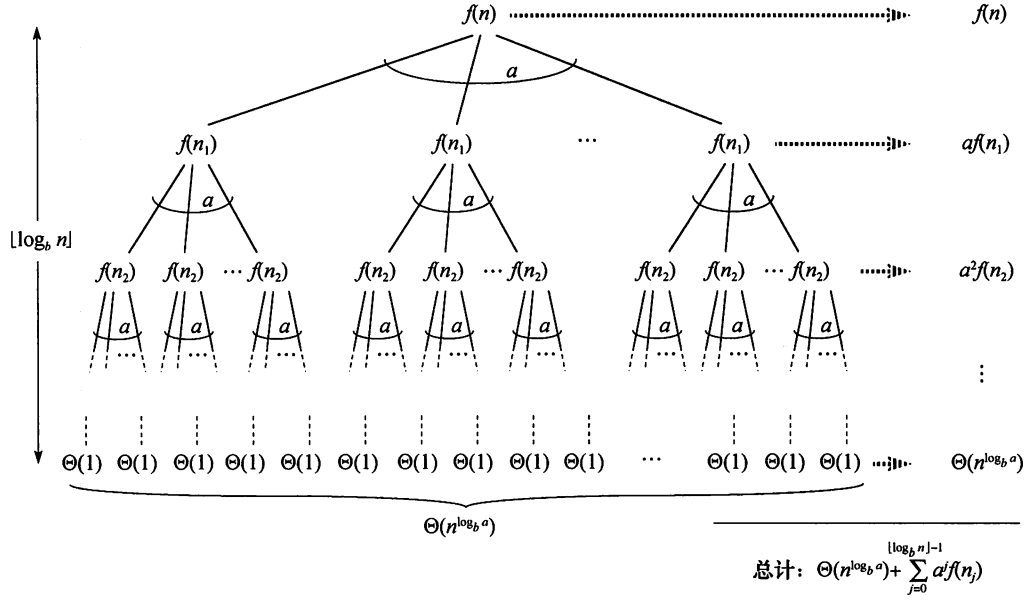


图 4-8 $T(n) = aT(\lceil n/b \rceil) + f(n)$ 的递归树。递归参数 n_j 的定义见公式(4.27)

从图 4-8 可以看出，

$$T(n) = \Theta(n^{\log_b a}) + \sum_{j=0}^{\lfloor \log_b n \rfloor - 1} a^j f(n_j) \tag{4.28}$$

除了 n 为任意整数，未局限为 b 的幂之外，这个公式与公式(4.21)几乎一样。

我们现在可以对公式(4.28)中的和式进行求值

$$g(n)=\sum_{j=0}^{\lfloor \log_b n\rfloor -1} a^j f(n_j) \tag{4.29}$$

方法与引理 4.3 的证明类似。我们从情况 3 开始，如果对 $n>b+b/(b-1)$，$af(\lceil n/b\rceil)\leqslant cf(n)$成立，其中 $c<1$ 是常数，则有 $a^j f(n_j)\leqslant c^j f(n)$。因此，我们可以像引理 4.3 的证明一样来对公式(4.29)的和式进行求值。对于情况 2，我们有 $f(n)=\Theta(n^{\log_b a})$。如果能证明 $f(n_j)=O(n^{\log_b a}/a^j)=O((n/b^j)^{\log_b a})$，则情况 2 的证明直接使用引理 4.3 证明的方法即可。观察到 $j\leqslant\lfloor \log_b n\rfloor$意味着 $b^j/n\leqslant 1$。界 $f(n)=O(n^{\log_b a})$意味着存在常数 $c>0$，使得对所有足够大的 n_j，

104
~
105

$$\begin{aligned}
f(n_j)&\leqslant c\left(\frac{n}{b^j}+\frac{b}{b-1}\right)^{\log_b a}=c\left(\frac{n}{b^j}\left(1+\frac{b^j}{n}\cdot\frac{b}{b-1}\right)\right)^{\log_b a}\\
&=c\left(\frac{n^{\log_b a}}{a^j}\right)\left(1+\left(\frac{b^j}{n}\cdot\frac{b}{b-1}\right)\right)^{\log_b a}\\
&\leqslant c\left(\frac{n^{\log_b a}}{a^j}\right)\left(1+\frac{b}{b-1}\right)^{\log_b a}=O\left(\frac{n^{\log_b a}}{a^j}\right)
\end{aligned}$$

因为 $c(1+b/(b-1))^{\log_b a}$是常量。因此，情况 2 得证。情况 1 的证明几乎是一样的。关键是证明界 $f(n_j)=O((n/b^j)^{\log_b a-\epsilon})$，这部分与情况 2 证明中的对应部分相似，尽管使用的代数方法更复杂些。

