第3章　分 治 策 略

3.1　一 般 方 法

给定函数来计算n个输入，**分治策略**（divide-and-conquer strategy）建议将输入分成k个子集，$1<k\leqslant n$，得到k个子问题。我们必须解决这些子问题，然后找到一个方法将子问题的解拼成原问题的解。如果子问题还是太大的话，可以重复进行分治。通常分治策略产生的子问题与原问题的类型相同。在这种情况下，重复进行分治非常自然地就可以以递归的方式表示。这样一来可以生成同一类型越来越小的子问题，直到子问题到足够小可以直接解决为止。

例 3.1[检测假硬币]　现在袋子中有 16 个硬币，我们知道其中的一个可能是假的。另外你知道假的硬币比真的硬币轻。你的任务是确定袋子中是否有一枚假的硬币。为了完成这个任务，你可以使用一台仪器来比较两组硬币的重量，并且告诉你那组硬币轻一些或者两组一样重。

我们可以比较硬币 1 和硬币 2。如果硬币 1 比硬币 2 轻，那么硬币 1 是假的，我们也完成了任务。如果硬币 2 比硬币 1 轻，那么硬币 2 是假的。如果两个的重量一样，那么比较硬币 3 和硬币 4。同样，如果其中一个硬币轻一些，就检测出一枚假硬币。如果没有，就比较硬币 5 和硬币 6。一直这么做，可以最多通过八次比较就确定袋子中是否包括假硬币。这个过程同时找出了那枚假硬币。

另外一种做法可以使用分治策略。假设把 16 个硬币的实例看成是一个大实例。第一步，我们将原始问题分成两个或者更小的实例。让我们将 16 个硬币的问题分成两个 8 个硬币的问题，我们的分法是随机选择 8 个硬币作为第一个实例（称为A），然后剩下的 8 个硬币就是第二个实例（称为B）。第二步中，我们需要确定A或B中是否有假硬币。这一步中，我们比较硬币集合A和B的重量。如果它们的重量不同，那么一定存在假硬币，并且在那个较轻的集合中。最后，在第三步中，我们得到第二步的结论，并用它来回答原始 16 个硬币的问题。对于假硬币问题，第三步是很容易的。这样通过一次重量比较，就可以完成检测是否存在假硬币的任务。

现在我们假设还需要确定哪个是假硬币。我们定义“小”实例是包括 2 个或者 3 个硬币的情况。注意，如果只有一个硬币，那我们无法判断它是不是假的。所有其他的实例都是大的。对于小实例，可以通过比较 1 枚硬币与其他的 1 枚或 2 枚硬币，也就是最多 2 次比较就可以判断哪个是假硬币。

16 个硬币的实例是大实例。所以它被分为了 2 个 8 个硬币的实例。比较这两个实例的重量，我们可以判断是否存在假硬币。如果不存在，算法终止。如果存在，继续那个存在假硬币的实例。假设B是较轻的那个集合。它再进一步被分为 2 个 4 硬币的集合。称它们为B_1和B_2。比较这两个集合。其中的一个集合一定更轻一些。如果B_1更轻，假硬币就在B_1中，

然后B_1就被分为 2 个 2 硬币的集合。称它们为B_{1a}和B_{2a}。比较这两个集合，我们继续处理那个更轻的集合。因为较轻的集合只有 2 枚硬币，它是一个小实例。比较这个集合里的两枚硬币的重量，可以确定哪个更轻。更轻的就是假硬币。 ■

更准确地，假设当输入分成两个与原始问题同类型的子问题时我们考虑分治策略。对于在这里讨论的许多问题，这样的划分是很典型的。我们可以写出一个控制抽象，来看分治策略的算法是什么样子的。**控制抽象**（control abstraction）是指一个过程，其控制流清晰，但是主要的操作需要由其他的过程来定义，从而省略其详细的意义。DAndC 是一个函数（程序 3.1），其初始调用是 DAndC(P)，其中 P 是待解决的问题。

```
Type DAndC(P)
{
    if Small(P) return S(P);
    else{
        将 P 划分为更小的实例 P1,P2,…,Pk,k≥1;
        在每个子问题上执行 DAndC;
        return Combine(DAndC(P1), DAndC(P2),…,DAndC(Pk));
    }
}
```

程序 3.1　分治策略的控制抽象

Small(P)是一个布尔函数用来判断输入是否足够小可以直接解问题而不需要分割。如果是这样的，那么调用函数 S。否则，问题 P 就被分成子问题。这些子问题$P_1, P_2, \cdots, P_k$由递归调用 DAndC 来解。将k个子问题的解合并成 P 的解是通过函数 Combine 来完成的。如果 P 的大小是n，k个子问题的大小分别是$n_1, n_2, \cdots, n_k$，那么计算 DAndC 的时间可以用如下的递归等式来表示：

$$T(n) = \begin{cases} g(n) & n \text{ 较小} \\ T(n_1) + T(n_2) + \cdots + T(n_k) + f(n) & \text{否则} \end{cases} \tag{3.1}$$

其中$T(n)$是对任意大小是n的输入 DAndC 的运行时间，$g(n)$是直接计算小输入所花的时间。函数$f(n)$是分割 P 以及合并子问题的解所花的时间。对于基于分治策略将原问题分成同类型的子问题的算法，很自然地首先用递归来描述算法。

解递归关系

很多分治算法的复杂度是如下形式的：

$$T(n) = \begin{cases} T(1) & n = 1 \\ aT(n/b) + f(n) & n > 1 \end{cases} \tag{3.2}$$

其中a和b是已知常数。假设$T(1)$已知并且n是b的幂（即$n = b^k$）。

一种解上述递归式的方法是**替换法**（substitution method）。这种方法就是不断替换等式右边的T，直到不再出现T为止。

例 3.2　考虑当$a = 2$并且$b = 2$。令$T(1) = 2$并且$f(n) = n$。有

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &= 2[2T(n/4) + n/2] + n \\ &= 4T(n/4) + 2n \end{aligned}$$

$$= 4[2T(n/8) + n/4] + 2n$$
$$= 8T(n/8) + 3n$$
$$\vdots$$

一般地，我们可以看到对于任意$\log_2 n \geqslant i \geqslant 1$，$T(n) = 2^i T(n/2^i) + in$。特别地，$T(n) = 2^{\log_2 n} T\left(n/2^{\log_2 n}\right) + n\log_2 n$，当$i = \log_2 n$时。因此$T(n) = nT(1) + n\log_2 n = n\log_2 n + 2n$。

■

从递归式（3.2）使用替换法，可以得到

$$T(n) = n^{\log_b^a}\,[T(1) + u(n)]$$

其中$u(n) = \sum_{j=1}^{k} h\left(b^j\right)$并且$h(n) = f(n)/n^{\log_b^a}$。表 3.1 列出了在不同的$h(n)$值下$u(n)$的渐进值。这个表可以帮助我们在分析分治算法时直接得到非常多的递归式$T(n)$的渐进值。

表 3.1　对于不同的 h(n)值 u(n)的渐进值

h(n)	u(n)
$O(n^r)$, r<0	O(1)
$\Theta((\log n)^i)$, i≥0	$\Theta((\log n)^{i+1}/(i+1))$
$\Omega(n^r)$, r>0	Θ(h(n))

