第5章

分治算法

5.1 算法思想

许多实用的算法都具有递归的结构：为解决一个给定的问题，通过递归地调用自己一次或多次来解决与该问题具有相似结构的子问题。分治算法(Divide and Conquer)就是这样一种具有递归结构的算法，其基本思想是把一个问题分解成若干个子问题(这些子问题与原问题在本质上是同一种问题类型，只是问题规模不同)，然后递归地解决子问题，最后把子问题的解组合成原问题的解。递归算法与分治算法情同手足，互不分离，经常同时应用在算法设计之中，并产生许多高效的算法。

分治算法在求解问题时，通常遵循以下 3 个步骤。

步骤 1：把一个问题分解成若干个子问题。

步骤 2：通过递归地解决子问题来解决原问题。如果子问题的问题规模小到可以用直接的方法求出，那么停止递归。

步骤 3：把这些子问题的解组合成原问题的解。

当一种算法的过程中含有对自身的递归调用时，它的运行时间常用一个递归方程来表示。递归方程描述了求解一个问题规模为 n 的问题所需要的运行时间可以由其子问题的运行时间来决定。对于递归方程而言，如果知道递归的初始边界，那么能够很方便地利用上一章介绍的方法来求解递归方程。

分治算法的运行时间主要由它的 3 个步骤决定。令 $T(n)$表示求解一个问题规模为 n 的问题所需要的运行时间，设问题规模小到可以用直接的方法求解，所花费的时间为 $\Theta(1)$，即存在一个正整数 c，使当 $n \leqslant c$ 时，问题的求解非常简单，只需要常数级的时间就可以得到该问题的解。假设把问题分解成 a 个子问题，每个子问题的大小为 n/b。值得注意的是，n/b 可能是 $\lfloor n/b \rfloor$ 或者 $\lceil n/b \rceil$，这里忽略底和顶是因为上一章已经分析；去掉底和顶，并不影响递归方程的求解。令把问题分解成 a 个子问题所花费的时间为 $D(n)$，每个子问题的解组合起来所花费的时间为 $C(n)$，则得到递归方程

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

如果将时间 $D(n)$和 $C(n)$合并成函数 $f(n)$，则许多分治算法的时间复杂度可以由上

一章介绍的公式法求出。从上面的递归方程可以看出，分治算法的优点是它的运行时间常常可以很容易地由其子问题的运行时间得到。正如上一章所介绍的，分治算法在设计的时候，应尽量减少子问题的个数 a 或者降低 $f(n)$ 的数量级，以便设计出更有效的分治算法。

下面介绍分治算法的应用。

5.2 合并排序

前面介绍了插入排序算法和选择排序算法，现在利用分治算法的思想，设计一种合并排序算法，其基本步骤如下。

步骤 1：将包含 n 个元素的序列均分成包含 $n/2$ 个元素的子序列。

步骤 2：对两个子序列进行递归划分。

步骤 3：把两个已经排序的子序列合并成一个有序序列。

合并排序算法 MergeSort(A，p，r)的伪代码如下。

```
MergeSort(A, p, r)
1    if p < r then
2       q←⌊(p + r)/2⌋
3    MergeSort(A, p, q)
4    MergeSort(A, (q + 1), r)
5    Merge(A, p, q, r)
```

在 MergeSort(A，p，r)中，要排序的元素保存在数组 A 中，当前要排序的范围是数组 A 中位置 $p\sim r$ 的元素。当 $p\geqslant r$ 时，停止递归调用，否则，找出中间划分点 q，并分别递归求解两个子问题。两个子问题求解后，它们的解合并在一起，得到原问题的解。求解原问题，只需调用 MergeSort(A，1，n)。

合并排序算法的关键步骤是如何把两个有序序列合并成一个有序序列，这个步骤可以用辅助函数 Merge(A，p，q，r)来完成，其中，p、q、r 是数组下标且 $p\leqslant q<r$。该函数假设数组 $A[p..q]$和 $A[(q+1)..r]$ 是已排好序的，然后将它们合并为一个有序的子数组 $A[p..r]$。详细的合并过程如下。

```
Merge(A, p, q, r)
1    n1←q - p + 1
2    n2←r - q
3    for i←1 to n1 do
4        L[i]←A[p + i - 1]
5    for j←1 to n2 do
6        R[j]←A[q + j]
7    L[n1 + 1]←∞
8    R[n2 + 1]←∞
9    i←1
10   j←1
11   for k ← p to r do
12       if L[i]≤R[j] then
13           A[k]←L[i]
14           i←i + 1
```

```
15        else
16            A[k]←R[j]
17            j←j + 1
```

Merge(A,p,q,r)用两个子数组 $L[1..(n_1+1)]$,$R[1..(n_2+1)]$分别保存两个已经排好序的序列,以便腾出数组 A 中的位置来保存合并后的元素。同时,两个数组均多开辟一个存储单元来存储一个非常大的整数∞,其好处是方便元素比较。由于数组 L 和 R 的元素均有序且按从小到大已完成排列,因此,只需依次比较两个数组中最前面的两个元素 $L[i]$ 和 $R[j]$,将较小的元素存储在 A 数组中。然后重复上述过程,直至两个数组为空。合并排序算法的过程如图 5.1 所示,其中,实线箭头指向逐步划分的过程,虚线箭头指向逐步合并的过程。