现在我们已经对所有整数 n 证明了主定理的上界。下界的证明类似。

练习

***4.6-1** 对 b 是正整数而非任意实数的情况，给出公式(4.27)中 n_j 的简单而准确的表达式。

***4.6-2** 证明：如果 $f(n)=\Theta(n^{\log_b a}\lg^k n)$，其中 $k\geqslant 0$，那么主递归式的解为 $T(n)=\Theta(n^{\log_b a}\lg^{k+1} n)$。为简单起见，假定 n 是 b 的幂。

***4.6-3** 证明：主定理中的情况 3 被过分强调了，从某种意义上来说，对某个常数 $c<1$，正则条件 $af(n/b)\leqslant cf(n)$成立本身就意味着存在常数 $\epsilon>0$，使得 $f(n)=\Omega(n^{\log_b a+\epsilon})$。

106

思考题

4-1 (*递归式例子*) 对下列每个递归式，给出 $T(n)$的渐近上界和下界。假定 $n\leqslant 2$ 时 $T(n)$是常数。给出尽量紧确的界，并验证其正确性。

a. $T(n)=2T(n/2)+n^4$

b. $T(n)=T(7n/10)+n$

c. $T(n)=16T(n/4)+n^2$

d. $T(n)=7T(n/3)+n^2$

e. $T(n)=7T(n/2)+n^2$

f. $T(n)=2T(n/4)+\sqrt{n}$

g. $T(n)=T(n-2)+n^2$

4-2 (*参数传递代价*) 我们有一个贯穿本书的假设——过程调用中的参数传递花费常量时间，即使传递一个 N 个元素的数组也是如此。在大多数系统中，这个假设是成立的，因为传递的是指向数组的指针，而非数组本身。本题讨论三种参数传递策略：

1. 数组通过指针来传递。时间$=\Theta(1)$。
2. 数组通过元素复制来传递。时间$=\Theta(N)$，其中 N 是数组的规模。
3. 传递数组时，只复制过程可能访问的子区域。若子数组 $A[p..q]$被传递，则时间$=\Theta(q-p+1)$。

a. 考虑在有序数组中查找元素的递归二分查找算法(参见练习 2.3-5)。分别给出上述三种参数传递策略下，二分查找最坏情况运行时间的递归式，并给出递归式解的好的上界。

令 N 为原问题的规模，n 为子问题的规模。

b. 对 2.3.1 节的 MERGE-SORT 算法重做(a)。 107

4-3 （更多的递归式例子） 对下列每个递归式，给出 $T(n)$的渐近上界和下界。假定对足够小的 n，$T(n)$是常数。给出尽量紧确的界，并验证其正确性。

a. $T(n)=4T(n/3)+n\lg n$

b. $T(n)=3T(n/3)+n/\lg n$

c. $T(n)=4T(n/2)+n^2\sqrt{n}$

d. $T(n)=3T(n/3-2)+n/2$

e. $T(n)=2T(n/2)+n/\lg n$

f. $T(n)=T(n/2)+T(n/4)+T(n/8)+n$

g. $T(n)=T(n-1)+1/n$

h. $T(n)=T(n-1)+\lg n$

i. $T(n)=T(n-2)+1/\lg n$

j. $T(n)=\sqrt{n}T(\sqrt{n})+n$

4-4 （斐波那契数） 本题讨论递归式(3.22)定义的斐波那契数的性质。我们将使用生成函数技术来求解斐波那契递归式。**生成函数**(又称为**形式幂级数**)$\mathcal{F}$定义为

$$\mathcal{F}(z)=\sum_{i=0}^{\infty}F_i z^i=0+z+z^2+2z^3+3z^4+5z^5+8z^6+13z^7+21z^8+\cdots$$

其中 F_i 为第 i 个斐波那契数。

a. 证明：$\mathcal{F}(z)=z+z\mathcal{F}(z)+z^2\mathcal{F}(z)$。 108

b. 证明：

$$\mathcal{F}(z)=\frac{z}{1-z-z^2}=\frac{z}{(1-\phi z)(1-\hat{\phi}z)}=\frac{1}{\sqrt{5}}\left(\frac{1}{1-\phi z}-\frac{1}{1-\hat{\phi}z}\right)$$

其中

$$\phi=\frac{1+\sqrt{5}}{2}=1.618\,03\cdots$$

$$\hat{\phi}=\frac{1-\sqrt{5}}{2}=-0.618\,03\cdots$$

c. 证明：

$$\mathcal{F}(z)=\sum_{i=0}^{\infty}\frac{1}{\sqrt{5}}(\phi^i-\hat{\phi}^i)z^i$$

d. 利用(c)的结果证明：对 $i>0$，$F_i=\phi^i/\sqrt{5}$，结果舍入到最接近的整数。(提示：观察到 $|\hat{\phi}|<1$。)