下面用这个表来看一些例子。

例 3.3　考虑当n是 2 的幂时下列递归式：

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

与式（3.2）相比，得到$a = 1$，$b = 2$并且$f(n) = c$。所以，$\log_b(a) = 0$并且$h(n) = \frac{f(n)}{n^{\log_b a}} = c = c(\log n)^0 = \theta((\log n)^0)$。从表 3.1 可得$u(n) = \theta(\log n)$。因此$T(n) = n^{\log_b a}[T(1) + \theta(\log n)] = \theta(\log n)$。 ■

例 3.4　下面来考虑$a = 2, b = 2$并且$f(n) = cn$。此时，$\log_b(a) = 1$并且$h(n) = \frac{f(n)}{n} = c = \theta((\log n)^0)$。因此，$u(n) = \theta(\log n)$并且$T(n) = n[T(1) + \theta(\log n)] = \theta(n\log n)$。 ■

例 3.5　这个例子我们考虑如下递归式$T(n) = 7T(n/2) + 18n^2$，n≥2 并且n是 2 的幂。我们有$a = 7$，$b = 2$并且$f(n) = 18n^2$。所以$\log_b a = \log_2 7 \approx 2.8$并且$h(n) = \frac{18n^2}{n^{\log_2 7}} = 18n^{2-\log_2 7} = O(n^r)$，其中$r = 2 - \log_2 7 < 0$。因此$u(n) = O(1)$。假设$T(1)$是常数，有：

$$T(n) = n^{\log_2 7}[T(1) + O(1)] = \theta\left(n^{\log_2 7}\right)$$

■

例 3.6　最后一个例子，来考虑递归式$T(n) = 9T(n/3) + 4n^6$，n≥3 并且n是 3 的幂。与式（3.2）相比，得到$a = 9, b = 3$并且$f(n) = 4n^6$。因此$\log_b a = 2$并且$h(n) = \frac{4n^6}{n^2} = \Omega(n^4)$。由表 3.1，有$u(n) = \theta\left(h(n)\right) = \theta(n^4)$。因此

$$T(n) = n^2[T(1) + \theta(n^4)] = \theta(n^6)$$

■

习题

1. 将例 3.1 中的分治方法扩展到包含 n>1 枚假硬币的情况。此时需要多少次重量比较？

2. 考虑例 3.1 中的假硬币问题。假设不是“假硬币比真硬币轻”，而是“假硬币和真硬币的重量不一样”。另外假设硬币袋中有 n 个硬币。
3. 设计一个分治算法，或者输出“不存在假硬币”，或者确认哪个是假硬币。你的算法应该递归地将大问题分成两个小问题。为了确认哪个是假硬币需要多少次重量比较（如果存在这样的硬币）？
4. 重复做习题 3，这次将大问题分成三个小问题。
5. 在下述 a、b 和 f(n)的情况下，解递归式（3.2）。
   （a）a=1、b=2 并且 f(n)=cn
   （b）a=5、b=4 并且 $f(n)=cn^2$
   （c）a=28、b=3 并且 $f(n)=cn^3$
6. 用替换法解下述解递归式：
   （a）习题 5 中的所有递归式。
   （b）$T(n)=\begin{cases}1 & n\leqslant 4\\ T(\sqrt{n})+c & n>4\end{cases}$
   （c）$T(n)=\begin{cases}1 & n\leqslant 4\\ 2T(\sqrt{n})+\log n & n>4\end{cases}$
   （d）$T(n)=\begin{cases}1 & n\leqslant 4\\ 2T(\sqrt{n})+\dfrac{\log n}{\log\log n} & n>4\end{cases}$

3.2　残 缺 棋 盘

残缺棋盘（defective chessboard）是指有 $2^k\times 2^k$ 个方格的棋盘中恰好有一个方格是坏的。图 3.1 给出了所有 k≤2 的可能的残缺棋盘。坏掉的方格用阴影表示。注意，当 k = 0 时，只有一种可能的残缺棋盘（图 3.1(a)）。事实上，对于任意 k，共有 2^{2k} 种不同的残缺棋盘。

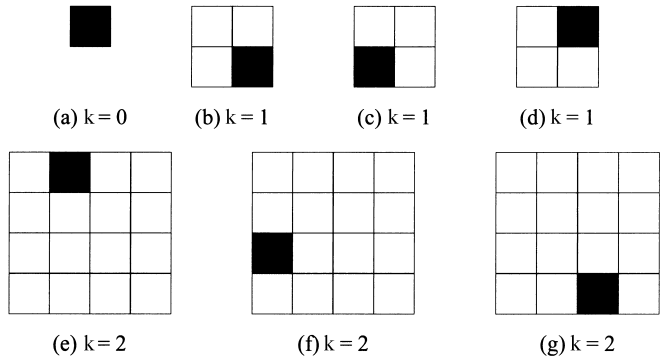


图 3.1　有残缺的棋盘

在残缺棋盘问题中，我们要求用三方块把残缺棋盘铺满（如图 3.2 所示）。我们要求在铺的过程中三方块不能重叠，不能盖住残缺的方块，并且要盖满其他所有的方块。在这样

的条件下，需要用$(2^{2k}-1)/3$块三方块。可以证明$(2^{2k}-1)/3$是一个整数。k=0 的残缺棋盘是很容易解的，因为没有任何坏掉的方块，使用的三方块数是 0。当 k=1 时，一共有 3 块好的方块恰好可以被一块三方块覆盖，如图 3.2 所示。

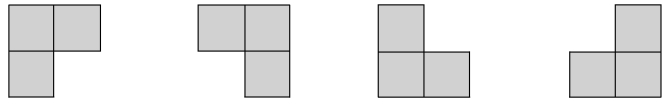
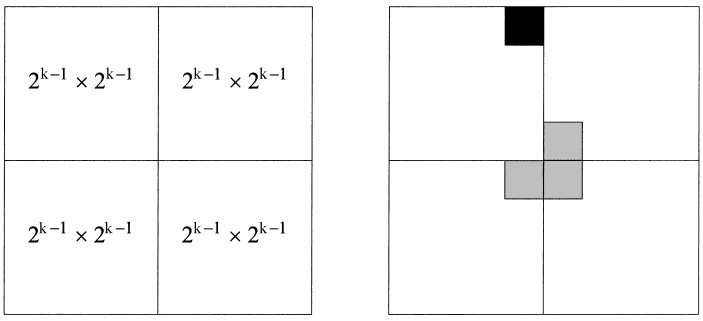


图 3.2　不同方向的三方块

用分治法可以解决残缺棋盘问题。顾名思义，分治法会将$2^k\times2^k$残缺棋盘问题划分为规模小一些的问题。对$2^k\times2^k$棋盘的一种自然的分法是将其分为四个$2^{k-1}\times2^{k-1}$的棋盘，如图 3.3(a)所示。注意，当这样划分时，只有一个小棋盘中有残缺（因为原始的$2^k\times2^k$棋盘中只有一块残缺）。因此，用三方块铺四个小棋盘会遇到一个是$2^{k-1}\times2^{k-1}$的残缺棋盘。为了将剩余的三个棋盘也变成残缺棋盘，我们用一个三方块来盖住由那三个棋盘构成的角。图 3.3(b)显示了当原始的$2^k\times2^k$棋盘的残缺方格出现在左上角的$2^{k-1}\times2^{k-1}$棋盘时如何放置。我们可以递归地使用这个划分技术来铺整个$2^k\times2^k$残缺棋盘。当棋盘的大小缩减为 1×1 时，递归终止。此时棋盘中唯一的那个方格有残缺，不需要放置任何三方块。