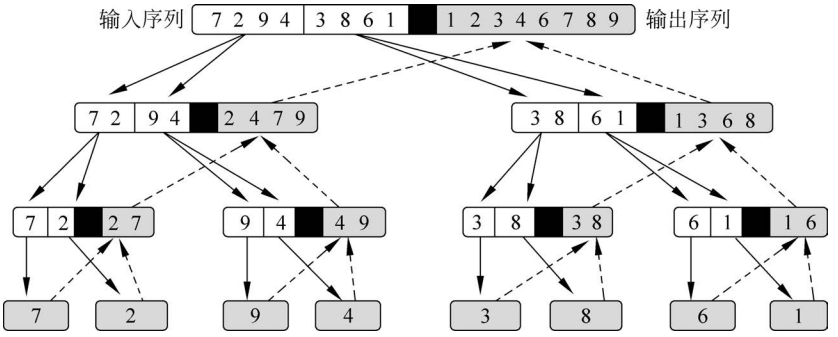


图 5.1　合并排序算法的过程

由于合并排序算法的正确性主要取决于合并算法,如果能证明合并算法的正确性,则合并排序算法的正确性就不难证得。下面证明合并算法的正确性。

考虑合并算法的循环不变量:

在 **for** 循环执行第 k 次迭代前,子数组 $A[p..(k-1)]$有序地存放了 $L[1..(n_1+1)]$和 $R[1..(n_2+1)]$中较小的 $k-p$ 个元素,$L[i]$和 $R[j]$是各自数组中还未复制到数组 A 的最小的元素。那么要证明的是:执行 **for** 循环的第 k 次迭代后,循环不变量依然为真。同样地,这个循环不变量在算法结束时依然为真,从而可以证明算法的正确性。具体步骤如下。

(1) 初始步:在 **for** 循环执行 $k=p$ 之前,子数组 $A[p..(p-1)]$不存在元素,是个空数组。这个空数组包含了 $k-p=0$ 个 L 和 R 的最小元素;同时,$i=j=1$,$L[1]$和 $R[1]$即为还未复制到数组 A 的两个数组中各自最小的元素,因此循环不变量为真。

(2) 归纳步:假设在执行 **for** 循环第 k 次迭代前,循环不变量为真。为了能清楚地了解每次迭代中循环不变量是否为真,不妨假设 $L[i]\leqslant R[j]$。此时 $L[i]$是还未复制到数组 A 的最小元素。因为假设 $A[p..(k-1)]$有序地存放了 $L[1..(n_1+1)]$和 $R[1..(n_2+1)]$中最小的 $k-p$ 个元素,执行 **for** 循环的第 k 次迭代后,在行 13 把 $L[i]$复制到 $A[k]$后,数组 $A[p..k]$将包含 $k-p+1$ 个最小元素。整数 k(在 **for** 循环中增加)和 i(代码行 14)在每次迭代后都发生改变,因而代码行 13、行 14 的执行使得循环不变量为真,即在执行 **for** 循环第 $k+1$ 次迭代前,循环不变量为真。

(3) 终止步:程序结束时 $k=r+1$,子数组 $A[p..(k-1)]$也就是 $A[p..r]$有序地包含了 $k-p=r-p+1$ 个 $L[1..(n_1+1)]$和 $R[1..(n_2+1)]$中最小的元素。数组 L 和 R 总共

包含了 $n_1+n_2+2=r-p+3$ 个元素。此时，除了两个最大的元素没有合并到数组 A 外，L 和 R 的其他元素都已经有序地合并到 A 了，因此循环不变量为真，合并算法正确地把两个有序数组合并成一个有序数组。

在合并排序算法中，令 $T(n)$ 表示对元素个数为 n 的数组进行排序所需要的时间，对数组均匀划分后，可得到两个规模均为 $n/2$ 的子问题。合并排序算法对这两个子问题的解进行合并，最多需要时间 $\Theta(n)$。综上所述，合并排序算法所需要的时间为

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

利用前面介绍的公式法，可得合并排序算法的时间复杂度为 $T(n)=O(n\lg n)$。由于排序问题的下界为 $\Omega(n\lg n)$，因此，合并排序算法是渐近最有效的算法，对解决问题规模大的问题比较有效。

5.3 快速排序

前面介绍了排序问题可以用分治算法来求解，下面设计另一种分治算法——快速排序算法，来求解排序问题。快速排序算法是一种非常有创意的算法，对问题规模比较大的排序问题非常有效，因而被誉为 20 世纪最好的 10 种算法之一。霍尔(C. A. R. Hoare)就因其代表性贡献——快速排序算法，而获得 1980 年的图灵奖。

快速排序算法采用分治算法的思想。令对 n 个元素排序的问题用数组 $A[1..n]$ 表示，考虑一般问题 $A[p..r]$ 的分治求解过程，具体如下。

(1) 把问题 $A[p..r]$ 分解成两个子问题 $A[p..(q-1)]$ 和 $A[(q+1)..r]$，并且满足 $A[p..(q-1)]$ 的元素都小于或等于 $A[q]$，$A[(p+1)..r]$ 的元素都比 $A[q]$ 大，其中，$A[q]$ 称为支点。计算索引 q 的过程就是划分子问题的过程。

(2) 对 $A[p..(q-1)]$ 和 $A[(q+1)..r]$ 这两个子问题分别进行递归求解。

当每个子问题都得到解决，数组也就排好序了。这时，每个元素在正确位置，因而也就没有必要把子问题的解组合在一起了，即分治算法的第三步对于快速排序算法来说，没有必要执行。下面给出快速排序算法 QuickSort(A, p, r) 的伪代码。

```
QuickSort(A, p, r)
1    if p < r then
2        q←Partition(A, p, r)
3        QuickSort(A, p, q - 1)
4        QuickSort(A, q + 1, r)
```

