算法分析

内容主要来源于《Introduction to Algorithms Third Edition》（算法导论）第2-4章，涉及算法时间复杂度分析、表示法、分治与递归、几种解递归算法时间复杂度的方法——substitution method（代入法）、recurtion-tree method（递归树法）、master method（主方法）。本文预估阅读时间超过2小时，数学推导较多，谨慎阅读。

算法分析与运行时间

算法分析的主要目的是预测运行一个算法所需要的资源。这种资源可以是内存、贷款等，但通常来说最关心的资源是运算时间。本书中使用的算法分析模型是RAM模型，认为算法会以指令方式一个一个执行，不存在并发操作。算法的运行时间（running time），是一个关于输入大小（input size）的函数。Input size虽然经常被认为为数组大小，在某些算法中也可以是比特数（两个整数相乘，以整数位数为输入大小）、结点与边的数量（图）。Running time与运行的原始操作相关，通常假定运行某一行i需要时间c_i，这里的c_i是一个常数。

作者在讲解running time计算的时候用了一个insertation sort（插入排序）的例子，详见P26。

对于不同的输入，可能会有不同的算法运行时间，对于插入排序而言，若数列是已经排序的（升序），运行时间为n的线性函数，若数列是逆向排序的（降序），运行时间为n的二次函数。从而便有了最好、最坏、平均运行时间。其中最好运行时间（best-case running time），是运行的下界，同理最坏运行时间（worst-case running time）是上界。本书一般来说专注于寻找最坏运行时间。平均运行时间涉及到本书的概率分析部分，通常假定所有input size出现的概率是相同的。在实际中可能并非如此，这里涉及到本书的randomized algorithm部分。

在计算运行时间的时候，我们可能会得到类似于$an^2 + bn + c$之类的式子，这里的n代表输入大小，a、b、c代表取决于c_i的常数。我们在表示运算时间的时候，相比于这种式子，更喜欢使用另外一种简化的版本（原文是simplifying abstraction）：增长率（rate of growth）或者增长极（order of growth），只关注最高次项而忽略对最终结果影响较小的低次项与常数。因此，这个运行时间被写作为$\Theta(n^2)$，读作为theta of n-squared。这里用了theta表示法（theta notation），它是一种渐进表示法（asymptotic notation），后面的部分会介绍其他类型的表示方法与它们之间的区别。

渐进表示法与Θ表示法、O表示法、Ω表示法

渐进表示法：
当input size 足够大的时候，它使得只有order of growth与算法运行时间是高度相关的，我们此时研究的是算法的渐进效应（asymptotic efficiency）。即inpuy size无限增长的时候，算法的运行时间会怎么增长。渐进表示法实际上是一种最高次项而忽略对最终结果影响较小的低次项与常数的表示法。除了表示运行时间，它可以被用作来研究算法的空间使用，而且也可以用于研究多种类型的运行时间（最坏运行时间等）。

Θ表示法、O表示法、Ω表示法：
下面这图+下面的注释比较简单地解释了三种表示法，总得来说，Θ表示法给出了运行时间函数的紧界（tight bound）（上界+下界）、O表示法给出了上界（upper bound）、Ω表示法给出了下界（lower bound）。

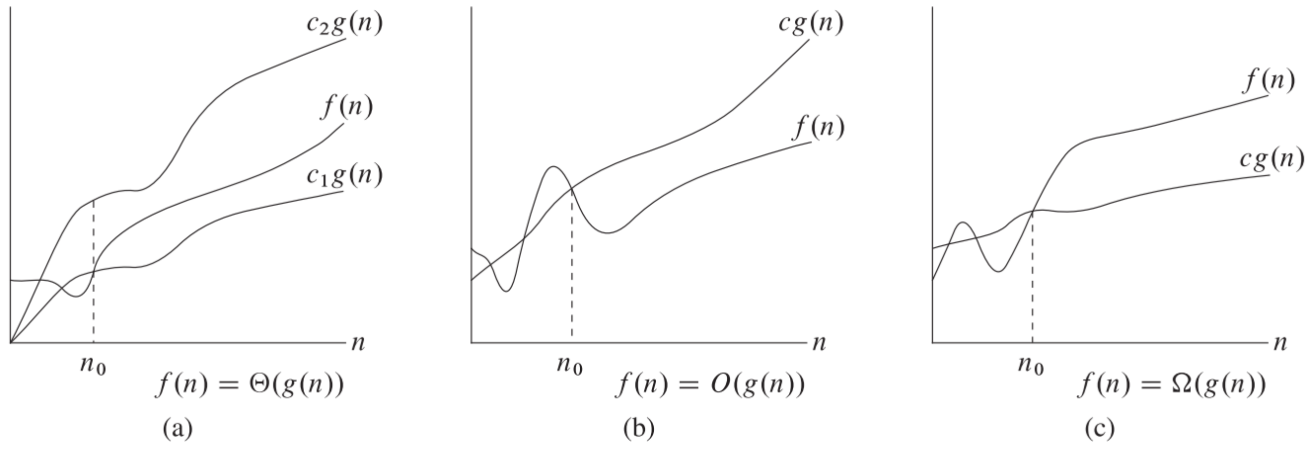


Figure 3.1 Graphic examples of the Θ, O, and Ω notations. In each part, the value of n_0 shown is the minimum possible value; any greater value would also work. **(a)** Θ-notation bounds a function to within constant factors. We write $f(n) = \Theta(g(n))$ if there exist positive constants n_0, c_1, and c_2 such that at and to the right of n_0, the value of $f(n)$ always lies between $c_1g(n)$ and $c_2g(n)$ inclusive. **(b)** O-notation gives an upper bound for a function to within a constant factor. We write $f(n) = O(g(n))$ if there are positive constants n_0 and c such that at and to the right of n_0, the value of $f(n)$ always lies on or below $cg(n)$. **(c)** Ω-notation gives a lower bound for a function to within a constant factor. We write $f(n) = \Omega(g(n))$ if there are positive constants n_0 and c such that at and to the right of n_0, the value of $f(n)$ always lies on or above $cg(n)$.

然后下面是三种方法的具体数学定义，最好也看一看。注意，这些表示法表示的是函数集合（a set of functions），这些函数满足特定的条件，数学定义是以集合形式写的。

Θ-notation

In Chapter 2, we found that the worst-case running time of insertion sort is $T(n) = \Theta(n^2)$. Let us define what this notation means. For a given function $g(n)$, we denote by $\Theta(g(n))$ the *set of functions*

$$\begin{aligned}\Theta(g(n)) = \{f(n) :\ & \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that} \\ & 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0\} .\end{aligned}$$[1]

O-notation

The Θ-notation asymptotically bounds a function from above and below. When we have only an ***asymptotic upper bound***, we use O-notation. For a given function $g(n)$, we denote by $O(g(n))$ (pronounced "big-oh of g of n" or sometimes just "oh of g of n") the set of functions

$$\begin{aligned}O(g(n)) = \{f(n) :\ & \text{there exist positive constants } c \text{ and } n_0 \text{ such that} \\ & 0 \le f(n) \le c g(n) \text{ for all } n \ge n_0\} .\end{aligned}$$