(a) 划分　　(b) 放好的三方块

图 3.3　划分$2^k\times2^k$的棋盘

使用程序 3.1 给出的分治控制抽象中的术语，当 k=0 时（即我们处理 1×1 的残缺棋盘）Small(P)为真；将 P 分为小问题实例是通过将 P 划分为四个$2^{k-1}\times2^{k-1}$的残缺棋盘实例，并且将一个三方块放置在P的中心来为三个本来不含残缺方格的$2^{k-1}\times2^{k-1}$实例制造一个残缺方格；合并四个子问题的解并不需要什么额外的工作。

可以将这个分治算法实现成 C++的递归函数 TileBoard（程序 3.2）。这个函数使用两个全局变量 tile 和 board。其中 board 是一个二维的整数数组代表棋盘，board[0][0]表示整个棋盘的最左上角的方格；tile 是一个整数变量，初始值为 1，代表下一个使用的三方块的下标。

程序初始调用是 TileBoard(0,0,dRow,dCol,size)其中 size 为2^k，dRow 和 dCol 是棋盘中残缺方格的行号和列号。为覆盖残缺棋盘需要的填充块一共是$(size^2-1)/3$块。函数 TileBoard 将这些填充块用 1 到$(size^2-1)/3$来表示，并且将棋盘中正常方格的值设为覆盖它的那个填充块的值。

```
//全局变量
int **board;    //棋盘
int tile;       //当前可用的填充块

void TileBoard(int topRow, int topColumn,
               int defectRow, int defectColumn, int size)
{//topRow 是棋盘左上角的行号, topColumn 是棋盘左上角的列号
//defectRow 是残缺块的行号, defectColumn 是残缺块的列号
//size 是棋盘一边的长度
    if (size == 1) return;
    int tileToUse = tile++, quadrantSize = size/2;

    //填充左上四分之一
    if (defectRow < topRow + quadrantSize &&
        defectColumn < topColumn + quadrantSize)
        //残缺在此四分之一中
        TileBoard(topRow, topColumn, defectRow, defectColumn,
                  quadrantSize);
    else
    {//在这四分之一中没有残缺
        //将填充块放在右下角
        board[topRow + quadrantSize - 1]
             [topColumn + quadrantSize - 1] = tileToUse;
        //填剩下的部分
        TileBoard(topRow, topColumn, topRow + quadrantSize - 1,
                  topColumn + quadrantSize - 1, );
    }
    //对其余 3 个四分之一块的代码相似
}
```

程序 3.2　填充残缺棋盘

令$t(k)$表示 TileBoard 来铺满整个 $2^k \times 2^k$ 残缺棋盘所需要的时间。当$k = 0$时，size 为 1，需要常数时间d。当$k > 0$时，需要四个递归调用。这些调用用时$4t(k-1)$。除此之外，需要花时间在 if 语句以及铺三个正常的方格。令这些时间为常数c。我们得到如下的递归式：

$$t(k) = \begin{cases} d & k = 0 \\ 4t(k-1) + c & k > 0 \end{cases} \tag{3.3}$$

可以用替换法得到$t(k) = \Theta(4^k) = \Theta(\text{需要三方块的个数})$。因为至少需要$\Theta(1)$的时间来放每个三方块，所以不可能得到比这个分治算法更快的算法。

习题

1. 考虑一个8×8的残缺棋盘，设$(0,3)$位置上的方格是有残缺的。求程序 3.2 给出的铺法。
2. 考虑一个8×8的残缺棋盘，设$(2,2)$位置上的方格是有残缺的。求程序 3.2 给出的铺法。
3. 编写一个完整的残缺棋盘程序。包括欢迎用户来到这个程序；输入棋盘的大小以及残缺方格的位置；输出铺好的棋盘。输出应该用彩色三方块来表示。我们要求相邻的三方块

的颜色都不相同。因为棋盘是一个平面图，最多可以只用四种颜色。不过在本习题中，不要求使用颜色的数目，你只要有一个贪心的策略尽量少使用颜色即可。

4. 使用替换法解式 3.3 中的递归式。

3.3 二 分 搜 索

令a_i，1≤i≤n，是已经按升序排列好的n个元素。现在来考虑该列表中是否存在元素x。如果列表中存在x，我们需要确定一个值j使得$a_j = x$。如果列表中不存在x，那么令j为 0。令$P = (n, a_i, \cdots, a_l, x)$表示这个搜索问题的任意实例（n是列表中元素的个数，$a_i, \cdots, a_l$是列表元素，x是要搜索的元素）。

可以用分治来解这个问题。令 Small(p)当$n = 1$的时候为真。此时，如果$x = a_i$那么 S(P)等于i；否则为 0。那么$g(1) = \Theta(1)$。如果P中的元素超过 1 个，那么通过以下方式将其分成更小的问题。选一个下标q（q在[i, l]范围中）并且比较x和a_q。有三种可能：（1）$x = a_q$：此时P已有解。（2）$x < a_q$：此时只需要在子列表$a_i, \cdots, a_{q-1}$中搜索x。因此问题可以缩减为$(q - 1, a_i, \cdots, a_{q-1}, x)$。（3）$x > a_q$：此时只需要在子列表$a_{q+1}, \cdots, a_l$中搜索x。因此问题可以缩减为$(l - q, a_{q+1}, \cdots, a_l, x)$。

在这个例子中，任意给定问题P都被分成了更小的问题。划分用时$\Theta(1)$。在与a_q比较之后，剩下需要求解的实例可以仍然用这个分支策略来解。如果能选择q，使得a_q是中间的元素（即$q = \lfloor (n + 1) \rfloor / 2$），那么这样的搜索算法被称为二分搜索。注意，新的子问题的解也是原问题的解，不需要其他的结合的过程。程序 3.3 给出了这个算法的 C++实现，其中函数 BinSrch 有四个输入 a[]、i、l 和 x。函数的初始调用是 BinSrch (a,1,n,x)。

```
int BinSrch (Type a[], int i, int l, Type x)
//已知数组 a[i:l]中的 n 个元素呈降序排列, 1<=i<=l, 判断 x 是否存在
//如果存在返回 j 使得 x == a[j], 否则返回 0
{
    if (l == i) { //If Small(P)
        if (x==a[i]) return i;
        else return 0;
    }
    else{ //将 P 化解为更小的子问题
        int mid = (i+l)/2;
        if (x == a[mid]) return mid;
        else if (x < a[mid]) return BinSrch (a, i, mid-1, x);
        else return BinSrch (a, mid+1, l, x);
    }
}
```

程序 3.3　递归的二分搜索

程序 3.4 给出了这个算法的非递归版本。函数 BinSearch 有三个输入 a[]、n 和 x。while 循环将持续直到没有元素需要检查。函数结束的时候，如果不存在 x 则返回 0，否则返回