4-5 （芯片检测） Diogenes 教授有 n 片可能完全一样的集成电路芯片，原理上可以用来相互检测。教授的测试夹具同时只能容纳两块芯片。当夹具装载上时，每块芯片都检测另一块，并报告它是好是坏。一块好的芯片总能准确报告另一块芯片的好坏，但教授不能信任坏芯片报告的结果。因此，4 种可能的测试结果如下：

芯片 A 的结果	芯片 B 的结果	结　论
B 是好的	A 是好的	两片都是好的，或都是坏的
B 是好的	A 是坏的	至少一块是坏的
B 是坏的	A 是好的	至少一块是坏的
B 是坏的	A 是坏的	至少一块是坏的

a. 证明：如果超过 $n/2$ 块芯片是坏的，使用任何基于这种逐对检测操作的策略，教授都不能确定哪些芯片是好的。假定坏芯片可以合谋欺骗教授。

109

b. 考虑从 n 块芯片中寻找一块好芯片的问题，假定超过 $n/2$ 块芯片是好的。证明：进行 $\lfloor n/2 \rfloor$ 次逐对检测足以将问题规模减半。

c. 假定超过 $n/2$ 块芯片是好的，证明：可以用 $\Theta(n)$ 次逐对检测找出好的芯片。给出描述检测次数的递归式，并求解它。

4-6 (Monge 阵列) 对一个 $m \times n$ 的实数阵列 A，若对所有满足 $1 \leqslant i < k \leqslant m$ 和 $1 \leqslant j < l \leqslant n$ 的 i，j，k 和 l 有

$$A[i,j] + A[k,l] \leqslant A[i,l] + A[k,j]$$

则称 A 是 **Monge 阵列**(Monge array)。换句话说，无论何时选出 Monge 阵列的两行和两列，对于交叉点上的 4 个元素，左上和右下两个元素之和总是小于等于左下和右上元素之和。例如，下面就是一个 Monge 阵列：

$$\begin{array}{rrrrr} 10 & 17 & 13 & 28 & 23 \\ 17 & 22 & 16 & 29 & 23 \\ 24 & 28 & 22 & 34 & 24 \\ 11 & 13 & 6 & 17 & 7 \\ 45 & 44 & 32 & 37 & 23 \\ 36 & 33 & 19 & 21 & 6 \\ 75 & 66 & 51 & 53 & 34 \end{array}$$

a. 证明：一个数组是 Monge 阵列当且仅当对所有 $i=1, 2, \cdots, m-1$ 和 $j=1, 2, \cdots, n-1$，有

$$A[i,j] + A[i+1,j+1] \leqslant A[i,j+1] + A[i+1,j]$$

(提示：对于“当”的部分，分别对行和列使用归纳法。)

b. 下面数组不是 Monge 阵列。改变一个元素使其变成 Monge 阵列。(提示：利用(a)的结果。)

$$\begin{array}{rrrr} 37 & 23 & 22 & 32 \\ 21 & 6 & 7 & 10 \\ 53 & 34 & 30 & 31 \\ 32 & 13 & 9 & 6 \\ 43 & 21 & 15 & 8 \end{array}$$

110

c. 令 $f(i)$ 表示第 i 行的最左最小元素的列下标。证明：对任意 $m \times n$ 的 Monge 阵列，$f(1) \leqslant f(2) \leqslant \cdots \leqslant f(m)$。

d. 下面是一个计算 $m \times n$ 的 Monge 阵列 A 每一行最左最小元素的分治算法的描述：

> 提取 A 的偶数行构造其子矩阵 A'。递归地确定 A' 每行的最左最小元素。然后计算 A 的奇数行的最左最小元素。

解释如何在 $O(m+n)$ 时间内计算 A 的奇数行的最左最小元素(在偶数行的最左最小元素已知的情况下)。

e. 给出(d)中描述的算法的运行时间的递归式。证明其解为 $O(m + n \log m)$。

本章注记

分治作为一种算法设计技术至少可以追溯到 1962 年 Karatsuba 和 Ofman 的一篇文章[194]。但是在这之前，分治技术已经有很好的应用，根据 Heideman、Johnson 和 Burrus 的论文[163]，卡尔·弗雷德里希·高斯在 1805 年设计了第一个快速傅里叶变换算法，而高斯的算法就是将问

题分解为更小的子问题，求解完子问题后将它们的解组合起来。

4.1节中讨论的最大子数组问题是Bently[43，第7章]研究的问题的一个简单变形。

Strassen算法[325]发表于1969年，它的出现引起了很大的轰动。在此之前，很少人敢设想一个算法能渐近快于平凡算法SQUARE-MATRIX-MULTIPLY。矩阵乘法的渐近上界自此被改进了。到目前为止，$n\times n$矩阵相乘的渐近复杂性最优的算法是Coppersmith和Winograd[78]提出的，运行时间为$O(n^{2.376})$。已知的最好的下界显然是$\Omega(n^2)$(这是显然的下界，因为我们必须填写结果矩阵的n^2个元素)。