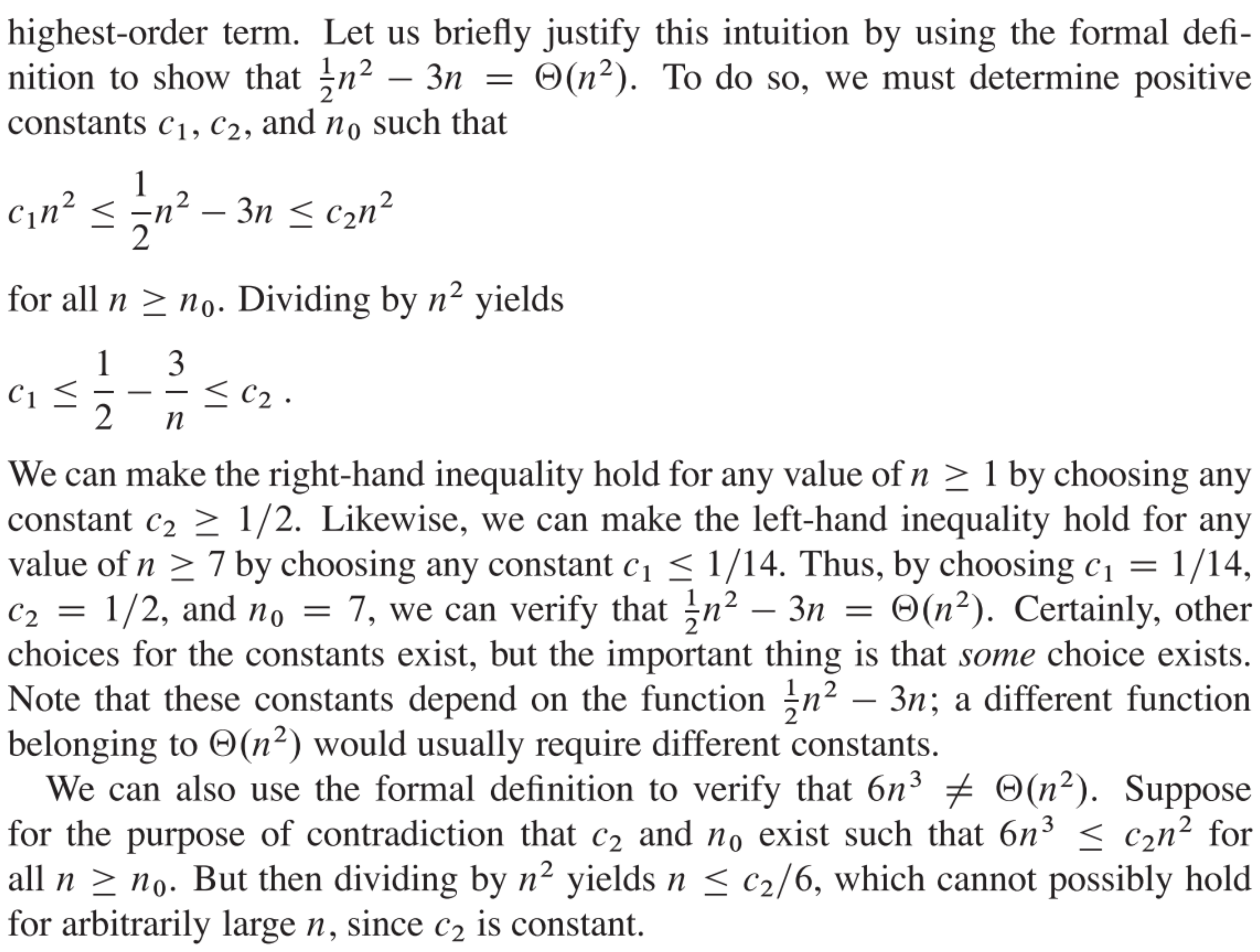
We use O-notation to give an upper bound on a function, to within a constant factor. Figure 3.1(b) shows the intuition behind O-notation. For all values n at and to the right of n_0, the value of the function $f(n)$ is on or below $cg(n)$.

Ω-notation

Just as O-notation provides an asymptotic *upper* bound on a function, Ω-notation provides an ***asymptotic lower bound***. For a given function $g(n)$, we denote by $\Omega(g(n))$ (pronounced "big-omega of g of n" or sometimes just "omega of g of n") the set of functions

$$\begin{aligned}\Omega(g(n)) = \{f(n) :\ & \text{there exist positive constants } c \text{ and } n_0 \text{ such that} \\ & 0 \le c g(n) \le f(n) \text{ for all } n \ge n_0\} .\end{aligned}$$

举个Θ表示法的例子。这里实际说明了$\Theta(n^2)$表示的是夹在c_1n^2和c_2n^2之间的一系列函数。


highest-order term. Let us briefly justify this intuition by using the formal definition to show that $\frac{1}{2}n^2 - 3n = \Theta(n^2)$. To do so, we must determine positive constants c_1, c_2, and n_0 such that

$$c_1 n^2 \le \frac{1}{2}n^2 - 3n \le c_2 n^2$$

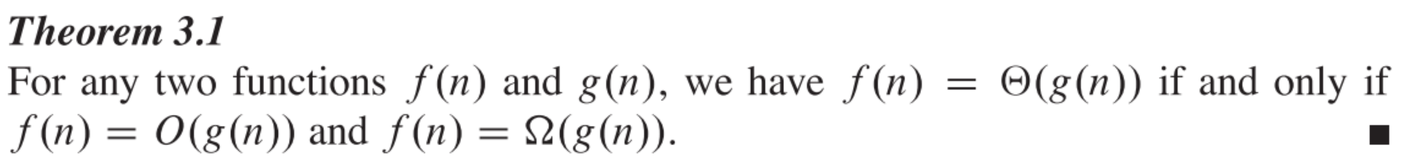
for all $n \ge n_0$. Dividing by n^2 yields

$$c_1 \le \frac{1}{2} - \frac{3}{n} \le c_2 .$$

We can make the right-hand inequality hold for any value of $n \ge 1$ by choosing any constant $c_2 \ge 1/2$. Likewise, we can make the left-hand inequality hold for any value of $n \ge 7$ by choosing any constant $c_1 \le 1/14$. Thus, by choosing $c_1 = 1/14$, $c_2 = 1/2$, and $n_0 = 7$, we can verify that $\frac{1}{2}n^2 - 3n = \Theta(n^2)$. Certainly, other choices for the constants exist, but the important thing is that *some* choice exists. Note that these constants depend on the function $\frac{1}{2}n^2 - 3n$; a different function belonging to $\Theta(n^2)$ would usually require different constants.

We can also use the formal definition to verify that $6n^3 \ne \Theta(n^2)$. Suppose for the purpose of contradiction that c_2 and n_0 exist such that $6n^3 \le c_2 n^2$ for all $n \ge n_0$. But then dividing by n^2 yields $n \le c_2/6$, which cannot possibly hold for arbitrarily large n, since c_2 is constant.

书上还介绍了一个定理：

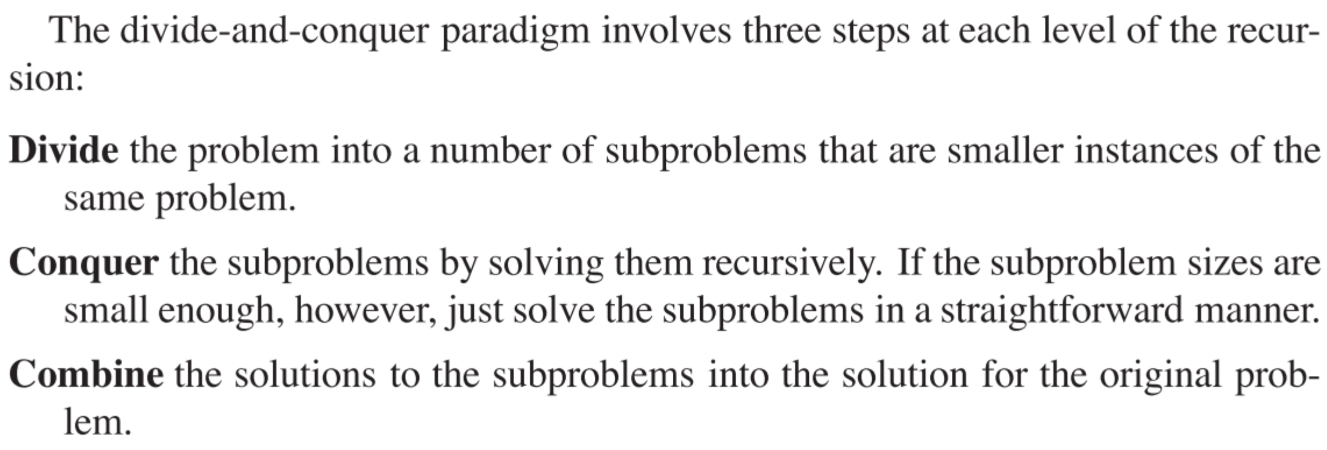

Theorem 3.1

For any two functions $f(n)$ and $g(n)$, we have $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$. ■

除此之外，书上还介绍了渐进表示法在等式与不等式中的情况，o表示法（大O表示法可能是渐进紧的（asymptitically tight）也可能不是，而小o表示法表示了非渐进紧的上界）、ω表示法等，篇幅有限，不再赘述。

递归、分治与递推式

许多实用的算法在结构上是递归（recursive）的，即在解决特定问题的时候，这些算法递归地调用自己去解决相关的子问题。这些算法通常符合一种分治的模式（a divide-and-conquer approach）：将原始问题分解为相似的但输入规模更小的子问题，递归地解决子问题，并将子问题的解合并起来得到原始问题的解。
分治范式（paradigm）通常包括以下三步：



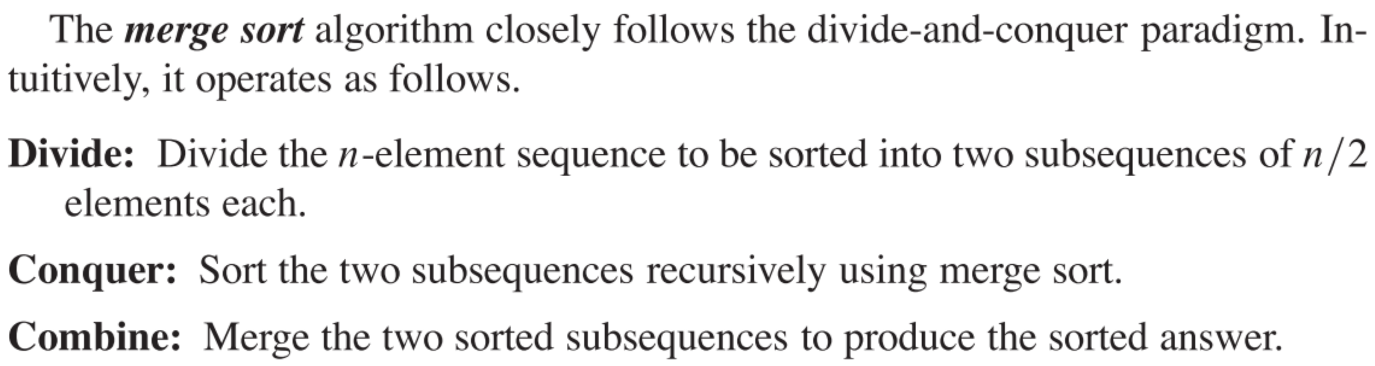
The divide-and-conquer paradigm involves three steps at each level of the recursion:

Divide the problem into a number of subproblems that are smaller instances of the same problem.