满足 a[j]==x 的 j。

```
int BinSearch(Type a[], int n, Type x)
//已知数组 a[i:l]中的 n 个元素呈降序排列, 1<=i<=l, 判断 x 是否存在
//如果存在返回 j 使得 x == a[j], 否则返回 0
{
    int low = 1, high = n;
    while (low <= high){
        int mid = (low + high)/2;
        if (x < a[mid]) high = mid - 1;
        else if (x > a[mid]) low = mid + 1;
        else return(mid);
    }
    return(0);
}
```

程序 3.4　循环的二分搜索

BinSearch 是个算法吗？我们必须确定所有 x 与 a[mid]之间的比较都是有定义的。进行比较的这些关系运算需要恰当的定义。BinSearch 会终止吗？我们观察到 low 和 high 是两个整数变量，并且每次循环之后，或者找到 x，或者 low 被至少增加 1，或者 high 被至少减少 1。所以有两个整数序列在不断的向一起靠拢，最终如果 x 不存在，经过有限步，low 必将超过 high 而导致函数终止。

例 3.7　我们来考虑下面的 14 个数：

$$-15,-6,0,7,9,23,54,82,101,112,125,131,142,151$$

被放入 a[1:14]。下面在这个数组上模拟 BinSearch 的执行。我们只需要记录 low、high 和 mid 就可以追踪 BinSearch 的执行。我们尝试三个 x 的值：151、−14和9，其中有两次成功的搜索和一次失败的搜索。表 3.2 中列出了在这三种输入情况下 BinSearch 的执行。 ■

表 3.2　在 14 个元素上进行二分搜索的三个例子

x=151	low	nigh	mid	x=−14	low	high	mid
	1	14	7		1	14	7
	8	14	11		1	6	3
	12	14	13		1	2	1
	14	14	14		2	2	2
			找到		2	1	没找到

x=9	low	high	mid
	1	14	7
	1	6	3
	4	6	5
			找到

这些例子让我们对算法 3.4 有了些信心，但是它们不能证明该算法是正确的。证明这些算法的正确性是很有用的，因为它可以证明算法在任何输入的情况下都是正确的，而测试不能保证这一点。不幸的是，程序的证明是非常困难的，有时一个完整的程序证明是程

序本身的几倍长。这里只给出 BinSearch 的非正式“证明”。

定理 3.1 函数 BinSearch(a,n,x)是正确的。

证明：假设所有的语句都正确执行，例如 x>a[mid]的比较也正确执行。初始时，low==1，high==n，n>=0，并且 a[1]<=a[2]<=…<=a[n]。如果 n==0，不进 while 循环并返回 0。否则，每次执行循环的时候，可能被检查相等的元素有 a[low]，a[low+1]，…，a[mid]，…，a[high]。如果 x==a[mid]，那么算法成功终止。否则这个范围会缩小，或者增加 low 到 mid+1，或者减小 high 到 mid−1。显然缩小范围不会影响搜索的结果。如果 low 比 high 变得更大，那么 x 不存在，从循环退出。 ■

注意，为了完全测试二分搜索，不需要考虑 a[1:n]中的具体值。通过变化 x 的值，可以看到 BinSearch 的各种执行序列，因此并不需要变化 a 的值。为了测试所有成功的搜索，x 必须取到 a 的n个值。因此测试 BinSearch 的复杂度是$2n+1$。

下面来分析 BinSearch 的执行情况。从算法需要的时间和空间两个角度来说明。对于 BinSearch 来说，需要存n个元素的数组，还有 low、high、mid 和 x 四个变量，所以一共是$n+4$个位置。时间则分为三种情况讨论：最好、平均和最差情况。

我们先确定在上个数据集上 BinSearch 的运行时间。我们观察到算法中唯一的运算是比较，以及一些算术和数据移动。我们主要来看 x 与 a[]中元素的比较，其他操作的次数与比较的次数的数量级是一样的。x 与 a[]中元素的比较被称为是**元素比较**（element comparisons）。假设只要一次比较就可以判断出 if 语句是哪种情况成立。那么找到以下 14 个元素的元素比较次数分别是：

a:	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]
元素:	−15	−6	0	7	9	23	54	82	101	112	125	131	142	151
比较:	3	4	2	4	3	4	1	4	3	4	2	4	3	4

没有需要超过 4 次以上比较次数的元素。我们可以通过累加找到 14 个元素所需次数之和再求平均的办法算出成功搜索平均需要的比较次数是 45/14，大约为 3.21。根据 x 的值的不同，共有 15 种可能的不成功搜索。如果 x<a[1]，算法需要 3 次比较就可以知道 x 不存在。对于其他的情况，BinSearch 需要比较 4 次。因此每个不成功搜索所需要的平均比较次数是：$(3+14*4)/15 = 59/15 \approx 3.93$。

上述的分析对任意 14 元素的有序序列都是成立的。但我们想得到的是一个关于n的式。可以考虑对于所有可能的 x，BinSearch 所产生的 mid 的值的序列来推导这个式，也方便我们更好地理解算法。这些值可以用一个二叉决策树来表示，每一个节点的值就是 mid 的值。例如，当 n=14 时，图 3.4 就给出一个跟踪 BinSearch 所产生的 mid 的值的决策树。

第一次比较是在 x 与 a[7]之间。如果 x<a[7]，那么下一个比较是与 a[3]；类似地，如果 x>a[7]，那么下一个比较是与 a[11]。树上的每一条路径都表示二分搜索法的一个比较的序列。如果 x 存在，那么算法会在圆形节点处终结并且节点内的下标就是数组中对应等于 x 的值的位置。如果 x 不存在，那么算法会在方形节点处终结。圆形节点叫**内**（internal）节点，方形节点叫**外**（external）节点。

定理 3.2 如果n的取值范围是$[2^{k-1}, 2^k)$，那么 BinSearch 最多做k次元素比较即可完成一次成功的搜索，并且做出$k-1$次或k次元素比较完成一次不成功的搜索。（换句话说，成

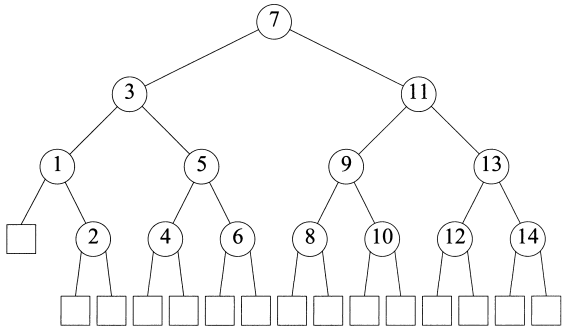


图 3.4　二分搜索的二叉决策树 n = 14

功搜索的时间代价是$O(\log n)$，不成功搜索的代价是$\theta(\log n)$。）

证明：我们来考虑描述 BinSearch 在n个元素上执行的二叉决策树。所有的成功搜索都终止于圆形节点，而所有的不成功搜索都终于方形节点。如果$2^{k-1} \leqslant n < 2^k$，那么所有的圆形节点都在第 1、2、…、$k$层上，而所有的方形节点都在第$k$和$k+1$层上（注意，树根在第 1 层上）。终止于第$i$层上的圆形节点的搜索所需的元素比较的次数为$i$，而终止于第$i$层上的方形节点的搜索所需的元素比较的次数为$i-1$。定理得证。 ■