要解决原问题，只需要调用 QuickSort(A, 1, n)。

QuickSort(A, p, r) 的过程非常简单，但其中的 Partition(A, p, r) 过程非常关键。下面来分析该过程。由于在分治算法划分的过程中，需要确定一个支点，这里简单地取 $A[r]$ 为支点。为了将比支点大的元素及比支点小或相等的元素分离，需要对数组进行扫描，即将数组中的元素逐一与 $A[r]$ 进行比较。同时，引入两个变量 i 和 j，标记两个子问题 $A[p..i]$ 和 $A[(i+1)..j]$ 的大小。具体地，Partition(A, p, r) 过程可以描述如下。

```
Partition(A, p, r)
1    x←A[r]
```

```
2    i←p-1
3    for j←p to r-1 do
4        if A[j]≤x then
5            i←i+1
6            A[i] ↔ A[j]
7    A[i+1] ↔ A[r]
8    return i+1
```

下面给出一个例子，分析快速排序算法的运行过程。

图 5.2 给出了一般问题 $A[p..r]$的划分过程。图 5.2(a)给出了执行扫描前的状态，$A[r]=4$ 为支点，将 $A[j]=2$ 与 $A[r]$进行比较，可得 $A[j]<A[r]$，因而 i 往前移动一格。此时 $i=j$，$A[j]$跟自己交换，得到图 5.2(b)。此时 $A[j]=8$ 与 $A[r]$比较，$A[j]>A[r]$，因此不改变，j 往前移动一格，得到图 5.2(c)。重复上述过程，最终得到两个子问题 $A[p..(q-1)]$和 $A[(q+1)..r]$。

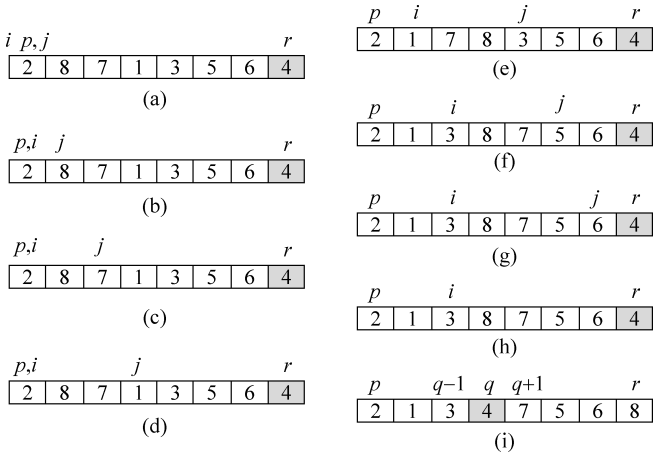


图 5.2　快速排序算法的运行过程

下面给出快速排序的运行时间复杂度分析。

对快速排序算法来说，最坏的情形发生在 Partition(A,p,r)过程所产生的两个子数组中，即一个子数组有 $n-1$ 个元素而另一个有 0 个，有

$$T(n)=T(n-1)+O(n)$$

利用递归树方法展开上式，可以得到

$$\begin{aligned}T(n)&=T(n-1)+O(n)\\&=T(n-2)+O(n)+O(n-1)\\&=\cdots=O(n^2)\end{aligned}$$

这样可以估计出快速排序算法的最坏情形时间复杂度为 $O(n^2)$。然后可以利用替换方法加以证明，读者可以自己证明。

快速排序算法最好情形发生在“二等分”的时候，即 Partition(A,p,r)生成的两个子问题，一个子问题的问题规模为$\lfloor n/2 \rfloor$，另一个子问题的问题规模为$\lceil n/2 \rceil-1$。在这种情况下，快速排序算法的速度是最快的。忽略底和顶，快速排序算法的运行时间表达式为

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

利用第4章介绍的公式法，可以得到快速排序算法最好情形的时间复杂度为 $T(n)=\Omega(n\lg n)$。

下面介绍快速排序算法平均情形时间复杂度的分析。由于快速排序算法只与要排序的序列值的相对顺序有关，与具体的值无关，因此，不妨假设要排序的序列的元素值是各不相同的。为了进一步简化分析，假设序列的每种排列作为算法的输入机会是均等的。把输入序列看成是随机的，这保证了数组的每个元素被选为支点的概率是相等的，即为 $1/n$。假设排序为 q 的元素被选择为支点，则原问题可以分解成两个子问题，一个问题规模为 $q-1$，另一个问题规模为 $n-q$。设 $T(n)$表示对一个元素数为 n 的数组 A 进行快速排序的平均运行时间，则有

$$T(n)=\sum_{q=1}^{n}\frac{1}{n}(T(q-1)+T(n-q)+n-1)=\frac{1}{n}\sum_{q=1}^{n}(T(q-1)+T(n-q))+n-1$$

由于 $\sum_{q=1}^{n}T(q-1)=\sum_{q=1}^{n}T(n-q)$，则有

$$\begin{aligned}T(n)&=\frac{1}{n}\sum_{q=1}^{n}(T(q-1)+T(n-q))+n-1\\&=\frac{2}{n}\sum_{q=1}^{n}T(q-1)+n-1\end{aligned}$$