Conquer the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.

Combine the solutions to the subproblems into the solution for the original problem.

这里举一个归并排序（merge sort）的例子。归并排序采用了分治范式：



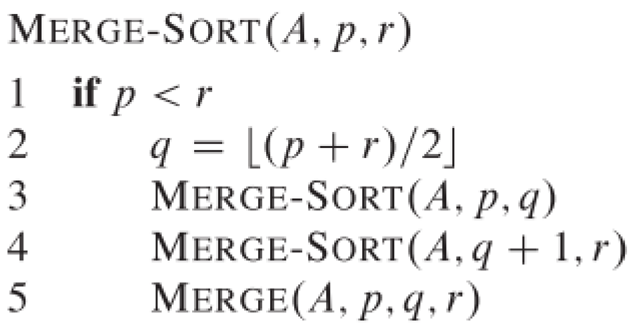
The ***merge sort*** algorithm closely follows the divide-and-conquer paradigm. Intuitively, it operates as follows.

Divide: Divide the n-element sequence to be sorted into two subsequences of $n/2$ elements each.

Conquer: Sort the two subsequences recursively using merge sort.

Combine: Merge the two sorted subsequences to produce the sorted answer.

描述该算法的伪代码见下。MERGE函数完成的步骤是将两个已经排序的子序列合并为一个排序序列，此处省略这一步完成的具体过程，有问题参见P30-P34。



```
MERGE-SORT($A, p, r$)
1  if $p < r$
2      $q = \lfloor (p + r)/2 \rfloor$
3      MERGE-SORT($A, p, q$)
4      MERGE-SORT($A, q + 1, r$)
5      MERGE($A, p, q, r$)
```

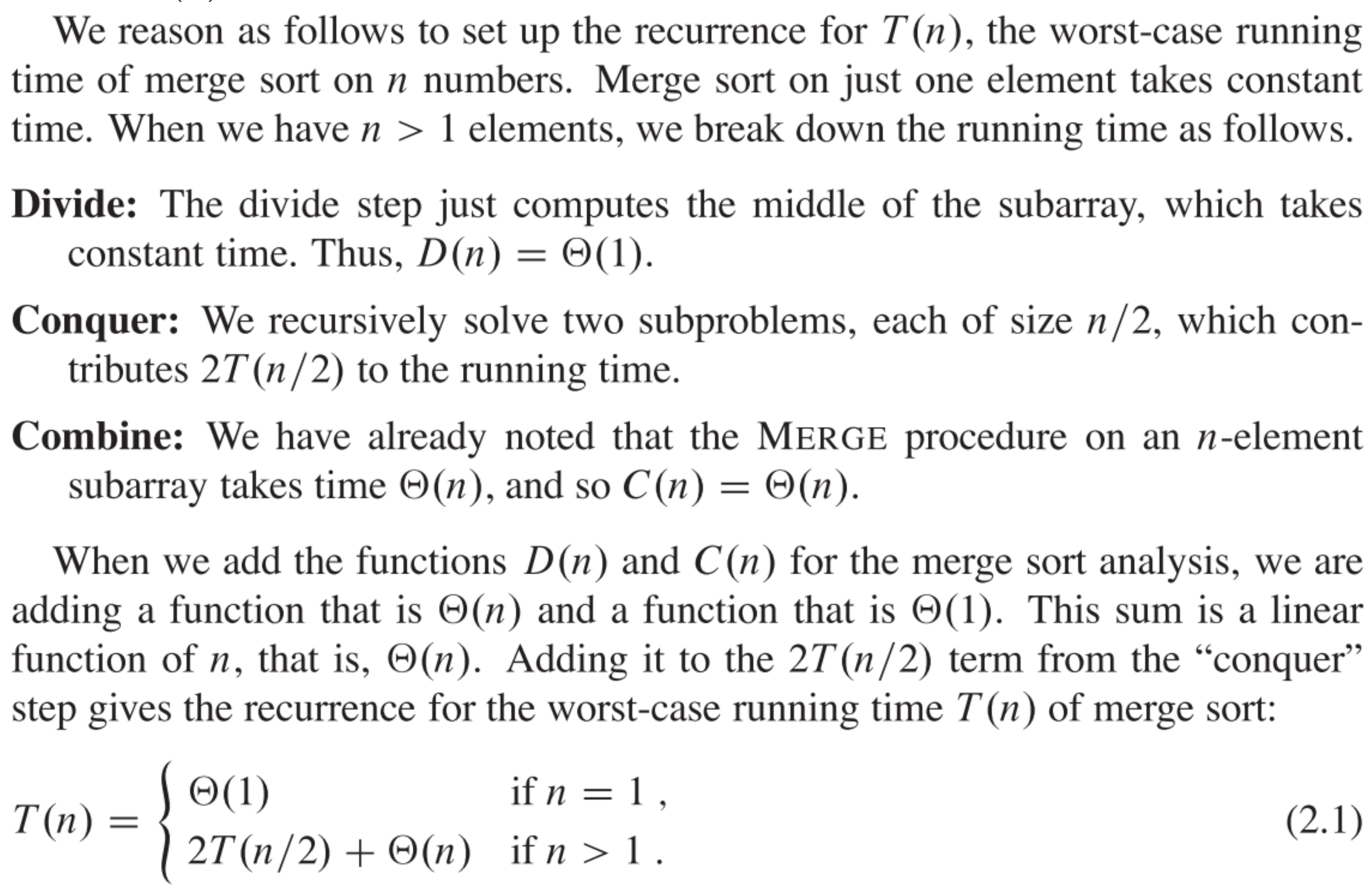
如果一个算法递归调用自身，我们在描述其运行时间的时候通常采用递推式（recurrence equation）或者递推（recurrence）。符合分治范式的递推式常见格式为:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \le c \ , \\ aT(n/b) + D(n) + C(n) & \text{otherwise} \ . \end{cases}$$

这个递推式描述的是，在input size(n)小于等于某一个常数c的时候，运行时间$T(n)$为一个常数时间$\Theta(n)$，当input size(n)大于常数c的时候，我们通过分治，将问题分解为a个子问题，每个子问题的input size为原始问题的$1/b$，因为子问题和原始问题是相似的，同样符合$T(n)$的运行时间函数，这a个子问

题总共花费的运行时间为$aT(n/b)$，$D(n)$表示的是将问题划分（divide）为子问题花费的运行时间，$C(n)$表示的是将子问题的解合并（combine）为原始问题的解需要花费的运行时间。

对于归并排序的案例，它每个步骤的运行时间和递推式（这里研究的是最坏情况（worst-case），MERGE需要$\Theta(n)$）：



We reason as follows to set up the recurrence for $T(n)$, the worst-case running time of merge sort on n numbers. Merge sort on just one element takes constant time. When we have $n > 1$ elements, we break down the running time as follows.

Divide: The divide step just computes the middle of the subarray, which takes constant time. Thus, $D(n) = \Theta(1)$.

Conquer: We recursively solve two subproblems, each of size $n/2$, which contributes $2T(n/2)$ to the running time.

Combine: We have already noted that the MERGE procedure on an n-element subarray takes time $\Theta(n)$, and so $C(n) = \Theta(n)$.