定理 3.2 描述了 BinSearch 在最差情况下的运行时间。为了得到平均的情况，需要更仔细地分析二叉决策树，并将其大小与算法中的元素比较次数联系起来。一个节点到根的**距离**（distance）比它的层数小 1。**内路径长度**（internal path length）I是所有内节点到根的距离之和。同样，**外路径长度**（external path length）E是所有外节点到根的距离之和。由数学归纳法可以证明任意一个有n个内节点的二叉树，E和I满足下面的关系

$$E = I + 2n$$

事实上，E和I以及二分搜索的平均比较次数之间存在一个简单的关系。令$A_s(n)$表示成功搜索的平均比较次数，$A_u(n)$表示不成功搜索的平均比较次数。找到一个内节点对应的元素的比较次数比该内节点到根的距离大 1，因此：

$$A_s(n) = 1 + I/n$$

从树根到任意外节点的路径上的比较次数与树根与该外节点的距离相等。因为每个有n个内节点的二叉树都有$n+1$个外节点，得到：

$$A_u(n) = \frac{E}{n+1}$$

由上述三个关于$A_s(n)$、$A_u(n)$和E的式，得到：

$$A_s(n) = \left(1 + \frac{1}{n}\right) A_u(n) - 1$$

由上述式我们看出$A_s(n)$和$A_u(n)$是直接相关的。当算法对应的二叉树的外路径和内路径的长度最小时，得到$A_s(n)$的最小值（也是$A_u(n)$的最小值）。当二叉树的所有外节点都在相邻的层上的时候达到这个最小值，这也正是二分搜索产生的二叉树。从定理 3.2 可知E与$n\log n$成正比。代入上述式，我们得到$A_s(n)$和$A_u(n)$均与$\log n$成正比。由此得到二分搜索的平均以及最差比较次数只相差常数因子。最好情况分析比较简单。一次成功的搜索最少仅需要一次比较。一次不成功的搜索，定理 3.2 告诉我们，在最好的情况下也需要$\lfloor \log n \rfloor$次

比较。

现在可以总结二分搜索在最好、平均以及最差情况下的计算时间：

成功搜索			不成功搜索
$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$
最好	平均	最差	最好、平均以及最差

可以找到另一个算法在最差情况下比二分算法的时间复杂度好很多吗？我们在第 10 章给出这个问题的严格的回答。这里可以先做一个简短的回答，答案是否定的。证明这一结论需要使用一种技术，将二叉决策树看成是任何基于比较的搜索算法的模型。基于这样的描述，我们观察到二分搜索最小化了发现任意元素的最长路径的长度，因此从这个意义上来说任何其他的算法都无法做得更好。

在结束本节之前，来看一个二分搜索的有趣的变形，其每次 while 循环只进行一次比较。程序 3.5 给出了这个算法。这个算法的正确性证明留作习题。

```
int BinSearch1(Type a[], int n, Type x)
//已知数组 a[i:l]中的 n 个元素呈降序排列, 1<=i<=l, 判断 x 是否存在
//如果存在返回 j 使得 x == a[j], 否则返回 0
{
    int low = 1, high = n+1;
    //high 比最后一个元素大 1
    while (low < (high-1)){
        int mid = (low + high)/2;
        if (x < a[mid]) high = mid;//循环中只有一次比较
        else low = mid;            //x >= a[mid]
    }
    if (x == a[low]) return(low);  //x 存在
    else return(0);                //x 不存在
}
```

程序 3.5　每次循环只进行一次比较的二分搜索

BinSearch 有时比 BinSearch1 做出的元素比较次数多一倍（例如当 x>a[n]时）。然而，对于成功的搜索 BinSearch1 可能比 BinSearch 多$(\log n)/2$次元素比较（例如当 x==a[n]时）。BinSearch1 的分析留作习题。易得对于成功搜索和不成功搜索，BinSearch1 在最好、平均和最差情况下的时间复杂度都是$\theta(\log n)$。

我们在一台 Sparc 10/30 上运行这两个算法。表 3.3 的头两行表示成功搜索的平均时间。下面两行是所有不成功搜索的平均时间。对于成功搜索和不成功搜索，BinSearch1 比 BinSearch 要略好一些。

表 3.3　两个二分搜索算法的计算时间，以微秒为单位

平均大小	5000	10000	15000	20000	25000	30000
成功搜索						
BinSearch	51.30	67.95	67.72	73.85	76.77	73.40
BinSearch1	47.68	53.92	61.98	67.46	68.95	71.11

续表

平均大小	5000	10000	15000	20000	25000	30000
不成功搜索						
BinSearch	50.40	66.36	76.78	79.54	78.20	81.15
BinSearch1	41.93	52.65	63.33	66.86	69.22	72.26

习题

1. 运行递归和循环版本的二分搜索并比较它们的运行时间。测试一些n的取值，让每个算法都搜索集合众的每个数。然后再进行所有$n+1$中可能的不成功搜索。
2. 用数学归纳法证明$E = I + 2n$对与任意有n个内节点的二叉树都成立。E和I分别表示外路径长度和内路径长度。
3. 在一个无限数组中，头n个数是有序的整数，剩下的由∞填满。写一个 C++程序，以x为输入，在$\theta(\log n)$时间内找到数组中x的位置。n的值不是已知的。
4. 改变二分搜索，不再将集合等分为两个子集，而是子集的大小分别是 1/3 和 2/3。这个算法与二分算法相比如何？
5. 设计一个三分搜索算法，该算法在$n/3$位置处判断该元素是否与x相等，然后判断$2n/3$位置处的元素是否与x相等，这两次比较或者发现了x，或者将原问题转化成一个大小为原始问题大小 1/3 的子问题。比较该算法与二分搜索。
6. （a）证明 BinSearch1 是正确的。
   （b）验证下面的程序片断是正确的，并讨论其计算时间。

```
int low = 1, high = n;
do {
   int mid = (low + high)/2;
   if (x >= a[mid]) low = mid;
   else high = mid;
}while ((low + 1) != high)
```

3.4　找最大值和最小值

现在来看另一个可以用分治策略解决的简单问题。这个问题就是从n个元素的集合中找出最大值和最小值。程序 3.6 给出了一个最直观的算法。

```
void StraightMaxMin(Type a[], int n, Type& max, Type& min)
//将 max 设为 a[1:n]中的最大值, min 设为 a[1:n]中的最小值
{
    max = min = a[1];
    for (int i=2; i<=n; i++){
        if (a[i] > max) max = a[i];
        if (a[i] < min) min = a[i];
```

```
    }
}
```

程序 3.6 找最大值和最小值的直观算法

在分析这个算法的复杂度的时候，我们仍然关注比较的次数。这是因为该算法中其他操作的频率与元素比较的次数的量级是相同的。更重要的是，当 a[1:n]中的元素是多项式、向量、大数或者是字符串的话，元素比较的代价要远远高于其他操作。因此算法的运行时间主要由元素比较的总代价来决定。

在最好、平价和最差情况下，函数 StraightMaxMin 都需要$2(n-1)$次元素比较。一个马上的改进就是只有在 a[i]>max 为 false 的情况下，才需要比较 a[i]<min。因此可以把 for 循环中的内容换成下面的语句：

```
if (a[i] > max) max = a[i];
else if (a[i] < min) min = a[i];
```