即

$$nT(n)=2\sum_{q=1}^{n}T(q-1)+n(n-1) \tag{5.1}$$

类似地有

$$(n-1)T(n-1)=2\sum_{q=1}^{n-1}T(q-1)+(n-1)(n-2) \tag{5.2}$$

式(5.1)和式(5.2)相减，可得

$$nT(n)-(n-1)T(n-1)=2T(n-1)+2(n-1)$$

整理可得 $nT(n)=(n+1)T(n-1)+2(n-1)$，两边同时除以 $n(n+1)$，可得

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

令 $D(n)=\dfrac{T(n)}{n+1}$，得

$$D(n)=\begin{cases}0, & n=1\\ D(n-1)+\dfrac{2(n-1)}{n(n+1)}, & n>1\end{cases}$$

利用递归树的方法，可以得

$$D(n)=2\sum_{j=1}^{n}\frac{(j-1)}{j(j+1)}=\Theta(\lg n)$$

故快速排序算法的平均时间复杂度为

$$T(n)=(n+1)\Theta(\lg n)=\Theta(n\lg n)$$

快速排序算法虽然不是渐近最有效的算法，但是当求解问题规模比较大的问题时，效果并不会比合并排序算法差。如果利用随机选择支点的策略，则快速排序算法的效率会更高。

5.4　大整数乘法

设 u 和 v 表示两个 n 位大整数，大整数乘法问题是计算 $u \times v$ 的值。当两个整数的位数比较小时，这个问题很简单；当 n 非常大时，这个问题就变得非常复杂。按照通常的乘法运算，大整数乘法需要 $\Theta(n^2)$ 次位运算，如图 5.3 所示。在密码系统与信息安全、数字信息、加密解密等领域，经常遇到大整数乘法问题，因此，寻找更快的求解算法变得尤为重要。下面介绍求解大整数乘法的分治算法。

假设大整数 u 和 v 分别表示为 n 位二进制形式，每个大整数可分解为高位和低位两部分，每部分为 $n/2$ 位。假设大整数 u 分成 w 和 x 两部分，大整数 v 分成 y 和 z 两部分，如图 5.4 所示。

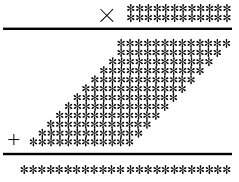


图 5.3　乘法的图形表示

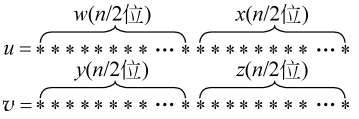


图 5.4　大整数 u 和 v 的划分

因此，大整数 u 和 v 可以分别表示为

$$u = w2^{n/2} + x, \quad v = y2^{n/2} + z$$

则

$$uv = (w2^{n/2} + x)(y2^{n/2} + z) = wy2^n + (wz + xy)2^{n/2} + xz$$

这样原问题可以转化为位数更少的两个整数相乘的问题。值得注意的是，乘以 2^n 表示向左移动 n 位，这个运算耗时 $\Theta(n)$。

令 $T(n)$ 表示两个 n 位整数相乘所需要的计算时间，要计算 uv，需要 4 次两个整数的乘法及 3 次加法运算，其中，后者耗时 $\Theta(n)$，即可计算出 $T(n)$ 为

$$T(n) = \begin{cases} \Theta(1), & n = 1 \\ 4T(n/2) + \Theta(n), & n > 1 \end{cases}$$

利用公式法可知，$T(n) = O(n^2)$。

从时间复杂度来看，这种求解大整数的分治算法比通常的乘法运算没有什么优势。如果能减少子问题的个数 a（这里 $a = 4$），则可以提高算法的效率。事实上，如果利用恒等式：$wz + xy = (w + x)(y + z) - wy - xz$，则可以得到算法 Multiply2Int($u$, v)，其伪代码如下。

```
Multiply2Int(u, v)
1    if |u| = |v| = 1 then return uv
2    else
3        A1 ← Multiply2Int(w, y)
4        A2 ← Multiply2Int(x, z)
5        A3 ← Multiply2Int(w + x, y + z)
6        return A1 2^n + (A3 - A1 - A2)2^(n/2) + A2
```

其中，$|u|$ 表示 u 的位数。Multiply2Int(u, v) 虽然增加了加法运算的次数，但是加法运算的

耗时不多，而且减少了乘法运算的次数，只需要 3 次乘法运算，即只需求解 3 个子问题，因此其时间复杂度可用递归方程表示为

$$T(n)=\begin{cases}\Theta(1), & n=1\\ 3T(n/2)+\Theta(n), & n>1\end{cases}$$

利用公式法得到 $T(n)=O(n^{\lg 3})=O(n^{1.59})$。由此可见，减少子问题两个数可得到具有更低时间复杂度的算法。

5.5 矩阵乘法

给定两个 $n\times n$ 阶矩阵 $\boldsymbol{A}$ 和 $\boldsymbol{B}$，问题是要求它们的乘积，即计算 $\boldsymbol{C}=\boldsymbol{A}\times\boldsymbol{B}$。令 $\boldsymbol{A}=(a_{ik})$，$\boldsymbol{B}=(b_{kj})$，$\boldsymbol{C}=(c_{ij})$，对于这个问题，可以用下式来计算，即