When we add the functions $D(n)$ and $C(n)$ for the merge sort analysis, we are adding a function that is $\Theta(n)$ and a function that is $\Theta(1)$. This sum is a linear function of n, that is, $\Theta(n)$. Adding it to the $2T(n/2)$ term from the "conquer" step gives the recurrence for the worst-case running time $T(n)$ of merge sort:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 , \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 . \end{cases} \qquad (2.1)$$

其实这个式子有点像高中数列里面的递推公式，我们想得到$T(n)$的函数表达式，实际上就是想求数列的通项公式，下面的部分将介绍几种求解方法。

求解递推式的几种方法

书上主要介绍了三种方法——substitution method（代入法）、recurtion-tree method（递归树法）、master method（主方法）。三种方法的简要描述如下：

This chapter offers three methods for solving recurrences—that is, for obtaining asymptotic “Θ” or “O” bounds on the solution:

- In the ***substitution method***, we guess a bound and then use mathematical induction to prove our guess correct.
- The ***recursion-tree method*** converts the recurrence into a tree whose nodes represent the costs incurred at various levels of the recursion. We use techniques for bounding summations to solve the recurrence.
- The ***master method*** provides bounds for recurrences of the form

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

下面详细介绍一下各种方法的具体使用：

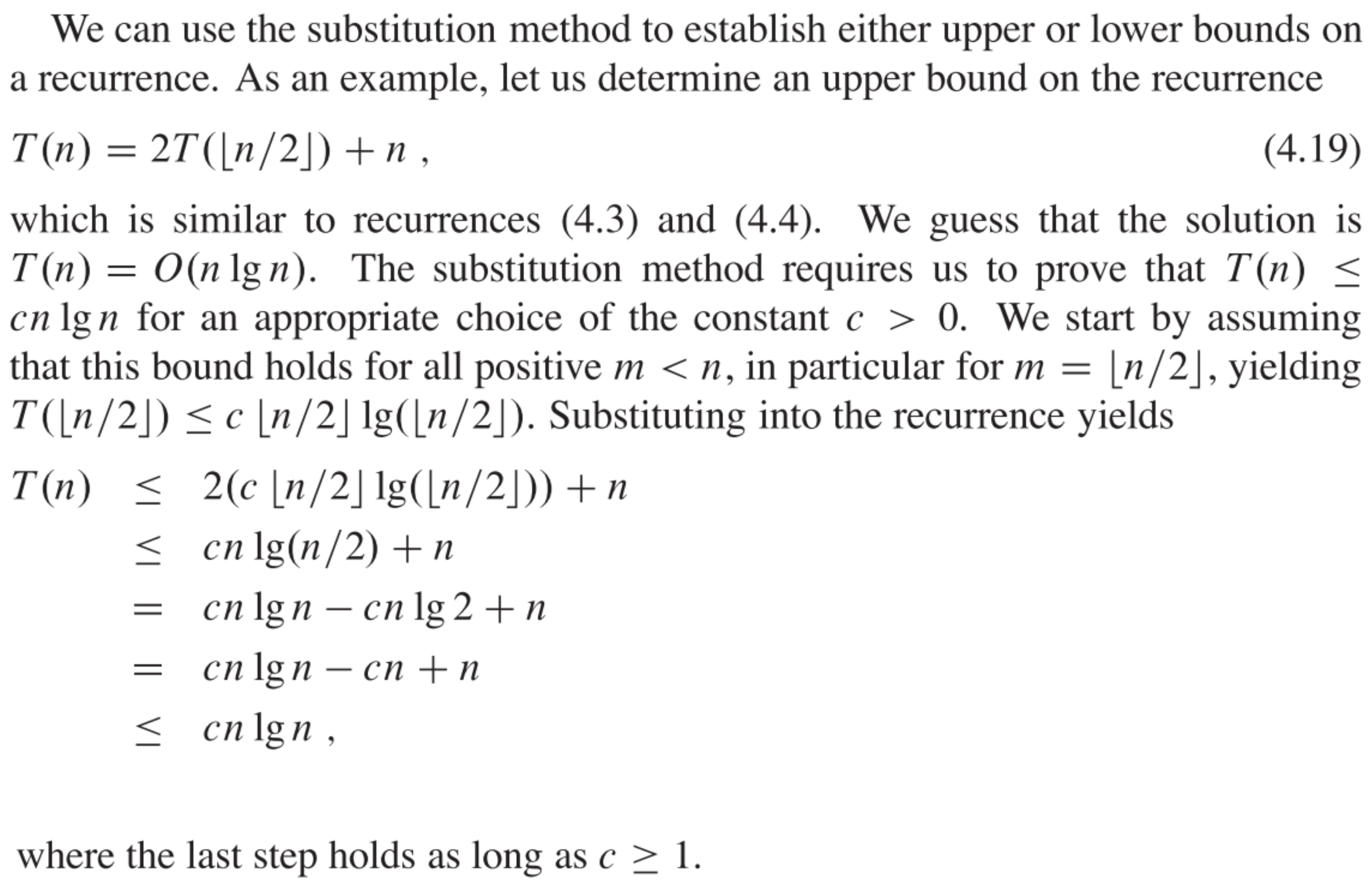
1. substitution method（代入法）

代入法解递推式主要包含两步，第一步是猜测解的形式，第二步是用数学归纳法（mathematical induction）找到相应常数并验证猜测的解：

The ***substitution method*** for solving recurrences comprises two steps:

1. Guess the form of the solution.
2. Use mathematical induction to find the constants and show that the solution works.

举一个例子：


We can use the substitution method to establish either upper or lower bounds on a recurrence. As an example, let us determine an upper bound on the recurrence

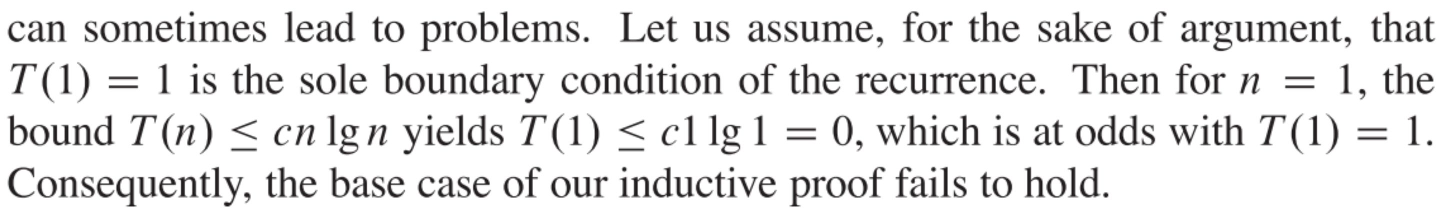
$$T(n) = 2T(\lfloor n/2 \rfloor) + n \ , \tag{4.19}$$

which is similar to recurrences (4.3) and (4.4). We guess that the solution is $T(n) = O(n \lg n)$. The substitution method requires us to prove that $T(n) \le cn \lg n$ for an appropriate choice of the constant $c > 0$. We start by assuming that this bound holds for all positive $m < n$, in particular for $m = \lfloor n/2 \rfloor$, yielding $T(\lfloor n/2 \rfloor) \le c \lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)$. Substituting into the recurrence yields

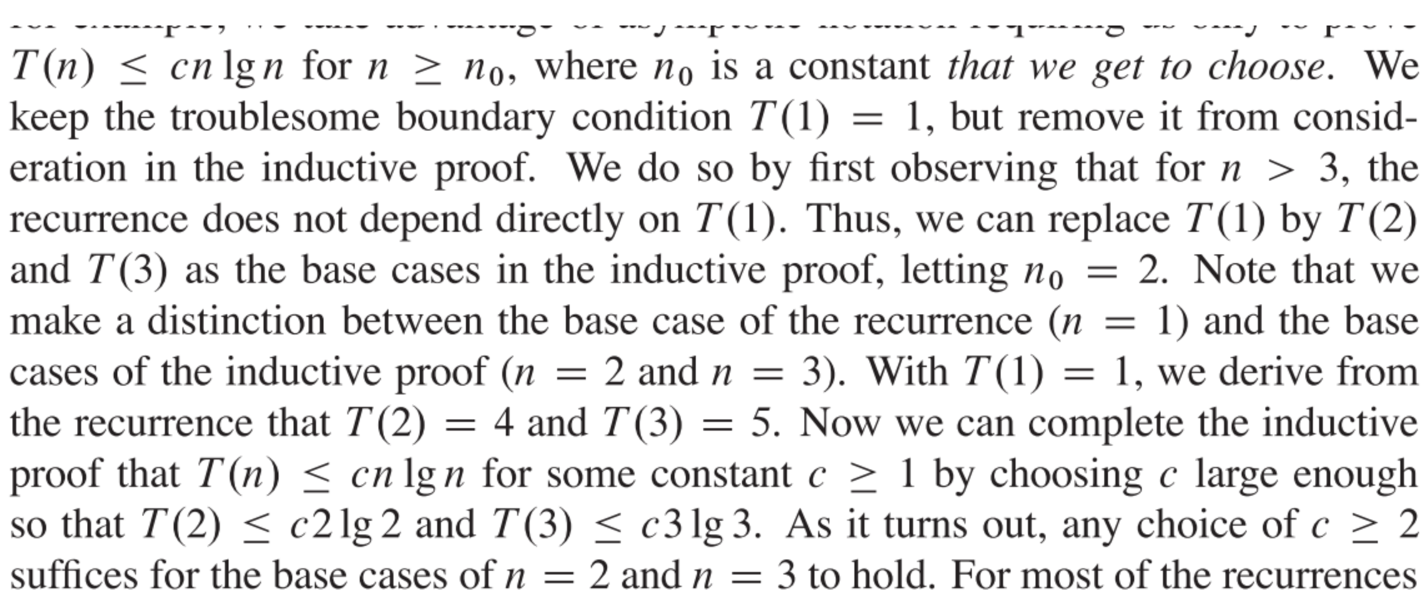
$$\begin{aligned} T(n) &\le 2(c \lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)) + n \\ &\le cn \lg(n/2) + n \\ &= cn \lg n - cn \lg 2 + n \\ &= cn \lg n - cn + n \\ &\le cn \lg n \ , \end{aligned}$$

where the last step holds as long as $c \ge 1$.