这样当元素以升序排列的时候出现最好的情况。此时元素比较次数是$n-1$。最差的情况是元素以降序排列。此时元素比较次数是$2(n-1)$。

这个问题的分治算法可以是这样的：令$P=(n,a[i],\cdots,a[j])$表示这个问题的任意实例。这里n是列表$a[i],\cdots,a[j]$中元素的个数，我们想找出这个列表中的最大值和最小值。令 Small(P)当 n≤2 时为真。当$n=1$时最大值和最小值都是$a[i]$。当$n=2$时最大值和最小值可以通过一次比较得出。

如果列表中包含超过两个元素，我们将P划分为更小的实例。例如，可以把P分成 $P_1=(\lfloor n/2\rfloor,a[1],\cdots,a[\lfloor n/2\rfloor])$和$P_2=(n-\lfloor n/2\rfloor,a[\lfloor n/2\rfloor+1],\cdots,a[n])$。让P分成两个小实例之后，可以通过递归地调用同样的分治算法来解决它们。如何合并P_1和P_2的解从而得到P的解呢？如果$MAX(P)$和$MIN(P)$分别是P中的最大值和最小值，那么$MAX(P)$是$MAX(P_1)$和$MAX(P_2)$中较大的那个。同样，$MIN(P)$是$MIN(P_1)$和$MIN(P_2)$中较小的那个。

程序 3.7 给出了实现上述思想的算法。MaxMin 是一个递归函数找到集合$\{a(i),a(i+1),\cdots,a(j)\}$中的最大值和最小值。集合的大小为 1（$i=j$）和集合大小为 2（$i=j-1$）的情况分别处理。对于集合中有超过 2 个元素的情况，先确定中间点（与二分搜索一样），然后声称两个子问题。当这些子问题的最大值和最小值返回以后，通过比较两个最大值，以及比较两个最小值来最终确定整个集合上解。

这个过程的初始调用是：

```
MaxMin(1,n,x,y)
```

```
1   void MaxMin(int i, int j, Type& max, Type& min)
2   //a[1:n]是全局数组
3   //i 和 j 是整数, 且满足 1 <= i <= j <= n
4   //本函数递归地将 max 设为 a[i:j]中的最大值
5   //min 设为 a[i:j]中的最小值
6   {
7       if (i == j) max = min = a[i];  //Small(P)
8       else if (i == j-1){ //Small(P)的另一种情况
```

```
9                if (a[i] < a[j]){ max = a[j]; min = a[i];}
10               else { max = a[i]; min = a[j]}
11            }
12       else { //如果 P 不是小问题
13              //将 P 分割成子问题
14           //找到从哪里分割这个集合
15               int mid=(i+j)/2; Type max1, min1;
16           //求解子问题
17               MaxMin(i, mid, max, min);
18               MaxMin(mid+1, j, max1, min1);
19           //合并子问题的解
20               if (max < max1) max = max1;
21               if (min > min1) min = min1;
22       }
23   }
```

程序 3.7　递归找最大值和最小值

假设用下面九个元素来模拟函数 MaxMin 的执行：

a:	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
	22	13	−5	−8	15	60	17	31	47

一个追踪递归调用的好方法是构建一棵调用树，在每次有新的调用的时候就增加一个节点。对于这个程序来说，每个节点有 4 项信息：i、j、max 和 min。对于上述的 a[]，MaxMin 生成如图 3.5 所示的调用树。

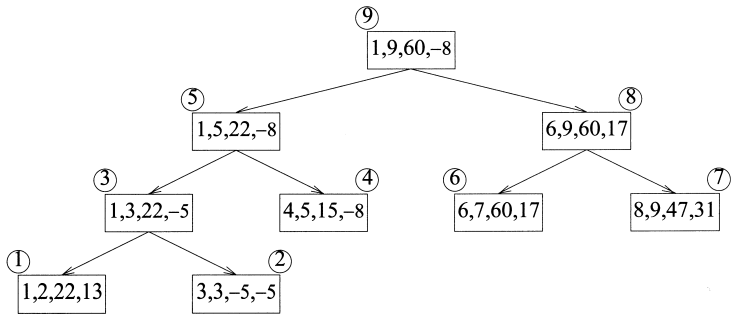


图 3.5　表示递归调用 MaxMin 的树

来看图 3.5，我们发现树根节点的 i 和 j 的值分别是 1 和 9，对应 MaxMin 的初始调用。函数的执行又调用了两次 MaxMin，其中 i 和 j 的值分别是 1 和 5，以及 6 和 9。这样基本分成了两个大小一样的子集。从这棵树上可以看出最深的递归是 4 层（包括第一次调用）。在每个节点的左上角上的圆圈中的数字对应 max 和 min 被赋值的顺序。

现在 MaxMin 到底需要多少次元素比较呢？如果$T(n)$表示这个次数，那么下面的递归关系式成立：

$$T(n)=\begin{cases}T(\lceil n/2\rceil)+T(\lceil n/2\rceil)+2 & n>2\\ 1 & n=2\\ 0 & n=1\end{cases}$$

当n是 2 的幂时，$n=2^k$，k是某个正整数，那么：

$$
\begin{aligned}
T(n) &= 2T(n/2) + 2 \\
&= 2(2T(n/4) + 2) + 2 \\
&= 4T(n/4) + 4 + 2 \\
&\vdots \\
&= 2^{k-1}T(2) + \sum_{1 \leqslant i \leqslant k-1} 2^i \\
&= 2^{k-1} + 2^k - 2 = \frac{3n}{2} - 2
\end{aligned}
\tag{3.4}
$$

注意，当n是 2 的幂时，$\frac{3n}{2} - 2$是最好、平均和最差情况下的比较数。

与直接算法所需的$2n - 2$次比较相比，上述算法节省了 25%的比较。可以证明，任何基于比较的算法都至少需要$\frac{3n}{2} - 2$次比较。因此 MaxMin 在这个意义上是最优的（详细证明见第 10 章）。但是否 MaxMin 在实际使用中也更好呢？不一定。从存储角度来考虑，MaxMin 就比直接算法差，因为它需要在栈中为 i、j、max、min、max1 和 min1 分配空间。给定n个元素，将有$\lfloor \log_2 n \rfloor + 1$层递归，我们需要为每一次调用保存这 7 个值（别忘了还需要返回地址）。

下面来看当元素间比较的代价与i和j之间比较的代价相同时，比较次数是多少。令$C(n)$表示这个次数。首先，我们观察到程序 3.7 的第 7 行和第 8 行可以由下面的语句所代替：

```
if (i >= j-1){ //Small(P)
```

因此一次 i 和 j–1 之间的比较就足以实现修改后的 if 语句。假设$n = 2^k$，k是某个正整数，得到

$$
C(n) = \begin{cases} 2C(n/2) + 3 & n > 2 \\ 2 & n = 2 \end{cases}
$$

解这个递推式，得到

$$
\begin{aligned}
C(n) &= 2C(n/2) + 3 \\
&= 4C(n/4) + 6 + 3 \\
&\vdots \\
&= 2^{k-1}C(2) + 3\sum^{k-2} 2^i \\
&= 2^k + 3 * 2^{k-1} - 3 \\
&= 5n/2 - 3
\end{aligned}
\tag{3.5}
$$