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

一般地，计算 $n\times m$ 矩阵 $\boldsymbol{A}$ 和 $p\times q$ 矩阵 $\boldsymbol{B}$ 乘积的算法，可以描述如下。

```
MatrixMultiply(A, B)
1    if m≠p then
2        print "Two matrices cannot multiply"
3    else
4        for i←1 to n do
5            for j←1 to q do
6                c_ij←0
7                for k←1 to m do
8                    c_ij←c_ij + a_ik × b_kj
9    return C
```

MatrixMultiply($\boldsymbol{A}$，$\boldsymbol{B}$)算法的运行时间复杂度为 $O(nmq)$。是否可以得到更有效的算法呢？现在考虑计算两个 $n\times n$ 阶矩阵乘积，并且假设 $n=2^k$ ($k\geqslant 0$)。如果 $n\geqslant 2$，那么 $\boldsymbol{A}$、$\boldsymbol{B}$，可以被分成 4 个$\frac{n}{2}\times\frac{n}{2}$阶矩阵，分别为

$$\boldsymbol{A}=\begin{pmatrix}\boldsymbol{A}_{11} & \boldsymbol{A}_{12}\\ \boldsymbol{A}_{21} & \boldsymbol{A}_{22}\end{pmatrix},\quad \boldsymbol{B}=\begin{pmatrix}\boldsymbol{B}_{11} & \boldsymbol{B}_{12}\\ \boldsymbol{B}_{21} & \boldsymbol{B}_{22}\end{pmatrix},\quad \boldsymbol{C}=\begin{pmatrix}\boldsymbol{C}_{11} & \boldsymbol{C}_{12}\\ \boldsymbol{C}_{21} & \boldsymbol{C}_{22}\end{pmatrix}$$

利用块矩阵的乘积，矩阵 $\boldsymbol{C}$ 可以表示为

$$\boldsymbol{C}=\begin{pmatrix}\boldsymbol{A}_{11}\boldsymbol{B}_{11}+\boldsymbol{A}_{12}\boldsymbol{B}_{21} & \boldsymbol{A}_{11}\boldsymbol{B}_{12}+\boldsymbol{A}_{12}\boldsymbol{B}_{22}\\ \boldsymbol{A}_{21}\boldsymbol{B}_{11}+\boldsymbol{A}_{22}\boldsymbol{B}_{21} & \boldsymbol{A}_{21}\boldsymbol{B}_{12}+\boldsymbol{A}_{22}\boldsymbol{B}_{22}\end{pmatrix}$$

由上式可知，原问题的求解可以转化为 8 个子问题的求解，子问题中的矩阵规模是$\frac{n}{2}\times\frac{n}{2}$。子问题求解完成后，仍然是一个$\frac{n}{2}\times\frac{n}{2}$阶矩阵。子问题求解完成后，还包括 4 个$\frac{n}{2}\times\frac{n}{2}$阶矩阵相加，因此原问题的计算量为

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

利用公式法，可以知道上述算法的时间复杂度为 $O(n^3)$，这跟两个矩阵直接相乘的计算量没有什么差别。是否可以计算得更快呢？Strassen 通过仔细地研究发现，可以牺牲加、减运算的开销来减少乘法的次数，并提出了 Strassen 算法，其思路如下：

要计算矩阵乘积

$$\boldsymbol{C}=\begin{pmatrix}\boldsymbol{A}_{11} & \boldsymbol{A}_{12}\\ \boldsymbol{A}_{21} & \boldsymbol{A}_{22}\end{pmatrix}\begin{pmatrix}\boldsymbol{B}_{11} & \boldsymbol{B}_{12}\\ \boldsymbol{B}_{21} & \boldsymbol{B}_{22}\end{pmatrix}$$

只需要计算

$$\boldsymbol{C}=\begin{pmatrix}d_1+d_4-d_5+d_7 & d_3+d_5\\ d_2+d_4 & d_1+d_3-d_2+d_6\end{pmatrix}$$

其中

$$\begin{aligned}d_1&=(\boldsymbol{A}_{11}+\boldsymbol{A}_{22})(\boldsymbol{B}_{11}+\boldsymbol{B}_{22})\\ d_2&=(\boldsymbol{A}_{21}+\boldsymbol{A}_{22})\boldsymbol{B}_{11}\\ d_3&=\boldsymbol{A}_{11}(\boldsymbol{B}_{12}-\boldsymbol{B}_{22})\\ d_4&=\boldsymbol{A}_{22}(\boldsymbol{B}_{21}-\boldsymbol{B}_{11})\\ d_5&=(\boldsymbol{A}_{11}+\boldsymbol{A}_{12})\boldsymbol{B}_{22}\\ d_6&=(\boldsymbol{A}_{21}-\boldsymbol{A}_{11})(\boldsymbol{B}_{11}+\boldsymbol{B}_{12})\\ d_7&=(\boldsymbol{A}_{12}-\boldsymbol{A}_{22})(\boldsymbol{B}_{21}+\boldsymbol{B}_{22})\end{aligned}$$

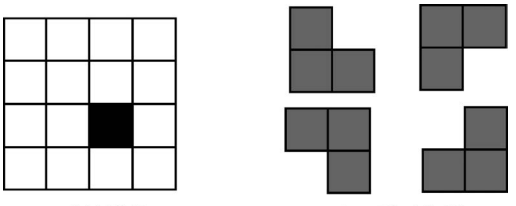
由上可知，两个 $n\times n$ 阶矩阵相乘的计算量是两个 $(n/2)\times(n/2)$ 阶矩阵相乘计算量的 7 倍，加上它们进行加或减运算的 18 倍，加减运算共需要 $\Theta(n^2)$，因此有