代入法之所以叫代入法，是因为图中第一步将子问题对应的猜测解代入到了递推式中，从而将等式转化为了不等式。以上的证明实际上还不算完整，因为数学归纳法中要求对边界条件进行检验（回忆一下高中数学归纳法，经常先验证第一项是否满足）。然而初始项（base case）通常是不满足的:


can sometimes lead to problems. Let us assume, for the sake of argument, that $T(1) = 1$ is the sole boundary condition of the recurrence. Then for $n = 1$, the bound $T(n) \le cn \lg n$ yields $T(1) \le c 1 \lg 1 = 0$, which is at odds with $T(1) = 1$. Consequently, the base case of our inductive proof fails to hold.

由于渐进表示法允许我们证明在n大于等于某个n_0时满足$T(n) \le cnlgn$即可，这里的n_0我们可以进行选择以保证初始项符合猜测解。下面这里剔除掉了$T(1)$并以$T(2)$和$T(3)$作为base case，常数项c也发生了改变。



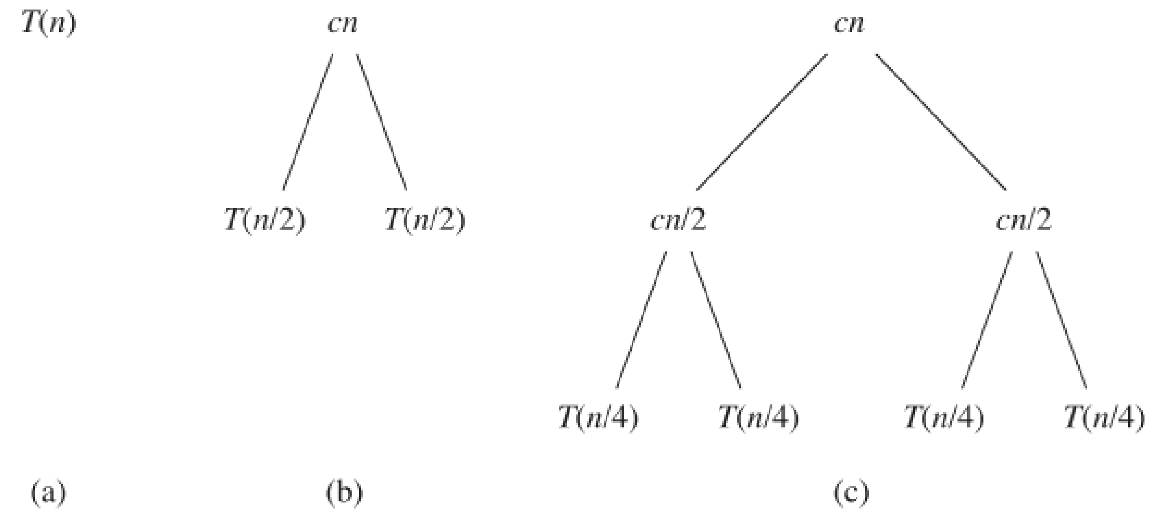
for example, we take advantage of asymptotic notation requiring us only to prove $T(n) \le cn \lg n$ for $n \ge n_0$, where n_0 is a constant *that we get to choose*. We keep the troublesome boundary condition $T(1) = 1$, but remove it from consideration in the inductive proof. We do so by first observing that for $n > 3$, the recurrence does not depend directly on $T(1)$. Thus, we can replace $T(1)$ by $T(2)$ and $T(3)$ as the base cases in the inductive proof, letting $n_0 = 2$. Note that we make a distinction between the base case of the recurrence ($n = 1$) and the base cases of the inductive proof ($n = 2$ and $n = 3$). With $T(1) = 1$, we derive from the recurrence that $T(2) = 4$ and $T(3) = 5$. Now we can complete the inductive proof that $T(n) \le cn \lg n$ for some constant $c \ge 1$ by choosing c large enough so that $T(2) \le c 2 \lg 2$ and $T(3) \le c 3 \lg 3$. As it turns out, any choice of $c \ge 2$ suffices for the base cases of $n = 2$ and $n = 3$ to hold. For most of the recurrences

代入法涉及到一些如何猜测解、递推假设是否足够强、换元的运用等问题，由于篇幅限制，此处不再展开，详见P84-87。

1. recurtion-tree method（递归树法）

在使用代入法的时候，猜测解可能是比较困难的一步。递归树法可以帮助猜测解，但是同时它也可以作为一种直接得出解的方法。递归树法使用树的形式表示递归过程，树的结点（node）表示了每个子问题所需要花费的运行时间，加和层结点得到每一层（level）所需花费的时间并把所有层需要花费的时间加和，可以得到总时间。

前面提到的归并排序的递归树：



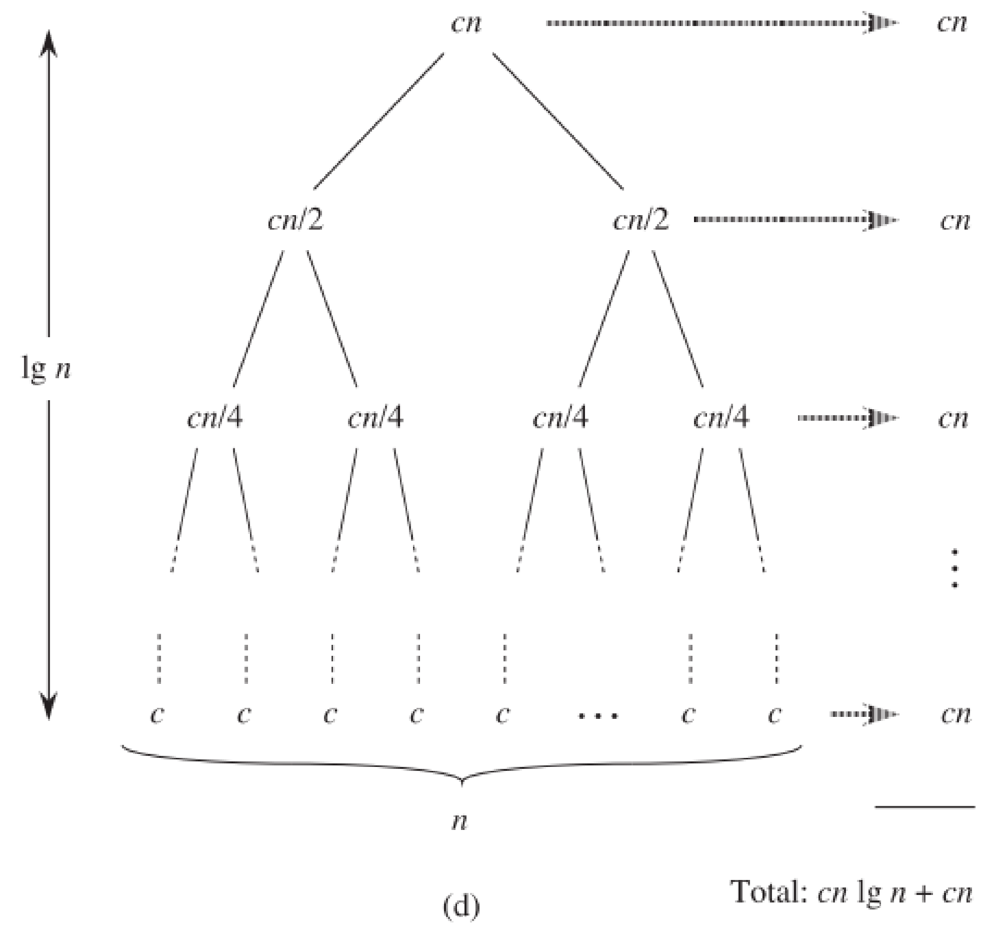


Figure 2.5 How to construct a recursion tree for the recurrence $T(n) = 2T(n/2) + cn$. Part **(a)** shows $T(n)$, which progressively expands in **(b)–(d)** to form the recursion tree. The fully expanded tree in part (d) has $\lg n + 1$ levels (i.e., it has height $\lg n$, as indicated), and each level contributes a total cost of cn. The total cost, therefore, is $cn \lg n + cn$, which is $\Theta(n \lg n)$.

到这里你应该大致了解了递归树的形式与求解，如果你有兴趣继续深入了解的话，可以看一下下面这两种相对来说比较不常见的递归树，与它们的求解过程。
第一种是一个三叉树：
它的递推式是

$$T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$$

由于向下取整（floors）与向上取整（ceilings）在求解通项公式的时候并没有特别大的影响，我们可以直接把问题看作：

tolerate), we create a recursion tree for the recurrence $T(n) = 3T(n/4) + cn^2$, having written out the implied constant coefficient $c > 0$.