StraightMaxMin 的比较次数是$3(n-1)$（包括执行 for 循环的比较）。这要比$5n/2 - 3$大。尽管如此，因为递归而将 i、j、max 和 min 入栈的代价使得 MaxMin 比 StraightMaxMin 更慢。

程序 3.7 说明了几点问题。如果 a[]中元素的比较比整数变量之间的比较代价高得多，那么分治策略得到效率更高（事实上是最优）的算法。另一方面，如果该假设不成立，那么这个策略得到的算法的效率就比较低。因此，分治策略可以作为算法设计中的一种思路，并不能保证总是成功的。另外，我们看到有时候算出算法运行时间的相关常数也是必需的。MaxMin 和 StraightMaxMin 都是$\theta(n)$，所以用渐进表示不能区分这两个算法。最后，习题

中介绍了一种用$\frac{3n}{2}-2$次比较就可以找到最大值和最小值的方法。

注意：在任意分治算法的设计中，一般来说 Small(p)和 S(P)都是很直观的。因此，今后将只讨论如何分割给定问题 P，以及如何将子问题的解合并。

习题

1. 将 MaxMin 改写成复杂度相同的无递归的版本。
2. 测试你的循环版本的 MaxMin，并与 StraightMaxMin 比较。计数所有的比较。
3. 下面是一个使用循环的找最大值和最小值的算法。该算法不基于分治，有可能比 MaxMin 效率更高。它比较每两个相邻的元素对，然后将其中的大者与当前的最大值进行比较，并且将其中的小者与当前的最小值进行比较。写出这个算法，并且分析算法所需的比较次数。
4. 在程序 3.7 中，如果去掉第 8 行到第 11 行会怎样？去掉之后是否还能正确得出最大值和最小值？

3.5　合 并 排 序

下面来看分治策略的另一例子，它是一种排序算法并且具有一个特别好的性质，即最差复杂度是$O(n\log n)$。这个算法叫做**合并排序**（merge sort）。假设元素按照升序排列。已知n个元素（也称为键值）的序列$a[1],\cdots,a[n]$；主要的思想就是想象将其分成两个集合$a[1],\cdots,a[\lfloor n/2\rfloor]$和$a[\lfloor n/2\rfloor+1],\cdots,a[n]$。两个集合分别排序，然后得到的两个有序序列被合并为一个包含n个元素的有序序列。这样就得到另一个分治策略的最好的例子，其中分的操作是将序列分成两个等大小的集合，而合的操作是将两个有序集合合并成一个。

函数 MergeSort（程序 3.8）用递归实现了这一过程，函数 Merge（程序 3.9）合并两个有序集合。在执行函数 MergeSort 之前，n个元素应该被放到 a[1:n]中。然后 MergeSort(1,n)会将这些待排序的键值在 a 中升序排列。

```
void MergeSort(int low, int high)
//a[low : high]是一个需要排序的全局数组
//如果数组只含一个元素则 Small(P)为真
//在这种情况下数组已经是有序的
{
    if (low < high){ //如果包含 2 个以上元素，则将 P 划分为子问题
        //找到划分的位置
        int mid = (low + high)/2;
        //解子问题
        MergeSort(low, mid);
        MergeSort(mid+1, high);
        //合并子问题的解
        Merge(low, mid, high);
    }
}
```

程序 3.8　归并排序

```
void Merge(int low, int mid, int high)
//a[low : high]是一个全局数组包含 a[low:mid]和 a[mid+1:high]两个有序子集
//目标是将两个子集合并到放回 a[low:high]中， b[]是额外的全局数组
{
    int h = low, i = low, j = mid+1, k;
    while ((h <= mid) && (j <= high)) {
        if (a[h] <= a[j]) { b[i] = a[h]; h++}
        else { b[i] = a[j]; j++;} i++;
    }
    if (h > mid) for (k=j; k<=high; k++) {
                    b[i] = a[k]; i++;
                 }
    else for (k=h; k<=mid; k++) {
            b[i] = a[k]; i++;
        }
    for (k=low; k<=high; k++) a[k] = b[k];
}
```

程序 3.9　使用额外的存储开销将两个有序子数组合并

例 3.8　我们来考虑 10 个元素 a[1:10]=(310, 285, 179, 652, 351, 423, 861, 254, 450, 520)。函数 MergeSort 先将 a[]分成了两个大小为 5 的子数组（a[1:5]和 a[6:10]）。a[1:5]中的元素又被分成了两个数组 a[1:3]和 a[4:5]。接着 a[1:3]中的元素被分成两个数组 a[1:2]和 a[3:3]。a[1:2]中的两个元素最后被分成大小为 1 的两个数组，然后开始合并。注意，目前为止还没有移动任何元素。子数组是由递归机制来维持的。可用下图来表示数组的划分：

(310 | 285 | 179 | 652, 351 | 423, 861, 254, 450, 520)

其中竖杠表示子数组的边界。a[1]和 a[2]中的元素合并为：

(285, 310 | 179 | 652, 351 | 423, 861, 254, 450, 520)

然后 a[3]与 a[1:2]合并生成：

(179, 285, 310 | 652, 351 | 423, 861, 254, 450, 520)

下面，元素 a[4]和 a[5]合并：

(179, 285, 310 | 351, 652 | 423, 861, 254, 450, 520)

接着是 a[1:3]与 a[4:5]合并，得到：

(179, 285, 310, 351, 652 | 423, 861, 254, 450, 520)

此时算法已经退回到刚刚调用 Mergesort 的时候，准备处理第二个递归调用。重复执行递归调用得到下面的子数组：

(179, 285, 310, 351, 652 | 423 | 861 | 254 | 450, 520)

元素 a[6]和元素 a[7]合并。然后 a[8]与 a[6:7]合并得到：

(179, 285, 310, 351, 652 | 254, 423, 861 | 450, 520)

下面 a[9]和 a[10]合并，接着是 a[6:8]与 a[9:10]合并：

(179, 285, 310, 351, 652 | 254, 423, 450, 520, 861)

此时，我们有两个已排序的子数组，最终的合并得到下面的排序结果：

(179, 254, 285, 310, 351, 423, 450, 520, 652, 861)

图 3.6 给出了 MergeSort 在处理 10 个元素时的整个递归调用序列的树形表示。每个节点上的一对值是参数 low 和 high。注意划分是如何持续进行直到最后每个集合都只包含一个元素。图 3.7 是 MergeSort 对过程 Merge 递归调用的树形表示。例如，包含 1、2 和 3 的节点表示合并 a[1:2]与 a[3]。 ■