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

根据 Strassen 算法，两个 $n\times n$ 阶矩阵相乘原来需要求解 8 个子问题，现在只需要求解 7 个子问题，当然这带来加减法运算量的增加。根据公式法，可得上述递归方程的解为 $T(n)=O(n^{\lg 7})=O(n^{2.81})$，因而 Strassen 算法的效率更高。虽然 Strassen 算法的渐近效率很高，但是这类漂亮的算法因常数系数很大而缺乏应用价值，因此对于小规模问题计算矩阵相乘常用的还是矩阵直接相乘的算法。

5.6　残缺棋盘游戏

残缺的棋盘表示棋盘中有一个格子残缺，图 5.5(a)所示为一个 4×4 的残缺棋盘，其中，黑格表示残缺格。给定一个能覆盖 3 个格子的 L 形三格板，其中，三格板有 4 个放置方向，如图 5.5(b)所示。



(a) 残缺棋盘　　(b) L形三格板

图 5.5　一个 4×4 的残缺棋盘

残缺棋盘游戏问题是给定一个 $2^n \times 2^n$ 的残缺棋盘，求解三格盘的放置方法，使除了残缺格外，棋盘中其他格子可被三格板覆盖，并满足放置的三格板互不重叠。现在可以很容易地计算出：对于一个 $2^n \times 2^n$ 的残缺棋盘，除了残缺格外，共需要放置 $(2^n \times (2^n - 1))/3$ 个三格板。图 5.6 展示了 $2^n \times 2^n$ 的残缺棋盘的三格板铺满过程，可知共需要 5 个三格板。

从图 5.7 可以看出，放置一个三格板后，棋盘中间区域已经铺满。如果把这个三格板看成 3 个残缺格，那么原来的残缺棋盘可以分解为 4 个残缺棋盘。虽然这 4 个残缺棋盘中有 3 个与原来的残缺棋盘不相似，但是通过放置一个三格板，并把三格板所在的格子看作残缺格，这样可以将原来不相似的 3 个棋盘构造成残缺棋盘，这 3 个棋盘的放置方法可以采用类似原来残缺棋盘的放置方法进行求解。当残缺棋盘的规模为 2×2 时，棋盘不再分解，递归终止，这是因为残缺棋盘此时只要放置一个三格板就可以被铺满。

从上述例子中可以得到启示，残缺棋盘游戏的问题可以利用分治算法求解。要完成残缺棋盘游戏，必须定位棋盘和残缺格的位置。当棋盘被一分为四，确定残缺格的位置后，便可以知道该对哪 3 个棋盘补残缺格。例如，对于图 5.7 而言，确定残缺格的位置后可知，左上棋盘的右下角为残缺格，左下棋盘的右上角为残缺格，右上棋盘的左下角为残缺格。

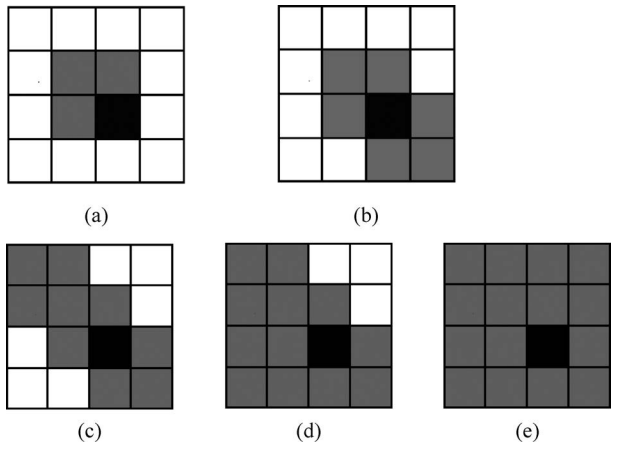


图 5.6　一个残缺棋盘的铺满过程

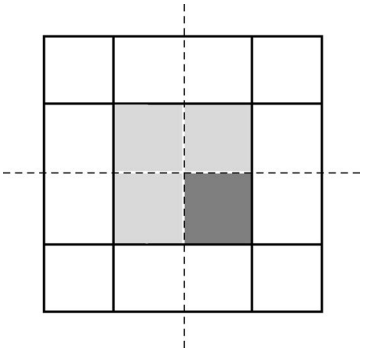


图 5.7　残缺棋盘的划分

令二维整数数组 Board 表示棋盘，其中，Board[0,0]表示棋盘左下角的方格；t 记录放置的三格板的数目；tr 表示残缺棋盘左下角方格所在行；tc 表示棋盘左下角方格所在列；dr 表示残缺格所在行；dc 表示残缺格所在列；size 表示棋盘的行数或列数，则利用分治算法求解残缺棋盘游戏问题的伪代码可以描述为 TileBoard(tr,tc,dr,dc,size)，具体如下。

```
TileBoard(tr, tc, dr, dc, size)
1    if size = 1 return ok
2    tile←tile+1; t←tile
3    s←size/2
4    if dr < tr + s and dc < tc + s then
5        TileBoard(tr, tc, dr, dc, s)
6    else
7        Board[tr + s-1, tc + s - 1]←t
8        TileBoard(tr, tc, tr+s-1, tc+s-1, s)
9    if dr < tr + s and dc ≥ tc + s then
```

```
10        TileBoard(tr, tc + s, dr, dc, s)
11    else
12        Board[tr + s - 1, tc + s]←t
13        TileBoard(tr, tc + s, tr + s - 1, tc + s, s)
14    if dr≥ tr + s and dc < tc + s then
15        TileBoard(tr + s, tc, dr, dc, s)
16    else
17        Board[tr + s, tc + s - 1]←t
18        TileBoard(tr + s, tc, tr + s, tc + s - 1, s)
19    if dr ≥ tr + s and dc ≥ tc + s then
20        TileBoard(tr + s, tc + s, dr, dc, s)
21    else
22        Board[tr + s, tc + s]←t
23        TileBoard(tr + s, tc + s, tr + s, tc + s, s)
```