这里需要注意的是，我们在这个问题中把n看是以4为底的指数，这样子问题的规模就是一个整数了，可以简化问题。递归树：

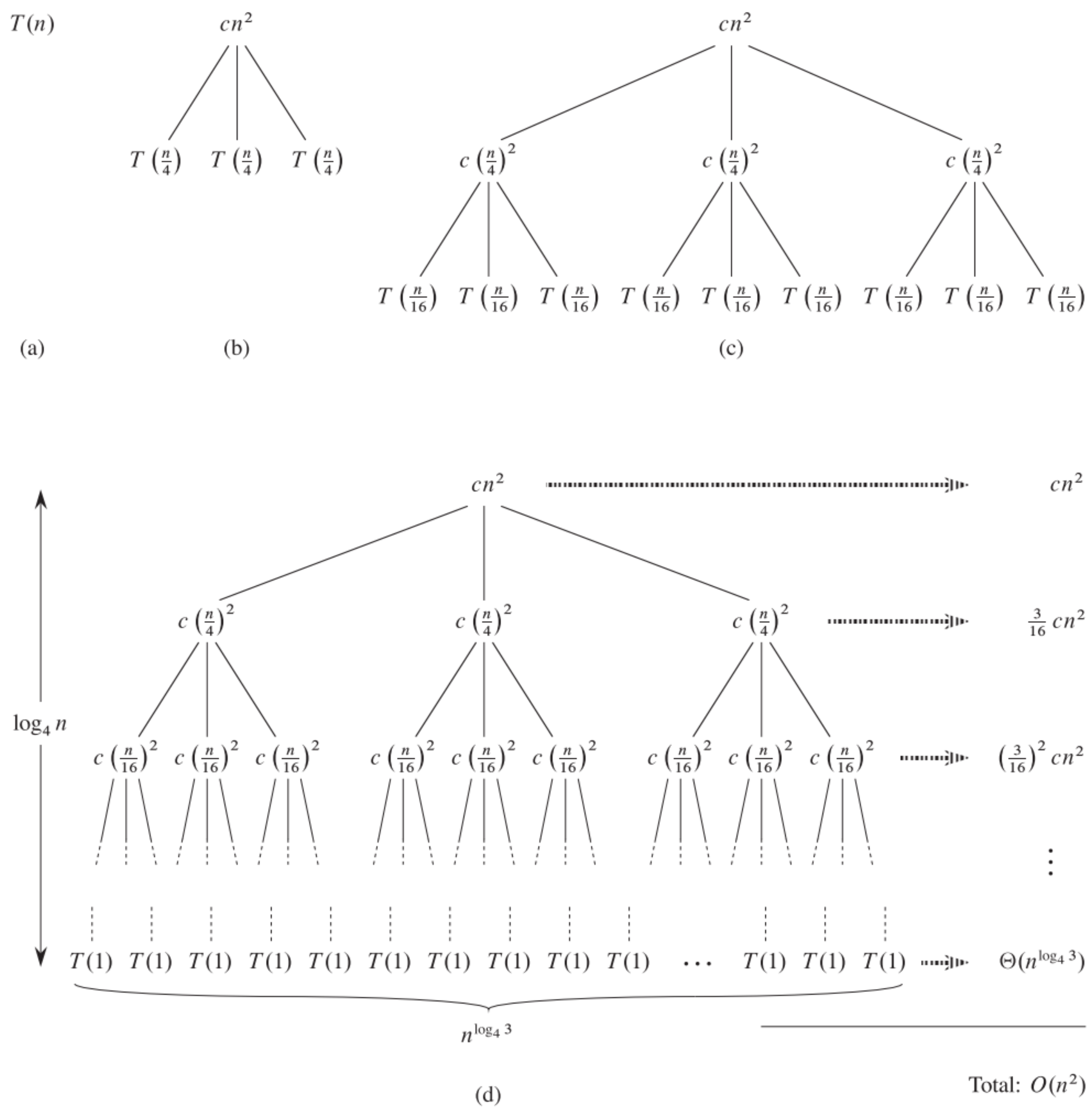


Figure 4.5 Constructing a recursion tree for the recurrence $T(n) = 3T(n/4) + cn^2$. Part **(a)** shows $T(n)$, which progressively expands in **(b)–(d)** to form the recursion tree. The fully expanded tree in part (d) has height $\log_4 n$ (it has $\log_4 n + 1$ levels).

稍微解释一下上图中的(b)，它实际上表示的是分治的divide和combine过程花费了cn^2的时间（根节点），然后是三个子问题，每个花费$T(n/4)$时间。具体的求和过程是一个完全的数学过程，实际上是把图(d)中右边列出的每一层的时间求和，求解过程比较繁复，但是我觉得还是有必要在这里粘贴出来，它给出的求解思路与计算非常清晰，如何计算子问题size、层数、每层时间总数、求和并用无穷递减等比级数放缩，值得学习：

Because subproblem sizes decrease by a factor of 4 each time we go down one level, we eventually must reach a boundary condition. How far from the root do we reach one? The subproblem size for a node at depth i is $n/4^i$. Thus, the subproblem size hits $n = 1$ when $n/4^i = 1$ or, equivalently, when $i = \log_4 n$. Thus, the tree has $\log_4 n + 1$ levels (at depths $0, 1, 2, \ldots, \log_4 n$).

Next we determine the cost at each level of the tree. Each level has three times more nodes than the level above, and so the number of nodes at depth i is 3^i. Because subproblem sizes reduce by a factor of 4 for each level we go down from the root, each node at depth i, for $i = 0, 1, 2, \ldots, \log_4 n - 1$, has a cost of $c(n/4^i)^2$. Multiplying, we see that the total cost over all nodes at depth i, for $i = 0, 1, 2, \ldots, \log_4 n - 1$, is $3^i c(n/4^i)^2 = (3/16)^i cn^2$. The bottom level, at depth $\log_4 n$, has $3^{\log_4 n} = n^{\log_4 3}$ nodes, each contributing cost $T(1)$, for a total cost of $n^{\log_4 3}T(1)$, which is $\Theta(n^{\log_4 3})$, since we assume that $T(1)$ is a constant.

Now we add up the costs over all levels to determine the cost for the entire tree:

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

This last formula looks somewhat messy until we realize that we can again take advantage of small amounts of sloppiness and use an infinite decreasing geometric series as an upper bound. Backing up one step and applying equation (A.6), we have

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

Thus, we have derived a guess of $T(n) = O(n^2)$ for our original recurrence

$T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$. In this example, the coefficients of cn^2 form a

这里其实得到了一个猜想解，用代入法可以验证这个猜想：

Now we can use the substitution method to verify that our guess was correct, that is, $T(n) = O(n^2)$ is an upper bound for the recurrence $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$. We want to show that $T(n) \le dn^2$ for some constant $d > 0$. Using the same constant $c > 0$ as before, we have

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

where the last step holds as long as $d \ge (16/13)c$.