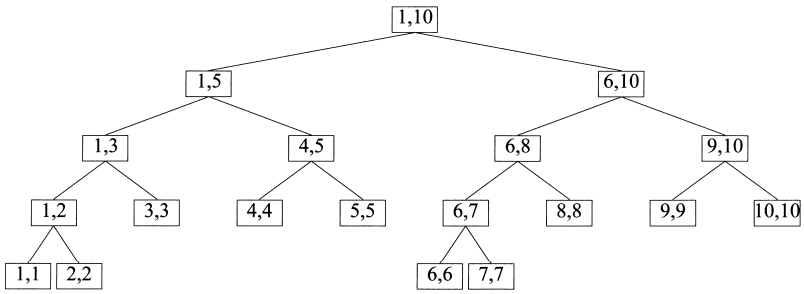


图 3.6　MergeSort(1,10)的调用树

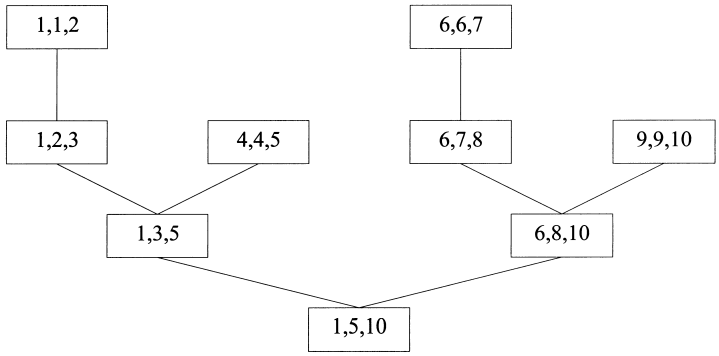


图 3.7　Merge 的调用树

如果合并所花的时间与n成正比，那么合并排序的计算时间可用如下的递归关系来表示：

$$T(n)=\begin{cases} a & n=1,\ a\text{ 是常数} \\ 2T(n/2)+cn & n>1,\ c\text{是常数} \end{cases}$$

当n是 2 的幂，$n=2^k$，可以用下面连续的替换来解这个递归式：

$$\begin{aligned} T(n) &= 2(T(n/4)+cn/2)+cn \\ &= 4T(n/4)+2cn \\ &= 4\,(2T(n/8)+cn/4)+2cn \\ &\vdots \\ &= 2^kT(1)+kcn \\ &= an+cn\log n \end{aligned}$$

容易看出，当$2^k<n\leqslant 2^{k+1}$，有$T(n)\leqslant T(2^{k+1})$。因此：

$$T(n)=O(n\log n)$$

尽管程序 3.8 很好地反映了合并排序的分治本质，它有些效率不高的地方可以进一步改进。我们下面讨论从三个方面进行改进，使得合并排序更好地执行。即使有这些改进，算法的复杂度仍然是$O(n\log n)$。在第 10 章我们将要分析任何基于比较的排序算法都不可能做得更好。

首先看到合并排序的一个问题是需要用到2n个位置。我们需要额外的n个位置，是因为我们不能原地将两个有序集合合并。即便这样，算法在每次调用 Merge 的时候，都需要把 b[low:high]中的结果复制回 a[low:high]。想不复制，还可以给每个键值增加一个信息域。（a[]中的元素成为是**键值**（key））这个域链接这些键值和它们在有序表中的相关信息（键值和它的相关信息称为**记录**（record））。这样合并有序表可以通过改变链接的值来完成，而不需要移动任何记录。一般只包含链接的域要比整个记录小，因此需要更少的空间。

除了原始数组 a[]外，我们再额外定义一个数组 link[1:n]，其中存放[0,n]之间的整数。这些整数是指向 a[]中元素的指针。一个表可以看成是由 0 结尾的指针序列。下面就是一组 link 的值，表示了两个表：Q和R。$Q=2$是其中一个表的起点，$R=5$是另一个表的起点。

```
link: [1] [2] [3] [4] [5] [6] [7] [8]
        6    4    7    1   3    0   8    0
```

这两个表是$Q=(2,4,1,6)$和$R=(5,3,7,8)$。这些表是 a[1:8]中的有序子集，我们得到 a[2]<=a[4]<=a[1]<=a[6]和 a[5]<=a[3]<=a[7]<=a[8]。

另一个 MergeSort 的问题是因递归而带来的栈空间的开销。因为合并排序将整个集合分成两个基本大小相等的子集，栈的深度与$\log n$成正比。使用栈是由于我们用自顶向下设计算法的缘故。如果我们用自底向上设计算法就可以避免栈的使用。详见习题。

从函数 MergeSort 以及之前的例子可以看出，对于大小为 2 的集合仍然会导致两次递归调用。对于规模小的集合来说，大部分的时间都花在处理递归而不是排序上面。这个情况可以通过在底层不使用递归的策略。用我们的分治控制抽象中的术语，当 Small 为真时，我们可以多做一些工作。在这种情况下，使用第二种适用于小规模集合的排序算法。

插入排序（insertion sort）在数组小于 16 个元素的时候是非常快的，尽管其在n很大时的复杂度是$O(n^2)$。其对 a[1:n]排序的基本思想是：

```
for (j=2; j <= n; j++) {
    将 a[j]在已排序的集合 a[1:j-1]中放到正确的位置
}
```

尽管 a[1:j−1]中的所有元素有可能都需要移动来为 a[j]腾出空间，算法在n很小的时候是很快的。程序 3.10 给出了算法的完整描述。

while 循环中的语句可以执行 0 到 j 次。因为 j 是从 2 增长到n的，这个过程的最差时间复杂度是：

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

其最优时间是$\Theta(n)$，此时 while 循环完全不用执行。这种情况出现在数据已经有序的情况下。

下面可以来描述用插入排序和 link 改进后的合并排序。函数 MergeSort1（程序 3.11）调用之前需将待排序的记录的键值放入 a[1:n]，然后将 link[1:n]置为 0。然后调用 MergeSort1(1,n)。返回的是指向表示 a[]中元素有序排列的下标的表的指针。当待排序的数据的个数小于 16 时调用插入排序。程序 3.10 中的所给出的插入排序需要稍做修改，从而

将 a[low:high]排序为一个链接表。修改后的版本是 InsertionSort1。程序 3.12 给出了修改后的合并函数 Merge1。

```
void InsertionSort(Type a[], int n)
//将数组 a[1:n]按照升序排列, n>=1
{
    for (int j=2; j <= n; j++){
        //a[1:j-1]已排好序
        Type item = a[j]; int i = j-1;
        while ((i >= 1) && (item < a[i])) {
            a[i+1] = a[i]; i--;
        }
        a[i+1] = item;
    }
}
```

程序 3.10　插入排序

```
void MergeSort1(int low, int high)
//使用额外数组 link[low:high]将全局数组 a[low:high]按升序排列
//link 中的值是从 low 到 high 的下标的列表, 得到 a[]的有序排列
//返回值是指向这个列表开始的指针
{
    if ((high-low+1)<16)
        return InsertionSort1(a, link, low, high);
    else{
        int mid = (low + high)/2;
        int q = MergeSort1(low, mid);
        int r = MergeSort1(mid+1, high);
        return(Merge1(q,r));
    }
}
```

程序 3.11　使用链表的归并排序

```
void Merge1(int q, int r)
//q 和 r 是指向全局数组 link[0:n]中列表的指针. link[0]的出现仅为了方便不用进行初始化
//由 q 和 r 指向的列表合并为一个列表，并且返回指向这个列表开始的指针
{
    int i=q, j=r, k=0;
    //新列表从 link[0]开始
    while (i && j) { //当两个列表都非空时
        if (a[i] <= a[j]) { //找到较小的那个值
            link[k] = i; k = i; i = link[i]; //将新值加入列表
        }
        else{
            link[k] = j; k = j; j = link[j];
        }
```