其中行 1 表示：如果棋盘大小为 $2^1 \times 2^1$，则算法结束，否则，将棋盘划分为 4 个小棋盘。然后，根据残缺格所在的位置，递归调用 4 次，以解决 4 个子问题。其中，在算法实现时，tile 是全局变量，初始为 0。

下面分析 TileBoard(tr,tc,dr,dc,size)的时间复杂度。

令 $T(n)$表示完成一个规模为 $2^n \times 2^n$ 的残缺棋盘游戏所花费的时间，当棋盘规模为 2×2 时，残缺棋盘游戏显然可以在常数时间 $\Theta(1)$内完成。因此，$T(n)$的递归方程为

$$T(n)=\begin{cases}\Theta(1), & n=1\\ 4T(n-2)+\Theta(1), & n>1\end{cases}$$

利用递归树的方法将其展开，可得

$$\begin{aligned}T(n)&=4T(n-1)+\Theta(1)\\&=4[4T(n-2)+\Theta(1)]+\Theta(1)\\&=4^2T(n-2)+4\Theta(1)+\Theta(1)\\&=4^3T(n-3)+4^2\Theta(1)+4\Theta(1)+\Theta(1)\\&\vdots\\&=4^{n-1}T(1)+4^{n-2}\Theta(1)+\cdots+4\Theta(1)+\Theta(1)\\&=\Theta(4^{n-1})\end{aligned}$$

上述例子表明，分治算法通过把问题分解为较小的子问题来解决原问题，简化或减少了求解原问题的计算量。

5.7　快速傅里叶变换

快速傅里叶变换(Fast Fourier Transform，FFT)，是求解离散傅里叶变换(Discrete Fourier Fransform，DFT)的快速算法，在数字信号处理、图像处理、计算大整数乘法、求解偏微分方程等问题上具有广泛的应用。FFT 的影响是如此巨大，以至于它被誉为 20 世纪最好的算法之一。

DFT 的计算公式为

$$b_j=\sum_{k=0}^{n-1}a_k\omega^{kj},\quad j=0,1,\cdots,n-1$$

其中，$\omega=e^{2\pi i/n}$ 为 n 次单位元根，$i=\sqrt{-1}$；$A=\langle a_0,a_1,\cdots,a_{n-1}\rangle$ 为已知。

DFT 逆变换为

$$a_k=\frac{1}{n}\sum_{j=0}^{n-1}b_j\omega^{-jk},\quad k=0,1,\cdots,n-1$$

如果 DFT 的快速求解算法找到了，那么其逆变换同样可以快速求得。DFT 与多项式的计算关系紧密，事实上，考虑多项式 $p(x)$

$$p(x)=a_0+a_1x+a_2x^2+\cdots+a_{n-1}x^{n-1}$$

则 DFT 的 b_j 是 $p(x)$在 $x=\omega^j$ 处的值。前面介绍过 Horner 法则，计算 $p(x)$在某点的值，其时间复杂度为 $O(n)$，因此计算 $b_j(j=0,1,\cdots,n-1)$的时间复杂度为 $O(n^2)$，而利用 FFT，其时间复杂度仅为 $O(n\lg n)$。下面介绍 FFT。

将多项式 $p(x)$分解为奇次幂部分和偶次幂部分，为

$$\begin{aligned}p(x)&=(a_1x+a_3x^3+\cdots+a_{n-1}x^{n-1})+(a_0+a_2x^2+\cdots+a_{n-2}x^{n-2})\\&=(a_1+a_3x^2+\cdots+a_{n-1}x^{n-2})x+(a_0+a_2x^2+\cdots+a_{n-2}x^{n-2})\end{aligned}$$

令 $y=x^2$，则有

$$\begin{aligned}p(x)&=(a_1+a_3y+\cdots+a_{n-1}y^{\frac{n}{2}-1})x+(a_0+a_2y+\cdots+a_{n-2}y^{\frac{n}{2}-1})\\&=q(y)x+r(y)\end{aligned}$$

因此 $p(x)$在 $x=\omega^j\left(j=0,1,\cdots,\frac{n}{2}-1\right)$处的值可表示为

$$p(\omega^j)=q(\omega^{2j})\omega^j+r(\omega^{2j})$$

根据复数的性质 $\omega^{j+\frac{n}{2}}=-\omega^j$，有

$$p(\omega^{j+\frac{n}{2}})=-q(\omega^{2j})\omega^j+r(\omega^{2j})$$

上述推导过程说明：求解次数为 n 的多项式 $p(x)$在 $x=\omega^j$ 处的值，可以转化为求两个次数为 $n/2$ 的多项式 $q(x)$和 $r(x)$在 $x=(\omega^2)^j$ 处的值的问题。如果 n 是 2 的幂，则可以利用分治算法继续分解，直到 $p(x)$是常数为止。下面给出算法 RecursiveFFT(A,ω)的伪代码，具体如下。

```
RecursiveFFT(A, ω)
1    if n = 1 then return A
2    x←ω^0
3    A_2←<a_0, a_2, …, a_{n-2}>
4    A_1←<a_1, a_3, …, a_{n-1}>
5    q←RecursiveFFT(A_2, ω^2)
6    r←RecursiveFFT(A_1, ω^2)
7    for k←0 to (n/2) - 1 do
8        b_k←x × r + q
9        b_{k+n/2}←- x × r + q
10       x←xω
11   return b
```