第二种把一个原始问题分成了不等规模的子问题，会导致到叶子结点路径不等：

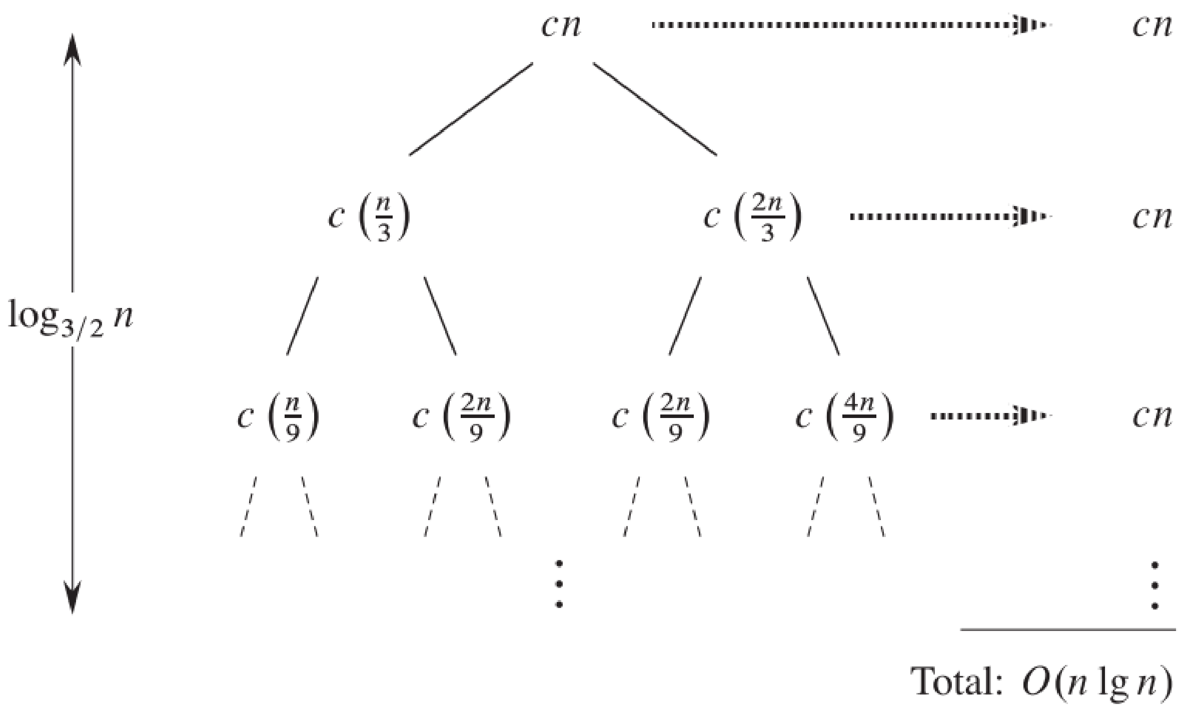

cn
cn
$c\left(\frac{n}{3}\right)$
$c\left(\frac{2n}{3}\right)$
cn
$\log_{3/2} n$
$c\left(\frac{n}{9}\right)$
$c\left(\frac{2n}{9}\right)$
$c\left(\frac{2n}{9}\right)$
$c\left(\frac{4n}{9}\right)$
cn
Total: $O(n \lg n)$

Figure 4.6 A recursion tree for the recurrence $T(n) = T(n/3) + T(2n/3) + cn$.

求解过程：

The longest simple path from the root to a leaf is $n \to (2/3)n \to (2/3)^2 n \to \cdots \to 1$. Since $(2/3)^k n = 1$ when $k = \log_{3/2} n$, the height of the tree is $\log_{3/2} n$.

Intuitively, we expect the solution to the recurrence to be at most the number of levels times the cost of each level, or $O(cn \log_{3/2} n) = O(n \lg n)$. Figure 4.6

但其实这棵递归树不是完全二叉树，所以不是每一层的时间和都为cn。以上这种方法有一些小问题，但由于我们只是猜测一个解，对精确度不需要有太高的要求。接下来用代入法验证猜想：

Indeed, we can use the substitution method to verify that $O(n \lg n)$ is an upper bound for the solution to the recurrence. We show that $T(n) \le dn \lg n$, where d is a suitable positive constant. We have

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

as long as $d \ge c/(\lg 3 - (2/3))$. Thus, we did not need to perform a more accurate accounting of costs in the recursion tree.

3. master method（主方法）

主方法其实是基于主定理（Master Theorem）给出解的，我个人认为这是求解较为通用且简便的方法，但主方法存在一定限制。

Theorem 4.1 (Master theorem)

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

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

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ∎

解释一下主定理。三种情况其实代表了$f(n)$与$n^{\log_b a}$的三种不同的大小关系，第一种情况是$f(n)$小，第二种情况是相等，第三种情况是$n^{\log_b a}$小。

Before applying the master theorem to some examples, let's spend a moment trying to understand what it says. In each of the three cases, we compare the function $f(n)$ with the function $n^{\log_b a}$. Intuitively, the larger of the two functions determines the solution to the recurrence. If, as in case 1, the function $n^{\log_b a}$ is the larger, then the solution is $T(n) = \Theta(n^{\log_b a})$. If, as in case 3, the function $f(n)$ is the larger, then the solution is $T(n) = \Theta(f(n))$. If, as in case 2, the two functions are the same size, we multiply by a logarithmic factor, and the solution is $T(n) = \Theta(n^{\log_b a} \lg n) = \Theta(f(n) \lg n)$.

注意，上面的“小”不只是数值上小，而是多项式小于（polynomially smaller），具体解释见下。这会导致主定理不能覆盖所有$f(n)$的情况，某些情况下不可以使用主定理：

Beyond this intuition, you need to be aware of some technicalities. In the first case, not only must $f(n)$ be smaller than $n^{\log_b a}$, it must be *polynomially* smaller.

That is, $f(n)$ must be asymptotically smaller than $n^{\log_b a}$ by a factor of n^{ϵ} for some constant $\epsilon > 0$. In the third case, not only must $f(n)$ be larger than $n^{\log_b a}$, it also must be polynomially larger and in addition satisfy the "regularity" condition that $af(n/b) \le cf(n)$. This condition is satisfied by most of the polynomially bounded functions that we shall encounter.

Note that the three cases do not cover all the possibilities for $f(n)$. There is a gap between cases 1 and 2 when $f(n)$ is smaller than $n^{\log_b a}$ but not polynomially smaller. Similarly, there is a gap between cases 2 and 3 when $f(n)$ is larger than $n^{\log_b a}$ but not polynomially larger. If the function $f(n)$ falls into one of these gaps, or if the regularity condition in case 3 fails to hold, you cannot use the master method to solve the recurrence.

下面举几个运用主方法求解的例子：

To use the master method, we simply determine which case (if any) of the master theorem applies and write down the answer.

As a first example, consider

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

For this recurrence, we have $a = 9$, $b = 3$, $f(n) = n$, and thus we have that $n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$. Since $f(n) = O(n^{\log_3 9-\epsilon})$, where $\epsilon = 1$, we can apply case 1 of the master theorem and conclude that the solution is $T(n) = \Theta(n^2)$.

Now consider

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

in which $a = 1$, $b = 3/2$, $f(n) = 1$, and $n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$. Case 2 applies, since $f(n) = \Theta(n^{\log_b a}) = \Theta(1)$, and thus the solution to the recurrence is $T(n) = \Theta(\lg n)$.

For the recurrence

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

we have $a = 3$, $b = 4$, $f(n) = n \lg n$, and $n^{\log_b a} = n^{\log_4 3} = O(n^{0.793})$. Since $f(n) = \Omega(n^{\log_4 3+\epsilon})$, where $\epsilon \approx 0.2$, case 3 applies if we can show that the regularity condition holds for $f(n)$. For sufficiently large n, we have that $af(n/b) = 3(n/4)\lg(n/4) \le (3/4)n \lg n = cf(n)$ for $c = 3/4$. Consequently, by case 3, the solution to the recurrence is $T(n) = \Theta(n \lg n)$.