从实用的角度看，Strassen算法通常并不是解决矩阵乘法的最好选择，原因有4个：

1. 隐藏在Strassen算法运行时间$\Theta(n^{\lg 7})$中的常数因子比过程SQUARE-MATRIX-MULTIPLY的$\Theta(n^3)$时间的常数因子大。

2. 对于稀疏矩阵，专用算法更快。 111

3. Strassen算法的数值稳定性不如SQUARE-MATRIX-MULTIPLY那么好。换句话说，由于计算机计算非整数值时有限的精度，Strassen算法累积的误差比SQUARE-MATRIX-MULTIPLY大。

4. 递归过程中生成的子矩阵消耗存储空间。

后两个原因在1990年左右得到了缓解。Higham[167]显示了数值稳定性上的差异被过分强调了；虽然Strassen算法对某些应用来说数值稳定性太差，但对其他应用来说，它所产生的数值误差还在可接受的范围内。Bailey、Lee和Simon[32]讨论了降低Strassen算法内存需求的技术。

在实际应用中，稠密矩阵的快速乘法程序在矩阵规模超过一个“交叉点”时使用Strassen算法，一旦子问题规模降低到交叉点之下，就切换到一个更简单的方法。交叉点的确切值高度依赖于具体系统。有一些分析统计操作次数，但忽略CPU缓存和流水线的影响，得出的交叉点低至$n=8$(Higham[167])或$n=12$(Huss-Lederman等人[186])。D'Alberto和Nicolau[81]设计了一个自适应方法，在软件包安装完毕后通过基准测试确定交叉点。他们发现，在不同的系统上，交叉点的值从$n=400$到$n=2\,150$变化，而在几个系统中无法找到交叉点。

递归式的研究最早可追溯到1202年李奥纳多·斐波那契的工作，斐波那契数就是以他命名的。A. De Moivre提出了用生成函数(参见思考题4-4)求解递归式的方法。主方法改自Bentley、Haken和Saxe[44]的方法，这篇文章提供了一种扩展方法，在练习4.6-2中已经得到验证。Knuth[209]和Liu[237]展示了如何使用生成函数的方法求解线性递归式。Purdom和Brown的论文[287]及Graham、Knuth和Patashnik的论文[152]包含了递归式求解的进一步讨论。

相对于主方法可求解的分治算法递归式，多名研究者，包括Akra和Bazzi[13]、Roura[299]、Verma[346]及Yap[360]，都给出过更一般的递归式的求解方法。我们介绍一下Akra和Bazzi的结果，这里给出的是Leighton[228]修改后的版本。Akra-Bazzi方法求解如下形式的递归式：

$$T(x)=\begin{cases}\Theta(1) & \text{若 } 1\leqslant x\leqslant x_0\\ \sum_{i=1}^{k}a_iT(b_ix)+f(x) & \text{若 } x>x_0\end{cases}\tag{4.30}$$

其中

- $x\geqslant 1$是一个实数，
- x_0是一个常数，满足对$i=1,2,\cdots,k$，$x_0\geqslant 1/b_i$且$x_0\geqslant 1/(1-b_i)$，
- 对$i=1,2,\cdots,k$，a_i是一个正常数， 112
- 对$i=1,2,\cdots,k$，b_i是一个常数，范围在$0<b_i<1$，
- $k\geqslant 1$是一个整数常数，且

- $f(x)$是一个非负函数，满足**多项式增长条件**：存在正常数 c_1 和 c_2，使得对所有 $x \geqslant 1$，$i=1, 2, \cdots, k$ 以及所有满足 $b_i x \leqslant u \leqslant x$ 的 u，有 $c_1 f(x) \leqslant f(u) \leqslant c_2 f(x)$。（若 $|f'(x)|$ 的上界是 x 的某个多项式，则 $f(x)$满足多项式增长条件。例如，对任意实常数 α 和 β，$f(x)=x^{\alpha} \lg^{\beta} x$ 满足此条件。）

虽然主定理不能应用于 $T(n)=T(\lfloor n/3 \rfloor)+T(\lfloor 2n/3 \rfloor)+O(n)$这样的递归式，但 Akra-Bazzi 方法可以。为了求解递归式(4.30)，我们首先寻找满足 $\sum_{i=1}^{k} a_i b_i^p = 1$ 的实数 p(这样的 p 总是存在的)。那么递归式的解为

$$T(n) = \Theta\left(x^p\left(1+\int_1^x \frac{f(u)}{u^{p+1}}\mathrm{d}u\right)\right)$$

Akra-Bazzi 方法可能有点儿难用，但它可以求解那些子问题划分不均衡的算法的递归式。主方法很容易使用，但只能用于子问题规模相等的情况。

113