其中，$b=\langle b_0,b_1,\cdots,b_{n-1}\rangle$即为所求。令 $T(n)$表示计算 b 需要的时间，则有

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

利用公式法可得 $T(n)=O(n\lg n)$。

DFT 与其逆变换有相同的形式，因此 FFT 也可以用来计算 DFT 的逆变换。此外，利用 FFT 还可以在 $O(n\lg n)$时间内计算两个多项式的乘积。值得注意的是，FFT 需要考虑多项式 $p(x)$在一些特殊点 $x=\omega^j$ $(j=0,1,\cdots,n-1)$处的计算。

5.8　小结

本章内容主要参考文献[2,5,7]，介绍了分治算法的设计及其时间复杂度的分析，以及分治算法的应用。提高分治算法的效率是算法设计的关键，对此，一种方法是减少子问题的个数来提高分治法的效率，另一种方法是减少划分问题及合并子问题解的计算量。当然，这里面也有一对矛盾的关系，即减少子问题的个数，可能会增加划分问题及合并子问题解的计算量，这就需要根据具体问题来平衡考虑。还有一种方法是，避免公共子问题的重复计算，这也是第 6 章动态规划算法里要考虑的问题。分治算法是算法领域最基本的技术。虽然分治算法有时求解问题的效率不太高，但是许多有效的算法都是基于分治的思想。

习题

5-1　将合并排序算法改写成迭代算法(非递归算法)，尽可能地使算法更有效。

5-2　给出算法 Strassen 计算下列矩阵乘积的过程。

$$\begin{pmatrix}3 & 1\\ 4 & -1\end{pmatrix}\begin{pmatrix}2 & -5\\ 6 & -3\end{pmatrix}$$

5-3　给定一个有序数组 $A[1..n]$，以及一个元素 x，设计一种寻找 x 的分治算法，并分析其时间复杂度，同时要求返回 x 在数组中的位置。

5-4　设计一种求 n 个元素中最小值和最大值的迭代算法，要求仅比较 $3n/2-2$ 次，其中，n 表示 2 的幂。

5-5　设计一种求 n 个整数数组 $A[1..n]$所有元素和的分治算法。求解思路：将输入元素近似地划分成两半入手。

5-6　给定 n 个整数的数组 $A[1..n]$，以及一个整数 x，设计一种分治算法，求出 x 在数组 A 中出现的次数，并分析所设计算法的时间复杂度。

5-7　给出一个由 n 个元素组成的数组 $A[1..n]$，以及两个元素 x_1 和 x_2，设计一种寻找两个元素位置的分治算法。

5-8　按照下述思路修改算法 MergeSort：首先把输入数组 $A[p..r]$划分成 4 个部分 A_1、A_2、A_3 和 A_4，并取代原来的两部分；然后分别对每部分进行递归排序；最后将 4 个已排序部分合并，得到一个有序的数组。为了简单起见，假设 n 是 4 的幂。请设计修改算法并分析其时间复杂度。

5-9　给定 n 个互不相同元素的数组 $A[1..n]$，要求设计找出数组中第 k 小元素的分治算法，并分析其时间复杂度。

5-10　修改算法 QuickSort，使它能够求解选择问题(习题 5-9)，并分析修改后算法的最坏情形及平均情形时间复杂度。

5-11 将算法 QuickSort 转化为迭代算法。

5-12 考虑在具有 n 个互不相同元素的数组 $A[1..n]$ 中找出前 k 个最小元素的问题，其中，k 不是常量，而是输入数据的一部分。可以用排序算法解此问题并返回 $A[1..k]$，然而耗费的时间为 $O(n\lg n)$，试设计一种时间复杂度为 $\Theta(n)$ 的算法。

5-13 设 $x=a+b\mathrm{i}$ 和 $y=c+d\mathrm{i}$ 是两个复数。只要 4 次乘法就可以很容易地计算乘积 xy，也就是 $xy=(ac-bd)+(ad+bc)\mathrm{i}$。设计一种算法，只用 3 次乘法就可以计算出 xy。

5-14 已知序列 $A=\langle a_0,a_1,a_2,a_3\rangle$，给出算法 RecursiveFFT 的计算过程及结果。

5-15 设计一种分治算法，使之能判断两个二叉树 T_1 和 T_2 是否相同。

5-16 设计一种分治算法，使之能计算一棵二叉树的高度。

5-17 设计一种分治算法，在一个具有 n 个数的数组中找出第二大的元素，并分析算法的时间复杂度。

5-18 说明如何用三格板来平铺没有方格缺失的 $2i\times 3j$ 平板，其中，i 和 j 都是正整数。

实验题

5-19 编写程序实现残缺棋盘游戏算法，并用图形演示。

5-20 分别将插入排序、选择排序、合并排序和快速排序算法进行编程实现，并使用实验分析法比较这些算法的效率。

5-21 继续改进石材切割问题。