The master method does not apply to the recurrence

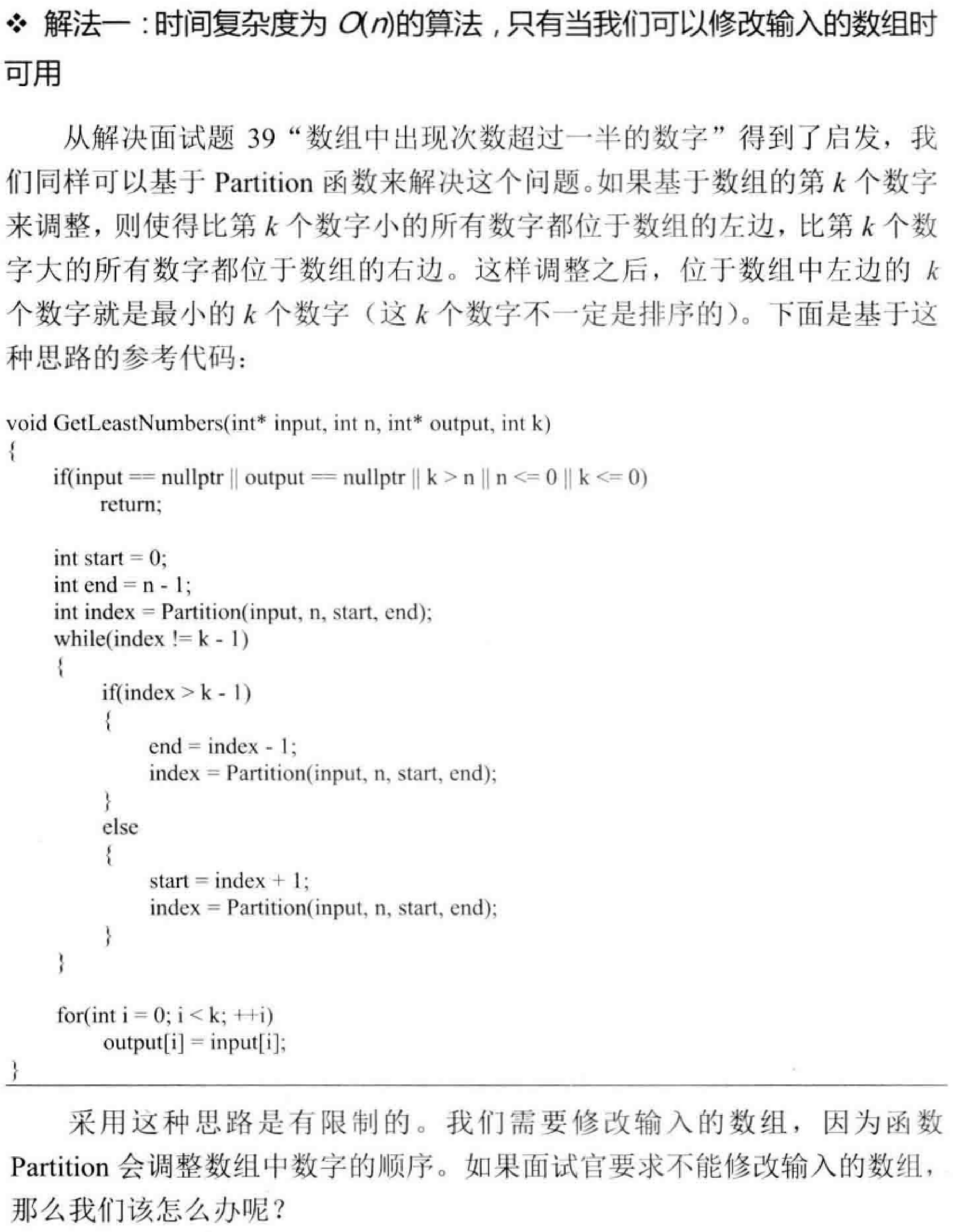
$$T(n) = 2T(n/2) + n \lg n \ ,$$

even though it appears to have the proper form: $a = 2$, $b = 2$, $f(n) = n \lg n$, and $n^{\log_b a} = n$. You might mistakenly think that case 3 should apply, since

$f(n) = n \lg n$ is asymptotically larger than $n^{\log_b a} = n$. The problem is that it is not *polynomially* larger. The ratio $f(n)/n^{\log_b a} = (n \lg n)/n = \lg n$ is asymptotically less than n^ϵ for any positive constant ϵ. Consequently, the recurrence falls into the gap between case 2 and case 3. (See Exercise 4.6-2 for a solution.)

回到原点

我一开始研究这个问题是因为在剑指上看到了一个不太理解的时间复杂度，后面发现其实这跟递归树关系也不怎么大，所以总得来说前面这么多是白看了hhh。这段代码来自剑指P210，我对这个算法是O(n)存疑。



❖ 解法一：时间复杂度为 $O(n)$的算法，只有当我们可以修改输入的数组时可用

从解决面试题 39“数组中出现次数超过一半的数字”得到了启发，我们同样可以基于 Partition 函数来解决这个问题。如果基于数组的第 k 个数字来调整，则使得比第 k 个数字小的所有数字都位于数组的左边，比第 k 个数字大的所有数字都位于数组的右边。这样调整之后，位于数组中左边的 k 个数字就是最小的 k 个数字（这 k 个数字不一定是排序的）。下面是基于这种思路的参考代码：

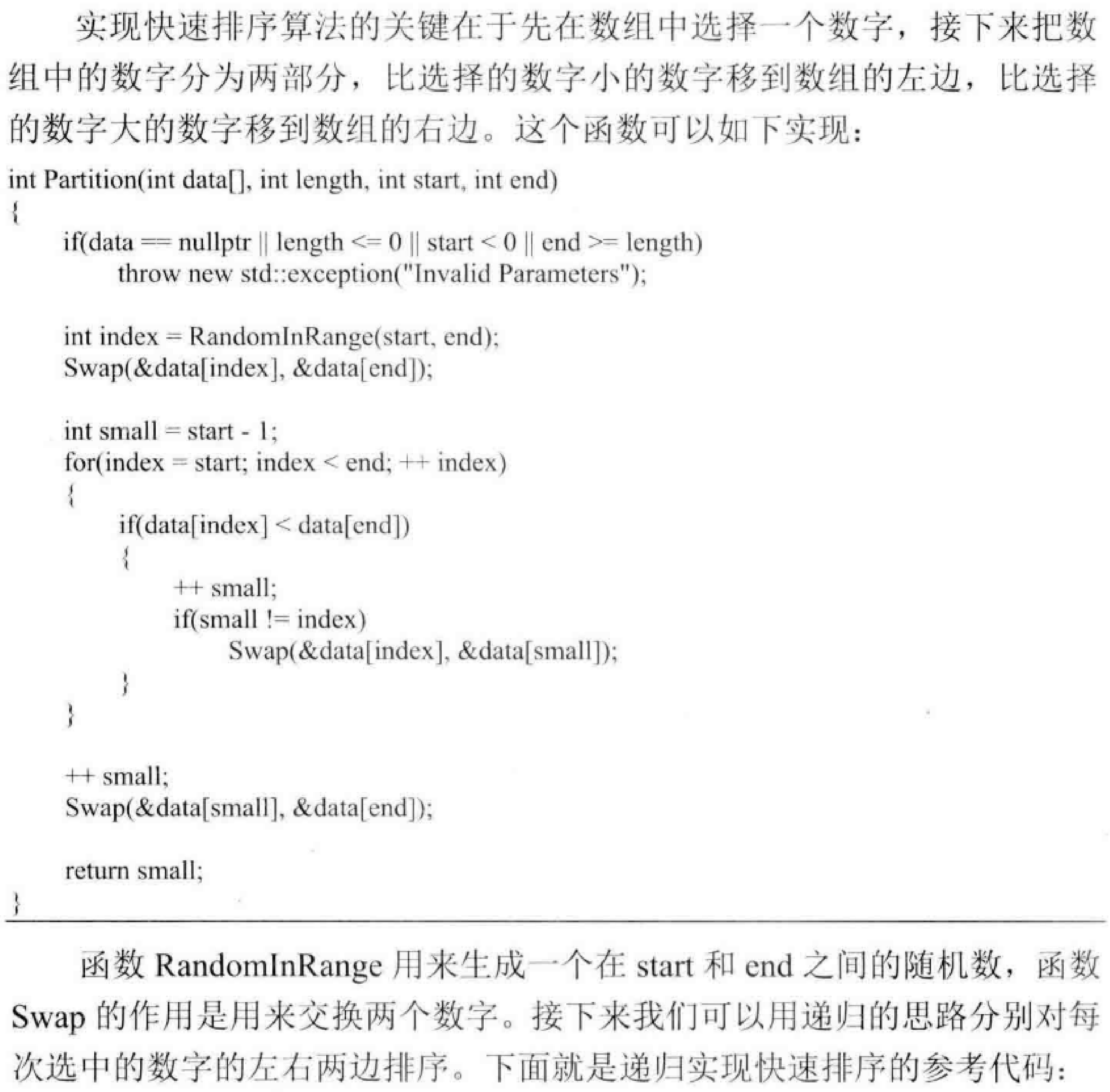
```
void GetLeastNumbers(int* input, int n, int* output, int k)
{
    if(input == nullptr || output == nullptr || k > n || n <= 0 || k <= 0)
        return;

    int start = 0;
    int end = n - 1;
    int index = Partition(input, n, start, end);
    while(index != k - 1)
    {
        if(index > k - 1)
        {
            end = index - 1;
            index = Partition(input, n, start, end);
        }
        else
        {
            start = index + 1;
            index = Partition(input, n, start, end);
        }
    }

    for(int i = 0; i < k; ++i)
        output[i] = input[i];
}
```

采用这种思路是有限制的。我们需要修改输入的数组，因为函数 Partition 会调整数组中数字的顺序。如果面试官要求不能修改输入的数组，那么我们该怎么办呢？

这个算法里面提到的Partition函数第一次出现好像是在快速排序里面，使某一个选出的数字左边都是小于它的，右边都是大于等于它的，选自剑指P80。



实现快速排序算法的关键在于先在数组中选择一个数字，接下来把数组中的数字分为两部分，比选择的数字小的数字移到数组的左边，比选择的数字大的数字移到数组的右边。这个函数可以如下实现：

```
int Partition(int data[], int length, int start, int end)
{
    if(data == nullptr || length <= 0 || start < 0 || end >= length)
        throw new std::exception("Invalid Parameters");

    int index = RandomInRange(start, end);
    Swap(&data[index], &data[end]);

    int small = start - 1;
    for(index = start; index < end; ++ index)
    {
        if(data[index] < data[end])
        {
            ++ small;
            if(small != index)
                Swap(&data[index], &data[small]);
        }
    }

    ++ small;
    Swap(&data[small], &data[end]);

    return small;
}
```

函数 RandomInRange 用来生成一个在 start 和 end 之间的随机数，函数 Swap 的作用是用来交换两个数字。接下来我们可以用递归的思路分别对每次选中的数字的左右两边排序。下面就是递归实现快速排序的参考代码：

为什么说白看了，是因为这根本不是递归树好吧，它都不是调用自己。好的，接下来都是我自己的看法。Partition这个函数选的index有随机性，导致getLeastNumber的时间复杂度有最好和最坏情况，Partition时间复杂度应该是O(n)吧。最好情况是，第一次就随机到了k-1就不执行while里面的东西了，复杂度应该是O(n)。最坏的情况是，我要最小的一个数（k取1），partition随机到的是最小的n、n-1、n-2....个数，复杂度应该是O(n2)吧。